Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Центр післядипломної освіти
(повна назва)
Кафедра Штучного інтелекту
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Методи генерації SQL-запитів з тексту за допомогою великих мовних моделей
(тема)

Виконав:
здобувач другого року навчання,
групи СШІздм-24-1
Світлана ДВОРЯНКІНА
(власне ім'я, прізвище)

Спеціальність 122 Комп'ютерні науки
(код і повна назва спеціальності)
Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)
Освітня програма Системи штучного інтелекту
(повна назва освітньої програми)

Керівник проф. Лариса ЧАЛА
(посада, власне ім'я, прізвище)

Допускається до захисту

Завідувач кафедри ШІ _____ Лариса ЧАЛА
(підпис) (власне ім'я, прізвище)

2026 р.

Харківський національний університет радіоелектроніки

Факультет Центр післядипломної освіти

Кафедра Штучного інтелекту

Рівень вищої освіти другий (магістерський)

Спеціальність 122 Комп'ютерні науки
(код і повна назва)

Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Освітня програма Системи штучного інтелекту
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«_____» _____ 20 ___ р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Дворянкіній Світлані Євгеніївні
(прізвище, ім'я, по батькові)

1. Тема роботи Методи генерації SQL-запитів з тексту за допомогою великих мовних моделей

затверджена наказом університету від 6 квітня 20 26 р. № 48Стз

2. Термін подання здобувачем роботи до екзаменаційної комісії 26 травня 20 26 р.

3. Вихідні дані до роботи великомасштабна реляційна база даних PostgreSQL; схема бази даних у вигляді SQL DDL-визначень; набір тестових запитань природною мовою; результати генерації SQL-запитів базовим та оптимізованим Text-to-SQL пайплайном; сучасні наукові джерела з тематики Text-to-SQL, великих мовних моделей, RAG та динаміного скорочення схеми.

4. Перелік питань, що потрібно опрацювати в роботі

1) Аналіз предметної галузі Text-to-SQL

2) Проєктування Text-to-SQL пайплайну

3) Реалізація модулів індексації схеми, вибору релевантних таблиць, генерації та валідації SQL-запитів

4) Експериментальне порівняння базового й оптимізованого підходів; аналіз результатів

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Строк / термін виконання етапів роботи	Примітка
1	Отримання завдання на кваліфікаційну роботу	06.04.2026	виконано
2	Проведення початкового аналізу теми та	06.04.2026 – 09.04.2026	виконано
	визначення основних напрямків дослідження		
3	Огляд технологій пошуку у великих базах даних	09.04.2026 – 13.04.2026	виконано
	та методів Text-to-SQL		
4	Аналіз предметної галузі та	13.04.2026 – 20.04.2026	виконано
	формулювання постановки задачі		
5	Проєктування Text-to-SQL пайплайну з	20.04.2026 – 01.05.2026	виконано
	динамічним скороченням схеми		
6	Реалізація модулів витягу схеми, індексації	01.05.2026 – 05.05.2026	виконано
	та динамічного скорочення схеми		
7	Реалізація генерації, валідації та	05.05.2026 – 06.05.2026	виконано
	безпечного виконання SQL-запитів		
8	Проведення експериментальної оцінки	06.05.2026 – 07.05.2026	виконано
	запропонованого підходу		
9	Аналіз результатів експериментів та	07.05.2026	виконано
	формування висновків		
10	Оформлення пояснювальної записки	07.05.2026 – 14.04	виконано
11	Нормоконтроль	15.05.2026	
12	Захист кваліфікаційної роботи	26.05.2026	

Дата видачі завдання 6 квітня 20 26 р.

Здобувач _____
(підпис)

Керівник роботи _____ проф. Лариса ЧАЛА
(підпис) (посада, власне ім’я, прізвище)

РЕФЕРАТ

Пояснювальна записка: 74 с., 20 рис., 2 табл., 1 дод., 35 джерел.

ГЕНЕРАЦІЯ SQL-ЗАПИТІВ, ДИНАМІЧНЕ СКОРОЧЕННЯ СХЕМИ, ПРИРОДНА МОВА, РЕЛЯЦІЙНА БАЗА ДАНИХ, API, LLM, POSTGRESQL, RAG, SQL, TEXT-TO-SQL.

Об'єкт дослідження – процес автоматизованої генерації SQL-запитів на основі запитів природною мовою.

Предмет дослідження – методи оптимізації Text-to-SQL пайплайнів для великомасштабних реляційних баз даних із використанням динамічного скорочення схеми.

Мета роботи – розробка та оцінка Text-to-SQL пайплайну, що зменшує затримку генерації SQL-запитів і підвищує їхню коректність шляхом динамічного скорочення схеми бази даних перед передачею контексту до великої мовної моделі.

Методи дослідження – аналіз сучасних підходів Text-to-SQL, великі мовні моделі, динамічне скорочення схеми, побудова індексу схеми, prompt engineering, експериментальне порівняння базового та оптимізованого пайплайнів, ручна перевірка коректності SQL-запитів у pgAdmin.

У кваліфікаційній роботі досліджено проблему автоматизованої генерації SQL-запитів з природної мови для великомасштабних реляційних баз даних та запропоновано пайплайн із динамічним скороченням схеми, що включає індексацію схеми, вибір релевантних таблиць для кожного запиту, формування скороченого контексту й механізми валідації та безпечного виконання read-only SQL-запитів. Експериментальна оцінка на наборі з понад 200 тестових запитань у 9 категоріях підтвердила ефективність підходу: середній час генерації скоротився з 5,4621 с до 0,5514 с (на 86,75%), а частка успішно згенерованих запитів зросла з 51,77% до 69,85%.

ABSTRACT

Master’s thesis contains: 74 pp., 20 fig., 2 tabl., 1 ann., 35 references.

API, DYNAMIC SCHEMA PRUNING, LLM, NATURAL LANGUAGE, POSTGRESQL, RAG, RELATIONAL DATABASE, SQL, SQL QUERY GENERATION, TEXT-TO-SQL.

Object of research – the process of automated SQL query generation based on natural language user requests.

Subject of research – methods for optimizing Text-to-SQL pipelines for large-scale relational databases using dynamic schema pruning.

Purpose of the work – to develop and evaluate a Text-to-SQL pipeline that reduces SQL query generation latency and improves query correctness by dynamically pruning the database schema before passing the context to a large language model.

Research methods – analysis of modern Text-to-SQL approaches, large language models, dynamic schema pruning, schema index construction, prompt engineering, experimental comparison of baseline and optimized pipelines, and manual verification of generated SQL queries using pgAdmin.

This thesis investigates automated SQL query generation from natural language for large-scale relational databases and proposes a pipeline with dynamic schema pruning, comprising schema indexing, per-query relevant table selection, pruned context construction, and mechanisms for SQL validation and safe read-only execution. Experimental evaluation on over 200 test questions across 9 categories confirmed the approach's effectiveness: average generation time decreased from 5.4621 s to 0.5514 s (86.75% reduction), and the successful SQL generation rate increased from 51.77% to 69.85%.

ЗМІСТ

Перелік умовних позначень, символів, одиниць, скорочень і термінів 8
Вступ 10
1 Аналіз предметної галузі 12
1.1 Опис предметної галузі 12
1.2 Методи генерації SQL-запитів у великомасштабних базах даних 15
1.2.1 Генерація з розширеним пошуком та динамічним скороченням схеми 15
1.2.2 Багатоагентні та ансамблеві архітектури 17
1.2.3 Масштабування під час тестування та самокоригування на основі результатів виконання 19
1.3 Постановка задачі 21
2 Проєктування пайплайну Text-to-SQL з динамічним скороченням схеми 23
2.1 Системні вимоги 23
2.1.1 Функціональні вимоги 24
2.1.2 Нефункціональні вимоги 25
2.2 Загальна архітектура запланованого пайплайну Text-to-SQL 26
2.3 Базовий підхід: використання підказок на основі повної схеми 28
2.4 Рекомендований підхід: динамічне скорочення схеми 32
2.4.1 Схема індексації 34
2.4.2 Вибір відповідної таблиці 36
2.4.3 Цілеспрямоване введення схеми 37
2.4.4 Створення підказок із використанням скороченої схеми контексту 40
2.5 Процес генерації, перевірки та безпечного виконання SQL-запитів. 43
3 Реалізація системи Text-to-SQL 47
3.1 Технологічний стек та середовище розробки 47
3.2 Витяг схеми бази даних та побудова індексів 49

3.3 Реалізація динамічного скорочення схеми 52
3.4 Побудова промптів, генерація SQL, валідація та обробка отриманих результатів 59
4 Експериментальна оцінка запропонованого підходу 63
4.1 Налаштування оцінювання 63
4.2 Конфігурації порівняння 64
4.3 Результати експерименту та їх аналіз 65
Висновки 68
Перелік джерел посилання 70
Додаток А Відомість кваліфікаційної роботи 74

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ШІ – штучний інтелект;

AI – Artificial Intelligence – штучний інтелект;

API – Application Programming Interface – програмний інтерфейс прикладного програмування;

DDL – Data Definition Language – мова визначення даних, що використовується для опису структури бази даних;

Dynamic schema pruning – динамічне скорочення схеми, підхід, за якого релевантні таблиці та фрагменти схеми вибираються окремо для кожного запиту користувача;

Latency – затримка, час між надсиланням запиту користувачем і отриманням відповіді від системи;

LLM – Large Language Model – велика мовна модель;

NLP – Natural Language Processing – обробка природної мови;

NLIDB – Natural Language Interface to Databases – інтерфейс природної мови до баз даних;

Prompt – текстова інструкція або вхідний контекст, що передається великій мовній моделі для генерації відповіді;

Prompt engineering – процес проєктування та оптимізації промптів для отримання якісніших і контрольованіших результатів від мовної моделі;

RAG – Retrieval-Augmented Generation – генерація з доповненням пошуком, підхід, за якого модель використовує зовнішні релевантні дані під час формування відповіді;

RDBMS – Relational Database Management System – система керування реляційними базами даних;

Read-only SQL query – SQL-запит, який лише читає дані з бази даних і не змінює її стан;

Schema linking – зв’язування схеми, процес зіставлення понять із запиту природною мовою з відповідними таблицями, стовпцями та зв’язками бази даних;

Schema pruning – скорочення схеми, процес вибору лише релевантних елементів схеми бази даних перед генерацією SQL-запиту;

SQL – Structured Query Language – мова структурованих запитів для роботи з реляційними базами даних;

Text-to-SQL – задача автоматизованого перетворення запиту природною мовою у SQL-запит.

ВСТУП

У час стрімкої цифрової трансформації компанії в різних галузях дедалі частіше інтегрують штучний інтелект (ШІ) у свої ключові операційні процеси для зниження витрат і збереження конкурентної переваги. Стикаючись зі зростаючим ринковим тиском, підприємства перестали розглядати ШІ просто як нову технологію, він перетворився на стратегічний інструмент бізнесу. Проте основою оптимізації будь-яких процесів залишається здатність швидко отримувати доступ, обробляти та інтерпретувати великі масиви даних, які є фундаментом для прийняття обгрунтованих управлінських рішень.

Незважаючи на впровадження інновацій, на стику між бізнес-користувачами та корпоративною інфраструктурою даних зберігається фундаментальне «вузьке місце». Домінуючою парадигмою зберігання структурованої інформації залишаються реляційні бази даних, взаємодія з якими вимагає знання мови запитів SQL. Бізнес-аналітики, операційні менеджери та галузеві експерти ті, хто безпосередньо ухвалює рішення, часто змушені покладатися на дата-інженерів для отримання потрібної аналітики. Це спричиняє затримки, комунікаційні бар'єри та організаційну неефективність, що нівелює ту гнучкість, заради якої впроваджується ШІ.

Останні досягнення в галузі обробки природної мови відкрили нові можливості для подолання цього розриву. Системи перетворення природної мови на SQL-запити (Text-to-SQL) стали перспективним напрямом, оскільки дозволяють нетехнічним користувачам отримувати та аналізувати дані без знання спеціалізованих мов програмування. Проте застосування таких систем до масштабу й складності реальних корпоративних баз даних, що налічують сотні таблиць і мільйони записів, залишається суттєвим викликом і потребує ґрунтовного наукового дослідження.

У цій кваліфікаційній роботі досліджується підхід динамічного скорочення схеми як механізм підвищення ефективності та коректності систем Text-to-SQL у великомасштабних реляційних базах даних.

У першому розділі проаналізовано сучасні підходи до побудови Text-to-SQL систем та визначено ключові виклики корпоративного середовища. У другому розділі спроєктовано пайплайн із динамічним скороченням схеми та описано його архітектурні компоненти. У третьому розділі описано реалізацію системи. У четвертому розділі наведено результати експериментальної оцінки запропонованого підходу.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Опис предметної галузі

Стрімке зростання обсягів даних практично в усіх галузях закріпило позиції систем управління реляційними базами даних (RDBMS) як основоположної інфраструктури для зберігання та пошуку даних. Для доступу до цих даних та їх обробки в якості основного інтерфейсу використовується мова структурованих запитів (SQL). Однак існує значна перешкода для входу: формулювання ефективних і точних запитів SQL вимагає спеціалізованих технічних знань, глибокого ознайомлення зі схемою конкретної бази даних та розуміння реляційної алгебри. Ця технічна перешкода заважає експертам у галузі, бізнес-аналітикам та звичайним користувачам самостійно отримувати корисну інформацію з даних, створюючи вузьке місце, де вилучення даних значною мірою залежить від команд з обробки даних.

З метою забезпечення рівного доступу до даних з’явилася галузь «Інтерфейс природної мови до баз даних» (NLIDB). Основна мета NLIDB полягає в перетворенні запитання користувача, сформульованого природною мовою, на структурно правильний та семантично точний виконуваний запит. Це конкретне завдання перетворення відоме як Text-to-SQL.

Історично системи Text-to-SQL базувалися на підходах, що ґрунтувалися на правилах, зіставленні ключових слів або нейронних мережах типу sequence-to-sequence раннього покоління. Ці застарілі системи часто стикалися з проблемами, пов’язаними з властивою людській мові багатозначністю, складними лінгвістичними структурами та нездатністю узагальнювати результати на основі невідомих схем баз даних.

Поява великих мовних моделей (LLM), побудованих на архітектурі Transformer, кардинально змінила сферу перетворення тексту в SQL.

Сучасні LLM володіють розширеними можливостями міркування, глибоким розумінням синтаксису та величезним масивом попередньо навчених знань щодо мов програмування, зокрема SQL. У сучасних пайплайну перетворення тексту в SQL, де LLM виступає в ролі основного механізму обґрунтування.

Основна галузь сучасного генерування Text-to-SQL передбачає взаємодію трьох основних компонентів:

– висловлювання природною мовою, запит користувача, який часто має неформальний характер, залежить від контексту та не містить явних посилань на фактичні структури бази даних (наприклад, запит про «найкращих клієнт» замість посилання на таблицю customer_revenue);

– контекст бази даних, структурні метадані бази даних. Сюди входять імена таблиць, імена стовпців, типи даних, обмеження первинних та зовнішніх ключів, а іноді – статистичні метадані або зразки рядків;

– велика мовна модель, що відповідає за зв’язування схем (відповідність слів у запиті природною мовою конкретним таблицям і стовпцям) та синтаксичне формування (складання синтаксису SQL відповідно до конкретного діалекту, такого як PostgreSQL та MySQL).

Хоча великі мовні моделі демонструють вражаючу ефективність у стандартних системах оцінки Text-to-SQL, таких як Spider або WikiSQL, ці набори даних, як правило, відображають ідеалізовані сценарії з базами даних малого та середнього розміру та простою архітектурою. У цій роботі, однак, дослідження базується на практичних реаліях роботи з великомасштабними базами даних, що є характерним для складних корпоративних сховищ даних.

Перехід від стандартних бенчмарків до контексту корпоративного масштабу пов’язаний із низкою серйозних викликів:

– великі розміри схеми, корпоративні бази даних зазвичай містять сотні або тисячі взаємопов’язаних таблиць, що складаються з десятками тисяч окремих стовпців. Введення всього цього структурного відображення

безпосередньо у вікно контексту великої мови (LLM) є обчислювально неможливим і вкрай неефективним.

