Universidade Federal de Pernambuco

Centro de Informática

Graduação em Ciência da Computação

Estudo comparativo para migração de banco de dados SQL para NoSQL em aplicações Android

Trabalho de Graduação

Otávio Vera Cruz Gomes

Orientador: Leopoldo Motta Teixeira

Agosto 2021

Otávio Vera Cruz Gomes

Estudo comparativo para migração de banco de dados SQL para NoSQL em aplicações Android

Trabalho apresentado ao Programa de Graduação em Ciência da Computação do Departamento de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Universidade Federal de Pernambuco - UFPE
Centro de Informática
Graduação em Ciência da Computação

Orientador: Leopoldo Motta Teixeira

Agradecimentos

Em primeiro lugar, gostaria de agradecer a Deus por ter guiado e iluminado o meu caminho durante todos esses anos. Encontrei nele força espiritual e mental para conseguir passar por todos os desafios que surgiram.

Agradeço a minha família, primeiramente a minha mãe, Celeide, por tudo que ela já fez e continua fazendo por mim. Agradeço também a minha esposa Camilla e a minha filha Alice, por toda motivação e força que encontrei nelas. Com elas, essa jornada se tornou mais tranquila e prazerosa.

Sou grato pelos amigos que fiz na faculdade, pelos projetos, grupos de estudos e por todos os momentos que passamos juntos durante o curso. Agradeço ao meu orientador, Leopoldo, por toda a ajuda e tempo que me foram dedicados durante a realização deste trabalho.

Resumo

Os bancos de dados não-relacionais (*NoSQL*), armazenam os dados em um formato diferente das tabelas relacionais. Eles fornecem esquemas flexíveis, escalam facilmente com grandes quantidades de dados e suportam altas cargas de usuário. A utilização desses bancos de dados, em aplicações *Android*, tem crescido com o passar dos anos. Desta forma, este trabalho tem como intuito, realizar uma análise comparativa entre bancos de dados - *SQL* e *NoSQL* - integrados em aplicações *Android* nativas. Mostrando as diferenças entre algumas soluções disponíveis no mercado e comparando-as através de testes de desempenho, depois de realizar a migração em um repositório selecionado. Com isso, auxiliando possíveis interessados em migrar de uma solução relacional para uma não-relacional, mais compatível com as suas necessidades. Finalmente, apresentamos diretrizes para a realização da migração entre os bancos.

Palavras-chave: *Android, SQL, NoSQL, Análise Comparativa, Aplicações Móveis.*

Abstract

Non-relational databases (NoSQL) store data in a different format than relational tables. They provide flexible schemes, scale easily with large amounts of data and support high user loads. The use of these databases in Android applications has grown over the years. Thus, this work aims to perform a comparative analysis between databases - SQL and NoSQL - integrated in native Android applications. Showing the differences between some solutions available on the market and comparing them through performance tests, after performing the migration in a selected repository. Therefore, helping potential interested in migrating from a relational to a non-relational solution, more compatible with their needs. In conclusion, we present guidelines for carrying out migration between databases.

Keywords: *Android, SQL, NoSQL, Comparative analysis, Mobile applications.*

Lista de Siglas

ACID - Atomicity, Consistency, Isolation, Durability
API - Application Program Interface
BASE - Basically Available, Soft state, Eventual consistency
BD - Banco de Dados
DAO - Data Access Object
FK - Foreign Key
IDE - Integrated Development Environment
JSON - JavaScript Object Notation
MVVM - Model View ViewModel
NoSQL - Not Only Structured Query Language
PK - Primary Key
SGBD - Sistema de Gerenciamento de Banco de Dados
SQL - Structured Query Language

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

Com o passar dos anos, os aplicativos *mobile* estão consumindo e gerando mais dados [1]. Para se ter noção dessa quantidade, no ano de **2018** eram gerados aproximadamente **2.5** quintilhões de *bytes* por dia [2], esse crescimento está associado - em grande parte - ao uso de redes sociais (tais como *Facebook, Instagram, WhatsApp, Telegram,* dentre outros), IoT (*Internet of Things*), serviços de *streaming*, etc [3]. Os desenvolvedores de *software* contam com uma solução utilizada há décadas pelo mercado para o armazenamento desses dados: os Bancos de Dados (BDs) Relacionais.

Esses BDs são bastante utilizados no desenvolvimento de aplicações *Web* e *Mobile* [4], sendo assim, não seria diferente no caso de aplicações *Android*. *SQLite* é o sistema de persistência de dados que a *Google* disponibiliza, de forma nativa, para o desenvolvimento de aplicativos no *Android Studio*[1] [5]. Os BDs relacionais são estruturados em *SQL* (*Structured Query Language*) e a sua estrutura (tabelas, relacionamentos e restrições) precisa ser definida previamente. Contudo, essa forma de estruturação força as aplicações a usarem esquemas fixos, não permitindo uma maior flexibilidade dos dados salvos, além disto, eles apresentam algumas ineficiências como: lidar com grandes volumes de dados, alta disponibilidade e escalabilidade [6].

A partir disso, os BDs *NoSQL* (*Not only SQL*) se destacam, eles dispõem de uma facilidade em se adequar e mudar a forma como os dados são armazenados [7]. Segundo *Walker* [8], a utilização de bancos de dados não-relacionais têm crescido rapidamente com o passar dos anos, tendo como principal propósito atuar em cenários onde os sistemas *SQL* apresentam dificuldades.

Diante desse cenário, as aplicações e sistemas que utilizam BDs *NoSQL* apresentam um grande potencial em melhorar o seu desempenho e evoluir rapidamente.

[1] O Android Studio é o ambiente de desenvolvimento integrado (IDE, na sigla em inglês) oficial para o desenvolvimento de apps Android.

Assim, este trabalho almeja fazer uma análise comparativa entre bancos de dados relacionais e não-relacionais, usados em *apps Android* de modo integrado. Percorrendo desde os pontos fortes e fracos de alguns tipos de BD, realizar a migração dos dados para BDs *NoSQL* e executar testes de desempenho, a fim de analisar os dados obtidos para a construção do trabalho. Fornecendo direções para possíveis interessados na migração, identificarem em quais casos e quando a mesma torna-se adequada.

1.1 Objetivos

O presente trabalho tem como finalidade, realizar um estudo comparativo entre bancos de dados relacionais e não-relacionais em aplicações *Android*. Realizando uma análise comparativa entre algumas das soluções utilizadas para cada tipo de banco de dados - relacional e não-relacional. Tendo isso em vista, os objetivos específicos deste trabalho são:

- Trazer um estudo sobre a utilização de bancos de dados integrados - relacionais e não-relacionais - em aplicações *Android* nativas;
- Coletar repositórios de código aberto para realização da migração e de testes de desempenho;
- Coletar, selecionar e estudar BDs *NoSQL*;
- Realizar análise e definir diretrizes para migração entre bancos de dados.

1.2 Estrutura do Trabalho

O trabalho encontra-se dividido em **5** capítulos e possui a seguinte estrutura:

- **Capítulo 2 - Fundamentação Teórica:** Contém definições sobre bancos de dados relacionais e não-relacionais. Neste capítulo, são apresentadas definições do modelo relacional e das estruturas dos BDs *SQL* e *NoSQL*. Além disto, é realizada uma análise dos pontos fortes e fracos deles.
- **Capítulo 3 - Metodologia:** Apresenta detalhes do processo de coleta e seleção dos repositórios e das soluções *NoSQL*. Também relata como foram os passos seguidos para a construção deste trabalho.
- **Capítulo 4 - Migração:** Nesta etapa, é descrita a migração do *SQLite* para os bancos de dados *NoSQL* e como foram aplicados os testes de desempenho e mudançass na estrutura do repositório selecionado.

- **Capítulo 5 - Análise Comparativa para Migração entre Bancos de Dados:** Neste capítulo, será realizada uma análise dos casos observados nos testes. Para dessa maneira, auxiliar no processo de migração entre BDs integrados em aplicações *Android*, bem como os benefícios trazidos e algumas ponderações na hora da escolha.
- **Capítulo 6 - Conclusão:** Contém as considerações finais deste trabalho.

Capítulo 2

Fundamentação Teórica

Este capítulo tem como principais objetivos: apresentar as definições do modelo relacional e as características de bancos de dados relacionais e não-relacionais, assim como, observar os pontos fortes e fracos desses tipos de BDs.

2.1 Modelo Relacional

No início dos bancos de dados, as aplicações armazenavam dados em sua própria estrutura única [9], obrigando os desenvolvedores a conhecer a estrutura de dados específica para encontrar os dados necessários. Para mitigar o problema de muitas estruturas de dados arbitrárias, surgiu o modelo relacional. Ele foi desenvolvido na década de 1970, tendo como base a álgebra relacional e o seu idealizador foi Edgar Frank Codd [10], as primeiras aplicações estiveram disponíveis no mercado na década de 80. Ele é o sucessor do modelo hierárquico, no qual os dados eram organizados em uma estrutura semelhante ao de uma árvore [11]. No modelo hierárquico, os relacionamentos possíveis entre os registros eram um para um e um para muitos, não permitindo um relacionamento de muitos para muitos. Assim, o modelo relacional conseguiu resolver esse problema do seu antecessor e cobrir todos os relacionamentos entre as entidades. Entretanto, este modelo apresenta algumas limitações como a falta de flexibilidade na sua estrutura por permitir apenas dados estruturados e a dificuldade de lidar com grande de volume de dados.

2.1.1 Características

Os sistemas *SQL* são baseados no modelo relacional e de acordo com *C.J.Date*[12], os sistemas baseados neste modelo possuem 3 aspectos: estrutural, integridade e manipulador. O aspecto estrutural é representado por tabelas, o de integridade corresponde a determinadas restrições que as tabelas devem seguir e o manipulador

refere-se a operações que o usuário pode realizar nas tabelas.

As tabelas são representações das entidades e relacionamentos, nas quais são compostas por linhas (tuplas) e colunas (atributos) [13]. Para obter-se um relacionamento entre tabelas são utilizadas chaves, as mesmas são conjuntos de um ou mais atributos que as tornam únicas. Os tipos de chave mais utilizadas são: Chave Primária (*Primary Key*), Chave Estrangeira (*Foreign Key*), Chave Candidata e Chave Alternativa. A PK é a principal chave da tabela - diferentemente das outras - onde pode existir apenas uma chave desse tipo na tabela [14]. Uma representação das entidades e seus relacionamentos pode ser vista na Figura 2.1[1].

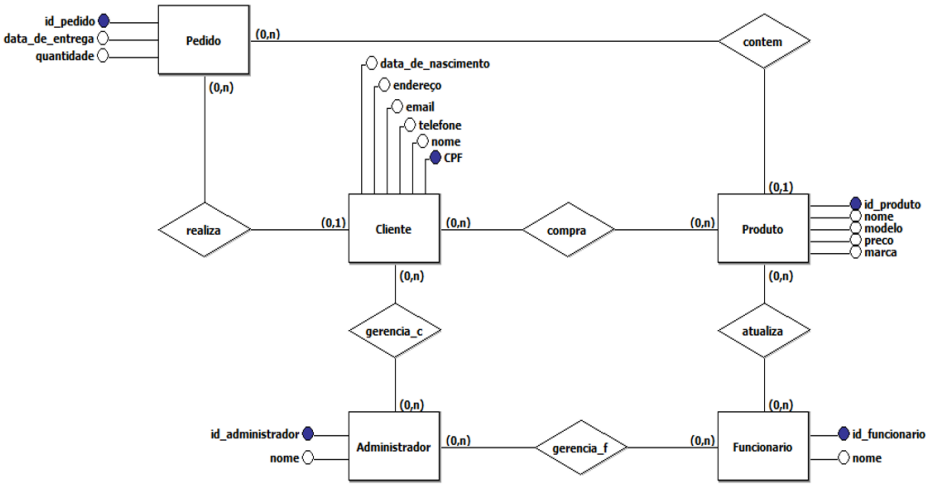

Figura 2.1: Diagrama de Entidade e Relacionamento - ER

As entidades do exemplo são representadas pelos retângulos, sendo elas: Cliente, Produto, Pedido, Administrador e Funcionário. Os atributos encontram-se ao lado de cada entidade e as PKs estão marcadas com uma bolinha azul. Os losangos retratam os relacionamentos entre as entidades com as suas respectivas cardinalidades, elas podem ser um-para-um (1:1), um-para-muitos (1:N) e muitos-para-muitos (N:N) [15].

