UNIVERSIDADE FEDERAL RURAL DO RIO DE JANEIRO
INSTITUTO MULTIDISCIPLINAR

GUILHERME PEREIRA MOURA DA COSTA
LUIZ FILIPE BRANDI DO NASCIMENTO

Luigui: um Sistema Web para Text-To-SQL baseado em LLM utilizando a técnica de RAG

Prof. Filipe Braida do Carmo, D.Sc.
Orientador

Nova Iguaçu, Junho de 2025

Luigui: um Sistema Web para Text-To-SQL baseado em LLM utilizando a técnica de RAG

Guilherme Pereira Moura da Costa

Luiz Filipe Brandi do Nascimento

Projeto Final de Curso submetido ao Departamento de Ciência da Computação do Instituto Multidisciplinar da Universidade Federal Rural do Rio de Janeiro como parte dos requisitos necessários para obtenção do grau de Bacharel em Ciência da Computação.

Apresentado por:

____

Guilherme Pereira Moura da Costa

____

Luiz Filipe Brandi do Nascimento

Aprovado por:

____

Prof. Filipe Braida do Carmo, D.Sc.

____

Prof. Bruno José Dembogurski, D.Sc.

____

Prof. Leandro Guimarães Marques Alvim, D.Sc.

NOVA IGUAÇU, RJ - BRASIL

Junho de 2025

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL RURAL DO RIO DE JANEIRO
SISTEMA INTEGRADO DE PATRIMÔNIO, ADMINISTRAÇÃO E CONTRATOS

FOLHA DE ASSINATURAS

DOCUMENTOS COMPROBATÓRIOS Nº 14864/2025 - CoordCGCC (12.28.01.00.00.98)

(Nº do Protocolo: NÃO PROTOCOLADO)

(Assinado digitalmente em 08/07/2025 15:45)
BRUNO JOSE DEMBOGURSKI
PROFESSOR DO MAGISTERIO SUPERIOR
DeptCC/IM (12.28.01.00.00.83)
Matrícula: ###249#4

(Assinado digitalmente em 04/07/2025 15:31)
FILIPE BRAIDA DO CARMO
PROFESSOR DO MAGISTERIO SUPERIOR
DeptCC/IM (12.28.01.00.00.83)
Matrícula: ###295#4

(Assinado digitalmente em 08/07/2025 08:46)
LEANDRO GUIMARAES MARQUES ALVIM
PROFESSOR DO MAGISTERIO SUPERIOR
DeptCC/IM (12.28.01.00.00.83)
Matrícula: ###008#2

(Assinado digitalmente em 05/07/2025 12:32)
GUILHERME PEREIRA MOURA DA COSTA
DISCENTE
Matrícula: 2021######5

(Assinado digitalmente em 04/07/2025 15:23)
LUIZ FILIPE BRANDI DO NASCIMENTO
DISCENTE
Matrícula: 2021######6

*Visualize o documento original em https://sipac.ufrrj.br/documentos/ informando seu número: **14864**, ano: **2025**, tipo: **DOCUMENTOS COMPROBATÓRIOS**, data de emissão: **04/07/2025** e o código de verificação: **f486d7f9bb***

“Ao descobrir em mim um desejo
que nenhuma experiência deste mundo
poderia satisfazer, a explicação mais provável
é que eu tenha sido feito para outro mundo.”

— *C.S. Lewis, Cristianismo Puro e Simples*

Agradecimentos

Guilherme Pereira Moura da Costa

Agradeço a Deus, criador dos céus e da terra, que com toda a sua graça me sustentou durante todos esses anos de graduação, sei que tudo que fiz, aprendi, falei e conquistei até aqui não foi pela minha força, mas pela providência dEle, então dedico esse trabalho ao Deus que me ajudou, desde o ano que comecei a estudar pro vestibular até o último período que me dediquei a ser aprovado nas disciplinas e nesse trabalho final do curso. A Ele toda glória, honra e poder, amém!

Agradeço aos meus pais, Paulo Sérgio e Georgina, e também a minha vó, Terezinha Alves de Moura que me ajudaram inúmeras vezes durante a graduação, sei que sem a ajuda e o apoio deles, nada disso seria possível, muito obrigado pelo apoio até aqui e agradeço a Deus por ter pais que me deram o privilégio de estudar, obrigado, amo vocês.

Agradeço ao meu parceiro de trabalho e melhor amigo, Luiz Filipe Brandi. Obrigado pelas risadas, conselhos, apoio, compreensão, e paciência comigo durante esses 4 anos. Sei que sua vida também é graça de Deus na minha vida e agradeço por ter conhecido um irmão tão precioso na faculdade, e espero que nossa amizade perdure por toda vida para além da graduação, sei que para além desse trabalho, você é alguém que impacta positivamente minha vida como estudante, filho, irmão, amigo e cristão, obrigado por tudo, meu amigo.

Agradeço aos meus amigos Hugo Vidal, Guilherme Macedo, Josué Bueno, Camily Pacheco, João Paulo Boechat, João Pedro Felix, Juliana Nogueira, João Victor Azevedo, Maxwel Batalha, Joyce Rangel, Pedro Poiares e Bianca Santana que conheci

desde os primeiros períodos do curso, sei que a amizade de vocês também é preciosa e a união que fizemos durante cada desafio e também momento de descontração foi crucial para suportar o curso. Vocês fizeram toda diferença nessa caminhada, desde as horas jogando *Gartic* no intervalo até o apoio nos momentos críticos durante as semanas de provas, obrigado por tudo que vocês me proporcionaram.

Agradeço aos meus amigos Rafael Ferreira, Renan Torreira, Brendon Lima e Victor Sodré que conheci ainda no Ensino Médio, a amizade de vocês é importante pra mim e desde o início do curso conversamos sobre meus dilemas, dificuldades e também contei sobre tudo que estava aprendendo, obrigado por cada risada e momentos juntos, pessoal, espero poder retribuir o apoio de cada um e oro pela vida de vocês sempre.

Agradeço ao meu professor e orientador Filipe Braida que nos ajudou durante todo esse processo e com paciência nos orientou da melhor forma que conseguiu. Obrigado pelas conversas, dicas e aprendizados, você é um exemplo profissional e acadêmico para mim. Além disso, agradeço a todos os professores do Departamento de Ciência da Computação da UFRRJ que me ensinaram muitas coisas durante essa fase da minha vida, sei que levarei tudo pro resto dela.

Agradeço também meus amigos que conheci na Igreja Presbiteriana, Roney Moraes, Gabriel Quirino, Maicon Marques, Felipe Beltrão, Marcelus Marques e Luiz Felipe Sangi que me apoiaram em várias situações durante esses anos de graduação, sei que a amizade de vocês contribuiu para que eu encontrasse alento e conforto em momentos de frustração, e cada risada e conversa me fortaleceu para continuar firme na faculdade, agradeço a Deus pela vida de cada um que e rogo para que o Senhor guarde vocês sempre.

Agradeço também pela CRU Campus e CRU Rural do Instituto Multidisciplinar (IM) que me apoiaram e me acolheram durante a maior parte da graduação. Essa comunidade foi essencial para manutenção da minha saúde mental e espiritual, e me ajudaram a manter minha fé, o amor, e a motivação na faculdade, agradeço por cada um que conheci tanto no IM quanto nos eventos da CRU nesses 4 anos, por cada amizade, risada e conforto, sou grato a Deus pela vida de cada um, em especial

destaco aqui meus amigos da liderança da CRU IM, Elizama Silva, Adrian Santos, Clara Garcia, Raquel Paiva, Lucas Silva, Lucas Novaes, Érika Nona e Mariana Marques.

Luiz Filipe Brandi do Nascimento

Agradeço a Deus, supremo benfeitor, cuja bondade e misericórdia me seguiram durante todos esses anos de graduação. Agradeço ao Deus Pai, pois em sua providência divina conduziu a história para que eu vivesse esses momentos que no passado eram apenas um sonho. Agradeço ao Deus Filho, pois me permitiu vislumbrar diariamente aspectos da redenção final que só ele mesmo pode realizar. Agradeço ao Deus Espírito, pois suas misericórdias me consolaram nos dias de angústia. Ao Deus triúno, glória e louvor sejam dados por toda a eternidade.

Agradeço aos meus pais, João e Silvia, por todo apoio e torcida durante toda minha trajetória acadêmica, mesmo sem entenderem muito bem o que eu faço. Vocês supriram as necessidades para que eu pudesse ser o homem e o profissional que hoje sou, muito obrigado.

Agradeço ao Guilherme Moura, meu parceiro neste trabalho e também meu melhor amigo. Um verdadeiro amigo, mais chegado que um irmão. Sua amizade é um sinal claro da bondade e do cuidado de Deus em minha vida. Durante os últimos anos, pudemos dividir muitos momentos de companheirismo, consolo e crescimento, que vão para além da faculdade. Foram anos repletos de risadas, muitas vezes por coisas que nem faziam sentido, mas que tornaram os dias mais leves. Sou profundamente grato por nossa amizade e espero que ela dure por toda a vida.

Também agradeço aos meus amigos do curso de Ciência da Computação, Hugo Vidal, Guilherme Macedo, Josué Bueno, Camily Pacheco, João Paulo Boechat, João Pedro Felix, Juliana Nogueira, João Victor Azevedo, Maxwel Batalha, Joyce Rangel, Pedro Poiares e Bianca Santana, sem a amizade de vocês esses anos teriam sido bem mais difíceis. Todas as horas que passamos estudando, reclamando ou jogando fizeram a graduação ser mais divertida. Valeu a pena fazer parte do grupinho da palhaçadinha.

Agradeço ao professor Filipe Braida, meu orientador neste trabalho, por toda paciência e suporte durante esse processo. Obrigado pelas conversas e incentivos, pois eles fizeram diferença nesse processo. Além disso, agradeço a todos os professores do

Departamento de Ciência da Computação da UFRRJ que contribuíram para minha formação como profissional.

Agradeço também aos meus amigos da Cru Campus, vocês foram meu consolo durante esses anos. Vocês me mostraram como viver a minha graduação para a glória de Deus. Não só isso, me mostraram o poder da vida em comunidade, certamente isso ficará guardado em meu coração. Agradeço especialmente a Elizama Silva, Adrian Santos, Clara Garcia, Raquel Paiva, Lucas Silva, Lucas Novaes, Érika Nona e Mariana Marques.

RESUMO

Luigui: um Sistema Web para Text-To-SQL baseado em LLM utilizando a técnica de RAG

Guilherme Pereira Moura da Costa e Luiz Filipe Brandi do Nascimento

Junho/2025

Orientador: Filipe Braida do Carmo, D.Sc.

O crescente volume de dados gerados por empresas, instituições e organizações de diferentes setores têm intensificado a necessidade de ferramentas eficazes para armazenamento, manipulação e análise dessas informações. Nesse cenário, os Sistemas de Gerenciamento de Bancos de Dados (SGBDs) desempenham um papel fundamental, permitindo que dados estruturados sejam acessados e manipulados com eficiência. A linguagem Structured Query Language (SQL) consolidou-se como padrão para essas interações, sendo amplamente utilizada por desenvolvedores. No entanto, seu domínio ainda representa uma barreira significativa para usuários sem formação técnica. Em ambientes corporativos, é comum que profissionais de áreas como Recursos Humanos, Marketing e Educação necessitem consultar dados diretamente dos bancos de dados, mas enfrentem limitações devido à falta de conhecimento técnico ou restrições de acesso. Essas barreiras comprometem a autonomia desses colaboradores e podem gerar gargalos operacionais, uma vez que dependem constantemente da equipe técnica para obtenção de informações. Nesse contexto, surgem os sistemas de Text-to-SQL como uma alternativa promissora. Ao permitir que usuários realizem consultas por meio da linguagem natural, esses sistemas traduzem automaticamente as perguntas em comandos SQL. O avanço dos Large Language Models (LLMs), viabilizou novas abordagens para esse problema, dado seu alto desempenho em tarefas de compreensão e geração de linguagem. Este trabalho propõe o desenvolvimento de um sistema web que utiliza LLM em conjunto com a técnica de Retrieval-Augmented Generation (RAG) para permitir a geração automática de consultas SQL a partir de perguntas em linguagem natural. A solução é voltada para usuários não técnicos, com foco na democratização do acesso aos dados.

ABSTRACT

Luigui: um Sistema Web para Text-To-SQL baseado em LLM utilizando a técnica de RAG

Guilherme Pereira Moura da Costa and Luiz Filipe Brandi do Nascimento

Junho/2025

Advisor: Filipe Braida do Carmo, D.Sc.

The growing volume of data generated by companies, institutions, and organizations across various sectors has intensified the need for effective tools for storing, manipulating, and analyzing such information. In this scenario, Database Management Systems (DBMS) play a fundamental role by enabling structured data to be accessed and handled efficiently. The Structured Query Language (SQL) language has become the standard for these interactions and is widely used by developers. However, mastering SQL still represents a significant barrier for users without technical training. In corporate environments, it is common for professionals from areas such as Human Resources, Marketing, and Education to need to query data directly from databases, but they often face limitations due to a lack of technical knowledge or access restrictions. These barriers compromise the autonomy of such employees and can create operational bottlenecks, as they constantly depend on the technical team to retrieve information. In this context, Text-to-SQL systems emerge as a promising alternative. By allowing users to perform queries through natural language, these systems automatically translate questions into SQL commands. The advancement of Large Language Models (LLMs) has enabled new approaches to this problem, given their strong performance in language understanding and generation tasks. This work proposes the development of a web-based system that leverages LLMs in combination with the Retrieval-Augmented Generation (RAG) technique to enable the automatic generation of SQL queries from natural language questions. The solution is designed for non-technical users, focusing on democratizing access to data.

Lista de Figuras

Figura 2.1: Representação esquemática comparativa de um modelo de linguagem treinado para a tarefa de tradução e um LLM que recebe como entrada um *prompt* com a instrução de traduzir um texto. . 11

Figura 2.2: Arquitetura *Naive RAG*. 17

Figura 4.1: Função de Text-To-SQL na ferramenta EverSQL. 37

Figura 4.2: Função de Text-To-SQL na ferramenta SQLAI.ai. 37

Figura 4.3: Criação de consultas na ferramenta Draxlr. 38

Figura 4.4: Diagrama C4 com a arquitetura do sistema proposto. 40

Figura 4.5: Diagrama do Módulo Luigui representando o processo de indexação, recuperação de *embeddings* dos *schemas* relevantes, contrução do *prompt*, geração da consulta e execução da consulta no banco de dados. 43

Figura 4.6: Diagrama de Entidade-Relacionamento do sistema proposto . . . 44

Figura 5.1: Diagrama do SimpleTextToSQLWorkflow que representa o fluxo utilizado para implementar da técnica de RAG para o caso onde o banco de dados cadastrado é do tipo *"minimal"*. 52

Figura 5.2: Diagrama do TextToSQLWorkflow que representa o fluxo utilizado para implementar da técnica de RAG para o caso onde o banco de dados cadastrado é do tipo *"complete"*. 55

Figura 5.3: Tela de cadastro de usuário do tipo "*employee*". 58

Figura 5.4: Tela de *login*. 58

Figura 5.5: Tela de *homepage* da aplicação onde é possível iniciar uma nova consulta. 59

Figura 5.6: Tela de Bancos de Dados na visão do "*admin*", onde é possível visualizar os bancos de dados cadastrados e também realizar novos cadastros. 60

Figura 5.7: Tela da etapa 1 do cadastro de banco de dados onde o usuário precisar informar o nome da conexão com o banco. 61

Figura 5.8: Tela da etapa 2 do cadastro de banco de dados onde o usuário precisar selecionar o tipo de banco a ser cadastrado. 61

Figura 5.9: Tela da etapa 3 do cadastro de banco de dados do tipo "*complete*" onde o usuário precisa informar os dados para realizar a conexão com o seu banco. 62

Figura 5.10: Tela da etapa 3 do cadastro de banco de dados do tipo "*minimal*" com as instruções de como o usuário pode extrair os esquemas das suas tabelas. 62

Figura 5.11: Tela de gerenciamento dos usuário que tem acesso a um banco de dados específico. 63

Figura 5.12: Tela que permite o cadastro de esquemas das tabelas e exibe quais as tabelas já tiveram seus esquemas armazenados na base de dados vetorial. 63

Figura 5.13: Tela que permite o cadastro de tabela em um banco de dados do tipo "*complete*". 63

Figura 5.14: *Homepage* exibindo lista de bancos de dados para disponíveis para seleção para iniciar o processo de perguntas. 65

Figura 5.15: Tela que permite a realização de perguntas pelo usuário com o tipo de pergunta selecionado como "Gerar Consulta". 66

Figura 5.16: Tela que permite a realização de perguntas pelo usuário com a opção de escolher *prompt* selecionada. A opções disponíveis para escolha são: "Gerar Consulta", "Otimizar Consulta", "Explicar Consulta" e "Corrigir Consulta". 66

Figura 5.17: Tela com a resposta em linguagem natural e a consulta SQL equivalente gerada para a pergunta "Quais alunos estão matriculados em Estruturas de Dados?". 67

Figura 5.18: Tela com a resposta em linguagem natural e a consulta SQL equivalente gerada para a pergunta "Qual é a média de nota da disciplina Algoritmos?". 67

Figura 5.19: Tela da opção "Otimizar Consulta" exibindo a consulta otimizada, além de uma explicação em linguagem natural da otimização feita. 68

Figura 5.20: Tela da opção "Explicar Consulta" exibindo uma explicação em linguagem natural da consulta enviada. 68

Figura 5.21: Tela da opção "Corrigir Consulta" exibindo a consulta corrigida e uma explicação em linguagem natural da correção feita. 69

Figura 5.22: Tela que exibe o histórico de perguntas e respostas para cada banco de dados do usuário. 69

Lista de Tabelas

Tabela 3.1: Definição da tabela `Clientes` 21

Tabela 3.2: Definição da tabela `Produtos` 21

Tabela 3.3: Definição da tabela `Pedidos` 21

Lista de Códigos

3.1 Consulta SQL para recuperar todos os pedidos de um cliente específico. 22

3.2 Consulta SQL para calcular a receita total de todos os pedidos. . . . 22

3.3 Consulta SQL para listar produtos comprados por um cliente específico. 23

3.4 Exemplo de esquema de uma tabela de Filmes. 27

3.5 Exemplo de prompt para Text-to-SQL contendo uma instrução clara sobre a geração da resposta em SQL, o esquema da tabela e a pergunta em linguagem natural. 28

5.1 Prompt de Text-to-SQL utilizado na TextToSQLPromptStrategy. . . 54

5.2 Exemplo de consulta SQL gerada para a pergunta "Quais alunos estão matriculados em Estruturas de Dados?". 64

5.3 Exemplo de consulta SQL gerada para a pergunta "Qual a média de nota para a disciplina de Algoritmos?". 64

Lista de Abreviaturas e Siglas

API Application Programming Interface

BD Banco de Dados

BDR Banco de Dados Relacional

BERT Bidirectional Encoder Representations from Transformers

GPT Generative Pre-trained Transformer

HTML HyperText Markup Language

HTTP Hypertext Transfer Protocol

IA Inteligência Artificial

JSON JavaScript Object Notation

LLM Large Language Model

LN Linguagem Natural

LSTM Long Short-term Memory

MVC Model-View-Controller

MVT Model-View-Template

ORM Object-Relational Mapping

PDF Portable Document Format

PLN Processamento de Linguagem Natural

RAG Retrieval-Augmented Generation

RNN Recurrent Neural Network

SQL Structured Query Language

XML Extensible Markup Language

Sumário

Agradecimentos **ii**

Resumo **vii**

Abstract **viii**

Lista de Figuras **ix**

Lista de Tabelas **xii**

Lista de Códigos **xiii**

Lista de Abreviaturas e Siglas **xiv**

1 Introdução **1**

1.1 Objetivo 2

1.2 Organização do Trabalho 3

2 Retrieval Augumented Generation **4**

2.1 Modelos de Linguagem 4

2.2 Large Language Models 9

2.3 Retrieval Augumented Generation 13
2.3.1 Tipos de Dados 14
2.3.2 Arquitetura de um sistema RAG 15
2.3.2.1 Indexação 16
2.3.2.2 Recuperação 17
2.3.2.3 Geração 18

3 Text-to-SQL **19**

3.1 Structured Query Language 19
3.1.1 Esquema Relacional 20
3.1.2 Consultas SQL 21
3.2 Linguagem Natural 23
3.3 Text-To-SQL 24
3.4 Estratégia de Text-to-SQL baseada em RAG 27
3.4.1 Pré-Processamento 28
3.4.1.1 Representação da Questão 28
3.4.1.2 Recuperação de Esquemas 29
3.4.1.3 Conhecimentos Adicionais 30
3.4.2 Inferência 30

4 Luigui: um Sistema Web para Text-To-SQL baseado em LLM utilizando a técnica de RAG **32**

4.1 Motivação 32
4.2 Trabalhos Relacionados 36

4.3 Proposta 38
4.3.1 Arquitetura do Sistema 40
4.3.1.1 Modulo Luigui 41
4.3.1.2 Banco de Dados Relacional e Vetorial 43
4.3.1.3 Luigui API 45

5 Implementação 47

5.1 Tecnologias Utilizadas 47
5.1.1 Django 48
5.1.2 PostgreSQL 48
5.1.3 PGVector 49
5.1.4 LlamaIndex 49
5.1.5 Next.js, Tailwind CSS e Shadcn/UI 50
5.2 Workflows 51
5.2.1 SimpleTextToSQLWorkflow 52
5.2.2 TextToSQLWorkflow 55
5.3 Interface e Funcionalidades 57
5.3.1 Cadastro e Autenticação de Usuários 57
5.3.2 Cadastro de Bancos de Dados 59
5.3.3 Cadastro de Esquemas 60
5.3.4 Produção de Consultas 62

6 Conclusão 70

6.1 Considerações finais 70

6.2 Limitações e trabalhos futuros 71

Referências **73**

Capítulo 1

Introdução

"Aqueles que não são capazes de sacrificar nada,
não são capazes de mudar nada."