– структурна неоднозначність та застаріла архітектура. Продуктивні бази даних часто страждають від наслідків застарілих архітектурних рішень. Це включає неінтуїтивні або загадкові конвенції іменування, надлишкові структури даних та надзвичайно заплутані реляційні залежності, в яких LLM важко орієнтуватися без чітких вказівок.

– суворі вимоги до затримки, генеративний процес LLM за своєю суттю спричиняє обчислювальні затримки. У реаліях сьогодення користувачі очікують майже миттєвих результатів. Обробка великих схем даних та формування складних запитів створюють значне вузьке місце, що гальмує швидкість реагування системи.

– підвищений ризик «галюцинацій» моделі, ймовірність того, що LLM генеруватиме помилкові результати – наприклад, вигадуватиме назви стовпців, визначатиме неправильні з’єднання таблиць або порушуватиме синтаксис, характерний для конкретного діалекту – зростає пропорційно до складності бази даних. У масштабних системах такі помилки є особливо руйнівними; вони можуть спричиняти обчислювально ресурсомісткі збій у виконанні або, що ще гірше, повертати користувачеві набори даних, які є синтаксично правильними, але фактично помилковими.

Як наслідок, напрям досліджень у галузі перетворення тексту в SQL змістився від простого перекладу послідовностей до комплексної координації систем. Створення надійного та якісного коду SQL у середовищах з великими базами даних вимагає застосування сучасних багатоетапних пайплайнів.

Такі системи повинні динамічно фільтрувати та витягувати лише найбільш релевантну інформацію про схему (використовуючи такі методи, як Retrieval-Augmented Generation) та застосовувати ретельну розробку запитів. Головним завданням і основним фокусом цього дослідження полягає в успішному пошуку компромісів між масштабом моделі, точністю

запитів та затримкою системи на перетині обробки природної мови та системної інженерії.

1.2 Методи генерації SQL-запитів у великомасштабних базах даних

Перехід від навчальних бенчмарків до великих корпоративних баз даних вимагає кардинальної зміни підходу до архітектури систем Text-to-SQL. Ранні підходи, що базувалися на передачі всієї схеми бази даних у велику мовну модель разом із запитом користувача, що часто називають повною серіалізацією схеми, є обчислювально нездійсненними у масштабних середовищах. Вони призводять до надмірної затримки, високих витрат на API та значного зниження продуктивності, оскільки LLM не справляється з перевантаженням контекстом.

Для того, щоб подолати ці обмеження, сучасні методи надають перевагу оркеструванню, цільовому пошуку та ітеративному вдосконаленню над простим перекладом Seq-to-Seq. Ці передові методології можна розділити на три основні фреймворки, призначені для оптимізації компромісу між якістю генерації та затримкою системи.

1.2.1 Генерація з розширеним пошуком та динамічним скороченням схеми

Для подолання обмеження контекстного вікна та зменшення затримки в обробці, у сучасних архітектурах перед етапом генерації застосовуються методи ретельного зв’язування та динамічного скорочення схеми. Головною метою полягає в тому, щоб виділити лише ті таблиці та стовпці, які мають безпосереднє відношення до запиту природною мовою, створюючи таким чином «динамічне контекстне вікно».

Замість того, щоб покладатися виключно на параметричну пам’ять LLM, ці фреймворки перетворюють структурні метадані бази

даних – такі як мови визначення даних (DDL), описи стовпців та статистичні профілі даних – у щільні векторні вбудовування за допомогою спеціалізованих моделей.

Коли користувач надсилає запит природною мовою, система виконує математичний пошук за схожістю, часто використовуючи косинусну схожість, у векторній базі даних. Цей процес вибирає топ k таблиць та стовпців, що мають найбільшу семантичну релевантність, на основі запиту користувача.

Однак якщо покладатися виключно на векторну схожість, іноді можна отримати стовпці, які є семантично пов’язаними в природній мові, але функціонально нерелевантними для фактичної логіки SQL. Щоб вирішити цю проблему, сучасні системи інтегрують ієрархічний процес відбору, керований запитами.

Спочатку спеціальний механізм відбору таблиць відфільтровує з початкового набору векторів повністю нерелевантні таблиці. Потім вторинний процес аналізує конкретний намір, що стоїть за запитом користувача, щоб ще більше скоротити кількість стовпців-кандидатів, що залишилися.

Поступово звужуючи структурні дані, остаточна LLM отримує сильно скорочений, високоточний словник схем. Цей вдосконалений контекст не тільки різко скорочує час обробки та затримку, але й значно мінімізує ризик того, що модель буде «галюцинувати» неіснуючі структури бази даних.

На рисунку 1.1 зображено архітектуру RAG пайплайну для Text-to-SQL. На офлайн-фазі візуалізовано процес вилучення та узагальнення вихідних схем баз даних у словники метаданих. На онлайн-фазі продемонстровано динамічне скорочення схем, під час якого агент-селектор витягує релевантний контекст на основі запиту користувача природною мовою, а потім передає відфільтровану схему агенту-генератору для формування SQL-запиту.

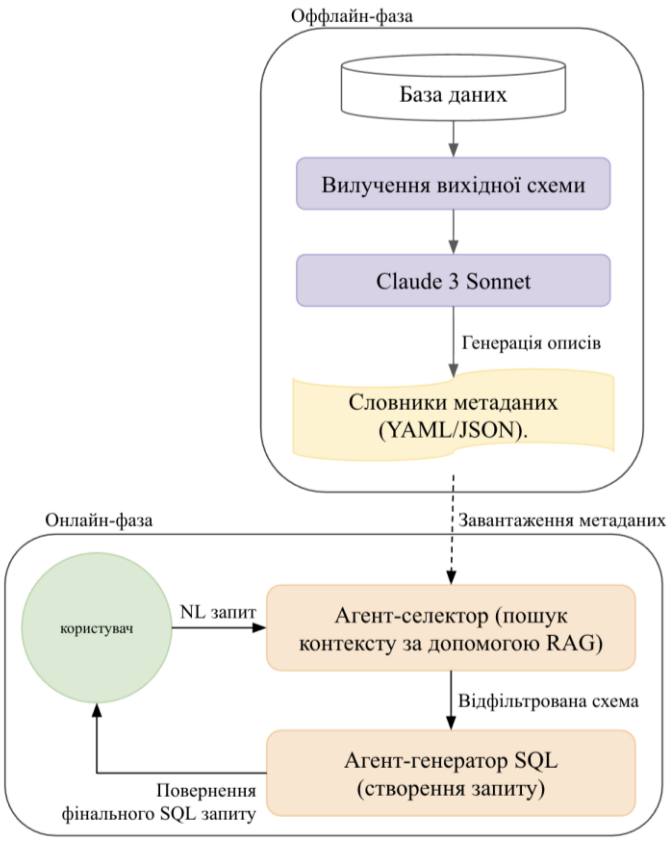


Рисунок 1.1 – Архітектура RAG пайплайну для Text-to-SQL

1.2.2 Багатоагентні та ансамблеві архітектури

Створення складних SQL-запитів, що передбачають об’єднання декількох таблиць, глибоко вкладені підзапити та специфічну бізнес-логіку підприємства, часто виходить за межі можливостей одноразового міркування, на які здатна одна LLM. Щоб подолати це обмеження у масштабних середовищах баз даних, останні архітектурні прориви використовують фреймворки з декількома генераторами, які навмисно розподіляють когнітивне навантаження. Відходячи від монолітного підходу до генерації, ці системи розподіляють складний робочий процес, призначаючи конкретні, ізольовані ролі різним екземплярам моделей, що дозволяє кожному з них спеціалізуватися на окремій фазі процесу перетворення тексту в SQL.

Поширеним і надзвичайно ефективним архітектурним шаблоном у цій галузі є послідовна взаємодія спеціалізованих агентів. Спочатку використовується агент ґрунтування схеми для встановлення жорсткого, детермінованого відображення абстрактних понять природної мови на конкретні, оптимізовані поля бази даних. Після того, як це відображення забезпечено, контекстні дані передаються агенту виконання, який діє виключно як генератор, зосереджений лише на формуванні точного синтаксису SQL на основі обґрунтованої схеми. Після генерації агент рефлексії або валідації аналізує вихідні дані генератора, оцінюючи їх на логічну узгодженість та пропонуючи структурні виправлення, якщо семантичні помилки виявляються ще до виконання запиту.

Для подальшого підвищення точності та управління неоднозначністю складних запитів у сучасних фреймворках часто застосовують ансамблевий підхід на додаток до цього багатоагентного пайплайну. Замість того, щоб покладатися на єдиний визначений шлях генерації, система одночасно генерує кілька різноманітних варіантів SQL, спонукаючи моделі досліджувати різні траєкторії міркування. Додатковий механізм відбору, який часто діє як агент голосування, згодом оцінює ці конкуруючі варіанти на основі структурної узгодженості, дотримання обмежень схеми та загальної ймовірності генерації. Хоча така ансамблева методологія за своєю природою збільшує обчислювальні навантаження та затримку системи, вона слугує необхідним механізмом для забезпечення високої точності виконання найскладніших взаємодій з базою даних.

На рисунку 1.2 зображено приклад багатоагентної взаємодії у пайплайні Text-to-SQL. Агент прив'язки до схеми отримує запит користувача та визначає таблиці й поля-кандидати. Агент виконання формує системний промпт із відібраним контекстом та передає його до LLM для генерації SQL-запиту. Агент рефлексії аналізує отриманий результат і за потреби оновлює пам'ять для покращення майбутніх запитів.

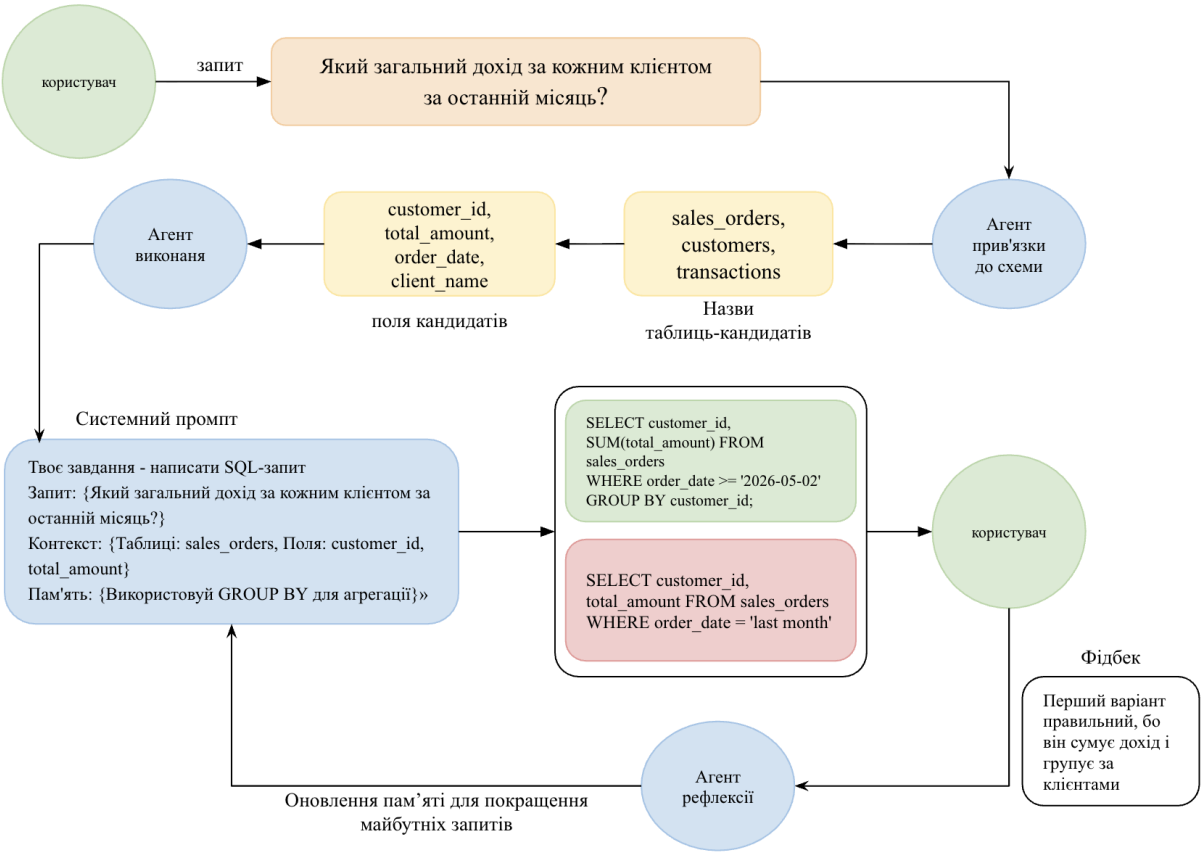


Рисунок 1.2 – Візуалізація багатоагентної взаємодії

1.2.3 Масштабування під час тестування та самокоригування на основі результатів виконання

Постійною та критичною проблемою у процесі генерації запитів SQL на основі тексту є схильність великих мовних моделей створювати запити, які здаються бездоганними з синтаксичної точки зору для людського рецензента, але дають збій під час фактичного виконання в базі даних.

Ці збій під час виконання зазвичай зумовлені ледь помітними структурними помилками, недійсними реляційними з’єднаннями або незначними розбіжностями у діалектах, які моделі важко передбачити за один прохід без попереднього навчання.

Щоб гарантувати надійність та цілісність даних у виробничих середовищах баз даних, сучасні передові фреймворки відійшли від розгляду

перетворення тексту в SQL як статичного завдання перекладу, натомість розглядаючи його як ітеративний процес, керований виконанням.

Ця зміна парадигми реалізується завдяки впровадженню масштабування під час тестування та автономних циклів самокорекції. У цих архітектурах спочатку згенерований кандидат SQL не повертається користувачеві відразу, натомість він автоматично виконується в ізольованій пісочниці бази даних, доступній лише для читання.

Якщо виконання викликає детерміновану помилку бази даних, таку як відсутнє посилання на стовпець, неоднозначне з’єднання або недійсна операція групування – точний журнал помилок фіксується та передається назад у спеціалізовану модель уточнення як цільовий контекст.

Працюючи в рамках безперервного циклу «думати-діяти-спостерігати», модель використовує цей детермінований зворотний зв'язок для логічного виведення своїх помилок та коригування стратегії генерації. Вона систематично коригує синтаксис SQL на основі відповідей бази даних у реальному часі, доки не буде досягнуто успішного виконання.

Перерозподіляючи обчислювальні ресурси на фазу виведення, що дозволяє моделі «думати» довше та повторювати свої власні помилки, ця методологія гарантує, що кінцевому користувачеві в кінцевому підсумку будуть представлені лише повністю перевірені, виконувані запити, що різко зменшує частоту прихованих збоїв у великомасштабних системах.

На рисунку 1.3 зображено механізм самокоригування в багатоагентному середовищі.

Агент генерації формує SQL-кандидат на основі системного промпту та контексту схеми.

Згенерований запит виконується в пісочниці в режимі read-only. У разі виявлення помилки журнал помилок передається агенту уточнення, який коригує промпт та генерує виправлений SQL-запит.

Цикл повторюється до отримання успішного результату.

