Орієнтовний, першочергова чернетка. Не відповідає дійсності

1. Розділ 1. Порівняльний аналіз існуючих методів …
   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
   8. Постановка задачі
2. Розділ 2. Метод … для української мови

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  
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. Розділ 3. Практичні результати та висновки

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

РОЗДІЛ 1 ПОРІВНЯЛЬНИЙ АНАЛІЗ ІСНУЮЧИХ МЕТОДІВ

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(Рисунок 1) [1]. Цей процес перекладу дозволяє користувачам взаємодіяти з базами даних, використовуючи свою повсякденну мову, фактично усуваючи необхідність у глибоких знаннях синтаксису SQL. Основною метою систем Text-to-SQL є спрощення доступу до **великих обсягів даних (часто в контексті Big Data)**, що зберігаються в реляційних базах даних. Ці системи долають розрив між запитами користувачів природною мовою та структурованою мовою SQL. Ця галузь також визначена в академічному та промисловому контекстах під назвою Natural Language Interface to Databases (NLIDB) [2].

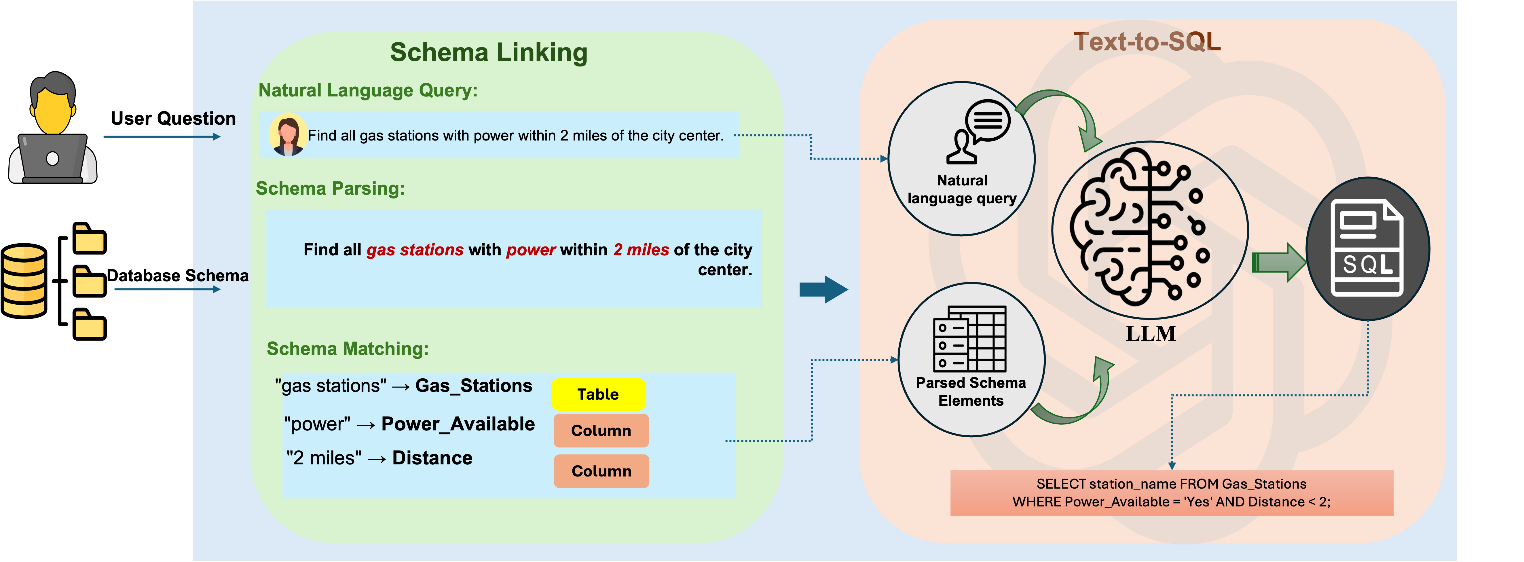


Рисунок 1 – приклад задачі Text-to-SQL [3]

У більш широкому контексті обробки природної мови (NLP, англ. Natural Language Processing - Обро́бка приро́дної мо́ви) і взаємодії з базами даних, Text-to-SQL займає важливе місце. Фактично, 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 досягненням у галузі штучного інтелекту, особливо у сферах генеративного штучного інтелекту та великих мовних моделей (LLM, англ. Large Language Model). Ці досягнення призвели до помітного підвищення точності та зручності систем Text-to-SQL протягом виконання типових академічних та комерційних завдань.

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, технологія досягає значної демократизації доступу до даних.

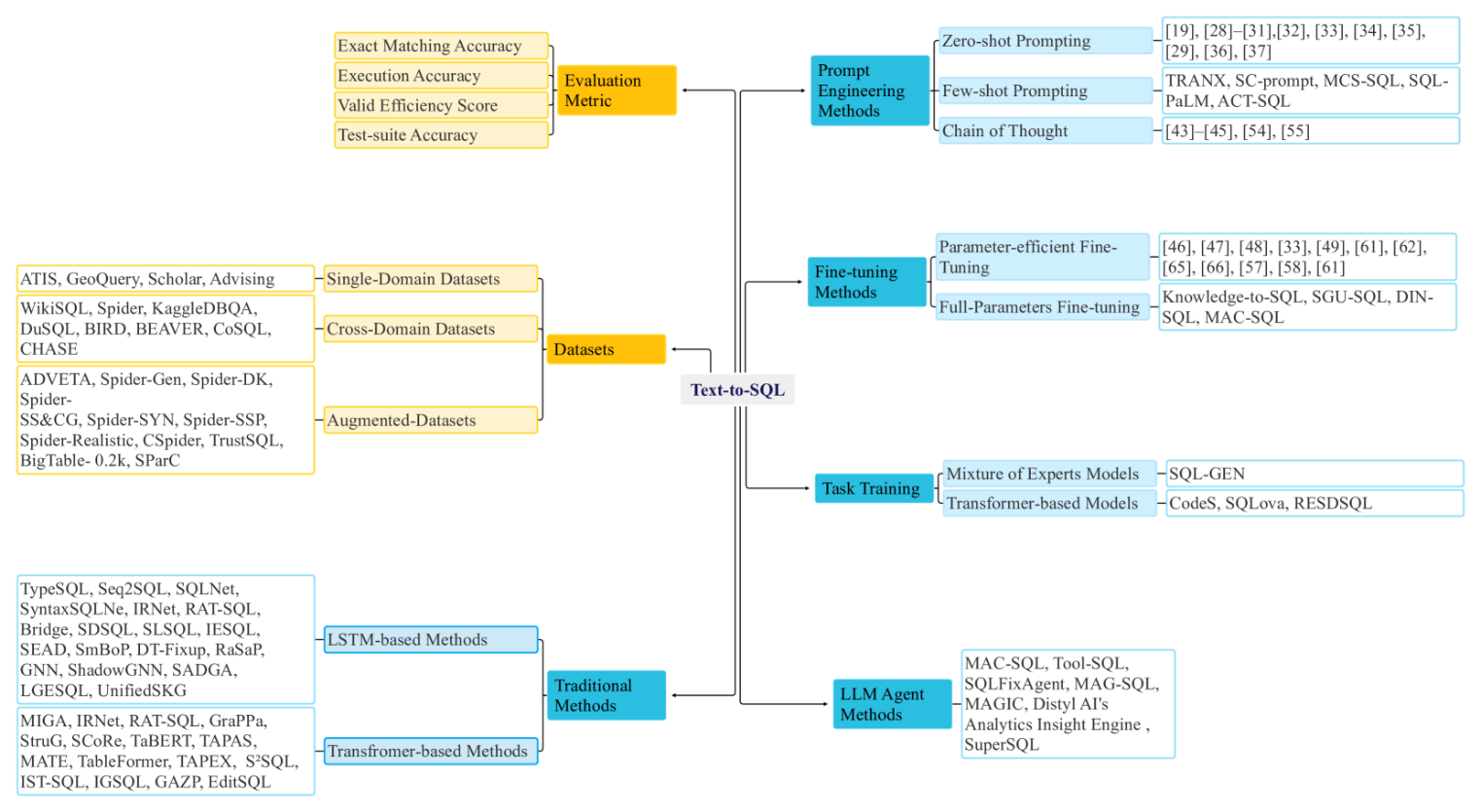


Рисунок 2 -- Огляд метрик, наборів даних та методів перетворення тексту в SQL [4].

* 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 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 Transformer, таких як BERT і GPT [5] , 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 вдосконалення запитів, щоб переконатися, що згенерований SQL-запит є не тільки коректним, але й ефективним у виконанні.

Фактично, 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.

На рисунку 2 представлена комплексна структура галузі **Text-to-SQL**. Вона показує, як оцінюються результати (метрики), на яких даних тренуються моделі, та які існують основні технологічні підходи для перетворення тексту на SQL.

* 1. Проблеми обробки природної мови

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 [6]. 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 існуючих комерційних чи академічних рішеннях. Зазвичай виробничі бази даних , особливо в контексті Big Data, 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 інтенсивних 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», приклад представлено на Рисунку 2 [7]. 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.

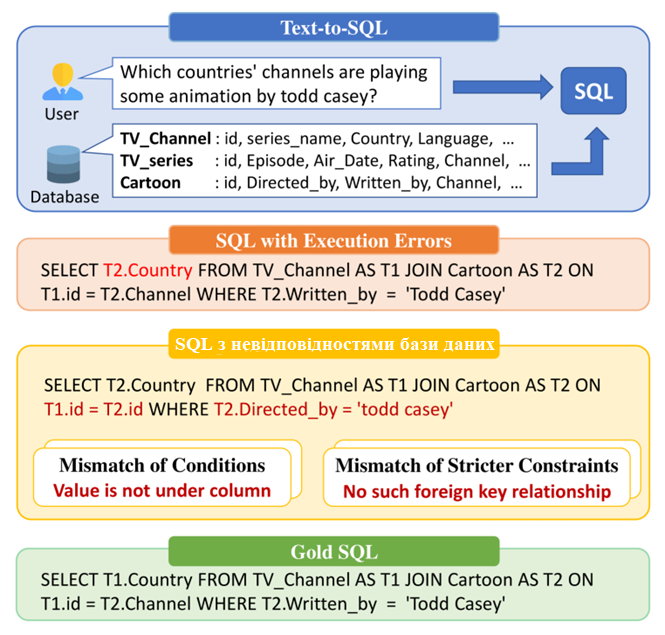


Рисунок 3 – Реальний приклад помилок завдання Text-to-SQL [7]

* 1. Складність SQL-запитів

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 [1].

Основним завданням перекладу з текстового на SQL є зіставлення природної мови з конкретними конструкціями SQL(Рисунок 3) 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.

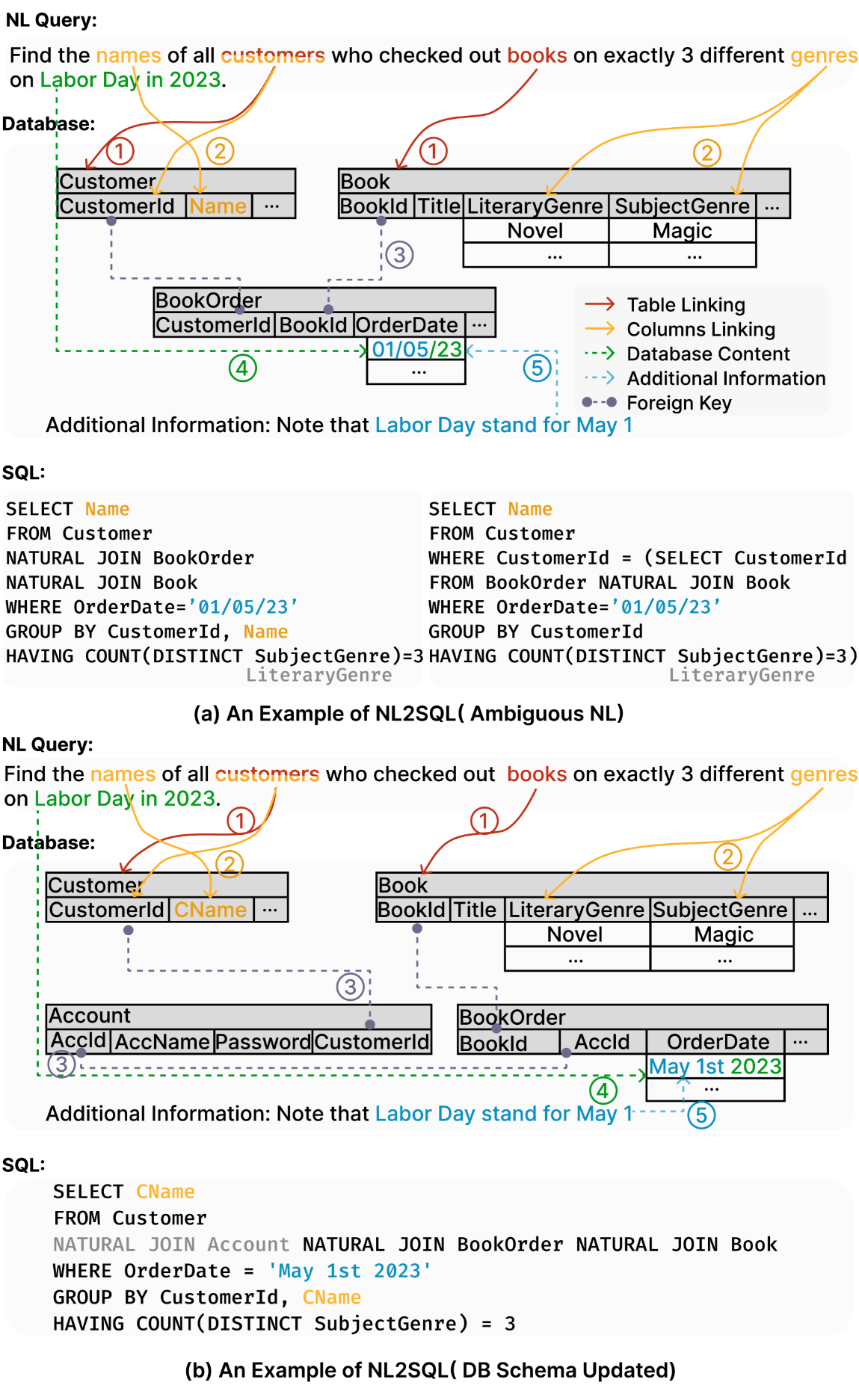


Рисунок 4 – складність співставлення мови з SQL запитом [8]

* 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 [44].