— Armin Arlet

O uso de banco de dados relacionais consolidou-se como padrão na indústria da tecnologia, impulsionado pela crescente demanda por armazenar grandes volumes de dados e pela necessidade de eficiência na extração de informações (LIBKIN, 2003). Nesse contexto, a linguagem SQL destaca-se como o padrão universalmente adotado para o gerenciamento e a manipulação de dados estruturados (CHAMBERLIN, 2012).

Contudo, o uso de bancos de dados exige que o usuário possua conhecimentos técnicos em SQL para realizar consultas, o que restringe o acesso de pessoas não especializadas a dados potencialmente valiosos em suas rotinas profissionais. Essa limitação representa uma barreira significativa à democratização do acesso à informação (FARIHA et al., 2020), visto que nesse contexto o seu acesso está limitado a um grupo específico de pessoas.

Nesse cenário, os avanços recentes em inteligência artificial generativa, especialmente no campo dos Large Language Models (LLMs), têm se mostrado promissores para mitigar a barreira de entrada imposta pelo conhecimento técnico necessário ao uso de banco de dados relacionais (SHI et al., 2024). Esses modelos são capazes

de interpretar comandos em linguagem natural e gerar respostas coerentes em domínios especializados, como o de consultas em SQL, permitindo que usuários sem conhecimento técnico avançado possam interagir com bases de dados de maneira acessível.

Além disso, o interesse crescente da comunidade científica nessa área reforça o potencial transformador dos LLMs. Segundo Zhao et al. (2023), após o lançamento do ChatGPT[1], houve um crescimento expressivo na produção científica relacionada aos LLMs, com a média de artigos contendo o termo "*large language model*" no título ou resumo saltando de 0,40 para 8,58 por dia no repositório *arXiv*.

Diante desse cenário, surge a oportunidade de tornar o acesso a dados estruturados mais inclusivo e intuitivo, mesmo para usuários sem formação técnica. Uma das abordagens promissoras para esse problema é a técnica de Text-to-SQL, que consiste em traduzir perguntas formuladas em linguagem natural para comandos SQL executáveis. Esse processo tem o potencial de eliminar a necessidade de conhecimentos técnicos específicos, permitindo que profissionais de diferentes áreas consultem dados diretamente a partir de suas próprias perguntas.

O uso de LLMs representa um avanço significativo nesse contexto, pois esses modelos têm se mostrado altamente eficazes em tarefas de compreensão e geração de linguagem. No entanto, a combinação dos LLMs com a técnica de Retrieval-Augmented Generation (RAG) amplia ainda mais as possibilidades. A técnica de RAG permite que os modelos acessem fontes externas de informação, como detalhes de uma banco de dados, durante o processo de geração das consultas. Essa abordagem contribui para aumentar a precisão e a contextualização das respostas geradas, resultando em sistemas mais robustos e confiáveis para a tarefa de Text-to-SQL.

1.1 Objetivo

Este trabalho tem como objetivo propor um sistema *web* que permita a usuários sem conhecimentos técnicos acessar e manipular informações armazenadas em bancos

[1] <https://chatgpt.com/>

de dados por meio de perguntas formuladas em linguagem natural. A proposta busca empregar o poder dos LLMs, com foco na técnica de Retrieval-Augmented Generation (RAG), a fim de converter linguagem natural em consultas SQL.

Ao final, pretende-se desenvolver um sistema capaz de aproveitar o grande potencial dos LLMs e da arquitetura RAG, proporcionando uma interface acessível para usuários de diferentes níveis técnicos e áreas do conhecimento, facilitando o acesso eficiente e simplificado às informações presentes em seus bancos de dados.

1.2 Organização do Trabalho

Este trabalho está organizado da seguinte forma:

- **Capítulo 2:** Apresentam-se os fundamentos teóricos necessários para o entendimento da técnica de RAG. Inicia-se com os conceitos de modelos de linguagem, avançando até o surgimento dos LLMs e a utilização da técnica de RAG.

- **Capítulo 3:** São discutidos os conceitos teóricos complementares à proposta deste trabalho, com foco nos fundamentos da linguagem SQL e na tarefa de *Text-to-SQL*. Além disso, é discutido a utilização de LLMs em conjunto com a técnica de RAG para executar da tarefa de *Text-to-SQL*.

- **Capítulo 4:** Aborda-se a motivação para o sistema proposto, além da análise de trabalhos relacionados. Também são apresentadas as funcionalidades e a arquitetura da solução desenvolvida.

- **Capítulo 5:** Detalha-se o processo de desenvolvimento do sistema proposto, detalhando as tecnologias utilizadas e justificando sua escolha. Além disso apresentam-se os resultados obtidos.

- **Capítulo 6:** Apresentam-se as considerações finais, destacando as limitações do trabalho e apontando possíveis direções para trabalhos futuros.

Capítulo 2

Retrieval Augumented Generation

"É perigoso ir sozinho! Pegue isso."

— The Legend of Zelda

Neste capítulo, apresentamos os fundamentos teóricos e os principais conceitos necessários para compreender a aplicação da técnica de RAG no contexto da tarefa de converter Linguagem Natural (LN) em uma consulta SQL. Para fins de organização, o conteúdo foi dividido em três seções, de forma a oferecer uma abordagem progressiva e bem delimitada dos tópicos abordados.

Na Seção 2.1, discutimos a evolução dos modelos de linguagem até o surgimento dos *Large Language Models* (LLMs) e na Seção 2.2 destacamos seu impacto sobre os avanços em Processamento de Linguagem Natural (PLN). Em seguida, na Seção 2.3, exploramos a técnica de Retrieval-Augmented Generation (RAG), uma abordagem baseada em engenharia de *prompts* que visa aprimorar as respostas geradas por LLMs por meio da recuperação de informações externas relevantes.

2.1 Modelos de Linguagem

O PLN é uma das tarefas de maior relevância dentro da área de Inteligência Artificial (IA), nesse contexto encontram-se desafios como o reconhecimento, inter-

pretação, tradução e geração de textos por máquinas. Essa tarefa tem como base diversas técnicas de análise automática e de representação de linguagem humana (CAMBRIA; WHITE, 2014). Segundo um levantamento da plataforma *Spacelift*, em 2024, cerca de 16 milhões de textos por minuto foram enviados pela *internet*, incluindo e-mails, mensagens instantâneas e publicações em redes sociais [1]. Diante desse cenário de geração contínua e massiva de informações textuais, o PLN torna-se ainda mais relevante pois oferece os meios técnicos para compreender, organizar e extrair valor de grandes volumes de linguagem natural produzidos diariamente.

Apesar dessa urgência atual, a necessidade de desenvolver algoritmos e sistemas capazes de extrair, interpretar, e gerar textos em linguagem natural é um desafio de longa data dentro da área de IA (CAMBRIA; WHITE, 2014). Um dos principais desafios do PLN é modelar a predição de uma sequência futura de palavras, mais comumente chamadas de *tokens*. Assim, um *token* pode ser definido como uma unidade básica de dados que pode representar uma palavra, uma parte de uma palavra ou até mesmo um caractere (MANNING; SCHUTZE, 1999). Os modelos recebem como entrada uma sequência de palavras que são segmentadas em *tokens* e retornam como saída outros *tokens* preditos, que completam o texto com a maior probabilidade estatística (ZHAO et al., 2023).

Ao longo dos anos, foram propostos diversos modelos de linguagem para realizar as tarefas de PLN. Entre as primeiras soluções, destacaram-se os modelos estatísticos (MANNING; SCHUTZE, 1999), como aqueles baseados em cadeias de Markov, que realizavam a predição de palavras com base no contexto mais recente (JELINEK, 1998). Um exemplo clássico são os modelos n-grama, nos quais se considera um número fixo n de palavras anteriores para prever a próxima. Esses modelos representaram um avanço significativo, ao demonstrar que o contexto anterior é uma informação poderosa para a predição de palavras.

No entanto, nos modelos n-grama, à medida que se tenta aumentar o valor de n, ou seja, considerar contextos mais longos para capturar melhor as dependências linguísticas, surge uma limitação prática: o número de combinações possíveis de

[1] <https://spacelift.io/blog/how-much-data-is-generated-every-day>

palavras cresce exponencialmente, o que exige o armazenamento e cálculo de um volume muito grande de probabilidades. Essa explosão combinatória dificulta o uso eficiente dos chamados modelos de linguagem de alta ordem, com valores grandes de n, tornando seu uso inviável em muitas aplicações devido a limitações de memória e processamento (RAO; GUDIVADA, 2018).

Posteriormente, destacaram-se os modelos de linguagem baseados em Redes Neurais que deram o enfoque na probabilidade sequencial das palavras (CAMBRIA; WHITE, 2014). Nesse contexto, houve a emergência de uma ferramenta muito poderosa para o contexto dos modelos de linguagem, os *word embeddings* (BENGIO et al., 2003). Um *embedding* refere-se a uma forma de representar uma palavra por de uma meio representação vetorial densa, distribuída e de tamanho fixo. Esses vetores são capazes de armazenar informações semânticas e sintáticas sobre um texto. Além disso, dada a sua natureza vetorial, é possível realizar operações vetoriais úteis que auxiliam na extração de informações, como adição, subtração e medidas de distância (ALMEIDA; XEXÉO, 2019). Nesse sentido, o contexto em que uma palavra está inserido se mostrou extremamente importante para a predição de palavras.

Dentre os modelos de Redes Neurais, as Recurrent Neural Networks (RNNs), desenvolvidas por Sutskever, Vinyals e Le (2014), destacaram-se por apresentarem dois componentes principais: um codificador e um decodificador. O codificador é uma RNN responsável por processar uma sequência de símbolos de entrada, resultando em um vetor de tamanho fixo que armazena o valor semântico da sequência. Ao final do processo, esse vetor será decodificado por outra RNN, que gerará uma representação textual composta por uma nova sequência de símbolos, possivelmente de comprimento variável.

Nesse contexto, esses modelos implementam um aprendizado retroativo, no qual as RNN utilizam uma estrutura de repetição interna que permite a propagação de informações de uma etapa para a seguinte. Dessa forma, as RNN podem "lembrar-se" de dados anteriores para processar os atuais (ZHANG et al., 2023a). Esse comportamento é viabilizado por um componente chamado estado oculto, que atua como a memória da rede. A cada nova entrada, a RNN atualiza seu estado com base

na entrada atual e na anterior. Esse processo pode ser expresso por:

$$h_t = \sigma(W_{xh}x_t + W_{hh}h_{t-1} + b_h), \tag{2.1}$$

onde h_t representa o estado oculto atual da rede, isto é, a memória da rede naquele momento, x_t é a entrada atual da sequência, como um *token*, e h_{t-1} é o estado oculto da etapa anterior, que carrega informações da memória passada. As matrizes W_{xh} e W_{hh} correspondem aos pesos aprendidos durante o treinamento, responsáveis por ponderar a influência da entrada e da memória anterior, respectivamente. O termo b_h representa o viés adicionado ao cálculo, enquanto σ é uma função de ativação não linear, responsável por introduzir não linearidades ao modelo e permitir uma maior capacidade de representação.

Essa característica das RNNs possibilitou o reconhecimento de relações entre palavras em sequências de curto prazo. No entanto, esses modelos não eram eficazes na captura de dependências de longo alcance, pois o contexto tendia a se deteriorar ao longo das iterações (RAO; GUDIVADA, 2018).

As redes Long Short-term Memory (LSTM) surgiram com o intuito de minimizar esse problema de memória das RNNs (SUTSKEVER; VINYALS; LE, 2014). Esses modelos correspondem a redes neurais profundas capazes de lidar com grandes quantidades de dados. Isso permitiu com que os modelos pudessem lidar com uma memória de longo prazo. Contudo, esses modelos ainda eram limitados para sequências ainda maiores, além de possuírem um processamento lento devido a sua arquitetura não paralelizada (VASWANI, 2017).

Sob essa ótica, a chegada da arquitetura Transformer revolucionou a área de modelos de linguagem, pois ela tinha o foco em reduzir o uso de recorrência e reduzir consideravelmente não só o tempo de processamento, mas o problema da memória para contextos muito grandes (VASWANI, 2017). Essa arquitetura foi proposta para ser mais paralelizável, o que mitigou o problema de processamento que era enfrentado pela maioria dos modelos anteriores. Além disso, ela introduziu os mecanismos de atenção como parte crucial do seu modelo.

Com isso, Vaswani (2017) propôs que o Transformer deveria ser uma rede baseada unicamente em mecanismos de atenção, esses mecanismos são uma estrutura que permite identificar as dependências entre as palavras do texto independentemente da sua distância. Isso permite com que o texto todo seja visto como uma estrutura global que influenciará na predição da próxima palavra. O Transformer foi o primeiro modelo a utilizar unicamente a auto-atenção para o cálculo da representação vetorial das entradas e saídas da rede. Esse mecanismo permite relacionar diferentes posições de palavras de uma mesma sequência para calcular a representação vetorial dessa sequência.

Após a arquitetura Transformer ser proposta por Vaswani (2017), em 2018 a OpenAI lançou o seu primeiro modelo grande de linguagem modelo baseado em Transformer, o GPT-1 (RADFORD et al., 2018). Esse modelo introduziu o conceito de Generative Pre-trained Transformer (GPT), ou seja esses modelos são pré-treinados em um conjunto diversificado de dados com o objetivo de aprender os padrões semânticos e sintáticos de uma linguagem e assim gerar novos textos. Esse passo foi um marco para a criação e desenvolvimento dos Large Language Models (LLMs).

O GPT-1 utilizou a arquitetura Transformer em um modelo de 12 camadas com 117 milhões de parâmetros, sendo treinado em um corpus de livros de ficção não publicados, conhecido como BookCorpus (RADFORD et al., 2018). Esse treinamento permitiu ao modelo capturar relações complexas de linguagem, superando modelos anteriores em diversas tarefas de PLN.

Em 2019, a OpenAI apresentou o GPT-2, uma expansão direta do GPT-1, com 1,5 bilhão de parâmetros (RADFORD et al., 2019). Treinado em um conjunto de dados chamado WebText, composto por 8 milhões de páginas da *web*, o GPT-2 demonstrou capacidades impressionantes na geração de texto coerente e contextualmente relevante, mesmo em tarefas para as quais não foi explicitamente treinado.

O avanço mais notável ocorreu em 2020 com o lançamento do GPT-3, que possui 175 bilhões de parâmetros (BROWN et al., 2020). Este modelo apresentou desempenho excepcional em tarefas de aprendizado com poucos exemplos, sendo

capaz de realizar traduções, responder perguntas e gerar textos com qualidade comparável à escrita humana, mesmo sem ajustes específicos para cada tarefa.

Além dos modelos da série GPT, outros trabalhos significativos contribuíram para o avanço dos LLMs. O Bidirectional Encoder Representations from Transformers (BERT), introduzido pelo Google em 2018, destacou-se por seu treinamento bidirecional, permitindo uma compreensão mais profunda do contexto em tarefas como resposta a perguntas e inferência textual (DEVLIN et al., 2019). Esses desenvolvimentos consolidaram os LLMs como ferramentas essenciais no campo do PLN, impulsionando pesquisas e aplicações em diversas áreas da inteligência artificial (ZHAO et al., 2023).

2.2 Large Language Models

Os Large Language Models (LLMs) são modelos de linguagem baseados em redes neurais profundas com bilhões de parâmetros que foram treinados com um volume massivo de dados (ZHAO et al., 2023). A principal tarefa desses modelos é compreender textos em linguagem natural e realizar geração de textos que completem de maneira precisa o que foi solicitado. Os LLMs possuem a capacidade de entender os padrões de construção de uma linguagem e utilizá-los para gerar novos dados textuais.

Segundo Shi et al. (2024), a tarefa de predição de um LLM pode ser descrita pela equação 2.2. Seja y_t o próximo *token* a ser predito na posição t da sequência, y_t é o *token* que maximiza a probabilidade condicional adicionando-o à sequência.

$$y_t = \arg\max P(y_t \mid y_{1:t-1}, \mathbf{x}) \tag{2.2}$$

O lado direito da equação representa a probabilidade condicional de que o *token* y_t seja o próximo elemento da sequência, dado o contexto anterior. Esse contexto é formado por dois componentes principais: a sequência de *tokens* já gerados até a posição $t-1$, representada por $y_{1:t-1}$, e uma entrada opcional $\mathbf{x}$, que pode incluir

informações adicionais como um *prompt*.

Um *prompt* é uma instrução ou entrada textual fornecida ao modelo com o objetivo de orientá-lo na execução de uma tarefa específica. Ele pode variar desde uma simples frase até exemplos mais elaborados, servindo como guia para que o modelo compreenda o contexto e produza uma resposta adequada. O modelo, então, calcula a probabilidade de todos os possíveis *tokens* que podem vir a seguir, e seleciona aquele que possui a maior probabilidade, ou seja, o mais provável de continuar corretamente a sequência.

Nesse contexto, os pesquisadores perceberam que aumentar a rede, ou a quantidade de dados dos modelos pré-treinados, na maior parte dos casos, permite uma melhoria na capacidade do modelo de obedecer as tarefas solicitadas pelo usuário (ZHAO et al., 2023). Esses modelos são treinados com representações de dados sensíveis ao contexto, o que permite com que eles sigam instruções de tarefas em diversos contextos. Eles podem ser considerados como solucionadores de tarefas de propósito geral (BROWN et al., 2020).

Antes do surgimento dos LLMs, a prática comum no PLN era treinar modelos específicos para cada tarefa. Isso significava desenvolver um modelo para classificar sentimentos, outro para traduzir textos, outro ainda para responder perguntas, entre diversas outras aplicações (DEVLIN et al., 2019). Cada novo problema exigia dados rotulados, ajustes específicos e um tempo considerável de desenvolvimento, tornando o processo caro e pouco reutilizável. Em outras palavras, era necessário criar ou adaptar um modelo praticamente do zero para cada aplicação.

A chegada dos LLMs representou uma mudança de paradigma, sobretudo com a consolidação do uso de *prompting*, a ideia de descrever a tarefa desejada diretamente como um texto (CHANG et al., 2024). Em vez de treinar um novo modelo para cada tarefa, passou-se a utilizar um único modelo genérico, previamente treinado com grandes volumes de dados, capaz de realizar diferentes tarefas apenas variando a instrução fornecida em linguagem natural (BROWN et al., 2020). Esse processo é ilustrado na Figura 2.1, que compara a geração de respostas entre um modelo de linguagem treinado especificamente para tradução e um LLM. No exemplo, o

primeiro modelo foi treinado exclusivamente para traduzir textos do inglês para o português, enquanto o LLM realiza a mesma tarefa apenas com base na instrução contida no *prompt*, mesmo sem ter sido treinado de forma supervisionada para isso. Com isso, tornou-se possível resolver uma ampla variedade de problemas com um único modelo, bastando descrever, em linguagem natural, o que se deseja que ele faça.

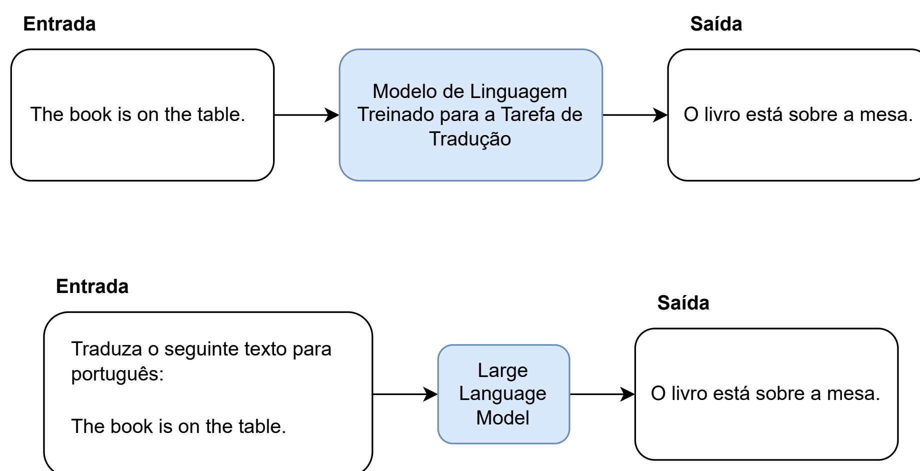

Figura 2.1: Representação esquemática comparativa de um modelo de linguagem treinado para a tarefa de tradução e um LLM que recebe como entrada um *prompt* com a instrução de traduzir um texto.

Vale destacar, no entanto, que o uso de *prompts* não surgiu com os LLMs. Já em modelos anteriores, especialmente voltados à modelagem de linguagem, era comum fornecer uma sequência de entrada para que o modelo previsse a próxima palavra (MANNING; SCHUTZE, 1999). Esse uso, contudo, era bastante limitado: os modelos não compreendiam instruções complexas, e o texto de entrada funcionava mais como um mecanismo auxiliar do que como um meio eficaz de controle de tarefas. Além disso, cada tarefa ainda demandava o treinamento ou o ajuste de um modelo próprio (RAO; GUDIVADA, 2018).

A virada mais significativa ocorreu com modelos como o GPT-3, lançado em 2020 pela OpenAI (BROWN et al., 2020). Ao escalar significativamente o número de parâmetros e ao treiná-lo com dados diversos e em larga escala, tornou-se possível obter respostas coerentes a instruções em linguagem natural, muitas vezes sem

qualquer ajuste adicional. Essa capacidade transformou os *prompts* de uma técnica secundária em uma interface central de interação com o modelo, consolidando o *prompting* como uma das inovações mais impactantes no uso de LLMs.