[1]Disponível em: https://sites.google.com/site/webquestlubvap/07

2.1.2 SGBD

Para o gerenciamento do Banco de Dados *SQL*, é utilizado um Sistema de Gerenciamento de Banco de Dados, o qual permite que as seguintes operações sejam realizadas pelo usuário: inserção de novos dados, remoção, atualização e leitura de dados existentes. Ademais, ele possibilita a aplicação do processo de Normalização, esse conceito consiste em um conjunto de regras que visa evitar redundância de dados, obtendo-se assim melhorias na integridade e no desempenho [16]. Alguns SGBDs conhecidos e bastantes difundidos pelo mercado são: *MySQL, SQLite, OracleDB, SQL Server, PostgreSQL*, dentre outros [17]. Além da Normalização, outro conceito prezado pelos SGBDs é o paradigma *ACID (Atomicity, Consistency, Isolation and Durability)*, formado pelas 4 propriedades que compõem a sigla: Atomicidade, Consistência, Isolamento e Durabilidade [18].

A propriedade da Atomicidade garante transações booleanas, isto é, a transação ocorrerá ou não. Na propriedade da Consistência, assegura-se que a transação respeitará as regras de integridade. Já a de Isolamento, permite que várias transações possam ocorrer de maneira isolada no mesmo banco de dados. Por fim, a propriedade da Durabilidade consiste em persistir os dados de uma transação mesmo com a ocorrência de falhas [19].

2.1.3 BDs SQL em Aplicações Android

Os bancos de dados baseados em *SQL* são usados em aplicações *Android* tanto no *back-end* de uma *API* - por exemplo - quanto localmente para o fluxo de vida dos aplicativos. O *SQLite* é empregado em aplicações *mobile* [20], além do mais, esse BD encontra-se disponível - nativamente - no *Android Studio* e o seu principal propósito é auxiliar os desenvolvedores a persistir os dados da aplicação nos dispositivos.

A *IDE* permite o acesso e gerenciamento desse BD por meio de duas formas, a primeira consiste em criar uma classe que herde de *SQLiteOpenHelper* e implemente os métodos *onCreate* e *onUpgrade* [21]. O primeiro método é chamado quando o BD é criado pela primeira vez e onde é recomendado definir as tabelas, relacionamentos e restrições. No segundo método, ocorre a chamada quando realiza-se uma mudança na versão do esquema, adição de tabelas, mudanças nas tabelas, etc.

A segunda forma é através da biblioteca de persitência de dados *Room* do *Android Jetpack*. O site *Android Developers* [2] descreve essa biblioteca da seguinte maneira: "O banco de dados *Room* oferece uma camada de abstração sobre o *SQLite* para permitir acesso fluente ao banco de dados e, ao mesmo tempo, aproveitar toda a capacidade do *SQLite*"[22]. Para inserir dados ou realizar consultas, essa

[2]Disponível em: https://developer.android.com/training/data-storage/room?hl=pt-br

biblioteca provisiona um componente chamado DAO, que são objetos criados por ela para implementação de operações do tipo *CRUD* [23]. Desse modo, evita-se o uso de código *boilerplate* [3] e obtém-se uma redução no tempo de desenvolvimento da aplicação.

2.2 BDs NoSQL

Os bancos de dados *NoSQL* surgiram para atenuar a ineficiência dos bancos de dados relacionais, que é a capacidade de gerenciar grandes volumes de dados - semiestruturados ou não estruturados - necessitados de alta disponibilidade e escalabilidade [24]. O termo *NoSQL* foi criado por Carlo Strozzi em 1998, com a intenção de nomear o seu projeto *open source*, que buscava ser uma implementação mais simples de um banco de dados relacional [25].

2.2.1 Características

Os bancos de dados *NoSQL*, diferentemente dos baseados no modelo relacional, conseguem ser estruturados de uma maneira versátil, permitindo salvar os dados, por exemplo, como arquivos *JSON* [26]. A demanda por bancos de dados *NoSQL* está crescendo, devido as aplicações *cloud* exigirem: alta disponibilidade, velocidade e opções para proteger-se de eventuais falhas [27]. Neste cenário, surge um conceito bastante difundido por esse tipo de BD que é o *BASE* (Basically Available, Softstate, Eventual consistency).

O conceito *BASE* é logicamente oposto ao paradigma *ACID*, no qual, a primeira propriedade consiste na resposta de uma operação, mesmo apresentando falhas, no retorno dos dados. Na segunda propriedade, o sistema pode ser alterado a qualquer momento por uma transação. Já a terceira propriedade, refere-se a consistência eventual do estado dos dados nos nós dos servidores, no caso de BDs em cloud [28].

2.2.2 Tipos de BDs NoSQL

Atualmente, existem bancos não-relacionais que podem se tornar concorrentes do *SQLite*, algumas soluções serão abordadas no Capítulo **3**. Os principais tipos de bancos de dados *NoSQL* apresentam as seguintes características: chave-valor, orientado a documentos, orientado a colunas e orientado a grafos [29]. Os BDs orientados a coluna, diferentemente dos bancos relacionais que são otimizados para armazenar linhas de dados, são otimizados para recuperar colunas de dados [30]. E

[3] Código *boilerplate* se refere a seções de código que devem ser incluídas em muitos lugares com pouca ou nenhuma alteração.

o BD orientado a Grafos apresenta a sua estrutura baseada no modelo matemático da Teoria dos Grafos, representado por vértices e arestas [31], onde os vértices são uma representação de entidades e as arestas dos relacionamentos. Contudo, esses dois últimos não serão observados no estudo.

2.2.3 Chave-Valor

Um BD chave-valor é um tipo de BD *NoSQL* que armazena os dados - de forma simples, como um conjunto de pares de chave-valor, no qual uma chave consiste em um identificador único [32]. A chave e valor podem ser de qualquer tipo, onde pode-se salvar desde objetos simples, como uma *String*, a objetos complexos. Esse tipo de BD possui como principal vantagem, a capacidade de lidar com grandes volumes de dados, a desvantagem é que os dados só podem ser recuperados pela chave.

2.2.4 Orientado a Documentos

Este tipo de BD foi projetado para consultar e armazenar dados como documentos do tipo *JSON*. Com isso, ele apresenta uma estrutura flexível e semiestruturada permitindo uma evolução à medida que os aplicativos necessitem. Este tipo de BD apresenta outras características como: indexação flexível, análises de dados em grupos de documentos e consultas ad hoc eficientes [33]. Uma representação de um documento baseado no modelo de dados incorporados pode ser visto na Figura 2.2[4].

Figura 2.2: Modelo de BD Orientado a Documentos Incorporado

[4]Disponível em: https://medium.com/@gpanassol/como-posso-fazer-modelagem-de-dados-em-mongodb-ea61268ee10b

Por permitir uma estrutura flexível, o exemplo acima pode ser representado através de um modelo normalizado de dados, assim como no modelo relacional, uma representação desse modelo pode ser observado na Figura 2.3 [34].

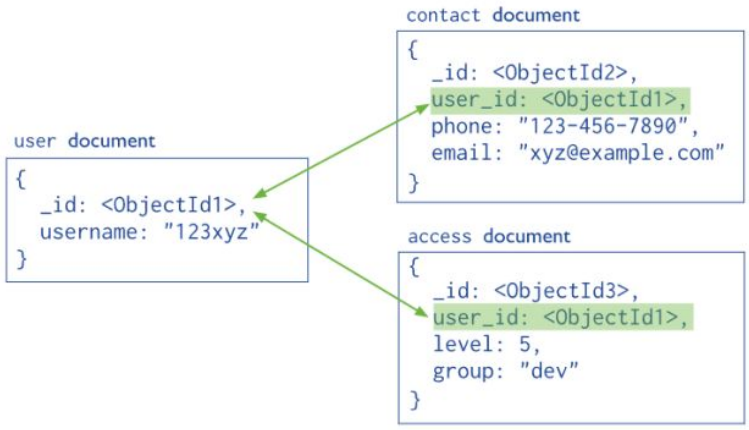

Figura 2.3: Modelo de BD Orientado a Documentos Normalizado

2.3 Diferenças entre BDs SQL e NoSQL

Os bancos de dados relacionais apresentam algumas limitações como a falta de flexibilidade na sua estrutura, de escalabilidade e de tratamento de dados que não sejam estruturados. Em contrapartida, os BDs *NoSQL* conseguem alterar a sua estrutura por não seguirem o conceito *ACID*, e sim o *BASE*. Os BDs não-relacionais não substituem completamente os relacionais, uma vez que, eles visam atuar em cenários que os bancos relacionais não cobrem.

2.3.1 SQL x NoSQL

Nesta etapa, será feita uma comparação entre alguns parâmetros, observando-se os pontos fortes e fracos desses tipos de bancos de dados. Os detalhes podem ser vistos na Tabela 2.1.

Parâmetro	***SQL***	***NoSQL***
Consistência	É considerado o principal ponto forte, permitindo integridade dos dados armazenados.	Apresenta uma eventual consistência, dependendo do SGBD.
Flexibilidade	Possui uma estrutura rígida, por causa da integridade das tabelas. Aceitando apenas dados estruturados.	É bastante flexível, devido a facilidade de trabalhar com dados estruturados, semiestrutu-rados e não estruturados.
Grande volume de dados	Não lida bem com grande volume de dados.	Trabalha bem com o armazenamento de grandes volumes de dados.

Tabela 2.1: Diferenças entre os paradigmas NoSQL e Relacional

2.4 Trabalhos Relacionados

Nesta Seção, é apresentado o estado da arte sobre o panorama de análises comparativas entre BDs *SQL* e *NoSQL*, levando-se em consideração testes de desempenho.

A avaliação de desempenho de BDs apresenta vários tópicos a serem debatidos. As limitações ao lidar com uma quantidade enorme de dados ou a dificuldade de ter esquemas adaptáveis, faz com que os SGBDS relacionais não sejam tão eficientes.

Para observar essa problemática no cenário de Banco de Dados relacionais integrados em aplicações *Android* nativas, existe o trabalho de Perrier [35], que avalia o desempenho de três BDs usados no desenvolvimento *mobile*. A motivação principal desse trabalho, é realizar uma análise comparativa a partir de testes de desempenho nos BDs *SQLite, CouchLite* e *DbO4*.

Outro trabalho que aborda a temática de BDs incorporados em dispositivos *Android* é o de Dissanayaka [36], onde ele analisa a partir de *queries* pré-definidas qual dos BDs, *SQLite* ou *CouchBase*, funciona melhor no ambiente *mobile*. A construção desses dois trabalhos é bem semelhante, ambos analisam casos em que o BD *SQLite* mostra um desempenho acima dos seus concorrentes considerando-se processamento de um conjunto dados que não seja grande.

Uma outra abordagem é trazida pelo artigo de Gökgöz [37], na qual visa realizar uma otimização de banco de dados a partir de um modelo de dados em uma aplicação *Android*. Comparando-se a execução da aplicação e operações *CRUD* através de *queries* simples, em dois tipos de BDs distintos.

Segundo Hammes [38], os BDs *NoSQL* apresentam como principais pontos positivos: o paradigma *BASE*, escalabilidade, disponibilidade e simplicidade é fundamental. Em seu trabalho ele propôs dois cenários distintos: o primeiro avalia uma comparação do desempenho com dados estruturados e o segundo com dados não-estruturados.

Este trabalho difere dos outros, por realizar uma análise do comportamento dos BDs através dos seguintes pontos: migração de dados estruturados de um BD relacional para BDs não-relacionais, testes de desempenho e por analisar uma mudança na estrutura dos dados.

Capítulo 3

Metodologia

A primeira parte deste trabalho constituiu-se em uma pesquisa do estado da arte dos paradigmas relacional e *NoSQL*. Para isto, realizou-se uma busca de informações através da leitura de artigos, livros e publicações relevantes. A partir disso, foram analisadas as principais características e definidos os critérios para o estudo.