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. (Рисунок 4)

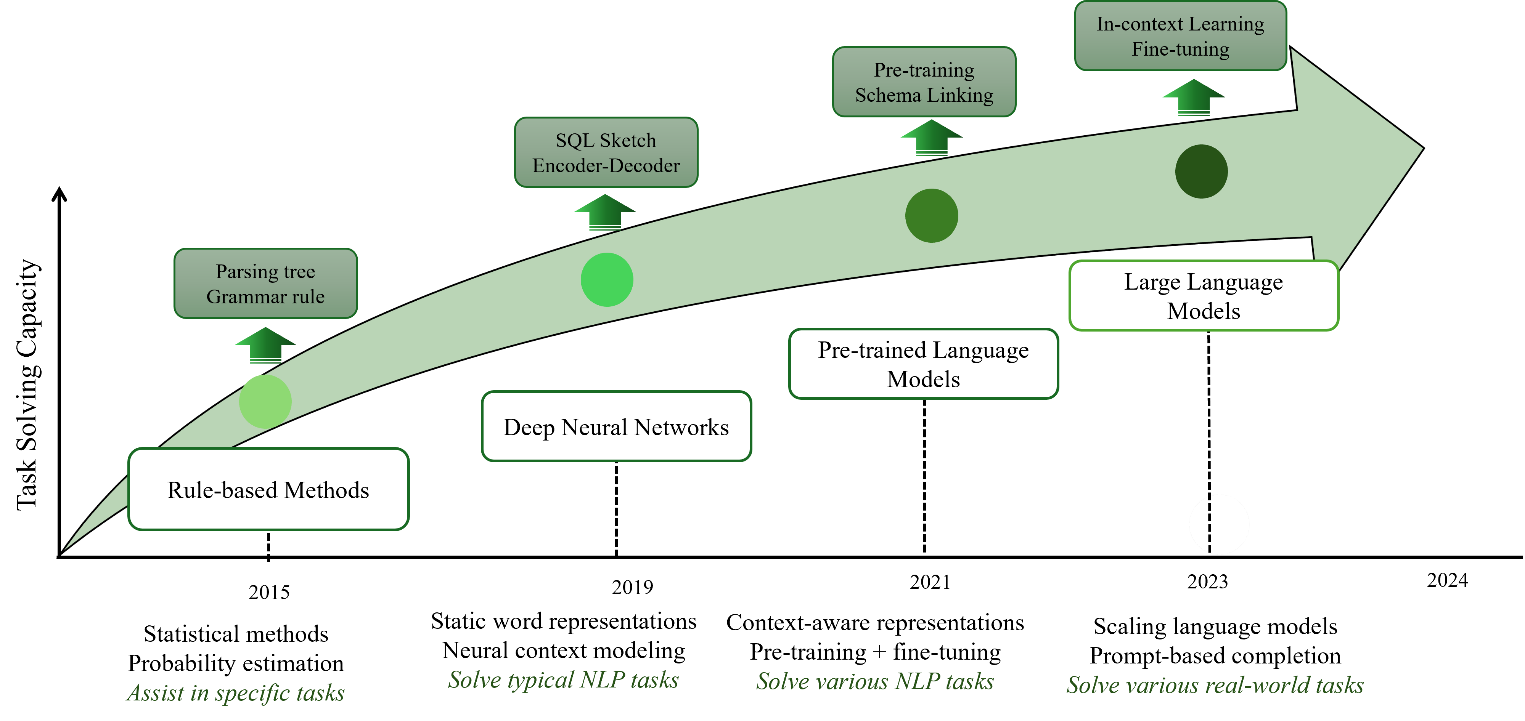


Рисунок 5 – розвиток дослідження задачі Text-to-SQL [3]

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 [9], LUNAR [10], LIFER/LADDER[11] і CHAT-80[12], 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[10] і NaLIX[13], 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) [14], ознаменувало значний стрибок вперед у можливостях цієї галузі. Моделі Sequence-to-sequence (Seq2Seq)[15], що часто використовують архітектуру кодера-декодера, побудовану на основі LSTM, стали популярною основою для безпосереднього перекладу запитів природною мовою в SQL-послідовності. Такі моделі, як TypeSQL[16], Seq2SQL[15], SQLNet[17] та SyntaxSQLNet[18] використовують Bi-LSTM для вивчення семантичного представлення пар question-SQL. 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. Архітектури на основі Transformer, такі як BERT[19], GPT[5] та T5[20], 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[21] та TaBERT[22], 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[23]. 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.

Сучасні методи Text-to-SQL використовують Великі Мовні Моделі (ВММ) за допомогою кількох ключових стратегій: Інженерія промптів (Prompt Engineering) застосовує ретельно розроблені вхідні запити — такі як підходи з кількома прикладами (few-shot, напр., SC-prompt[24], MCS-SQL[25]) або методологію Ланцюжка міркувань (Chain of Thought[26], напр., Chat2Query[27], ACT-SQL[28]) — для скеровування попередньо навчених ВММ без додаткового навчання, пропонуючи швидкість, але потенційно змінну точність. Тонке налаштування (Fine-Tuning) адаптує попередньо навчені ВММ шляхом подальшого навчання, або повнопараметричного (full fine-tuning, напр., DIN-SQL[29], Knowledge-to-SQL[30]) для максимальної точності на конкретному завданні, або використовуючи параметро-ефективні техніки (parameter-efficient, напр., DAIL-SQL[31], StructLM[32], методи, що використовують LoRA[33]/QLoRA[34]) для збалансування продуктивності та витрат ресурсів. Навчання для конкретного завдання (Task-Training) передбачає розробку моделей спеціально для генерації SQL, таких як Суміш експертів (Mixture-of-Experts[35], SQL-GEN[36]) або власні архітектури на основі Трансформерів (CodeS[37], RESDSQL[38]), часто глибоко навчені на даних SQL. Нарешті, новітній підхід LLM-агентів використовує фреймворки, такі як SQLFixAgent[39], MAC-SQL[40], Tool-SQL[7], та SuperSQL[41], де спеціалізовані ВММ або ВММ, оснащені зовнішніми інструментами, співпрацюють для декомпозиції складних запитів, отримання інформації, виявлення помилок та уточнення кінцевого SQL-запиту для підвищеної надійності та точності.

Таблиця 1 – порівняння підходів вирішення Text-to-SQL

|  |  |  |  |
| --- | --- | --- | --- |
| **Ера / Підхід** | **Ключові Технології / Концепції** | **Приклади Моделей / Методів** | **Основні Характеристики / Внесок** |
| **Раннє Глибоке Навчання** | RNN (Рекурентні Нейронні Мережі), LSTM, Seq2Seq, Кодер-декодер, Увага (Attention) | TypeSQL[16], Seq2SQL[15], SQLNet[17], SyntaxSQLNet[18] (використовують Bi-LSTM) | Значний стрибок у можливостях; прямий переклад запитів на SQL; використання Bi-LSTM для семантики; увага для фокусування. |
| **Ера Трансформерів та ПНММ** | Трансформери (Transformers), Попередньо Навчені Мовні Моделі (ПНММ), Самоувага (Self-Attention), Pre-training & Fine-tuning | BERT[19], GPT[5], T5[20], RAT-SQL[21], TaBERT[22] | Домінуючий підхід; виняткове розуміння контексту (запит + схема); парадигма "pre-train, fine-tune"; інтеграція схеми БД. |
| **Підхід з Графовими Нейронними Мережами (ГНМ)** | Графові Нейронні Мережі (GNNs) | Загальне застосування [23], RAT-SQL[21] (використовує *relation-aware transformer*) | Ефективне кодування структури та зв'язків схеми БД; покращене розуміння контексту бази даних. |
| **Сучасні Методи (з Великими Мовними Моделями - ВММ)** |  |  | Використання ВММ через різні стратегії для підвищення точності та надійності. |
| Інженерія Промптів (Prompt Engineering) | Few-shot, Chain of Thought (CoT) | Few-shot: SC-prompt[24], MCS-SQL[25]. <br> CoT: Chat2Query[27], ACT-SQL[28] (заг. [26]) | Скеровування ВММ без дод. навчання; швидкість; потенційно змінна точність. |
| Доналаштування (Fine-Tuning) | Повнопараметричне (Full), Параметро-ефективне (PEFT, LoRA[33], QLoRA[34]) | Повне: DIN-SQL[29], Knowledge-to-SQL[30]. <br> Ефективне: DAIL-SQL[31], StructLM[32] | Адаптація ВММ до задачі; максимальна точність (повне) vs баланс продуктивності/ресурсів (ефективне). |
| Навчання для Конкретного Завдання (Task-Training) | Суміш Експертів (Mixture-of-Experts - MoE), Спеціалізовані архітектури | MoE: SQL-GEN[36] (заг. [35]). <br> Трансформери: CodeS[37], RESDSQL[38] | Розробка моделей спеціально для генерації SQL; глибоке навчання на даних SQL. |
| LLM-агенти | Фреймворки співпраці, Зовнішні інструменти | SQLFixAgent[39], MAC-SQL[40], Tool-SQL[7], SuperSQL[41] | Декомпозиція запитів, отримання інформації, виявлення помилок, уточнення SQL; підвищена надійність та точність. |

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 Transformer-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 та метрик, що використовуються для оцінки продуктивності. Ранні дослідження часто спиралися на однодоменні набори даних, такі як ATIS[42] (для інформації про авіаперельоти) та GeoQuery[43] (для географічних запитів), які забезпечували сфокусований контекст для оцінки початкових підходів.

Значним поворотним моментом у цій галузі стала поява міждоменних наборів данних, метою яких було оцінити здатність моделей до узагальнення на різних схемах баз даних та типах запитів. WikiSQL[15] 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[44] — 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[45] та CoSQL[46], 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[47], SEDE[48] та EHRSQL[49], які мають на меті представити більш реалістичні сценарії, а також набори даних, такі як Spider-DK[50] Spider-Syn[51] та Spider-Realistic[52], 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.

Продуктивність систем Text-to-SQL зазвичай оцінюється за допомогою кількох ключових метрик. Точність виконання (Execution Accuracy, EX) [44] – це критично важлива метрика, яка вимірює, чи згенерований SQL-запит при виконанні в базі даних повертає правильну відповідь. Точна відповідність (Exact Match, EM) [44] – ще одна поширена метрика, яка перевіряє, чи згенерований SQL-запит точно збігається (посимвольно) з еталонним (анотованим людиною) SQL-запитом

Незважаючи на прогрес, досягнутий завдяки цим наборам даних та метрикам, зростає усвідомлення їхніх обмежень. Існуючі набори даних 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 наборів даних та метрик оцінки, які б точніше відображали виклики реальних завдань Text-to-SQL та справжні наміри користувацьких запитів.

Корпус BIRD (BIg Bench for LaRge-Scale Database Grounded in Text-to-SQLs)[53] 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 корпусу запропонували нову метрику оцінки — Valid Efficiency Score (VES) — для оцінки ефективності згенерованих SQL-запитів. За їхніми твердженнями, BIRD є першим корпусом 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 наборів даних спонукав дослідників розробляти моделі, здатні обробляти більш реалістичні та складні сценарії. Однак поточна дискусія щодо обмежень існуючих наборів даних та метрик оцінки підкреслює необхідність постійного вдосконалення способів оцінки продуктивності та реальної застосовності систем Text-to-SQL.