Além da mudança no modo de interação, os LLMs também passaram a exibir comportamentos ausentes em versões menores e que não podiam ser previstos por simples extrapolação de desempenho. Essas chamadas habilidades emergentes foram descritas por Wei et al. (2022), que caracterizam o fenômeno como a aparição de capacidades, como raciocínio lógico, resolução de problemas matemáticos ou compreensão de instruções complexas, que só se manifestam a partir de certo nível de escala. A existência dessas habilidades emergentes levanta questões relevantes sobre os limites e o potencial do escalonamento contínuo desses modelos, sugerindo que novos comportamentos e competências podem surgir à medida que a escala aumenta.

Nesse contexto de avanços significativos, uma das aplicações que mais se destacou nos últimos anos foi o ChatGPT[2]. Essa ferramenta consiste em uma versão do GPT adaptada para conversação com humanos, permitindo interações mais naturais e acessíveis. O ChatGPT exemplifica como os LLMs podem ser integrados em aplicações práticas, facilitando o acesso de usuários comuns a tecnologias de ponta em processamento de linguagem natural.

Entretanto, apesar desses avanços, os LLMs ainda representam limitações, como a sua restrição a conhecimentos que estavam nos presentes dados até a data do treinamento, o que significa que que o seu conhecimento é estático. Nesse sentido, esses modelos não possuem realmente uma consciência sobre o que eles geram, na verdade, segundo Bender et al. (2021) os modelos estão "probabilisticamente vinculando palavras e frases sem considerar o significado". Assim, caso o usuário realize uma pergunta sobre informações com a qual o LLM nunca foi apresentado, como os dados privados do banco de dados de sua empresa, ele provavelmente irá gerar alucinações, isto é, respostas imprecisas ou fora de contexto (ZHANG et al., 2023b).

Alguns LLMs possuem a capacidade de ser integrados a sistemas que permi-

[2]<https://chatgpt.com/>

tem a busca de informações na *internet* (ZIEMS et al., 2023). Contudo, esses modelos ainda enfretam dificuldades para acessar dados que não estavam presentes em seu treinamento, especialmente se essas informações não estiverem disponíveis publicamente.

Sob essa ótica, uma aplicação promissora para os LLMs é a tarefa de Text-to-SQL, em que perguntas em linguagem natural são convertidas em consultas SQL. No entanto, quando o domínio envolve bases de dados empresariais específicas, cujas informações não estiveram presentes no pré-treinamento do modelo, torna-se necessário enriquecer o contexto com dados externos para evitar respostas imprecisas. Para isso, recorreremos à técnica de Retrieval-Augmented Generation (RAG), que combina recuperação de informações com geração de texto, garantindo consultas mais precisas e alinhadas ao contexto corporativo.

2.3 Retrieval Augumented Generation

Os LLMs demonstram grande capacidade de geração textual, mas mantêm-se estáticos após o pré-treinamento, sem incorporar novos dados ou contextos específicos de um domínio (BENDER et al., 2021; ZHAO et al., 2023). A técnica de Retrieval-Augmented Generation (RAG) surge para superar essa limitação, unindo dois estágios complementares: (i) recuperação de trechos relevantes em uma base externa e (ii) geração de texto condicionada a esse material recuperado (GAO et al., 2023).

No primeiro estágio, utiliza-se um índice ou repositório, que pode ser um banco de dados relacional, um conjunto de documentos ou até mesmo páginas web, para buscar conteúdos que atendam à consulta de entrada. Esses trechos são então incorporados ao *prompt* do LLM, ampliando seu contexto de geração. No segundo estágio, o modelo gera a resposta final apoiado nas informações recém-recuperadas, para isso é construído um *prompt* com as informações recuperadas e a instrução da tarefa a ser realizada.

Esse fluxo de recuperação e geração reduz drasticamente o risco de alucinações típicas dos LLMs (ZHANG et al., 2023b), pois a base externa fornece evidências

factuais no momento da geração. Em cenários de alto risco, como finanças ou saúde, onde decisões críticas dependem da exatidão dos dados, a RAG reduz a probabilidade de que informações não verificadas entrem no processo de resposta (GAO et al., 2023).

Segundo Gao et al. (2023), a eficácia do sistema depende de um gerenciamento rigoroso dos dados. Isso inclui processos para assegurar que os dados sejam claros, bem estruturados e governados. Sem isso, o modelo pode introduzir erros ou gerar respostas baseadas em dados desatualizados ou irrelevantes. Nesse contexto, é extremamente importante analisar quais os tipos de dados que serão utilizados na construção de um sistema de RAG.

2.3.1 Tipos de Dados

Um dos principais fatores a se considerar durante a criação de uma aplicação de RAG é qual o tipo de dado que será suportado (GAO et al., 2023). Os dados podem ser classificados em três categorias principais: estruturados, não estruturados e semi-estruturados (CHENG et al., 2025). Cada tipo possui características específicas que determinam como serão utilizados nos processos de recuperação e geração.

- **Dados Estruturados**: São aqueles organizados de forma rigorosa, ou seja, os dados possuem um formato fixo e previamente definido, como as tabelas em Banco de Dados Relacionals (BDRs). Exemplos incluem dados de vendas, listas de produtos, registros financeiros e informações armazenadas em sistemas de gerenciamento como SQL. Esses dados são facilmente acessados e interpretados por sistemas de busca, graças à sua organização e padronização.

- **Dados Não Estruturados**: Representam a maior parte das informações disponíveis na *internet*[3] e não seguem um formato ou estrutura rígida. Exemplos incluem textos em documentos, *e-mails*, artigos, imagens, vídeos e gravações de áudio. Por serem mais desorganizados, sua utilização em RAG exige a aplicação de técnicas avançadas para extração de *embeddings* e métodos de

[3] <https://blog.box.com/90-your-data-unstructured-and-its-full-untapped-value>

busca semântica, para torná-los compreensíveis e úteis no fornecimento de contexto aos LLMs.

- **Dados Semi-Estruturados**: Esses dados ocupam uma posição intermediária entre os estruturados e não estruturados. Eles possuem alguma organização, mas não seguem um esquema rigoroso. Exemplos incluem arquivos em formato JavaScript Object Notation (JSON), Extensible Markup Language (XML) ou dados provenientes de planilhas com informações parcialmente categorizadas. Por apresentarem uma estrutura flexível, os dados semi-estruturados são ideais para integração em sistemas de RAG, pois podem ser adaptados para fornecer informações relevantes sem a necessidade de transformações complexas.

Cada tipo de dado traz desafios e oportunidades únicos para a aplicação de RAG (GAO et al., 2023). Enquanto os dados estruturados são mais diretos e eficientes para consultas, os dados não estruturados permitem explorar um volume muito maior de informações, visto que representam a maior parte dos dados disponíveis para serem utilizados. Já os semi-estruturados oferecem um equilíbrio entre flexibilidade e organização, sendo amplamente utilizados em soluções práticas. A capacidade de lidar com essas três categorias torna a RAG uma ferramenta versátil e poderosa para aplicações baseadas em LLMs.

Sob essa ótica, a RAG introduz um método dinâmico para armazenar, processar e utilizar informações, expandindo significativamente as possibilidades de aplicação prática dos LLMs em áreas onde a precisão e a atualização são cruciais para um resultado satisfatório. Na próxima subseção, analisamos uma arquitetura proposta para esse processo de recuperação e geração de informações.

2.3.2 Arquitetura de um sistema RAG

O funcionamento básico de um sistema RAG tem início quando um usuário deseja realizar uma pergunta. Suponha, por exemplo, que uma empresa possua milhares de páginas de documentação interna. Caso o sistema esteja alimentado com essas informações, ao receber uma pergunta específica sobre o funcionamento de

algum processo da empresa, ele iniciará uma busca pelos trechos mais relevantes e relacionados à pergunta formulada. Em seguida, o sistema construirá um *prompt* que combina os trechos recuperados com a pergunta do usuário. A partir disso, o LLM gerará uma resposta baseada nesse conteúdo, o que reduz a chance de erro e aumenta a precisão e a confiabilidade da resposta (GAO et al., 2023).

Dentro desse panorama, diversas arquiteturas têm sido propostas para a implementação da técnica de RAG. Uma das abordagens mais conhecidas é a *Naive RAG* (GAO et al., 2023), amplamente discutida na literatura e adotada como base para a proposta deste trabalho.

A *Naive RAG* representa uma abordagem inicial que ganhou destaque um pouco depois da popularização do *ChatGPT* e de modelos de linguagem similares (GAO et al., 2023). Esse método se baseia em um processo tradicional dividido em três etapas principais: indexação, recuperação e geração, como demonstrado na figura 2.2. A estrutura central da *Naive* RAG é frequentemente descrita como um modelo de *Retrieve-Read*, isto é, "Recuperar-Ler".

Nessa abordagem, uma base de dados é primeiramente indexada, ou seja, organizada de maneira a facilitar a busca eficiente por informações relevantes. Em seguida, quando uma consulta é realizada, o sistema recupera os dados mais pertinentes por meio de técnicas de busca, como vetorização de texto e medidas de similaridade. O conteúdo obtido é então utilizado como contexto para que o LLM gere uma resposta, combinando a precisão das informações recuperadas com a capacidade linguística do modelo. Essa simplicidade tornou o *Naive* RAG uma base fundamental para o desenvolvimento de abordagens mais sofisticadas em IA. A seguir, analisamos com mais detalhes os processos de indexação, recuperação e geração presentes no *Naive* RAG, conforme descrito por Gao et al. (2023).

2.3.2.1 Indexação

A etapa de indexação na *Naive* RAG inicia-se com a limpeza e extração de dados brutos em vários formatos, como *Portable Document Format (PDF), HyperText Markup Language (HTML), Word, Markdown*, entre outros. Esses dados são conver-

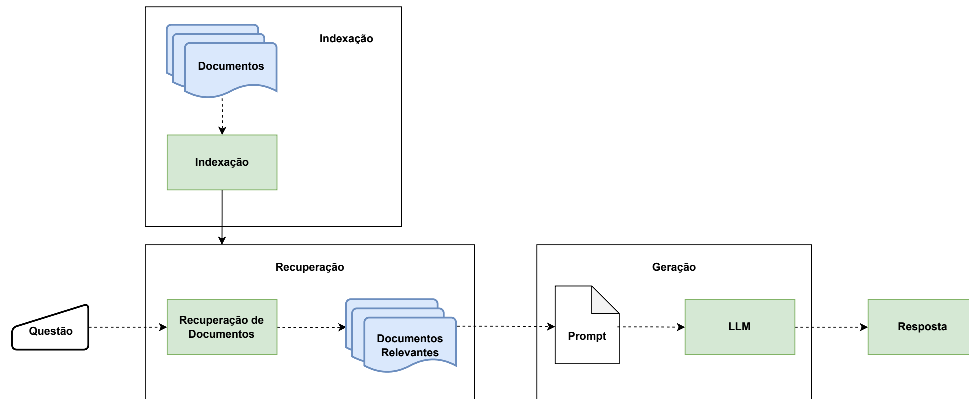

Figura 2.2: Arquitetura *Naive RAG*.

tidos para um formato uniforme de texto simples, garantindo consistência para o processamento seguinte. Dada a limitação de contexto dos modelos de linguagem, o texto é segmentado em pequenos blocos ou fragmentos, conhecidos como *chunks*, que são mais fáceis de gerenciar e interpretar.

Logo após o dados serem transformados em *chunks*, podemos adicionar mais contexto a ele com metadados, como o número da página, nome do documento, autor, entre outros, a depender do tipo de documento que está sendo utilizado. Isso permite com que possamos realizar uma busca baseada na filtragem desses metadados, o que pode garantir maior precisão na recuperação de informações.

Então, esses fragmentos são então codificados em representações vetoriais por meio de um modelo de *embeddings*, que transforma as informações textuais em vetores numéricos. Esses vetores são armazenados em um banco de dados vetorial, um componente essencial para permitir buscas de similaridade de forma eficiente na fase de recuperação. Esse processo de indexação é importante para garantir que o sistema possa identificar e recuperar os trechos mais relevantes em consultas futuras.

2.3.2.2 Recuperação

A etapa de recuperação na *Naive* RAG começa quando o sistema recebe uma consulta do usuário. Nesse momento, utiliza-se o mesmo modelo de *embeddings* empregado na fase de indexação para transformar a consulta em uma representação vetorial. Em seguida, calcula-se a similaridade entre o vetor da consulta e os vetores

dos *chunks* previamente indexados na base de dados vetorial.

Com base nos escores de similaridade, o sistema prioriza e recupera os K principais *chunks* que apresentam maior correspondência com a consulta. Esses *chunks* selecionados são então incorporados como contexto expandido no *prompt* utilizado pelo modelo de geração, garantindo que a resposta gerada seja fundamentada nas informações mais relevantes disponíveis no Banco de Dados (BD).

2.3.2.3 Geração

A etapa de geração na *Naive* RAG combina a consulta do usuário com os documentos selecionados, criando um *prompt* coeso que será processado por um LLM, assim, o modelo então formula uma resposta que pode variar em abordagem dependendo dos critérios de cada tarefa.

Ele pode tanto basear-se em seu conhecimento prévio quanto restringir sua resposta às informações contidas nos documentos fornecidos. Em cenários de diálogo contínuo, o histórico da conversa também pode ser integrado ao *prompt*, permitindo que o modelo participe de interações de múltiplos turnos de maneira eficaz e contextualizada.

Essa abordagem representa um avanço significativo no uso de LLMs, permitindo aplicações mais eficientes e dinâmicas em domínios que exigem informações atualizadas e respostas contextualizadas. Sob essa ótica, a RAG configura-se como uma ferramenta altamente promissora para o domínio de Text-to-SQL. No próximo capítulo, discutiremos como essa técnica pode ser aplicada para aprimorar o processo de conversão de linguagem natural em consultas SQL.

Capítulo 3

Text-to-SQL

"Transmutação é a lei da troca equivalente."

— Fullmetal Alchemist

Neste capítulo, damos continuidade à apresentação dos fundamentos teóricos necessários para o entendimento da solução proposta neste trabalho. Na Seção 3.1, abordaremos os conceitos e definições básicas sobre a linguagem SQL. Em contraste, na Seção 3.2, discutiremos sobre linguagem natural utilizada pelos seres humanos. Em seguida, na Seção 3.3, apresentaremos a tarefa de Text-to-SQL, que consiste no mapeamento de sentenças em linguagem natural para consultas em SQL. Por fim, exploraremos como os LLMs podem ser utilizado para realizar a tarefa de Text-to-SQL e analisaremos como como a técnica de RAG pode ser aplicada para aprimorar esse processo.

3.1 Structured Query Language

A Structured Query Language (SQL) é a linguagem de programação padrão utilizada para o gerenciamento e manipulação de BDRs (ELMASRI et al., 2005). BDRs são sistemas de armazenamento de dados que possuem relacionamentos entre si por meio de chaves primárias e estrangeiras (ELMASRI et al., 2005); estes, por sua vez, são, respectivamente, identificadores internos de cada registro nas tabelas e

identificadores "externos" que fazem referência à chave primária de outra tabela dos registros do BD.

A partir de dados estruturados em tabelas, e relações formadas entre chaves primárias e estrangeiras, a SQL permite a realização de consultas estruturadas e diversas operações em bases de dados, como adição, exclusão e atualização de dados, além da recuperação eficiente de informações (LIBKIN, 2003). Essas operações são essenciais quando se trabalha com um grande volume de dados.

Além disso, a linguagem também oferece recursos para manipulação e gerenciamento de BDs, incluindo a criação de tabelas e o estabelecimento de relacionamentos entre as informações, falaremos mais desse último ponto nos próximos parágrafos. Nesse contexto, é possível organizar e estruturar os objetos do banco de dados utilizando esquemas, que funcionam como contêineres lógicos. Os esquemas permitem agrupar tabelas, *views*, que são tabelas virtuais baseadas em consultas, índices, procedimentos armazenados e outros objetos relacionados, facilitando a administração e a organização de BDs (LIBKIN, 2003).

3.1.1 Esquema Relacional

Um esquema relacional define a estrutura lógica de um banco de dados no modelo relacional, especificando as tabelas, também chamadas de relações, seus atributos, também chamados de colunas, e as restrições de integridade que se aplicam, como chaves primárias e estrangeiras (ELMASRI et al., 2005). Cada tabela corresponde a uma relação matemática cujos elementos são chamados de tuplas, ou linhas, que armazenam instâncias de dados. As chaves primárias garantem a unicidade de cada tupla em uma relação, enquanto as chaves estrangeiras estabelecem vínculos entre tabelas, permitindo representarmos associações entre diferentes entidades.

A fim didático, consideremos o esquema relacional de uma loja online, cujas tabelas principais são apresentadas nas Tabelas 3.1, 3.2 e 3.3. Essas tabelas estão interligadas por meio de chaves primárias (PK) e estrangeiras (FK), permitindo organizar, relacionar e recuperar informações de forma eficiente.

Clientes	Tipo	Descrição
id_cliente	INT (PK)	Identificador único do cliente
nome	VARCHAR(100)	Nome do cliente
email	VARCHAR(150)	E-mail do cliente

Tabela 3.1: Definição da tabela `Clientes`

Produtos	Tipo	Descrição
id_produto	INT (PK)	Identificador único do produto
nome_produto	VARCHAR(100)	Nome do produto
preco	DECIMAL(10,2)	Preço do produto

Tabela 3.2: Definição da tabela `Produtos`

Pedidos	Tipo	Descrição
id_pedido	INT (PK)	Identificador único do pedido
id_cliente	INT (FK)	Chave estrangeira para `Clientes.id_cliente`
data_pedido	DATE	Data em que o pedido foi realizado

Tabela 3.3: Definição da tabela `Pedidos`

Essas relações entre tabelas, por exemplo, a que liga `Pedidos.id_cliente` à `Clientes.id_cliente`, garantem a integridade referencial do banco de dados e fornecem o alicerce para consultas que combinem informações de múltiplas entidades.

3.1.2 Consultas SQL

Com o esquema relacional estabelecido, é possível formular consultas SQL que exploram essas interconexões para extrair informações úteis (ELMASRI et al., 2005). A consulta do Código 3.1 ilustra a importância das relações estabelecidas por meio de chaves estrangeiras para unir informações distribuídas em diferentes tabelas. Ao relacionar a tabela `Pedidos` com a tabela `Clientes`, a consulta garante que cada pedido seja associado ao cliente correto, assegurando a integridade dos dados e garantindo a precisão das informações. Essa integração é fundamental para a manutenção da consistência dos registros, além de facilitar futuras operações de inserção ou atualização, onde a integridade referencial pode ser preservada.

Já a consulta do Código 3.2 demonstra a relevância dos relacionamentos entre as tabelas para a realização de operações agregadas. Através da junção das tabelas

Pedidos, ItensPedido e Produtos, é possível consolidar dados financeiros dispersos e somar de forma eficiente. Essa integração permite que o sistema extraia e sintetize informações críticas para possíveis análises financeiras.

Por fim, na consulta do Código 3.3, podemos observar a importância de múltiplos relacionamentos entre tabelas para a consolidação de informações detalhadas. Ao fazer a junção entre Clientes, Pedidos, ItensPedido e Produtos, a consulta possibilita a agregação de dados de diferentes fontes, fornecendo uma visão ampla do comportamento de compra e interesses do cliente. Essa abordagem permite não só a recuperação de informações relevantes para análises do perfil do usuário e personalização de ofertas, como também permite que, caso novas informações sejam inseridas nas tabelas, as projeções do comportamento do cliente sejam preservadas.

Esses exemplos evidenciam o poder e a flexibilidade da SQL para manipular e recuperar grandes volumes de dados de forma estruturada e eficiente (LIBKIN, 2003). Contudo, escrever consultas complexas exige familiaridade com a sintaxe da linguagem, conhecimento do esquema de dados e entendimento das operações de junção e agregação. Em ambientes corporativos, nem sempre os usuários que necessitam dessas informações possuem esse perfil técnico, o que pode se tornar um gargalo na extração de dados (FARIHA et al., 2020).

Diante dessa necessidade de tornar a consulta de dados acessível a usuários leigos, surge a motivação de desenvolver aplicações que sejam capazes de transformar linguagem natural em consultas SQL. Nas próximas seções, definiremos o que é linguagem natural e como ela pode ser utilizada para permitir pessoas que não possuem conhecimento em SQL possam formular perguntas no seu idioma nativo.

```
1 SELECT p.id_pedido, p.data_pedido
2 FROM Pedidos p
3 JOIN Clientes c ON p.id_cliente = c.id_cliente;
```

Código 3.1: Consulta SQL para recuperar todos os pedidos de um cliente específico.

```
1 SELECT SUM(pr.preco) AS receita_total
2 FROM Pedidos p
3 JOIN ItensPedido i ON p.id_pedido = i.id_pedido
```

```
4 JOIN Produtos pr ON i.id_produto = pr.id_produto;
```

Código 3.2: Consulta SQL para calcular a receita total de todos os pedidos.

```
1 SELECT pr.nome_produto, pr.preco
2 FROM Pedidos p
3 JOIN Clientes c ON p.id_cliente = c.id_cliente
4 JOIN ItensPedido i ON p.id_pedido = i.id_pedido
5 JOIN Produtos pr ON i.id_produto = pr.id_produto
6 WHERE c.nome = "Giovanni Moura";
```

Código 3.3: Consulta SQL para listar produtos comprados por um cliente específico.

3.2 Linguagem Natural

Desde os primórdios da humanidade, o ser humano desenvolveu a capacidade de pensar, construir, raciocinar, compreender, imaginar, perceber, provar e manipular o mundo ao seu redor. Dentre os muitos desdobramentos dessas habilidades, destacam-se expressões como a poesia, a música e a arte (GOMES, 2010). No entanto, tais manifestações não teriam sido possíveis, tampouco preservadas, sem a habilidade de se comunicar. A comunicação permite algo fundamental: a transmissão de conhecimento e a preservação histórica da memória de um povo, bem como de seus avanços culturais e tecnológicos. Nesse contexto, dentre as múltiplas formas de comunicação humana, destaca-se a linguagem natural, elemento central tanto para a humanidade quanto para os objetivos deste trabalho.