A segunda parte tem como fundamentação a coleta e seleção de repositórios localizados no *Github*[1], detalhando-se as particularidades deles e como foi escolhido o repositório para a migração entre BDs e, posteriormente, a análise comparativa descrita no Capítulo 5. Assim como, é possível observar, o modo no qual foi realizado o levantamento das soluções *NoSQL* disponíveis para aplicações *Android* e uma análise das qualidades e deficiências de cada solução, que auxiliaram na escolha.

A terceira parte consistiu na migração do *SQLite/Room* para as ferramentas não-relacionais selecionadas, descrevendo-se as alterações praticadas neste processo. Em vista disso, conduziram-se testes de desempenho com a finalidade de comparar os tempos de execução das operações propostas. Por último, é feita uma análise comparativa dos resultados obtidos e a descrição de um guia para realizar a migração.

3.1 Processo de Coleta dos Repositórios

O motivo para aplicar este método, de pesquisar exemplos existentes, é a busca de simular um cenário condizente com a literatura. Para isso, inicialmente, foram coletados repositórios através de pesquisas no *GitHub* - de forma manual - aplicando-se uma combinação das seguintes *tags*: *Android, CRUD, SQLite* e *Room*. Outro

[1]Disponível em: https://github.com/. Acessado em: 20/04/2021

critério aplicado foi o filtro para as linguagens *Java* e *Kotlin*, ignorando-se os repositórios que destoassem desses parâmetros. Na primeira consulta[2], foi aplicada uma combinação das *tags Android, SQLite* e *CRUD*, resultando em **711** repositórios divididos da seguinte forma: **597** em *Java*, **38** em *Kotlin* e o restante em outras linguagens. Na segunda pesquisa[3], foram utilizadas as *tags Android, Room* e *CRUD* conseguindo-se um total de **117** repositórios, com a seguinte divisão: **68** em *Java*, **42** em *Kotlin* e o que restou para as outras.

3.2 Seleção dos Repositórios

Os critérios definidos na seleção dos repositórios encontram-se listados a seguir :

- *Java* ou *Kotlin* como as linguagens principais;
- Relacionamentos *one-to-one, one-to-many* e/ou *many-to-many* entre as entidades/tabelas;
- Padrão arquitetural *MVP* ou *MVVM*.

As linguagens *Java* e *Kotlin* são as mais utilizadas no desenvolvimento nativo de *apps Android* [39], uma vez que, *Java* era a linguagem oficial até o ano de **2019** [40], passando o posto para *Kotlin*. Por conta disso, foi realizada uma pesquisa[4] no *GitHub*, para ter-se uma dimensão desse critério. O resultado da busca mostrou uma dominância dessas duas linguagens, onde mais da metade dos repositórios utilizam-as, como observado na Figura 3.1.

[2]Disponível em: https://github.com/search?q=android+sqlite+crud&type=Repositories. Acessado em: 21/05/2021

[3]Disponível em: https://github.com/search?q=android+room+crud&type=Repositories. Acessado em: 21/05/2021

[4]Disponível em: https://github.com/search?q=android. Acessado em: 21/05/2021

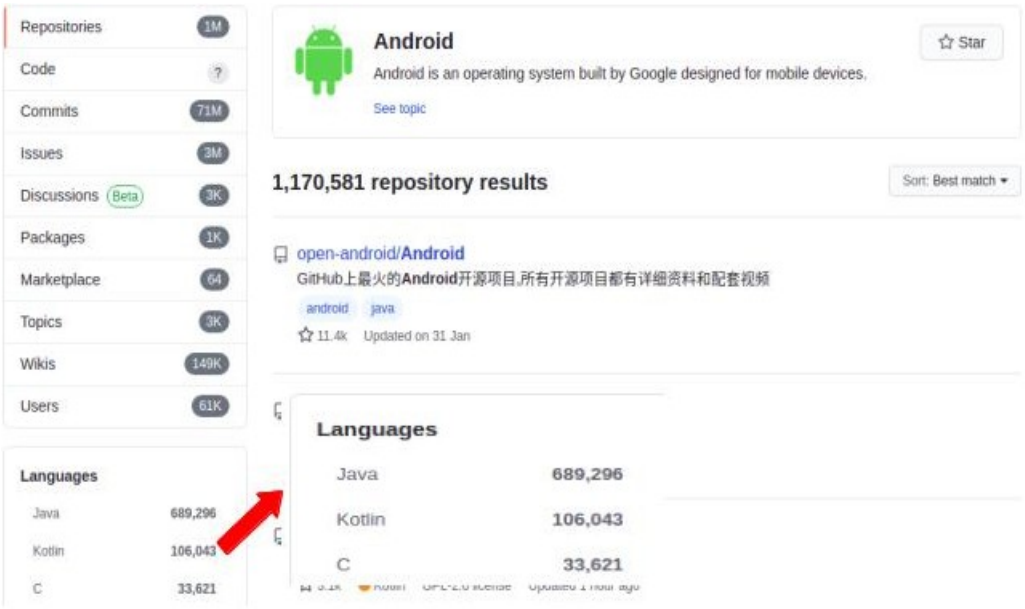

Figura 3.1: Pesquisa de repositórios Android no GitHub

Os relacionamentos definidos no segundo item representam a forma como as entidades/tabelas interagem entre si. Tendo isso em vista, os relacionamentos entre as entidades são evidenciados por meio das *Android Annotations*[5], nos repositórios integrados com a biblioteca *Room*. Já no caso do BD *SQLite*, verificaram-se as tabelas definidas e se elas relacionavam-se por meio de suas chaves PK e FK.

No terceiro item, definiu-se os padrões arquiteturais a serem procurados nos repositórios. O desenvolvimento de *software Android* com esses padrões têm crescido com o passar dos anos [41], por eles trazerem benefícios como: facilidade de manutenção do código, testabilidade, pouca dependência da *API Android* e por serem modulares. Atualmente, a arquitetura de *app* recomendada pela *Google* é o *MVVM* [42]. Com base nisso, o esforço para realizar a migração pôde ser minimizado, devido a modularidade dessas arquiteturas, permitindo alterar os trechos de códigos pertinentes.

Após o processo descrito anteriormente, foi escolhido um repositório - de um total de dez - para a realização da migração para os BDs *NoSQL* e submissão de testes de *benchmark*[6] abordados no Capítulo 4. A Tabela 3.1 mostra a listagem dos repositórios selecionados e detalhes como *commits*, estrelas e *forks*.

[5]O Android Annotations é uma estrutura orientada a anotações que permite simplificar o código em seus aplicativos e reduz o clichê de padrões comuns.

[6]Benchmark é o processo de realizar testes com o propósito de avaliar a performance.

Nome	Url do Repositório	Commits	Forks	Estrelas
Android-Room-CRUD-Sample	https://github.com/husaynhakeem/Android-Room-CRUD-Sample	8	23	13
mvp-android-arch-component	https://github.com/jeremy-giles/mvp-android-arch-component	65	100	0
Room Relation	https://github.com/Ikhiloya/RoomRelation	3	4	17
Udemy Course Walkthrough	https://github.com/ProductivityCafe/UdemyCourseWalkthrough	46	4	9
android-aws-architecture-components-dialogflow-chatbot	https://github.com/VeeShostak/android-aws-architecture-components-dialogflow-chatbot/tree/master/app/src/main/java/io/github/veeshostak/aichat	49	7	12
NewsMVP	https://github.com/graviton57/NewsMVP	8	8	10
android-mvvm-architecture	https://github.com/MindorksOpenSource/android-mvvm-architecture	121	863	2.600
Android Room Lib Example	https://github.com/tianbin-dev/AndroidRoomLibExample	15	1	8
PetShop	https://github.com/Elizaveta99/PetShop	1	1	0
Market Place Android	https://github.com/DorianWorkbench/MarketPlace_Android	11	0	0

Tabela 3.1: Informações dos Repositórios Selecionados

3.2.1 Repositório Selecionado

O repositório selecionado para a migração foi o *Udemy Course Walkthrough*, esse repositório utiliza o padrão arquitetural *MVVM*, *Kotlin* como linguagem principal e a biblioteca de persitência *Room* para comunicar-se com o banco de dados *SQLite*. Ele consiste em um *To-do list app*[7], no qual é dividido em duas partes: na primeira é possível criar tarefas e adicionar um *checklist* para cada tarefa, já a segunda é permitido criar notas. A migração terá como foco essa primeira parte do *app*, na qual existe o relacionamento *one-to-many* entre as entidades *Task* e *Todo*. Os Listings 3.1, 3.2 e 3.3 mostram como essas entidades estão estruturadas.

Listing 3.1: Classe de retorno na consulta pelo Room

```
class Task @JvmOverloads constructor(
    title: String,
    @Relation(
        parentColumn = "uid",
        entityColumn = "taskId",
        entity = Todo::class
    )
    val todos: MutableList<Todo> = mutableListOf()
) : TaskEntity(
    title = title
) {
    fun isComplete(): Boolean = todos.none {!it.isComplete}

    init {
        todos.forEach { it.taskId = uid }
    }
}
```

Listing 3.2: Entidade Task mapeada pelo Room

```
@Entity(tableName = "tasks")
open class TaskEntity(
    @PrimaryKey
    var uid: String = UUID.randomUUID().
       leastSignificantBits.toString(),
    @ColumnInfo var title: String
)
```

[7]É um aplicativo que consiste em uma lista de tarefas a serem realizadas.

Listing 3.3: Entidade Todo mapeada pelo Room

```
@Entity(tableName = "todos")
data class Todo(
    @PrimaryKey(autoGenerate = true)
    var uid: Int = 0,
    @ForeignKey(
        parentColumns = ["uid"],
        childColumns = ["taskId"],
        entity = TaskEntity::class,
        onDelete = CASCADE
    )
    var taskId: String? = null,
    @ColumnInfo
    var description: String,
    @ColumnInfo
    var isComplete: Boolean = false
)
```

A escolha desse repositório ocorreu fundamentada em uma análise da estrutura do código - como a arquitetura estava definida - e do esforço que seria empregado para adaptá-lo para a migração. Por possuir um código bem modularizado e limpo, métodos de inserção, leitura e remoção de dados, bem como ter se encaixado nos critérios descritos anteriormente, fez com que quantidade de *forks* e estrelas não se tornassem tão relevantes em comparação aos outros repositórios.

3.3 Coleta e Seleção de BDs *NoSQL*

Para obter-se uma solução *NoSQL* condizente a sua realidade, em detrimento de uma *SQL*, é fundamental fazer uma listagem das soluções existentes no mercado e realizar um estudo sobre elas. Desta forma, consegue-se ter dimensão dos benefícios trazidos por cada solução, bem como as suas limitações.

A seleção apropriada de uma ferramenta *NoSQL* acarreta em um melhor desempenho para a aplicação e uma maior flexibilidade no momento de estruturação dos dados. Assim, esta Seção apresenta como foi realizado o processo de coleta e seleção das soluções abordadas na literatura, através das pesquisas e estudos realizados e de todo o conhecimento obtido a partir disso.

3.3.1 Processo de Coleta das Soluções

Para fundamentar a definição das ferramentas mais adequadas ao estudo, torna-se necessário explicar como foi realizado o processo de coleta das mesmas.

Nome	Url	Tipo
LevelDB	https://github.com/google/leveldb	Chave-valor
SnappyDB	https://www.snappydb.com/	Chave-valor
Paper	https://github.com/pilgr/Paper	Chave-valor
Hawk	https://github.com/orhanobut/hawk	Chave-valor
UnQLite	https://unqlite.org/	Chave-valor/Orientado a documentos
MongoDB Realm	https://docs.mongodb.com/realm/get-started/introduction-mobile/	Orientado a documentos
ObjectBox DB	https://github.com/objectbox/objectbox-java	Orientado a documentos
CouchBase Lite	https://docs.couchbase.com/couchbase-lite/current/android/quickstart.html	Orientado a documentos

Tabela 3.2: Soluções Coletadas

Primeiro, realizou-se uma leitura do seguinte artigo "Choosing the right NoSQL database for the job: a quality attribute evaluation"[43], no qual traz uma listagem de algumas soluções *NoSQL* utilizadas no mercado e realiza uma avaliação das mesmas utilizando atributos de qualidade. Deste modo, o artigo visa trazer informações relevantes para auxiliar engenheiros de *software* e arquitetos na escolha da solução que se adapte melhor ao seu cenário.