Таблиця 2 - Огляд Еталонних Наборів Даних та метрик для Text-to-SQL

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Набір Даних (Dataset)** | **Тип Діалогу** | **Охоплення Доменів** | **Мова** | **# Прикладів** | **# Баз Даних** | **# Таблиць (прибл.)** | **# Доменів** | **Ключові Метрики** | **Примітки / Фокус** |
| GenQuery | Одноходовий | Однодоменний | en | 880 | 1 | 6 | 1 | EX, EM | Ранній, однодоменний |
| Scholar | Одноходовий | Однодоменний | en | 817 | 1 | 7 | 1 | EX, EM | Ранній, однодоменний (академічні публікації) |
| GeoQuery [43] | Одноходовий | Однодоменний | en | ~500+ | 1 | ~8 | 1 | EX, EM | Класичний ранній, однодоменний (географія США) |
| ATIS [42] | Багатоходовий | Однодоменний | en | 5,418 | 1 | 27 | 1 | EX, EM¹ | Класичний ранній, однодоменний (авіаперельоти), контекст діалогу |
| WikiSQL [15] | Одноходовий | Крос-доменний | en | 80,654 | 26,521 | 1 на БД | N/A | EX, EM | Перший великомасштабний крос-доменний, відносно прості запити |
| Spider [44] | Одноходовий | Крос-доменний | en | 10,181 | 200 | 1020 (Ø 5.1/БД) | 138 | **EX, EM** | Складний крос-доменний (joins, nested queries), стандарт де-факто |
| Spider-SYN [51] | Одноходовий | Крос-доменний | en | 7,990 | 166 | 876 | ~138 | EX, EM | Варіант Spider для перевірки стійкості до синонімів |
| Spider-DK [50] | Одноходовий | Крос-доменний | en | 535 | 10 | 48 | ~138 | EX, EM | Варіант Spider для перевірки стійкості до знань предметної області |
| Spider-Realistic [52] | Одноходовий | Крос-доменний | en | - | ~200 | ~1020 | ~138 | EX, EM | Варіант Spider (відсутність явних назв стовпців у запитаннях) |
| Spider-SSP | Одноходовий | Крос-доменний | en | - | - | - | ~138 | EX, EM | Варіант Spider |
| CSpider | Одноходовий | Крос-доменний | zh | 10,181 | 200 | 1020 | 138 | EX, EM | Китайська версія Spider |
| SQUALL | Одноходовий | Крос-доменний | en | 15,620 | 2,108 | 2108 | N/A | EX, EM | Ймовірно, на основі WikiTableQuestions |
| DuSQL | Одноходовий | Крос-доменний | zh | 23,797 | 200 | 820 | N/A | EX, EM | Китайський крос-доменний |
| SParC [45] | Багатоходовий | Крос-доменний | en | 4,298 (int) | 200 | 1020 | 138 | EX, EM¹ | Багатоходові діалоги на основі Spider (контекстуальна складність) |
| CoSQL [46] | Багатоходовий | Крос-доменний | en | 3,007 (dlg) | 200 | 1020 | 138 | EX, EM¹ | Багатоходові діалоги на основі Spider (взаємодія користувача) |
| CHASE | Багатоходовий | Крос-доменний | zh | 5,489 | 280 | 1280 | N/A | EX, EM¹ | Китайський багатоходовий крос-доменний |
| KaggleDBQA [47] | Одноходовий | Крос-доменний | en | 272 | 8 | Ø 2.3/БД | 8 | EX, EM | На основі Kaggle, реалістичні сценарії, вимагає зовнішніх знань (Knowledge=✓) |
| SEDE [48] | Одноходовий | Крос-доменний | en | - | ~25 | - | ~9 | EX, EM | Stack Exchange Data Explorer, реалістичні сценарії |
| EHRSQL [49] | Одноходовий | Однодоменний | en | - | 1 | - | 1 | EX, EM | Реалістичний сценарій (Електронні Медичні Картки - EHR) |
| BIRD [53] | Одноходовий | Крос-доменний | en | 12,751 | 95 | Ø 7.3/БД (великі) | 37 | **EX, EM, VES** | **Великомасштабний**, реалістичний, **враховує ефективність SQL (VES)** |

* 1. Сучасний стан досліджень

Сильні сторони цих сучасних передових моделей, особливо тих, що базуються на LLM(ВММ), включають їхню здатність до перенесення знань (transferability), що дозволяє їм легше адаптуватися до нових завдань та предметної області. 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) [26], де модель явно генерує проміжні кроки міркувань, та декомпозиція завдань, де складні запити розбиваються на менші, атомарні 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 нерелевантного 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.

Адаптація та застосування технологій Text-to-SQL до української мови становлять окремий набір викликів. Українська мова характеризується багатою морфологією (відмінки, роди, дієвідміни) та відносно вільним порядком слів, що значно ускладнює однозначне зіставлення природномовних запитів зі структурою бази даних та командами SQL. Крім того, існує суттєвий дефіцит великих, високоякісних та анотованих наборів даних (пар "питання-SQL" та схем БД) українською мовою, подібних до англомовних еталонів, що обмежує можливості тренування та надійної оцінки моделей. Недостатня кількість спеціалізованих лінгвістичних інструментів та попередньо навчених моделей, оптимізованих саме для української мови та задачі Text-to-SQL, також є вагомою перешкодою. Ці фактори вимагають розробки специфічних підходів та адаптації існуючих рішень.

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, є необхідною умовою для ширшого впровадження та підвищення впливу технології Text-to-SQL.

Аналіз сучасних підходів до Text-to-SQL, особливо тих, що займають провідні позиції на корпусах на кшталт BIRD, показує значну поширеність рішень, які покладаються на донавчання (fine-tuning) великих мовних моделей під конкретну задачу [31, 30, 29]. Хоча цей підхід може забезпечити високу точність, він пов'язаний зі значними обчислювальними витратами, потребою у великих специфічних датасетах для донавчання та складністю відтворення, що робить його менш доступним для багатьох дослідницьких та практичних сценаріїв. Альтернативою є використання попередньо навчених LLM через інженерію промптів у рамках певного фреймворку. Однак, знайти дійсно ефективний та, що важливо, доступний і зрозумілий фреймворк, який покладається переважно на промптування, є нетривіальною задачею. Багато агентних підходів можуть містити закриті компоненти або мати складну реалізацію. У цьому контексті, мультиагентний фреймворк MAC-SQL [40] виділяється як привабливий варіант саме тому, що він розроблений для роботи з LLM через промпти, не вимагаючи дороговартісного донавчання базової моделі для досягнення високої ефективності. Його модульна архітектура (агенти Selector, Decomposer, Refiner), що логічно відповідає загальним етапам вирішення задачі (підготовка/фільтрація схеми, декомпозиція/генерація, оптимізація), пропонує відносну простоту реалізації та прозорість логіки. Це розбиття складного процесу на керовані підзадачі створює кращий потенціал для адаптації до специфіки української мови та системи PostgreSQL шляхом модифікації агентної логіки та інструкцій, а не перенавчання моделі. Саме поєднання цієї архітектурної гнучкості, відсутності необхідності у донавчанні та доступності коду зробило MAC-SQL найбільш практичним та обґрунтованим вибором для досягнення цілей даної дипломної роботи, що включають адаптацію до нового мовного та технологічного середовища в умовах обмежених ресурсів для донавчання.

1.9 Постановка задачі

Метою даної дипломної роботи є адаптація для відкритих мовних моделей та українськомовних систем та оцінка існуючого мультиагентного фреймворку **MAC-SQL** для роботи в новому технологічному та мовному середовищі.

Ключові завдання роботи включають:

1. **Заміна Базової Моделі та API:** Модифікувати фреймворк для використання великої мовної моделі **Llama[57] 3.1 70B** замість GPT-4[58], інтегрувавши виклики через API платформи **Together AI** замість OpenAI.
2. **Підтримка PostgreSQL:** Адаптувати компоненти фреймворку (зокрема, агенти Selector та Refiner, утиліти роботи з БД) для коректної взаємодії з системою управління базами даних **PostgreSQL**, включаючи отримання метаданих схеми та виконання згенерованих SQL-запитів.
3. **Розробка Українського Датасету:** Створити новий тестовий набір даних **«BIRD-UKR»**, який включає:
   * Схеми баз даних для PostgreSQL з **назвами таблиць та стовпців українською мовою**.
   * Наповнення баз даних реалістичними (хоча й синтетичними) **україномовними даними**.
   * Пари «**питання українською мовою** – еталонний **SQL-запит** для PostgreSQL».
4. **Тестування та Оцінка:** Провести експериментальну оцінку ефективності адаптованого фреймворку MAC-SQL (з Llama 3.1 та підтримкою PostgreSQL) на:
   * Стандартних англомовних корпусах (Spider, BIRD-minidev на SQLite) для перевірки збереження базової функціональності та порівняння.
   * Новоствореному україномовному набору данних **BIRD-UKR** для оцінки здатності системи працювати з українською мовою та PostgreSQL.

Виконання цих завдань дозволить визначити ефективність та практичну придатність адаптованого рішення для взаємодії з базами даних українською мовою.

РОЗДІЛ 2 МЕТОД ЗАСТОСУВАННЯ ТЕХНОЛОГІЇ TEXT2SQL ДЛЯ УКРАЇНСЬКОЇ МОВИ

2.1 Огляд структури MAC-SQL[40]

MAC-SQL був розроблений для подолання обмежень шляхом застосування мультиагентного підходу [40, с. 1, 3]. Він складається з трьох спеціалізованих агентів, які взаємодіють для генерації та уточнення SQL запитів [40 с. 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 [40, с. 3, розділ 3.2].

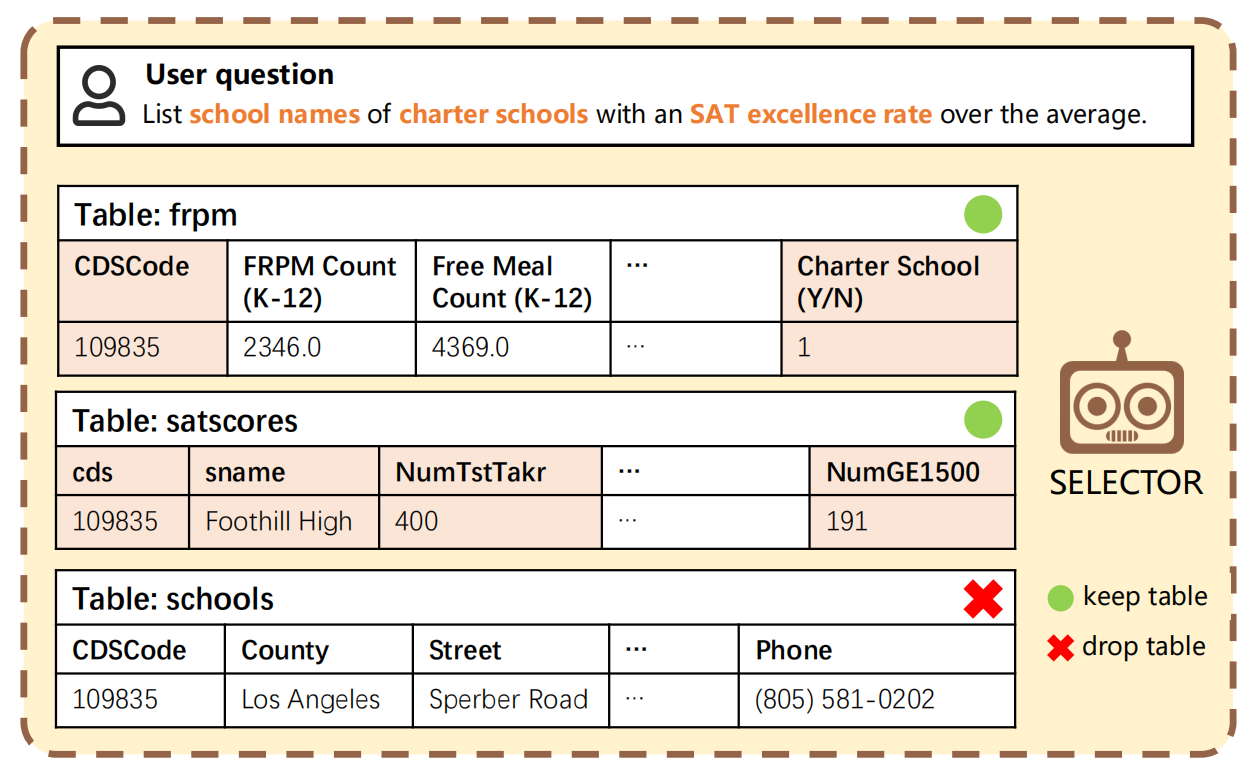


Рисунок 6 – приклад роботи агента Selector

1. **Агент Decomposer:** Використовує підхід "ланцюжка думок" (chain-of-thought) для декомпозиції складного запиту користувача на послідовність простіших підзапитів. Для кожного підзапиту генерується відповідний запит SQL. Цей підхід полегшує моделі обробку запитів, що вимагають складних логічних міркувань [40, с. 3, розділ 3.3; с. 4, Рис. 3].

