Орієнтовний, першочергова чернетка. Не відповідає дійсності

1. Вступ
   1. Визначення проблеми t2sql
   2. Лінгвістична складність
   3. Складність розуміння баз даних
   4. Sql синтаксис
   5. Історія розвитку text-to-sql
   6. Існуючі набори даних
   7. Проблеми української мови
2. Теоретична частина

2.1 Опис роботи і фреймворку mac-sql   
2.2 Обгрунтування вибору великої мовної моделі та спосіб доступу  
2.3 Обгрунтування вибору PostgreSQL  
2.4 Обгрунтування способу тестування роботи фреймворку  
2.5 спосіб тестування фреймворку на українській мові

1. Практична частина

3.1 Постановка задачі  
3.2 Розробка MAC-SQL з адаптацією під together api та postgresql  
3.2.1 Розробка копії(чи прототипу) mac-sql  
3.2.2 Реалізація виклику API together.ai   
3.2.3 Реалізація створення схеми бази даних з якою буде працювати фреймворк

3.3 Розробка українського датасету «BIRD-UKR»  
3.3.1 Генерація штучних даних   
3.3.2 Створення пар питання – sql-запит  
3.3.3 Написання скрипта для тестування

4. Результати

4.1 Результати base model  
4.2 Результати фреймворку на BIRD&Spider   
4.3 Висновки про здатність фреймворку та моделі працювати з українською мовою та базами даних

5. Висновок

Додатки

1. **Вступ**

Text-to-SQL, також відома як Natural Language to SQL (NL2SQL), є ключовою технологією у сфері управління даними та їх доступності. В основі цієї технології лежить здатність перекладати запити, виражені природною людською мовою, в команди мови структурованих запитів (SQL), які можуть бути виконані реляційною базою даних. Цей процес перекладу дозволяє користувачам взаємодіяти з базами даних, використовуючи свою повсякденну мову, фактично усуваючи необхідність у глибоких знаннях синтаксису SQL. Першочерговою метою систем Text-to-SQL є *демократизація доступу* до багатства інформації, що міститься в реляційних базах даних, тим самим подолавши розрив між людським спілкуванням і структурованою мовою запитів до баз даних. Ця галузь також визнана в академічному та промисловому контекстах під назвою Natural Language Interface to Databases (NLIDB).

У більш широкому контексті обробки природної мови (NLP) і взаємодії з базами даних, Text-to-SQL займає життєво важливе місце. По суті, це завдання належить до області NLP і вважається специфічною формою семантичного аналізу. Семантичний аналіз включає в себе процес перетворення вхідних даних на природній мові в структуровану, зрозумілу машині логічну форму, причому цільовою логічною формою у випадку Text-to-SQL є SQL. Ця галузь зазнала значної еволюції, значною мірою завдяки значним досягненням у галузі штучного інтелекту, особливо у сферах генеративного штучного інтелекту та великих мовних моделей (LLM). Ці досягнення призвели до помітного підвищення точності та зручності систем Text-to-SQL з плином часу.

Основною проблемою, яку ефективно вирішує Text-to-SQL, є бар'єр, пов'язаний з необхідністю знання SQL для осіб, які не володіють технічними знаннями щодо запитів до баз даних. Пропонуючи інтерфейс на природній мові, вона досягає значної демократизації доступу до даних.

* 1. Визначення Text2SQL(NL2SQL)

Процес перекладу запитів на природній мові в команди SQL зазвичай включає послідовність чітко визначених кроків. Він починається з введення даних користувачем, коли людина формулює запит на своїй природній мові. Після цього до вхідних даних застосовуються методи обробки природної мови (NLP), щоб визначити наміри користувача і контекст запиту. Потім оброблені дані проходять етап розуміння запиту, якому часто сприяють передові моделі машинного навчання, зокрема великі мовні моделі (LLM), які аналізують оброблені компоненти, щоб інтерпретувати основний запит користувача. Кульмінацією цих етапів є генерація SQL, де система створює синтаксично правильний SQL-запит, призначений для точного відображення даних, які користувач хоче отримати.

Ефективність систем Text-to-SQL ґрунтується на кількох ключових компонентах і технологічних основах. Обробка природної мови (Natural Language Processing, NLP) слугує початковим важливим етапом, на якому вона використовується для аналізу природної мови, введеної користувачем, визначення релевантних ключових слів і, зрештою, визначення конкретного наміру, що стоїть за запитом. Великі мовні моделі (LLM), особливо ті, що базуються на архітектурах трансформаторів, таких як BERT і GPT, відіграють ключову роль у розумінні контексту і нюансів, притаманних природній мові, а також у створенні відповідних SQL-запитів. Процес семантичного аналізу є центральним для Text-to-SQL, що включає в себе перетворення запиту на природній мові в машинно-інтерпретоване представлення, яке потім зіставляється з відповідним SQL-запитом. Відображення схеми - це ще один важливий крок, на якому модель встановлює зв'язки між словами і фразами, що використовуються в запиті, і конкретними таблицями, стовпцями бази даних та їх взаємозв'язками. Нарешті, часто застосовуються методи оптимізації запитів, щоб переконатися, що згенерований SQL-запит є не тільки коректним, але й ефективним у виконанні.

По суті, завдання Text-to-SQL - це завдання семантичного розбору, основною метою якого є перетворення запиту користувача на природній мові в логічну форму, яку може зрозуміти і виконати машина, якою в даному контексті є мова SQL. Цей процес вимагає не лише розуміння буквального значення слів, використаних у запиті природною мовою, але й розуміння намірів користувача і того, як ці наміри відображаються на конкретних елементах і операціях, доступних у схемі бази даних.

* 1. Лінгвістична складність

Незважаючи на значний прогрес у технології Text-to-SQL, кілька складних викликів і постійних проблем продовжують перешкоджати її широкому і бездоганному застосуванню. Однією з найголовніших перешкод є неоднозначність, притаманна природній мові. Користувачі часто формулюють свої запити, використовуючи нечіткі терміни, розмовну лексику або вузькоспецифічний жаргон, пов'язаний з конкретною галуззю, який не має прямого або однозначного відповідника в структурованій схемі бази даних. Ця неоднозначність проявляється в різних формах, включаючи лексичну неоднозначність, коли одне слово може мати кілька різних значень залежно від контексту; синтаксичну неоднозначність, коли граматична структура речення допускає більш ніж одну інтерпретацію; і недостатню конкретизацію, коли лінгвістичному виразу не вистачає деталей, щоб чітко передати конкретні наміри користувача. Крім того, складності природної мови поширюються на референційну неоднозначність, коли може бути незрозуміло, до чого відноситься займенник або займенникова фраза, і неоднозначність еліпсису, коли частини речення маються на увазі, але не вказані явно. У контексті запитів до бази даних неоднозначність запиту може виникнути, коли один запит природною мовою може бути інтерпретований кількома семантично різними способами, виходячи зі структури та змісту однієї таблиці бази даних. На додаток до цих лінгвістичних проблем, неоднозначність також може виникати через перекриття імен схем в базі даних або через існування декількох заплутаних шляхів зв'язків між різними об'єктами бази даних.

Значний аспект складності полягає в ефективній обробці лінгвістичних варіацій. Користувачі можуть виражати одну й ту саму інформаційну потребу або мету запиту різними способами, використовуючи різні синоніми, перефразування та граматичні структури. Надійна система Text-to-SQL повинна бути здатна розпізнавати і правильно інтерпретувати ці різноманітні лінгвістичні форми. Крім того, в ідеалі система повинна бути стійкою до недосконалостей у введенні даних користувачем, таких як типові орфографічні та граматичні помилки, які можуть значно ускладнити завдання розуміння природної мови.

* 1. Складність схем баз даних

Ще один значний рівень труднощів виникає через складність і різноманітність схем баз даних, що зустрічаються в реальних додатках. Зазвичай виробничі бази даних містять сотні, якщо не тисячі, таблиць, кожна з яких має численні стовпці, а зв'язки між цими таблицями можуть бути надзвичайно складними. Цей величезний обсяг і складність ускладнюють для систем Text-to-SQL ефективне включення всієї релевантної інформації про схему в єдиний контекст обробки, особливо з огляду на обмеження розмірів вхідних даних моделі. Проблема ще більше ускладнюється тим, що бази даних у різних галузях часто дотримуються дуже різних угод про імена, формати даних і загальні структури таблиць. Це включає використання неінтуїтивно зрозумілих назв стовпців, переважання абревіатур, які можуть бути незрозумілими для всіх, і загальну неоднозначність імен, що вимагає від моделі Text-to-SQL потужних міркувань для правильного виведення зв'язків між таблицями і стовпцями. Крім того, реальні бази даних не є статичними; їхні схеми можуть бути динамічними і розвиватися, що вимагає від систем Text-to-SQL механізмів безперервної адаптації до цих змін.

Іншим важливим аспектом складності є розуміння семантики бази даних. Це виходить за рамки простого знання назв таблиць і стовпців; модель повинна володіти глибоким розумінням всієї схеми бази даних, включаючи типи даних стовпців, зв'язки, визначені первинними і зовнішніми ключами, і загальну логічну структуру, яка з'єднує різні таблиці. Більше того, в ідеалі система повинна мати певний рівень розуміння значення даних, що містяться в цих таблицях, які часто можуть бути вузькоспецифічними і вимагати контекстних знань, що виходять за рамки самої схеми бази даних.

Критично важливим аспектом перетворення Text-to-SQL є зв'язування та вирівнювання схем, що передбачає точне відображення сутностей, згаданих у природномовному запиті, у відповідні таблиці, стовпці та SQL-операції в схемі бази даних. Це основне завдання, яке лежить в основі всього процесу перекладу. Найпоширенішою проблемою є невідповідність між описами природною мовою та реальними назвами стовпців у схемі SQL, яку часто називають «проблемою невідповідності». Навіть з розвитком моделей великих мов, ці моделі все ще не можуть точно узгодити запит користувача з правильними елементами схеми, що іноді призводить до вибору надмірної кількості стовпців або таблиць у спробі охопити всі аспекти вхідних даних.

* 1. SQL syntax

Складність SQL-запитів, які користувачі можуть створювати на основі вхідних даних природною мовою, також становить значну проблему. Хоча прості запити з базовими критеріями відбору часто обробляються з достатньою точністю, створення більш складних SQL-запитів, які включають об'єднання декількох таблиць, вкладені підзапити і складну умовну фільтрацію, залишається складним завданням для багатьох систем. Для багатотабличних запитів система повинна бути здатна точно визначати зв'язки між таблицями, наприклад, за допомогою обмежень зовнішнього ключа, щоб створити відповідні речення JOIN. Крім того, певні домени або специфічні аналітичні вимоги можуть вимагати використання спеціалізованих функцій або операцій SQL, що додає ще один рівень складності до процесу генерації SQL. Навіть такі, здавалося б, прості аспекти SQL, як порядок предикатів у реченні WHERE, що не впливає на результат запиту, можуть стати проблемою для систем Text-to-SQL, які намагаються зіставити природну мову з певним синтаксисом SQL.

Основним завданням перекладу з текстового на SQL є зіставлення природної мови з конкретними конструкціями SQL. Це передбачає правильне визначення ключових слів і речень SQL (таких як SELECT, FROM, WHERE, GROUP BY), які маються на увазі в запиті природною мовою. Крім того, природна мова часто містить порівняльні та вищі ступені порівняння (наприклад, «менше», «найвищий»), які необхідно перевести у відповідні оператори порівняння SQL (<, >, =) та агрегатні функції (SUM, AVG, MAX, MIN, COUNT). Складність значно зростає при роботі з вкладеними запитами (запит всередині іншого запиту) та операціями над множинами (такими як UNION, INTERSECT, EXCEPT), які вимагають більш глибокого розуміння логічної структури запиту.

* 1. Проблема навчання

Здатність систем Text-to-SQL добре працювати в різних сценаріях, відома як узагальнення, а також їхня ефективність при застосуванні до баз даних або доменів, для яких вони не були спеціально підготовлені (так звані міждоменні завдання), залишаються серйозними проблемами. Моделі, які досягають високої точності на конкретних навчальних наборах даних, часто не можуть підтримувати цю продуктивність, коли стикаються з новими, небаченими базами даних або варіаціями того, як користувачі можуть формулювати свої запити. Варіації мовних виразів, такі як використання абревіатур або синонімів, які не зустрічалися під час навчання, можуть призвести до помітного зниження продуктивності системи. Завдання стає особливо складним, коли ми маємо справу з міждоменними сценаріями, які вимагають від системи роботи з різноманітними темами та структурами баз даних, прикладом чого є еталонні набори даних на кшталт Spider.