No segundo momento, realizaram-se pesquisas de BDs *NoSQL* utilizados localmente em *apps Android* no *Google Scholar*, *Google* e também no *GitHub*. Os termos buscados nessas pesquisas foram "Bancos de dados NoSQL para aplicações Android", "NoSQL embedded database for Android", etc. A partir disso, obteve-se um total de **8** BDs e os tipos encontrados foram chave-valor e orientado a documentos, na Tabela 3.2 estão listadas as soluções coletadas.

Nome	Tipo
LevelDB	Chave-valor
Paper	Chave-valor
Hawk	Chave-valor

Tabela 3.3: BDs Chave-valor

Nome	Tipo
MongoDB Realm	Orientado a documentos
ObjectBox DB	Orientado a documentos
CouchBase Lite	Orientado a documentos

Tabela 3.4: BDs Orientado a Documentos

3.3.2 Processo de Seleção das Soluções NoSQL

Após a fase de coleta das soluções, faz-se necessário detalhar o procedimento adotado para selecionar as ferramentas *NoSQL*, evidenciando o passo-a-passo aplicado.

A princípio, realizou-se um refinamento das soluções por tipo de BD, chave-valor e orientado a documentos, e em seguida uma análise das documentações e das vantagens e desvantagens de cada solução. Descartaram-se as ferramentas com as versões e documentações desatualizadas, garantindo dessa maneira futuras manutenções, melhorias e correções de *bugs*. Esse processo originou duas tabelas, a Tabela 3.3 e a Tabela 3.4 com os BDs chave-valor e orientado a documentos, repectivamente.

3.3.2.1 Escolha do BD Chave-valor

Após o procedimento anterior, comparou-se os bancos de dados entre si, analisando-se a documentação do *Paper* existe um teste[8] de *Benchmark* entre ele e o *Hawk*, no qual, o *Paper* obteve um desempenho superior em relação ao seu concorrente, desta maneira o *Hawk* foi removido das opções. Em seguida, executou-se um estudo para definir qual das duas soluções remanescentes seria a escolhida, o *LevelDB* salva suas chaves e valores como *byte arrays* arbitrários [44], tornando obrigatória a transformação das chaves e dos valores para esse tipo, tanto na consulta quanto na escrita de dados. Um ponto positivo em seu favor é que ele é mantido pela *Google* e está recebendo melhorias e manutenções quando necessário.

Em contrapartida, o *Paper* busca de uma forma simples, porém rápida, armazenar classes *Java/Kotlin* como elas são: sem anotações, *factory methods*, extensões

[8]Disponível em: https://github.com/pilgr/Paper/blob/master/paperdb/src/androidTest/java/io/paperdb/benchmark/Benchmark.java

obrigatórias para as classes, etc [45]. As suas chaves são sempre salvas como *Strings*. Outro ponto evidenciado é a flexibilidade desse banco de dados, permitindo o tratamento de mudanças de estruturas das classes, por exemplo, quando é removido um atributo o mesmo vai ser ignorado na próxima leitura e na adição de um novo atributo o BD atribui o valor padrão para ele. Uma limitação que esses dois BDs compartilham é a ausência de suporte cliente-servidor em suas bibliotecas, consequentemente, não oferece uma *API* que se comunica com um banco de dados remoto. Desta maneira, a solução chave-valor escolhida foi o *Paper*, por ser a opção que se sobressaiu dentre as outras e ser adequada ao estudo proposto.

3.3.2.2 Escolha do BD Orientado a Documentos

Das três opções obtidas, depois do refinamento, o *ObjectBox DB* é a mais simples e o seu desempenho comparado as outras é inferior, assim sendo, a análise restringiu-se aos dois bancos de dados mais populares, *MongoDB Realm* e *CouchBase Lite*. Levando isso em consideração, buscou-se comparativos entre as duas ferramentas, o *CouchBase* traz uma comparação [9] entre elas, em que ele atingiu uma performance melhor do que o seu concorrente, além de apresentar uma interface mais amigável para realizar *queries*. Um outro fator importante na escolha, é a possibilidade de integrar o banco de dados sem precisar adicionar anotações para mapear as entidades e chaves. Logo, o BD orientado a documentos escolhido foi o *Couchbase Lite*, ele é um banco de dados *NoSQL* que pode ser utilizado integrado em aplicações *mobile* ou com o *Sync Gateway* e o servidor *CouchBase*, dessa forma, fornecendo uma solução *cloud* completa [46]. A escolha dessa solução não deve ser considerada como unânime, pois para o caso em debate ela foi a que teve uma melhor combinação, porém dependendo do cenário as outras soluções podem ser ideais.

3.3.3 Comparação entre as Soluções Escolhidas

Possuindo como base o método aplicado pelo artigo mencionado anteriormente, definiram-se atributos com a finalidade de comparar as soluções escolhidas com o SQLite. Deste jeito, consegue-se observar as vantagens e as limitações de cada BD, auxiliando na tomada de decisão para a migração. A Tabela 3.5 traz esse comparativo com dados relevantes para o projeto.

[9] Disponível em: https://www.couchbase.com/comparing-couchbase-vs-mongodb-1

Parâmetro	***SQLite***	***Paper***	***CouchBase Lite***
Consistência	Principal ponto forte, permitindo integridade dos dados armazenados.	Apresenta uma eventual consistência.	Apresenta uma eventual consistência.
Flexibilidade	Possui uma estrutura rígida, por causa da integridade das tabelas. Aceitando apenas dados estruturados.	É bastante flexível, devido a facilidade de trabalhar com dados estruturados, semiestruturados e não estruturados	É bastante flexível, devido a facilidade de trabalhar com dados estruturados, semiestruturados e não estruturados
Grande volume de dados	Não lida bem com grande volume de dados.	Trabalha bem com o armazenamento de grandes volumes de dados.	Lida bem com o armazenamento de grandes volumes de dados.
Joins entre entidades/tabelas	Por ser um BD *SQL*, permite *Joins* de forma nativa.	Não possui suporte nativo para a realização de *Joins*. Tornando-se difícil a realização de algumas queries.	Possui suporte nativo para a execução de *Joins* nas *queries*.
Suporte a sincronização com BDs em cloud	Não apresenta integração, sendo necessária realizar um mapeamento das entidades e relacionamentos para armazenar os dados.	Não apresenta integração com BDs externos, necessitando consumir *APIs*.	Possui integração com o *CouchBase cloud*, sincronizando os dados automaticamente a partir de alterações locais.

Tabela 3.5: Diferenças entre os BDs SQL e NoSQL

Capítulo 4

Migração

Neste capítulo, será detalhada a migração para os bancos de dados escolhidos na Seção 3.3, mostrando as alterações e configurações realizadas no projeto. Também realizou-se testes de desempenho após a migração, com o objetivo de comparar os tempos de execução das operações, além disso, estudou-se o comportamento dos BDs em uma eventual mudança de estrutura.

4.1 Estrutura

O *CouchBase Lite* contém dois tipos de relacionamento entre as coleções, incorporado e normalizado, como visto no Capítulo **2**, contudo, o modelo a ser observado nesse estudo é o incorporado. O *Paper*, tal como o *CouchBase Lite*, utilizou o mesmo modelo. Torna-se necessário mencionar que os dados armazenados nos bancos de dados *NoSQL* são iguais aos do *Room*. Na Figura 4.1 é apresentado o diagrama entidade relacionamento do repositório e a Figura 4.2 traz o diagrama do modelo de dados usado nos bancos de dados não-relacionais.

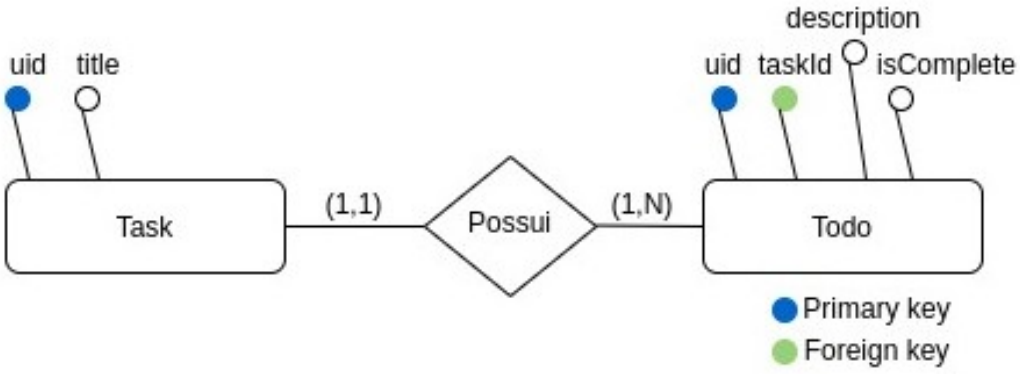

Figura 4.1: Entidades e relacionamento do banco de dados

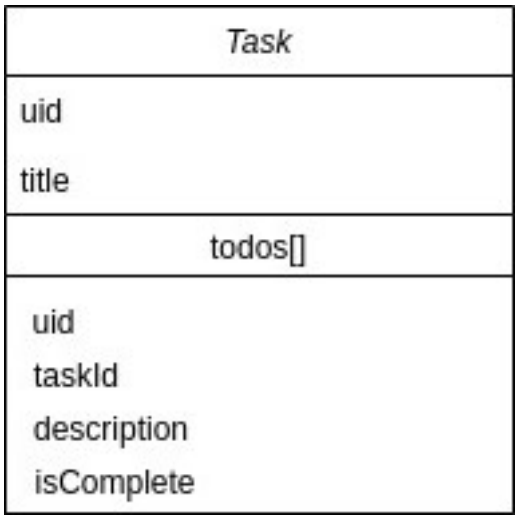

Figura 4.2: Modelo de Dados Incorporado

4.2 Configuração

Após a definição da estrutura dos dados, realizou-se a configuração dos BDs *NoSQL*, com base nisso inseriu-se as dependências necessárias para utilizar as *APIs* no projeto.

Para as *APIs* do *Paper* ficarem disponíveis no projeto *Android*, primeiro adiciona-se a seguinte linha "io.paperdb:paperdb:x.x.x"ao arquivo *build.gradle* - com a versão desejada - e em seguida deve-se inicializar apenas uma vez no método *Application.onCreate()*, através do "Paper.init(context)"passando-se o contexto da aplicação como parâmetro.

Para configurar o BD orientado a documentos, seguiu-se o procedimento adotado anteriormente, adicionando a dependência "com.couchbase.lite:couchbase-lite-android:x.x.x"e chamando o método "CouchbaseLite.init(context)"para ter acesso as funcionalidades. Depois da inclusão das dependências, criou-se uma classe para auxiliar na inicialização e configuração do banco de dados.

```
class CouchBaseDB(context: Context) {
    var db: Database

    init {

        val config = DatabaseConfiguration()

        config.directory = context.filesDir.absolutePath

        db = Database("db_tasks", config)

    }
}
```

4.3 Migração dos Dados do Room

Esta etapa descreve como foi realizada a migração dos dados do *Room* para os BDs não-relacionais. Para facilitar o entedimento de como o *Room* se comunica com os módulos da aplicação, foi criado um diagrama com os componentes do padrão arquitetural aplicado nesse projeto. Através da Figura 4.3 consegue-se visualizar essa estrutura, que se divide em: *UI* (*Activities*, *Fragments*, etc), *View Model* com *LiveData*[1], *ITaskModel* (interface disponibilizada por injeção de dependência, que é implementada por uma classe concreta para acessar o *DAO* do *Room*) e o *RoomDatabase* (formado pelo modelo *TaskEntity*, *DAO* e *SQLite*).