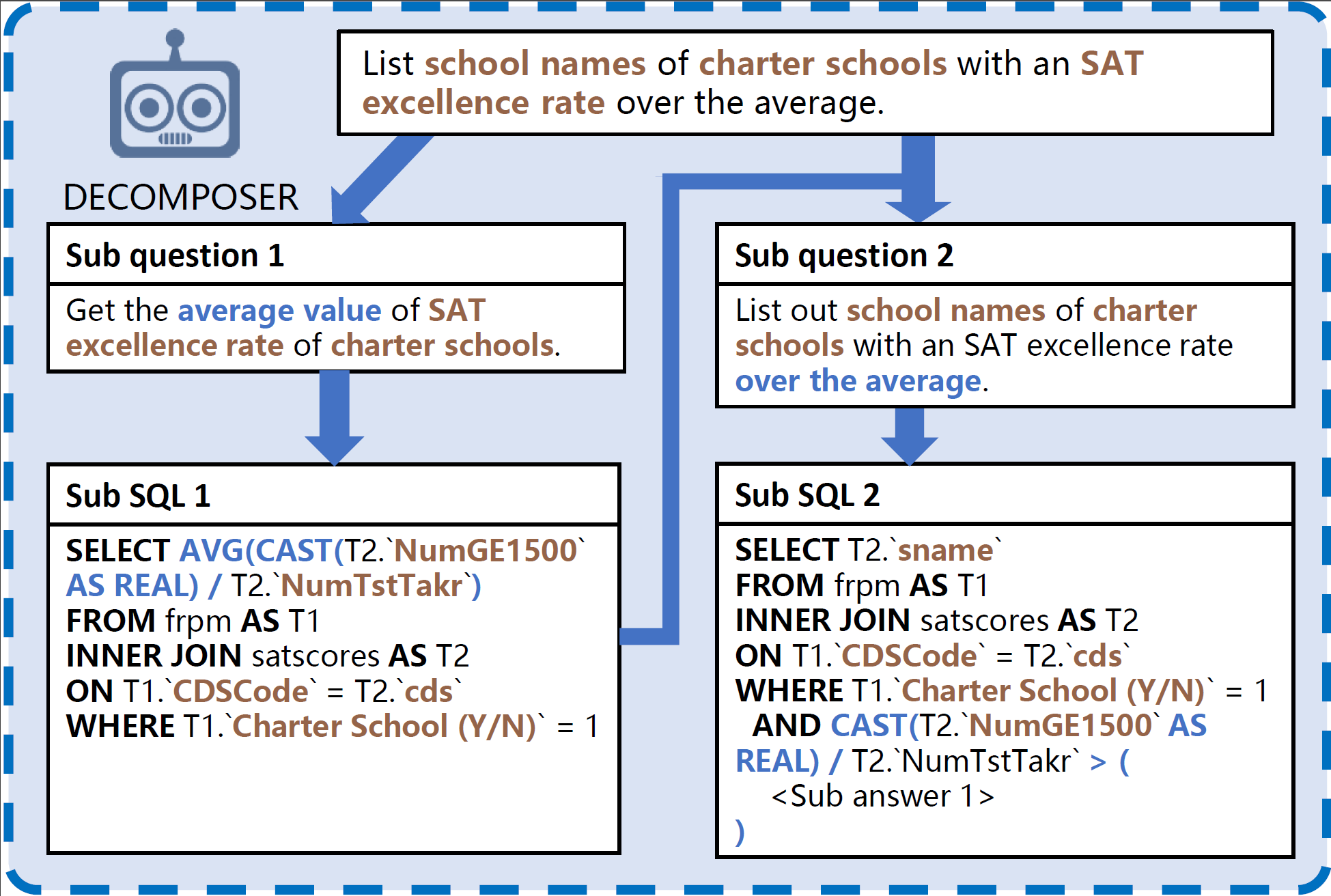


Рисунок 7 – приклад роботи агента Decomposer

1. **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 [40, с. 4, розділ 3.4; стор. 4, Рис. 4].

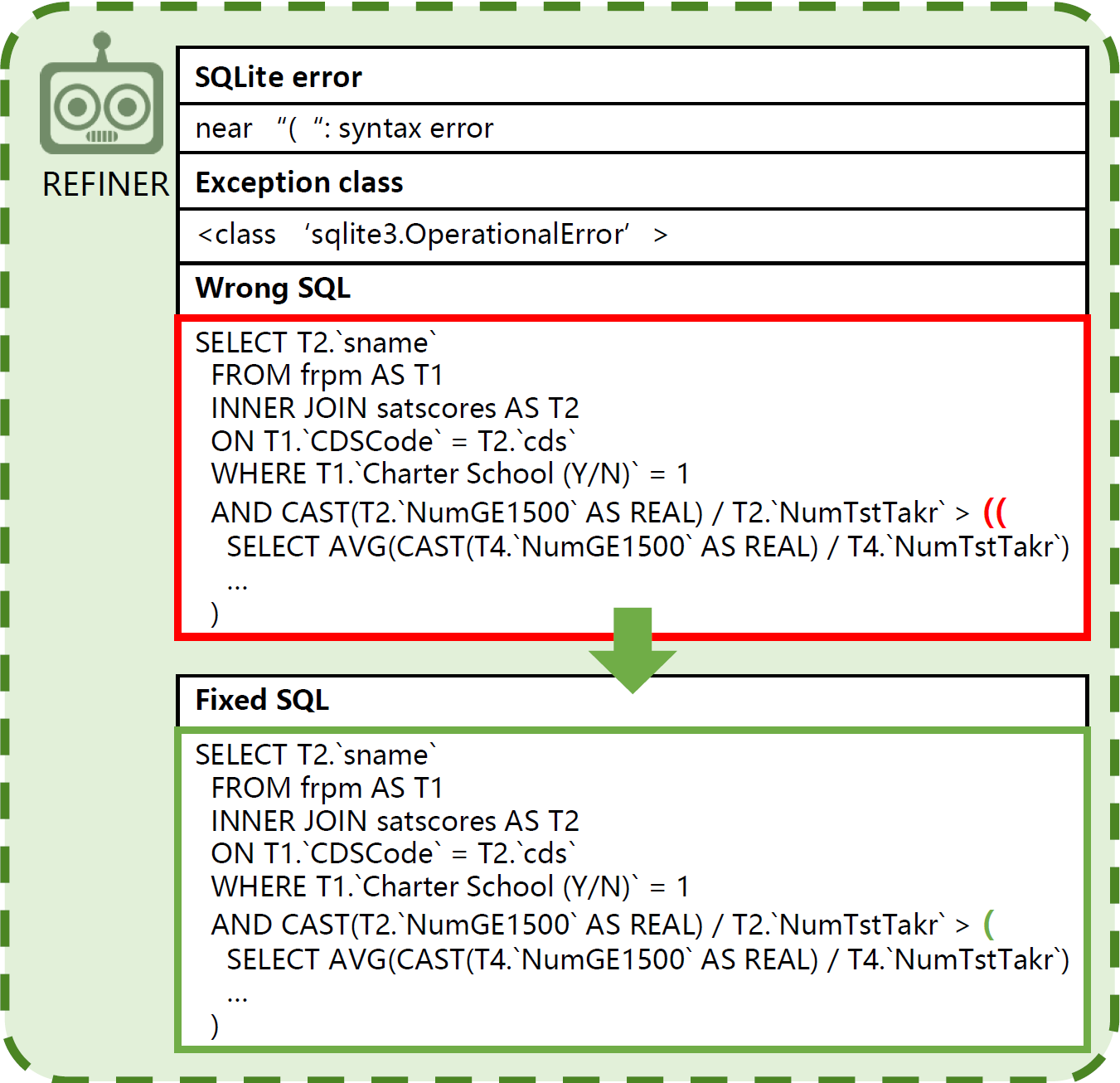


Рисунок 8 – приклад роботи агента Refiner

**Ефективність**

Ефективність архітектури MAC-SQL підтверджується результатами експериментів, представленими в роботі Xu et al [40]. Комбінація MAC-SQL з моделлю GPT-4 досягла точності виконання (Execution Accuracy) 59.59% на тестовому наборі корпусі BIRD, що на момент публікації дослідження було передовим результатом (state-of-the-art) [40, с. 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 [40, с. 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 Вибір Базової Мовної Моделі: Meta Llama 3.1 70B

В оригінальному дослідженні фреймворку MAC-SQL[40] 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 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 [40, с. 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 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 PostgreSQL

Вибір середовища для виконання та тестування згенерованих SQL запитів є важливим аспектом оцінки систем Text-to-SQL. Хоча SQLite часто використовується в академічних наборах даних (наприклад, для перевірки виконання в деяких частинах Spider[44] або BIRD[53]) 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 для баз даних, що потребують масштабування.
   * SQLite часто має менш жорсткі вимоги до 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 виконання 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[53] (minidev) та Spider[44]

Для всебічної оцінки ефективності фреймворку MAC-SQL[40] на основі Llama 3.1 70B та перевірки його здатності генерувати коректні SQL запити для PostgreSQL, було обрано два ключових корпуси: Spider[44] та піднабір BIRD[53] (minidev). Вибір цих наборів даних дозволяє оцінити різні аспекти продуктивності системи Text-to-SQL.

1. Spider:
   * Призначення: Spider[44] є широко визнаним стандартним корпусом 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 піднабір):
   * Призначення: корпус BIRD[53] 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

1. Мотивація та Необхідність

Стандартні набори даних 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 наборів даних не дозволяє оцінити здатність моделі коректно інтерпретувати лінгвістичні особливості, термінологію та структуру запитів українською мовою в контексті взаємодії з базами даних.

2. Процес Створення штучного набору даних автоматизованим формуванням

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) [54]. Для генерації пар запит-SQL було обрано Gemini 2.5 PRO[55]. Цей вибір обґрунтований значним фокусом компанії Google на розвитку можливостей своїх моделей у сфері перетворення тексту в SQL (Text-to-SQL) [56 ], а також спостереженнями щодо високої ефективності Gemini у вирішенні подібних завдань, що відзначається у професійній спільноті.

Процес включав наступні кроки:

1. Аналіз BIRD: Було вивчено структуру та характеристики корпусу 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" стандарт (silver standard dataset). На відміну від "золотого стандарту" (gold standard), який зазвичай передбачає ретельну ручну анотацію або валідацію експертами та слугує еталонною "наземною правдою" (ground truth), "срібний" набір даних генерується автоматизованими або напівавтоматизованими методами. У нашому випадку, дані були згенеровані штучно з використанням великих мовних моделей та проходили обмежену/спрощену валідацію. Важливо зазначити, що хоча такий підхід дозволяє швидко отримати значний обсяг даних для тестування, отриманий набір може потенційно містити певні артефакти автоматичної генерації (наприклад, неточності перекладу, менш природні формулювання запитів порівняно з органічно створеними людиною, можливі помилки в еталонних SQL). Тому результати, отримані на "срібному" датасеті, слід інтерпретувати з урахуванням цих потенційних обмежень, розглядаючи їх як корисний індикатор продуктивності системи, але не як абсолютну міру якості.

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.

2.6 Розробка прототипу MAC-SQL

2.6.1 Загальний огляд структури проекту

```

/

├── core/                          # Основні компоненти системи

│   ├── agents.py                  # Базові класи для всіх агентів

│   ├── api.py                     # Функції для взаємодії з LLM API

│   ├── api\_config.py              # Конфігурація API (ключі, URL, параметри)

│   ├── bird\_extensions.py         # Розширення для датасету BIRD

│   ├── bird\_ukr\_extensions.py     # Розширення для українського BIRD

│   ├── chat\_manager.py            # Базовий менеджер взаємодії між агентами

│   ├── const.py                   # Константи та системні промпти

│   ├── const\_ukr.py               # Українські константи та промпти

│   ├── db\_utils.py                # Утиліти для роботи з базами даних

│   ├── enhanced\_chat\_manager.py   # Розширений менеджер чату

│   ├── enhanced\_chat\_manager\_pg.py# Менеджер для PostgreSQL

│   ├── llm.py                     # Основний інтерфейс для LLM

│   ├── macsql\_together\_adapter.py # Адаптер для Together AI

│   ├── spider\_extensions.py       # Розширення для датасету Spider

│   └── utils.py                   # Загальні утиліти

│

├── data/                          # Датасети та бази даних

│   ├── bird/                      # Датасет BIRD

│   ├── bird-ukr/                  # Український BIRD датасет

│   └── spider/                    # Датасет Spider

│

├── utils/                         # Додаткові утиліти

│   ├── bird\_ukr\_loader.py

│   ├── bird\_ukr\_tables\_adapter.py

│   ├── pg\_connection.py           # Пул з'єднань для PostgreSQL

│   └── pg\_selector.py             # Реалізація селектора для PostgreSQL

│

├── logs/                          # Директорія для логів

│   ├── api\_trace.json             # Трейси API викликів

│   └── debug/                     # Логи відлагодження

│

├── scripts/                       # Скрипти для запуску та тестування

│   ├── run\_bird.py                # Запуск тестування на BIRD

│   ├── run\_bird\_ukr.py            # Запуск тестування на українському BIRD

│   └── run\_spider.py              # Запуск тестування на Spider

│  ├── test\_macsql\_agent\_bird\_ukr.py #

├── .env                           # Змінні середовища

├── requirements.txt               # Залежності проекту

└── README.md                      # Документація проекту