Define-se por linguagem natural o conjunto de idiomas usados nas interações cotidianas entre seres humanos, permitindo que se comuniquem de maneira compreensível, frequentemente sem a necessidade de normas formais durante o diálogo. Línguas como o português, o inglês, o japonês, o italiano e o mandarim são exemplos de linguagem natural. Em contraste com linguagens artificiais, como linguagens de programação e notações matemáticas, as linguagens naturais evoluem ao longo do tempo, à medida que são transmitidas entre gerações e influenciadas por contextos

culturais e sociais. Essa natureza dinâmica torna difícil definir regras formais e explícitas para seu funcionamento (BIRD; KLEIN; LOPER, 2009).

Agora que definimos o que é linguagem natural, observa-se que ela é tão antiga e comum à humanidade quanto o próprio pensamento. Com o avanço tecnológico alcançado nas últimas décadas — especialmente o aumento da capacidade computacional e as inovações no campo da IA, tornou-se possível o reconhecimento e o processamento de linguagem natural por máquinas. Nesse sentido, como já visto anteriormente, o Processamento de Linguagem Natural (PLN) refere-se ao conjunto de técnicas que permitem a uma máquina interagir com seres humanos por meio de computação linguística e ferramentas que operam em linguagem natural (MOREIRA, 2021).

Sob essa ótica, no ambiente corporativo, muitos profissionais, inclusive em cargos de gerência ou em áreas que se relacionam com tecnologia, mas sem perfil técnico aprofundado, não dispõem de conhecimento para escrever consultas complexas em SQL ou manipular grandes volumes de dados de forma precisa. Nesse cenário, a linguagem natural torna-se especialmente relevante, pois viabiliza a criação de interfaces acessíveis a usuários leigos. Ao converter a linguagem natural utilizada por esses usuários em consultas ou processos automáticos de recuperação de informações, é possível aumentar a eficiência, agilizar fluxos de trabalho e aliviar a carga de tarefas repetitivas que, de outra forma, recairiam sobre os desenvolvedores (KIM et al., 2020).

3.3 Text-To-SQL

Dado que grande parte das companhias e empresas de diversas áreas e tamanhos possui uma infinidade de dados armazenados em BDRs, a necessidade de buscar dados e informações nesses repositórios se tornou uma demanda cada vez mais frequente nos últimos anos (SHI et al., 2024). Os BDRs tornaram-se essenciais para a manutenção e o armazenamento eficiente de informações (ELMASRI et al., 2005).

No entanto, a interação com esses sistemas geralmente exige conhecimento técnico

em SQL para realizar buscas e recuperar dados (FARIHA et al., 2020). Esse tipo de tarefa pode não ser trivial para usuários que não pertencem à área de tecnologia e que, muitas vezes, não estão familiarizados com o universo da manipulação e manutenção de dados, embora necessitem explorá-los para cumprir suas atividades (KIM et al., 2020). Assim, o acesso à informação acaba por se restringir a usuários com conhecimento técnico avançado, como os desenvolvedores.

Nesse cenário, destaca-se a tarefa de converter texto em linguagem natural em consultas SQL, conhecida como Text-to-SQL, a qual possui fundamentos consolidados dentro da área de PLN (SHI et al., 2024). Essa tarefa consiste, essencialmente, em transformar instruções escritas em linguagem natural em consultas estruturadas em SQL, o que pode permitir com que usuários leigos acessem e utilizem bancos de dados relacionais para extrair informações de forma relativamente simples e acessível.

Por exemplo, considere a seguinte tabela denominada `Funcionarios`, com as colunas `id`, `nome`, `departamento`, `salario` e `data_admissao`. Dada a pergunta em linguagem natural "Qual é o funcionário com o maior salário?", a consulta SQL correspondente seria “`SELECT nome FROM Funcionarios WHERE salario = (SELECT MAX(salario) FROM Funcionarios);`”. Esse exemplo ilustra como a Text-to-SQL pode servir de ponte entre a linguagem humana e a linguagem formal de bancos de dados, democratizando o acesso à informação por meio da linguagem natural.

O mapeamento de linguagem natural para consultas SQL não é uma abordagem recente. Técnicas clássicas, como o formato *template-based* desenvolvido por Zelle e Mooney (1996), consistem basicamente na criação de estruturas lógicas que representam os elementos de um banco de dados. A partir disso, a pergunta do usuário é interpretada por meio de uma análise semântica, utilizando estruturas em forma de árvore. Nesse contexto, a análise semântica realiza a decomposição da consulta em seus elementos significativos, organizando-os hierarquicamente. Cada termo é representado como um nó, e as conexões entre os nós seguem a estrutura lógica da linguagem. Essa abordagem facilita a tradução dos termos da linguagem natural para a lógica formal exigida por uma consulta SQL, permitindo identificar, por ordem de relevância, os elementos fundamentais da pergunta (KIM et al., 2020).

Os avanços na área de LLMs têm se apresentado como uma feramenta poderosa para a realização da tarefa de Text-to-SQL (SHI et al., 2024). Esses modelos possuem a capacidade de realizar tarefas apenas seguindo uma instrução em linguagem natural fornecida em um *prompt* (BROWN et al., 2020). Nesse cenário, é possível se aproveitar dessa caracteríscа para utilizar um LLM como o componente responsável por fazer o mapeamento entre uma pergunta em linguagem natural e a consulta SQL correspondente, bastando fornecer a instrução solicitando essa conversão (HONG et al., 2024). Esse processo, pode ser aprimorado, caso o LLM tenha conhecimentos sobre a base de dados em que se deseja realizar a consulta (SHI et al., 2024). Assim, se na instrução fornecida ao LLM conter informações como o esquema da tabela que desejamos extrair uma informação, é possível que o LLM produza uma resposta mais precisa.

Sob essa ótica, é possível dividir esse processo de Text-To-SQL em três principais aspectos: a compreensão da pergunta, a compreensão do esquema e a geração do SQL (HONG et al., 2024). A compreensão da pergunta refere-se à capacidade do modelo conseguir alinhar a necessidade semântica da pergunta em linguagem natural com a consulta SQL correspondente. Para auxiliar no processo de compreensão do esquema, o fornecimento de informações como a estrutura e as colunas da tabela permitem com que o modelo identifique quais informações o auxiliarão a produzir uma resposta alinhada a necessidade do usuário. Por fim, ao fornecer essas informações no *prompt*, aumentam-se significativamente as chances de o LLM produzir uma resposta e uma consulta SQL precisa (SHI et al., 2024).

Um dos principais desafios, no entanto, está no acesso aos esquemas das tabelas (HONG et al., 2024). Uma abordagem possível seria exigir que o próprio usuário informe manualmente o esquema da tabela onde deseja realizar a consulta. Contudo, considerando o perfil de usuários sem conhecimento técnico, é improvável que conheçam as tabelas disponíveis ou que saibam o que são esquemas e como obtê-los. Dessa forma, ainda existem barreiras que dificultam o acesso pleno desses usuários às bases de conhecimento.

Para contornar esse problema, propomos a utilização da técnica de RAG, que

permite o armazenamento e a recuperação de informações estruturadas, como os esquemas das tabelas, a fim de auxiliar na geração de consultas SQL (SHI et al., 2024). Essa abordagem pode facilitar o desenvolvimento de soluções que permitam a usuários sem conhecimento técnicos acessarem dados de maneira mais intuitiva e precisa. A seguir, exploraremos como podemos utilizar essa técnica para aprimorar o processo de geração consultas SQL.

3.4 Estratégia de Text-to-SQL baseada em RAG

A aplicação de Retrieval-Augmented Generation (RAG) à tarefa de Text-to-SQL aproveita bases de dados vetoriais para armazenar dados estruturados, nesse caso em particular, esquemas de tabelas, que descrevem colunas, tipos de dados e restrições de cada relação (SHI et al., 2024). Diante de uma pergunta em linguagem natural, realizamos (i) a recuperação dos esquemas mais relevantes para o contexto, (ii) a montagem de um *prompt* que combina esses esquemas com a instrução do usuário, e (iii) o envio desse contexto ao LLM para gerar a consulta SQL correspondente.

```
1 CREATE TABLE Filmes (
2     id_filme INT PRIMARY KEY,
3     titulo VARCHAR(100) NOT NULL,
4     diretor VARCHAR(100),
5     ano_lancamento INT,
6     genero VARCHAR(50)
7 );
```

Código 3.4: Exemplo de esquema de uma tabela de Filmes.

Por exemplo, observe o Código 3.4, que representa uma estrutura simplificada para o armazenamento de informações sobre filmes em um sistema de cadastro. Esta tabela é composta por cinco colunas, cada uma com um tipo de dado adequado à natureza da informação que armazena. Nesse contexto, temos a possibilidade de construir um *prompt* bem estruturado, combinando esse esquema com a pergunta feita pelo usuário e outros conhecimentos adicionais. Segundo Shi et al. (2024),

podemos dividir esse processo em duas etapas: pré-processamento e inferência.

3.4.1 Pré-Processamento

Na descrição das tarefas a serem realizadas, é fundamental que estas sejam apresentadas de forma clara e explícita, incluindo todas as informações necessárias para que o problema em questão possa ser resolvido. A estrutura do *prompt* deve conter, principalmente: a representação da pergunta feita pelo usuário, o esquema da tabela e, se necessário, conhecimentos adicionais relacionados à tarefa, para oferecer mais contexto.

Um dos principais problemas encontrados nesse processo é a falta de clareza nas descrições das tarefas a serem realizadas e a ambiguidade presente nos esquemas de bancos de dados. Esses problemas impactam significativamente a qualidade das respostas geradas pelo LLM (REYNOLDS; MCDONELL, 2021).

Dessa forma, faz-se necessário adotar métodos eficientes para a construção dos *prompts*. Esse processo pode ser dividido em três etapas: a representação da questão, a recuperação dos esquemas das tabelas armazenadas na base de dados vetorial e a adição de conhecimentos adicionais.

3.4.1.1 Representação da Questão

No contexto de tarefas de Text-to-SQL, a questão representada no *prompt* é composta pela declaração do problema em linguagem natural, juntamente com as informações necessárias sobre a base de dados.

Um exemplo de *prompt* que pode ser utilizado é o *"Create Table"*, proposto por Shi et al. (2024). Podemos ver uma adaptação desse modelo no Código 3.5. O *prompt* é constituído por três elementos principais: uma instrução clara indicando que a resposta gerada deve ser composta exclusivamente pela consulta em SQL, os esquemas das tabelas armazenadas na base de dados vetorial relacionados à questão e a pergunta em linguagem natural.

```
1 ### Complete sqlite SQL query only and with no explanation.
```

```
2  ### Sqlite SQL tables, with their properties:
3  CREATE TABLE schools (
4      SchID INTEGER PRIMARY KEY,
5      city TEXT
6  );
7
8  CREATE TABLE teachers (
9      TechID INTEGER PRIMARY KEY,
10     SchID INTEGER REFERENCES schools(SchID)
11 );
12 ### {Question}
```

Código 3.5: Exemplo de prompt para Text-to-SQL contendo uma instrução clara sobre a geração da resposta em SQL, o esquema da tabela e a pergunta em linguagem natural.

3.4.1.2 Recuperação de Esquemas

Outro passo essencial no processo de Text-to-SQL baseado em RAG é a recuperação de esquemas, que é responsável por identificar quais tabelas e colunas se relacionam diretamente com o problema especificado na pergunta. A utilização de esquemas redundantes ou irrelevantes pode prejudicar significativamente a geração de respostas pelo LLM (SHI et al., 2024).

Como demonstrado por Guo et al. (2023), os esquemas são armazenados em uma base de dados vetorial, dessa forma, é possível realizar uma busca comparando os termos utilizados na pergunta em linguagem natural com os elementos do banco de dados representados pelos esquemas. Os elementos que possuírem maior similaridade são retornados.

Esse processo apresenta diversas vantagens. Primeiramente, ele permite reduzir o número de *tokens* utilizados na questão, o que é especialmente útil em casos de grandes bases de dados, dado que o número de *tokens* disponíveis para a construção do *prompt* é limitado. Além disso, essa abordagem pode potencialmente melhorar o

desempenho do LLM, pois diminui as distrações causadas por informações irrelevantes, facilitando a identificação dos elementos mais importantes da tarefa.

3.4.1.3 Conhecimentos Adicionais

Adicionar conhecimentos extras à questão, além da pergunta em linguagem natural e dos esquemas retornados, é extremamente benéfico. Esses conhecimentos fornecem ao LLM um contexto mais amplo, o que, consequentemente, permite a produção de respostas mais precisas.

Esses conhecimentos podem estar relacionados ao próprio SQL, como informações de sintaxe, palavras-chave e padrões de escrita. Além disso, conhecimentos externos, como termos específicos do domínio em questão, podem auxiliar o LLM a gerar respostas mais precisas, uma vez que é difícil para o modelo identificar algo como relevante sem compreender seu significado dentro daquele contexto.

3.4.2 Inferência

Uma vez que a pergunta e os esquemas relevantes foram definidos, torna-se possível a construção do *prompt*, como demonstrado no Código 3.5. Esse *prompt* irá integrar, de forma coesa, a instrução da tarefa, os esquemas retornados pela base vetorial e, se disponível, o conhecimento adicional pertinente à questão. Com isso, o próximo passo é submeter o *prompt* ao LLM, que, por sua vez, realizará a geração da consulta SQL correspondente à pergunta em linguagem natural.

É importante destacar que, durante o processo de inferência, a qualidade da resposta gerada depende diretamente da qualidade do *prompt* construído (BROWN et al., 2020). *Prompts* mal formulados, esquemas irrelevantes ou a ausência de informações podem comprometer a precisão da resposta gerada (SHI et al., 2024). Por esse motivo, o uso de técnicas de recuperação e estruturação cuidadosa do contexto é essencial para garantir bons resultados.

Dessa forma, a estratégia baseada em RAG para a tarefa de Text-to-SQL oferece uma solução flexível e adaptável a diferentes domínios e bancos de dados, sem a

necessidade de treinamento adicional dos modelos. Dada sua eficiência e capacidade de generalização, essa abordagem será adotada como base para a proposta do sistema a ser desenvolvido neste trabalho.

Capítulo 4

Luigui: um Sistema Web para Text-To-SQL baseado em LLM utilizando a técnica de RAG

"Portanto, se vocês comem, ou bebem
ou fazem qualquer outra coisa,
façam tudo para a glória de Deus."

— 1ª Coríntios 10:31

Esse capítulo será responsável por explicar como nosso sistema *web* executará a tarefa do Text-to-SQL por meio de um LLM em conjunto com a técnica RAG, para assim, transformar perguntas em linguagem natural em consultas SQL. Assim, também será incluída a modelagem da solução, sua arquitetura e conceitos necessários para que o leitor consiga entender como a proposta de solução funciona. Além disso, apresentaremos alguns trabalhos relacionados relevantes ao presente trabalho.

4.1 Motivação

Todo tipo de empresa e organização, como universidades, *start-ups*, escritórios e hospitais, gera dados e precisa manuseá-los, seja para consultar, atualizar ou remover

alguma informação. Além disso, esses dados podem ser utilizados para a tomada de decisões e para análises diversas. Contudo, para gerenciar essas informações, são necessários sistemas capazes de armazenar e recuperar dados de forma eficiente. É nesse contexto que surgem os sistemas de Banco de Dados (BD) (SHARMA et al., 2021).

A SQL emergiu como a ferramenta que se tornou referência no mercado de trabalho quando se trata da manipulação de BD (LIBKIN, 2003). Essa relevância está relacionada a diversas características intrínsecas da linguagem, como a capacidade de empregar a álgebra relacional, uma linguagem de consulta formal que define operações sobre conjuntos. A utilização desse conceito teórico foi o que garantiu ganhos significativos de desempenho na manipulação de dados[1].

De acordo com o Stack Overflow Developer Survey de 2024[2], a SQL é a quarta linguagem de programação mais utilizada pelos programadores. Nesse contexto, é perceptíviel que independente da área de desenvolvimento, se o desenvolvedor precisar interagir com BDRs, o SQL será o padrão[3].

Entretanto, no dia a dia das empresas, muitos funcionários que não possuem conhecimento técnico sobre essa ferramenta precisam acessar os bancos de dados para extrair, analisar e compreender os dados (FARIHA et al., 2020). Por exemplo, profissionais de Recursos Humanos podem precisar analisar dados de seus colaboradores, como desempenho, informações pessoais e dados financeiros. Já profissionais da educação poderiam avaliar o rendimento dos alunos, número de faltas e taxas de evasão. As necessidades são concretas e se aplicam a diversas áreas do conhecimento.

Segundo Fariha et al. (2020), a democratização dos sistemas computacionais exige acesso igualitário para pessoas com diferentes níveis de habilidade e formações. Nesse sentido, faz-se necessário o desenvolvimento de aplicações ou técnicas capazes de facilitar o uso desses BDs por usuários que não possuem conhecimento técnico especializado em SQL.

[1] <https://cacm.acm.org/practice/the-singular-success-of-sql/>
[2] <https://survey.stackoverflow.co/2024/>
[3] <https://spectrum.ieee.org/the-rise-of-sql>

Nesse cenário, o desenvolvimento de uma interface humano-computador que torne mais acessível o acesso de usuários com baixo ou nenhum conhecimento técnico se torna algo extremamente relevante. Esse tipo de aplicação pode reduzir a necessidade de que esses usuários estejam em comunicação constante com o setor de desenvolvimento da empresa, visto que, a cada nova consulta, eles precisam solicitar o suporte para a extração de informações. Essa situação pode gerar gargalos no dia a dia de uma empresa. Todo esse processo poderia ser minimizado com o fornecimento de uma aplicação que facilitasse o acesso a esses dados para usuários sem esse conhecimento técnico.

Além disso, usuários não técnicos enfrentam barreiras não apenas relacionadas ao domínio da linguagem SQL, mas também ao próprio acesso aos bancos de dados. Em muitos ambientes corporativos, os privilégios de acesso são restritos a equipes técnicas, tanto por razões de segurança quanto por limitações operacionais. Isso impede que colaboradores de áreas como marketing, finanças ou recursos humanos realizem consultas simples por conta própria, o que pode atrasar decisões importantes e reduzir a eficiência organizacional.

Mesmo quando o acesso é concedido, a interface dos sistemas de gerenciamento de bancos de dados costuma ser técnica e pouco amigável, o que dificulta ainda mais o uso por parte de pessoas sem formação em computação. Esses usuários podem não saber como estruturar uma consulta corretamente, interpretar mensagens de erro ou compreender os relacionamentos entre tabelas. Com isso, consultas que poderiam ser feitas em segundos acabam sendo postergadas ou descartadas.

Portanto, é urgente o desenvolvimento de soluções que reduzam essas barreiras, tanto em termos de acesso quanto de usabilidade. Sistemas baseados em linguagem natural, que é a linguagem conhecida pelos usuário, capazes de interpretar perguntas e convertê-las automaticamente em comandos SQL, oferecem uma alternativa promissora. Nesse sentido, a exploração de como utilizar técnicas de Text-To-SQL para o desenvolvimento dessas aplicações é essencial. Nesse cenário, uso de LLMs para realizar esse mapeamento entre a linguagem natural e consultas SQL tem se destacado (SHI et al., 2024).

Os LLM se apresentam como uma ferramenta poderosa para seguir e executar instruções dadas pelos usuários (BROWN et al., 2020). Assim, podemos nos aproveitar dessa característica para o desenvolvimento soluções que beneficiem usuário que não possuam a capacidade de formular consultas SQL, devido a falta de conhecimento. Nesse caso, deseja-se uma aplicação que seja capaz dado uma pergunta em linguagem natural, acessar os dados estruturados de um banco de dados para obter informações sobre as tabelas, gerar a consulta SQL equivalente e por fim executá-la para que o usuário tenha acesso aos dados que ele necessita.

Uma das principais dificuldades técnicas na construção desse tipo de sistema de Text-To-SQL está na identificação da tabela, ou conjunto de tabelas, mais relevante para a pergunta feita. Quando um usuário realiza uma consulta como "Quais alunos tiveram mais de três faltas no último mês?", o modelo precisa inferir não apenas a intenção da pergunta, mas também quais tabelas e colunas do banco contêm essa informação. Esse processo exige uma capacidade de compreensão semântica que nem sempre é trivial, especialmente em bancos com esquemas complexos ou mal documentados.

Contudo, surge o desafio de como tornar esse processo eficiente. Uma abordagem ingênua consistiria em gerar consultas SQL candidatas para todo esquema de todas as tabelas do banco a partir da pergunta, mas isso rapidamente se torna inviável conforme o número de tabelas aumenta. Além do custo computacional, isso pode levar à geração de consultas irrelevantes ou incorretas. A alternativa de filtrar previamente quais tabelas devem ser consideradas, utilizando *embeddings* e mecanismos de recuperação semântica, torna-se, portanto, fundamental para a escalabilidade da solução.

Nesse contexto, a utilização de *embeddings* para representar os esquemas das tabelas é promissora. Ao transformar os metadados, como nomes das tabelas, colunas, chaves estrangeiras e tipos de dados, em vetores num espaço semântico, é possível utilizar técnicas de similaridade vetorial para estimar quais tabelas mais se relacionam com a consulta do usuário (ALMEIDA; XEXÉO, 2019). Essa etapa de recuperação de contexto estrutural pode ser integrada à arquitetura de RAG, fornecendo ao LLM apenas o subconjunto relevante do esquema para a tarefa de geração da consulta

SQL.

Outro ponto que pode ser considerado é a utilização de *embeddings* não apenas do esquema, mas também dos dados propriamente ditos, isto é, das linhas contidas nas tabelas. Embora tecnicamente possível, essa abordagem apresenta sérios entraves práticos. A criação e o armazenamento de *embeddings* para cada linha de um banco de dados relacional pode ser extremamente custosa em termos de tempo e recursos. Nesse sentido, faz necessária uma abordagem que seja capaz de fornecer as informações necessárias para a geração e que mantenha um desempenho computacional aceitável.