[1] *LiveData* é uma classe armazenadora de dados observável. Diferente de um observável comum, o LiveData conta com reconhecimento de ciclo de vida, ou seja, ele respeita o ciclo de vida de outros componentes do app, como atividades, fragmentos ou serviços. Disponível em: https://developer.android.com/topic/libraries/architecture/livedata?hl=pt-br

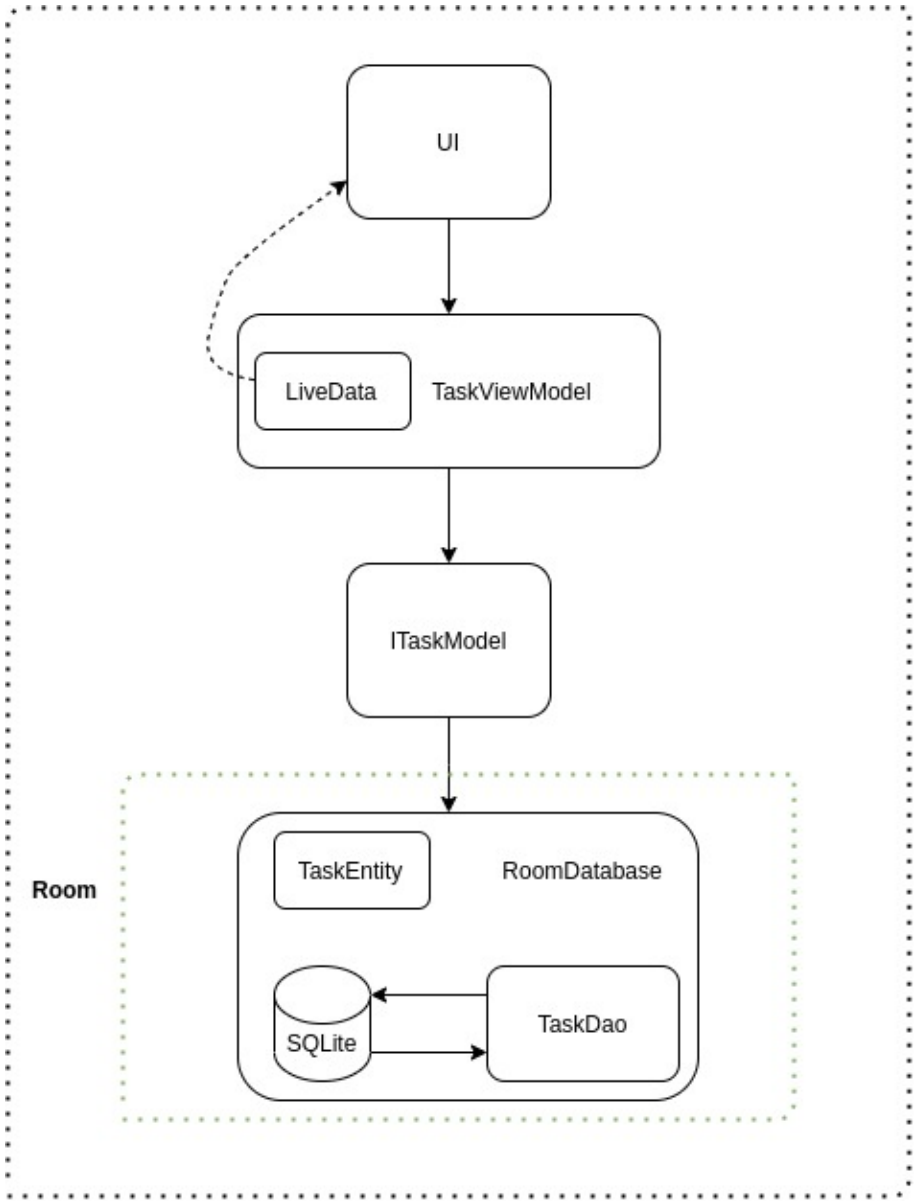

Figura 4.3: Componentes do padrão arquitetural

Inicialmente, não existiam dados salvos no *Room*, pois o aplicativo nunca tinha sido instalado. Logo, elaborou-se um método para gerar as *tasks* e outro para gerar uma quantidade aleatória de *todos*, com no máximo 5 objetos. Também criou-se um método para inserir registros das entidades *Task* e *Todo* no *Room*, com a finalidade de existirem dados para a migração. Esses métodos foram introduzidos na classe *TaskViewModel*.

```
    companion object {
        const val maxCount = 500
    }

    fun createTasks(): List<Task> {
        val tasks = mutableListOf<Task>()
        for (i in 0 until maxCount) {
          tasks.add(
            Task(title="Task$i", todos = generateTodos())
          )
        }
        return tasks
    }

    fun generateTodos(): List<Todo> {

      val todos = mutableListOf<Todo>()
      val randomAmountOfTodos = Random.nextInt(1, 6)

      for (i in 0 until randomAmountOfTodos) {
          todos.add( Todo(description = "todo $i"))
      }

      return todos
    }


typealias SuccessCallback = (Boolean) -> Unit

    interface ITaskModel {

        suspend fun addTask(task: Task, callback:
           SuccessCallback)
        [...]
```

```
        suspend fun deleteTask(task: Task, callback:
            SuccessCallback)
        suspend fun deleteAllTasks(callback:
            SuccessCallback)
        fun retrieveTasks(callback: (List<Task>?) -> Unit)
    }
    suspend fun insertTasksRoom(tasks: List<Task>) {

        tasks.forEach { task ->
            model.addTask(task)
        }
    }
```

Esse método recebe como parâmetro uma lista de *tasks* que é gerada a partir do método "createTasks"e, posteriormente, no método "insertTasksRoom"itera-se através de um *loop* para salvar cada objeto. O número de inserções foi definido a partir da constante *maxCount* e o valor, a princípio, foi de 500 registros. Para recuperar os dados do BD utilizou-se o método fornecido pela interface *ITaskModel*, onde é retornada uma lista de *Tasks* a partir da *query* "SELECT * FROM tasks".

```
    fun getTasksRoom(): List<Task> {

        return model.retrieveTasks()

    }
```

4.3.1 BDs NoSQL

Para inserir os dados nos BDs *NoSQL* foram criados dois métodos, os mesmos recebem como parâmetro a lista de objetos obtida no método descrito anteriormente. Procurando-se simplificar a inserção dos dados no *CouchBase Lite*, se fez uso da biblioteca *Gson*[2] para converter a lista de objetos da classe *Todo* em uma *String*. Com isso, pode-se observar um exemplo do documento salvo neste banco de dados e as mudanças necessárias para que isso seja possível.

[2]Disponível em: https://github.com/google/gson

Listing 4.1: Documento salvo no CouchBase Lite

```
{
    "uid": "1",
    "title":"title",
    "todos": [
        {
            "uid": 1,
            "taskId": "1",
            "description" :"desc",
            "isComplete": false
        }
    ]
}
```

```
private lateinit var couchDb: Database

init {
    couchDb = CouchBaseDB(context).db
}

fun insertTasksCouch(tasks: List<Task>) {
    Log.i(``Task",``START_TIME_TO_WRITE_DATA_ON_COUCH")
    tasks.forEach { task ->
     val mutableDoc = MutableDocument()
        task.uid = mutableDoc.id
        mutableDoc.setString("uid", mutableDoc.id)
        mutableDoc.setString("title", task.title)
        mutableDoc.setValue("todos", Gson().toJson(task
            .todos))

        try {
         couchDb.save(mutableDoc)
        } catch (e: CouchbaseLiteException) {
          Log.e(``ERROR",e.toString());
        }
    }
    Log.i(``Task",``END_TIME_TO_WRITE_DATA_ON_COUCH")
}
```

O método para a inserção de dados no *Paper* é mais simples do que o do banco de dados orientado a documentos, sendo necessário seguir estes passos: fazer uma leitura dos dados, se existirem, passando a chave como parâmetro no método "read(key)"e em seguida adicionar os valores a lista que foi retornada por esse método. Por último, os dados são inseridos passando-se a chave e a nova lista que foi gerada com o auxílio do método "write(key,object)".

```
fun insertDataOnPaper(key: String, tasks: List<Task>) {

  Log.i(``Task'',``START_TIME_TO_WRITE_DATA_ON_PAPER'')

  val tasksOnDB = Paper.book()
                  .read(key, arrayListOf<Task>())
  tasksOnDB.addAll(tasks)
  Paper.book().write(key,tasksOnDB)

  Log.i(``Task'',``END_TIME_TO_WRITE_DATA_ON_PAPER'')
}
```

Por fim, definiram-se testes para averiguar o desempenho dos BDs após a migração dos dados e compará-los com o *Room*, esse processo será evidenciado na Seção 4.4.

4.4 Testes de Benchmark

O teste de *benchmark* fundamenta-se na avaliação de um padrão de testes, sendo aplicado na análise do desempenho, essa avaliação pode ser definida como comparativa, através de uma comparação com outro sistema [47]. Segundo Ferreira[48], os programas de *benchmark* efetuam um número de operações pré-definidas, conhecidas como carga de trabalho, e apresentam um resultado em forma de métrica, relatando o comportamento do sistema.

Para a realização dos testes de desempenho, recorreu-se a um Dispositivo virtual *Android*[3] com a versão 10 do *Android*, 1536 MB de mémoria RAM dedicada e 6144 MB de armazenamento. Nesta etapa, foram realizados testes de inserção, leitura e remoção de dados estruturados[4] nos BDs *Room*, *Paper* e *Couchbase Lite*.

[3]Um Dispositivo virtual Android (AVD, na sigla em inglês) é uma configuração que define as características de um smartphone ou tablet Android, Wear OS, Android TV ou um dispositivo Automotive OS que você queira simular no Android Emulator. Disponível em: https://developer.android.com/studio/run/managing-avds?hl=pt-br

[4]Os dados estruturados são organizados em um padrão fixo e constante, seguem uma estrutura mais rígida.

Após as execuções, calculou-se uma média aritmética dos tempos obtidos em cada transação e os seus respectivos desvios padrões, a descrição dos dados e valores obtidos dos testes serão mostrados na sequência.

4.4.1 Teste de Inserção de Dados

Para a realização deste teste, os dados foram gerados a partir do método utilizado na Seção 4.3, no qual passou a retornar uma lista contendo 10 mil objetos da classe *Task* e cada objeto apresenta uma lista de *Todos* que é gerada randômicamente com um número máximo de 5 objetos.

O motivo para inserir essa quantidade de dados é o de simular um cenário, em que seja necessário lidar com um grande volume de dados, verificando-se desse jeito o comportamento de cada solução. A partir disso, utilizou-se os mesmos métodos descritos na Seção anterior, para os BDs *NoSQL*, e para cada BD repetiu-se três vezes o processo de inserção, desconsiderando-se a primeira execução, para observar-se o comportamento dos mesmos. Os BDs *Paper* e *Couchbase Lite* tiveram um melhor desempenho em relação ao *Room*, na Tabela abaixo consegue-se visualizar os tempos médios e o desvios padrões.

Banco de Dados	**Média**	**Desvio Padrão**
CouchBase Lite	5,943s	0,04
Paper	2,475s	0,89
Room	13,464s	0,34

Em seguida, realizou-se um teste de inserção para apenas um registro, com o objetivo de verificar o desempenho de cada transação, isoladamente. O banco de dados relacional obteve o melhor desempenho, seguido do *CouchBase Lite* e *Paper*. O BD chave-valor teve um desempenho muito abaixo em comparação aos seus concorrentes, a explicação para isso é que ele precisa carregar a lista de *Tasks* antes de adicionar o novo valor e, logo depois, salvá-la.