```

Детальний опис ключових директорій і файлів

core/ - Основні компоненти системи

* **agents.py**: Містить базові абстрактні класи для всіх агентів системи
  + `BaseAgent` - абстрактний базовий клас з методом `talk()`
  + `Selector` - базова реалізація селектора схеми бази даних
  + `Decomposer` - агент для розбиття задачі та генерації SQL
  + `Refiner` - агент для виконання і виправлення SQL-запитів
* **chat\_manager.py**: Реалізує базовий менеджер взаємодії між агентами
  + Клас `ChatManager` - координує потік даних між агентами
  + Метод `\_chat\_single\_round()` - один цикл обміну повідомленнями
  + Метод `start()` - ініціює обробку запиту
* **enhanced\_chat\_manager.py**: Розширений менеджер з підтримкою різних датасетів
  + Клас `EnhancedChatManager` - розширює базовий `ChatManager`
  + Метод `\_create\_agents()` - створює агентів відповідно до типу датасету
  + Методи для покращеного логування та дебагу
* **enhanced\_chat\_manager\_pg.py**: Спеціалізований менеджер для PostgreSQL
  + Клас `PgEnhancedChatManager` - адаптований для PostgreSQL
  + Методи для роботи з PostgreSQL з'єднаннями та схемою
  + Інтерфейс для українського датасету
* **llm.py**: Основний інтерфейс для роботи з мовними моделями
  + Функція `api\_func()` - базовий виклик API
  + Функція `safe\_call\_llm()` - обгортка з обробкою помилок
  + Глобальні змінні для конфігурації та логування
* **api.py**: Функції для взаємодії з різними LLM API
  + Функція `together\_api\_call()` - взаємодія з Together AI
  + Функція `safe\_call\_llm()` - з обробкою помилок
  + Функція `call\_llm()` - універсальний інтерфейс для різних моделей
* **macsql\_together\_adapter.py**: Адаптер для Together AI
  + Клас `TogetherAIAdapter` - інтеграція з Together AI
  + Методи для форматування запитів і обробки відповідей
  + Механізми обробки помилок та rate limiting
* **db\_utils.py**: Утиліти для роботи з базами даних
  + Функція `get\_db\_connection()` - універсальний інтерфейс підключення
  + Функція `get\_schema()` - отримання схеми з різних СКБД
  + Функція `execute\_query()` - виконання запитів
* **bird\_extensions.py, spider\_extensions.py, bird\_ukr\_extensions.py**:
  + - Розширені версії агентів для конкретних датасетів
  + - Специфічна логіка форматування схеми для різних СКБД
  + - Адаптовані методи виконання SQL
* - **const.py та const\_ukr.py**:
  + - Системні промпти для агентів
  + - Константи для налаштування поведінки системи
  + - Шаблони для взаємодії з LLM

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

Першим кроком адаптації була заміна мовної моделі GPT-4 на meta-llama/Llama-3.1-70B-Instruct-Turbo (Llama 3.1 70B), доступ до якої здійснювався через платформу Together AI. Це потребувало таких змін:

* Модифікація викликів API: Замість використання бібліотеки OpenAI, була реалізована логіка для взаємодії з REST API платформи Together AI. Це включало формування HTTP-запитів з відповідними заголовками авторизації та параметрами моделі в тілі запиту. Основні зміни торкнулися модулів core/api.py(код наведений в додатку А) та core/llm.py(додаток Б), де з'явилися функції для роботи саме з Together AI (наприклад, умовна together\_api\_call).
* Оновлення конфігурації: Файл core/api\_config.py був доповнений для зчитування налаштувань Together AI (API-ключ, назва моделі) з .env файлу або змінних середовища. Було додано прапорець (USE\_TOGETHER\_AI), який дозволяє системі динамічно обирати, яке API використовувати – Together AI чи резервне (наприклад, OpenAI).
* Обробка помилок та обмежень API: Було реалізовано механізм повторних спроб (retry logic) та експоненційної затримки (exponential backoff) у функції safe\_call\_llm або безпосередньо у функції виклику API Together AI. Це дозволяє системі автоматично обробляти тимчасові проблеми з мережею або досягнення лімітів на кількість запитів до API (status code 429), підвищуючи стабільність роботи.

Адаптація промптів для нової моделі не знадобилася, оскільки Llama 3.1 виявилася сумісною з існуючими інструкціями для агентів.

2.6.2 Адаптація для PostgreSQL та українського набору даних (BIRD-UKR)

Ключовою модифікацією для досягнення мети роботи стала підтримка СКБД PostgreSQL та україномовного корпусу BIRD-UKR. На відміну від стандартних наборів даних MAC-SQL (Spider, BIRD), які використовують SQLite та англійські схеми, BIRD-UKR вимагав роботи з іншою СКБД та українськими ідентифікаторами.

Робота з базою даних PostgreSQL:

* Підключення: Модуль core/db\_utils.py(додаток В) був розширений. Функція get\_db\_connection тепер аналізує назву набору даних (dataset\_name). Для bird-ukr вона використовує бібліотеку psycopg2 для підключення до PostgreSQL, використовуючи облікові дані з конфігурації (.env файл: PG\_HOST, PG\_PORT, PG\_USER, PG\_PASSWORD). Для інших наборів даних зберігається логіка підключення до SQLite.
* Отримання схеми: Спеціалізований PostgreSQLSelector (реалізований, наприклад, в utils/pg\_selector.py(додаток Г) та інтегрований через core/bird\_ukr\_extensions.py)(додаток Д) використовує запити до системних таблиць information\_schema PostgreSQL замість PRAGMA (SQLite) для отримання метаданих (таблиці, стовпці, типи, ключі). Важливою особливістю є отримання кількох прикладів реальних значень (sample\_values) для кожного стовпця, що надає LLM більше контексту.
* Виконання SQL та перевірка: Агент PostgreSQLRefiner (реалізований у core/bird\_ukr\_extensions.py) використовує psycopg2 для виконання згенерованих SQL-запитів безпосередньо на PostgreSQL. Він також адаптований для обробки специфічних помилок PostgreSQL.

Підтримка української мови:

* Адаптація промптів: Файл core/const\_ukr.py(додаток Е) містить спеціально розроблені шаблони промптів (selector\_template\_ukr, decomposer\_template\_ukr, refiner\_template\_ukr). Ці промпти інструктують LLM:
  + Очікувати українські назви в схемі бази даних.
  + Генерувати SQL з урахуванням синтаксису PostgreSQL (використання TRUE/FALSE, правила цитування українських ідентифікаторів, якщо вони містять спеціальні символи або є зарезервованими словами тощо).
  + Не перекладати українські назви таблиць та стовпців англійською мовою при генерації SQL.
* Форматування схеми: Схема бази даних, що передається LLM, тепер містить українські ідентифікатори та приклади українських даних, отримані PostgreSQLSelector.

Ці зміни дозволили системі коректно інтерпретувати запити користувача щодо українських баз даних та генерувати працездатні SQL-запити для PostgreSQL..

2.6.3 Про форматування схеми:

* **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.

2.6.4 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.

2.7 Розробка україномовного набору даних

Для оцінки та вдосконалення систем Text2SQL необхідні якісні та репрезентативні набори даних. У той час як для англійської мови існують розвинені ресурси, такі як Spider та BIRD, аналогічні інструменти для української мови практично відсутні. Англомовний корпус 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 набору даних 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 набір даних 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 СКБД 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.

2.7.1 Проектування

Основою будь-якого NL2SQL набору даних 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 набір даних, 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 9 – фрагмент коду створення схеми однієї з БД

2.7.2 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 високопродуктивну 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, "згенеруй 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 цілісності: 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(Рисунок 10).

```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)

```

Рисунок 10 – Приклад згенерованих даних (фрагмент `data\_\*.sql`)

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 набору даних BIRD-UKR.

2.7.3 Створення пар Питання – SQL запит

Методологія створення пар:

1. 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, охоплювали різні рівні складності (простий, середній, складний) та вимагали різноманітних SQL-операцій (фільтрація, з'єднання, агрегація).
2. 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[55], 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.

1. 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 продуктивність LLM для масової генерації з людським контролем для забезпечення найвищої якості еталонних запитів.

1. 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 11 – приклад згенерованих пар питання-SQL

2.7.4 Структура та компоненти набору даних BIRD-UKR

Розроблений набір даних 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):

1. `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`).

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

3. `database/` (директорія): Містить піддиректорії для кожної з 8 баз даних, що входять до набору даних. 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.

2.7.5 Інструменти тестування та оцінки

Для ефективного використання набору даних BIRD-UKR та оцінки NL2SQL систем було розроблено набір допоміжних скриптів та стандартизовано процес тестування.

Утиліти для роботи з датасетом:

1. 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 – приведення їх до стандартного вигляду для усунення несуттєвих відмінностей (наприклад, регістру, пробілів) перед порівнянням на точну відповідність (Exact Match).

2. `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).

3. `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.

4. 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 наборі даних 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 наборі даних BIRD-UKR.

2.7.6 Метрики оцінки якості моделей на наборі даних 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 наборі даних BIRD-UKR.

Розділ 3 Практичні результати та висновки

Цей розділ присвячений представленню та аналізу результатів експериментальної оцінки створеного методу.

У цьому розділі представлені результати тестування на всіх трьох зазначених датасетах. Проаналізовані 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 порівняні з доступними базовими показниками (baselines), зокрема з результатами оригінального фреймворку MAC-SQL, для кращого розуміння досягнутої продуктивності.

3.1 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 офіційною рейтинговою таблицею корпусу BIRD, оригінальна реалізація фреймворку MAC-SQL у поєднанні з моделлю GPT-4 (станом на листопад 2023 року) демонструвала Точність Виконання (EX) на тестовому (Test) наборі BIRD: 59.59%. Це слугує основним орієнтиром для порівняння нашої адаптації на датасеті BIRD.

Таблиця 3 - результати MAC-SQL на корпусі BIRD

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

* 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 4 - Mini Dev Execution Accuracy (EX)

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

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 вдосконалення якості генерації SQL.

3.2 Результати на стандартних корпусах (Spider & BIRD)

Таблиця 5 – результати

|  |  |  |  |
| --- | --- | --- | --- |
| 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% |
| Llama 3.1 70B | BIRD-ukr(postgresql) | 75 | 57.33% |

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.

1. Вдосконалення 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%)

3.3 Аналіз здатності моделі працювати з українським датасетом

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.

**4 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 СКБД 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 набір даних «BIRD-UKR». Цей корпус 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 наборах даних 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 корпусах 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 наборі даних 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 відкритою LLM Llama 3.1 70B(або будь-якою моделлю доступною через API call до Together AI) та промисловою СКБД PostgreSQL.~~
2. ~~Створенні та оприлюдненні першого комплексного україномовного набору даних 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. Набір даних 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 корпусу 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 СКБД PostgreSQL. Це робить важливий внесок у розвиток українського NLP та сприяє демократизації доступу до даних для ширшого кола користувачів.~~

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1 “Large Language Model Enhanced Text-to-SQL Generation: A Survey,” *Arxiv.org*, 2024. https://arxiv.org/html/2410.06011v1

2 P. Ma and S. Wang, “MT-Teql: Evaluating and Augmenting Neural NLIDB on Real-world Linguistic and Schema Variations,” *PVLDB*, vol. 15, no. 3, pp. 2150–8097, 2022, doi: https://doi.org/10.14778/3494124.3494139.

3 “From Natural Language to SQL: Review of LLM-based Text-to-SQL Systems,” *Arxiv.org*, 2023. https://arxiv.org/html/2410.01066v1

4 B. Qin *et al.*, “A survey on Text-to-SQL Parsing: Concepts, methods, and future Directions,” *arXiv.org*, Aug. 29, 2022. https://arxiv.org/abs/2208.13629

5 A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, “Attention is all you need,” in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30.   Curran Associates, Inc., 2017. [Online]. URL: https://proceedings.neurips.cc/paper\_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

6 G. Katsogiannis-Meimarakis and G. Koutrika, “A deep dive into deep learning approaches for text-to-sql systems,” in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 2846–2851

7 “Tool-Assisted Agent on SQL Inspection and Refinement in Real-World scenarios.” https://arxiv.org/html/2408.16991v1

8 “A Survey of NL2SQL with Large Language Models: Where are we, and where are we going?” https://arxiv.org/html/2408.05109v1

9 Bert F. Green, Alice K. Wolf, Carol Chomsky, and Kenneth Laughery. 1961. Baseball: an automatic question-answerer. In Papers presented at the May 9-11, 1961, western joint IRE-AIEE-ACM computer conference (IRE-AIEE-ACM '61 (Western)). Association for Computing Machinery, New York, NY, USA, 219–224. https://doi.org/10.1145/1460690.1460714

10 W. A. Woods. 1973. Progress in natural language understanding: an application to lunar geology. In Proceedings of the June 4-8, 1973, national computer conference and exposition (AFIPS '73). Association for Computing Machinery, New York, NY, USA, 441–450. https://doi.org/10.1145/1499586.1499695

11 Gary G. Hendrix, Earl D. Sacerdoti, Daniel Sagalowicz, and Jonathan Slocum. 1978. Developing a natural language interface to complex data. ACM Trans. Database Syst. 3, 2 (June 1978), 105–147. https://doi.org/10.1145/320251.320253

12 David H.D. Warren and Fernando C.N. Pereira. 1982. An Efficient Easily Adaptable System for Interpreting Natural Language Queries. American Journal of Computational Linguistics, 8(3-4):110–122. URL: https://aclanthology.org/J82-3002/

13 Y. Li, H. Yang, and H. V. Jagadish, “NaLIX,” Association for Computing Machinery  New York, NY, United States, Jun. 2005, doi: 10.1145/1066157.1066281.

14 S. Hochreiter and J. Schmidhuber, “Long Short-Term memory,” Neural Computation, vol. 9, no. 8, pp. 1735–1780, Nov. 1997, doi: 10.1162/neco.1997.9.8.1735.