Якість і доступність даних, що використовуються для навчання моделей Text-to-SQL, також відіграють вирішальну роль у їхній загальній ефективності. Відсутність великих, високоякісних навчальних наборів даних, особливо для конкретних доменів або складних типів запитів, може суттєво знизити продуктивність цих систем. Крім того, існуючі еталонні набори даних можуть мати внутрішні обмеження з точки зору різноманітності SQL-запитів, які вони містять, діапазону операторів SQL, які вони використовують, і того, наскільки реалістично вони відображають складність реальних баз даних. Навіть у межах однієї бази даних невідповідності у форматуванні даних можуть створювати додаткові проблеми для систем Text-to-SQL, які намагаються інтерпретувати запити користувачів.

* 1. Історія розвитку

Розвиток технології Text-to-SQL охоплює кілька десятиліть, позначених значними змінами в підходах і можливостях, що відображають більш широкі досягнення в галузі штучного інтелекту та обробки природної мови. Перші спроби подолати розрив між людською мовою і взаємодією з базами даних почалися з ранніх систем, заснованих на правилах, в епоху до глибокого навчання. Ці системи, що з'явилися в 1960-х і 1970-х роках, включаючи такі відомі приклади, як BASEBALL, LUNAR, LIFER/LADDER і CHAT-80, покладалися на ретельно розроблені граматичні правила та евристики для перекладу запитів на природній мові у виконувані команди SQL. Хоча ці системи продемонстрували початкову перспективність, особливо в контексті простих і вузькоспецифічних баз даних, вони зіткнулися зі значними обмеженнями, коли зіткнулися з більш складними запитами або базами даних, що характеризуються різноманітними схемами, які постійно розвиваються. Це було значною мірою пов'язано з притаманною негнучкістю правил, визначених вручну, і значними зусиллями, необхідними для функціональної інженерії, щоб адаптувати ці системи до нових доменів. Такі системи, як LUNAR і NaLIX, демонструючи потенціал семантичного синтаксичного аналізу, страждали від недостатньої масштабованості та адаптивності до ширших застосувань.

У міру того, як бази даних зростали як у розмірі, так і в складності, галузь стала свідком переходу до статистичних методів і методів машинного навчання. Ці підходи використовували алгоритми, які могли вивчати основні відповідності між вхідними даними на природній мові та відповідними їм SQL-запитами з даних. Порівняно зі своїми попередниками, що базувалися на правилах, ці системи, засновані на машинному навчанні, демонстрували здатність обробляти ширший спектр більш складних запитів. Критично важливим компонентом цих методів був інжиніринг ознак, який передбачав ручне вилучення релевантних ознак як з вхідного тексту природною мовою, так і зі схеми бази даних для навчання алгоритмів навчання.

Основним поворотним моментом в історії Text-to-SQL стала поява глибокого навчання. Впровадження рекурентних нейронних мереж (RNN), зокрема мереж з довгою короткочасною пам'яттю (LSTM), ознаменувало значний стрибок вперед у можливостях цієї галузі. Моделі Sequence-to-sequence (Seq2Seq), що часто використовують архітектуру кодера-декодера, побудовану на основі LSTM, стали популярною основою для безпосереднього перекладу запитів природною мовою в SQL-послідовності. Для подальшого підвищення продуктивності цих моделей були інтегровані механізми уваги, що дозволяють декодеру зосереджуватися на найбільш релевантних частинах вхідної послідовності при генерації кожної частини вихідного SQL-запиту.

Потім ця галузь перейшла в еру Трансформер-моделей та Попередньо навчених Мовних Моделей (ПНММ). Архітектури на основі Трансформерів, такі як BERT, GPT та T5, з того часу стали домінуючим підходом у дослідженнях та застосунках Text-to-SQL. Їхній успіх значною мірою зумовлений винятковою здатністю розуміти контекстуальні зв'язки як у запиті природною мовою, так і в схемі бази даних за допомогою механізмів самоуваги. Ключовою технікою цього періоду стало попереднє навчання цих великих мовних моделей на величезних обсягах загальних текстових даних, з подальшим їх доналаштуванням (fine-tuning) на специфічних наборах даних Text-to-SQL. Ця парадигма значно підвищила продуктивність систем Text-to-SQL. Такі моделі, як RAT-SQL та TaBERT, були спеціально розроблені для ефективної інтеграції інформації про схему бази даних у процес навчання моделі, що призвело до підвищення точності.

Графові Нейронні Мережі (ГНМ) також все частіше застосовуються в Text-to-SQL для ефективного кодування структури та зв'язків у схемі бази даних. Представляючи схему у вигляді графа, ГНМ можуть формувати насичені представлення, що покращують розуміння моделлю контексту бази даних. RAT-SQL – це помітна модель, яка використовує трансформерну мережу, що враховує зв'язки (relation-aware), для кодування схеми та зв'язування, демонструючи ефективність цього підходу.

Останнім часом у цій галузі спостерігається швидка поява та впровадження Великих Мовних Моделей (LLM), таких як GPT-4 та Gemini, які здійснили подальшу революцію у можливостях систем Text-to-SQL. Ці моделі, завдяки своїй величезній здатності до розуміння та генерації людиноподібного тексту, продемонстрували вміння обробляти все складніші SQL-запити, сформовані на основі природномовних запитів. Поточні дослідження в цій галузі часто досліджують використання навчання в контексті (in-context learning), де ВММ надають кілька прикладів для скеровування генерації SQL без необхідності явного доналаштування ваг моделі, а також традиційні підходи до доналаштування, за яких ВММ навчаються на специфічних наборах даних Text-to-SQL для адаптації до завдання.

Історичний розвиток Text-to-SQL демонструє чітку тенденцію переходу від систем, що базувалися на чітко визначених правилах, які вимагали значних ручних зусиль і мали обмежену сферу застосування, до все більш досконалих моделей глибокого навчання, здатних вивчати складні закономірності безпосередньо з даних. Останні досягнення, зумовлені Трансформер-моделями та, особливо, великими мовними моделями, ознаменували значний зсув парадигми в цій галузі. Ці сучасні підходи дозволили значно покращити точність та здатність систем Text-to-SQL ефективно обробляти складніші та міждоменні сценарії, наближаючи втілення ідеї справді природномовної взаємодії з базами даних.

* 1. Набори даних

На розробку та оцінювання систем Text-to-SQL значно вплинула наявність еталонних наборів даних (бенчмарків) та метрик, що використовуються для оцінки продуктивності. Ранні дослідження часто спиралися на однодоменні набори даних, такі як ATIS (для інформації про авіаперельоти) та GeoQuery (для географічних запитів), які забезпечували сфокусований контекст для оцінки початкових підходів.

Значним поворотним моментом у цій галузі стала поява міждоменних бенчмарків, метою яких було оцінити здатність моделей до узагальнення на різних схемах баз даних та типах запитів. WikiSQL був одним із перших великомасштабних наборів даних такого типу, що містив відносно прості SQL-запити до великої кількості таблиць, взятих з Вікіпедії. Після WikiSQL було представлено Spider — складніший та більш вимогливий міждоменний набір даних, що містить різноманітний набір SQL-запитів, включно із запитами зі з'єднаннями (joins) та вкладеними структурами, для колекції баз даних з різних доменів. Для завдань, що включають діалогову взаємодію з базами даних, були створені набори даних, такі як SParC та CoSQL, для оцінки здатності моделей обробляти багатоходові діалоги.

На додаток до цих базових наборів даних, були розроблені різноманітні спеціалізовані та доповнені набори даних для вирішення специфічних проблем або оцінки окремих аспектів систем Text-to-SQL. Прикладами є KaggleDBQA, SEDE та EHRSQL, які мають на меті представити більш реалістичні сценарії, а також набори даних, такі як Spider-DK, Spider-Syn та Spider-Realistic, які вносять варіації в оригінальний набір даних Spider для перевірки стійкості моделі до знань предметної області, використання синонімів та відсутності явних назв стовпців у запитаннях. Крім того, докладалися зусилля для створення доповнених наборів даних, таких як GRAPPA, GAP2SQL та Spider-CG, які генерують синтетичні навчальні приклади для збільшення розміру та різноманітності навчальних даних і потенційного покращення продуктивності моделей Text-to-SQL.

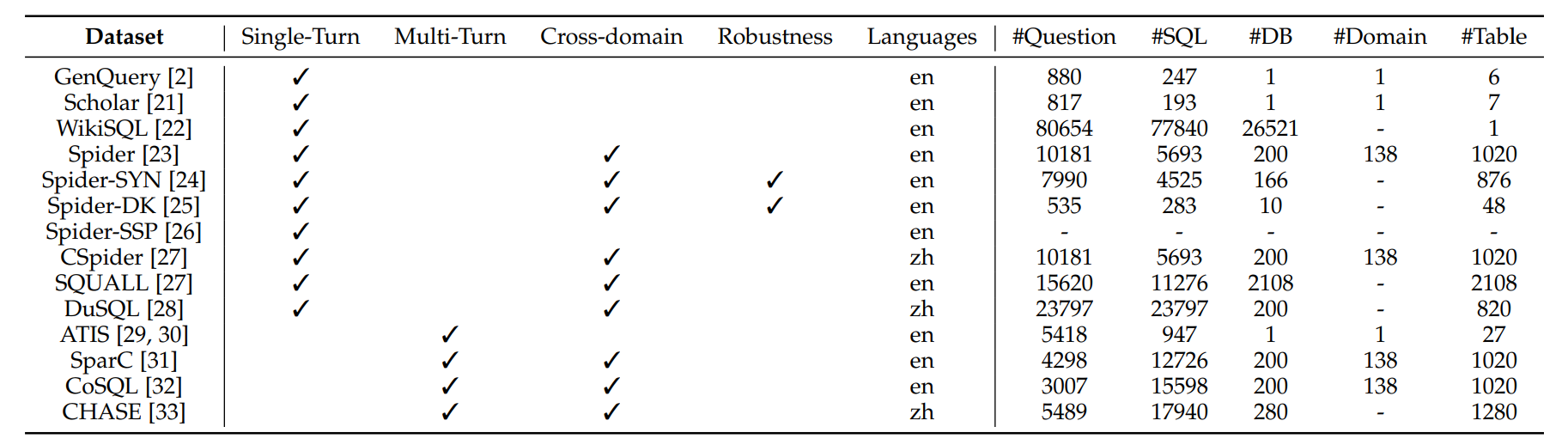
Продуктивність систем Text-to-SQL зазвичай оцінюється за допомогою кількох ключових метрик. Точність виконання (Execution Accuracy, EX) – це критично важлива метрика, яка вимірює, чи згенерований SQL-запит при виконанні в базі даних повертає правильну відповідь. Точна відповідність (Exact Match, EM) – ще одна поширена метрика, яка перевіряє, чи згенерований SQL-запит точно збігається (посимвольно) з еталонним (анотованим людиною) SQL-запитом. Точність логічної форми (Logical Form Accuracy, LF) має на меті оцінити семантичну еквівалентність згенерованого SQL, часто шляхом порівняння їхніх абстрактних синтаксичних дерев, що може бути більш поблажливим до синтаксичних варіацій, які не змінюють значення запиту.

Незважаючи на прогрес, досягнутий завдяки цим наборам даних та метрикам, зростає усвідомлення їхніх обмежень. Існуючі бенчмарки можуть не повністю охоплювати різноманітність SQL-запитів, що зустрічаються в реальних застосунках, повний спектр операторів SQL, складність реальних схем баз даних або нюанси природномовних запитань, поставлених користувачами. Метрики оцінки, такі як точна відповідність, можуть бути надто суворими, штрафуючи згенеровані запити, які семантично еквівалентні еталонним, але відрізняються синтаксично. Точність виконання, хоч і важлива, іноді може призводити до хибнопозитивних спрацювань, коли неправильний SQL-запит може випадково повернути правильний результат для певного стану бази даних. Отже, існує постійна потреба в розробці бенчмарків та метрик оцінки, які б точніше відображали виклики реальних завдань Text-to-SQL та справжні наміри користувацьких запитів.