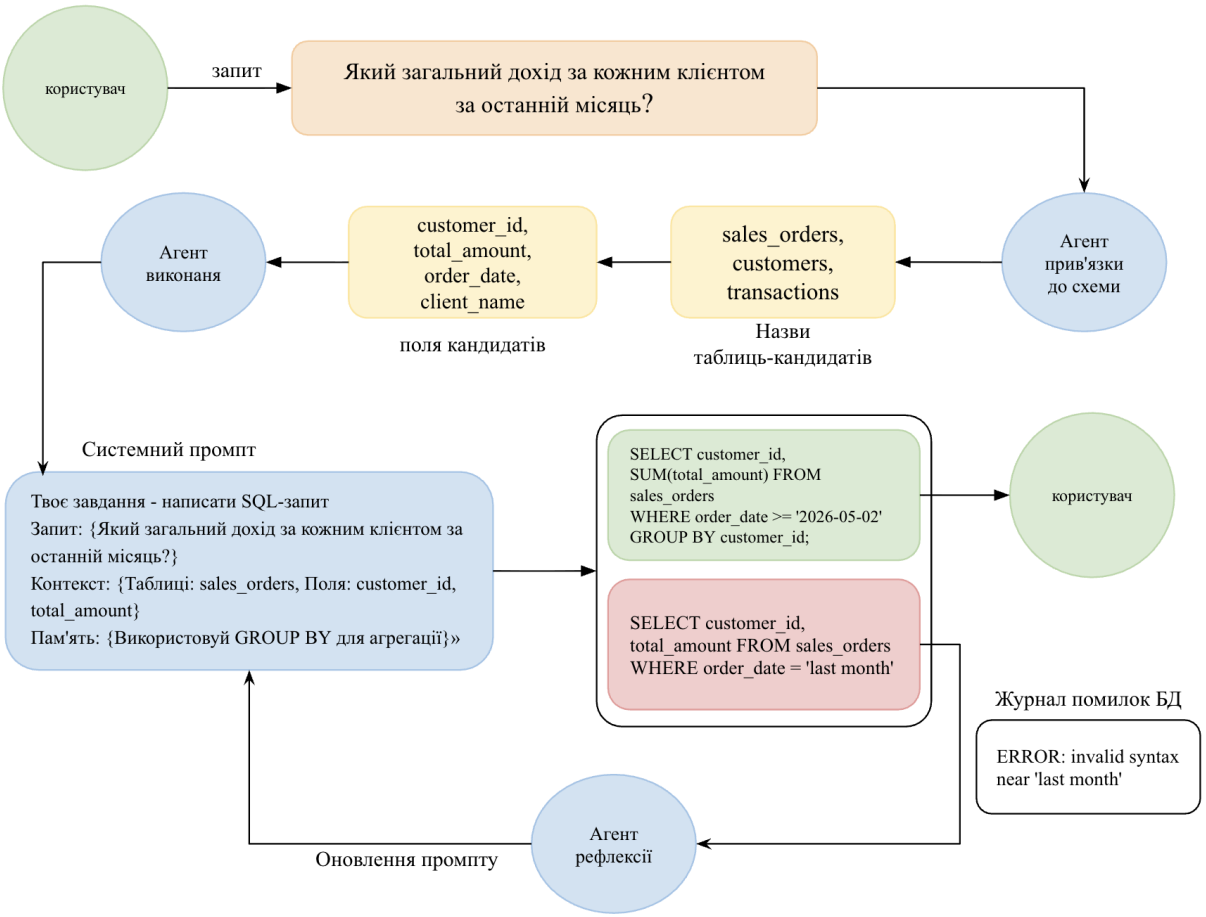


Рисунок 1.3 – Візуалізація самокоригування в багатоагентному середовищі

1.3 Постановка задачі

Основною метою є розробка та оцінка пайплайну Text-to-SQL, що зменшує затримку генерації SQL-запитів і підвищує їхню коректність для великомасштабних реляційних баз даних шляхом динамічного скорочення схеми перед передачею контексту до великої мовної моделі.

Практичне застосування Text-to-SQL систем у реальних корпоративних середовищах суттєво відрізняється від умов академічних бенчмарків. Сучасні LLM ефективно справляються з невеликими і добре структурованими схемами, проте при роботі з базами даних, що містять сотні таблиць, тисячі стовпців та специфічну бізнес-логіку, їхня надійність знижується. Передача повної схеми до моделі збільшує розмір промпту,

вартість API та час відповіді, а надлишковий контекст підвищує ймовірність вибору нерелевантних таблиць або галюцинації неіснуючих стовпців. З іншого боку, надмірне скорочення контексту може призвести до семантично некоректних запитів. Таким чином, ключовим питанням є визначення оптимального обсягу контексту схеми, що забезпечує баланс між точністю та швидкодією.

Запропонований підхід ґрунтується на динамічному скороченні схеми: система індексує схему бази даних, для кожного запиту користувача відбирає лише релевантні таблиці та поля і передає скорочений контекст до LLM замість повної схеми.

Для досягнення поставленої мети визначено такі завдання:

– проаналізувати сучасні підходи до побудови Text-to-SQL систем та виявити їхні обмеження в умовах великомасштабних реляційних баз даних;

– спроєктувати пайплайн з динамічним скороченням схеми, що включає модулі індексації схеми, вибору релевантних таблиць та формування скороченого контексту;

– реалізувати базовий та оптимізований пайплайни з механізмами валідації та безпечного виконання SQL-запитів у режимі read-only;

– провести експериментальне порівняння пайплайнів на реальній базі даних за показниками часу генерації та частки успішно згенерованих запитів;

– проаналізувати результати та визначити напрями подальшого вдосконалення системи.

2 ПРОЄКТУВАННЯ ПАЙПЛАЙНУ TEXT-TO-SQL З ДИНАМІЧНИМ СКОРОЧЕННЯМ СХЕМИ

2.1 Системні вимоги

Аналіз, представлений у попередньому розділі, продемонстрував, що сучасні системи Text-to-SQL стикаються зі значними труднощами при застосуванні до великомасштабних реляційних баз даних. Хоча великі мовні моделі (LLM) показали високу ефективність на академічних бенчмарках, реальні корпоративні середовища створюють додаткові виклики, включаючи розгалужені схеми баз даних, складні зв'язки між таблицями, специфічну термінологію предметної галузі, неоднозначні запитання користувачів та суворі вимоги до затримки. Нещодавні корпоративно-орієнтовані бенчмарки, такі як Spider 2.0, підтверджують, що реальні завдання Text-to-SQL часто вимагають від моделей роботи з великими просторами метаданих, декількома діалектами SQL, контекстом на рівні проєкту та базами даних, що містять понад тисячу стовпців, що робить пряме використання повної схеми в промптах непрактичним для промислового використання.

Таким чином, запропонована система повинна не лише перетворювати запит природною мовою на синтаксично правильний SQL-запит, а й виконувати це перетворення ефективно, безпечно та з достатньою семантичною коректністю. У контексті цієї кваліфікаційної роботи система спроєктована як пайплайн Text-to-SQL, який використовує динамічне скорочення схеми для зменшення непотрібного контексту схеми перед генерацією SQL. Основною метою такого конструктивного рішення є покращення балансу між точністю генерації та затримкою відповіді.

Системні вимоги можна поділити на дві основні категорії: функціональні та нефункціональні. Функціональні вимоги визначають операції, які система повинна виконувати, тоді як нефункціональні

описують очікувані якісні атрибути, такі як швидкість, масштабованість, надійність та безпека.

2.1.1 Функціональні вимоги

Функціональні вимоги визначають операції, які система повинна виконувати. Запропонований пайплайн Text-to-SQL має забезпечувати такі функції:

– приймання запитів природною мовою – система надає користувачам можливість ставити аналітичні запитання без знання синтаксису SQL або внутрішньої структури бази даних;

– інтерпретація запиту – система аналізує запитання користувача та визначає цільові об'єкти, умови фільтрування, вимоги до агрегації та очікуваний формат виводу;

– зв'язування схеми (schema linking) – система зіставляє поняття природної мови з реальними елементами бази даних: назвами таблиць, стовпців, первинними та зовнішніми ключами;

– динамічне скорочення схеми (schema pruning) – замість передачі повної схеми система відбирає лише фрагменти, релевантні для поточного запиту, що є критично важливим для великомасштабних баз даних;

– цільова побудова промпту – на основі скороченої схеми система формує промпт із запитанням користувача, контекстом схеми та явними інструкціями щодо генерації;

– генерація SQL-запиту – велика мовна модель генерує SQL на основі сформованого промпту, тоді як вибір схеми та побудова промпту обробляються зовнішніми модулями пайплайну;

– валідація SQL перед виконанням – згенерований запит перевіряється на відповідність read-only операціям, відсутність деструктивних операторів і коректність посилань на таблиці та стовпці;

– безпечне виконання запиту – система виконує лише перевірені SQL-запити в режимі read-only, що запобігає ненавмисній зміні або видаленню даних;

– обробка результатів та формування відповіді – результат виконання SQL перетворюється на зрозумілу для користувача відповідь природною мовою, засновану на фактичних даних бази даних.

2.1.2 Нефункціональні вимоги

Нефункціональні вимоги описують якісні атрибути системи та є особливо важливими в контексті цієї роботи, оскільки центральним питанням є компроміс між точністю генерації та затримкою відповіді. До основних нефункціональних вимог належать:

– низька затримка – система повинна генерувати SQL-запити в часових межах, придатних для взаємодії з користувачем у режимі, близькому до реального часу;

– масштабованість – система має підтримувати схеми баз даних великого обсягу, що містять сотні таблиць і тисячі стовпців, використовуючи індексоване представлення схеми замість її повної серіалізації;

– коректність SQL – згенерований запит має бути синтаксично правильним і семантично відповідати запитанню користувача, включаючи правильний вибір таблиць, стовпців, з'єднань та логіки агрегації;

– мінімізація галюцинацій – система знижує ймовірність вигадування неіснуючих таблиць, стовпців або зв'язків шляхом надання моделі меншого та більш релевантного контексту схеми;

– економічна ефективність – зменшення розміру промпту через динамічне скорочення схеми безпосередньо знижує витрати на токени при зверненні до API великих мовних моделей;

– безпека – система блокує деструктивні оператори SQL (INSERT, UPDATE, DELETE, DROP, ALTER, TRUNCATE) і обмежує виконання виключно read-only запитами;

– надійність – система має передбачати резервні стратегії для запитів, у яких не вдається впевнено визначити необхідні елементи схеми;

– підтримуваність – архітектура системи є модульною, що дозволяє вдосконалювати компоненти скорочення схеми, побудови промпту, генерації SQL та валідації незалежно один від одного;

– пояснюваність та спостережуваність – система дозволяє розробникам перевіряти вибрані елементи схеми, сформований промпт, згенерований SQL та результати валідації для цілей налагодження й оптимізації.

2.2 Загальна архітектура запланованого пайплайну Text-to-SQL

Запропонована система Text-to-SQL побудована у вигляді багатоетапного пайплайну, який перетворює запит користувача природною мовою на перевірений SQL-запит, а потім – на кінцеву відповідь природною мовою. На відміну від одноетапного підходу, за якого повна схема бази даних передається безпосередньо до великої мовної моделі, запропонована архітектура передбачає проміжний етап вибору схеми. Цей етап відповідає за скорочення обсягу нерелевантної інформації про схему перед генерацією SQL-запиту.

На рисунку 2.1 зображено загальну архітектуру запропонованого пайплайну Text-to-SQL з динамічним скороченням схеми.

Процес складається з двох основних етапів: офлайн-етапу та онлайн-етапу. Офлайн-етап виконується одноразово або періодично і включає вилучення схеми бази даних, перетворення структур таблиць у DDL-представлення та побудову індексу схеми. У цьому індексі кожна таблиця може бути зіставлена з відповідною інструкцією CREATE TABLE та

пов’язаними метаданими. Це дає змогу ефективно отримувати вибрані фрагменти схеми під час обробки запитів.

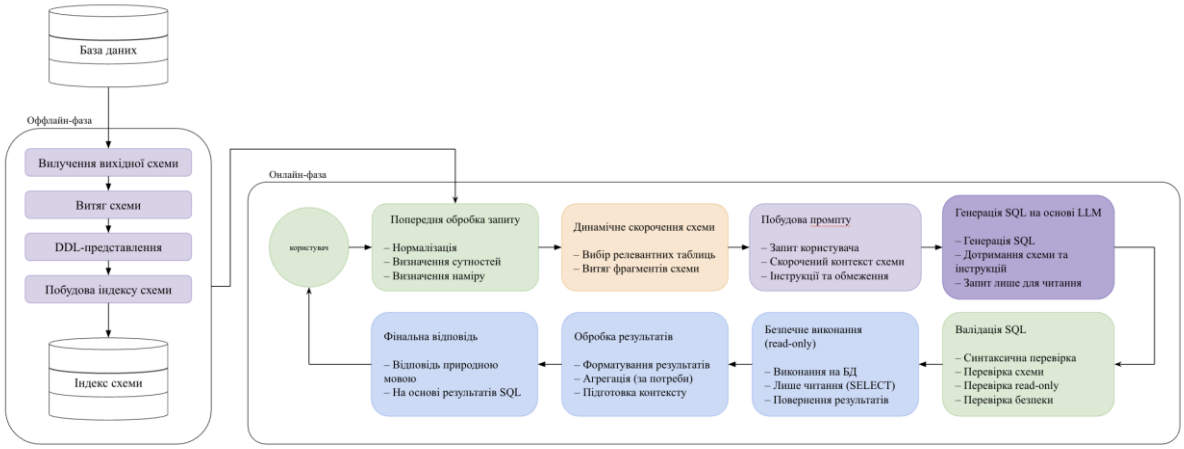


Рисунок 2.1 – Архітектура запропонованого пайплайну Text-to-SQL з динамічним скороченням схеми

Онлайн-фаза виконується для кожного запиту користувача. Спочатку система отримує запит природною мовою та виконує базову попередню обробку, таку, як ідентифікація важливих сутностей, обмежень та аналітичного наміру. Потім модуль динамічного обрізання схеми відбирає таблиці та фрагменти схеми, які є найбільш релевантними для запиту. Замість того, щоб вставляти всю схему бази даних у запит, система будує скорочений контекст схеми, що містить лише вибрані визначення таблиць та взаємозв’язки.

Після обрізки схеми модуль побудови запиту поєднує запитання користувача, обрізаний контекст схеми та інструкції щодо генерації. Потім велика мовна модель генерує SQL-запит на основі цього цільового контексту. Згенерований запит передається до модуля валідації, де перевіряється на синтаксичну правильність, узгодженість схеми та обмеження безпеки. Зокрема, система обмежує виконання лише SQL-запитами для читання, щоб запобігти ненавмисним змінам у базі даних.

Якщо запит проходить перевірку, він виконується в базі даних. Отриманий результат потім обробляється та використовується як контекст, що ґрунтується на фактичних даних, для формування остаточної відповіді. У результаті користувач отримує відповідь природною мовою, засновану на фактичних даних з бази даних, а не на безпідставних припущеннях моделі.

Головною перевагою запропонованої архітектури є те, що вона зменшує обсяг контексту схеми, який обробляє LLM, зберігаючи при цьому інформацію, необхідну для генерації SQL. Це безпосередньо вирішує проблему компромісу між якістю та затримкою, про яку йшлося в попередньому розділі: менші та більш релевантні підказки зменшують затримку генерації, а цілеспрямоване введення схеми допомагає моделі зосередитися на правильних таблицях і стовпцях.

Таким чином, ця архітектура розглядає генерацію Text-to-SQL як узгоджений пайплайн, а не як окреме завдання з автоматичного заповнення командного рядка. Основним компонентом цього пайплайн є динамічне скорочення схеми, що дозволяє системі ефективніше працювати з великими схемами реляційних баз даних.

2.3 Базовий підхід: використання підказок на основі повної схеми

Перш ніж представити запропонований підхід до динамічного скорочення схеми, необхідно визначити базовий метод, який буде використовуватися для порівняння. У цій роботі базовим методом є запит із повною схемою, коли повна схема бази даних включається в запит разом із запитами користувача, сформульованим природною мовою. Цей підхід є одним із найпростіших способів побудови системи Text-to-SQL на основі LLM, оскільки він не вимагає наявності окремого модуля для вибору або пошуку схеми.

На рисунку 2.2 зображено метод формування запитів із використанням повної схеми бази даних. У базовому підході користувач

надсилає запит, після чого повна схема бази даних передається безпосередньо до блоку побудови промпту, який містить інструкцію, повну схему та запит користувача. Сформований промпт передається до великої мовної моделі для генерації SQL-запиту (рисунок 2.3).