15 V. Zhong, C. Xiong, and R. Socher, “Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning,” arXiv.org, Aug. 31, 2017. <https://arxiv.org/abs/1709.00103>

16 T. Yu, Z. Li, Z. Zhang, R. Zhang, and D. Radev, “TypeSQL: Knowledge-based Type-Aware neural Text-to-SQL generation,” arXiv.org, Apr. 25, 2018. <https://arxiv.org/abs/1804.09769>

17 X. Xu, C. Liu, and D. Song, “SQLNet: Generating structured queries from natural language without reinforcement learning,” arXiv.org, Nov. 13, 2017. <https://arxiv.org/abs/1711.04436>

18 T. Yu et al., “SyntaxSQLNet: Syntax tree networks for complex and Cross-DomainText-to-SQL task,” arXiv.org, Oct. 11, 2018. <https://arxiv.org/abs/1810.05237>

19 J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding,” arXiv.org, Oct. 11, 2018. <https://arxiv.org/abs/1810.04805>

20 C. Raffel et al., “Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer,” arXiv.org, Oct. 23, 2019. <https://arxiv.org/abs/1910.10683>

21 B. Wang, R. Shin, X. Liu, O. Polozov, and M. Richardson, “RAT-SQL: Relation-Aware schema encoding and linking for Text-to-SQL parsers,” arXiv.org, Nov. 10, 2019. <https://arxiv.org/abs/1911.04942>

22 P. Yin, G. Neubig, W.-T. Yih, and S. Riedel, “TaBERT: Pretraining for joint understanding of textual and tabular data,” arXiv.org, May 17, 2020. <https://arxiv.org/abs/2005.08314>

23 B. Bogin, M. Gardner, and J. Berant, “Representing Schema Structure with Graph Neural Networks for Text-to-SQL Parsing,” arXiv.org, May 15, 2019. <https://arxiv.org/abs/1905.06241>

24 Z. Gu et al., “Few-shot Text-to-SQL Translation using Structure and Content Prompt Learning,” Proceedings of the ACM on Management of Data, vol. 1, no. 2, pp. 1–28, Jun. 2023, doi: 10.1145/3589292.

25 D. Lee, C. Park, J. Kim, and H. Park, “MCS-SQL: Leveraging Multiple Prompts and Multiple-Choice selection for Text-to-SQL generation,” arXiv (Cornell University), May 2024, doi: 10.48550/arxiv.2405.07467.

26 J. Wei et al., “Chain-of-Thought prompting elicits reasoning in large language models,” arXiv.org, Jan. 28, 2022. <https://arxiv.org/abs/2201.11903>

27 J.-P. Zhu et al., “Chat2Query: A Zero-Shot Automatic Exploratory Data Analysis System with Large Language Models,” 2022 IEEE 38th International Conference on Data Engineering (ICDE), pp. 5429–5432, May 2024, doi: 10.1109/icde60146.2024.00420.

28 H. Zhang, R. Cao, L. Chen, H. Xu, and K. Yu, “ACT-SQL: In-Context Learning for Text-to-SQL with Automatically-Generated Chain-of-Thought,” arXiv.org, Oct. 26, 2023. <https://arxiv.org/abs/2310.17342>

29 M. Pourreza and D. Rafiei, “DIN-SQL: Decomposed In-Context Learning of Text-to-SQL with Self-Correction,” arXiv (Cornell University), Jan. 2023, doi: 10.48550/arxiv.2304.11015.

30 Z. Hong, Z. Yuan, H. Chen, Q. Zhang, F. Huang, and X. Huang, “Knowledge-to-SQL: Enhancing SQL Generation with Data Expert LLM,” arXiv (Cornell University), Feb. 2024, doi: 10.48550/arxiv.2402.11517.

31 D. Gao et al., “Text-to-SQL empowered by large language models: a benchmark evaluation,” arXiv (Cornell University), Jan. 2023, doi: 10.48550/arxiv.2308.15363.

32 A. Zhuang et al., “STRUctLM: towards building generalist models for structured knowledge grounding,” arXiv.org, Feb. 26, 2024. <https://arxiv.org/abs/2402.16671>

33 E. J. Hu et al., “LORA: Low-Rank adaptation of Large Language Models,” arXiv.org, Jun. 17, 2021. <https://arxiv.org/abs/2106.09685>

34 T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, “QLORA: Efficient Finetuning of Quantized LLMS,” arXiv.org, May 23, 2023. <https://arxiv.org/abs/2305.14314>

35 R. A. Jacobs, M. I. Jordan, S. J. Nowlan, and G. E. Hinton, “Adaptive mixtures of local experts,” Neural Computation, vol. 3, no. 1, pp. 79–87, Feb. 1991, doi: 10.1162/neco.1991.3.1.79.

36 M. Pourreza, R. Sun, H. Li, L. Miculicich, T. Pfister, and S. O. Arik, “SQL-GEN: Bridging the dialect gap for Text-to-SQL via synthetic data and model merging,” arXiv.org, Aug. 22, 2024. <https://arxiv.org/abs/2408.12733>

37 H. Li et al., “CODES: Towards building open-source language models for Text-to-SQL,” arXiv (Cornell University), Feb. 2024, doi: 10.48550/arxiv.2402.16347.

38 H. Li, J. Zhang, C. Li, and H. Chen, “RESDSQL: Decoupling schema linking and skeleton parsing for Text-to-SQL,” arXiv (Cornell University), Jan. 2023, doi: 10.48550/arxiv.2302.05965.

39 J. Cen, J. Liu, Z. Li, and J. Wang, “SQLFixAgent: Towards Semantic-Accurate Text-to-SQL parsing via Consistency-Enhanced Multi-Agent Collaboration,” arXiv.org, Jun. 19, 2024. https://arxiv.org/abs/2406.13408

40 B. Wang et al., “MAC-SQL: a Multi-Agent collaborative framework for Text-to-SQL,” arXiv.org, Dec. 18, 2023. <https://arxiv.org/abs/2312.11242>

41 “The Dawn of Natural Language to SQL: Are we fully ready?” <https://arxiv.org/html/2406.01265v1>

42 P. Price, “Evaluation of Spoken Language Systems: the ATIS Domain,” ACL Anthology, 1990. <https://aclanthology.org/H90-1020>

43 J. M. Zelle and R. J. Mooney, “Learning to parse database queries using inductive logic programming,” in Proceedings of the Thirteenth National Conference on Artificial Intelligence - Volume 2, ser. AAAI’96. AAAI Press, 1996, p. 1050–1055.

44 T. Yu et al., “Spider: a Large-Scale Human-Labeled dataset for complex and Cross-Domain semantic parsing and Text-to-SQL task,” Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, Jan. 2018, doi: 10.18653/v1/d18-1425.

45 Tao Yu, Rui Zhang, Michihiro Yasunaga, Yi Chern Tan, Xi Victoria Lin, Suyi Li, Heyang Er, Irene Li, Bo Pang, Tao Chen, Emily Ji, Shreya Dixit, David Proctor, Sungrok Shim, Jonathan Kraft, Vincent Zhang, Caiming Xiong, Richard Socher, and Dragomir Radev. 2019. SParC: Cross-Domain Semantic Parsing in Context. In Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics, pages 4511–4523, Florence, Italy. Association for Computational Linguistics.

46 T. Yu et al., “CoSQL: A Conversational Text-to-SQL Challenge Towards Cross-Domain Natural Language Interfaces to Databases,” Association for Computational Linguistics, Jan. 2019, doi: 10.18653/v1/d19-1204.

47 C.-H. Lee, O. Polozov, and M. Richardson, “KaggleDBQA: Realistic Evaluation of Text-to-SQL Parsers,” Association for Computational Linguistics, Jan. 2021, doi: 10.18653/v1/2021.acl-long.176.

48 M. Hazoom, V. Malik, and B. Bogin, “Text-to-SQL in the Wild: a Naturally-Occurring dataset based on stack exchange data,” arXiv (Cornell University), Jan. 2021, doi: 10.48550/arxiv.2106.05006.

49 G. Lee et al., “EHRSQL: a practical Text-to-SQL benchmark for electronic health records,” arXiv (Cornell University), Jan. 2023, doi: 10.48550/arxiv.2301.07695.

50 Y. Gan, X. Chen, and M. Purver, “Exploring underexplored limitations of cross-domain text-to-sql generalization,” 2021

51 Y. Gan, X. Chen, Q. Huang, M. Purver, J. R. Woodward, J. Xie, and P. Huang, “Towards robustness of text-to-sql models against synonym substitution,” 2021.

52 B. Zhang et al., “Benchmarking the Text-to-SQL capability of large Language Models: A Comprehensive Evaluation,” arXiv (Cornell University), Mar. 2024, doi: 10.48550/arxiv.2403.02951.

53 J. Li et al., “Can LLM already serve as a database interface? a BIG bench for Large-Scale database grounded Text-to-SQLs,” arXiv.org, May 04, 2023. https://arxiv.org/abs/2305.03111

54 Anthropic, "Claude 3.7 Sonnet System Card," Anthropic, Feb 24, 2025. [Online]. URL: <https://assets.anthropic.com/m/785e231869ea8b3b/original/claude-3-7-sonnet-system-card.pdf>

55 Google DeepMind, "Gemini 2.5 Pro Preview Model Card”, April 28, 2025, [Online]. URL: <https://storage.googleapis.com/model-cards/documents/gemini-2.5-pro-preview.pdf>

56 Bernard Chang, Wei Yih Yap " Getting started with NL2SQL (natural language to SQL) with Gemini and BigQuery", Google Cloud Blog, November 8, 2024. [Online]. URL: https://cloud.google.com/blog/products/data-analytics/nl2sql-with-bigquery-and-gemini.

57 H. Touvron et al., “LLAMA: Open and Efficient Foundation Language Models,” arXiv.org, Feb. 27, 2023. https://arxiv.org/abs/2302.13971

58 OpenAI et al., “GPT-4 Technical Report,” arXiv.org, Mar. 15, 2023. <https://arxiv.org/abs/2303.08774>

ДОДАТОК А

"""

Together AI API Functions for MAC-SQL

"""

*import* os

*import* json

*import* requests

*import* time

*import* logging

*import* random

*from* typing *import* Dict, Any, List, Tuple

*# Load environment variables from .env file*

*try*:

*from* dotenv *import* load\_dotenv

    print("Loading .env file from current directory...")

    load\_dotenv()

    print("Loaded .env file successfully")

*except* ImportError:

    print("dotenv module not found, using environment variables as is")

*# Configure logging*

logging.basicConfig(*level*=logging.INFO)

logger = logging.getLogger(\_\_name\_\_)

*# Get environment variables*

*# Remove these global reads/prints - they happen too early*

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

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

*# Print environment variable values for debugging*

*# print(f"API Key (exists): {'Yes' if TOGETHER\_API\_KEY else 'No'}")*

*# print(f"API Key (length): {len(TOGETHER\_API\_KEY)} characters")*

*# print(f"Model: {TOGETHER\_MODEL}")*

*# Initialize tracking variables for logging*

total\_prompt\_tokens = 0

total\_response\_tokens = 0

log\_path = None

api\_trace\_json\_path = None

*# Rate limiting parameters*

MAX\_RETRIES = 20  *# Increased from 5 to 20 for more resilience*

RETRY\_DELAY = 5  *# seconds*

MAX\_BACKOFF = 120  *# Maximum delay in seconds (cap the exponential backoff)*

def **init\_log\_path**(*my\_log\_path*):

    """Initialize log path for API call logging"""

    global log\_path

    global api\_trace\_json\_path

    global total\_prompt\_tokens

    global total\_response\_tokens

    log\_path = *my\_log\_path*

    total\_prompt\_tokens = 0

    total\_response\_tokens = 0

*# Create log directory if needed*

*if* log\_path:

        log\_dir = os.path.dirname(log\_path)

        os.makedirs(log\_dir, *exist\_ok*=True)

*# Set up API trace log file*

        api\_trace\_json\_path = os.path.join(log\_dir, 'api\_trace.json')

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

    """

    Call Together AI API to generate a response

    Args:

        prompt: The prompt to send to the API

    Returns:

        Tuple of (response text, prompt tokens, completion tokens)

    """

*# Read API Key and Model directly from environment \*inside\* the function*

*# This ensures it picks up changes from .env or arguments*

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

    default\_model = "meta-llama/Llama-3.3-70B-Instruct-Turbo" *# Define default here*

    model = os.getenv("TOGETHER\_MODEL", default\_model)

*# Check if API key is available*

*if* not api\_key:

*raise* ValueError("Together API key not found. Set TOGETHER\_API\_KEY environment variable.")