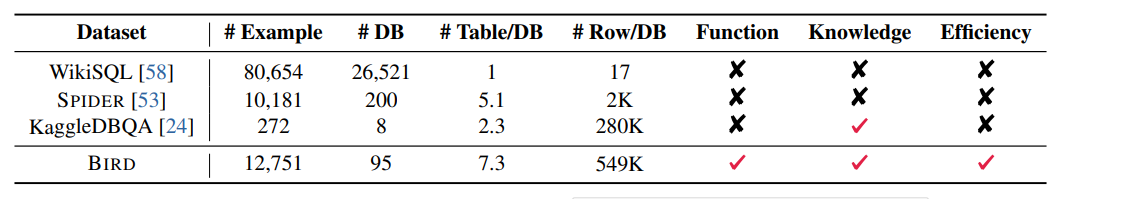
Бенчмарк BIRD (BIg Bench for LaRge-Scale Database Grounded in Text-to-SQLs) був створений як великий еталон для оцінки Text-to-SQL у реальних застосунках. Згідно з описом, BIRD містить 12 751 складний приклад запитів до інформації з 95 великих баз даних загальним розміром 33,4 ГБ, що охоплюють 37 професійних доменів. Для навчання та розробки його творці зібрали та модифікували 80 реляційних баз даних з відкритим кодом з реальних аналітичних платформ (Kaggle, Relation.vit). Щоб додатково уникнути витоку даних, було підготовлено 15 додаткових реляційних баз даних для прихованого тестового набору. На основі цих баз даних для збору питань природною мовою та відповідних SQL-запитів використовувався краудсорсинг. Додатково, автори бенчмарку запропонували нову метрику оцінки — Valid Efficiency Score (VES) — для оцінки ефективності згенерованих SQL-запитів. За їхніми твердженнями, BIRD є першим бенчмарком Text-to-SQL, який враховує ефективність, сприяючи використанню більш ефективних методів запитів у контексті великих обсягів даних та "зашумлених" значень у базах даних.

Розвиток наборів даних у галузі Text-to-SQL відіграв ключову роль у стимулюванні прогресу можливостей моделей. Перехід від однодоменних до все складніших міждоменних бенчмарків спонукав дослідників розробляти моделі, здатні обробляти більш реалістичні та складні сценарії. Однак поточна дискусія щодо обмежень існуючих бенчмарків та метрик оцінки підкреслює необхідність постійного вдосконалення способів оцінки продуктивності та реальної застосовності систем Text-to-SQL.

Таблиця 1 – Приклади датасетів



[1 <https://arxiv.org/pdf/2208.13629>]



[2 https://arxiv.org/pdf/2305.03111]

* 1. Стан на сьогодні

Сильні сторони цих сучасних передових моделей, особливо тих, що базуються на LLM(ВММ?), включають їхню надійну здатність до перенесення знань (transferability), що дозволяє їм легше адаптуватися до нових завдань та доменів. Вони також демонструють покращені здібності до логічного мислення, що дає їм змогу обробляти складніші SQL-завдання, які виходять за межі обмежень попередньо визначених фреймворків. Техніки, такі як промптинг "Ланцюжок Думок" (Chain-of-Thought, COT), де модель явно генерує проміжні кроки міркувань, та декомпозиція завдань, де складні запити розбиваються на менші, легші для керування частини, сприяли покращенню логіки та точності згенерованого SQL.

Незважаючи на ці помітні досягнення, сучасні моделі Text-to-SQL все ще стикаються з низкою властивих їм обмежень та викликів, коли йдеться про реальні застосунки. Однією з постійних проблем є обізнаність щодо схеми (schema awareness). Великі мовні моделі потребують глибокого розуміння відповідної схеми бази даних, щоб генерувати точні SQL-запити.

Навіть із передовими моделями згенерований SQL іноді може бути неточним через такі проблеми, як галюцинації ШІ (генерація неправильного або безглуздого SQL), використання помилкових назв стовпців, неправильне розуміння схеми бази даних або погано сформульовані промпти у випадку ВММ.

Крім того, здатність сучасних моделей до узагальнення на небачених раніше даних, особливо нових баз даних та доменів, які не були включені в їхнє навчання, все ще є важливою сферою для вдосконалення.(це проблема будь-яких моделей машинного навчання?)

Додатково, інтеграція систем Text-to-SQL у реальні середовища викликає занепокоєння щодо вразливостей безпеки, таких як потенційна можливість атак SQL-ін'єкцій, якщо система не розроблена ретельно з урахуванням безпеки.

Подолання цих обмежень, особливо в таких сферах, як розуміння складних схем баз даних, ефективна обробка властивої природній мові неоднозначності, покращення узагальнення на нові та небачені дані, а також забезпечення безпеки та ефективності цих систем, буде вирішальним для ширшого впровадження та впливу технології Text-to-SQL.

2.1 MAC-SQL[3 https://arxiv.org/abs/2312.11242] вибір

Фреймворк MAC-SQL був розроблений для подолання цих обмежень шляхом застосування мультиагентного підходу (Xu et al., стор. 1, 3). Він складається з трьох спеціалізованих агентів, які взаємодіють для генерації та уточнення SQL запитів (Xu et al., стор. 2, Рис. 2):

1. **Агент Selector:** Призначений для роботи з великими схемами баз даних. Він аналізує запит користувача та повну схему для ідентифікації та відбору лише релевантних таблиць і стовпців. Це дозволяє зменшити вхідний контекст для LLM, знижуючи ймовірність помилок, пов'язаних із зайвою інформацією (Xu et al., стор. 3, розділ 3.2).
2. **Агент Decomposer:** Використовує підхід "ланцюжка думок" (chain-of-thought) для декомпозиції складного запиту користувача на послідовність простіших підзапитів. Для кожного підзапиту генерується відповідний SQL. Цей підхід полегшує моделі обробку запитів, що вимагають складних логічних міркувань (Xu et al., стор. 3, розділ 3.3; стор. 4, Рис. 3).
3. **Агент Refiner:** Застосовує зовнішні інструменти для перевірки синтаксичної коректності та виконуваності згенерованого SQL запиту. У разі виявлення помилки або отримання неочікуваного результату (наприклад, порожній набір даних), Refiner використовує отриманий зворотний зв'язок для корекції SQL запиту (Xu et al., стор. 4, розділ 3.4; стор. 4, Рис. 4).

**Емпірична Ефективність**

Ефективність архітектури MAC-SQL підтверджується результатами експериментів, представленими в роботі Xu et al. Комбінація MAC-SQL з моделлю GPT-4 досягла точності виконання (Execution Accuracy) 59.59% на тестовому наборі бенчмарку BIRD, що на момент публікації дослідження було передовим результатом (state-of-the-art) (Xu et al., стор. 1, 6). Це свідчить про переваги запропонованого мультиагентного підходу для вирішення складних завдань Text-to-SQL.

Важливим фактором при виборі MAC-SQL є його доступність для дослідницької спільноти та розробників.

Надання доступу до коду фреймворку через репозиторій GitHub (Xu et al., стор. 1, виноска 1) значно спрощує процес відтворення результатів, адаптації та інтеграції фреймворку в інші проекти. Це усуває залежність від пропрієтарних моделей та сприяє подальшому розвитку напрямку Text-to-SQL.

2.2 Base model(?)

Вибір Базової Мовної Моделі: Llama 3.1 70B

В оригінальному дослідженні фреймворку MAC-SQL (Xu et al.) як основна велика мовна модель (LLM), що забезпечувала функціонування агентів Selector, Decomposer та Refiner, використовувалася модель GPT-4 від OpenAI. У даній роботі, як базова LLM для всіх агентів фреймворку MAC-SQL, була обрана модель **Llama 3.1 70B**. Цей вибір обумовлений кількома ключовими перевагами порівняно з використанням GPT-4:

1. Відкритий Доступ та Гнучкість**:** Llama 3.1 70B є моделлю з відкритим доступом (з відповідною ліцензією), на відміну від GPT-4, яка доступна лише через пропрієтарний API. Це надає значно більший контроль над моделлю та процесом її використання:
   * Глибоке донавчання (Fine-tuning): Можливість донавчати модель безпосередньо на специфічних даних та завданнях агентів MAC-SQL (спрощення схеми, декомпозиція, генерація/корекція SQL) дозволяє точніше адаптувати її поведінку до конкретних вимог задачі Text-to-SQL, потенційно покращуючи точність та релевантність результатів.
   * Прозорість та Відтворюваність: Використання конкретної версії моделі з відкритими вагами підвищує прозорість дослідницького процесу та спрощує відтворення результатів іншими дослідниками, оскільки немає залежності від можливих змін у закритій API-моделі.
   * Гнучкість Розгортання: Відкриті моделі можуть бути розгорнуті локально або у приватній хмарі, що забезпечує повний контроль над інфраструктурою та конфіденційністю даних, що є важливим при роботі з чутливими базами даних.
2. Економічна Ефективність: Використання GPT-4 через API пов'язане зі значними операційними витратами, особливо при обробці великих наборів даних або при ітеративному процесі розробки та тестування, що також було відзначено як обмеження у оригінальному дослідженні (Xu et al., стор. 7). Локальне розгортання та використання Llama 3.1 70B усуває ці витрати, пов'язані з API-викликами, роблячи процес розробки та експлуатації значно економічно ефективнішим.
3. Висока Продуктивність: Llama 3.1 70B належить до останнього покоління потужних відкритих LLM, демонструючи високу продуктивність у широкому спектрі завдань, включаючи розуміння природної мови, генерацію коду та логічні міркування, які є критичними для Text-to-SQL. Хоча GPT-4 показала передові результати в оригінальній роботі MAC-SQL, сучасні відкриті моделі, такі як Llama 3.1 70B, мають потенціал досягти порівнянної або навіть вищої продуктивності, особливо після цільового донавчання, при цьому зберігаючи переваги відкритості та контролю.

2.3 Вибір Цільової Бази Даних для Тестування: PostgreSQL

Вибір середовища для виконання та тестування згенерованих SQL запитів є важливим аспектом оцінки систем Text-to-SQL. Хоча SQLite часто використовується в академічних бенчмарках (наприклад, для перевірки виконання в деяких частинах Spider або BIRD) завдяки своїй простоті інтеграції та відсутності необхідності в окремому сервері, у даній роботі для валідації та оцінки точності виконання (Execution Accuracy) було обрано реляційну систему керування базами даних (РСКБД) PostgreSQL.

Цей вибір обумовлений наступними міркуваннями щодо обмежень SQLite та переваг PostgreSQL для цілей ретельного тестування:

1. Обмеженість Функціональності SQLite:
   * SQLite, будучи вбудованою, файловою базою даних, має спрощений діалект SQL порівняно з повнофункціональними серверними РСКБД, такими як PostgreSQL, MySQL, Oracle або SQL Server, які зазвичай використовуються в корпоративних (enterprise) середовищах.
   * SQLite може не підтримувати або мати іншу реалізацію розширених функцій SQL, таких як віконні функції, Common Table Expressions (CTE), складні типи даних, розширені оператори JOIN, специфічні функції для роботи з датами/часом, текстом чи JSON, а також складні обмеження (constraints). Це означає, що SQL запит, який коректно виконується в SQLite, може бути невалідним або повертати інші результати в більш потужній РСКБД.
   * SQLite часто є менш суворим щодо типізації даних та синтаксису запитів порівняно з PostgreSQL.
2. Переваги PostgreSQL для Реалістичного Тестування:
   * Репрезентативність для підприємницьких Сценаріїв: PostgreSQL є однією з найпоширеніших РСКБД з відкритим кодом, що широко використовується в промислових та корпоративних системах. Тестування на PostgreSQL забезпечує більш реалістичну оцінку здатності системи генерувати SQL, придатний для використання в типових робочих середовищах.
   * Багатший та Строгіший Діалект SQL: PostgreSQL підтримує широкий спектр стандартних та розширених функцій SQL. Вимога генерації коректного SQL для PostgreSQL ставить перед системою Text-to-SQL складніше завдання, змушуючи її враховувати більшу кількість синтаксичних та семантичних нюансів, що веде до більш надійної оцінки її можливостей.
   * Ретельна Валідація: Успішне виконання складного запиту в PostgreSQL, з його строгим дотриманням стандартів та підтримкою розширених можливостей, надає вищий рівень впевненості у коректності та загальній придатності згенерованого SQL порівняно з тестуванням на більш обмеженій платформі як SQLite.

2.4 Вибір Тестових Наборів Даних: BIRD (minidev) та Spider

Для всебічної оцінки ефективності фреймворку MAC-SQL на основі Llama 3.1 70B та перевірки його здатності генерувати коректні SQL запити для PostgreSQL, було обрано два ключових бенчмарки: Spider та піднабір BIRD (minidev). Вибір цих наборів даних дозволяє оцінити різні аспекти продуктивності системи Text-to-SQL.