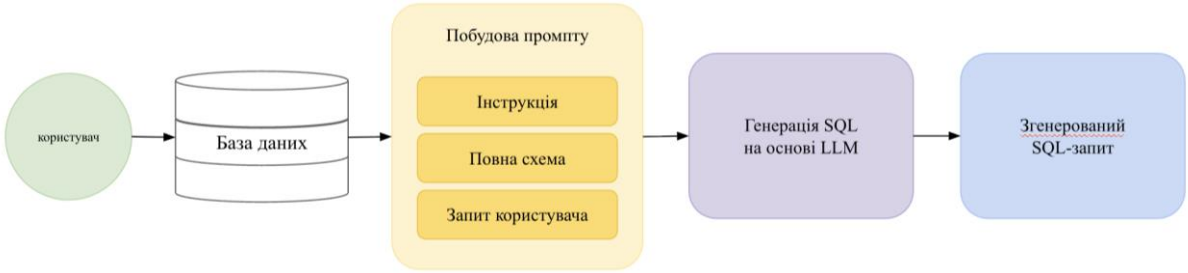


Рисунок 2.2 – Метод формування запитів із використанням повної схеми бази даних

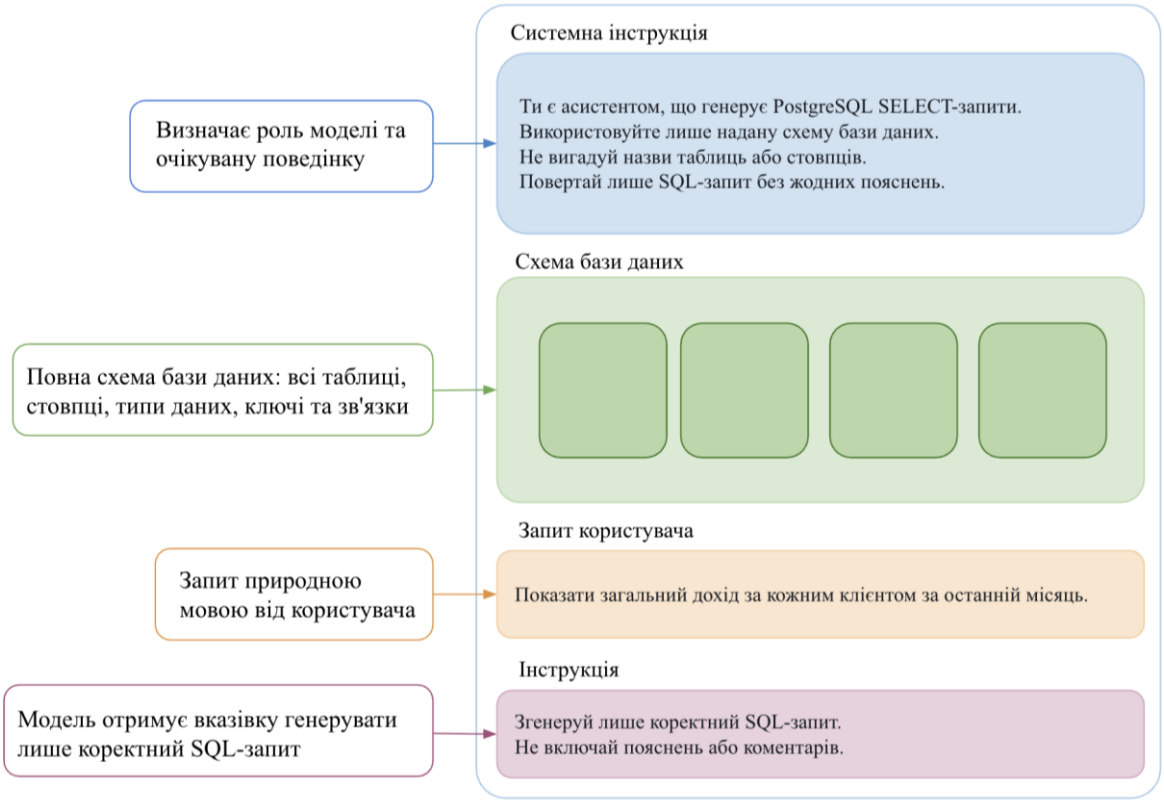


Рисунок 2.3 – Приклад промпту з повною схемою бази даних

Зазвичай запит складається з трьох основних компонентів: запиту користувача, повного контексту схеми та інструкцій щодо формування SQL-запиту. Контекст схеми може містити імена таблиць, імена стовпців, типи даних, первинні ключі, зовнішні ключі, а іноді й додаткові метадані, такі як описи таблиць або зразки значень. Мета полягає в тому, щоб надати моделі всю наявну структурну інформацію, щоб вона могла визначити, які таблиці та стовпці є релевантними для запиту.

Головною перевагою цього базового підходу є його простота. Оскільки моделі надається повна схема, системі не потрібно реалізовувати окремий механізм вилучення схеми. Велика мовна модель отримує всі доступні метадані бази даних і теоретично може самостійно обирати необхідні таблиці та стовпці. Це робить підхід із використанням повних схем корисним для невеликих баз даних, навчальних прикладів, прототипів та тестових завдань, де кількість таблиць обмежена.

Однак цей підхід стає неефективним при застосуванні до великих реляційних баз даних. Корпоративні бази даних можуть містити сотні таблиць і тисячі стовпців, а реальні робочі процеси Text-to-SQL часто передбачають великі простори метаданих, численні діалекти SQL, складні робочі процеси та системи баз даних, такі як BigQuery або Snowflake. Spider 2.0, сучасний тест Text-to-SQL, орієнтований на корпоративне середовище, особливо підкреслює, що реальні завдання часто вимагають розуміння метаданих бази даних та роботи з базами даних, які можуть містити понад тисячу стовпців.

Першим обмеженням використання повних схем у запитах є великий розмір запиту. Коли всі визначення таблиць вставляються в запит, кількість вхідних токенів значно зростає. Це може призвести до підвищення вартості API, уповільнення відповіді моделі та, в деяких випадках, до обмежень вікна контексту. Навіть якщо повна схема технічно вміщується у вікно контексту моделі, це все одно може знизити практичну ефективність, оскільки модель

мусить обробити великий обсяг нерелевантної інформації перед генерацією SQL-запиту.

Другим обмеженням є збільшення затримки. LLM потребують більше часу для обробки довгих запитів, оскільки на увагу та декодування впливає розмір вхідного контексту. У виробничій системі Text-to-SQL користувачі очікують взаємодії майже в режимі реального часу. Тому додавання повної корпоративної схеми до кожного запиту створює значне вузьке місце в продуктивності.

Третім обмеженням є знижена надійність генерації. Більш розгалужена схема не завжди допомагає моделі. Навпаки, надмірний контекст схеми може ускладнити завдання генерації. Коли запит містить багато нерелевантних таблиць і стовпців, модель може вибрати неправильну таблицю, «уявити» неіснуючий стовпець або побудувати недійсне з’єднання. Останні дослідження в галузі перетворення тексту в SQL визначають розуміння схеми та зв'язування схем як основні виклики, оскільки система повинна правильно зіставити поняття природної мови з фактичними елементами бази даних. Це стає складнішим, коли схема є великою, специфічною для певної галузі або структурно складною.

Четвертим обмеженням є погана масштабованість. Запит із повною схемою розглядає схему бази даних як єдиний монолітний контекст. У міру зростання бази даних зростає й запит. Це означає, що кожна нова таблиця, стовпець або зв’язок збільшує обсяг інформації, що передається до LLM, навіть якщо більша її частина не має відношення до конкретного запиту користувача. З цієї причини запит із повною схемою погано масштабується до корпоративних баз даних.

У цій роботі підхід із використанням повного опису схеми застосовується як базовий, оскільки він наочно ілюструє компроміс між якістю та затримкою. Цей метод надає моделі повний доступ до метаданих бази даних, що може бути корисним для невеликих схем або простих прототипів. Однак у великих реляційних базах даних така конструкція стає

неефективною, оскільки запит містить значний обсяг інформації, яка не має відношення до поточного запиту користувача.

Як результат, LLM змушена обробляти не лише таблиці та стовпці, необхідні для генерації SQL, а й багато елементів схеми, що не мають відношення до запиту. Це збільшує використання токенів, затримку генерації та обчислювальні витрати. Крім того, надмірний контекст схеми може негативно вплинути на коректність SQL, оскільки модель має більше можливостей вибрати нерелевантну таблицю, використати неправильний стовпець або побудувати недійсне з'єднання.

Тому підказка з повною схемою не вважається достатнім рішенням для орієнтованих на виробництво систем Text-to-SQL, що працюють з великими корпоративними схемами. Вона корисна як базова точка для порівняння, але її обмеження спонукають до переходу на більш селективний підхід. У наступному розділі представлено динамічне обрізання схеми, де вибираються лише фрагменти схеми, що мають відношення до конкретного запиту користувача, і вставляються в підказку.

2.4 Рекомендований підхід: динамічне скорочення схеми

Обмеження підходу з використанням повної схеми спонукають до застосування більш вибіркового підходу до побудови контексту схеми. У цій роботі запропоноване рішення базується на динамічному скороченні схеми, що означає: система не передає великій мовній моделі всю схему бази даних. Натомість вона визначає лише ті фрагменти схеми, які мають відношення до поточного запиту користувача, і вставляє їх у підказку.

Цей підхід безпосередньо вирішує проблему компромісу між якістю та затримкою. З одного боку, модель повинна отримати достатньо інформації про схему, щоб згенерувати правильний SQL-запит. З іншого боку, надмірний контекст схеми збільшує довжину запиту, використання токенів, затримку генерації та ймовірність вибору нерелевантних таблиць

або стовпців. Динамічне обрізання схеми запроваджено як механізм для збалансування цих двох вимог.

На рисунку 2.4 зображено загальну ідею запропонованого підходу з динамічним скороченням схеми. Починаючи із запиту користувача, система послідовно звертається до індексу схеми, вибирає лише релевантні таблиці, витягує для них оператори CREATE TABLE, формує скорочений промпт та передає його до LLM для генерації SQL-запиту. На відміну від базового підходу, модель на жодному етапі не отримує повну схему бази даних.

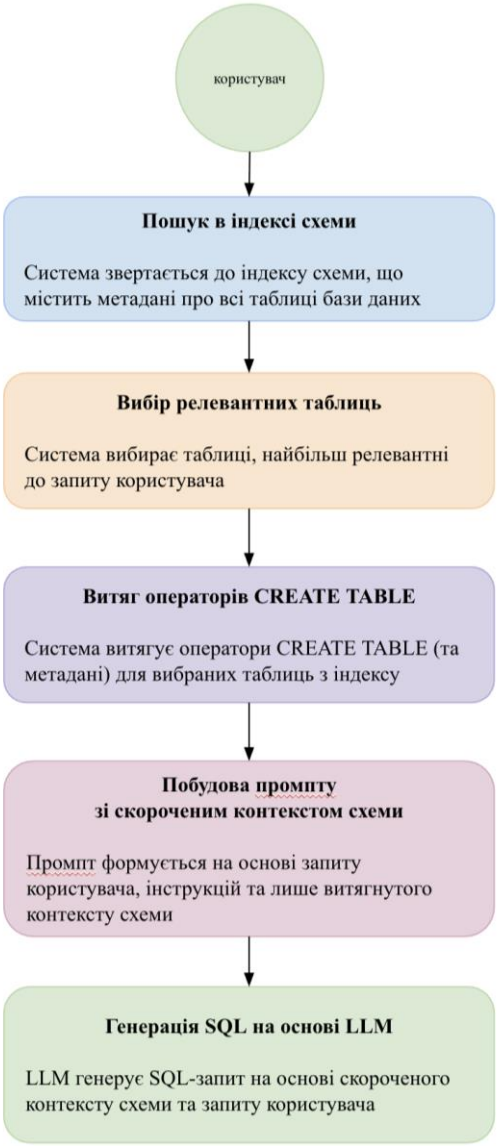


Рисунок 2.4 – Загальна ідея запропонованого підходу з динамічним скороченням схеми

На відміну від підказки з повною схемою, де до підказки включаються всі наявні таблиці та стовпці, динамічне скорочення схеми формує контекст схеми, специфічний для конкретного запиту. Це дозволяє великій мові обробки (LLM) зосередитися лише на тих таблицях і зв’язках, які, ймовірно, будуть потрібні для відповіді на запитання користувача.

Останні дослідження підтверджують важливість цього напрямку. Spider 2.0, орієнтований на корпоративний сектор тест Text-to-SQL, показує, що реальні робочі процеси з базами даних часто передбачають великі простори метаданих, складні хмарні або локальні бази даних, численні діалекти SQL та схеми, що можуть містити понад тисячу стовпців. Такі умови роблять пряме підказування повної схеми неефективним і вимагають від моделей пошуку серед метаданих бази даних перед генерацією SQL. Аналогічно, підходи до зв'язування схем з розширеним пошуком, такі як RASL, розкладають схеми баз даних на менші семантичні одиниці, індексують їх та витягують відповідні елементи схеми, щоб масштабувати системи Text-to-SQL до величезних корпоративних баз даних.

2.4.1 Схема індексації

Першим компонентом запропонованого підходу є індексування схеми. Замість того, щоб зберігати схему бази даних у вигляді одного великого текстового блоку, схема розділяється на менші елементи, які можна витягувати окремо. У найпростішому варіанті кожна таблиця зіставляється з відповідним представленням DDL.

На рисунку 2.5 зображено побудову індексу схеми на основі SQL-визначень таблиць. Повна схема бази даних розбивається на окремі одиниці рівня таблиць: для кожної таблиці формується відповідний оператор CREATE TABLE, який зберігається в індексі разом із пов'язаними метаданими – первинним ключем, кількістю стовпців, зовнішніми ключами та коротким описом. Така структура дозволяє системі швидко витягувати

лише необхідні фрагменти схеми під час обробки кожного запиту користувача, не звертаючись до повної схеми бази даних.

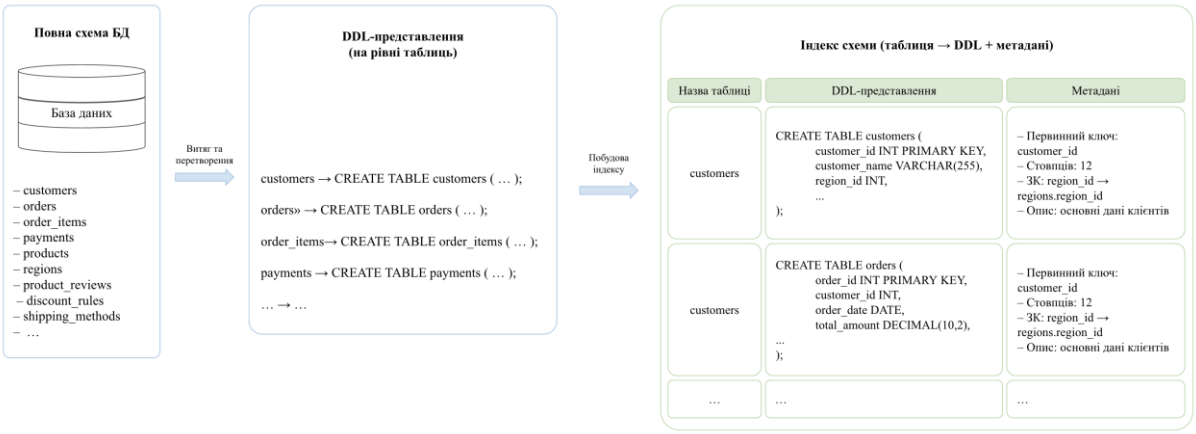


Рисунок 2.5 – Побудова індексу схеми на основі SQL-визначень таблиць

Таке представлення дає змогу отримати визначення конкретної таблиці без необхідності включати в запит всю схему бази даних. Індекс також може містити додаткові метадані, такі як описи стовпців, первинні ключі, зовнішні ключі, зв’язки між таблицями та короткі описи бізнес-процесів. Однак основна ідея залишається незмінною: інформація про схему має бути організована таким чином, щоб забезпечити швидкий і вибірковий пошук.

Індексування схеми є особливо важливим у великих базах даних, оскільки модель не повинна повторно обробляти одну й ту саму повну схему для кожного запиту користувача. Натомість, ресурсомісткий етап підготовки схеми можна виконувати в автономному режимі або періодично, тоді як онлайн-пайплайні отримує лише фрагменти, необхідні для поточного запиту.