4.2 Trabalhos Relacionados

O ChatGPT[4] é uma aplicação que adapta um LLM para conversação com humanos, nesse sentido ele preserva uma das principais características desses modelos que é a capacidade de seguir instruções para gerar respostas. Ele pode ser utilizado em diversas tarefas, incluindo a conversão de linguagem natural para SQL. Durante a interação, é possível enviar mensagens contendo os esquemas das tabelas do usuário e, em seguida, realizar perguntas em linguagem natural para que o modelo gere as respectivas consultas em SQL. No entanto, não é possível realizar o cadastro ou a utilização direta de BDs pessoais por parte dos usuários, sendo necessário executar manualmente as consultas geradas.

O *EverSQL*[5] é uma ferramenta voltada para a otimização do uso de consultas em bancos de dados, além de oferecer diagnósticos de desempenho. Entre suas funcionalidades, destaca-se a conversão de linguagem natural em consultas SQL. Como ilustrado na Figura 4.1, a ferramenta permite que o usuário selecione o tipo de banco de dados utilizado, insira a pergunta em linguagem natural e, opcionalmente, descreva a estrutura das tabelas. Apesar de ser útil para uso pontual, a ferramenta pode gerar respostas menos precisas, além de não realizar a persistência dos dados utilizados ou gerados.

[4] <https://chatgpt.com/>

[5] <https://www.eversql.com/text-to-sql/>

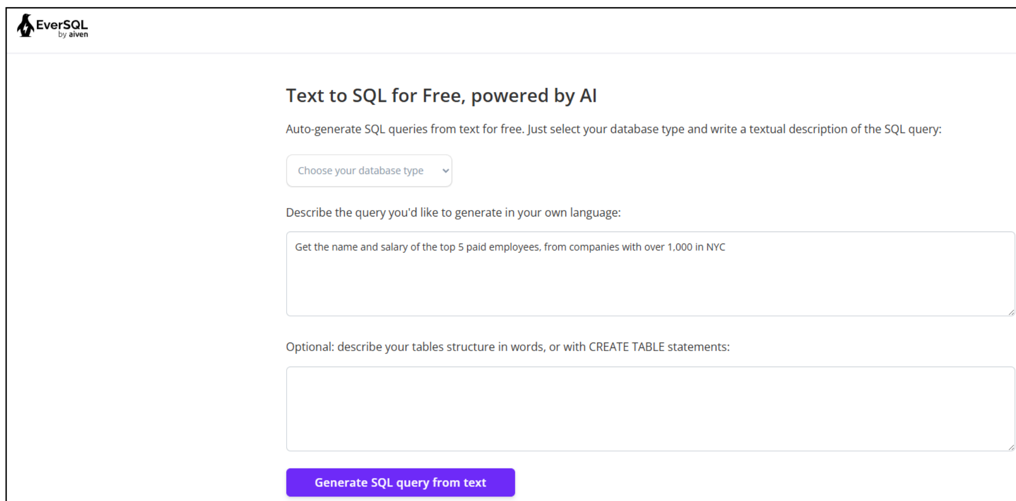

Figura 4.1: Função de Text-To-SQL na ferramenta EverSQL.

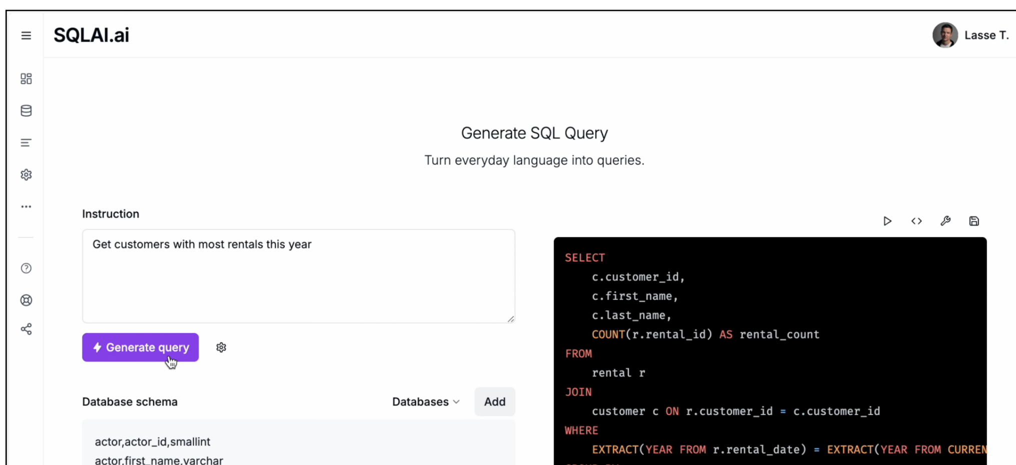

Figura 4.2: Função de Text-To-SQL na ferramenta SQLAI.ai.

O SQLAI.ai[6] é uma aplicação que utiliza IA para realizar diversas operações com bancos de dados, uma delas é a conversão de linguagem natural em consultas SQL, como visto na Figura 4.2. Além disso essa aplicação permite realização dessas operações não só com bancos de dados SQL, mas também com bancos de dados NoSQL. Diferentemente dos bancos relacionais, os bancos NoSQL não utilizam o modelo de tabelas com esquemas fixos, sendo mais flexíveis na representação de dados. Com isso, o suporte a bancos NoSQL poderia ampliar o escopo da aplicação para atender a cenários mais diversos.

Alternativamente, ferramentas visuais como o *Draxlr*[7] fornecem editores intera-

[6] <https://www.sqlai.ai/>
[7] <https://www.draxlr.com/>

tivos que abstraem a sintaxe SQL por meio de componentes de arrastar e soltar, facilitando a construção de consultas complexas mesmo para usuários sem conhecimento técnico aprofundado na linguagem. Como pode ser observado na Figura 4.3, é possível definir uma consulta de agrupamento apenas selecionando a tabela e os campos desejados por meio da interface gráfica da aplicação.

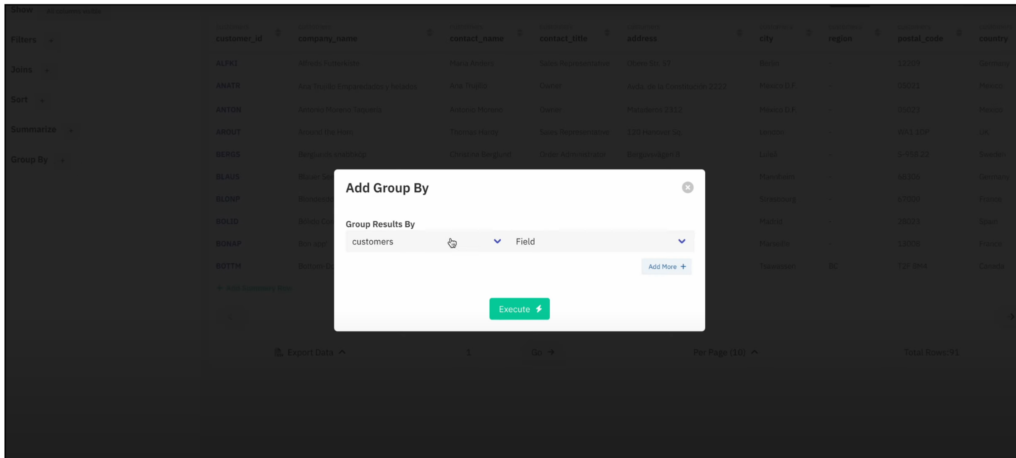

Figura 4.3: Criação de consultas na ferramenta Draxlr.

4.3 Proposta

Este trabalho tem como objetivo desenvolver um sistema *web* capaz de democratizar o acesso e a utilização de bancos de dados por usuários que não possuem conhecimento técnico sobre SQL. Para isso, propomos uma abordagem que utiliza LLMs em conjunto com técnicas de RAG para realizar a tarefa de *Text-to-SQL*. A proposta concentra-se em fornecer ao modelo apenas as partes relevantes do banco de dados, isto é, os esquemas das tabelas selecionadas com base na similaridade semântica com a pergunta, permitindo que a consulta seja gerada de forma precisa, eficiente e com menor dependência de intervenção humana.

O sistema proposto será capaz de estabelecer conexão direta com o banco de dados do usuário. Dessa forma, será possível realizar o cadastro das tabelas por meio do armazenamento dos respectivos esquemas em uma base de dados vetorial. Após esse cadastro, o usuário poderá realizar perguntas em linguagem natural. A partir dessas perguntas, o sistema irá recuperar os k esquemas mais relevantes da

base vetorial, utilizando mecanismos de similaridade semântica. Em seguida, será construído um *prompt* contendo os esquemas recuperados e a pergunta formulada, o qual será enviado a um LLM para geração da consulta SQL. Essa consulta será então executada diretamente no banco de dados do usuário. Ao final do processo, o usuário receberá uma resposta em linguagem natural, gerada a partir do resultado da consulta, bem como a própria consulta SQL que originou aquela resposta. O sistema também oferecerá funcionalidades adicionais, como: otimização, explicação e correção de consultas SQL, por meio de *prompts* especializados, além disso, o usuário poderá visualizar, por meio de uma tela, o histórico de consultas que já foram realizadas. Todo esse processo será detalhado nas seções subsequentes.

O sistema permitirá dois modos de cadastro de bancos de dados: o modo "*complete*" e o modo "*minimal*". No modo "*complete*", o usuário autoriza uma conexão direta com seu banco de dados, permitindo que o sistema extraia automaticamente os esquemas das tabelas que forem cadastradas pelo usuário e execute as consultas diretamente no banco de dados. Já no modo "*minimal*", o usuário não precisa estabelecer essa conexão. Nesse caso, o cadastro é feito informando os esquemas das tabelas. Com isso, será possível apenas a geração das consultas SQL, ficando a execução a cargo do próprio usuário.

Além disso, o sistema contará com dois tipos de cargos de usuário: administrador e funcionário. O cargo administrador é destinado a usuários com conhecimento técnico em bancos de dados. Esse cargo terá permissão para cadastrar novas tabelas e também realizar perguntas. Já o cargo funcionário é voltado a usuários sem conhecimento técnico avançado, que poderão apenas fazer perguntas ao sistema, utilizando linguagem natural.

Sob essa ótica, o propósito central deste trabalho é facilitar tarefas como inserção, visualização, correção e exclusão de dados, especialmente para usuários com menor familiaridade com tecnologia. A proposta visa proporcionar autonomia na exploração de dados, permitindo que usuários acessem informações que antes estavam restritas a profissionais da área técnica.

Assim, esse capítulo descreverá a arquitetura do sistema, elencando os principais

pontos do fluxo, cadastro de bancos de dados, tabelas e geração de perguntas. Além disso, abordaremos com mais detalhes o fluxo da técnica de RAG, exploraremos a modelagem do banco de dados relacional do nosso sistema e também a utilização do banco de dados vetorial. Por fim, encerraremos o capítulo aprofundando a camada da Application Programming Interface (API) que realiza a chamada das funções do RAG.

4.3.1 Arquitetura do Sistema

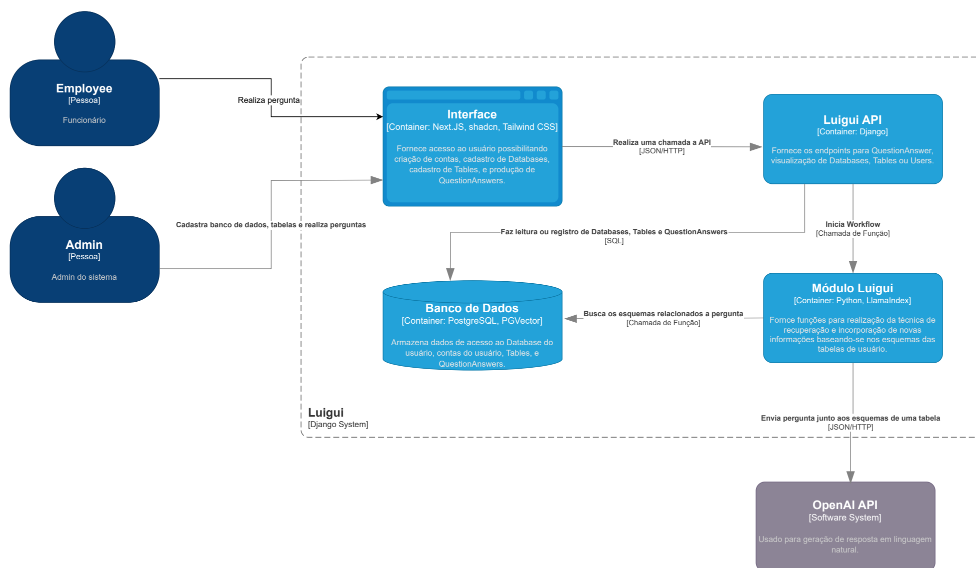

Figura 4.4: Diagrama C4 com a arquitetura do sistema proposto.

A Figura 4.4 apresenta uma visão geral da arquitetura do sistema denominado "Luigui". O sistema conta com uma interface *web*, projetada para garantir acessibilidade, clareza e facilidade de uso para diferentes perfis de usuários. Por meio dessa interface, é possível realizar funcionalidades como criação de contas, autenticação, cadastro de bancos de dados e tabelas, além da realização de perguntas em linguagem natural.

As funcionalidades da interface são disponibilizadas por meio de um *backend*, que expõe uma camada de serviços através da Luigui API. Essa API disponibiliza diversos *endpoints*, responsáveis por orquestrar as operações realizadas no sistema, os quais serão detalhados nas seções subsequentes.

A implementação da arquitetura baseada na técnica RAG está centralizada no Módulo Luigui. Esse módulo é responsável por controlar todo o processo de recuperação dos esquemas das tabelas relevantes e pela geração da consulta SQL. A esse processo, demos o nome de *workflow*.

O fluxo de execução ocorre da seguinte forma: ao realizar uma ação na interface, como o envio de uma pergunta, um *endpoint* da Luigui API é acionado. Esse *endpoint* processa os dados fornecidos pelo usuário e inicia o *workflow* de execução da técnica de RAG, utilizando os recursos da arquitetura interna.

A persistência dos dados fornecidos pelos usuários, como credenciais de acesso ao banco, metadados das tabelas, perguntas realizadas e os históricos de respostas geradas, é feita por meio de um banco de dados relacional. Por outro lado, os esquemas das tabelas são armazenados em um banco de dados vetorial, que viabiliza a busca semântica eficiente, fundamental para a etapa de recuperação no *workflow* do RAG.

Por fim, a Luigui API integra-se à API da OpenAI por meio de uma chave de acesso. Essa abordagem viabiliza o uso de LLMs sem a necessidade de treinar ou manter um modelo próprio, o que seria inviável dentro do escopo deste trabalho. Para os testes realizados durante o desenvolvimento, utilizou-se o modelo de linguagem "gpt-4-turbo-2024-04-09", que, à época, se destacava como uma versão mais barata do GPT-4. Além disso, esse modelo oferece suporte à funcionalidade de *Structured Outputs*, permitindo que a API da OpenAI retorne as respostas no formato JSON. Isso facilitou o isolamento da consulta SQL em um campo específico do JSON, sem a presença de caracteres adicionais, o que possibilitou o envio direto do comando para execução no banco de dados do usuário.

4.3.1.1 Modulo Luigui

O Módulo Luigui é responsável por gerenciar todo o processo RAG, conforme discutido na Seção 3.4. A arquitetura é dividida em quatro etapas principais: indexação, recuperação, geração e execução. O processo inicia-se pela etapa de indexação, em que os dados estruturados, ou seja, os esquemas das tabelas cadastradas

pelo usuário, são convertidos em representações vetoriais, conhecidas como *embeddings*. Esses *embeddings* são então armazenados em um banco de dados vetorial.

Essa abordagem possibilita a busca semântica por esquemas de tabelas que tenham maior relação com a pergunta formulada pelo usuário. Além disso, ao manter os esquemas indexados, o sistema dispensa a necessidade de que o usuário conheça todos os detalhes do banco de dados, ou que informe explicitamente as tabelas envolvidas em cada pergunta. Assim, a experiência do usuário é simplificada, especialmente para aqueles com pouca familiaridade com SQL.

A etapa seguinte é a de recuperação. Nela, o usuário formula uma pergunta em linguagem natural e seleciona o tipo de operação desejada. As opções disponíveis incluem: gerar uma consulta SQL com base em linguagem natural, otimizar uma consulta, explicar uma consulta ou corrigir uma consulta. A pergunta do usuário também é transformada em uma representação vetorial, então busca-se os esquemas mais relevantes, com base em similaridade semântica, que são então recuperados para compor o contexto da próxima etapa.

Na fase de geração, o sistema constrói um *prompt* contendo as instruções específicas para o tipo de operação selecionada, os esquemas recuperados e a pergunta original em linguagem natural. Esse *prompt* é enviado a um LLM, que é responsável por gerar a consulta SQL correspondente à solicitação do usuário, ou uma resposta em linguagem natural caso a operação desejada tenha sido de otimização, explicação ou correção de consulta.

Após isso, caso o usuário tenha selecionado a operação de converter uma pergunta em uma consulta SQL, na etapa de execução, o sistema pode executar a consulta gerada diretamente no banco de dados do usuário, desde que ele tenha previamente fornecido as credenciais necessárias. Após a execução, o resultado da consulta é transformado em uma resposta em linguagem natural, retornada ao usuário juntamente com a consulta SQL correspondente.

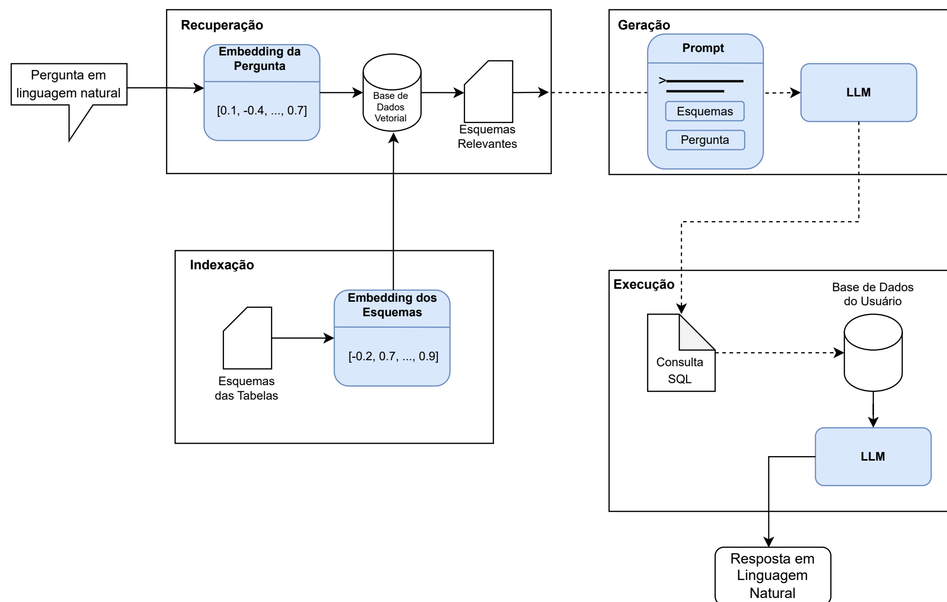

Figura 4.5: Diagrama do Módulo Luigui representando o processo de indexação, recuperação de *embeddings* dos *schemas* relevantes, contrução do *prompt*, geração da consulta e execução da consulta no banco de dados.

4.3.1.2 Banco de Dados Relacional e Vetorial

A persistência dos dados é uma parte crucial do sistema proposto, precisaremos armazenar em um banco de dados relaciona informações sobre o registro de usuários, os dados de conexão com os bancos cadastrados, as tabelas disponíveis para recuperação de informações e todas as perguntas feitas pelos usuários.

Além disso, como já discutido, uma das estratégias utilizadas na tarefa de *Text-To-SQL* baseada em RAG é a recuperação de esquemas relevantes. Para viabilizar isso, é utilizado um banco de dados vetorial que permite operações baseadas em *embeddings*. Esse banco de dados vetorial está diretamente relacionado com os dados armazenados no banco de dados relacional, com isso, a seguir, detalharemos como essa integração ocorre.

O banco de dados relacional foi modelado com base em quatro entidades principais: `User`, `Database`, `Table` e `QuestionAnswer`, conforme apresentado no Diagrama Entidade-Relacionamento da Figura 4.6. A entidade `User` representa os usuários do sistema e armazena informações como `name`, `password` e `email`. Um relacionamento

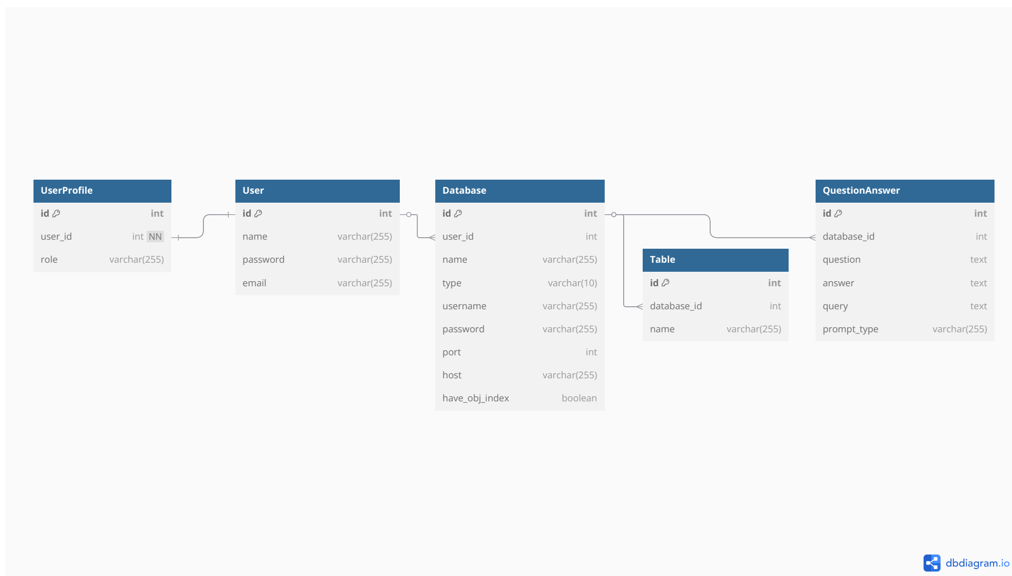

Figura 4.6: Diagrama de Entidade-Relacionamento do sistema proposto

do tipo “um-para-muitos” liga `User` à entidade `Database`, indicando que um mesmo usuário pode cadastrar múltiplos BDs no sistema.