1. Spider(***не дуже велике бажання включати***):
   * Призначення: Spider (Yu et al., 2018) є широко визнаним стандартним бенчмарком для оцінки можливостей систем Text-to-SQL щодо узагальнення на різних доменах (cross-domain generalization). Він містить велику кількість баз даних (200) з різних предметних областей, але самі бази даних, як правило, є відносно невеликими за розміром та обсягом даних.
   * Основні Виклики: Головний акцент Spider робиться на складності структури запитів та необхідності коректно інтерпретувати схему бази даних (назви таблиць, стовпців, зв'язки між ними) для відповіді на запитання користувача. Він перевіряє здатність моделі генерувати складні SQL конструкції (вкладені запити, агрегації, JOINи по багатьох таблицях) незалежно від специфіки домену.
   * Обґрунтування Вибору: Включення Spider до процесу тестування є необхідним для порівняння продуктивності розробленого рішення з попередніми роботами та для оцінки його фундаментальної здатності розуміти різноманітні структури баз даних та перекладати запити на SQL без глибокого аналізу *змісту* даних. Це дозволяє оцінити ефективність, зокрема, агента Decomposer у роботі зі складними запитами.
2. BIRD (minidev Subset):
   * Призначення: Бенчмарк BIRD (Li et al., 2023) був створений для подолання обмежень Spider та моделювання реалістичних сценаріїв взаємодії з великими базами даних. Він фокусується на викликах, пов'язаних із значними обсягами даних, "брудними" даними (dirty values), потребою в зовнішніх знаннях та ефективністю SQL запитів.
   * Основні Виклики: BIRD вимагає від систем Text-to-SQL не лише розуміння схеми, але й здатності міркувати над вмістом бази даних (database content reasoning), обробляти великі схеми (десятки таблиць) та генерувати запити, які є не тільки коректними, але й потенційно ефективними при виконанні на великих обсягах даних.
   * Використання minidev: Для даного дослідження було використано піднабір minidev з BIRD, що містить 500 запитів. Вибір цього піднабору є компромісом між потребою оцінити систему на реалістичних даних та практичними обмеженнями щодо часу та ресурсів, необхідних для повного тестування на BIRD. Піднабір minidev, хоча й менший за повний набір розробки (dev set), все ще містить репрезентативні приклади викликів, характерних для BIRD, включаючи великі схеми та запити, що вимагають аналізу вмісту.
   * Обґрунтування Вибору: Включення BIRD (minidev) є критично важливим для оцінки здатності фреймворку MAC-SQL справлятися саме з тими проблемами, на вирішення яких він був спрямований — робота з великими схемами (тестування агента Selector), складне багатоетапне міркування (Decomposer) та потенційна потреба в корекції запитів через складну взаємодію з даними (Refiner). Це дозволяє оцінити продуктивність системи в сценаріях, максимально наближених до реальних умов експлуатації.

2.5 Створення та Використання Українського Тестового Набору Даних

Мотивація та Необхідність

Стандартні бенчмарки для Text-to-SQL, такі як Spider та BIRD, розроблені переважно для англійської мови. Це створює прогалину в оцінці ефективності систем Text-to-SQL для інших мов, зокрема для української. Оскільки однією з цілей даної роботи є розробка та оцінка системи, здатної обробляти запити українською мовою, виникла необхідність у створенні відповідного тестового набору даних. Пряме застосування англомовних бенчмарків не дозволяє оцінити здатність моделі коректно інтерпретувати лінгвістичні особливості, термінологію та структуру запитів українською мовою в контексті взаємодії з базами даних.

Процес Створення "Silver Dataset"

Для перевірки роботи фреймворку MAC-SQL на українській мові було створено спеціальний тестовий набір даних, який ми називаємо “BIRD-UKR”. Цей датасет був розроблений шляхом штучної генерації даних та пар питання--SQL-запит. Для генерації даних було переважно використано Claude 3.7 Sonnet(Thinking)[4 <https://assets.anthropic.com/m/785e231869ea8b3b/original/claude-3-7-sonnet-system-card.pdf>]. Для генерації пар запит-SQL було обрано Gemini 2.5 PRO[5 <https://blog.google/technology/google-deepmind/gemini-model-thinking-updates-march-2025/#building-on-best-gemini>], що ***(те що геміні дуже гарно генерує sql бо гугл мабуть fine-tune’ли їх для свого google cloud) (no source)***

Процес включав наступні кроки:

1. Аналіз BIRD: Було вивчено структуру та характеристики бенчмарку BIRD, зокрема:
   * Типи запитів та їх складність (необхідність JOIN-ів, агрегацій, вкладених запитів, аналізу вмісту даних).
   * Складність схем баз даних (кількість таблиць, зв'язків).
   * Реалістичність та різноманітність сценаріїв.
2. Розробка Сценаріїв та Запитів: На основі аналізу BIRD було розроблено нові сценарії та відповідні їм природномовні запити безпосередньо українською мовою. Ці запити були сформульовані так, щоб відображати аналогічний рівень складності та типи викликів, що присутні в BIRD (наприклад, запити, що вимагають міркування над значеннями в БД, обробку потенційно великих схем тощо).
3. Створення/Адаптація Схем та Даних: Для підтримки розроблених українських запитів могли бути створені нові схеми баз даних або адаптовані існуючі, потенційно з використанням даних, релевантних для українського контексту.

Важливо зазначити, що отриманий датасет класифікується як "срібний" (silver), а не "золотий" (gold). Це означає, що він є корисним для оцінки, але може містити певні артефакти перекладу або не повністю відображати все різноманіття того, як носії української мови формулювали б запити до баз даних органічно.

Мета Використання в Тестуванні

Включення українського срібного датасету до процесу тестування дозволяє:

1. Оцінити Розуміння Української Мови: Перевірити здатність базової моделі (Llama 3.1 70B) коректно інтерпретувати семантику та синтаксис українських запитів у контексті наданої схеми бази даних.
2. Валідувати Роботу Фреймворку: Переконатися, що логіка агентів Selector, Decomposer та Refiner в рамках MAC-SQL залишається ефективною при зміні мови вхідного запиту.
3. Виявити Специфічні Виклики: Ідентифікувати потенційні труднощі, пов'язані з обробкою української мови (наприклад, багатозначність слів, відмінювання, специфічна термінологія).
4. Забезпечити Базову Оцінку: Надати кількісну оцінку продуктивності розробленої системи безпосередньо для українськомовних сценаріїв.

3 ПРАКТИЧНА ЧАСТИНА

Я манав

3.1 Постановка задачі

В рамках даної дипломної роботи було виконано адаптацію існуючого фреймворку MAC-SQL ([wbbeyourself/MAC-SQL](https://github.com/wbbeyourself/MAC-SQL))[3] для специфічних умов та вимог. Оригінальний фреймворк призначений для перетворення запитів природною мовою на SQL-запити, використовуючи архітектуру з кількох програмних агентів (Selector, Decomposer, Refiner) та спираючись на великі мовні моделі (LLM) через API OpenAI (GPT-4) і бази даних SQLite.

Ключовими завданнями нашої адаптації були:

1. **Заміна LLM та API**: Ми модифікували систему для використання моделі Llama 3.1 70B замість GPT-4. Це включало інтеграцію з платформою Together AI шляхом зміни логіки викликів API у файлах `core/api.py`, `core/llm.py`, `core/api\_config.py` та додавання адаптера `core/macsql\_together\_adapter.py` для управління специфікою API Together AI, зокрема обробкою обмежень швидкості.
2. **Підтримка PostgreSQL та українського датасету**: Ми розширили можливості фреймворку для роботи з СУБД PostgreSQL та українським датасетом BIRD-UKR. Для цього було розроблено спеціалізовані версії агентів (`PostgreSQLSelector`, `PostgreSQLRefiner` у `core/bird\_ukr\_extensions.py`), оновлено утиліти для роботи з базою даних (`core/db\_utils.py`, `utils/pg\_connection.py`) та адаптовано системні промпти для агентів в `core/const\_ukr.py` для коректної роботи з PostgreSQL та українськими назвами в схемі даних.

Ці модифікації дозволили використовувати MAC-SQL з альтернативною LLM та працювати зі специфічним стеком технологій (PostgreSQL) і лінгвістичними особливостями (українська мова), що було необхідно для наших дослідницьких цілей.

3.2 Розробка прототипу MAC-SQL

3.2.1. Адаптація LLM: Перехід з GPT-4 на Llama 3.1 70B через Together AI

Наша мета полягала в інтеграції моделі `Llama 3.1 70B `, доступної через платформу Together AI. Це потребувало модифікації кількох основних компонентів системи.

Зміна логіки виклику API:

Основні зміни були внесені до файлів `core/api.py` та `core/llm.py`. Ми створили нову функцію `together\_api\_call` у `core/api.py`, яка інкапсулює логіку взаємодії з Together AI. Вона формує HTTP POST-запит до ендпоінту `https://api.together.xyz/v1/chat/completions`, передаючи API-ключ у заголовку `Authorization` та необхідні параметри в тілі JSON-запиту:

```python

# Приклад формування запиту в core/api.py (спрощено)

def together\_api\_call(prompt: str) -> Tuple[str, int, int]:

api\_key = os.getenv("TOGETHER\_API\_KEY", "")

model = os.getenv("TOGETHER\_MODEL", "meta-llama/Llama-3.1-70B-Instruct-Turbo")

if not api\_key:

raise ValueError("Together API key not found.")

api\_url = "https://api.together.xyz/v1/chat/completions"

headers = {

"Authorization": f"Bearer {api\_key}",

"Content-Type": "application/json"

}

data = {

"model": model,

"messages": [{"role": "user", "content": prompt}],

"temperature": 0.1, # Або інше значення

"max\_tokens": 4096 # Або інше значення

}

# ... (логіка виконання запиту та обробки відповіді) ...

response = requests.post(api\_url, headers=headers, json=data)

# ... (перевірка статусу, парсинг відповіді) ...

result = response.json()

text = result["choices"][0]["message"]["content"].strip()

prompt\_tokens = result["usage"]["prompt\_tokens"]

completion\_tokens = result["usage"]["completion\_tokens"]

return text, prompt\_tokens, completion\_tokens

```

Рисунок 1 – функція together\_api\_call

Функція-обгортка `safe\_call\_llm` (у `core/llm.py` або `core/api.py`) була оновлена, щоб перевіряти конфігурацію і викликати `together\_api\_call` замість функцій OpenAI, коли це вказано.

# Приклад адаптованої функції safe\_call\_llm (спрощено)

from core.api\_config import USE\_TOGETHER\_AI, MODEL\_NAME

# Можливо, імпорт функції для OpenAI теж потрібен як fallback

# from somewhere import openai\_api\_call

from core.api import together\_api\_call # Імпортуємо нашу функцію

def safe\_call\_llm(input\_prompt, \*\*kwargs) -> str:

global total\_prompt\_tokens, total\_response\_tokens # Для логування

global log\_path, api\_trace\_json\_path # Для логування

for i in range(MAX\_TRY): # MAX\_TRY - кількість спроб

try:

# Вибір функції API на основі конфігурації

if USE\_TOGETHER\_AI:

logger.info(f"Using Together AI model: {MODEL\_NAME}")

sys\_response, prompt\_token, response\_token = together\_api\_call(input\_prompt)

else:

# Виклик резервної функції (наприклад, OpenAI)

logger.info(f"Using OpenAI model: {MODEL\_NAME}")

# sys\_response, prompt\_token, response\_token = openai\_api\_call(input\_prompt)

# Потрібно реалізувати або імпортувати openai\_api\_call

raise NotImplementedError("OpenAI call function is not defined here")

# Логування результату (приклад)

total\_prompt\_tokens += prompt\_token

total\_response\_tokens += response\_token

if log\_path:

with open(log\_path, 'a+', encoding='utf8') as log\_fp:

# ... (запис промпту, відповіді, токенів у лог) ...

pass

if api\_trace\_json\_path:

# ... (запис деталей виклику у JSONL) ...

pass

return sys\_response # Повертаємо успішну відповідь

except Exception as ex:

logger.error(f"API call failed (attempt {i+1}/{MAX\_TRY}): {ex}")

if i < MAX\_TRY - 1:

time.sleep(RETRY\_DELAY \* (2 \*\* i)) # Exponential backoff

else:

logger.error("LLM call failed after multiple retries.")

raise # Перевикинути помилку після всіх спроб

# Цей рядок не мав би досягатися, але про всяк випадок

raise Exception("safe\_call\_llm failed unexpectedly")

Рисунок 2

Налаштування конфігурації:

Файл `core/api\_config.py` був розширений для зчитування конфігурації Together AI з середовища або `.env` файлу. Було додано змінні `TOGETHER\_API\_KEY`, `TOGETHER\_MODEL` та прапорець `USE\_TOGETHER\_AI` для керування вибором API:

```python

# Фрагмент core/api\_config.py

import os

from dotenv import load\_dotenv

load\_dotenv()

# Together AI configuration

TOGETHER\_API\_KEY = os.getenv("TOGETHER\_API\_KEY", "")

TOGETHER\_MODEL = os.getenv("TOGETHER\_MODEL", "meta-llama/Llama-3.1-70B-Instruct-Turbo")

USE\_TOGETHER\_AI = os.getenv("USE\_TOGETHER\_AI", "true").lower() == "true"

# OpenAI configuration (залишено як резерв)

# ... (OPENAI\_API\_KEY, OPENAI\_API\_BASE, etc.)

# Визначення активної моделі

if USE\_TOGETHER\_AI:

MODEL\_NAME = TOGETHER\_MODEL

else:

MODEL\_NAME = os.getenv("OPENAI\_MODEL", "gpt-4-...")