*# Log model being used*

    logger.info(f"\nUsing Together AI model: {model}\n") *# Log the currently resolved model*

*# Prepare API request*

    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,

        "max\_tokens": 4096

    }

*# Make API request with retry logic*

*for* attempt *in* range(MAX\_RETRIES):

*try*:

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

*# Check for rate limiting*

*if* response.status\_code == 429:

                wait\_time = min(RETRY\_DELAY \* (2 \*\* attempt), MAX\_BACKOFF)  *# Exponential backoff with cap*

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

                time.sleep(wait\_time)

*continue*

*# Check for other errors*

*if* response.status\_code != 200:

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

*if* attempt < MAX\_RETRIES - 1:

                    time.sleep(RETRY\_DELAY)

*continue*

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

*# Parse response*

            result = response.json()

*# Extract text and token counts*

            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

*except* Exception *as* e:

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

*if* attempt < MAX\_RETRIES - 1:

                time.sleep(RETRY\_DELAY)

*else*:

*raise*

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

    """

    Safe wrapper for LLM API call with logging

    Args:

        input\_prompt: The prompt to send to the LLM

        \*\*kwargs: Additional context for logging

    Returns:

        Generated response text

    """

    global total\_prompt\_tokens

    global total\_response\_tokens

    global log\_path

    global api\_trace\_json\_path

    global TOGETHER\_MODEL

*# Try to make the API call with retries*

*for* attempt *in* range(MAX\_RETRIES):

*try*:

*# Make API call*

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

*# Track token usage for rate limiting*

            total\_prompt\_tokens += prompt\_token

            total\_response\_tokens += response\_token

*# Log the results based on logging configuration*

*if* log\_path is None:

*# Just print to console if no log path set*

                print(f"\nResponse: \n{sys\_response}")

                print(f"\nTokens (prompt/response): {prompt\_token}/{response\_token}\n")

*else*:

*# Full logging to file*

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

                    print('\n' + f'\*'\*20 +'\n', *file*=log\_fp)

                    print(*input\_prompt*, *file*=log\_fp)

                    print('\n' + f'='\*20 +'\n', *file*=log\_fp)

                    print(sys\_response, *file*=log\_fp)

                    print(f'\nTokens (prompt/response): {prompt\_token}/{response\_token}\n', *file*=log\_fp)

*# Also log to API trace JSON if available*

*if* api\_trace\_json\_path:

*# Get current model name (environment variable might have changed)*

                    current\_model = os.getenv("TOGETHER\_MODEL", TOGETHER\_MODEL)

*# Create trace entry with all context*

                    trace\_entry = {

                        "prompt": *input\_prompt*.strip(),

                        "response": sys\_response.strip(),

                        "prompt\_tokens": prompt\_token,

                        "response\_tokens": response\_token,

                        "total\_prompt\_tokens": total\_prompt\_tokens,

                        "total\_response\_tokens": total\_response\_tokens,

                        "timestamp": time.strftime("%Y-%m-%d %H:%M:%S"),

                        "model": current\_model

                    }

*# Add any additional context from kwargs*

*for* k, v *in* *kwargs*.items():

                        trace\_entry[k] = v

*# Write to trace file*

*with* open(api\_trace\_json\_path, 'a+', *encoding*='utf8') *as* trace\_fp:

                        trace\_fp.write(json.dumps(trace\_entry, *ensure\_ascii*=False) + "\n")

*# Return the response*

*return* sys\_response

*except* Exception *as* e:

            logger.error(f"API call failed: {str(e)}")

*# Get current model for error message*

            current\_model = os.getenv("TOGETHER\_MODEL", TOGETHER\_MODEL)

            print(f"Request {current\_model} failed. Try {attempt+1} of {MAX\_RETRIES}. Sleeping {RETRY\_DELAY} seconds.")

            time.sleep(RETRY\_DELAY)

*# If all retries failed*

    error\_msg = f"Failed to call LLM API after {MAX\_RETRIES} attempts"

    logger.error(error\_msg)

*raise* Exception(error\_msg)

def **call\_llm**(*model\_name*: str, *messages*: List[Dict[str, str]]) -> Dict[str, Any]:

    """

    Call the LLM with a structured message list.

    Args:

        model\_name: The name of the model to use

        messages: List of message dictionaries with role and content

    Returns:

        Dictionary with response content

    """

*# Format the messages into a prompt for our current API implementation*

*if* not *messages*:

*return* {"content": ""}

*# For now, we'll just use the last user message as the prompt*

*# This is a simplification - a proper implementation would format all messages*

    user\_messages = [m *for* m *in* *messages* *if* m["role"] == "user"]

*if* not user\_messages:

*return* {"content": ""}

    prompt = user\_messages[-1]["content"]

*# Call the LLM*

    response\_text = safe\_call\_llm(prompt)

*# Return a structured response*

*return* {

        "content": response\_text,

        "model": *model\_name*

    }

*if* \_\_name\_\_ == "\_\_main\_\_":

*# Test the API*

    response = safe\_call\_llm("Explain how a relational database works in one paragraph.")

    print(f"Test response: {response}")

ДОДАТОК Б

*import* sys

*import* json

*import* time

*import* os

*import* logging

*from* core.api\_config *import* \*

*# Configure logging*

logging.basicConfig(*level*=logging.INFO)

logger = logging.getLogger(\_\_name\_\_)

MAX\_TRY = 5

*# 用来传递外面的字典进来*

world\_dict = {}

log\_path = None

api\_trace\_json\_path = None

total\_prompt\_tokens = 0

total\_response\_tokens = 0

def **init\_log\_path**(*my\_log\_path*):

    global total\_prompt\_tokens

    global total\_response\_tokens

    global log\_path

    global api\_trace\_json\_path

    log\_path = *my\_log\_path*

    total\_prompt\_tokens = 0

    total\_response\_tokens = 0

    dir\_name = os.path.dirname(log\_path)

    os.makedirs(dir\_name, *exist\_ok*=True)

*# 另外一个记录api调用的文件*

    api\_trace\_json\_path = os.path.join(dir\_name, 'api\_trace.json')

def **api\_func**(*prompt*:str):

    """

    Call the appropriate API based on configuration

    """

    global MODEL\_NAME

*if* USE\_TOGETHER\_AI:

*# Use Together AI API*

*try*:

*from* core *import* api

*return* api.together\_api\_call(*prompt*)

*except* ImportError:

            logger.warning("Together API module not found, falling back to OpenAI")

*# Fall back to OpenAI API*

    print(f"\nUse OpenAI model: {MODEL\_NAME}\n")

*try*:

*import* openai

*if* 'Llama' in MODEL\_NAME:

            openai.api\_version = None

            openai.api\_type = "open\_ai"

            openai.api\_key = "EMPTY"

            response = openai.ChatCompletion.create(

*model*=MODEL\_NAME,

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

            )

*else*:

            response = openai.ChatCompletion.create(

*engine*=MODEL\_NAME,

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

*temperature*=0.1

            )

        text = response['choices'][0]['message']['content'].strip()

        prompt\_token = response['usage']['prompt\_tokens']

        response\_token = response['usage']['completion\_tokens']

*return* text, prompt\_token, response\_token

*except* Exception *as* e:

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

*raise*

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

    """

    Call LLM with error handling and logging

    """

    global MODEL\_NAME

    global log\_path

    global api\_trace\_json\_path

    global total\_prompt\_tokens

    global total\_response\_tokens

    global world\_dict

*# If Together API is enabled and available, use its own safe\_call\_llm*

*if* USE\_TOGETHER\_AI:

*try*:

*from* core *import* api

*return* api.safe\_call\_llm(*input\_prompt*, \*\**kwargs*)

*except* ImportError:

            logger.warning("Together API module not found, using default implementation")

*# Default implementation with OpenAI*

*for* i *in* range(MAX\_TRY):

*try*:

*if* log\_path is None:

*# Simple logging to console*

                sys\_response, prompt\_token, response\_token = api\_func(*input\_prompt*)

                print(f"\nsys\_response: \n{sys\_response}")

                print(f'\n prompt\_token,response\_token: {prompt\_token} {response\_token}\n')

*else*:

*# Comprehensive logging to file*

*if* (log\_path is None) or (api\_trace\_json\_path is None):

*raise* FileExistsError('log\_path or api\_trace\_json\_path is None, init\_log\_path first!')

*with* open(log\_path, 'a+', *encoding*='utf8') *as* log\_fp, open(api\_trace\_json\_path, 'a+', *encoding*='utf8') *as* trace\_json\_fp:

                    print('\n' + f'\*'\*20 +'\n', *file*=log\_fp)

                    print(*input\_prompt*, *file*=log\_fp)

                    print('\n' + f'='\*20 +'\n', *file*=log\_fp)

                    sys\_response, prompt\_token, response\_token = api\_func(*input\_prompt*)

                    print(sys\_response, *file*=log\_fp)

                    print(f'\n prompt\_token,response\_token: {prompt\_token} {response\_token}\n', *file*=log\_fp)

                    print(f'\n prompt\_token,response\_token: {prompt\_token} {response\_token}\n')

*# Reset dict for this invocation*

*if* len(world\_dict) > 0:

                        world\_dict = {}

*# Add kwargs to world\_dict*

*if* len(*kwargs*) > 0:

                        world\_dict = {}

*for* k, v *in* *kwargs*.items():

                            world\_dict[k] = v

*# Add prompt and response to world\_dict*

                    world\_dict['response'] = '\n' + sys\_response.strip() + '\n'

                    world\_dict['input\_prompt'] = *input\_prompt*.strip() + '\n'

                    world\_dict['prompt\_token'] = prompt\_token

                    world\_dict['response\_token'] = response\_token

*# Track total tokens*

                    total\_prompt\_tokens += prompt\_token

                    total\_response\_tokens += response\_token

                    world\_dict['cur\_total\_prompt\_tokens'] = total\_prompt\_tokens

                    world\_dict['cur\_total\_response\_tokens'] = total\_response\_tokens

*# Write to trace file*

                    world\_json\_str = json.dumps(world\_dict, *ensure\_ascii*=False)

                    print(world\_json\_str, *file*=trace\_json\_fp)

*# Clean up*

                    world\_dict = {}

                    world\_json\_str = ''

*# Log token totals*

                    print(f'\n total\_prompt\_tokens,total\_response\_tokens: {total\_prompt\_tokens} {total\_response\_tokens}\n', *file*=log\_fp)

                    print(f'\n total\_prompt\_tokens,total\_response\_tokens: {total\_prompt\_tokens} {total\_response\_tokens}\n')

*return* sys\_response

*except* Exception *as* ex:

            print(ex)

            print(f'Request {MODEL\_NAME} failed. try {i} times. Sleep 20 secs.')

            time.sleep(20)

*raise* ValueError('safe\_call\_llm error after multiple retries!')

*if* \_\_name\_\_ == "\_\_main\_\_":

    res = safe\_call\_llm('Test query: what is SQL?')

    print(res)

ДОДАТОК В

*#!/usr/bin/env python3*

*# -\*- coding: utf-8 -\*-*

"""

Database utility functions for MAC-SQL.

Provides connection handling for both SQLite and PostgreSQL databases.

"""

*import* os

*import* sqlite3

*from* dotenv *import* load\_dotenv

*# Try to import psycopg2, but make it optional*

*try*:

*import* psycopg2

    PSYCOPG2\_AVAILABLE = True

*except* ImportError:

    PSYCOPG2\_AVAILABLE = False

*# Load environment variables*

load\_dotenv()

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

    """

    Get a database connection based on the dataset type.

    Args:

        dataset\_name: Name of the dataset ('spider', 'bird', or 'bird-ukr')

        db\_id: Database identifier

        db\_base\_path: Base path for SQLite databases (only used for SQLite)

    Returns:

        tuple: (connection, db\_type)

    """

    conn = None

    db\_type = 'sqlite'  *# Default*

*# Determine database type from dataset name*

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

        db\_type = 'postgres'

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

        db\_type = 'sqlite'

*else*:

*# Default to SQLite for unknown datasets*

        db\_type = 'sqlite'

*# PostgreSQL connection*

*if* db\_type == 'postgres':

*if* not PSYCOPG2\_AVAILABLE:

*raise* ImportError("psycopg2 is required for PostgreSQL connections. "

                              "Please install it with: pip install psycopg2-binary")

*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*

            )

*except* psycopg2.Error *as* e:

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

*# SQLite connection*

*elif* db\_type == 'sqlite':

*if* *db\_base\_path* is None:

*# Try to determine base path based on dataset*

*if* *dataset\_name* == 'spider':

*db\_base\_path* = os.path.join(os.path.dirname(os.path.dirname(\_\_file\_\_)),

                                           "data", "spider", "database")

*elif* *dataset\_name* == 'bird':

*db\_base\_path* = os.path.join(os.path.dirname(os.path.dirname(\_\_file\_\_)),

                                           "data", "bird", "database")

*else*:

*raise* ValueError(f"No db\_base\_path provided and couldn't determine it for dataset {*dataset\_name*}")

*# Construct full path to SQLite file*

        db\_file\_path = os.path.join(*db\_base\_path*, *db\_id*, f"{*db\_id*}.sqlite")

*if* not os.path.exists(db\_file\_path):

*raise* FileNotFoundError(f"SQLite database file not found: {db\_file\_path}")

*try*:

            conn = sqlite3.connect(db\_file\_path)

*except* sqlite3.Error *as* e:

*raise* ConnectionError(f"Failed to connect to SQLite database {db\_file\_path}: {e}")

*return* conn, db\_type

def **get\_schema**(*conn*, *db\_type*):

    """

    Get database schema (tables, columns, types) from a database connection.

    Works with both SQLite and PostgreSQL.