У цій роботі індекс схеми використовується як основа для цільового введення схеми. Коли користувач задає питання, система ідентифікує таблиці-кандидати, а потім витягує відповідні оператори CREATE TABLE з

індексу. Це зменшує розмір запиту та дозволяє системі контролювати, який контекст схеми надається LLM.

2.4.2 Вибір відповідної таблиці

Наступним кроком є вибір відповідних таблиць, метою яких є визначити, які таблиці бази даних, найімовірніше, знадобляться для формування SQL-запиту. Це один із найважливіших етапів пайплайна, оскільки неправильний вибір таблиць може призвести до того, що LLM сформує неповний або семантично неправильний SQL-запит.

Відбір відповідних таблиць можна реалізувати за допомогою кількох стратегій. Простим підходом є зіставлення ключових слів, коли система порівнює терміни з запитання користувача з назвами таблиць, назвами стовпців та описами метаданих. Наприклад, якщо користувач запитує про дохід клієнта, система може ідентифікувати такі таблиці, як customers, orders, payments або order_items.

Більш досконалим підходом є пошук на основі вбудовування. У цьому випадку елементи схеми та запит користувача перетворюються на векторні представлення, а система відбирає таблиці на основі семантичної схожості. Це корисно, коли назви таблиць не повністю збігаються зі словами, використаними користувачем. Наприклад, користувач може запитати про «клієнтів», тоді як таблиця в базі даних може називатися «customers». Дослідження в галузі зв’язування схем з розширеним пошуком йдуть у цьому напрямку, індексуючи елементи схеми та використовуючи механізми пошуку для звуження простору пошуку перед генерацією.

Ще однією можливою стратегією є вибір таблиць на основі LLM, коли мовній моделі пропонується вибрати релевантні таблиці зі списку доступних елементів схеми. Однак цей метод може спричинити додаткову затримку та витрати, особливо якщо схема є великою. Тому в системах, орієнтованих на виробниче середовище, вибір таблиць часто реалізується як

легкий етап, що звужує простір пошуку перед основним етапом генерації SQL.

У запропонованому підході вибір відповідних таблиць слугує мостом між запитом користувача та індексом схеми. Система спочатку ідентифікує таблиці-кандидати, а потім отримує їхні визначення DDL для побудови запиту. Це робить генерацію SQL більш цілеспрямованою та зменшує ймовірність того, що модель використовуватиме невідповідні елементи схеми.

2.4.3 Цілеспрямоване введення схеми

Після вибору відповідних таблиць система виконує цільове введення схеми. Це означає, що до запиту для великої мови (LLM) включаються лише вибрані фрагменти схеми. Замість того, щоб надавати моделі всі таблиці бази даних, запит містить скорочений контекст схеми, який відповідає поточному запиту користувача.

На рисунку 2.6 зображено порівняння двох підходів до формування контексту схеми. У базовому підході промпт містить повну схему бази даних, включаючи всі таблиці незалежно від їхньої релевантності до запиту користувача. У запропонованому підході до промпту передається лише скорочена схема, що охоплює виключно таблиці, необхідні для відповіді на конкретний запит. Таке цільове введення схеми дозволяє скоротити розмір промпту та зосередити модель на релевантних елементах бази даних.

У базовому підході запит містить повну схему бази даних. Наприклад, якщо база даних містить такі таблиці, як customers, orders, order_items, payments, products, regions, product_reviews, discount_rules та shipping_methods, усі ці структури передаються до моделі. Однак для такого запитання, як «Який загальний дохід за кожним клієнтом за останній місяць. » фактично потрібна лише невелика частина цих таблиць. У цьому випадку до відповідних таблиць можуть належати «клієнти», «замовлення»,

«платежі» та, можливо, «позиції_замовлення», тоді як такі таблиці, як « discount_rules», «product_reviews» або «shipping_methods», не потрібні для формування SQL-запиту.

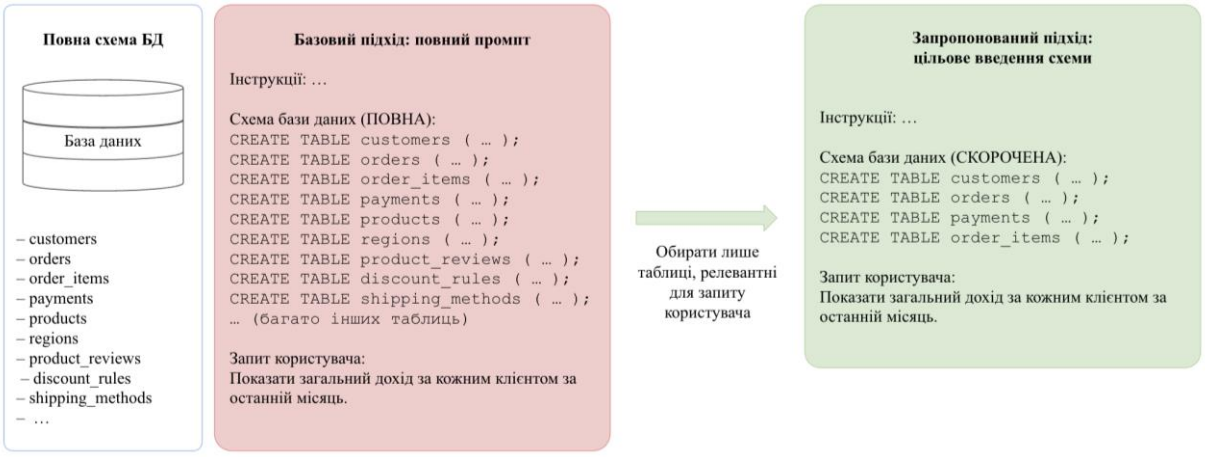


Рисунок 2.6 – Порівняння повної схеми та цілеспрямованого введення релевантних фрагментів схеми

Отже, підхід із запитом повної схеми можна представити так: «Запит = запит користувача + повна схема бази даних + інструкції щодо генерації»

Це призводить до створення довгого запиту, що містить великий обсяг нерелевантної інформації. Як наслідок, модель змушена обробляти зайві елементи схеми, що збільшує кількість токенів, обчислювальні витрати та затримку генерації. Крім того, наявність нерелевантних таблиць і стовпців може відволікати модель і підвищувати ризик неправильного вибору таблиць, некоректних з’єднань або генерації помилкових елементів схеми.

У запропонованому підході в підказку вводяться лише вибрані фрагменти схеми. Після того, як на етапі вибору таблиць визначено відповідні таблиці, система отримує відповідні описи DDL з індексу схеми. Ці описи зазвичай містять імена таблиць, імена стовпців, типи даних, первинні ключі, зовнішні ключі та інші метадані про зв’язки, необхідні для генерації SQL.

Запропонований підхід можна представити так: «Запит = запит користувача + обрані оператори CREATE TABLE + інструкції щодо формування»

Цей підхід має кілька важливих переваг. По-перше, він зменшує розмір запиту, оскільки таблиці, що не мають відношення до справи, видаляються ще до того, як запит надсилається до моделі. По-друге, він скорочує час відгуку, оскільки велика мовна модель (LLM) обробляє менше вхідних токенів. По-третє, він знижує вартість генерації, оскільки багато API-інтерфейсів LLM розраховують вартість на основі кількості використаних токенів. Нарешті, він підвищує надійність генерації SQL, допомагаючи моделі зосередитися на правильних таблицях, стовпцях та зв’язках.

Цільове введення схеми також важливе для зменшення галюцинацій. Коли запит містить багато не пов’язаних елементів схеми, модель може поєднати їх неправильно або вигадати стовпці, які здаються правдоподібними, але не існують у базі даних. Обмежуючи запит вибраними фрагментами схеми, система зменшує кількість нерелевантних альтернатив, доступних для моделі. Це робить процес генерації SQL більш контрольованим і зменшує ймовірність недійсних посилань на таблиці або неправильних з’єднань.

Однак цільове введення схеми має зберігати достатній контекст для правильного формування запиту. Якщо вибраний контекст схеми є занадто вузьким, модель може не отримати всі таблиці, необхідні для з’єднань, фільтрування або агрегації. Тому етап вибору схеми має включати не лише таблиці, що безпосередньо відповідають запиту, а й пов’язані таблиці, з’єднані через відносини первинних та зовнішніх ключів. Наприклад, якщо вибрано таблицю замовлень, системі може також знадобитися включити клієнтів, якщо запит вимагає інформації на рівні клієнта.

2.4.4 Створення підказок із використанням скороченої схеми контексту

Після того, як відповідні фрагменти схеми були відібрані та отримані з індексу схеми, система формує остаточний запит, який передається великій мовній моделі. Цей етап є важливим, оскільки якість генерації SQL залежить не лише від того, які елементи схеми обрано, а й від того, як вони подаються моделі. Недосконало структурований запит може призвести до некоректного SQL, навіть якщо обраний контекст схеми є правильним.

У запропонованому пайплайні побудова запиту базується на скороченому контексті схеми. Замість вставки повної схеми бази даних, запит містить лише оператори CREATE TABLE для таблиць, вибраних на етапі обрізки схеми. Ці оператори надають моделі структурну інформацію, необхідну для генерації SQL, включаючи імена таблиць, імена стовпців, типи даних, первинні ключі, зовнішні ключі та взаємозв’язки між таблицями.

На рисунку 2.3 зображено приклад промпту з повною схемою бази даних. Промпт складається з чотирьох компонентів:

– системної інструкції, що визначає роль моделі та обмежує її поведінку;

– схеми бази даних, що містить визначення всіх таблиць;

– запиту користувача природною мовою;

– інструкції генерації, що вказує моделі повертати лише коректний SQL-запит без пояснень.

На рисунку 2.7 зображено структуру промпту зі скороченим контекстом схеми. Промпт складається з чотирьох компонентів:

– системної інструкції, що визначає роль моделі та обмежує її поведінку;

– скороченого контексту схеми, що містить DDL-оператори лише для вибраних релевантних таблиць;

– запиту користувача природною мовою;

– інструкції виводу, що вказує моделі повертати виключно коректний SQL-запит без пояснень.



Рисунок 2.7 – Структура промпту зі скороченим контекстом схеми

На відміну від підходу з повною схемою, представленого на рисунку 2.3, контекст схеми тут обмежений лише необхідними таблицями.

Перший компонент – це блок інструкцій. Він визначає роль моделі та вказує очікуване поведінку. Наприклад, модель можна налаштувати на генерацію запитів PostgreSQL SELECT, використання лише наданої схеми, уникнення вигадування імен таблиць або стовпців та повернення лише дійсного запиту SQL. Цей блок є необхідним, оскільки в іншому випадку LLM можуть генерувати пояснення, припущення або SQL на основі попередніх знань замість наданої схеми бази даних.

Другий компонент – це контекст обрізаної схеми. Цей блок містить лише вибрані визначення таблиць, отримані з індексу схеми. Наприклад, для запитання про загальний дохід за клієнтами підказка може містити лише таблиці customers, orders, payments та order_items. Непов’язані таблиці видаляються з підказки. У результаті модель отримує меншу та більш цілеспрямовану репрезентацію бази даних.

Третім компонентом є вихідне запитання користувача. Важливо зберегти формулювання запиту природною мовою, оскільки воно містить аналітичний задум, очікуваний результат, умови фільтрації та можливу логіку агрегації. Наприклад, запитання «Який загальний дохід за клієнтами за останній місяць» вказує на те, що SQL-запит повинен згрупувати результати за клієнтами та застосувати фільтр за датою.

Четвертим компонентом є обмеження щодо результату. Ця частина визначає необхідний формат відповіді моделі. У цій роботі модель отримує вказівку повертати лише SQL-запит, без пояснень чи додаткових коментарів. Це спрощує подальшу валідацію та виконання, оскільки результат можна обробляти безпосередньо як SQL.

Ця структура сприяє досягненню головної мети динамічного скорочення схеми: зменшення зайвого контексту при збереженні інформації, необхідної для правильного формування SQL-запитів. Запит залишається компактним, але все одно містить структуру бази даних, необхідну для побудови з’єднань, фільтрів, операцій групування та агрегації.

Використання обрізаного контексту схеми надає кілька переваг. По-перше, це покращує фокусування, оскільки модель бачить лише ті елементи схеми, які, ймовірно, мають відношення до запиту користувача. По-друге, це підвищує ефективність, оскільки підказка містить менше лексичних одиниць, ніж у підході з підказкою повної схеми. По-третє, це підвищує надійність, оскільки модель має у своєму розпорядженні менше

нерелевантних таблиць і стовпців, що зменшує ймовірність виникнення галюцинованих елементів схеми або неправильних з’єднань.

Водночас запит повинен містити достатньо інформації про схему, щоб забезпечити формування коректного SQL-коду. Якщо скорочений контекст є надто вузьким, модель може пропустити необхідний зв’язок або стовпець фільтрації. Тому побудова запиту залежить від якості попереднього етапу вибору схеми. На практиці доцільно включати не лише безпосередньо вибрані таблиці, а й пов’язані таблиці, з’єднані через зв’язки первинних та зовнішніх ключів, якщо вони потрібні для об’єднання.

Таким чином, побудова підказки з обрізаним контекстом схеми є ключовою частиною запропонованого пайплайна Text-to-SQL. Вона перетворює результат обрізання схеми на структурований вхід для LLM і контролює, як модель використовує метадані бази даних під час генерації SQL. Поєднуючи явні інструкції, вибрані визначення схеми, запитання користувача та суворі обмеження на вихідні дані, система підвищує ефективність та надійність генерації SQL для великомасштабних реляційних баз даних.

2.5 Процес генерації, перевірки та безпечного виконання SQL-запитів

Після формування контексту схеми з відфільтрованими елементами наступним етапом запропонованого пайплайна Text-to-SQL є генерація SQL-запиту. На цьому етапі велика мовна модель отримує структурований запит, що містить запитання користувача, вибрані фрагменти схеми та чіткі інструкції щодо генерації. Мета цього етапу – перетворити запит природною мовою на синтаксично правильний та семантично релевантний SQL-запит.

У запропонованому підході LLM не отримує повну схему бази даних. Натомість вона працює лише з контекстом схеми, обраним на етапі динамічного скорочення схеми. Така конструкція зменшує обсяг нерелевантної інформації в запиті та допомагає моделі зосередитися на

таблицях, стовпцях та взаємозв’язках, які, найімовірніше, будуть потрібні для відповіді на запитання користувача.

Згенерований SQL-запит не виконується одразу. Оскільки великі мовні моделі можуть генерувати некоректні, неповні або небезпечні SQL-оператори, запит спочатку має пройти етап перевірки. Цей етап необхідний для того, щоб гарантувати, що згенерований запит можна безпечно виконати у цільовій базі даних.

У контексті цієї кваліфікаційної роботи процес перевірки охоплює три ключові аспекти згенерованого SQL-запиту. По-перше, система перевіряє, чи запит є лише для читання. У контексті цієї роботи система призначена для аналітичного доступу до даних, а не для модифікації записів у базі даних. Тому дозволені лише запити SELECT. Оператори, такі як INSERT, UPDATE, DELETE, DROP, ALTER та TRUNCATE, повинні бути відхилені.

По-друге, система перевіряє, чи використовує згенерований запит лише таблиці та стовпці, що існують у наданому контексті схеми. Цей крок перевірки допомагає зменшити ризик виникнення уявних елементів схеми, що є однією з поширених проблем у генеруванні Text-to-SQL на основі LLM. Якщо модель посилається на таблицю або стовпець, що не існує, запит не повинен виконуватися.

По-третє, система може виконати етап синтаксичної перевірки. Це може включати перевірку того, чи можна правильно проаналізувати SQL-запит відповідно до цільового діалекту SQL. У цій роботі запропонована система орієнтована на безпечне генерування SQL-запитів тільки для читання, тому синтаксична перевірка та перевірка на рівні схеми є важливими запобіжними заходами перед виконанням.

На рисунку 2.8 зображено повний процес генерації, перевірки та безпечного виконання SQL-запиту. Після того як LLM генерує SQL-запит на основі скороченого промпту, запит послідовно проходить три рівні перевірки: перевірку read-only (відсутність деструктивних операторів), перевірку відповідності схемі (існування всіх таблиць і стовпців у наданому

контексті) та синтаксичну перевірку. Якщо запит не проходить будь-який із цих рівнів, він відхиляється без виконання. Лише запити, що успішно пройшли всі перевірки, виконуються через підключення до бази даних у режимі read-only, після чого результат повертається користувачеві.