```

Рисунок 3 – фрагмент використання Together API

Обробка відповідей, помилок та обмежень:

Логіка парсингу відповідей у `together\_api\_call` була написана з урахуванням структури відповіді Together AI (доступ до `result["choices"][0]["message"]["content"]` та `result["usage"]`).

Механізм обробки помилок та обмежень швидкості (rate limiting) був реалізований всередині циклу повторних спроб у `together\_api\_call` (як показано в `core/api.py`) або винесений в адаптер (`core/macsql\_together\_adapter.py`). Цей механізм перевіряє статус-код відповіді (наприклад, 429) та за потреби робить паузу перед наступною спробою:

```python

# Приклад логіки retry у core/api.py (спрощено)

MAX\_RETRIES = 5

RETRY\_DELAY = 5 # seconds

for attempt in range(MAX\_RETRIES):

try:

response = requests.post(api\_url, headers=headers, json=data)

if response.status\_code == 429: # Rate limited

wait\_time = RETRY\_DELAY \* (2 \*\* attempt) # Exponential backoff

logger.warning(f"Rate limited. Waiting {wait\_time} seconds...")

time.sleep(wait\_time)

continue # Наступна спроба

if response.status\_code != 200:

# Обробка інших помилок

raise Exception(f"API error: {response.status\_code} - {response.text}")

# Успішна відповідь - парсимо та повертаємо результат

# ... (парсинг result = response.json()) ...

return text, prompt\_tokens, completion\_tokens

except Exception as e:

logger.error(f"Error calling Together API: {str(e)}")

if attempt < MAX\_RETRIES - 1:

time.sleep(RETRY\_DELAY) # Проста затримка перед наступною спробою

else:

raise # Помилка після всіх спроб

```

Рисунок 4 – приклад обробки помилок з API

Адаптація промптів:

Суттєвих змін у системних промптах для агентів Selector, Decomposer та Refiner не знадобилося. Тестування показало, що Llama 3.1 ефективно працює з існуючими інструкціями.

Ці зміни дозволили успішно інтегрувати модель Llama 3.1 через Together AI, забезпечивши гнучкість у виборі LLM та стабільну роботу завдяки механізмам обробки помилок API.

3.2.2 Адаптація для PostgreSQL та українського датасету (BIRD-UKR)

Другим важливим етапом адаптації MAC-SQL була реалізація підтримки системи управління базами даних PostgreSQL та специфічного датасету BIRD-UKR. Цей датасет відрізняється тим, що використовує PostgreSQL як СУБД, а всі назви таблиць, стовпців та дані представлені українською мовою. Це вимагало значних модифікацій у компонентах, що взаємодіють з базою даних та генерують SQL-запити.

Мотивація: Необхідність роботи з бенчмарком BIRD-UKR, який є адаптацією BIRD для української мови та використовує PostgreSQL, зробила стандартну реалізацію MAC-SQL, орієнтовану на SQLite, непридатною. Потрібно було навчити систему "розуміти" українські схеми та генерувати синтаксично коректні запити для PostgreSQL.

Спеціалізовані агенти для PostgreSQL:

* PostgreSQLSelector`: Було створено новий клас селектора (розміщений в `utils/pg\_selector.py` та інтегрований у `core/bird\_ukr\_extensions.py` через функцію `load\_pg\_selector`), який спеціалізується на роботі з PostgreSQL. На відміну від стандартного `Selector`, що використовує `PRAGMA table\_info` для SQLite, цей агент підключається до PostgreSQL за допомогою бібліотеки `psycopg2` і виконує запити до системних таблиць `information\_schema` для отримання інформації про структуру бази даних (таблиці, стовпці, типи даних, первинні та зовнішні ключі). Цей підхід дозволяє коректно працювати з українськими ідентифікаторами, оскільки `information\_schema` є стандартизованим способом отримання метаданих.
* PostgreSQLRefiner`: Було створено клас `PostgreSQLRefiner` у файлі `core/bird\_ukr\_extensions.py`. Цей агент замінив стандартний `Refiner`. Його ключові відмінності:

1. Виконання SQL: Метод `\_execute\_sql` використовує `psycopg2` для підключення до відповідної бази даних PostgreSQL (використовуючи `db\_id` як ім'я бази даних та облікові дані з конфігурації) та виконання переданого SQL-запиту. Він також обробляє специфічні винятки `psycopg2.Error`.

```python

# Фрагмент методу \_execute\_sql в PostgreSQLRefiner (core/bird\_ukr\_extensions.py)

import psycopg2

from psycopg2.extras import RealDictCursor

# ...

def \_execute\_sql(self, db\_id: str, sql: str) -> Tuple[bool, Optional[List[Dict]], Optional[str]]:

try:

conn = psycopg2.connect(

dbname=db\_id,

user=self.pg\_user, # Змінні з os.environ

password=self.pg\_password,

host=self.pg\_host,

port=self.pg\_port

)

cursor = conn.cursor(cursor\_factory=RealDictCursor) # Зручний курсор

cursor.execute("SET statement\_timeout = 30000") # Обмеження часу

cursor.execute(sql)

result = cursor.fetchall()

result\_list = [dict(row) for row in result]

cursor.close()

conn.close()

return True, result\_list, None

except psycopg2.Error as e: # Обробка помилок PostgreSQL

logger.error(f"SQL execution error: {str(e)}")

# Повернення помилки для подальшого аналізу

return False, None, str(e)

# ... (інші обробки винятків)

```

Рисунок 5 – виконання запиту PostgreSQL

1. Обробка помилок: Метод `\_refine` був адаптований для аналізу помилок, специфічних для PostgreSQL (наприклад, синтаксичні помилки, неправильні типи даних, помилки з ідентифікаторами). Він використовує оновлений шаблон (`refiner\_template\_ukr` з `core/const\_ukr.py`) для генерації промпту LLM, передаючи текст помилки PostgreSQL для виправлення.

З'єднання з БД:

Утиліти для роботи з базами даних, такі як `core/db\_utils.py` та `utils/pg\_connection.py` (якщо використовується пул з'єднань), були модифіковані. Функція `get\_db\_connection` у `core/db\_utils.py` тепер перевіряє `dataset\_name`. Якщо це `bird-ukr`, вона використовує `psycopg2` для встановлення з'єднання з PostgreSQL, зчитуючи параметри підключення (хост, порт, користувач, пароль) зі змінних середовища (`PG\_HOST`, `PG\_PORT`, `PG\_USER`, `PG\_PASSWORD`). Для інших датасетів використовується стандартне підключення до SQLite.

```python

# Фрагмент get\_db\_connection в core/db\_utils.py

def get\_db\_connection(dataset\_name, db\_id, db\_base\_path=None):

# ...

if dataset\_name == 'bird-ukr':

db\_type = 'postgres'

if not PSYCOPG2\_AVAILABLE: # Перевірка наявності psycopg2

raise ImportError("psycopg2 is required for PostgreSQL...")

try:

conn = psycopg2.connect(

host=os.getenv("PG\_HOST", "localhost"),

port=os.getenv("PG\_PORT", "5432"),

user=os.getenv("PG\_USER", "postgres"),

password=os.getenv("PG\_PASSWORD", ""),

dbname=db\_id # db\_id використовується як ім'я БД

)

except psycopg2.Error as e:

raise ConnectionError(f"Failed to connect to PostgreSQL DB {db\_id}: {e}")

elif dataset\_name in ['spider', 'bird']:

db\_type = 'sqlite'

# ... (логіка для SQLite) ...

# ...

return conn, db\_type

```

Рисунок 6 – фрагмент коду з’єднання з PostgreSQL

Промпти для української схеми та PostgreSQL:

Файл `core/const\_ukr.py` містить адаптовані версії системних промптів для агентів (`selector\_template\_ukr`, `decomposer\_template\_ukr`, `refiner\_template\_ukr`). Ці промпти інструктують LLM враховувати особливості PostgreSQL синтаксису (наприклад, використання `TRUE`/`FALSE` замість `1`/`0`, правила цитування ідентифікаторів, використання `COUNT(\*)`) та наявність українських назв таблиць і стовпців. LLM надається явна вказівка не перекладати ідентифікатори та використовувати їх у згенерованому SQL "як є".

```python

# Фрагмент decomposer\_template\_ukr з core/const\_ukr.py

"""

...

[IMPORTANT: UKRAINIAN DATABASE WITH POSTGRESQL SYNTAX]

- This is a UKRAINIAN dataset - ALL table and column names are in UKRAINIAN, not English

- This is a POSTGRESQL database, not SQLite - follow PostgreSQL syntax

- Use TRUE/FALSE for boolean values, not 1/0

- Never translate table or column names to English in your response

- Do not use = 1 for boolean conditions, use = TRUE instead

- Do not use = 0 for boolean conditions, use = FALSE instead

- ALWAYS use COUNT(\*) instead of COUNT(column\_name) unless you need to count specific non-null values

...

[Important PostgreSQL Constraints]

...

2. \*\*Quoting Identifiers:\*\* Use double quotes (`"`) for Ukrainian table and column names \*if\* they contain spaces, special characters, or are PostgreSQL reserved words...

...

"""

```

Рисунок 7 – приклад коду з промптами для роботи з PostgreSQL

Обробка кодування: Хоча можуть виникати проблеми з кодуванням при роботі з різними мовами, бібліотека `psycopg2` зазвичай коректно обробляє UTF-8 (стандартне кодування для PostgreSQL), тому додаткових специфічних кроків для обробки українських символів на рівні підключення не знадобилося, за умови правильного налаштування самої бази даних PostgreSQL.

Завдяки цим змінам фреймворк MAC-SQL було успішно адаптовано для роботи з PostgreSQL та україномовним датасетом BIRD-UKR, що відкрило можливість його застосування для специфічних завдань, пов'язаних з українськими даними.

3.2.3 Про форматування схеми:

* **PostgreSQLSelector (utils/pg\_selector.py):** Було створено спеціалізований клас селектора, успадкований від BaseAgent. Його ключова функція - отримання та форматування схеми бази даних PostgreSQL для подальшої обробки LLM.
* **Отримання схеми (метод get\_schema):** Цей метод реалізує логіку взаємодії з PostgreSQL для збору метаданих:

1. Встановлюється **нове з'єднання** з цільовою базою даних (dbname=db\_id) за допомогою psycopg2.connect, використовуючи облікові дані (хост, порт, користувач, пароль), отримані зі змінних середовища (PG\_HOST, PG\_PORT, PG\_USER, PG\_PASSWORD). Використання нового з'єднання на цьому етапі (замість пулу з utils/pg\_connection.py) може бути специфікою реалізації PostgreSQLSelector.
2. Виконується SQL-запит до information\_schema.tables для отримання списку всіх таблиць у схемі public.
3. Для кожної таблиці виконується запит до information\_schema.columns та information\_schema.table\_constraints, щоб отримати детальну інформацію про стовпці: ім'я (column\_name), тип даних (data\_type), чи є стовпець nullable (is\_nullable), значення за замовчуванням (column\_default) та чи є він частиною первинного ключа (is\_primary).
4. **Важливо:** Для кожного стовпця робиться спроба отримати до 5 **прикладів значень** (sample\_values) безпосередньо з таблиці за допомогою запиту SELECT "{column\_name}" FROM "{table}" WHERE "{column\_name}" IS NOT NULL LIMIT 5. Це збагачує схему контекстом реальних даних.
5. Виконується запит до information\_schema (комбінуючи table\_constraints, key\_column\_usage, constraint\_column\_usage) для отримання інформації про **зовнішні ключі** (зв'язки між таблицями).
6. Вся зібрана інформація (назви таблиць, деталі стовпців з прикладами значень, зовнішні ключі) зберігається у словнику schema\_info.
7. З'єднання з базою даних закривається.