Banco de Dados	**Média**	**Desvio Padrão**
CouchBase Lite	21,66ms	0,02
Paper	1,346s	0,16
Room	7,66ms	0,01

4.4.2 Teste de Leitura de Dados

Este teste foi dividido em duas etapas: primeiramente, é observado o desempenho alcançado na leitura de todos os registros inseridos nos BDs e depois definiu-se uma *query* específica. A primeira *query* é igual a utilizada na migração dos dados

no caso do *Room* e para os outros BDs foram criados métodos que se assemelham a consulta executada no BD relacional. A quantidade de registros retornada nas consultas - tanto no BD relacional quanto nos *NoSQL* - foi de **30.500**, esse valor foi obtido por conta da migração e dos testes de inserção descritos previamente. Adiante, pode-se observar os métodos criados para a realização deste teste.

```
fun getTasksPaper(key: String): List<Task> {

    Log.i(``Task'',``START_TIME_TO_READ_DATA_ON_PAPER'')

    val tasksOnDB =
        Paper.book().read(key, arrayListOf<Task>())

    Log.i(``Task'',``END_TIME_TO_READ_DATA_ON_PAPER'')

    return tasksOnDB
}


private fun deserialize(results: List<Result>):
   MutableList<Task> {

    val lista = mutableListOf<Task>()
    val gson = Gson()

    results.forEach { result ->

      val dictionary = it.getDictionary("db_tasks")
        val uid = dictionary?.getString("uid")
        val title = dictionary?.getString("title")

        val todosString =
            dictionary?.getValue("todos") as String

        val type =
            object: TypeToken<MutableList<Todo>>(){}.
               type

        val todos: MutableList<Todo> =
                                gson.fromJson(todosString,
                                   type)
```

```
            val task = Task(
            title = title!!,
            todos = todos!!
            )
            task.uid = uid!!
            lista.add(task)
        }

        return lista
    }


    fun getTasksCouch(): List<Task> {

        Log.i(``Task'',``START_TIME_TO_READ_DATA_ON_COUCH'')
        val query = QueryBuilder
                    .select(SelectResult.all())
                    .from(DataSource.database(couchDb))
        val result = query.execute()
        try {
            val results = result.allResults()
            val tasksOnDB = deserialize(results)
         } catch (e: Exception) {
            Log.e("TaskError", e.message)
        }

        Log.i(``Task'',``END_TIME_TO_READ_DATA_ON_COUCH'')
        return tasksOnDB
    }
```

O *Paper* conseguiu o menor tempo médio, seguido por *CouchBase Lite* e *Room*. O BD relacional levou mais que o dobro do tempo para executar a mesma tarefa que o chave-valor, isso decorre por ele precisar realizar um *inner join* com a tabela *todos*, para recuperar os valores de cada *Task*.

Banco de Dados	**Média**	**Desvio Padrão**
CouchBase Lite	1,48s	0,03
Paper	0,806s	0,11
Room	1,94s	0,04

A segunda *query* tem como propósito retornar todas as *Tasks* que contenham quatro *Todos*. Para isso, utilizou-se a consulta "SELECT * FROM tasks

WHERE 4 IN(SELECT COUNT(todos.uid) FROM todos WHERE todos.taskId = tasks.uid)"no *Room* e para os não-relacionais alterou-se os métodos que traziam todos os dados, através da função *filter* presente na interface *MutableList*. O método abaixo recebe uma lista de *Tasks* e retorna uma nova lista com o filtro aplicado, fez-se uso desse método para as duas alterações citadas.

```
fun getTasksFiltered(tasks: MutableList<Task>):
    MutableList<Task> {

    return tasks.filter {
      task -> task.todos.count() == 4
    }

}
```

As médias de tempo e desvios para esse teste, evidenciaram, mais uma vez, um desempenho inferior do banco de dados relacional, os resultados estão presentes a seguir.

Banco de Dados	**Média**	**Desvio Padrão**
CouchBase Lite	1,682s	0,3
Paper	0,914s	0,17
Room	4,79s	0,55

4.4.3 Teste de Remoção de Dados

A última etapa concentrou-se na remoção de registros dos BDs, na qual divide-se em duas fases: a fase inicial compreendeu a remoção de apenas um registro e a segunda fase removeu todos os registros de cada ferramenta.

Nesta primeira fase, bem como nos testes anteriores, restringiu-se à observação das transações em separado. Assim sendo, para aplicar o teste no *Room*, recorreu-se ao método existente na interface *ITaskModel* - para remover o registro, em que um objeto da classe *Task* é passado como argumento.

No caso do BD orientado a documentos, o método é o mesmo que foi descrito no processo anterior. Para o BD chave-valor adotou-se o procedimento a seguir: primeiro carregou-se todos os dados e removeu-se o elemento da lista com o auxílio do método "remove(element)", no qual o parâmetro remete-se ao objeto que se dejesa remover e, logo após, salvou-se a lista.

```
fun deleteTaskPaper(key: String, task: Task) {

    Log.i(``Task", ``START_TIME_TO_DELETE")

    val tasks = Paper.book()
                    .read(key, arrayListOf<Task>())

    tasks.remove(task)

    Paper.book().write(key, tasks)

    Log.i(``Task", ``END_TIME_TO_DELETE")

}
```

Os resultados desse teste demonstram uma superioridade do banco de dados relacional em relação aos BDs *NoSQL*. O *Paper*, assim como na inserção de um registro, teve uma performance bem abaixo dos seus concorrentes.

Banco de Dados	**Média**	**Desvio Padrão**
CouchBase Lite	12,66ms	0,012
Paper	3,67s	0,034
Room	6ms	0,55

No BD relacional executou-se a *query*: *"DELETE FROM tasks"*, removendo-se os dados das *Tasks* e os registros da tabela *todos*, o *Room* cuida automaticamente de propagar essa remoção com a FK da tabela *tasks*. Já no *Paper*, é passada uma chave como argumento no método "*Paper.book().delete(key)*", para remover o valor associado à mesma. Por fim, no *Couchbase Lite* é utilizado o método "*db.delete(document)*", passando como parâmetro um *Document*. Pôde-se constatar com os resultados atingidos, que o chave-valor apresenta o melhor desempenho por ser preciso somente passar a chave para remover todos os registros. Enquanto que no *CouchBase Lite* recupera-se o documento antes de excluí-lo, por isso esse BD obteve o pior desempenho desse teste.

Banco de Dados	**Média**	**Desvio Padrão**
CouchBase Lite	6,96s	0,9
Paper	32ms	0,033
Room	138ms	0,55

4.5 Mudança de Estrutura

Com intenção de analisar como se portam os bancos de dados a partir de uma mudança na estrutura e no relacionamento das entidades, propôs-se dois cenários com modificações na classe *Task*. Antes de qualquer alteração nos BDs definiu-se uma nova classe, chamada de *Tag*, contendo um *label* e um inteiro para representar o endereço de memória de um recurso de cor[5]. A primeira mudança de esquema fundamenta-se na incorporação dos valores dessa classe na entidade *Task*, possibilitando, desta forma, agrupar as *tasks* pela sua *tag*. A segunda, por outro lado, visa retornar uma lista de *imagens* nas *tags*. Na sequência, estão descritos os procedimentos adotados em cada cenário para o BD *Room* e os BDs *NoSQL*.

```
data class Tag(
    @ColumnInfo
    val name: String,
    @ColumnInfo(name = "colour_resource_id")
    val colourResId: Int
)
```

4.5.1 Primeiro Cenário

Este cenário tem como principal objeto de estudo, verificar o comportamento das soluções observadas nesse projeto, tomando como base uma eventual alteração do esquema das entidades, quer dizer, uma adição de coluna na tabela do banco de dados relacional ou um novo objeto - incorporado - a ser retornado pelos BDs não-relacionais.

4.5.1.1 Alterações no Room

Para obter-se essa mudança na estrutura do *Room*, incluiu-se um atributo chamado *tag* nas classes *Task* e *TaskEntity*. Para isso, adotou-se o padrão sugerido pela documentação desse BD [49] para alterar a estrutura das tabelas/entidades. O *Room* apresenta uma anotação chamada de *Embedded*, que é usada para mapear os valores de colunas em um objeto, desse modo os dados da *tag* são preenchidos.

[5]Recurso XML que carrega um valor de cor (uma cor hexadecimal). Disponível em: https://developer.android.com/guide/topics/resources/more-resources

```
@Entity(tableName = "tasks")
open class TaskEntity(
    [...]
    @Embedded
    var tag: Tag? = null
)

class Task @JvmOverloads constructor(
    [...]
    tag: Tag? = null
) : TaskEntity(
    title = title,
    tag = tag
)
```

Logo após, tornou-se essencial efetuar uma migração do banco de dados *Room*, por causa da adição de novas colunas. Essa alteração consistiu na criação de um objeto da classe *Migration* [50], que é passado como argumento ao recuperar-se a instância do banco de dados *RoomDatabase*. Esse processo indica para o banco de dados que ocorreram alterações nas tabelas e na versão do banco de dados. No momento em que executou-se essa alteração, existiam dados com a estrutura antiga das tabelas e o BD realizou corretamente essa ação, preenchendo - na consulta, as *tags* com valores nulos.

```
const val DATABASE_SCHEMA_VERSION = 2
const val DB_NAME = "local-db"

@Database(version = DATABASE_SCHEMA_VERSION, entities = [
   TaskEntity::class, Todo::class, Tag::class])
abstract class RoomDatabaseClient : RoomDatabase() {

    val MIGRATION_1_2 = object : Migration(1, 2) {
        override fun migrate(database:
           SupportSQLiteDatabase) {
            database.execSQL(
                    "ALTER TABLE `tasks` ADD COLUMN `
                       colour_resource_id` INTEGER")
            database.execSQL(
                "ALTER TABLE `tasks` ADD COLUMN `name` TEXT
                    ")
        }
```

```
    }

    private fun createDatabase(context: Context):
        RoomDatabaseClient {
        return Room.databaseBuilder(context,
                RoomDatabaseClient::class.java, DB_NAME)
            .addMigrations(MIGRATION_1_2)
            .build()
    }
```

Depois, para validar se a *tag* estava sendo inserida de forma correta, usou-se o método aplicado nos testes de desempenho com uma pequena diferença, na qual, era instanciado um objeto da classe *Tag* gerado pelo método "addTag()"e, a seguir, passava-se ele no construtor da *Task* a ser inserida.

```
fun addTag(): Tag {
        val randomTag = Random.nextInt(1, 3)
        val tag = Tag(name = "tag$randomTag", colourResId =
            R.color.colorPrimaryDark )
        return tag
    }
```

4.5.1.2 Alterações no CouchBase Lite

A estrutura da classe *Task* - depois das alterações anteriores - permaneceu inalterada. As intervenções nos códigos que inserem e recuperam os dados no *CouchBase Lite*, decorreram-se de um modo mais simples do que no BD relacional. Esses métodos são semelhantes aos dos testes de *benchmark*, porém com as algumas diferenças. Na leitura dos dados, passou a recuperar uma chave que contém os dados da *tag*, se ela existir é feito o *parse*, senão retorna-se *null* e no caso da inserção, fez-se uso do método "setValue"para salvar o valor da *tag*. Desta forma, os métodos após as adequações apresentam-se listados abaixo nos *Listings*.

```
fun deserialize(results: List<Result>): MutableList<
    Task> {
    val lista = mutableListOf<Task>()
    val gson = Gson()
    results.forEach {
        [...]
        val tagString = dictionary?.getValue("tag") as?
            String
        val tag = if (tagString != null ) {
            gson.fromJson(tagString, Tag::class.java)
        } else {
            null
        }

        val task = Task(
            title = title!!,
            todos = todos!!,
            tag = tag
        )
        task.uid = uid!!
        lista.add(task)

    }

    return lista
}

fun insertTasksCouch(tasks: List<Task>) {
    tasks.forEach { task ->
        [...]
        mutableDoc.setValue("tag", Gson().toJson(task.
            tag))
        try {
            couchDb.save(mutableDoc)
        } catch (e: CouchbaseLiteException) {
            Log.e("ERROR", e.toString());
        }
    }
}
```

4.5.1.3 Alterações no Paper

Bem como nos BDs anteriores, manteve-se a estrutura da classe *Task* com o novo atributo *tag*. Como o método que gera a carga de *tasks* já tinha sofrido alterações para receber o novo atributo, dessa forma, nenhuma alteração foi executada. Assim sendo, o *Paper* conseguiu guardar e recuperar os dados sem nenhuma modificação que causasse impacto direto sobre essas ações.

4.5.2 Segundo Cenário

Esta etapa teve como foco central, uma mudança na estrutura da classe *Tag*, almejando-se, deste modo, estudar a flexibilidade da estrutura dos bancos de dados. Para atingir-se esse objetivo, praticaram-se modificações no método que gera os dados e adicionou-se - na classe em questão - uma lista de objetos para salvar dados de *urls* de imagens - assim, adquiriu-se um relacionamento de um-para-muitos.