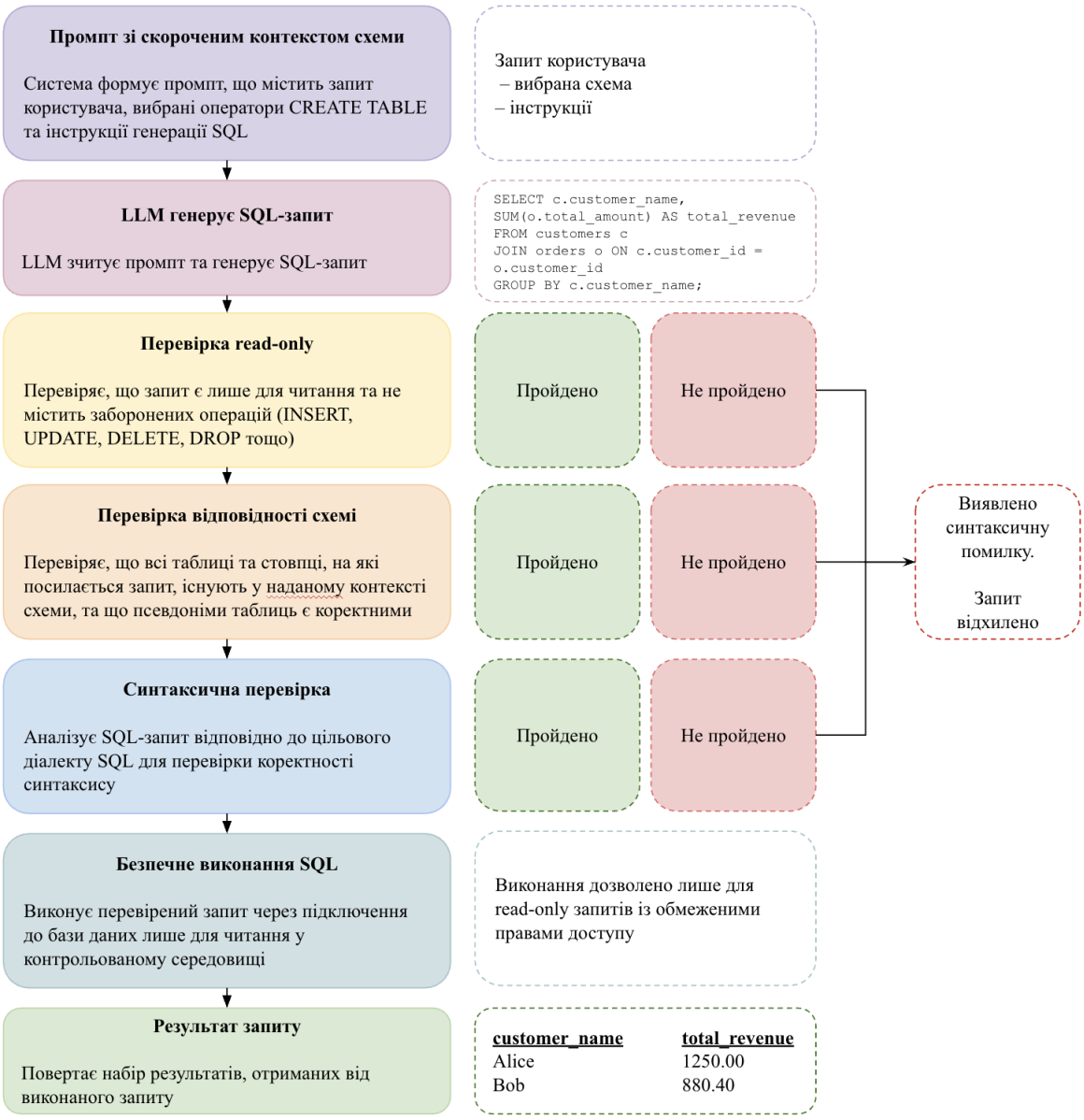


Рисунок 2.8 – Процес генерації, перевірки та безпечного виконання SQL-запиту

Якщо згенерований запит проходить перевірку, він виконується в базі даних у контрольованому середовищі. Система повинна використовувати

підключення до бази даних лише для читання або обмежувати права на виконання, щоб запобігти випадковій зміні даних. Це особливо важливо в виробничих середовищах, де автоматично згенерований SQL-запит не повинен мати можливості змінювати або видаляти дані.

Якщо запит не проходить перевірку, система не повинна його виконувати. Натомість вона може повернути повідомлення про помилку, попросити уточнення або запустити механізм виправлення. У рамках цієї кваліфікаційної роботи основна увага приділяється зменшенню затримки та підвищенню коректності за допомогою обрізки схеми, тоді як більш просунуті стратегії виправлення, такі як самовиправлення на основі виконання, можна розглядати як майбутні вдосконалення.

Результат виконаного SQL-запиту потім передається на наступний етап пайплайна. Залежно від архітектури системи, необроблений результат може бути повернутий безпосередньо або перетворений у відповідь природною мовою. У цій роботі результат виконання SQL розглядається як обґрунтований структурований контекст, який може бути використаний для генерації остаточної відповіді для користувача.

Цей інтегрований робочий процес, що поєднує генерацію та перевірку, є важливим, оскільки він поєднує здатність великої мови обробки (LLM) до логічного міркування з детермінованими засобами захисту на рівні системи. LLM відповідає за формування SQL-запиту, тоді як модуль перевірки гарантує, що запит є безпечним, відповідає схемі бази даних та придатним для виконання. У результаті система стає більш надійною та краще пристосованою для практичного використання з великими реляційними базами даних.

3 РЕАЛІЗАЦІЯ СИСТЕМИ TEXT-TO-SQL

3.1 Технологічний стек та середовище розробки

Реалізація запропонованої системи Text-to-SQL базується на модульній архітектурі, в якій кожен компонент пайплайна відповідає за окремий етап обробки. Ці етапи включають отримання запиту користувача, вилучення та індексацію метаданих схеми бази даних, вибір відповідних фрагментів схеми, формування запиту, генерацію SQL за допомогою великої мовної моделі, перевірку згенерованого запиту, його безпечне виконання та обробку кінцевого результату.

Python було обрано основною мовою програмування для реалізації, оскільки він забезпечує зрілу екосистему для розробки бекенду, обробки природної мови, взаємодії з базами даних та інтеграції з API великих мовних моделей. Крім того, Python широко використовується в робочих процесах машинного навчання та інженерії даних, що робить його придатним для розробки та експериментів з пайплайнами Text-to-SQL на основі LLM.

Для бекенд частину було обрано FastAPI, сучасний веб-фреймворк на Python для створення API на основі підказок щодо типів. FastAPI підходить для систем такого типу, оскільки підтримує асинхронну обробку запитів, автоматичну документацію API, перевірку за допомогою моделей даних та розробку API, орієнтовану на виробниче середовище. Ці властивості є корисними для сервісу Text-to-SQL, де запити користувачів мають оброблятися на кількох етапах пайплайна та можуть передбачати виклики зовнішніх моделей або операції з базою даних.

Як систему управління реляційними базами даних було використано PostgreSQL. Вона підходить для цієї роботи, оскільки завдання Text-to-SQL зосереджене на генеруванні SQL-запитів над структурованими реляційними даними. PostgreSQL підтримує стандартні концепції реляційних баз даних, такі як таблиці, стовпці, первинні ключі, зовнішні ключі, з’єднання,

агрегація, фільтрування та транзакції, які є необхідними для оцінки згенерованих SQL-запитів.

Для взаємодії зі схемою бази даних та моделями додатків можна використовувати ORM або клієнт бази даних. У контексті реалізації цієї роботи актуальним є Prisma Client для Python, оскільки його можна застосовувати в бекенд-додатках на Python, які потребують доступу до бази даних. Загальний робочий процес Prisma Client базується на визначенні моделей у схемі Prisma, генеруванні клієнта та використанні згенерованого API для взаємодії з базою даних. У запропонованому пайплайні такий рівень доступу до бази даних є корисним для отримання структурованої інформації про схему та підтримки взаємодії з реляційною базою даних.

Компонент LLM відповідає за генерацію SQL-запитів на основі сформованого промпта. У запропонованій системі мовна модель використовується саме як модуль генерації SQL, тоді як вибір схеми, побудова промпта, валідація та контроль виконання здійснюються зовнішніми компонентами пайплайна. Ця конструкція є важливою, оскільки головний внесок кваліфікаційної роботи полягає не лише у використанні LLM, а в інтеграції моделі в керований пайплайн Text-to-SQL з динамічним обрізанням схеми. Середовище розробки також включає управління конфігурацією для ключів API, рядків підключення до бази даних, параметрів моделі та налаштувань, специфічних для середовища. Ці параметри не повинні бути жорстко закодовані безпосередньо у вихідному коді. Натомість їх можна зберігати у змінних середовища або файлах конфігурації, таких як файл .env. Це спрощує розгортання, тестування та модифікацію системи.

Середовище розробки також передбачає управління конфігурацією ключів API, рядків підключення до баз даних, параметрів моделей та налаштувань, що залежать від середовища. Ці параметри не слід вказувати безпосередньо у вихідному коді. Натомість їх можна зберігати у змінних

середовищах або конфігураційних файлах, таких як файл .env. Це спрощує розгортання, тестування та модифікацію системи без зміни її реалізації.

Обраний стек забезпечує реалізацію основних цілей запропонованої системи. FastAPI надає легкий та модульний рівень API, PostgreSQL – реалістичне середовище реляційної бази даних, а pgAdmin – можливість ручної перевірки та валідації згенерованих SQL-запитів. Рівень доступу до бази даних дозволяє отримувати та обробляти метадані схеми, тоді як API LLM забезпечує перетворення природної мови в SQL. Водночас модулі налаштованого скорочення та валідації схеми роблять систему більш керованою, ніж простий однокроковий виклик LLM.

Тому реалізацію можна розглядати як послідовність взаємодіючих модулів, а не як монолітний додаток. Кожен модуль має чітко визначену відповідальність: витяг схеми готує метадані, індексація схеми зберігає визначення таблиць, динамічне обрізання схеми відбирає відповідні фрагменти, побудова підказки готує вхідні дані для моделі, LLM генерує SQL, а рівень перевірки визначає, чи можна безпечно виконати запит. Таке розділення полегшує налагодження, оцінку та розширення системи в майбутніх роботах.

3.2 Витяг схеми бази даних та побудова індексів

На другому етапі реалізації відбувається вилучення схеми бази даних та побудова індексу. На цьому етапі готуються структурні метадані, необхідні для динамічного скорочення схеми. Замість того, щоб надсилати повну схему бази даних до мовної моделі для кожного запиту користувача, система спочатку вилучає метадані бази даних, перетворює їх на компактне представлення на рівні таблиць і зберігає в індексі, який можна запитувати під час онлайн-обробки.

У PostgreSQL метадані схеми можна отримати з системних таблиць метаданих. information_schema містить представлення, що описують

об’єкти бази даних, доступні в поточній базі даних, і визначається стандартом SQL, що робить її стабільним та переносимим способом перевірки структури бази даних. Зокрема, представлення information_schema.tables містить інформацію про таблиці та представлення, доступні поточному користувачеві, тоді як information_schema.columns містить інформацію про стовпці таблиць, включаючи їхні імена, типи даних, можливість прийняття нульових значень та інші властивості на рівні стовпців.

Метадані схеми, що витягуються, містять імена схем баз даних, імена таблиць, імена стовпців, типи даних, обмеження щодо дозволених нульових значень та ключові зв’язки (за наявності). Ця інформація є необхідною, оскільки мовна модель має розуміти, які таблиці та стовпці існують, перш ніж вона зможе згенерувати правильний SQL-запит. Без цих метаданих модель може спиратися на припущення та генерувати вигадані імена таблиць або стовпців.

У реалізованому пайплайн схема представлена на рівні таблиць. Кожна таблиця перетворюється на текстове представлення, схоже на DDL, яке відповідає структурі оператора CREATE TABLE. Офіційна документація PostgreSQL визначає CREATE TABLE як команду, що використовується для створення таблиці шляхом зазначення імені таблиці, імен стовпців та типів даних стовпців. Цей формат є корисним для генерації «Текст-в-SQL», оскільки він є компактним, структурованим і вже знайомим для мовних моделей, навчених на даних, пов’язаних із SQL.

На рисунку 3.1 наведено програмний код витягу SQL-визначень таблиць, представлень і матеріалізованих представлень.

Для кожної таблиці в базі даних генерується подібне представлення, яке зберігається окремо. Це важливо, оскільки запропонований підхід до динамічного скорочення схеми вимагає від системи вилучення окремих визначень таблиць, а не одного великого документа схеми.

```sql
CREATE TABLE customers (
    customer_id INTEGER PRIMARY KEY,
    full_name TEXT,
    email TEXT,
    city TEXT,
    created_at TIMESTAMP
);
```

Рисунок 3.1 – Програмний код витягу SQL-визначень таблиць, представлень і матеріалізованих представлень

Індекс схеми можна представити у вигляді структури «ключ-значення», де ключем є назва таблиці, а значенням – відповідне представлення DDL.

На практиці індекс може також містити додаткові поля метаданих, такі як імена стовпців, зв’язки, описи таблиць або посилання на зовнішні ключі. Однак основним елементом індексу залишається відповідність між кожною таблицею та її оператором CREATE TABLE. Ця відповідність забезпечує швидке та економічне отримання схеми під час обробки запитів у режимі реального часу.

Цей індекс схеми створюється до генерації SQL і може оновлюватися щоразу, коли змінюється структура бази даних. Як наслідок, онлайн-частині пайплайна Text-to-SQL не потрібно повторно обробляти повну схему бази даних. Натомість вона може отримувати лише ті визначення таблиць, які мають відношення до поточного запиту користувача.

Якщо використовується ORM або клієнт бази даних, витяг схеми також може підтримуватися за допомогою інтроспекції. Наприклад, Prisma забезпечує інтроспекцію бази даних за допомогою команди prisma db pull, яка підключається до бази даних і відображає поточну схему бази даних у моделях Prisma. Це може бути корисним для розуміння існуючих структур

баз даних та узгодження представлення схеми на рівні додатка з фактичною базою даних.

Головною метою цього етапу є перетворення схеми бази даних з великого монолітного об’єкта на сукупність фрагментів схеми, які можна отримувати незалежно один від одного. Це перетворення є необхідним для запропонованого підходу, оскільки динамічне скорочення схеми залежить від можливості вибрати та ввести в підказку лише підмножину схеми.

Таким чином, вилучення схеми бази даних та побудова індексу становлять основу для всього пайплайна Text-to-SQL. Представляючи кожну таблицю як окремий фрагмент DDL та зберігаючи його в індексованій структурі, система отримує можливість зменшувати розмір підказки, підвищувати ефективність пошуку та підтримувати цільове введення схеми під час генерації SQL.

3.3 Реалізація динамічного скорочення схеми

Динамічне скорочення схеми було реалізовано як основний механізм оптимізації запропонованого пайплайна Text-to-SQL. Його мета – зменшити обсяг схеми бази даних, що надається великій мовній моделі під час генерації SQL. Замість того, щоб надсилати повну схему бази даних для кожного запиту користувача, система спочатку визначає таблиці, які найбільше відповідають наміру користувача, а потім формує компактний фрагмент схеми, що містить лише ці вибрані таблиці.

Ця реалізація безпосередньо підтримує головну мету кваліфікаційної роботи: зменшення розміру запиту та затримки генерації при збереженні або покращенні коректності SQL. Обмежуючи контекст схеми, доступний для моделі, система зменшує обсяг нерелевантної інформації, звужує простір пошуку для генерації SQL та знижує ймовірність генерації запитів до нерелевантних таблиць або неіснуючих стовпців.

Механізм динамічного скорочення схеми складається з чотирьох основних етапів:`

– розклад схеми на одиниці рівня таблиць;

– побудова індексу схеми на рівні таблиць;

– оцінка релевантності під час виконання запиту та вибір таблиць;

– вставлення скороченого фрагмента схеми в запит на генерацію SQL.