* **Форматування схеми (метод format\_schema):** Цей метод приймає словник schema\_info, отриманий від get\_schema, і перетворює його на два рядки, призначені для промпту LLM:
* desc\_str: Містить опис таблиць та їх стовпців у форматі, схожому на той, що використовується в core/const.py (наприклад, # Table: назва\_таблиці \n [ (назва\_стовпця тип\_даних [PRIMARY KEY]. Value examples: [значення1, 'значення2']), ... ]). Це допомагає LLM зрозуміти структуру.
* fk\_str: Містить опис зовнішніх ключів у форматі таблиця1.стовпець1 references таблиця2.стовпець2.
* **Взаємодія з LLM (метод talk):** Метод talk оркеструє процес. Він викликає get\_schema, потім format\_schema, формує повний промпт для LLM, використовуючи шаблон selector\_template\_ukr (з core/const\_ukr.py) та отримані desc\_str, fk\_str, а також запит користувача та докази (evidence). Потім він викликає LLM (call\_llm з core/api.py) для визначення релевантних частин схеми. Після отримання відповіді від LLM, він намагається витягти JSON з описом релевантних таблиць/стовпців і формує **відфільтровані** desc\_str та fk\_str, які передаються наступному агенту (Decomposer). Якщо парсинг відповіді LLM не вдається, використовується повна схема як резервний варіант.

3.2.4 Інтеграція та тестування

Після розробки адаптованих компонентів для роботи з Together AI та PostgreSQL, необхідно було інтегрувати їх у загальний робочий процес фреймворку MAC-SQL та перевірити працездатність системи на всіх цільових конфігураціях.

Інтеграція в ChatManager:

* Основний клас, що керує взаємодією агентів, – `EnhancedChatManager` (`core/enhanced\_chat\_manager.py`). В його конструкторі реалізовано логіку вибору відповідних агентів залежно від параметра `dataset\_name`:
  + Для `bird-ukr`: завантажуються агенти з підтримкою PostgreSQL (`PostgreSQLSelector`, `PostgreSQLRefiner`) через `load\_bird\_ukr\_extensions`.
  + Для `spider` та `bird`: завантажуються відповідні стандартні агенти (`Selector`, `Refiner`) або їх розширені версії для SQLite (наприклад, `EnhancedSpiderSelector`, `EnhancedBirdRefiner`), якщо вони доступні (`HAS\_SPIDER\_EXTENSIONS`, `HAS\_BIRD\_EXTENSIONS` = True).
* Незалежно від обраного набору агентів, система використовувала адаптовану функцію `safe\_call\_llm` для взаємодії з LLM (Llama 3.1 через Together AI), якщо `USE\_TOGETHER\_AI` встановлено в `True`.
* Механізм маршрутизації повідомлень між агентами залишився універсальним, спираючись на імена агентів.

Тестування:

Було проведено комплексне тестування для перевірки різних аспектів адаптації:

1. Тестування на BIRD-UKR: Основний сценарій для перевірки коректної роботи з PostgreSQL, українською схемою та Llama 3.1. Процес включав запуск запитів з BIRD-UKR, генерацію SQL через адаптований ланцюжок агентів (`PostgreSQLSelector` -> `Decomposer` -> `PostgreSQLRefiner`), виконання згенерованого та еталонного SQL на PostgreSQL та порівняння результатів для розрахунку `Execution Accuracy`.

2. Тестування на Spider та BIRD (SQLite): Цей етап був важливим для:

* Валідації адаптації LLM: Перевірки, наскільки добре Llama 3.1 (через Together AI) справляється з генерацією SQL для стандартних англомовних датасетів Spider та BIRD при роботі зі SQLite.
* Перевірки зворотної сумісності: Гарантії того, що зміни, внесені для підтримки PostgreSQL, не порушили функціональність системи при роботі з оригінальними SQLite-базами даних.
* Процес тестування для Spider/BIRD був аналогічним:
* `EnhancedChatManager` запускався з `dataset\_name='spider'` або `dataset\_name='bird'`.
* Використовувався ланцюжок агентів, призначений для SQLite (стандартні або розширені `Selector`/`Refiner`).
* Виклики LLM йшли до Llama 3.1 через адаптований `safe\_call\_llm`.
* Згенеровані та еталонні SQL-запити виконувались на відповідних SQLite базах даних.
* Результати порівнювалися, і розраховувалась `Execution Accuracy` для цих датасетів.
* Аналіз результатів: Для всіх датасетів збиралася статистика точності виконання (`Execution Accuracy`), аналізувалися логи (`api\_trace.json`, файли логів) для виявлення помилок у генерації SQL, проблем з виконанням запитів чи взаємодією з API. Результати тестування на всіх трьох датасетах підтвердили, що:
  + Інтеграція з Llama 3.1 через Together AI є робочою та забезпечує генерацію SQL-запитів.
  + Адаптація для PostgreSQL та BIRD-UKR є функціональною і дозволяє працювати зі специфічним стеком.
  + Зворотна сумісність зі Spider та BIRD (SQLite) збережена.

3.3 Розробка україномовного набору даних

Для об'єктивної оцінки та покращення систем Text2SQL необхідні якісні та репрезентативні бенчмарки. У той час як для англійської мови існують розвинені ресурси, такі як Spider та BIRD, аналогічні інструменти для української мови практично відсутні. Англомовний бенчмарк BIRD вирізняється тим, що він фокусується на запитах середньої та високої складності, які потребують багатоетапного логічного мислення та врахування зовнішніх знань (evidence), що робить його гарним інструментом для оцінки глибини "розуміння" моделі. Відсутність подібного україномовного бенчмарку створювала значні перешкоди для:

* **Оцінки реальної продуктивності LLM:** Неможливо було надійно оцінити, наскільки добре існуючі моделі (навіть ті, що підтримують українську мову) справляються зі складними NL2SQL завданнями саме в українському контексті.
* **Цільової розробки та донавчання моделей:** Без спеціалізованого датасету складно було спрямовано покращувати моделі для кращого розуміння української мови та генерації коректних SQL-запитів для типових українських даних.
* **Порівняння різних підходів:** Не було спільної основи для порівняння ефективності різних архітектур NL2SQL систем (як, наприклад, MAC-SQL) при роботі з українською мовою.

Саме для вирішення цих проблем було створено україномовний набір даних. **Основна мета BIRD-UKR:** розробити комплексний та реалістичний бенчмарк для оцінки та розвитку NL2SQL систем, орієнтованих на українську мову. Ключові характеристики та цілі BIRD-UKR включають:

* **Українська мова:** Усі компоненти датасету (назви таблиць, стовпців, дані, запити природною мовою, еталонні SQL-запити) виконані виключно українською мовою.
* **PostgreSQL:** Використання СУБД PostgreSQL, яка є поширеною у багатьох реальних системах, на відміну від SQLite, що використовується в оригінальному BIRD для спрощення.
* **Різноманітність доменів:** Включення 8 баз даних з різних предметних областей (медицина, освіта, торгівля тощо) для забезпечення широкого покриття сценаріїв.
* **Рівні складності:** Наявність запитів трьох рівнів складності (простий, середній, складний) для перевірки різних аспектів генерації SQL (від простих SELECT до складних JOIN, підзапитів та віконних функцій).
* **Реалістичність даних:** Генерація синтетичних, але правдоподібних даних українською мовою для наповнення баз даних.

3.3.1 Проектування

Основою будь-якого NL2SQL бенчмарку є набір реалістичних та різноманітних баз даних. Для BIRD-UKR було обрано 8 предметних областей, щоб охопити широкий спектр можливих запитів та сценаріїв використання:

1. **Лікарня (лікарня):** Медична інформація (пацієнти, лікарі, діагнози, візити).
2. **Бібліотека (бібліотека):** Книги, читачі, видачі, бронювання.
3. **Університет (університет):** Студенти, викладачі, курси, розклад, оцінки.
4. **Інтернет-магазин (інтернет\_магазин):** Товари, категорії, клієнти, замовлення, відгуки.
5. **Ресторан (ресторан):** Меню, персонал, замовлення, столики, резервації.
6. **Туристичне агентство (туристичне\_агентство):** Тури, готелі, клієнти, бронювання, платежі.
7. **Авіакомпанія (авіакомпанія):** Рейси, літаки, аеропорти, пасажири, бронювання, персонал.
8. **Спортивний клуб (спортивний\_клуб):** Члени клубу, тренери, заняття, абонементи, розклад.

Процес проектування схеми для кожної бази даних включав наступні кроки:

* **Розробка логічної структури:** Для кожного домену визначалися основні сутності та зв'язки між ними, які потім трансформувалися у таблиці та стовпці.
* **Визначення атрибутів та типів даних:** Для кожного стовпця підбирався відповідний тип даних PostgreSQL (наприклад, VARCHAR, INTEGER, DECIMAL, DATE, TIMESTAMP, BOOLEAN), що забезпечувало адекватне представлення інформації.
* **Забезпечення цілісності даних:** Визначалися первинні ключі (PRIMARY KEY) для унікальної ідентифікації записів у таблицях та зовнішні ключі (FOREIGN KEY) для встановлення та підтримки зв'язків між таблицями. Додаткові обмеження (CHECK, NOT NULL, UNIQUE) використовувалися для забезпечення валідності даних.
* **Використання українських ідентифікаторів:** **Ключовою особливістю** проектування було використання **виключно українських назв** для всіх об'єктів бази даних – таблиць (наприклад, пацієнти, рейси) та стовпців (наприклад, прізвище, дата\_народження, аеропорт\_відправлення\_ід). Це було зроблено свідомо, щоб створити бенчмарк, максимально наближений до реальних українських інформаційних систем, та перевірити здатність LLM працювати з неанглійськими ідентифікаторами.

Результатом цього етапу стали SQL-скрипти schema.sql для кожної з 8 баз даних (розміщені у відповідних піддиректоріях MAC-SQL/data/bird-ukr/database/<db\_name>/), які містять CREATE TABLE інструкції для створення повної структури бази даних у PostgreSQL.

-- Фрагмент схеми бази даних "Ресторан" (ілюстрація)

-- Таблиця "Категорії страв" (довідник)

CREATE TABLE категорії (

ід SERIAL PRIMARY KEY, -- PostgreSQL автоінкремент

назва VARCHAR(100) NOT NULL UNIQUE,

опис TEXT,

активна BOOLEAN DEFAULT TRUE -- Тип BOOLEAN

);

-- Таблиця "Страви"

CREATE TABLE страви (

ід SERIAL PRIMARY KEY,

назва VARCHAR(255) NOT NULL,

опис TEXT,

ціна DECIMAL(10, 2) NOT NULL, -- Тип DECIMAL для грошей

категорія\_ід INTEGER NOT NULL REFERENCES категорії(ід), -- Зовнішній ключ

вегетаріанська BOOLEAN DEFAULT FALSE,

вага\_грам INTEGER,

час\_приготування\_хв INTEGER,

активна BOOLEAN DEFAULT TRUE,

CONSTRAINT ціна\_не\_відємна CHECK (ціна >= 0) -- Обмеження CHECK

);

-- Таблиця "Інгредієнти"

CREATE TABLE інгредієнти (

ід SERIAL PRIMARY KEY,

назва VARCHAR(150) NOT NULL UNIQUE,

одиниця\_виміру VARCHAR(20) NOT NULL, -- кг, л, шт

алерген BOOLEAN DEFAULT FALSE

);

-- Таблиця "Склад страви" (зв'язок багато-до-багатьох)

CREATE TABLE склад\_страви (

страва\_ід INTEGER NOT NULL REFERENCES страви(ід) ON DELETE CASCADE, -- ON DELETE CASCADE

інгредієнт\_ід INTEGER NOT NULL REFERENCES інгредієнти(ід),

кількість DECIMAL(8, 3) NOT NULL, -- Наприклад, 0.150 кг

PRIMARY KEY (страва\_ід, інгредієнт\_ід) -- Складений первинний ключ

);

-- Індекс для прискорення пошуку страв за категорією

CREATE INDEX idx\_страви\_категорія ON страви(категорія\_ід);

Рисунок 8 – фрагмент коду створення схеми однієї з БД

3.3.2 Генерація синтетичних даних

Після проектування схем баз даних наступним кроком було їх наповнення реалістичними, але синтетичними даними українською мовою. Це критично важливо для того, щоб запити до бази даних повертали осмислені результати, і щоб LLM могли краще розуміти семантику стовпців на основі їх вмісту.

Інструмент генерації: Для цього завдання було обрано потужну велику мовну модель **Claude 3.7 Sonnet**. Її можливості у генерації тексту, що відповідає заданим інструкціям та контексту, дозволили створити великі обсяги різноманітних та правдоподібних даних.

Процес генерації:

Формулювання промптів: Для кожної таблиці в базі даних формувався спеціальний промпт для Claude. Цей промпт зазвичай містив:

* `CREATE TABLE` інструкцію для відповідної таблиці (щоб модель бачила назви стовпців, їх типи та обмеження).
* Вказівку генерувати певну кількість рядків даних (наприклад, "згенерируй 50 записів").
* Вимоги до даних: використовувати \*\*виключно українську мову\*\*, генерувати реалістичні значення (наприклад, правдоподібні імена, назви, дати, ціни), дотримуватися форматів даних (наприклад, для дат, телефонів).
* Вказівку генерувати результат у вигляді SQL-команд `INSERT INTO`.