4.5.2.1 Alterações no Room

Buscando-se atingir a proposta desse cenário, realizaram-se mudanças nas classes *Task, TaskEntity* e *Tag*, nas quais, a *tag* se tornou um *POJO* [6] e os dados dela passaram a ser mapeados no BD pela entidade nomeada de *TagEntity*. Já na classe *Task*, utilizou-se a anotação "@Relation"para sinalizar ao BD qual chave correspondia ao *id* da *task* e, com isso, retornar a *tag* correspondente. Para evitar uma incoerência no mapeamento dos dados pelo *Room*, removeu-se o atributo *tag* da entidade *TaskEntity*. Por último, fez-se uso de uma nova entidade para salvar os dados das imagens das *tags*, chamada de *TagImg*, ela armazena as *urls* dessas imagens. Para que essas novas entidades existissem no BD, inseriu-se uma migração para criar as tabelas no *Room*.

```
class Task @JvmOverloads constructor(
    [...]
    @Relation(
        entity = TagEntity::class,
        parentColumn = "uid",
        entityColumn = "taskId"
    )
    var tag: Tag? = null
)
```

[6] POJO do inglês Plain Old Java Objects, é um objeto Java sem ligação a restrições especiais. Disponível em: https://www.geeksforgeeks.org/pojo-vs-java-beans/

```
class Tag @JvmOverloads constructor(
    colourResId: Int,
    name: String,
    taskId: String? = null,
    @Relation(
        parentColumn = "tagUid",
        entityColumn = "tagId",
        entity = TagImg::class
    )
    val imgs: MutableList<TagImg> = mutableListOf()
) : TagEntity(
    taskId = taskId,
    name = name,
    colourResId = colourResId
){
    init {
        imgs.forEach {
            it.tagId = tagUid
        }
    }
}

@Entity(tableName = "tags")
open class TagEntity(
    @PrimaryKey
    var tagUid: String = UUID.randomUUID().
       leastSignificantBits.toString(),
    @ForeignKey(
        parentColumns = ["uid"],
        childColumns = ["taskId"],
        entity = TaskEntity::class,
        onDelete = CASCADE
    )
    val taskId: String? = null,
    @ColumnInfo val name: String,
    @ColumnInfo(name = "colour_resource_id")
    val colourResId: Int,
)
```

```
@Entity(tableName = "tag_imgs")
data class TagImg(
    @PrimaryKey(autoGenerate = true)
    var id: Int = 0,
    @ColumnInfo val url: String,
    @ForeignKey(
        parentColumns = ["tagUid"],
        childColumns = ["tagId"],
        entity = TagEntity::class,
        onDelete = CASCADE
    )
    var tagId: String? = null
)

val MIGRATION_2_3 = object : Migration(2, 3) {
    override fun migrate(database: SupportSQLiteDatabase) {
        database.execSQL(
            "CREATE TABLE `tags` (`tagUid` TEXT, `taskId`
               TEXT, `NAME` TEXT, `colour_resource_id`
               INTEGER, " +
                    "PRIMARY KEY(`tagUid`), FOREIGN KEY(`
                       taskId`) REFERENCES `tasks`(`uid`)")
        database.execSQL(
            "CREATE TABLE `tag_imgs` (`id` INTEGER, `url`
               TEXT, `tagId` TEXT, " +
                    "PRIMARY KEY(`id`), FOREIGN KEY(`tagId
                       `) REFERENCES `tags`(`tagUid`))")

    }
}
```

Outras modificações praticadas foram: a adição de um novo atributo no objeto gerado pelo método "addTag", para dessa maneira, a cada nova *tag* inserida, exista pelo menos uma imagem associada e o *id* da *task* foi passado como parâmetro. Os detalhes dessa mudança estão descritas no *Listing* adiante.

```
fun addTag(taskId: String): Tag {
    val randomTag = Random.nextInt(1, 3)
    val imgs = mutableListOf(TagImg(url = "
       preencherUrlImagem"))
    val tag = Tag(name = "tag$randomTag", colourResId =
        R.color.colorPrimaryDark, imgs = imgs, taskId =
        taskId)
    return tag
}
```

4.5.2.2 Alterações nos Bancos de Dados NoSQL

Os bancos de dados *NoSQL* passaram a utilizar a estrutura das classes *Task, Tag* e *TagImg*, que foi definida previamente na adequação para o BD *Room*. O método para gerar as *tasks* com as *tags* atreladas permaneceu, também, sem mudanças. Os dados salvos previamente no cenário anterior, tanto no *Paper* quanto no *CouchBase Lite*, foram preservados com essas alterações.

Capítulo 5

Análise Comparativa para Migração entre Bancos de Dados

Neste capítulo, serão analisados os dados gerados no capítulo anterior, além disso, se darão as diretrizes abordadas e alguns questionamentos para a realização da migração de um banco de dados *SQL* para um *NoSQL* em aplicações *Android*.

5.1 Análise dos Dados Obtidos

Esta análise tem como base os resultados trazidos anteriormente, ela encontra-se dividida em duas partes: a primeira concentra-se nos testes de desempenho executados pelos BDs e a segunda apresenta como foco as alterações praticadas nas estruturas das entidades e nos métodos de inserção e de carga dos dados.

5.1.1 Análise dos Testes de Benchmark

A partir dos tempos médios obtidos, atestou-se que o *Room* - nas tarefas que um usuário normalmente executa, tal como inserção e remoção de um objeto, apresentou um resultado melhor em comparação aos concorrentes. Contudo, em aplicações que necessitem salvar localmente vários dados, para o seu fluxo de vida, a performance desse BD é prejudicada. Vale salientar que na *query* criada para consultar uma quantidade específica de *todos*, esse BD também registrou um tempo elevado. Enquanto que os outros BDs executaram a tarefa de recuperar todos os registros para em seguida filtrar a lista, mesmo assim o *Room* ficou atrás deles.

Dentre os BDs analisados, o *Paper* conseguiu se sobressair nos testes de inserção e remoção de vários dados, por causa da sua característica principal de ser um BD chave-valor. Entretanto, por conta disso, ele teve uma performance abaixo dos concorrentes nas operações de inserção e remoção de apenas um regis-

tro. Necessitando recuperar todos os dados salvos no BD, para em seguida realizar a operação desejada.

O *CouchBase Lite* apresentou o segundo melhor desempenho, considerando-se o somatório dos tempos dos testes, ficando atrás do *Paper*. No teste de remoção de todos os dados, esse BD apresentou um resultado bem abaixo das outras soluções. Isso decorreu, por ser preciso recuperar os documentos antes de removê-los, penalizando, dessa forma, o seu desempenho.

5.1.2 Análise da Mudança de Estrutura

Esta análise concentra-se nos cenários criados para estudar a versatilidade dos BDs em modificar as estruturas das entidades do projeto. Isso é um fator importante no desenvolvimento de uma aplicação, pois os *schemas* demandam de uma evolução com o passar do tempo.

5.1.2.1 Primeira Mudança

O primeiro cenário apresentou uma alteração simples na classe *Task*, onde adicionou-se um atributo da nova classe *Tag*. A fim de que essa mudança surtisse efeito no *Room*, foi primordial inserir esse atributo na entidade *TaskEntity* e anotar ele com "@Embedded", para assim, o BD mapear os dados e, por fim, incluíram-se duas novas colunas na tabela *tasks*. Nesse caso, o BD relacional mostrou-se bem receptivo as mudanças executadas, no entanto, um ponto a ser ressaltado é a quantidade de passos para possibilitar esse feito. Para tal, precisa-se definir uma migração no *RoomDatabase* alterando-se a versão do banco de dados e criar um *script* com o comando desejado (*CREATE TABLE*, *ALTER TABLE*, etc).

Por outro lado, os bancos de dados não-relacionais sofreram poucas variações nos seus métodos. A única variação na estrutura da entidade foi aplicada na adaptação para o *Room*, como pontuado previamente. Desta forma, o BD orientado a documentos incluiu nos seus métodos de leitura e inserção de dados um novo valor para mapear a *tag*. Por último, o BD chave-valor não apresentou alteração nos seus métodos e conseguiu tanto salvar quanto recuperar, sem problemas, os valores.

5.1.2.2 Segunda Mudança

Este cenário expôs um grau de dificuldade acima da mudança anterior, no qual a estrutura da classe, recentemente criada, recebeu uma transformação. Visando testar, mais uma vez, a adaptabilidade dessas soluções propôs-se a adição de novos campos na classe *Tag* e de um novo relacionamento.

O *Room* teve que aplicar várias modificações para permitir essa nova configuração, antes de tudo, foram criadas duas novas entidades: *TagEntity* e *TagImg*. Outra alteração praticada, foi a criação de duas novas tabelas a partir de uma *migration* no banco de dados, para salvar os dados dessas entidades.

Logo depois, os estudos direcionaram-se para os BDs *NoSQL*. Esses BDs fizeram uso da nova estrutura da *tag*, criada para o *Room* e no caso dos registros das *tags* que não continham a lista de imagens, eles adicionavam o valor padrão indicado no construtor. Os métodos de salvar e recuperar os dados, assim como no cenário antecedente, mantiveram-se iguais.

Alicerçado nos resultados gerados por esta análise, pode-se afirmar que, conforme são incluídos novos relacionamentos e mudanças nas entidades, o trabalho para executar essas modificações no banco de dados relacional torna-se mais complexo, em relação aos BDs *NoSQL*. Pois, além da criação dessas entidades é imprescindível criar migrações, classes POJO e relações, corroborando o fato de que esse BD não apresenta esquemas flexíveis.

5.1.3 Resultados da Migração

A migração do *Room* para o *Paper* e o *CouchBase Lite*, foi feita analisando-se os relacionamentos entre as entidades. Observou-se que não era necessária a criação de duas tabelas, como no BD *SQL*. Então, um modelo com dados incorporados foi escolhido para as soluções *NoSQL* e os impactos dessa escolha refletiram positivamente nos testes.

Logo, para o caso analisado, o BD *Paper* mostra-se como uma boa opção, por ser leve e de fácil utilização. Outra sugestão de aplicação para esse BD, é a substituição do uso das *SharedPreferences*[1]. No entanto, em uma eventual evolução do *app* em que fosse necessária a comunicação do mesmo com um BD hospedado na nuvem - por exemplo - permitindo a sincronização dos dados em outros dispositivos, a escolha ideal seria o *CouchBase Lite*, devido ao desempenho apresentado e a possibilidade de integração com uma solução *cloud*.

5.1.4 Comparação do Estado da Arte pós Trabalho

Neste trabalho, foram propostas além da migração dos dados de um BD relacional para BDs não-relacionais e testes de desempenho para obter-se uma análise dos tempos médios de execução, contemplou-se uma eventual mudança na estrutura do modelo de dados. Caracterizando desse modo, aspectos que não haviam sido levado em conta nos trabalhos elencados.

[1]Interface para salvar pares de chave-valor. Disponível em: https://developer.android.com/training/data-storage/shared-preferences?hl=pt-br

Devido ao pouco conhecimento existente sobre a temática de bancos de dados *NoSQL* incorporados em aplicações Android, após a construção deste trabalho podem surgir novos estudos que se aprofundem nesse tema.

5.2 Passos para a Migração

A primeira etapa do processo seria questionar se, é possível realizar a migração? Essa pergunta não apresenta uma resposta trivial, por ser essencial levar em consideração alguns fatores. O primeiro deles é a mudança de paradigma, isto é, deve-se descontruir a ideia original de como o banco de dados relacional foi concebido. Partindo-se desse ponto, o primeiro procedimento a ser exercido é um mapeamento das entidades e relacionamentos entre as mesmas. Tendo isso em mente, algo que pode auxiliar na visualização de como os modelos de dados se comunicam é uma ferramenta de criação de diagramas. Para este projeto, utilizou-se a ferramenta *Diagrams*[2], tanto na construção do modelo de dados incorporado quanto no diagrama ER.

Outro fator importante a ser considerado, é se a aplicação comunica-se com um BD *NoSQL* no *back-end*. Desse modo, o esforço para a realização do mapeamento do banco local da aplicação para o que está localizado no servidor, pode ser minimizado. Também deve-se levar em conta, se o projeto necessita de mudanças na estrutura de suas entidades/tabelas, um exemplo disso é a alteração ou a adição de um tipo de atributo. Os BDs relacionais são sensíveis a essas mudanças, no caso do *Room* é essencial realizar uma migração para conseguir-se o esperado. Por outro lado, os BDs *NoSQL* possuem mecanismos na sua estrutura que auxiliam nessas alterações.