Спершу було зроблено розклад схеми. Схема бази даних завантажується як текстове визначення SQL, а потім розбивається на окремі визначення таблиць, представлень або матеріалізованих представлень. Це перетворює схему з одного великого документа на менші одиниці, які можна витягувати. Кожна витягнута інструкція зберігається окремо, що дозволяє системі згодом витягувати лише ті фрагменти схеми, які необхідні для конкретного запиту користувача.

Витягування реалізовано за допомогою регулярного виразу, який ідентифікує інструкції CREATE TABLE, CREATE VIEW та CREATE MATERIALIZED VIEW.

На рисунку 3.2 наведено програмний код відповідності ключових слів релевантним таблицям.

```
stmt_pattern = re.compile(
    r"(CREATE\s+(?:TABLE|VIEW|MATERIALIZED\s+VIEW)\s+([^\s(]+).*?;)",
    re.DOTALL | re.IGNORECASE,
)

for match in stmt_pattern.finditer(schema_text):
    full_statement = match.group(1).strip()
    table_name = match.group(2)
    key = table_name.replace('"', "").lower()
    schema_index[key] = full_statement
```

Рисунок 3.2 – Програмний код відповідності ключових слів релевантним таблицям

Цей етап індексації виконується один раз під час ініціалізації служби генерації SQL. Утворена структура пов’язує нормалізовані імена таблиць із

їхніми повними визначеннями в SQL. Наприклад, така таблиця, як public.airnodes, зберігається як окремий запис в індексі схеми, де ключем є нормалізоване ім’я таблиці, а значенням – повне визначення цієї таблиці в SQL.

Ця структура є важливою, оскільки забезпечує ефективне вилучення схем під час обробки запитів. Системі більше не потрібно вставляти повну схему в кожен запит. Натомість вона може відбирати лише невелику кількість відповідних визначень схем.

На другому етапі, оцінюється релевантності під час запиту. Для кожного запиту користувача система обчислює релевантність для кожної індексованої таблиці. Функція оцінки поєднує кілька простих стратегій зіставлення: пряме зіставлення назв таблиць, зіставлення однини, зіставлення назв схем та зіставлення ключових слів, специфічних для домену. Цей підхід було обрано, оскільки він простий, швидкий та підходить для зменшення затримки в орієнтованому на виробництво пайплайні Text-to-SQL.

На рисунку 3.3 наведено програмний код оцінювання релевантності таблиць.

```
for table_key, statement in schema_index.items():
    score = 0
    base_name = table_key.split(".")[-1]
    singular = base_name[:-1] if base_name.endswith("s") else ""

    if base_name and base_name in prompt_lower:
        score += 5

    if singular and singular in prompt_lower:
        score += 3

    schema_name = table_key.split(".")[0]
    if schema_name and schema_name in prompt_lower:
        score += 1

    for keyword, tables in keyword_table_map.items():
        if keyword in prompt_lower and table_key in tables:
            score += 3

    if score > 0:
        table_scores.append((score, table_key, statement))
```

Рисунок 3.3 – Програмний код оцінювання релевантності таблиць

Прямі збіги з назвами таблиць отримують найвищу вагу, оскільки вони є найпереконливішим свідченням того, що запит користувача стосується конкретної таблиці. Збіг у однині використовується для врахування простих лексичних відмінностей між мовою користувача та правилами іменування в базі даних. Наприклад, користувач може вживати слово «завдання», тоді як таблиця в базі даних має назву «завдання». Збіг з назвою схеми надає додатковий слабкий сигнал, коли запит користувача містить назву схеми бази даних.

Однак користувачі рідко знають точні назви таблиць бази даних. З цієї причини система також використовує відображення ключових слів, специфічне для домену. Це відображення пов’язує лексику, з якою стикається користувач, із внутрішніми структурами бази даних. Наприклад, користувач може запитати про «ціни AirNode», «покриття», «завдання» або «ціну WMTX», не знаючи, у яких таблицях зберігається ця інформація. Відображення ключових слів заповнює цю прогалину, пов’язуючи такі терміни з відповідними таблицями.

На рисунку 3.4 наведено приклад відповідності ключових слів, що використовується в реалізації.

```
keyword_table_map = {
    "airnode": [
        "public.airnodes",
        "public.airnode_inventory",
        "public.host_locations",
    ],
    "coverage": [
        "public.hexes",
        "public.v_airnodes_per_hex",
    ],
    "pricing": [
        "public.product_pricing_matrix_items",
    ],
    "task": [
        "public.tasks",
    ],
    "wmtx": [
        "public.token_spot_prices",
    ],
}
```

Рисунок 3.4 – Приклад відповідності ключових слів

Таке зіставлення є особливо корисним для галузевих баз даних, де назви таблиць можуть бути незрозумілими для кінцевих користувачів. Воно підвищує ймовірність того, що модуль відбору вибере правильні фрагменти схеми навіть у тому випадку, якщо запит користувача не містить точних назв таблиць.

Після обчислення оцінок релевантності система ранжує таблиці за оцінками та вибирає лише підмножину з найвищими оцінками. У реалізованій системі максимальна кількість вибраних таблиць за замовчуванням становить вісім. Це обмеження запобігає надмірному збільшенню розміру підказки, водночас дозволяючи моделі отримувати достатній контекст схеми для запитів середньої складності.

На рисунку 3.5 наведено програмний код вибору найрелевантніших таблиць.

```
if not table_scores:
    fallback_tables = list(schema_index.items())[:max_tables]
    return "\n\n".join(statement for _, statement in fallback_tables)

table_scores.sort(key=lambda item: (-item[0], item[1]))
selected_tables = table_scores[:max_tables]

schema_snippet = "\n\n".join(
    statement for _, _, statement in selected_tables
)
```

Рисунок 3.5 – Програмний код вибору найрелевантніших таблиць

Якщо жодна таблиця не отримає позитивного показника релевантності, система не переходить до запиту з повною схемою. Натомість вона повертає невелику підмножину таблиць за замовчуванням. Це проектне рішення є важливим, оскільки воно зберігає головну мету скорочення схеми: запобігання обробці моделлю повної схеми бази даних. Хоча такий запасний варіант може не завжди містити всі необхідні таблиці, він дозволяє уникнути затримок та ризиків «галюцинацій», пов’язаних із розкриттям повної схеми.

Вибрані визначення таблиць потім об'єднуються у фрагмент схеми з обрізанням. Цей фрагмент стає єдиним контекстом схеми, видимим для мовної моделі під час генерації SQL. Моделі явно вказується використовувати лише таблиці та стовпці, включені до фрагмента. Якщо необхідна інформація відсутня, модель повинна повернути відповідь поза контекстом замість того, щоб вигадувати запит.

На рисунку 3.6 зображено промпт для перевірки read-only SQL-запиту.

```
Use ONLY tables and columns that appear in the provided schema snippet.
If anything needed is missing, return:

{
  "sql_query": "none",
  "out_of_context": true,
  "extra": "short reason"
}
```

Рисунок 3.6 – Промпт для перевірки read-only SQL-запиту

Ця інструкція робить скорочення схеми частиною механізму безпеки системи, а не лише засобом оптимізації продуктивності. Обмежуючи модель обраним фрагментом схеми, система знижує ймовірність недійсних з’єднань, неправильного вибору таблиць та виникнення «галюцинованих» стовпців. Якщо обраний контекст схеми є недостатнім, модель має вказати, що запит не відповідає контексту, а не генерувати SQL-запит, який не підтримується.

На додаток до обрізки схеми, система застосовує фільтрацію намірів перед генерацією SQL. SQL генерується лише тоді, коли запит користувача чітко пов'язаний зі структурованим доступом до бази даних, таким як підрахунки, підсумки, списки, фільтри, рейтинги або пошук конкретних записів. Запитання, що вимагають пояснення, усунення несправностей або загальних знань, не надходять до генерації SQL і обробляються за допомогою генерації, доповненої пошуком.

На рисунку 3.7 наведено інструкцію щодо фільтрації намірів.

```
Generate SQL only when the user is requesting structured data,
such as counts, totals, listings, filters, rankings, or specific records.

For explanation, how-to, troubleshooting, or knowledge-base questions,
do not generate SQL.
```

Рисунок 3.7 – Інструкція щодо фільтрації намірів

Цей додатковий крок запобігає непотрібному формуванню SQL-запитів для питань, на які не слід відповідати за допомогою запитів до бази даних. У результаті система стає більш надійною, оскільки вона відокремлює запити на структуровані дані від запитів, що базуються на знаннях або мають пояснювальний характер.

Нарешті, після формування SQL-запиту система перед його виконанням застосовує рівень перевірки, що працює лише в режимі читання. Запит повинен починатися з SELECT або WITH; операції, що змінюють дані, такі як INSERT, UPDATE, DELETE, команди DDL або множинні оператори, відхиляються.

Загалом, динамічне обрізання схеми було реалізовано як механізм вибору схеми під час запиту. Система спочатку розкладає схему бази даних на одиниці рівня таблиць, оцінює ці одиниці відповідно до запиту користувача, вибирає найбільш релевантну підмножину та вводить лише цю підмножину в запит для генерації SQL. Ця реалізація зменшує розмір контексту, покращує ефективність запиту та обмежує мовну модель до меншого та безпечнішого простору пошуку.

У поєднанні з відображенням ключових слів для конкретної галузі, фільтрацією намірів та перевіркою SQL «тільки для читання» механізм обрізки забезпечує практичну основу для контрольованого доступу природною мовою до складної реляційної бази даних. Ця реалізація

підтримує головну мету запропонованої системи: підвищення ефективності та надійності генерації Text-to-SQL у середовищах великих баз даних.

3.4 Побудова промптів, генерація SQL, валідація та обробка отриманих результатів

Після того як на етапі динамічного відбору схем відібрано найбільш релевантні фрагменти схем, система формує остаточний запит для генерації SQL. Цей запит містить запитання користувача, відібраний фрагмент схеми та чіткі інструкції, що визначають, як має поводитися мовна модель. Мета цього етапу – надати моделі достатньо контексту для генерації коректного SQL-запиту, одночасно запобігаючи використанню таблиць або стовпців, які не були відібрані модулем відбору.

Підказка побудована навколо трьох основних компонентів: фрагмента схеми, запиту користувача та обмежень генерації. Фрагмент схеми містить лише вибрані оператори CREATE TABLE, CREATE VIEW або CREATE MATERIALIZED VIEW. Запит користувача зберігає оригінальне запитання природною мовою. Обмеження генерації вказують моделі генерувати SQL лише тоді, коли запитання вимагає структурованого доступу до бази даних, та використовувати лише надану схему.

На рисунку 3.8 зображено спрощену структуру запиту.

Така конструкція запиту забезпечує безпосередній вплив етапу скорочення схеми на генерацію SQL. Оскільки модель може використовувати лише видимий фрагмент схеми, ймовірність генерації запитів до таблиць, що не мають відношення до контексту, або до вигаданих стовпців зменшується. Якщо необхідна інформація відсутня у вибраному контексті схеми, модель отримує вказівку повернути відповідь, що не відповідає контексту, замість того, щоб вигадувати запит.

Етап генерації SQL виконується великою мовною моделлю. Модель отримує сконструйовану підказку і повертає структуровану відповідь, що

містить або згенерований SQL-запит, або вказівку на те, що генерація SQL не підходить для запиту користувача. Це важливо, оскільки не на всі питання слід відповідати за допомогою SQL. Наприклад, питання щодо усунення несправностей, пояснень або загальних знань краще обробляти за допомогою генерації, доповненої пошуком, а не прямим запитом до бази даних.

```
You are an assistant that generates SQL queries.

Use ONLY tables and columns that appear in the provided schema snippet.
Generate SQL only when the user is requesting structured data,
such as counts, totals, listings, filters, rankings, or specific records.

For explanation, how-to, troubleshooting, or knowledge-base questions,
do not generate SQL.

Schema snippet:
{schema_snippet}

User question:
{user_question}

Return the result in JSON format:
{
  "sql_query": "...",
  "out_of_context": false,
  "extra": ""
}
```

Рисунок 3.8 – Спрощена структура запиту

Після генерації SQL-запиту він не виконується одразу. Система спочатку застосовує рівень валідації, щоб переконатися, що запит є безпечним і придатним для виконання. Цей крок є необхідним, оскільки SQL, згенерований LLM, може містити недійсну синтаксичну конструкцію, непідтримувані операції або посилання на таблиці та стовпці, які недоступні у вибраному контексті схеми.

На другому етапі перевірки перевіряється, чи відповідає згенерований SQL-запит наявній схемі. У запиті повинні використовуватися лише ті таблиці та стовпці, які були включені до скороченого фрагмента схеми. Така перевірка допомагає уникнути випадків, коли модель вигадує назву стовпця,

вибирає таблицю, що не має відношення до запиту, або будує з’єднання на основі недоступної інформації про схему.

Якщо запит проходить перевірку, він виконується у базі даних у контрольованому режимі «тільки для читання». Мета цього етапу – отримати фактичний структурований результат із бази даних. Результат виконання може містити рядки, агреговані значення, відфільтровані записи або результати ранжування залежно від запиту користувача.

На рисунку 3.9 зображено повний робочий процес генерації, перевірки та обробки SQL-запиту. Пайплайн послідовно охоплює сім етапів: формування скороченого фрагмента схеми, побудову промпту, генерацію SQL на основі LLM, перевірку read-only, перевірку відповідності схемі, безпечне виконання SQL та обробку результатів. Така послідовність гарантує, що до бази даних передаються лише синтаксично коректні, семантично узгоджені та безпечні SQL-запити.



Рисунок 3.9 – Повний пайплайн генерації, перевірки та обробки SQL-запиту

Після виконання результат SQL обробляється, перш ніж бути повернутий користувачеві або переданий на етап формування остаточної відповіді. Якщо результат є невеликим і зрозумілим без додаткових пояснень, його можна повернути у вигляді структурованих даних. Якщо результат потребує пояснення, його можна перетворити на відповідь природною мовою на основі фактичних даних з бази. Це гарантує, що остаточна відповідь ґрунтується на реальних результатах бази даних, а не на безпідставних припущеннях моделі.

Якщо перевірка не проходить, система не виконує запит. Натомість вона може повернути помилку, позначити запит як такий, що не відповідає контексту, або перенаправити запитання до пайплайна пошуку, що не використовує SQL. Така поведінка є важливою для забезпечення безпеки та надійності, особливо коли вихідні дані моделі є невизначеними або коли вибраний фрагмент схеми не містить достатньо інформації для відповіді на запитання.

Таким чином, остаточний етап реалізації поєднує побудову запиту, генерацію SQL, перевірку, виконання та обробку результатів в один контрольований робочий процес. Модель мови відповідає за генерацію SQL-запиту, але супутні компоненти системи контролюють контекст схеми, обмежують небезпечні операції, перевіряють згенерований запит та гарантують, що остаточна відповідь базується на фактичних даних бази даних. Ця конструкція робить запропонований пайплайн «Текст-в-SQL» більш надійним та придатним для практичного використання зі складними реляційними базами даних.

4 ЕКСПЕРИМЕНТАЛЬНА ОЦІНКА ЗАПРОПОНОВАНОГО ПІДХОДУ

4.1 Налаштування оцінювання

Метою експериментальної оцінки було дослідження ефективності динамічного скорочення схеми та вибору оптимальної конфігурації моделі для пайплайну Text-to-SQL. Експеримент проводився у два послідовні етапи.

На першому етапі оцінювався ізольований вплив динамічного скорочення схеми: базовий пайплайн із передачею повної схеми порівнювався з пайплайном із динамічним скороченням схеми при однаковій базовій моделі. Це дозволило розмежувати ефект скорочення схеми від ефекту зміни моделі.

На другому етапі, після підтвердження ефективності скорочення схеми, досліджувався вплив оновленого системного промпту та різних великих мовних моделей на показники часу генерації та коректності SQL-запитів. Для цього було протестовано чотири моделі: Claude Haiku 3.1, Llama 3.1 8B Instant, Llama 4 Scout 17B 16E та GPT OSS 20B усі через API Groq із однаковим параметром max_tokens = 1200.