Генерація та збереження даних: Claude генерував набір `INSERT` команд для кожної таблиці. Ці команди потім зберігалися в окремих SQL-файлах, зазвичай з префіксом `data\_`, наприклад:

* `data\_reference.sql` (для довідникових таблиць типу статусів, категорій тощо).
* `data\_персонал.sql`, `data\_страви.sql`, `data\_рейси.sql` (для основних таблиць).

Цей поділ на файли полегшував керування процесом та забезпечував правильний порядок завантаження даних з урахуванням залежностей (зовнішніх ключів).

Забезпечення консистентності: Під час генерації даних для таблиць, пов'язаних зовнішніми ключами, особлива увага приділялася тому, щоб згенеровані значення ключів відповідали існуючим записам у батьківських таблицях. Наприклад, при генерації даних для таблиці `страви`, значення `категорія\_ід` мали відповідати реальним `ід` з таблиці `категорії`, які генерувалися раніше. Це частково контролювалося промптами, а частково – ручною перевіркою та коригуванням згенерованих даних.

Приклад згенерованих даних (фрагмент `data\_\*.sql`):

```sql

-- Приклад для data\_reference.sql (категорії ресторану)

INSERT INTO категорії (назва, опис, активна) VALUES

('Гарячі закуски', 'Різноманітні теплі закуски до основного столу', TRUE),

('Супи', 'Перші страви, борщі, бульйони', TRUE),

('Основні страви', 'Страви з м''яса, риби та птиці', TRUE),

('Десерти', 'Солодкі страви та випічка', TRUE);

-- Приклад для data\_страви.sql (страви ресторану)

INSERT INTO страви (назва, опис, ціна, категорія\_ід, вегетаріанська, вага\_грам, час\_приготування\_хв, активна) VALUES

('Борщ український', 'Традиційний борщ з пампушками', 150.00, 2, FALSE, 400, 30, TRUE), -- категорія\_ід = 2 (Супи)

('Деруни зі сметаною', 'Картопляні деруни, подаються зі сметаною', 120.50, 1, TRUE, 300, 20, TRUE), -- категорія\_ід = 1 (Гарячі закуски)

('Київський торт', 'Класичний київський торт з горіхами', 180.00, 4, FALSE, 150, 10, TRUE); -- категорія\_ід = 4 (Десерти)

```

Імпорт даних: Для автоматизації процесу наповнення бази даних PostgreSQL був створений скрипт `import.sql` для кожного домену. Цей скрипт містить послідовність команд `\i` (команда `psql` для виконання іншого скрипту), які в правильному порядку викликають файли `data\_\*.sql`, забезпечуючи завантаження даних з дотриманням залежностей. Також може використовуватися Python-скрипт `import\_data.py` для більш гнучкого керування процесом імпорту.

Таким чином, за допомогою LLM Claude 3.7 Sonnet та ретельно розроблених промптів було згенеровано значний обсяг синтетичних, але реалістичних україномовних даних, що забезпечило основу для функціонування бенчмарку BIRD-UKR.

3.3.3 Створення пар Питання – SQL запит

Методологія створення пар:

Формулювання запитань: Для кожної з 8 баз даних було сформульовано набір запитань **природною українською мовою**. Запитання створювалися так, щоб вони відображали типові інформаційні потреби користувачів у відповідній предметній області... охопити різні рівні складності (простий, середній, складний)...

Генерація еталонних SQL (Gold SQL) за допомогою LLM: Замість повністю ручного написання, для створення еталонних SQL-запитів було використано можливості великої мовної моделі Gemini 2.5 Pro. Ця модель була обрана через її відомі сильні сторони у генерації SQL-коду та велике контекстне вікно[no source, community-proven], що дозволяло їй ефективно аналізувати схему бази даних (надану у промпті) та складне запитання українською мовою.

* Процес генерації: Для кожного запитання природною мовою формувався промпт для Gemini 2.5 Pro, який включав:
* Текст запитання українською.
* Повну або релевантну частину схеми бази даних PostgreSQL (українською мовою).
* Чітку інструкцію згенерувати коректний та оптимальний SQL-запит для PostgreSQL, що відповідає на поставлене запитання.
* Модель Gemini 2.5 Pro генерувала SQL-запит, який потім розглядався як кандидат на еталонний ("золотий") стандарт.

Валідація та коригування: Незважаючи на високу якість генерації SQL моделлю Gemini 2.5 Pro, кожен згенерований запит проходив етап перевірки та потенційного коригування\*. Це було необхідно для забезпечення:

* Точної відповідності семантиці оригінального запитання природною мовою.
* Синтаксичної коректності саме для PostgreSQL.
* Правильності використання українських ідентифікаторів зі схеми.
* Логічної коректності виконання запиту на базі даних, наповненій синтетичними даними (перевірка результатів).

У випадках, коли згенерований SQL потребував доопрацювання, вносилися ручні правки. Цей підхід поєднав потужність LLM для масової генерації з людським контролем для забезпечення найвищої якості еталонних запитів.

Структура та зберігання: Усі фінальні (перевірені та, за потреби, скориговані) пари "питання-SQL" разом з метаданими зберігаються у централізованому JSON-файлі `questions.json`:

[

{

"question\_id": "авіакомпанія\_gen\_001", // Унікальний ID питання

"db\_id": "авіакомпанія", // ID бази даних

"question": "Скільки аеропортів знаходиться у місті Київ?", // Питання українською

"gold\_sql": "SELECT COUNT(\*) FROM аеропорти WHERE місто = 'Київ';", // Еталонний SQL

"difficulty": "simple" // Рівень складності

},

{

"question\_id": "лікарня\_gen\_014",

"db\_id": "лікарня",

"question": "Знайдіть пацієнтів, які мали візити до кардіолога та невролога.",

"gold\_sql": "WITH PatientVisits AS ( SELECT в.пацієнт\_ід, с.назва AS спеціалізація FROM візити в JOIN персонал п ON в.лікар\_ід = п.ід JOIN спеціалізації\_лікарів сл ON п.ід = сл.лікар\_ід JOIN спеціалізації с ON сл.спеціалізація\_ід = с.ід WHERE с.назва IN ('Кардіологія', 'Неврологія') ) SELECT п.прізвище, п.імя FROM пацієнти п JOIN PatientVisits pv ON п.ід = pv.пацієнт\_ід GROUP BY п.ід, п.прізвище, п.імя HAVING COUNT(DISTINCT pv.спеціалізація) = 2;",

"difficulty": "complex"

}

// ... інші питання ...

]

Рисунок 9 – приклад згенерованих пар питання-SQL

3.3.4 Структура та компоненти бенчмарку BIRD-UKR

Розроблений бенчмарк BIRD-UKR має чітку файлову структуру, що полегшує його використання та інтеграцію з різними системами тестування. Основні компоненти розташовані у директорії `MAC-SQL/data/bird-ukr` (або аналогічній, залежно від розміщення проекту):

`questions.json`: Центральний файл, що містить масив JSON-об'єктів. Кожен об'єкт представляє одну пару "питання-SQL" і включає поля:

* `question\_id`: Унікальний ідентифікатор.
* `db\_id`: Ідентифікатор бази даних, до якої відноситься запит (наприклад, `лікарня`, `авіакомпанія`).
* `question`: Текст запитання природною українською мовою.
* `gold\_sql`: Еталонний SQL-запит для PostgreSQL.
* `difficulty`: Рівень складності запиту (`simple`, `medium`, `complex`).

`tables.json`: Файл, що містить опис схем усіх баз даних у форматі JSON. Цей формат може використовуватися деякими інструментами аналізу або безпосередньо моделями.

`database/` (директорія): Містить піддиректорії для кожної з 8 баз даних, що входять до бенчмарку. Назва кожної піддиректорії відповідає `db\_id` (наприклад, `database/лікарня/`, `database/університет/`).

Внутрішня структура `database/<db\_name>/`: Кожна директорія бази даних містить стандартизований набір файлів:

* `schema.sql`: SQL-скрипт з командами `CREATE TABLE` для створення повної структури таблиць, індексів та обмежень для даної бази даних у PostgreSQL.
* `data\_\*.sql` / `data\_reference.sql`: Набір SQL-скриптів, що містять команди `INSERT INTO` для наповнення таблиць синтетичними даними. Зазвичай існує файл `data\_reference.sql` для довідників та окремі файли `data\_<table\_name>.sql` для основних таблиць. Порядок виконання цих скриптів важливий через залежності зовнішніх ключів.
* `import.sql`: Головний скрипт для імпорту даних. Він використовує команду `\i` клієнта `psql` для послідовного виконання всіх необхідних файлів `schema.sql` та `data\_\*.sql`, створюючи та наповнюючи базу даних.

Така структура дозволяє чітко розділити схеми, дані та запити, а також надає інструменти для автоматизованого розгортання та тестування.

3.3.5 Інструменти тестування та оцінки

Для ефективного використання бенчмарку BIRD-UKR та об'єктивної оцінки NL2SQL систем було розроблено набір допоміжних скриптів та стандартизовано процес тестування.

Утиліти для роботи з датасетом:

utils/bird\_ukr\_loader.py`: Цей скрипт відповідає за завантаження та обробку даних з файлу `questions.json`. Він дозволяє:

* Завантажити повний набір питань або обмежену вибірку (`num\_samples`).
* Вибрати питання випадковим чином (`random\_sample=True`) або послідовно.
* Фільтрувати питання за певними `db\_id`.
* Містить функцію `normalize\_ukr\_query` для нормалізації SQL-запитів перед порівнянням на точну відповідність (Exact Match).

`utils/pg\_connection.py`: Надає функціонал для взаємодії з базами даних PostgreSQL. Ключові можливості:

* Ініціалізація та управління пулами з'єднань (`init\_connection\_pool`, `get\_pool\_connection`, `return\_connection`, `close\_connection\_pool`) для ефективного використання ресурсів при тестуванні великої кількості запитів.
* Виконання SQL-запитів (`execute\_query`) з вимірюванням часу виконання та обробкою помилок PostgreSQL.
* Порівняння результатів виконання двох запитів (`compare\_query\_results`) з урахуванням порядку рядків (перетворення на множини або сортування перед порівнянням).

`utils/bird\_ukr\_tables\_adapter.py`: Допоміжний скрипт для конвертації формату `tables.json` (якщо оригінальний формат BIRD-UKR відрізняється від очікуваного іншими інструментами, наприклад, MAC-SQL). Функція `generate\_compatible\_tables\_json` може використовуватися для підготовки файлу схеми перед тестуванням.

Основний скрипт оцінки (`test\_macsql\_agent\_bird\_ukr.py`): Це центральний скрипт, який проводить оцінку конкретної NL2SQL системи (в нашому випадку, адаптованого MAC-SQL) на бенчмарку BIRD-UKR. Його типовий робочий процес включає:

1. Парсинг аргументів командного рядка (кількість запитів, шлях до даних, модель тощо).

2. Завантаження тестової вибірки питань за допомогою `utils/bird\_ukr\_loader.py`.

3. Ініціалізація системи, що тестується (наприклад, створення екземпляру `UkrainianBirdAdapter` або `EnhancedChatManager`).

4. Ініціалізація пулів з'єднань для всіх необхідних баз даних за допомогою `utils/pg\_connection.py`.

5. Ітерація по завантажених питаннях:

* Для кожного питання викликається метод системи для генерації SQL-запиту (`pred\_sql`).
* За допомогою `utils/pg\_connection.execute\_query` виконується як згенерований `pred\_sql`, так і еталонний `gold\_sql` на відповідній базі даних PostgreSQL, фіксується успішність та час виконання.
* Результати виконання порівнюються за допомогою `utils/pg\_connection.compare\_query\_results` для визначення `execution\_match`.
* Розраховується `exact\_match` шляхом порівняння нормалізованих версій `pred\_sql` та `gold\_sql`.
* Результати (статус, згенерований SQL, час, метрики) зберігаються.

6. Після обробки всіх питань розраховуються агреговані метрики (Execution Accuracy, Exact Match Accuracy, середній час виконання).

7. Результати виводяться на екран та/або зберігаються у JSON-файл.

8. Закриття пулів з'єднань.

Скрипти-обгортки (`run\_bird\_ukr\_evaluation.py` або аналогічні): Можуть використовуватися для зручного запуску `test\_macsql\_agent\_bird\_ukr.py` з різними параметрами конфігурації (наприклад, різні моделі, кількість запитів).

Ці інструменти забезпечують стандартизований та автоматизований підхід до тестування NL2SQL систем на бенчмарку BIRD-UKR.