A entidade `Database` representa os bancos de dados registrados pelos usuários. Ela contém o atributo `type`, que indica se o BD é do tipo *“complete”* ou *“minimal”*. No caso *“complete”*, o usuário fornece os dados necessários para a conexão direta com seu BD pessoal, como `name`, `username`, `password`, `host` e `port`. Nesse trabalho, a conexão só é permitida para bancos de dados PostgreSQL. Com essas informações, o sistema constrói a *string* de conexão que permite o acesso ao banco de dados do usuário para a extração dos esquemas das tabelas e realização de consultas. O atributo `have_obj_index` indica se o índice vetorial já foi gerado, onde os *embeddings* dos esquemas das tabelas serão armazenados, o índice só é gerado após o armazenamento de uma tabela.

No tipo *“minimal”*, o usuário opta por não fornecer acesso direto ao seu banco de dados. Nesse cenário, apenas os campos `name` e `type` são obrigatórios. Esse modelo é útil em contextos onde o usuário deseja maior privacidade ou quando há necessidade apenas da geração da consulta em SQL, sem sua execução.

Após o registro de um `Database`, o usuário deve informar quais tabelas deseja cadastrar, o que é representado pela entidade `Table`. Ela possui um relacionamento

"muitos-para-um"com `Database`, já que um banco pode conter várias tabelas. No caso *"complete"*, o usário precisa cadastrar as tabelas manualmente, apenas fornecendo o nome de quais tabelas devem ter seu esquema indexado. No caso *"minimal"*, o usuário precisa fornecer o esquema manualmente, que é enviado diretamente ao banco de dados vetorial, nesse caso armazenamos apenas o nome da tabela no banco de dados relacional.

Por fim, a entidade `QuestionAnswer` registra as perguntas feitas em linguagem natural pelo usuário, por meio do atributo `question`, e as respostas geradas pelo sistema no atributo `answer`. A entidade também armazena o tipo de tarefa executada através do atributo `prompt_type`, e, quando aplicável, a consulta em SQL gerada, armazenada no atributo `query`. Existe um relacionamento "muitos-para-um" entre `QuestionAnswer` e `Database`, visto que diversas perguntas podem ser feitas com base em um mesmo banco de dados.

Esses modelos são fundamentais para viabilizar as operações oferecidas pelos *endpoints* da camada Luigui API, garantindo organização, integridade e facilidade na integração entre a interface *web* e os componentes internos do sistema.

4.3.1.3 Luigui API

Uma Application Programming Interface (API) é um conjunto de definições e protocolos que permite a comunicação entre diferentes partes de um *software* ou entre sistemas distintos. APIs são comumente utilizadas para abstrair a complexidade de funcionalidades internas, fornecendo interfaces bem definidas que possibilitam a reutilização de serviços e a integração com outras aplicações. No contexto de aplicações *web*, os *endpoints* representam os pontos de acesso expostos por uma API. Cada *endpoint* está associado a uma operação específica e pode ser acessado por meio de requisições Hypertext Transfer Protocol (HTTP), como `GET`, `POST`, `PUT` e `DELETE`.

A Luigui API funciona como uma camada acima do Módulo Luigui. Ela atua tanto como interface de comunicação entre a aplicação *web* e os componentes internos do sistema quanto como uma camada de abstração, ao ocultar os detalhes técnicos

da arquitetura do RAG. Dessa forma, desenvolvedores podem utilizar os serviços oferecidos pelo sistema, como a geração de consultas em SQL a partir de linguagem natural, sem a necessidade de compreender em profundidade os mecanismos de RAG.

Nesse sentido, a Luigui API expõe *endpoints* específicos para diferentes funcionalidades, como o cadastro de usuários, registro de banco de dados, definição de tabelas e realização de perguntas. Além disso, os *endpoints* permitem a personalização e o controle sobre os `Databases`, possibilitando operações como criação, que pode ser do tipo *"minimal"* ou *"complete"*, atualização e remoção. Também é possível realizar o gerenciamento completo das operações relacionadas às `Tables`, incluindo seu cadastro, alteração e exclusão.

O *endpoint* responsável pela realização de perguntas está diretamente associado a um dos `Databases` cadastrados pelo usuário. Ele permite a seleção do tipo de tarefa a ser executada, de acordo com os *workflows* definidos na aplicação, os quais serão detalhados no capítulo seguinte. Na tarefa de Text-To-SQL, por exemplo, o usuário fornece uma pergunta em linguagem natural; caso o `Database` seja do tipo *"complete"*, o *endpoint* retorna tanto a consulta gerada em SQL quanto uma resposta em linguagem natural baseada nos resultados da execução dessa consulta. Cada *endpoint* foi para atender a uma etapa específica do fluxo de uso do sistema, garantindo modularidade, clareza e eficiência no processo de integração com a interface *web*.

Capítulo 5

Implementação

"O medo é o assassino da mente. O medo é a pequena morte que leva à aniquilação total. Enfrentarei meu medo. Permitirei que passe por cima e através de mim. E, quando tiver passado, voltarei o olho interior para ver seu rastro. Onde o medo não estiver mais, nada haverá. Somente eu restarei."

— Frank Herbert, Duna

Neste capítulo, será apresentada a implementação da solução proposta, detalhando as tecnologias utilizadas no desenvolvimento, justificando sua escolha e relacionando suas funções específicas, os padrões utilizados e os resultados obtidos.

5.1 Tecnologias Utilizadas

O servidor *web* foi desenvolvido utilizando o *framework* Django, enquanto o banco de dados é gerenciado pelo PostgreSQL. Além disso, a persistência e busca semântica das informações dos esquemas das tabelas é realizada com a extensão PGVector, a partir dos *embeddings*. A arquitetura de RAG foi implementada utilizando o *framework* LlamaIndex, abrangendo assim todos os passos para recuperação, incorporação de informação e também a manipulação de dados no banco de dados vetorial. Por fim, a interface foi desenvolvida utilizando o Next.js, um *framework*

React, além da utilização do Tailwind para estilização das páginas. As subseções a seguir detalham cada uma dessas tecnologias e suas funções na construção do sistema.

5.1.1 Django

O Django[1] é um *framework* de alto nível para a linguagem Python, voltado para o desenvolvimento rápido de aplicações *web* com um *design* limpo e pragmático. Ele segue o padrão de arquitetura Model-View-Template (MVT), que é uma variação do padrão Model-View-Controller (MVC), facilitando a organização do código e a separação de responsabilidades.

Para o presente trabalho, o Django foi utilizado como base para o desenvolvimento do servidor *web*, lidando com a definição dos *endpoints*, os *models* representam a estrutura de cada uma das entidades no BD, e as *views*, fazem todo o controle do receber requisições HTTP e retornar respostas HTTP. As respostas podem ser páginas HTML completas, dados em formato JSON e mensagens de erro em casos de exceções, dessa forma, as *views* servem como intermediários entre a interface do usuário, e os serviços do sistema.

No módulo de *serializers* se encontram as funcionalidades que traduzem os dados que são recebidos como entrada, validando e processando eles em um JSON, o que ajuda em termos de controle e validação de dados que chegam a partir da entrada, sem que seja necessária uma implementação mais complexa para tratar essas informações. Assim, o uso do Django se mostra conveniente por sua integração nativa com bancos relacionais, suporte robusto a APIs, além de possuir uma comunidade ativa e extensa documentação.

5.1.2 PostgreSQL

O PostgreSQL[2] é um Sistema de Gerenciamento de Banco de Dados Relacional (SGBD) de código aberto, reconhecido por sua robustez, extensibilidade e

[1] <https://www.djangoproject.com/>

[2] <https://www.postgresql.org/>

conformidade com os padrões SQL.

No projeto, o PostgreSQL é responsável por armazenar os dados estruturados da aplicação, como as informações de contas do usuário, os dados de acesso ao banco de dados do usuário, o nome das tabelas, as perguntas feitas e respostas geradas pela LLM, e os esquemas das tabelas no PGVector.

Sua integração com o Django via Object-Relational Mapping (ORM) permite a manipulação dos dados de forma intuitiva, além de suportar extensões que enriquecem sua funcionalidade, como é o caso do próprio PGVector.

5.1.3 PGVector

PGVector[3] é uma extensão do PostgreSQL que permite o armazenamento e a busca por similaridade entre vetores diretamente no banco de dados. Ela é fundamental para aplicações que fazem uso de *embeddings*, como sistemas que utilizam recuperação semântica de informações.

Na aplicação desenvolvida, a extensão PGVector foi empregada para armazenar os *embeddings* dos esquemas das tabelas do usuários, indexados pelo mecanismo RAG, armazenando informações vetoriais sobre cada esquema, essas informações servem como espécies de "coordenadas". Com isso, é possível realizar buscas vetoriais eficientes com base na similaridade semântica, utilizando as coordenadas, entre a consulta do usuário e os documentos armazenados, garantindo que apenas os contextos mais relevantes sejam utilizados na geração da respostas e as consultas SQL.

5.1.4 LlamaIndex

O LlamaIndex[4] é um *framework* que atua como ponte entre dados externos e LLMs, oferecendo funcionalidades para indexação, busca e consulta contextualizada. Ele é especialmente útil em sistemas baseados em RAG, pois organiza a base de

[3] <https://github.com/pgvector/pgvector>
[4] <https://www.llamaindex.ai/>

conhecimento de maneira estruturada para facilitar a recuperação de contexto.

No sistema implementado, o LlamaIndex foi utilizado para criar índices vetoriais a partir de documentos que descrevem o esquema do BD. Esses índices são posteriormente consultados durante a etapa de recuperação da arquitetura RAG, garantindo que o modelo de linguagem receba os contextos mais relevantes antes de gerar a consulta SQL correspondente.

5.1.5 Next.js, Tailwind CSS e Shadcn/UI

A interface do sistema foi implementada utilizando uma stack moderna baseada em React, com o *framework* Next.js[5] como base principal. O Next.js oferece recursos avançados como renderização do lado do servidor (SSR), geração de páginas estáticas, divisão automática de código e pré-carregamento inteligente, o que contribui significativamente para a performance e escalabilidade da aplicação. O uso do Next.js também facilita a criação de rotas e a integração direta com a API desenvolvida em Django.

A biblioteca React serve como núcleo da aplicação, oferecendo a flexibilidade necessária para a construção de interfaces baseadas em componentes reutilizáveis. A linguagem TypeScript foi adotada para adicionar tipagem estática ao código, aumentando a segurança e a previsibilidade no desenvolvimento da interface.

A estilização da interface foi realizada com Tailwind CSS, um *framework utility-first* que permite criar estilos diretamente nas classes dos elementos HTML. Foram utilizados também os utilitários `tailwindcss-animate` para animações suaves e `tailwind-merge` para controle eficiente da composição de classes. Essa abordagem oferece maior agilidade no desenvolvimento, além de promover consistência visual e responsividade.

Para a construção dos componentes da interface, foi utilizada a biblioteca `shadcn/ui`, que integra `Radix UI` e Tailwind CSS para fornecer uma coleção de componentes acessíveis, modernos e altamente personalizáveis.

[5] <https://nextjs.org/>

A implementação dos formulários foi feita com a biblioteca `react-hook-form`, que gerencia o estado dos formulários de forma eficiente e integrada com a biblioteca de validação `zod`, por meio do pacote `@hookform/resolvers`. Essa integração permite validação baseada em esquemas, oferecendo robustez e segurança à entrada de dados.

Essa combinação de tecnologias no *frontend* foi escolhida com o objetivo de proporcionar uma experiência de usuário fluida, acessível e moderna, com interfaces bem estruturadas, responsivas e altamente reutilizáveis.

5.2 Workflows

Uma parte crucial da solução proposta é o RAG. Como já foi mencionado, o *framework* LlamaIndex foi a principal ferramenta utilizada para a implementação das tarefas e etapas do RAG. Esse *framework* oferece um componente essencial para a definição da sequência de execução do RAG, os chamados *workflows*[6]. Um *workflow* no Llamaindex representa uma forma de controle do fluxo de execução em uma aplicação baseada em eventos, sendo estruturado em etapas sequenciais e modulares.

Um *workflow* é formado por seções chamadas *steps*, que são acionadas por *events* e, por sua vez, emitem novos eventos que podem disparar outras etapas subsequentes. A partir da combinação de *steps* e *events*, é possível criar fluxos complexos, que encapsulam a lógica do sistema e tornam a aplicação mais modular, compreensível e de fácil manutenção.

Para este trabalho, foram construídos dois *workflows.* A necessidade da criação desses dois fluxos surgiu devido à existência de dois cenários principais no contexto do RAG: o caso em que as perguntas são realizadas em um `Database` do tipo *"minimal"* e o caso em que o `Database` é do tipo *"complete"*. Dessa forma, para o cenário *"minimal"*, é utilizado o `SimpleTextToSQLWorkflow`, enquanto para o cenário *"complete"* emprega-se o `TextToSQLWorkflow`.

[6]<https://docs.llamaindex.ai/en/stable/understanding/workflows/>

5.2.1 SimpleTextToSQLWorkflow

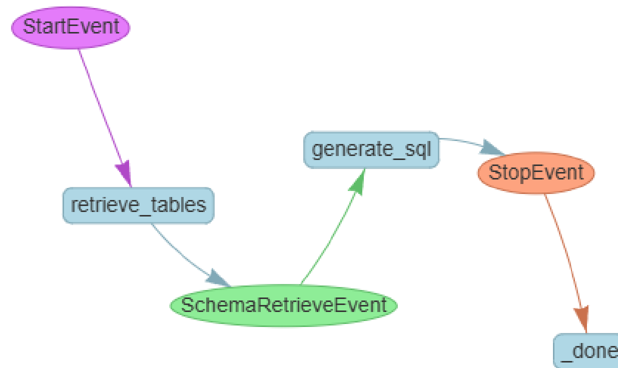

Figura 5.1: Diagrama do SimpleTextToSQLWorkflow que representa o fluxo utilizado para implementar da técnica de RAG para o caso onde o banco de dados cadastrado é do tipo *"minimal"*.

O `SimpleTextToSQLWorkflow` é a implementação do fluxo de RAG responsável por converter uma pergunta em linguagem natural em uma consulta SQL sem acesso direto ao banco de dados do usuário para a execução dessa consulta gerada, ou seja, é um cenário do tipo *"minimal"*. A classe herda de `Workflow` do LlamaIndex e define explicitamente dois componentes fundamentais no construtor:

- **schema_retriever**: instância de `SQLSchemaRetriever`, encarregada de, dado o nome do banco, recuperar apenas os esquemas de tabela previamente cadastrados no banco de dados vetorial e filtrá-los conforme a pergunta do usuário.
- **sql_generator**: instância de `OpenAISQLGenerator`, que encapsula o modelo LLM e a estratégia de *prompt* específica.

A geração das respostas em linguagem natural do *workflow* é realizada por meio de estratégias de *prompt* específicas, encapsuladas por meio do padrão de projeto *Strategy*. Cada estratégia implementa uma interface comum, chamada

`IPromptStrategy`, que define o método `create_prompt()`, responsável por construir dinamicamente o *prompt* a ser enviado ao LLM.

Dependendo da natureza da tarefa, uma das seguintes estratégias é utilizada: `Text ToSQLPromptStrategy`, para gerar consultas SQL a partir de perguntas em linguagem natural; `OptimizesSQLQueryPromptStrategy`, para otimizar consultas existentes; `E xplainSQLQueryPromptStrategy`, para explicar detalhadamente o funcionamento de uma consulta SQL; e `FixSQLQueryPromptStrategy`, para corrigir erros de sintaxe em consultas.

A escolha da estratégia adequada ocorre em tempo de execução, com base no tipo de requisição do usuário. Essas estratégias contribuem para modularizar o processo de construção dos *prompts* e permitem adaptar facilmente o comportamento do sistema a diferentes cenários de uso.

Nesse contexto, o fluxo demonstrado na Figura 5.1 é disparado pelo método de conveniência `starts_simple_workflow`, que instancia o *workflow* e chama o método `run(query, timeout)`. Internamente, o *workflow* executa dois `@step` sequenciais:

1. **retrieve_tables (*Step*)**
   Recebe o evento inicial `StartEvent`, que contém a pergunta do usuário (`ev. query`). Chama `schema_retriever.retrieve(ev.query)`, retornando uma lista de objetos com os esquemas relevantes. Esses esquemas estão em formato de textos e são concatenados em uma única *string*. Essa *string*, junto com a pergunta, é empacotada em um `SchemaRetrieveEvent`, que será utilizado na próxima etapa.

2. **generate_sql (*Step*)**
   Este passo é acionado após a recuperação do esquema. Ele utiliza o conteúdo do `SchemaRetrieveEvent` para construir o dicionário de contexto e pergunta, que é passado para o método `sql_generator.generate()`. A resposta do LLM é processada por `_parse_response_to_sql`, que remove marcações desnecessárias e extrai a instrução SQL. Por fim, retorna-se um `StopEvent` com a consulta SQL final.

Além dos dois `@step` principais, a classe também define um método auxiliar `_get_table_context_str` responsável por adicionar mais contexto ao objeto do esquema. Esse método extrai por meio de um LLM descrições adicionais sobre o conteúdo das tabelas.

No Código 5.1, é apresentado um exemplo do *prompt* utilizado na estratégia `TextToSQLPromptStrategy`, baseado no *template* `DEFAULT_TEXT_TO_SQL_PROMPT`, fornecido pelo LlamaIndex. Nesse *prompt*, é possível especificar a linguagem do banco de dados por meio do parâmetro `dialect`, além de definir os esquemas das tabelas por meio do parâmetro `schemas` e a pergunta em linguagem natural por meio do parâmetro `query_str`.

```
Given an input question, first create a syntactically correct {dialect} query to run, then look at the results of the query and return the answer. You can order the results by a relevant column to return the most interesting examples in the database.

Never query for all the columns from a specific table, only ask for a few relevant columns given the question.

Pay attention to use only the column names that you can see in the schema description. Be careful to not query for columns that do not exist. Pay attention to which column is in which table. Also, qualify column names with the table name when needed.

You are required to use the following format, each taking one line:

Question: Question here
SQLQuery: SQL Query to run
SQLResult: Result of the SQLQuery
Answer: Final answer here
```

```
13
14 Only use tables listed below.
15 {schema}
16
17 Question: {query_str}
18 SQLQuery:
```

Código 5.1: Prompt de Text-to-SQL utilizado na TextToSQLPromptStrategy.

Dessa forma, o `SimpleTextToSQLWorkflow` organiza claramente o processo em três fases: recepção da pergunta, recuperação do contexto e geração da consulta, o que permite modularidade, reutilização e fácil manutenção do código.

5.2.2 TextToSQLWorkflow

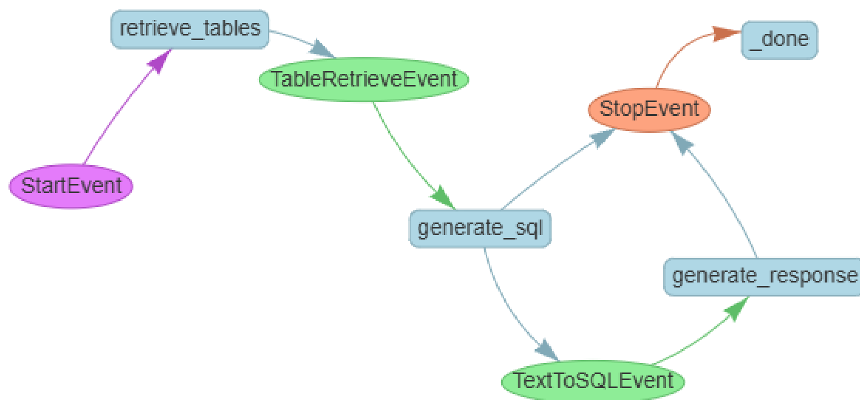

Figura 5.2: Diagrama do TextToSQLWorkflow que representa o fluxo utilizado para implementar da técnica de RAG para o caso onde o banco de dados cadastrado é do tipo *"complete"*.

O `TextToSQLWorkflow` representa a versão para o caso onde o `Database` é do tipo *"complete"* do fluxo de RAG, sendo responsável não apenas pela geração da consulta em SQL, mas também pela execução da mesma em um banco de dados real

e pela construção da resposta final em linguagem natural com base nos resultados obtidos. Essa versão do fluxo é acionada pelo método `starts_workflow`, que recebe como parâmetros as credenciais de acesso ao banco, os nomes das tabelas visíveis, a pergunta do usuário, o tipo de *prompt* a ser utilizado e um indicador sobre a existência de índice vetorial dos objetos de *schema*.

No construtor do *workflow*, são instanciados três componentes principais: `SQLTableRetriever`, responsável por recuperar os esquemas das tabelas; `OpenAIQLGenerator`, que utiliza o modelo LLM para gerar a consulta SQL e também a resposta final; e `SQLRunQuery`, encarregado de executar a consulta no banco real. Além disso, uma conexão `SQLAlchemy` com o banco de dados é criada para permitir a execução direta das consultas.

A execução do fluxo representado pela Figura 5.2 segue três etapas principais, representadas pelos métodos anotados com `@step`:

1. **retrieve_tables (*Step*)**
   A partir do `StartEvent`, que carrega a pergunta original do usuário, este passo invoca o método `retrieve()` do `obj_retriever` para recuperar os objetos de esquemas relevantes. Esses objetos são processados por `_get_table_context_str()`, que consulta o banco de dados para extrair informações detalhadas sobre cada tabela, complementando com descrições adicionais, se disponíveis. O resultado é uma *string* com os contextos das tabelas, empacotada em um `TableRetrieveEvent`.