Оцінка проводилася на спеціалізованій реляційній базі даних із великою схемою, що охоплює кілька функціональних областей. Тестовий набір містив понад 200 запитів природною мовою, об'єднаних у 9 категорій: test-check-1, network-builder, analytics, air-node, earth-node, revenue, predictive, affiliates та eSim/AMS.

Категорії охоплювали різні типи структурованих запитів:

– пошук записів;

– фільтрування;

– агрегацію;

– ранжування;

– галузеву аналітику.

Згенеровані SQL-запити виконувалися на базі даних, а для ручної перевірки результатів використовувався pgAdmin. Запит вважався успішним, якщо він був синтаксично правильним, виконуваним та семантично відповідав наміру користувача.

4.2 Конфігурації порівняння

В експерименті було протестовано шість конфігурацій пайплайну, згрупованих у три етапи дослідження.

Базовий пайплайн. Використовувалася модель claude-sonnet-4-5 із передачею повної схеми бази даних у промпт та максимальною довжиною виводу 8192 токени. Ця конфігурація є точкою відліку для оцінки всіх наступних оптимізацій.

Пайплайн із динамічним скороченням схеми. Та сама базова модель, але замість повної схеми до промпту передавалися лише релевантні таблиці, відібрані модулем вибору схеми. Порівняння цієї конфігурації з базовою дозволяє ізолювати вплив скорочення схеми незалежно від зміни моделі.

Пайплайн з оновленим промптом та різними моделями. На основі результатів попереднього етапу було розроблено оновлений системний промпт із більш чіткими інструкціями щодо генерації SQL, форматом відповіді JSON та обмеженням max_tokens = 1200.

Цей промпт тестувався з чотирма моделями через API Groq:

– Claude Haiku 3.1;

– Llama 3.1 8B Instant;

– Llama 4 Scout 17B 16E;

– GPT OSS 20B.

У таблиці 4.1 наведено зведені результати всіх конфігурацій за показниками середнього часу генерації, частки успішних запитів та відносного скорочення часу відносно базової конфігурації.

Таблиця 4.1 – Порівняння базового та оптимізованого пайплайна

№	Конфігурація	Середній час (с)	Успішних (%)	Зміна часу (%)
1	Повна схема + claude-sonnet-4-5	54.62	51.77	-
2	Динамічне скорочення схеми + claude-sonnet-4-5	4.18	60.38	-23.07
3	Новий промпт + Claude Haiku 3.1	2.45	33.87	-56.98
4	Новий промпт + Llama 3.1 8B Instant	0.77	33.81	-81.04
5	Новий промпт + Llama 4 Scout 17B 16E	0.55	685	-86.75
6	Новий промпт + GPT OSS 20B	0.84	38.02	-79.17

4.3 Результати експерименту та їх аналіз

Експериментальна оцінка проводилася у два послідовні етапи, результати яких наведено в таблиці 4.1. На першому етапі порівнювався базовий пайплайн та пайплайн із динамічним скороченням схеми при незмінній моделі. Середній час генерації SQL-запиту скоротився з 5,4621 с до 4,1814 с, що становить зменшення на 23,07%, а частка успішних запитів зросла з 51,77% до 60,38%. Ці результати підтверджують, що динамічне скорочення схеми саме по собі покращує обидва ключові показники: зменшення кількості токенів у промпті прискорює генерацію, а усунення нерелевантних таблиць знижує ймовірність вибору неправильних елементів схеми.

На другому етапі тестувалися чотири моделі з оновленим системним промптом та параметром max_tokens = 1200. Серед них лише Llama 4 Scout 17B 16E забезпечила одночасно найнижчу затримку та найвищу частку успішних запитів (таблиця 4.1). Claude Haiku 3.1 та Llama 3.1 8B Instant показали значне прискорення генерації, проте частка успішних запитів впала до 33,87% та 33,81% відповідно – нижче за базову конфігурацію. GPT OSS 20B продемонстрував помірне прискорення з

часткою успішних запитів 38,02%. Це свідчить про те, що зменшення затримки не гарантує покращення коректності і вибір моделі має критичне значення для якості генерації SQL у великомасштабних схемах.

За сукупністю показників оптимальною виявилася конфігурація з моделлю Llama 4 Scout 17B 16E, оновленим системним промптом та динамічним скороченням схеми. У таблиці 4.2 наведено порівняння базового пайплайну та оптимальної конфігурації за усіма ключовими показниками.

Таблиця 4.2 – Порівняння базового пайплайну та оптимальної конфігурації

Показник	Базовий пайплайн	Оптимальна конфігурація	Зміна
Середній час генерації (с)	5.46	0.55	-86.75
Частка успішних запитів (%)	51.77	69.85	18.08
Провайдер	Anthropic	Groq	-
Модель	claude-sonnet-4-5	Llama 4 Scout 17B 16E	-
Контекст схеми	Повна схема	Динамічне скорочення	-
Максимум токенів виводу	8192	1200	-
Формат відповіді	Текст	JSON	-

Аналіз результатів на рівні окремих категорій показує, що найбільше покращення спостерігалося в категоріях air-node, earth-node, revenue та network-builder – там, де запитання користувачів містять термінологію, безпосередньо пов'язану з назвами таблиць бази даних. Нижчі показники в категоріях predictive та analytics свідчать про те, що аналітичні запити з неявними зв'язками між таблицями потребують більш складних механізмів вибору схеми, ніж зіставлення ключових слів.

Необхідно зазначити, що на другому етапі одночасно змінювалися системний промпт, модель та параметр max_tokens, тому спостережуване покращення є результатом їх комбінованого впливу. Ізоляція внеску кожного з цих факторів є предметом подальших досліджень.

Загалом результати підтверджують ефективність запропонованого підходу: динамічне скорочення схеми у поєднанні з оптимізованим промптом та моделлю Llama 4 Scout 17B 16E дозволило скоротити середній час генерації на 86,75% та підвищити частку успішних запитів з 51,77% до 69,85%, що підтверджує головну гіпотезу роботи.

ВИСНОВКИ

У кваліфікаційній роботі було проведено дослідження та розробку Text-to-SQL пайплайну для автоматизованої генерації SQL-запитів на основі запитів природною мовою.

Основною метою роботи було створення та оцінка підходу, який дозволяє зменшити затримку генерації SQL-запитів і підвищити їхню коректність у середовищі великомасштабної реляційної бази даних. Для досягнення цієї мети було запропоновано використання динамічного скорочення схеми, за якого до великої мовної моделі передається не повна схема бази даних, а лише релевантні фрагменти схеми для конкретного запиту користувача.

У теоретичній частині було проаналізовано предметну галузь Text-to-SQL, особливості використання великих мовних моделей для генерації SQL-запитів, а також сучасні підходи до покращення якості генерації, зокрема Retrieval-Augmented Generation, динамічне скорочення схеми, багатоагентні архітектури та самокорекцію на основі результатів виконання. Було визначено, що одним із головних викликів у великих базах даних є компроміс між точністю SQL-запиту та затримкою відповіді системи.

У практичній частині було спроєктовано та реалізовано Text-to-SQL пайплайн із динамічним скороченням схеми. Було реалізовано витяг SQL-визначень таблиць, побудову індексу схеми, вибір релевантних таблиць, формування скороченого контексту схеми, побудову промпту, генерацію SQL-запиту та його валідацію перед виконанням. Для тестування та перевірки коректності згенерованих SQL-запитів використовувалися PostgreSQL і pgAdmin.

Експериментальна оцінка була проведена на наборі з понад 200 тестових запитань, згрупованих у 9 категорій. У результаті оптимізований пайплайн зменшив середній час генерації SQL-запиту з 5,4621 с до 0,5514 с,

що відповідає скороченню затримки на 86,75%. Частка успішно згенерованих SQL-запитів зросла з 51,77% до 69,85%.

Отримані результати підтверджують ефективність динамічного скорочення схеми для покращення швидкодії та коректності Text-to-SQL систем. Запропонований підхід є практично корисним для систем, що працюють із великими реляційними схемами та потребують швидкого доступу до даних природною мовою.

Перспективами подальшого розвитку є використання embedding-based retrieval для точнішого вибору релевантних таблиць, додавання execution-guided self-correction, багатоагентної валідації SQL-запитів та проведення оцінювання на відкритих Text-to-SQL бенчмарках.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1) A Survey of text-to-sql in the era of LLMs: where are we, and where are we going? / X. Liu et al. *arXiv*. 2025. URL: https://doi.org/10.48550/arXiv.2408.05109 (date of access: 18.05.2026).

2) A survey on employing large language models for text-to-sql tasks / L. Shi et al. *ACM computing surveys*. 2025. URL: https://doi.org/10.1145/3737873 (date of access: 18.05.2026).

3) A survey on text-to-sql parsing: concepts, methods, and future directions / B. Qin et al. *arXiv*. 2021. URL: https://doi.org/10.48550/arXiv.2208.13629 (date of access: 18.05.2026).

4) Large language model enhanced text-to-sql generation: a survey / X. Zhu et al. *arXiv*. 2022. URL: https://doi.org/10.48550/arXiv.2410.06011 (date of access: 18.05.2026).

5) Rethinking data in NL2SQL: a survey of what we have and what we expect / Y. Fan et al. *Vicinagearth*. 2025. Vol. 2, no. 1. URL: https://doi.org/10.1007/s44336-025-00026-9 (date of access: 18.05.2026).

6) The dawn of natural language to SQL: are we fully ready? / B. Li et al. *Proceedings of the VLDB Endowment*. 2024. Vol. 17, no. 11. P. 3318–3331. URL: https://doi.org/10.14778/3681954.3682003 (date of access: 18.05.2026).

7) PARSQL: enhancing text-to-sql through SQL parsing and reasoning / Y. Dai et al. *Findings of the association for computational linguistics: ACL 2025*, Vienna, Austria. Stroudsburg, PA, USA, 2025. P. 661–681. URL: https://doi.org/10.18653/v1/2025.findings-acl.37 (date of access: 18.05.2026).

8) Semantic parsing with syntax- and table-aware SQL generation / Y. Sun et al. *Proceedings of the 56th annual meeting of the association for computational linguistics (volume 1: long papers)*, Melbourne, Australia. Stroudsburg, PA, USA, 2018. URL: https://doi.org/10.18653/v1/p18-1034 (date of access: 18.05.2026).

9) RECPARSER: a recursive semantic parsing framework for text-to-sql task / Y. Zeng et al. *Twenty-ninth international joint conference on artificial intelligence and seventeenth pacific rim international conference on artificial intelligence {IJCAI-PRICAI-20}*, Yokohama, Japan, 11–17 July 2020. California, 2020. URL: https://doi.org/10.24963/ijcai.2020/504 (date of access: 18.05.2026).

10) PARROT: A benchmark for evaluating llms in cross-system SQL translation / W. Zhou et al. *arXiv*. 2024. URL: https://doi.org/10.48550/arXiv.2509.23338 (date of access: 18.05.2026).

11) Samantapudi R. K. R., Dhanagari M. R., Tarun S. Natural Language Query to SQL query generation using LSTMs, Transformers, LLMs and Gen AI. *American journal of technology*. 2025. Vol. 4, no. 1. P. 30–55. URL: https://doi.org/10.58425/ajt.v4i1.352 (date of access: 18.05.2026).

12) Spider: a large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task / T. Yu et al. *Proceedings of the 2018 conference on empirical methods in natural language processing*, Brussels, Belgium. Stroudsburg, PA, USA, 2018. URL: https://doi.org/10.18653/v1/d18-1425 (date of access: 18.05.2026).

13) Spider 2.0: evaluating language models on real-world enterprise text-to-sql workflows / F. Lei et al. *arXiv*. 2025. URL: https://doi.org/10.48550/arXiv.2411.07763 (date of access: 18.05.2026).

14) Spider 2.0. Official benchmark website. *Spider 2.0*. URL: https://spider2-sql.github.io/ (date of access: 18.05.2026).

15) Eben J., Ahmad A., Lau S. RASL: retrieval augmented schema linking for massive database text-to-sql. *arXiv*. 2024. URL: https://doi.org/10.48550/arXiv.2507.23104 (date of access: 18.05.2026).

16) Pourreza M., Rafiei D. DIN-SQL: Decomposed In-Context Learning of Text-to-SQL with Self-Correction. *arXiv*. 2023. URL: https://doi.org/10.48550/arXiv.2304.11015 (date of access: 18.05.2026).

17) Scholak T., Schucher N., Bahdanau D. PICARD: parsing incrementally for constrained auto-regressive decoding from language models. *Proceedings of*

the 2021 conference on empirical methods in natural language processing, Online and Punta Cana, Dominican Republic. Stroudsburg, PA, USA, 2021. URL: https://doi.org/10.18653/v1/2021.emnlp-main.779 (date of access: 18.05.2026).

18) Chapter 35. The information schema. *PostgreSQL Documentation*. URL: https://www.postgresql.org/docs/current/information-schema.html (date of access: 18.05.2026).

19) 35.17. The information schema: columns. *PostgreSQL Documentation*. URL: https://www.postgresql.org/docs/current/infoschema-columns.html (date of access: 18.05.2026).

20) 35.54. The information schema: tables. *PostgreSQL Documentation*. URL: https://www.postgresql.org/docs/current/infoschema-tables.html (date of access: 18.05.2026).

21) Create table. *PostgreSQL Documentation*. URL: https://www.postgresql.org/docs/current/sql-createtable.html (date of access: 18.05.2026).

22) PostgreSQL 18.3, 17.9, 16.13, 15.17, and 14.22 released. *PostgreSQL News*. URL: https://www.postgresql.org/about/news/postgresql-183-179-1613-1517-and-1422-released-3246/ (date of access: 18.05.2026).

23) Query Tool – pgAdmin 4 9.14 documentation. *PostgreSQL Tools*. URL: https://www.pgadmin.org/docs/pgadmin4/9.14/query_tool.html (date of access: 18.05.2026).

24) PostgreSQL Tools. *pgAdmin*. URL: https://www.pgadmin.org/ (date of access: 18.05.2026).

25) FastAPI Documentation. *FastAPI*. URL: https://fastapi.tiangolo.com/ (date of access: 18.05.2026).

26) What is introspection? (Reference). *Prisma | Serverless Postgres, Type-Safe ORM, and Database Tools*. URL: https://www.prisma.io/docs/orm/prisma-schema/introspection (date of access: 18.05.2026).

27) Prisma db pull | introspect database to prisma schema. *Prisma | Serverless Postgres, Type-Safe ORM, and Database Tools*. URL: https://www.prisma.io/docs/cli/db/pull (date of access: 18.05.2026).

28) Introducing claude sonnet 4.5. *Home \ Anthropic*. URL: https://www.anthropic.com/news/claude-sonnet-4-5 (date of access: 18.05.2026).

29) Models overview. *Claude API Docs*. URL: https://platform.claude.com/docs/en/about-claude/models/overview (date of access: 18.05.2026).

30) Llama 4 | model cards and prompt formats. *Industry Leading, Open-Source AI | Llama*. URL: https://www.llama.com/docs/model-cards-and-prompt-formats/llama4/ (date of access: 18.05.2026).

31) The Llama 4 herd: The beginning of a new era of natively multimodal AI innovation. *AI at Meta: Meta AI Products, Models and Research | AI at Meta*. URL: https://ai.meta.com/blog/llama-4-multimodal-intelligence/ (date of access: 18.05.2026).

32) Meta-llama/Llama-4-Scout-17B-16E-Instruct · hugging face. *Hugging Face – The AI community building the future.* URL: https://huggingface.co/meta-llama/Llama-4-Scout-17B-16E-Instruct (date of access: 18.05.2026).

33) Structured Outputs - GroqDocs. *GroqDocs*. URL: https://console.groq.com/docs/structured-outputs (date of access: 18.05.2026).

34) API Reference - GroqDocs. *GroqCloud*. URL: https://console.groq.com/docs/api-reference (date of access: 18.05.2026).

35) Structured model outputs | OpenAI API. *OpenAI Developers*. URL: https://developers.openai.com/api/docs/guides/structured-outputs (date of access: 18.05.2026).