Метрики оцінки якості моделей на бенчмарку BIRD-UKR:

Для кількісної оцінки продуктивності NL2SQL систем на датасеті BIRD-UKR використовуються дві основні метрики точності та показники ефективності:

1) Точність виконання (Execution Accuracy - EX): Це ключова метрика, що відображає реальну працездатність згенерованого SQL. Вона розраховується як відсоток запитів у тестовій вибірці, для яких виконуються обидві умови:

* + Згенерований SQL-запит успішно виконується на цільовій базі даних PostgreSQL без синтаксичних чи логічних помилок під час виконання.
  + Результат виконання згенерованого запиту (набір повернутих рядків) є повністю ідентичним результату виконання еталонного (`gold\_sql`) запиту. Порівняння результатів проводиться з урахуванням можливої різниці у порядку рядків (зазвичай шляхом порівняння множин кортежів або відсортованих списків кортежів). Ця метрика показує, наскільки добре система генерує запити, що дають правильну відповідь. Розрахунок відбувається у скрипті `test\_macsql\_agent\_bird\_ukr.py`.

2) Точність точної відповідності (Exact Match Accuracy - EM): Це більш сувора метрика, яка вимагає, щоб згенерований SQL-запит **текстово збігався** з еталонним запитом після процесу нормалізації. Нормалізація зазвичай включає: переведення в нижній регістр, видалення коментарів, уніфікацію пробілів, видалення крапки з комою в кінці, потенційно – стандартизацію псевдонімів та форматування літералів (як реалізовано у функції `normalize\_sql` або `normalize\_ukr\_query`). Ця метрика менш показова з точки зору правильності відповіді (бо синтаксично різні запити можуть давати однаковий результат), але може бути корисною для оцінки стилістичної близькості генерації до еталону. Розрахунок відбувається у скрипті `test\_macsql\_agent\_bird\_ukr.py`.

3) Час виконання (Execution Time): Для оцінки ефективності згенерованих запитів вимірюється середній час їх виконання (`avg\_pred\_time`) та порівнюється з середнім часом виконання еталонних запитів (`avg\_gold\_time`). Це дозволяє оцінити, чи не генерує система надто складні або неоптимальні запити. Вимірювання часу відбувається у функції `execute\_query` модуля `utils/pg\_connection.py`.

Ці метрики разом дають комплексне уявлення про якість та ефективність роботи NL2SQL системи на бенчмарку BIRD-UKR.

4 Результати

Після детального опису процесу адаптації фреймворку MAC-SQL для використання великої мовної моделі Llama 3.1 70B через API Together AI та для підтримки СУБД PostgreSQL на основі україномовного бенчмарку BIRD-UKR, цей розділ присвячений представленню та аналізу результатів експериментальної оцінки модифікованої системи.

У цьому розділі будуть представлені результати тестування на всіх трьох зазначених датасетах. Ми проаналізуємо основні метрики якості, такі як точність виконання (Execution Accuracy) та точність точної відповідності (Exact Match Accuracy), а також показники ефективності (час виконання запитів). Результати адаптованої системи будуть порівняні з доступними базовими показниками (baselines), зокрема з результатами оригінального фреймворку MAC-SQL, для кращого розуміння досягнутої продуктивності.

4.1 Базові показники

Для об'єктивної оцінки ефективності адаптованого фреймворку MAC-SQL з моделлю Llama 3.1 необхідно порівняти отримані результати з релевантними базовими показниками (baselines).

* 1. Оригінальний MAC-SQL (з GPT-4):

Згідно з офіційним лідербордом бенчмарку BIRD, оригінальна реалізація фреймворку MAC-SQL у поєднанні з моделлю GPT-4 (станом на листопад 2023 року) демонструвала Точність Виконання (EX) на тестовому (Test) сеті BIRD: 59.59%. Це слугує основним орієнтиром для порівняння нашої адаптації на датасеті BIRD.

Таблиця 2 -- ….

|  |  |
| --- | --- |
| **Few-shot** | **BIRD - EX** |
| 0-shot | 55.54 |
| 1-shot | 57.26 |
| 2-shot | **59.39** |

[3 https://arxiv.org/abs/2312.11242

* 1. Продуктивність базової LLM Llama 3.1 70B (без MAC-SQL):

Як додатковий орієнтир, лідерборд BIRD Mini-Dev показує продуктивність самої моделі Llama3-70b-instruct без агентного фреймворку. На цьому скороченому наборі даних (Mini-Dev) для баз даних SQLite модель досягла Точності Виконання (EX) 40.80%.

Таблиця 3 - Mini Dev - Execution Accuracy (EX)

| **Model** | **Size** | **SQLite** | **MySQL** | **PostgreSQL** |
| --- | --- | --- | --- | --- |
| Llama3-70b-instruct | **70B** | 40.80 | 37.00 | 29.40 |

[? <https://bird-bench.github.io/> ??]

Цей показник демонструє вихідний рівень можливостей моделі Llama 3.1 на цьому типі завдань. Порівняння результатів нашого адаптованого фреймворку MAC-SQL з цим показником (враховуючи різницю між Mini-Dev та повним Dev/Test сетом) дозволить оцінити внесок агентної архітектури MAC-SQL у покращення генерації SQL.

4.2 Результати на стандартних бенчмарках (Spider & BIRD)

Таблиця 4

|  |  |  |  |
| --- | --- | --- | --- |
| Модель | Набір даних | Кількість зразків | EX |
| Llama 3.1 70B | BIRD(minidev-sqlite) | 500 | 51.8% |
| Llama 3.1 70B | BIRD(minidev-sqlite) | 100 | 60% |
| Llama 3.3 70B | BIRD(minidev-sqlite) | 100 | 54% |
| Llama 3.1 70B | Spider(sqlite) | 100 | 64% |
| Deepseek-V3 | BIRD(minidev-sqlite) | 100 | 54% |

1. Різниця в розмірах вибірки:

* Модель Meta-Llama-3.1-70B-Instruct-Turbo показує кращі результати на меншій вибірці (60.0% на 100 зразках) порівняно з більшою вибіркою (51.8% на 500 зразках), що може свідчити про варіативність складності запитань у повному наборі даних BIRD.

1. Порівняння версій Llama:

* Meta-Llama-3.1-70B-Instruct-Turbo (60.0%) перевершує Llama-3.3-70B-Instruct-Turbo (54.0%) на однаковій вибірці у 100 зразків, що є цікавим спостереженням, оскільки версія 3.3 теоретично мала б бути кращою.

1. Покращення завдяки MAC-SQL:

* Використання фреймворку MAC-SQL значно покращує результати Llama-3-70B: з базових 40.8% до 51.8% на 500 зразках і навіть до 60.0% на 100 зразках
* Це демонструє ефективність методології MAC-SQL, яка допомагає структурувати генерацію SQL-запитів

1. Порівняння різних моделей з MAC-SQL:

* Llama-3.1-70B з MAC-SQL (60.0% на 100 зразках) показує результати на рівні або навіть кращі, ніж GPT-4 з MAC-SQL (2-shot: 59.39%)
* На більшій вибірці (500 зразків) Llama-3.1-70B з MAC-SQL (51.8%) показує результати близькі до GPT-4 з MAC-SQL (0-shot: 55.54%)
* Це свідчить про конкурентоспроможність Llama-3.1-70B порівняно з GPT-4 при використанні в рамках спеціалізованого фреймворку MAC-SQL

1. Різниця між розмірами вибірки:

* Зниження продуктивності на більшій вибірці (з 60.0% на 100 зразках до 51.8% на 500 зразках) вказує на можливу непослідовність у обробці складніших запитів
* Проте навіть на 500 зразках поліпшення порівняно з базовим результатом (40.8%) залишається значним (+11%)

4.3 Аналіз здатності моделі працювати з українським датасетом

Таблиця?

|  |  |
| --- | --- |
| Модель | EX |
| Llama 3.1 70B | 57.33% |

Результат 57.33% на українському датасеті є надзвичайно позитивним показником і демонструє, що сучасні великі мовні моделі, зокрема Meta-Llama-3.1-70B-Instruct-Turbo, цілком здатні ефективно працювати з українською мовою та генерувати правильні SQL-запити для PostgreSQL.

**5 Висновок**

Для досягнення поставленої мети було виконано низку ключових завдань. По-перше, було адаптовано існуючий мультиагентний фреймворк MAC-SQL. Ця адаптація включала заміну базової моделі GPT-4 на відкриту модель Llama 3.1 70B, інтеграцію з платформою Together AI для доступу до LLM, та, що найважливіше, реалізацію підтримки цільової СУБД PostgreSQL замість оригінальної SQLite. Були модифіковані компоненти фреймворку (Selector, Refiner) та утиліти для коректної взаємодії зі схемою та виконання запитів у середовищі PostgreSQL, враховуючи його синтаксичні особливості та можливості.

По-друге, усвідомлюючи відсутність спеціалізованих інструментів для оцінки NL2SQL систем для української мови, було розроблено новий україномовний бенчмарк «BIRD-UKR». Цей бенчмарк моделює реалістичні сценарії взаємодії з даними у восьми різних предметних областях (медицина, освіта, торгівля тощо), містить бази даних зі схемами та назвами об'єктів українською мовою, наповнені синтетичними, але правдоподібними даними, та набір пар "питання українською – еталонний SQL-запит для PostgreSQL" різної складності. Створення цього "срібного" датасету включало етапи проектування схем, генерації даних (з використанням Claude 3.7 Sonnet) та генерації пар питання-SQL (з використанням Gemini 2.5 Pro з подальшою валідацією).

По-третє, було проведено комплексне експериментальне тестування адаптованої системи MAC-SQL з моделлю Llama 3.1 70B. Оцінка проводилася на трьох наборах даних: стандартних англомовних бенчмарках Spider та BIRD (minidev subset, з використанням SQLite для порівняння з базовими показниками) та на новоствореному BIRD-UKR (з PostgreSQL).

Результати експериментів продемонстрували високу ефективність та працездатність запропонованого підходу. На англомовних бенчмарках адаптована система показала конкурентоспроможні результати, досягнувши точності виконання (Execution Accuracy, EX) близько 60% на 100 зразках BIRD minidev (SQLite), що підтверджує ефективність архітектури MAC-SQL у поєднанні з Llama 3.1 70B. Найважливішим результатом є показник точності виконання у 57.33% на україномовному бенчмарку BIRD-UKR при використанні PostgreSQL. Це свідчить про те, що обрана модель Llama 3.1 70B здатна ефективно розуміти семантику та структуру запитів українською мовою та генерувати коректні SQL-запити для PostgreSQL у рамках адаптованого фреймворку MAC-SQL. Успішна інтеграція та функціонування системи на стеку Llama 3.1 + Together AI + PostgreSQL + BIRD-UKR є ключовим підтвердженням досягнення цілей роботи.

~~Основні науково-практичні внески роботи полягають у:~~

1. ~~Адаптації мультиагентного фреймворку MAC-SQL для використання з потужною відкритою LLM Llama 3.1 70B та промисловою СУБД PostgreSQL.~~
2. ~~Створенні та оприлюдненні першого комплексного україномовного бенчмарку BIRD-UKR для задач NL2SQL, що включає схеми, дані та пари питання-SQL для PostgreSQL.~~
3. ~~Експериментальній оцінці та підтвердженні спроможності сучасних LLM ефективно працювати з українською мовою в контексті генерації SQL-запитів для реалістичних сценаріїв.~~

Незважаючи на досягнуті позитивні результати, робота має певні обмеження. Бенчмарк BIRD-UKR є "срібним" стандартом, і хоча було докладено зусиль для валідації, він може містити певні артефакти автоматичної генерації. Продуктивність системи залежить від конкретної LLM, і її поведінка на значно складніших або специфічних запитах потребує подальшого вивчення. Питання оптимізації згенерованих SQL-запитів та аспекти безпеки (наприклад, запобігання SQL-ін'єкціям у реальних розгортаннях) не були глибоко досліджені в рамках цієї роботи.

Подальші дослідження можуть бути спрямовані на вдосконалення бенчмарку BIRD-UKR (створення "золотого" стандарту з ручною верифікацією), тестування інших сучасних LLM (включаючи потенційно менші, донавчені моделі), підвищення робастності системи до неоднозначностей та помилок у запитах користувачів, а також інтеграцію механізмів оптимізації SQL та посилення безпеки.

~~Загалом, дана дипломна робота успішно демонструє життєздатність та значний потенціал використання великих мовних моделей, таких як Llama 3.1 70B, у поєднанні з мультиагентними підходами для створення ефективних систем перетворення запитів українською мовою на SQL для СУБД PostgreSQL. Це робить важливий внесок у розвиток українського NLP та сприяє демократизації доступу до даних для ширшого кола користувачів.~~