2. **generate_sql (*Step*)**
   Utiliza o contexto das tabelas e a pergunta do usuário para construir o dicionário de entrada que será passado ao `sql_generator`. Dependendo do tipo de *prompt* selecionado, a geração da consulta pode ser o passo final ou intermediário. Quando o tipo for `TextToSQLPromptStrategy`, retorna-se um `TextToSQLEvent` com a *query* gerada; nos demais casos, o fluxo é encerrado com um `StopEvent`, contendo apenas a consulta.

3. **generate_response (*Step*)**

Essa etapa é executada apenas no caso do tipo `text_to_sql`, e é responsável por consultar o banco de dados real com a instrução SQL gerada. O resultado da execução é passado novamente para o `sql_generator`, que muda dinamicamente sua estratégia de *prompt* para `SynthesisPromptStrategy`, a fim de sintetizar uma resposta em linguagem natural baseada nos dados retornados. O resultado final é encapsulado em um `WorkFlowResult`, contendo tanto a consulta SQL quanto a resposta textual.

Dessa forma, o `TextToSQLWorkflow` implementa um fluxo mais robusto e completo, integrando recuperação de contexto, geração de consultas, execução e síntese da resposta em linguagem natural, sendo ideal para aplicações em ambientes produtivos com acesso ao banco de dados real.

5.3 Interface e Funcionalidades

Como dito anteriormente, um dos objetivos do sistema proposto é desenvolver uma aplicação que facilite usuários que não possuem conhecimento técnico terem acesso aos dados estruturados de um banco de dados. Nesse sentido, o desenvolvimento de uma interface clara e acessível é essencial. Sob essa ótica, iremos detalhar nessa seção as telas desenvolvidas para aplicação bem como uma exemplificação das funcionalidades do sistema.

5.3.1 Cadastro e Autenticação de Usuários

Uma funcionalidade essencial em sistemas web é a capacidade de realizar o cadastro e a autenticação de usuários. Nesse contexto, o sistema proposto permite o registro de usuários do tipo "*employee*", ou seja, funcionários que têm permissão apenas para realizar perguntas. Esse processo é realizado por meio de uma interface simplificada, que solicita apenas nome de usuário, *e-mail* e senha, conforme ilustrado na Figura 5.3. Por outro lado, o cadastro de usuários do tipo "*admin*" é feito externamente, diretamente pela interface de gerenciamento do PGAdmin.

Para realizar o acesso às funcionalidades do sistema é necessário que o usuário esteja autenticado, ou seja, só é possível acessa a aplicação após a realização do *login* que pode ser feito na tela representada pela Figura 5.4. Logo a após o *login* bem sucedido, o usuário é redirecionado para a tela de *homepage* da aplicação, como demonstrado na Figura 5.5.

A *homepage*, assim como todas as telas, possui uma barra lateral com as funcionalidades da aplicação, como o botão de gerar uma nova consulta que redireciona para a própria *homepage*, o botão de Consultas, Bancos de Dados e *Templates*. Essas funcionalidades serão descritas a seguir.

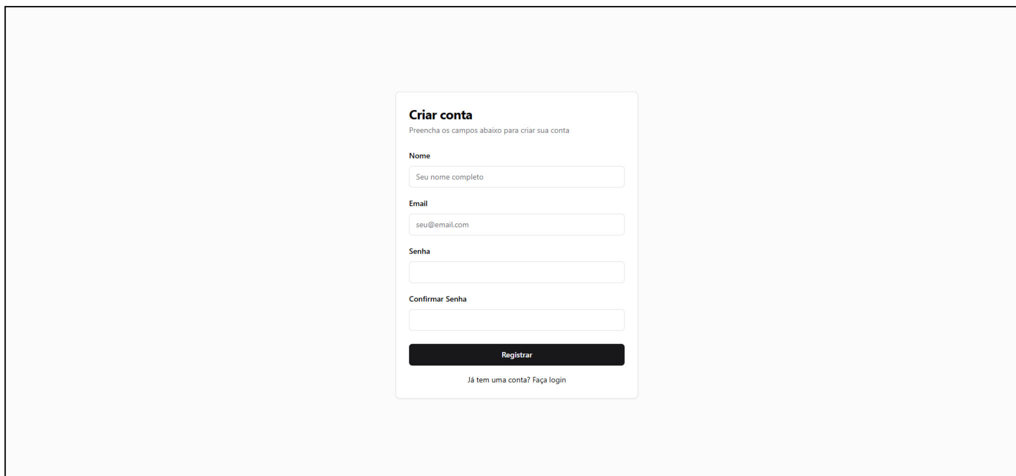

Figura 5.3: Tela de cadastro de usuário do tipo "*employee*".

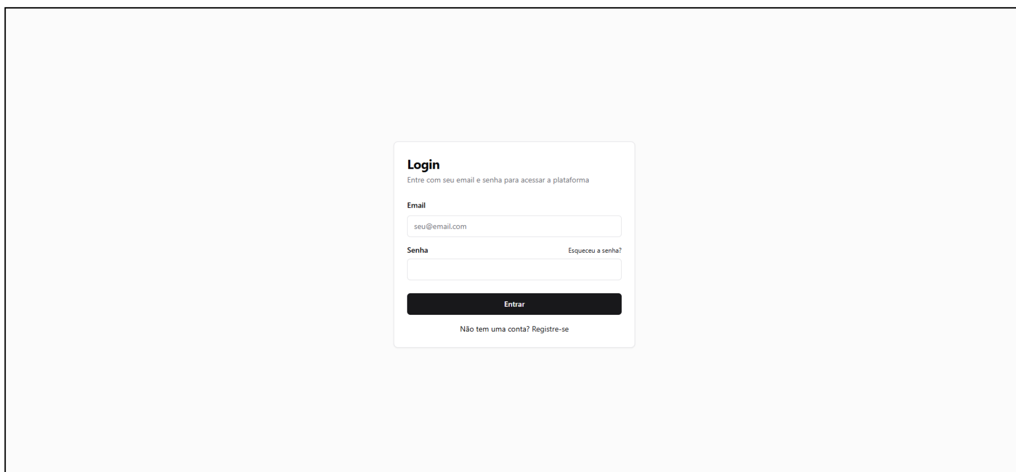

Figura 5.4: Tela de *login*.

Figura 5.5: Tela de *homepage* da aplicação onde é possível iniciar uma nova consulta.

5.3.2 Cadastro de Bancos de Dados

A fim de exemplificar o processo a ser seguido pelo usuário para utilização da aplicação, nesse momento ele deve seguir para a tela de Bancos de Dados, representada pela Figura 5.6. Nessa figura, está representada a visão dessa tela para um usuário do tipo "*admin*" que pode tanto visualizar todos os bancos de dados que ele cadastrou como também pode cadastrar novos bancos de dados. Se o usuário fosse do tipo "*employee*" nessa tela seriam exibido para ele apenas a visualização dos bancos de dados que ele tem permissão de realizar perguntas.

Ao iniciar o cadastro de um novo banco de dados é solicitado o nome do banco de dados desejado, como visto na Figura 5.7 se o usuário desejar realizar a conexão com o seu banco de dados é necessário que o nome seja correspondente ao de seu banco real. Logo após isso, o usuário deve selecionar o tipo de banco de dados que ele deseja cadastrar, a interface representada na Figura 5.8 apresenta um texto explicativo sobre os tipos de bancos.

Caso o usuário tenha selecionado a opção "Conexão Direta com o Banco de Dados", ele irá cadastrar um banco do tipo "*complete*", para isso ele precisará informar os dados para realizar essa conexão, como demonstrado na Figura 5.9, após isso essas informações serão armazenadas em nosso banco, permitindo o registro dos esquemas das tabelas do usuário e também a conexão para execução de consultas SQL.

Caso o usuário tenha selecionado cadastrar um banco de dados na opção de

"Fonecer Esquemas", ele irá cadastrar um banco de dados do tipo "*minimal*". Para isso o usuário precisa apenas informar os esquemas das tabelas do seu banco na tela representada pela Figura 5.10. Para facilitar esse processo, a interface fornece instruções de como executar um *script* no banco de dados PostgreSQL que extrai automaticamente um JSON contendo os esquemas de todas as tabelas do banco. Após isso, o banco e os esquemas das tabelas são cadastrados.

Além disso, após o cadastro, o usuário "*admin*" pode distribuir as permissões para os usuários que poderão ter acesso a cada banco cadastrado. Para isso, ele deve selecionar a opção "Gerenciar Usuários" de um banco, como visto na Figura 5.6. Assim, o usuário será redirecionado para a tela da Figura 5.11 onde ele poderá adicionar ou remover o acesso de usuários do tipo "*employee*" a um banco de dados.

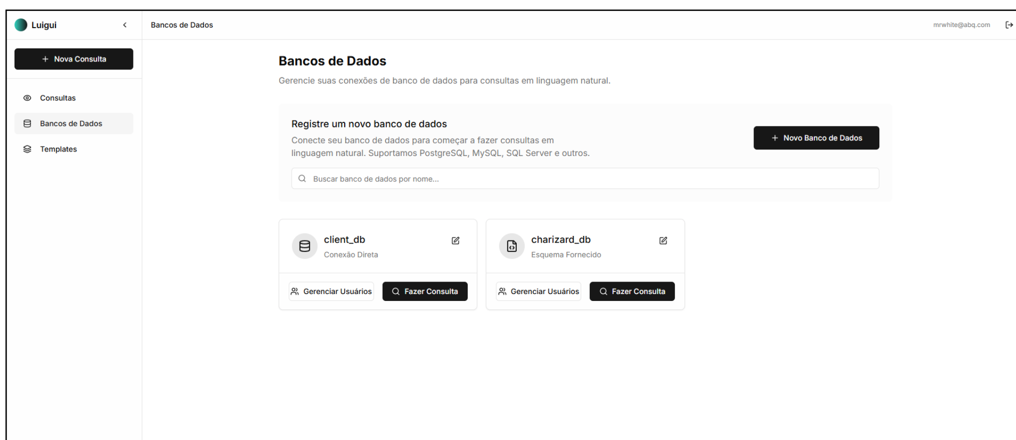

Figura 5.6: Tela de Bancos de Dados na visão do "*admin*", onde é possível visualizar os bancos de dados cadastrados e também realizar novos cadastros.

5.3.3 Cadastro de Esquemas

Depois de um banco de dados ser cadastrado, podemos realizar o cadastro dos esquemas das tabelas desse banco. Para realizar o cadastro, é necessário selecionar um banco de dados listado na tela representada na Figura 5.6. Após isso o usuário será redirecionado para a tela onde são listadas quais as tabelas já foram cadastradas e tiveram seus esquemas transformados em *embedding* e armazenados na base de dados vetorial, como visto na Figura 5.12.

Para cadastrar uma tabela em um banco de dados do tipo "*complete*" o usuário

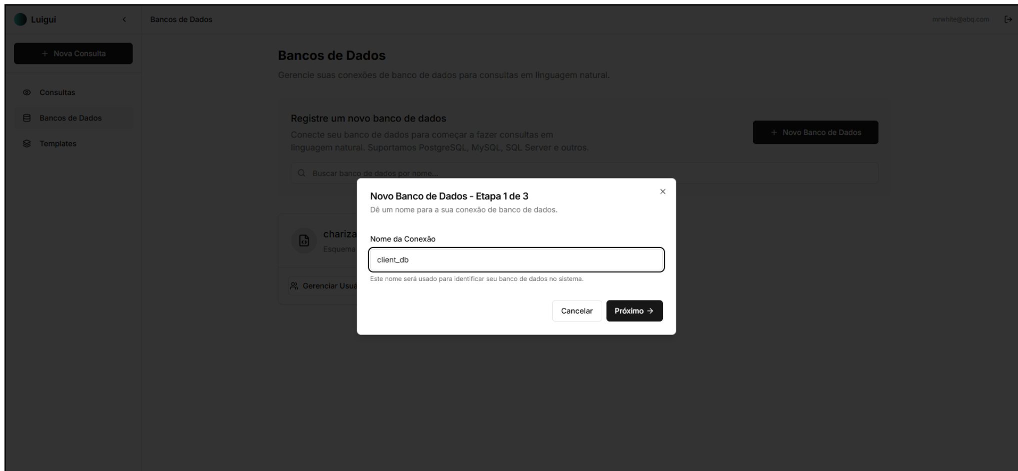

Figura 5.7: Tela da etapa 1 do cadastro de banco de dados onde o usuário precisar informar o nome da conexão com o banco.

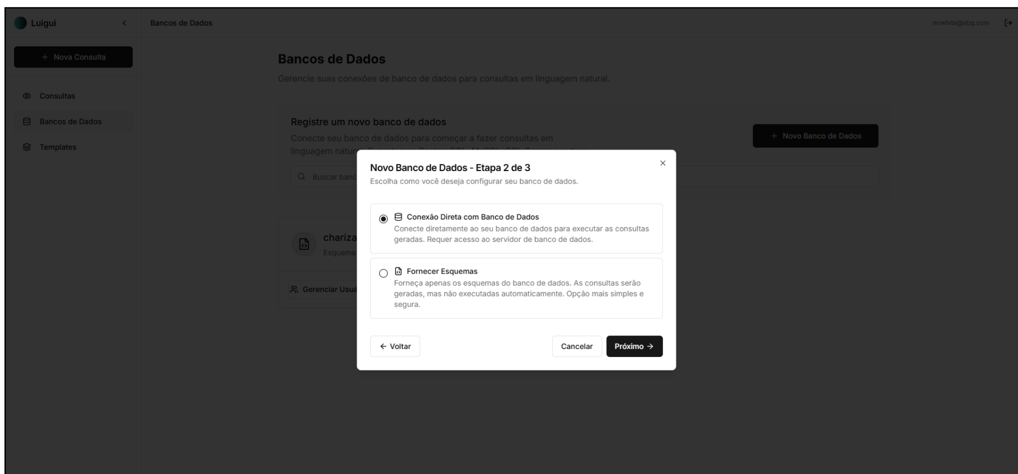

Figura 5.8: Tela da etapa 2 do cadastro de banco de dados onde o usuário precisar selecionar o tipo de banco a ser cadastrado.

precisa apenas informar o nome da tabela e a senha do banco, como visto na Figura. Com essas informações o sistema consegue realizar a conexão direta com o banco do usuário e extrair o esquema da tabela desejada. Caso o banco de dados seja do tipo "*minimal*" o usuário pode cadastrar uma tabela da mesma forma como vista anteriormente, apenas informando um JSON extraído com o comando visto na Figura 5.10.

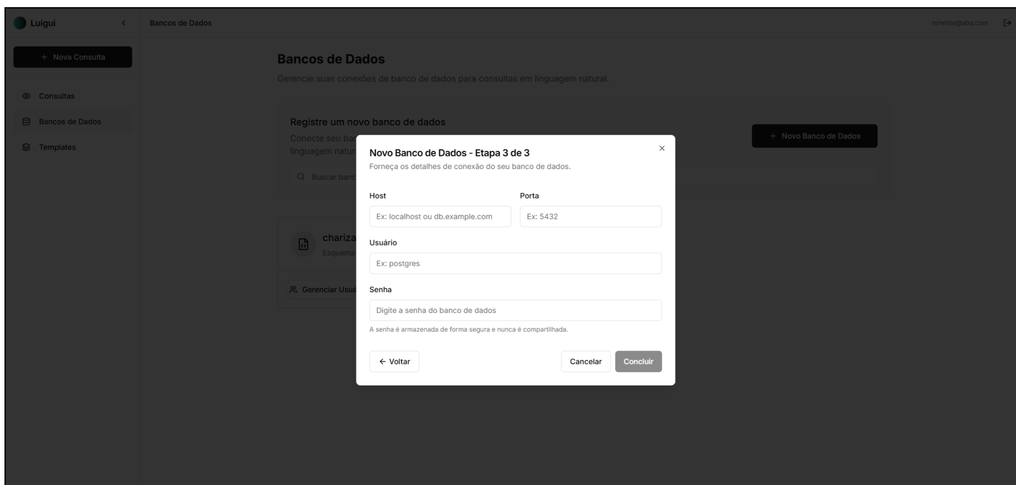

Figura 5.9: Tela da etapa 3 do cadastro de banco de dados do tipo "*complete*" onde o usuário precisa informar os dados para realizar a conexão com o seu banco.

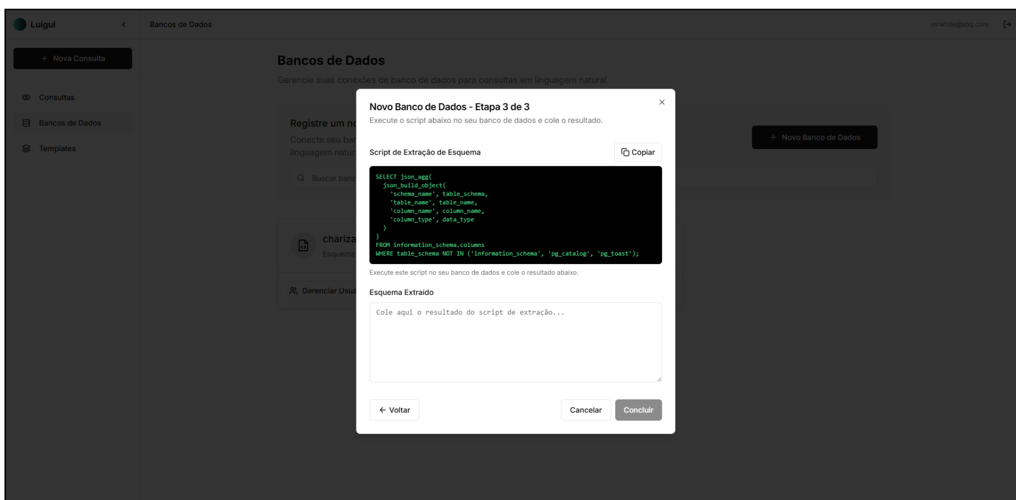

Figura 5.10: Tela da etapa 3 do cadastro de banco de dados do tipo "*minimal*" com as instruções de como o usuário pode extrair os esquemas das suas tabelas.

5.3.4 Produção de Consultas

Após o as tabelas serem cadastradas por um usuário "*admin*" é possível começar realizar as perguntas. Esse processo poder ser realizado pelos dois tipos de usuários. Para isso, basta clicar no botão "Nova Consulta" localizado na barra lateral das páginas. Após clicar o usuário é redirecionado para a *homepage*, como na Figura 5.5. Para iniciar o processo, o usuário deve selecionar a opção "Selecionar Banco de Dados", e será listado os bancos de dados disponíveis para ele selecionar, como mostra a Figura 5.14.

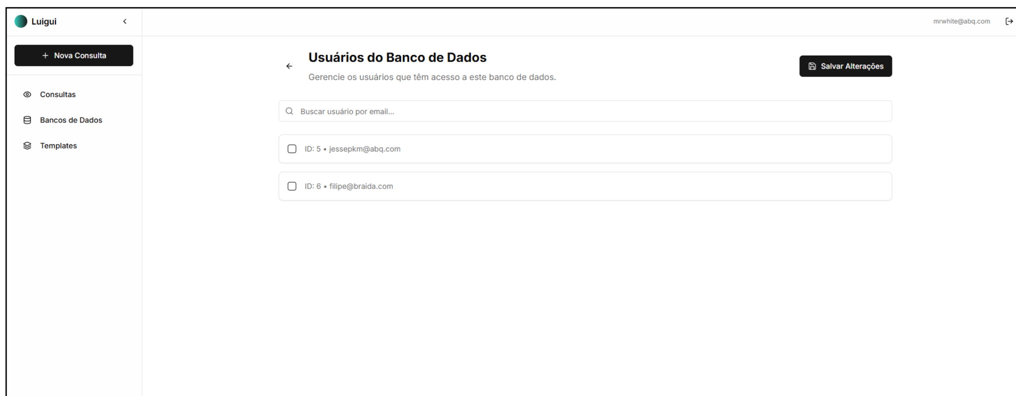

Figura 5.11: Tela de gerenciamento dos usuário que tem acesso a um banco de dados específico.

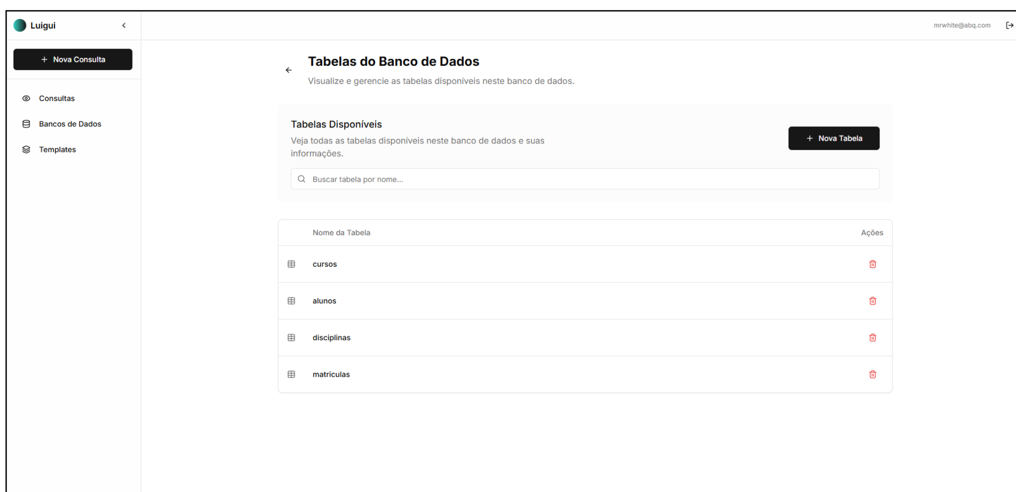

Figura 5.12: Tela que permite o cadastro de esquemas das tabelas e exibe quais as tabelas já tiveram seus esquemas armazenados na base de dados vetorial.