Então, após a aplicação de todos os passos citados e questionamentos levantados, se os critérios propostos para a migração forem atingidos, pode-se responder a pergunta do início dessa Seção com a seguinte afirmação: a migração é, sim, possível.

5.2.1 Considerações Finais

No presente capítulo, tomando-se como base os resultados do capítulo anterior, analisou-se o desempenho de cada ferramenta e obteve-se conclusões dos pontos fortes e fracos na execução dos testes. Também foram apresentados cenários de mudança nos *schemas* da aplicação, com a finalidade de analisar a flexibilidade dos BDs. Após isto, foram sugeridas diretrizes para a migração de um banco relacional para não-relacional.

[2]Disponível em: https://www.diagrams.net/

Com isso em vista, enxerga-se como pontos futuros de melhoria para este trabalho: a formulação de um algorítimo de migração entre bancos relacionais e não-relacionais e uma análise mais aprofundada das vantagens e desvantagens de uma aplicação de ponta-a-ponta utilizando-se bancos de dados *NoSQL*.

Capítulo 6

Conclusão

Os principais objetivos deste trabalho foram: realizar uma análise comparativa entre bancos de dados relacionais e não-relacionais utilizados em aplicações *Android* e fornecer direções para uma migração entre os mesmos. Com os resultados trazidos pelos testes e os pontos levantados no capítulo anterior, foi permitido observar que as opções *NoSQL* apresentam-se como o futuro no desenvolvimento de aplicações *mobile*, que utilizam essas ferramentas no seu fluxo de vida.

Um ponto a ser levantado na discussão seria, a adaptação com a estrutura na qual o desenvolvedor pode armazenar os dados. Visto que, a mudança de paradigma pode ser considerada mais impeditiva do que a própria execução da migração. Por vezes, busquei aplicar a forma em que estruturava entidades e relacionamentos do modelo relacional quando não era necessário em um BD *NoSQL*.

Em suma, a realização deste trabalho tornou-se esclarecedora, no ponto de vista de um desenvolvedor de aplicações *Android* nativas. Abrindo-se dessa maneira, opções fora do convencional no desenvolvimento de aplicações escaláveis, com melhor desempenho e versatilidade em adaptar-se a mudanças constantes.

Bibliografia

[1] "NoSQL Databases." endereço: https://www.ibm.com/cloud/learn/nosql-databases, (acessado em: 21.06.2021).

[2] B. Marr, "How Much Data Do We Create Every Day? The Mind-Blowing Stats Everyone Should Read." endereço: https://www.forbes.com/sites/bernardmarr/2018/05/21/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read/?sh=2070135960ba, (acessado em: 21.06.2021).

[3] M. Roser, H. Ritchie e E. Ortiz-Ospina, "Internet." endereço: https://ourworldindata.org/internet, (acessado em: 21.06.2021).

[4] "The Most Popular Databases – 2006/2021." endereço: https://statisticsanddata.org/data/the-most-popular-databases-2006-2021/#:~:text=What%20is%20the%20most%20popular,is%20SQL%20Server%20with%2013.21%25., (acessado em: 11.02.2021).

[5] S. Lee, "Creating and Using Databases for Android Applications," *International Journal of Database Theory and Application*, v. 5, n. 2, 2012.

[6] V. S. Garcia e E. C. S. Sotto, "COMPARATIVO ENTRE OS MODELOS DE BANCO DE DADOS RELACIONAL E NÃO-RELACIONAL," *Interface Tecnológica*, v. 16, n. 2, 2019.

[7] C. Khawas e P. Shah, "Application of Firebase in Android App Development-A Study," *International Journal of Computer Applications*, v. 179, n. 46, 2018.

[8] J. Walker, "5 ways NoSQL can help your business grow," Monitis, 2016. endereço: https://www.monitis.com/blog/5-ways-nosql-can-help-your-business-grow/, (acessado em: 11.02.2021).

[9] "O Que É um Banco de Dados Relacional." endereço: https://www.oracle.com/br/database/what-is-a-relational-database/, (acessado em: 13.05.2021).

[10] S. M. Macário Carla Geovana do N. e Baldo, "O Modelo Relacional," 2005, pp. 1–2.

[11] "Compreendendo bancos de dados relacionais." endereço: https://www.digitalocean.com/community/tutorials/understanding-relational-databases-pt, (acessado em: 11.05.2021).

[12] C. J. Date, "Introdução a Sistemas de Bancos de Dados," 2004, pp. 50–51.

[13] "Abordagem Relacional." endereço: https://www.gsigma.ufsc.br/~popov/aulas/bd1/abordagem_relacional.html, (acessado em: 13.05.2021).

[14] "Banco de dados e SQL: Chaves," Dev, 2020. endereço: https://dev.to/thmmarra/banco-de-dados-e-sql-chaves-1dia, (acessado em: 05.03.2021).

[15] "Relacionamento entre entidades: tipos e cardinalidade." endereço: https://www.luis.blog.br/relacionamento-entre-entidades-tipos-e-cardinalidade.html, (acessado em: 11.02.2021).

[16] P. W. S. Da Costa, "Uma Abordagem para Escolha entre os Paradigmas de Banco de Dados Relacional e NoSQL em Projetos de Software," 2019, pp. 19–22.

[17] "Os principais SGBDs relacionais," TreinaWeb, 2019. endereço: https://www.treinaweb.com.br/blog/os-principais-sgbds-relacionais/, (acessado em: 10.02.2021).

[18] V. Sharma e M. Dave, "SQL and NoSQL Databases," *International Journal of Advanced Research in Computer Science and Software Engineering*, v. 2, n. 46, 2012.

[19] P. Barros, "O que é ACID?," Medium, 2016. endereço: https://medium.com/opensanca/o-que-%C3%A9-acid-59b11a81e2c6, (acessado em: 11.02.2021).

[20] "What Is SQLite?," SQLite. endereço: https://www.sqlite.org/index.html, (acessado em: 05.03.2021).

[21] "SQLiteOpenHelper." endereço: https://developer.android.com/reference/android/database/sqlite/SQLiteOpenHelper#onCreate(android.database.sqlite.SQLiteDatabase), (acessado em: 21.06.2021).

[22] "Salvar dados em um banco de dados local usando o Room," Google, 2021. endereço: https://developer.android.com/training/data-storage/room, (acessado em: 10.02.2021).

[23] "Introdução ao Banco de Dados Room e LiveData — Android Jetpack." endereço: https://medium.com/@alifyzfpires/introdu%C3%A7%C3%A3o-ao-banco-de-dados-room-e-livedata-android-jetpack-33b2f58b0ae8, (acessado em: 21.06.2021).

[24] F. A. d. M. Farias, "Avaliação de Desempenho entre Bancos de Dados Relacionais e NoSQL," 2014, pp. 17–18.

[25] A. Porcelli, "O que é noSQL?," DevMedia, 2010. endereço: https://www.devmedia.com.br/o-que-e-nosql-java-magazine-86/18777#:~:text=Nota%3A%20Ao%20que%20tudo%20indica,n%C3%A3o%20expor%20a%20interface%20SQL., (acessado em: 11.02.2021).

[26] "O que é NoSQL?." endereço: https://aws.amazon.com/pt/nosql/, (acessado em: 13.05.2021).

[27] M. Pune, "NoSQL: A Database for Cloud Computing," *IJCSN - International Journal of Computer Science and Network*, v. 3, n. 6, 2014.

[28] W. B. Rodrigues, "NoSQL - A análise da modelagem e consistência dos dados na era do Big Data.," 2017, pp. 27–28.

[29] "Banco de dados NoSQL: um manual prático e didático," 2019. endereço: https://blog.geekhunter.com.br/banco-de-dados-nosql-um-manual-pratico-e-didatico/, (acessado em: 20.03.2021).

[30] "O que é um banco de dados em colunas?," AWS. endereço: https://aws.amazon.com/pt/nosql/columnar/, (acessado em: 11.02.2021).

[31] P. Oliveira, R. K. Stamboni e J. F. R. Jr, "Analisando o desempenho de bancos de dados orientados a grafos e relacionais sobre discos mecânicos e de estado sólido: uma abordagem comparativa," 2016, p. 3.

[32] "O que é um banco de dados de chave-valor?," AWS. endereço: https://aws.amazon.com/pt/nosql/key-value/, (acessado em: 11.02.2021).

[33] "O que é um banco de dados de documentos?," AWS. endereço: https://aws.amazon.com/pt/nosql/key-value/, (acessado em: 11.02.2021).

[34] G. Panassol, "Como posso fazer modelagem de dados em MongoDB?," Medium, 2018. endereço: https://medium.com/@gpanassol/como-posso-fazer-modelagem-de-dados-em-mongodb-ea61268ee10b, (acessado em: 11.02.2021).

[35] T. Perrier e F. Pervaiz, "NoSQL in a Mobile World: Benchmarking Embedded Mobile Databases," University of Washington, 2013. endereço: https://courses.cs.washington.edu/courses/cse544/13sp/final-projects/p19-tperrier.pdf, (acessado em: 30.01.2021).

[36] D. Dissanayaka, "Performance evaluation of embedded mobile databases: RDBMS VS NoSQL," University of Moratuwa, 2015. endereço: http://dl.lib.uom.lk/bitstream/handle/123/12205/pre-text.pdf?sequence=1&isAllowed=y, (acessado em: 30.01.2021).

[37] B. Gököz, O. Duman e M. Kolçak, "Examining Database Optimization Using Database Management System Models in a Mobile Application," 2018. endereço: https://www.researchgate.net/profile/Baki-Goekgoez/publication/331174842_Examining_Database_Optimization_Using_Database_Management_System_Models_in_a_Mobile_Application/links/5c6aacec4585156b5703683a/Examining-Database-Optimization-Using-Database-Management-System-Models-in-a-Mobile-Application.pdf, (acessado em: 30.01.2021).

[38] D. Hammes, H. Medero e H. Mitchell, "Comparison of NoSQL and SQL Databases in the Cloud," Georgia Southern University, 2014. endereço: http://aisel.aisnet.org/sais2014/12?utm_source=aisel.aisnet.org%2Fsais2014%2F12&utm_medium=PDF&utm_campaign=PDFCoverPages, (acessado em: 30.01.2021).

[39] "Top Programming Languages for Android App Development." endereço: https://www.geeksforgeeks.org/top-programming-languages-for-android-app-development/, (acessado em: 21.05.2021).

[40] K. Carter, "How Android App Development Became Kotlin-first." endereço: https://medium.com/hackernoon/how-android-app-development-became-kotlin-first-c79e493e02fb, (acessado em: 21.05.2021).

[41] "Android Architecture Patterns." endereço: https://www.geeksforgeeks.org/android-architecture-patterns/, (acessado em: 21.05.2021).

[42] "Guia para a arquitetura do app." endereço: https://developer.android.com/jetpack/guide?hl=pt-br, (acessado em: 21.05.2021).

[43] J. R. Lourenço, B. Cabral, P. Carreiro, M. Vieira e J. Bernardino, "Choosing the right NoSQL database for the job: a quality attribute evaluation," *Journal of Big Data*, v. 2, n. 18, 2015.

[44] "LevelDB library documentation." endereço: https://github.com/google/leveldb/blob/master/doc/index.md, (acessado em: 21.06.2021).

[45] "Paper." endereço: https://github.com/pilgr/Paper, (acessado em: 11.02.2021).

[46] "Introducing Couchbase Lite," Couchbase. endereço: https://docs.couchbase.com/couchbase-lite/current/index.html, (acessado em: 10.02.2021).

[47] "What Is a Benchmark?." endereço: https://www.lifewire.com/what-is-a-benchmark-2625811, (acessado em: 02.07.2021).

[48] E. R. Ferreira e S. M. T. Junior, "Análise de desempenho de Bancos de Dados," 2012, p. 6.

[49] "Definir relações entre objetos." endereço: https://developer.android.com/training/data-storage/room/relationships, (acessado em: 05.07.2021).

[50] “Como migrar bancos de dados Room.” endereço: https://developer.android.com/training/data-storage/room/migrating-db-versions, (acessado em: 05.07.2021).