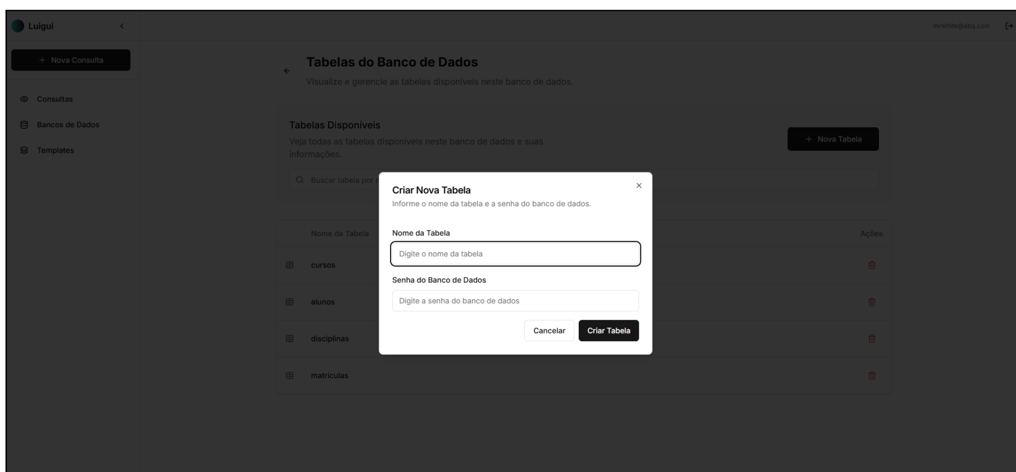

Figura 5.13: Tela que permite o cadastro de tabela em um banco de dados do tipo "*complete*".

Depois de selecionado o banco de dados, ele será redirecionado para a tela onde poderá realizar as perguntas, representada pela Figura 5.15. Nessa tela é possível trocar o banco de dados onde serão realizadas as recuperação das tabelas para geração das respostas. Além disso, é possível selecionar o tipo de *prompt* que será utilizada na geração da resposta, como visto na Figura 5.16.

Na Figura 5.17 está representada a geração de resposta para o tipo de *prompt* "Gerar Consulta", esse tipo de *prompt* é responsável por converter a pergunta em linguagem natural do usuário em uma consulta SQL. No exemplo dessa figura, o usuário fez a pergunta "Quais alunos estão matriculados em Estruturas de Dados?", então foi iniciado o `TextToSQLWorkflow` descrito na seção anterior que gerou como resposta consulta do Código 5.2 além de resposta em linguagem natural. Essa resposta em linguagem natural foi gerada pelo LLM e contém as informações que foram retornadas pela execução da consulta no banco de dados do usuário.

```
1 SELECT alunos.nome FROM alunos JOIN matriculas ON alunos.id =
    matriculas.aluno_id JOIN disciplinas ON matriculas.
    disciplina_id = disciplinas.id WHERE disciplinas.nome = '
    Estruturas de Dados' ORDER BY alunos.nome;
```

Código 5.2: Exemplo de consulta SQL gerada para a pergunta "Quais alunos estão matriculados em Estruturas de Dados?".

```
1 SELECT AVG(matriculas.nota) as media_nota FROM matriculas JOIN
     disciplinas ON matriculas.disciplina_id = disciplinas.id
    WHERE disciplinas.nome = 'Algoritmos'
```

Código 5.3: Exemplo de consulta SQL gerada para a pergunta "Qual a média de nota para a disciplina de Algoritmos?".

Outro exemplo para o caso de geração de consulta SQL é demonstrado na Figura 5.18. Nesse caso, o usuário realizou a pergunta "Qual é a média de nota da disciplina Algoritmos?", então ele obteve como resposta a consulta SQL do Código 5.3 e uma resposta em linguagem natural com o resultado da execução da consulta.

Um exemplo para a opção de "Otimizar Consulta" está representada na Fi-

gura 5.19. Nesse caso, o usuário informa a consulta que ele deseja otimizar então o *workflow* é iniciado e os esquemas retornados são usados como base para gerar a consulta otimizada, além de uma resposta em linguagem natural com a explicação da otimização.

Já para a opção de "Explicar Consulta, podemos ver uma representação da resposta gerada para esse caso na Figura 5.20. Nesse caso, o usuário envia uma consulta SQL e então é gerada uma resposta com base nos esquemas recuperados contendo uma explicação detalhada sobre a consulta solicitada,

Por fim, a opção de "Corrigir Consulta" está representada na Figura 5.21. Nesse cenário, o usuário envia a consulta que está com algum problema ou erro, então ele recebe como resposta a consulta corrigida e uma explicação em linguagem natural sobre o erro encontrado na consulta. Além disso, o sistema permite exibir o histórico de todas as perguntas e respostas feitas em cada banco de dados. Para acessar essa opção, basta selecionar o botão "Consultas" na barra lateral, então o usuário será redirecionado para a tela representada na Figura 5.22.

Figura 5.14: *Homepage* exibindo lista de bancos de dados para disponíveis para seleção para iniciar o processo de perguntas.

Figura 5.15: Tela que permite a realização de perguntas pelo usuário com o tipo de pergunta selecionado como "Gerar Consulta".

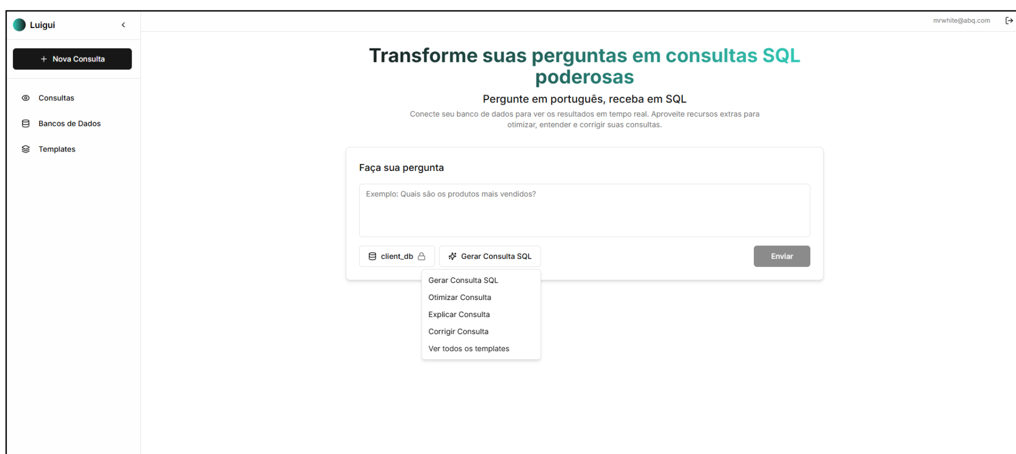

Figura 5.16: Tela que permite a realização de perguntas pelo usuário com a opção de escolher *prompt* selecionada. A opções disponíveis para escolha são: "Gerar Consulta", "Otimizar Consulta", "Explicar Consulta" e "Corrigir Consulta".

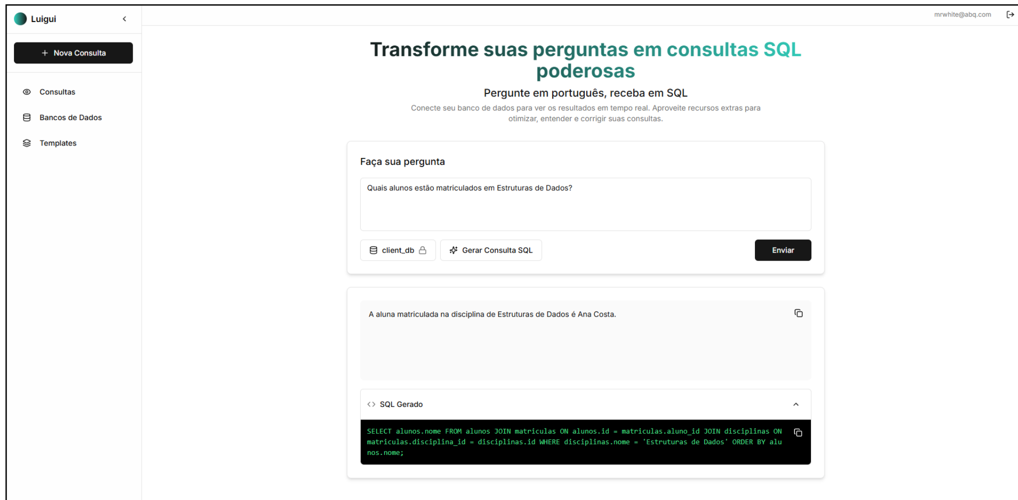

Figura 5.17: Tela com a resposta em linguagem natural e a consulta SQL equivalente gerada para a pergunta “Quais alunos estão matriculados em Estruturas de Dados?”.

Figura 5.18: Tela com a resposta em linguagem natural e a consulta SQL equivalente gerada para a pergunta “Qual é a média de nota da disciplina Algoritmos?”.

Figura 5.19: Tela da opção "Otimizar Consulta" exibindo a consulta otimizada, além de uma explicação em linguagem natural da otimização feita.

Figura 5.20: Tela da opção "Explicar Consulta" exibindo uma explicação em linguagem natural da consulta enviada.

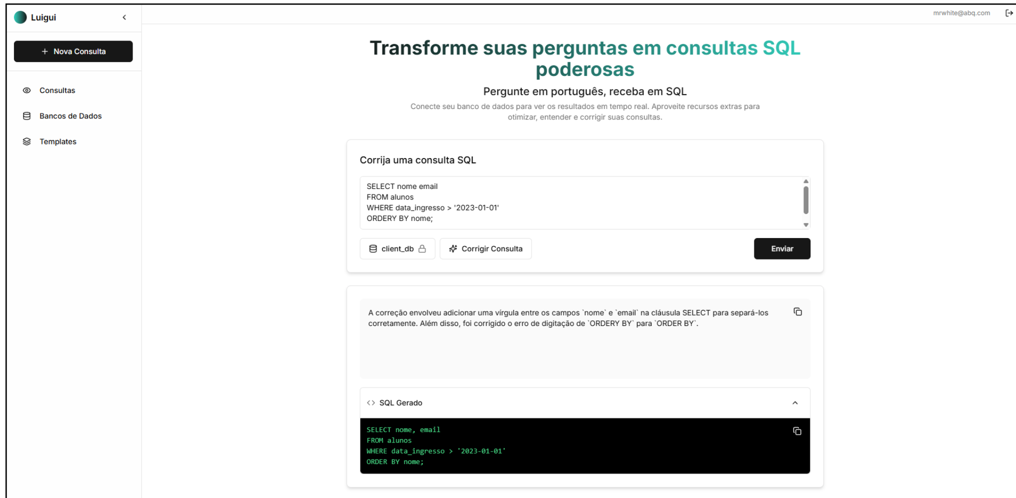

Figura 5.21: Tela da opção “Corrigir Consulta” exibindo a consulta corrigida e uma explicação em linguagem natural da correção feita.

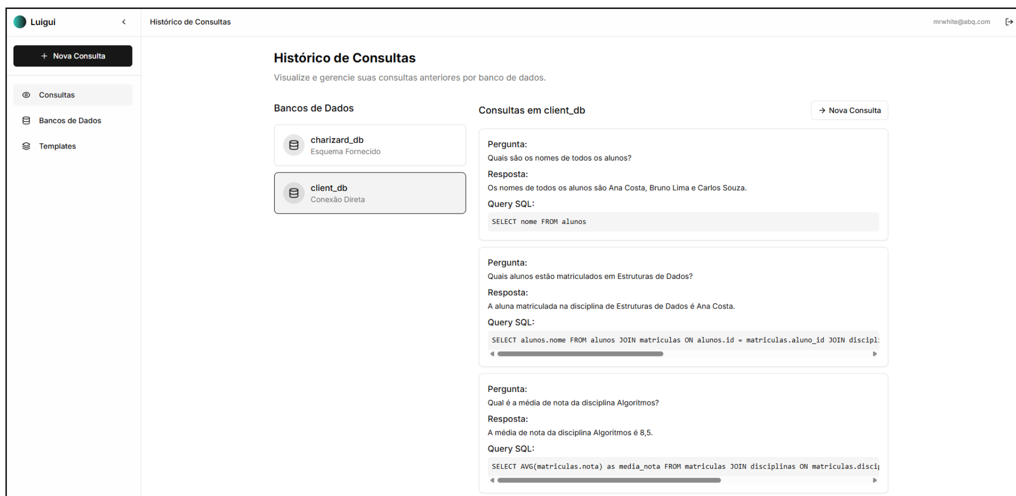

Figura 5.22: Tela que exibe o histórico de perguntas e respostas para cada banco de dados do usuário.

Capítulo 6

Conclusão

"Parte da jornada é o fim."

— Anthony Stark

Este capítulo apresenta as considerações finais do trabalho. São discutidas as contribuições do sistema implementado no contexto da geração automática de consultas SQL a partir de linguagem natural, bem como os impactos do uso de LLMs e da técnica de RAG nesse processo. Em seguida, são abordadas as limitações identificadas durante o desenvolvimento e a execução da aplicação, além de serem sugeridas possíveis direções para trabalhos futuros que possam aprimorar e expandir a solução desenvolvida.

6.1 Considerações finais

Neste trabalho, foi desenvolvido um sistema *web* de Text-To-SQL que combina o poder dos LLMs com a técnica de RAG, com o objetivo de facilitar o acesso a bancos de dados relacionais por usuários com pouco ou nenhum conhecimento técnico em SQL. A solução proposta permite que os usuários conectem seus próprios bancos de dados à plataforma e realizem consultas em linguagem natural, as quais são transformadas em instruções SQL equivalentes e, assim, executadas sobre o banco de dados.

Além disso, o sistema também oferece suporte para usuários que já possuem conhecimento em SQL, permitindo que esses cadastrem bancos de dados e tabelas, e concedam acesso a outros usuários da organização. Dessa forma, colaboradores leigos podem explorar e consultar os dados diretamente, sem a necessidade de depender de um intermediador com conhecimento em SQL para realizar as consultas por elas.

A implementação deste sistema possibilitou a exploração do potencial dos LLMs na tarefa de Text-To-SQL, confirmando sua capacidade de compreender linguagem natural e gerar instruções precisas em diferentes domínios. Adicionalmente, pudemos também observar como a técnica de RAG é uma grande ferramenta para aumentar a capacidade de fornecer contextos relevantes para a geração de resposta pelos LLMs. Tais ferramentas revelam grande potencial para o desenvolvimento de soluções que visam à democratização do acesso a informações e à automação de tarefas complexas de forma intuitiva.

6.2 Limitações e trabalhos futuros

Uma das limitações do sistema atual está relacionada ao cadastro de tabelas no caso em que o usuário opta por utilizar um banco de dados do tipo "*complete*". Nessa modalidade, o usuário precisa informar manualmente o nome das tabelas que deseja cadastrar, ainda que os dados de acesso ao banco estejam disponíveis. Uma melhoria futura seria implementar a extração automática das tabelas do banco, sem a necessidade que o usuário informe manualmente o nome de cada uma delas.

Outra limitação refere-se ao suporte exclusivo ao PostgreSQL. Esta escolha foi motivada pela conveniência de trabalhar com um único sistema gerenciador de banco de dados durante o desenvolvimento. No entanto, trabalhos futuros podem expandir a aplicação para suportar outros bancos relacionais, como MySQL, SQLite e MariaDB. Além disso, seria interessante investigar a adaptação do sistema para bases de dados NoSQL, como o MongoDB.

Além disso, em nossa aplicação, o processo de recuperação semântica implementado na arquitetura de RAG depende da seleção prévia do banco de dados por parte

do usuário. Embora a identificação da tabela apropriada seja feita automaticamente durante a recuperação, um avanço possível seria permitir que o sistema identificasse, de forma autônoma, o banco de dados mais relevante com base na pergunta em linguagem natural, tornando o processo ainda mais fluido para o usuário.

Por fim, limitação deste trabalho é a ausência de uma avaliação prática da solução desenvolvida. A implementação concentrou-se em aplicar as estratégias encontradas na literatura para a construção do sistema, porém não foram realizados testes em cenários ampliados ou experimentos controlados para atestar a qualidade e a robustez das consultas geradas. Avaliar a qualidade das consultas SQL geradas por sistemas Text-to-SQL representa um desafio, pois diferentes consultas podem produzir o mesmo resultado. Uma alternativa viável para superar essa dificuldade seria definir previamente conjuntos de respostas esperadas a partir de um banco de dados de teste e verificar se as consultas geradas pelo sistema retornam os valores corretos. Portanto, um trabalho futuro seria a realização de experimentos estruturados que avaliem tanto a precisão quanto a utilidade prática das consultas geradas, especialmente em ambientes com maior volume e diversidade de dados.

Referências

ALMEIDA, F.; XEXÉO, G. Word embeddings: A survey. *arXiv preprint arXiv:1901.09069*, 2019.

BENDER, E. M. et al. On the dangers of stochastic parrots: Can language models be too big? In: *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency (FAccT)*. [S.l.]: ACM, 2021. p. 610–623.

BENGIO, Y. et al. A neural probabilistic language model. *Journal of Machine Learning Research*, v. 3, p. 1137–1155, 2003.

BIRD, S.; KLEIN, E.; LOPER, E. *Natural Language Processing with Python*. 1st. ed. [S.l.]: O'Reilly Media, Inc., 2009.

BROWN, T. et al. Language models are few-shot learners. *Advances in neural information processing systems*, v. 33, p. 1877–1901, 2020.

CAMBRIA, E.; WHITE, B. Jumping nlp curves: A review of natural language processing research. *IEEE Computational intelligence magazine*, IEEE, v. 9, n. 2, p. 48–57, 2014.

CHAMBERLIN, D. D. Early history of sql. *IEEE Annals of the History of Computing*, IEEE, v. 34, n. 4, p. 78–82, 2012.

CHANG, K. et al. Efficient prompting methods for large language models: A survey. *arXiv preprint arXiv:2404.01077*, 2024.

CHENG, M. et al. A survey on knowledge-oriented retrieval-augmented generation. *arXiv preprint arXiv:2503.10677*, 2025.

DEVLIN, J. et al. Bert: Pre-training of deep bidirectional transformers for language understanding. In: *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*. [S.l.: s.n.], 2019. p. 4171–4186.

ELMASRI, R. et al. *Sistemas de banco de dados*. [S.l.]: Pearson Addison Wesley São Paulo, 2005.

FARIHA, A. et al. Example-driven user intent discovery: Empowering users to cross the sql barrier through query by example. *arXiv preprint arXiv:2012.14800*, 2020.

GAO, Y. et al. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*, v. 2, n. 1, 2023.

GOMES, D. d. S. Inteligência artificial: Conceitos e aplicações. *Revista Olhar Científico*, 2010.

GUO, C. et al. Prompting gpt-3.5 for text-to-sql with de-semanticization and skeleton retrieval. In: SPRINGER. *Pacific Rim International Conference on Artificial Intelligence*. [S.l.], 2023. p. 262–274.

HONG, Z. et al. Next-generation database interfaces: A survey of llm-based text-to-sql. *arXiv preprint arXiv:2406.08426*, 2024.

JELINEK, F. *Statistical methods for speech recognition*. [S.l.]: MIT press, 1998.

KIM, H. et al. Natural language to sql: Where are we today? *Proceedings of the VLDB Endowment*, VLDB Endowment, v. 13, n. 10, p. 1737–1750, 2020.

LIBKIN, L. Expressive power of sql. *Theoretical Computer Science*, Elsevier, v. 296, n. 3, p. 379–404, 2003.

MANNING, C.; SCHUTZE, H. *Foundations of statistical natural language processing*. [S.l.]: MIT press, 1999.

MOREIRA, A. Processamento de linguagem natural aplicado à inteligência artificial. *Revista UiLPS*, 2021.

RADFORD, A. et al. *Improving Language Understanding by Generative Pre-Training*. San Francisco, CA, USA, 2018. Preprint.

RADFORD, A. et al. Language models are unsupervised multitask learners. *OpenAI blog*, v. 1, n. 8, p. 9, 2019.

RAO, C.; GUDIVADA, V. N. *Computational analysis and understanding of natural languages: principles, methods and applications*. [S.l.]: Elsevier, 2018. v. 38.

REYNOLDS, L.; MCDONELL, K. Prompt programming for large language models: Beyond the few-shot paradigm. In: *Extended abstracts of the 2021 CHI conference on human factors in computing systems*. [S.l.: s.n.], 2021. p. 1–7.

SHARMA, A. et al. Database management systems—an efficient, effective, and augmented approach for organizations. In: *ICT with Intelligent Applications: Proceedings of ICTIS 2021, Volume 1*. [S.l.]: Springer, 2021. p. 465–478.

SHI, L. et al. A survey on employing large language models for text-to-sql tasks. *arXiv preprint arXiv:2407.15186*, 2024.

SUTSKEVER, I.; VINYALS, O.; LE, Q. V. Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, v. 27, 2014.

VASWANI, A. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.

WEI, J. et al. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682*, 2022.

ZELLE, J. M.; MOONEY, R. J. Learning to parse database queries using inductive logic programming. In: *Proceedings of the national conference on artificial intelligence*. [S.l.: s.n.], 1996. p. 1050–1055.

ZHANG, A. et al. *Dive into deep learning*. [S.l.]: Cambridge University Press, 2023.

ZHANG, Y. et al. Siren's song in the ai ocean: a survey on hallucination in large language models. *arXiv preprint arXiv:2309.01219*, 2023.

ZHAO, W. X. et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 2023.

ZIEMS, N. et al. Large language models are built-in autoregressive search engines. In: *Findings of the Association for Computational Linguistics: ACL 2023*. Toronto, Canada: Association for Computational Linguistics, 2023. p. 2666–2678. Disponível em: <https://aclanthology.org/2023.findings-acl.167/>.