    Args:

        conn: Database connection

        db\_type: Database type ('sqlite' or 'postgres')

    Returns:

        dict: Schema information containing tables and their columns

    """

    schema\_info = {

        'tables': [],

        'columns': {},

        'primary\_keys': {},

        'foreign\_keys': {}

    }

*if* *db\_type* == 'postgres':

        cursor = *conn*.cursor()

*# Get tables*

        cursor.execute("""

            SELECT table\_name FROM information\_schema.tables

            WHERE table\_schema = 'public' AND table\_type = 'BASE TABLE';

        """)

        schema\_info['tables'] = [row[0] *for* row *in* cursor.fetchall()]

*# Get columns and their types for each table*

*for* table\_name *in* schema\_info['tables']:

            cursor.execute("""

                SELECT column\_name, data\_type

                FROM information\_schema.columns

                WHERE table\_schema = 'public' AND table\_name = %s

                ORDER BY ordinal\_position;

            """, (table\_name,))

            schema\_info['columns'][table\_name] = [(col[0], col[1]) *for* col *in* cursor.fetchall()]

*# Get primary keys*

*for* table\_name *in* schema\_info['tables']:

            cursor.execute("""

                SELECT c.column\_name

                FROM information\_schema.table\_constraints tc

                JOIN information\_schema.constraint\_column\_usage AS ccu

                USING (constraint\_schema, constraint\_name)

                JOIN information\_schema.columns AS c

                ON c.table\_schema = tc.constraint\_schema

                AND tc.table\_name = c.table\_name

                AND ccu.column\_name = c.column\_name

                WHERE constraint\_type = 'PRIMARY KEY' AND tc.table\_name = %s;

            """, (table\_name,))

            pk\_columns = [row[0] *for* row *in* cursor.fetchall()]

*if* pk\_columns:

                schema\_info['primary\_keys'][table\_name] = pk\_columns

*# Get foreign keys*

        cursor.execute("""

            SELECT

                tc.table\_name,

                kcu.column\_name,

                ccu.table\_name AS foreign\_table\_name,

                ccu.column\_name AS foreign\_column\_name

            FROM information\_schema.table\_constraints AS tc

            JOIN information\_schema.key\_column\_usage AS kcu

              ON tc.constraint\_name = kcu.constraint\_name

              AND tc.table\_schema = kcu.table\_schema

            JOIN information\_schema.constraint\_column\_usage AS ccu

              ON ccu.constraint\_name = tc.constraint\_name

              AND ccu.table\_schema = tc.table\_schema

            WHERE tc.constraint\_type = 'FOREIGN KEY';

        """)

        foreign\_keys = cursor.fetchall()

*for* fk *in* foreign\_keys:

            table\_name, column\_name, ref\_table, ref\_column = fk

*if* table\_name not in schema\_info['foreign\_keys']:

                schema\_info['foreign\_keys'][table\_name] = []

            schema\_info['foreign\_keys'][table\_name].append({

                'column': column\_name,

                'ref\_table': ref\_table,

                'ref\_column': ref\_column

            })

        cursor.close()

*elif* *db\_type* == 'sqlite':

        cursor = *conn*.cursor()

*# Get tables (excluding sqlite\_\* tables)*

        cursor.execute("""

            SELECT name FROM sqlite\_master

            WHERE type='table' AND name NOT LIKE 'sqlite\_%';

        """)

        schema\_info['tables'] = [row[0] *for* row *in* cursor.fetchall()]

*# Get columns and their types for each table*

*for* table\_name *in* schema\_info['tables']:

            cursor.execute(f'PRAGMA table\_info("{table\_name}");')

*# PRAGMA table\_info returns: cid, name, type, notnull, dflt\_value, pk*

            pragma\_results = cursor.fetchall()

            schema\_info['columns'][table\_name] = [(row[1], row[2]) *for* row *in* pragma\_results]

*# Extract primary keys*

            pk\_columns = [row[1] *for* row *in* pragma\_results *if* row[5] == 1]

*if* pk\_columns:

                schema\_info['primary\_keys'][table\_name] = pk\_columns

*# Get foreign keys for each table*

*for* table\_name *in* schema\_info['tables']:

            cursor.execute(f'PRAGMA foreign\_key\_list("{table\_name}");')

*# PRAGMA foreign\_key\_list returns: id, seq, table, from, to, on\_update, on\_delete, match*

            fk\_results = cursor.fetchall()

*if* fk\_results:

                schema\_info['foreign\_keys'][table\_name] = []

*for* fk *in* fk\_results:

                    schema\_info['foreign\_keys'][table\_name].append({

                        'column': fk[3],  *# 'from' column*

                        'ref\_table': fk[2],  *# referenced table*

                        'ref\_column': fk[4]  *# 'to' column*

                    })

        cursor.close()

*return* schema\_info

def **format\_schema\_for\_prompt**(*schema\_info*, *include\_pk*=True, *include\_fk*=True):

    """

    Format schema information into a string for use in LLM prompts.

    Args:

        schema\_info: Schema information from get\_schema

        include\_pk: Whether to include primary key information

        include\_fk: Whether to include foreign key information

    Returns:

        str: Formatted schema string

    """

    schema\_str = []

*for* table\_name *in* *schema\_info*['tables']:

        column\_info = []

*for* col\_name, col\_type *in* *schema\_info*['columns'][table\_name]:

*# Mark primary keys if included*

            pk\_marker = ""

*if* *include\_pk* and table\_name in *schema\_info*['primary\_keys'] and col\_name in *schema\_info*['primary\_keys'][table\_name]:

                pk\_marker = " [PRIMARY KEY]"

            column\_info.append(f"{col\_name} ({col\_type}){pk\_marker}")

*# Add table definition with columns*

        schema\_str.append(f"Table: {table\_name}")

        schema\_str.append("Columns: " + ", ".join(column\_info))

*# Add foreign key information if included*

*if* *include\_fk* and table\_name in *schema\_info*['foreign\_keys']:

            fk\_info = []

*for* fk *in* *schema\_info*['foreign\_keys'][table\_name]:

                fk\_info.append(f"{fk['column']} -> {fk['ref\_table']}.{fk['ref\_column']}")

*if* fk\_info:

                schema\_str.append("Foreign Keys: " + ", ".join(fk\_info))

        schema\_str.append("")  *# Add blank line between tables*

*return* "\n".join(schema\_str)

def **execute\_query**(*conn*, *query*, *db\_type*, *fetch*=True):

    """

    Execute an SQL query and return results.

    Args:

        conn: Database connection

        query: SQL query to execute

        db\_type: Database type ('sqlite' or 'postgres')

        fetch: Whether to fetch and return results

    Returns:

        list: Query results (if fetch=True)

    """

    cursor = *conn*.cursor()

*try*:

        cursor.execute(*query*)

*if* *fetch*:

            results = cursor.fetchall()

            column\_names = [desc[0] *for* desc *in* cursor.description] *if* cursor.description *else* []

            cursor.close()

*return* {'rows': results, 'columns': column\_names}

*else*:

*conn*.commit()

            cursor.close()

*return* True

*except* (sqlite3.Error, psycopg2.Error) *as* e:

*conn*.rollback()

        cursor.close()

*raise* Exception(f"Query execution error: {e}")

ДОДАТОК Г

*#!/usr/bin/env python*

"""

PostgreSQL Selector for the BIRD-UKR dataset.

Optimized for Ukrainian database schema handling.

"""

*import* os

*import* logging

*import* json

*from* typing *import* Dict, List, Any, Tuple, Optional

*import* re

*import* psycopg2

*from* psycopg2.extras *import* RealDictCursor

*# Import base Selector*

*from* core.agents *import* Selector, BaseAgent

*from* core.const\_ukr *import* selector\_template\_ukr, SELECTOR\_NAME, DECOMPOSER\_NAME

*from* core.utils *import* parse\_json

*from* core.api *import* call\_llm

logger = logging.getLogger(\_\_name\_\_)

class PostgreSQLSelector(BaseAgent):

    """

    Smart PostgreSQL Selector optimized for BIRD-UKR dataset.

    """

    def **\_\_init\_\_**(*self*, *data\_path*: str, *tables\_json\_path*: str,

*model\_name*: str, *dataset\_name*: str):

        """

        Initialize the PostgreSQL Selector.

        Args:

            data\_path: Path to the dataset

            tables\_json\_path: Path to the tables.json file

            model\_name: Name of the model to use

            dataset\_name: Name of the dataset

        """

        super().\_\_init\_\_()

*self*.name = SELECTOR\_NAME

*self*.data\_path = *data\_path*

*self*.tables\_json\_path = *tables\_json\_path*

*self*.model\_name = *model\_name*

*self*.dataset\_name = *dataset\_name*

*# Get PostgreSQL credentials from environment*

*self*.pg\_user = os.environ.get('PG\_USER', 'postgres')

*self*.pg\_password = os.environ.get('PG\_PASSWORD', '')

*self*.pg\_host = os.environ.get('PG\_HOST', 'localhost')

*self*.pg\_port = os.environ.get('PG\_PORT', '5432')

*# Cache for database schema information*

*self*.schema\_cache = {}

        logger.info("Initialized PostgreSQL Selector")

    def **talk**(*self*, *message*: Dict[str, Any]) -> Dict[str, Any]:

        """

        Process a message, analyze the query, and select relevant tables and columns.

        Args:

            message: The message to process

        Returns:

            The updated message with selected schema information

        """

*# Extract relevant information from the message*

        db\_id = *message*.get("db\_id", "")

*# Support both 'query' and 'question' keys for flexibility*

        query = *message*.get("query", "")

*if* not query:

            query = *message*.get("question", "")

*# If query came from 'question', add it back as 'query' for consistency*

*if* query:

*message*["query"] = query

        evidence = *message*.get("evidence", "")

*if* not db\_id or not query:

            logger.warning("Missing db\_id or query in message")

*message*["send\_to"] = DECOMPOSER\_NAME

*return* *message*

*# Load database schema*

        logger.info(f"Loading schema for {db\_id}")

        schema\_info = *self*.get\_schema(db\_id)

*if* not schema\_info or not schema\_info.get("tables"):

            logger.warning(f"No schema information found for {db\_id}")

*message*["desc\_str"] = f"Database {db\_id} contains no tables."

*message*["fk\_str"] = ""

*message*["send\_to"] = DECOMPOSER\_NAME

*return* *message*

*# Format full schema descriptions*

        desc\_str, fk\_str = *self*.format\_schema(schema\_info)

*# CRITICAL: Log the actual schema that's generated*

        logger.info(f"Generated schema for {db\_id}. Schema length: {len(desc\_str)}")

        logger.debug(f"Schema:\n{desc\_str}")

        logger.debug(f"Foreign keys:\n{fk\_str}")

*# Now use the LLM to select relevant tables and columns based on the question*

        selection\_prompt = selector\_template\_ukr.format(

*question*=query,

*db\_id*=db\_id,

*desc\_str*=desc\_str,

*fk\_str*=fk\_str,

*evidence*=evidence

        )

*# Call LLM to analyze the question and select relevant tables/columns*

        selection\_response = call\_llm(

*model\_name*=*self*.model\_name,

*messages*=[

                {"role": "system", "content": "You are a database schema expert that helps identify relevant tables and columns needed to answer specific questions."},

                {"role": "user", "content": selection\_prompt}

            ]

        )

*# Extract selected tables and columns from response*

        selection\_content = selection\_response.get("content", "")

*# CRITICAL: Log the actual response from the LLM*

        logger.info(f"Got selection response. Length: {len(selection\_content)}")

        logger.debug(f"Selection response:\n{selection\_content}")

*try*:

*# Look for JSON block in the response*

            json\_match = re.search(r'```json\s\*(.\*?)\s\*```', selection\_content, re.DOTALL)

*if* json\_match:

                json\_str = json\_match.group(1)

                logger.info(f"Found JSON block in response: {json\_str[:100]}...")

                selection\_data = json.loads(json\_str)

*else*:

*# Try to extract any JSON-like structure*

                logger.info("No JSON code block found, trying to parse content directly")

                json\_start = selection\_content.find('{')

                json\_end = selection\_content.rfind('}')

*if* json\_start >= 0 and json\_end > json\_start:

                    json\_str = selection\_content[json\_start:json\_end+1]

                    logger.info(f"Extracted JSON-like structure: {json\_str[:100]}...")

                    selection\_data = json.loads(json\_str)

*else*:

*# Last resort: try to parse the entire content*

                    logger.info("Trying to use custom JSON parser for content")

                    selection\_data = parse\_json(selection\_content)

            logger.info(f"Parsed selection data structure: {selection\_data}")

*# IMPORTANT: Handle different JSON structure variations*

*# The LLM might return different formats:*

*# 1. {"selected\_tables": ["table1", "table2"]}*

*# 2. {"table1": ["col1", "col2"], "table2": "\*"}*

*# 3. {"table1": "keep\_all", "table2": ["col1", "col2"], "table3": "drop\_all"}*

            selected\_tables = []

*# Handle format 1: "selected\_tables" key with list of tables*

*if* "selected\_tables" in selection\_data:

                selected\_tables = selection\_data["selected\_tables"]

                logger.info(f"Found 'selected\_tables' format: {selected\_tables}")

*# Handle format 2 & 3: Dictionary with table names as keys*

*else*:

*# Treat any table listed in the response as selected*

*for* table\_name, cols *in* selection\_data.items():

*# Skip if explicitly marked to drop*

*if* cols == "drop\_all" or cols == False:

                        logger.info(f"Table {table\_name} explicitly dropped")

*continue*

*# Include if it exists in our schema*

*if* table\_name in schema\_info["tables"]:

                        selected\_tables.append(table\_name)

                        logger.info(f"Table {table\_name} selected from structure")

*# Explanation may or may not be present*

            explanation = selection\_data.get("explanation", "")

*# Filter schema to only include selected tables*

            selected\_schema = {"tables": {}, "foreign\_keys": []}

*for* table *in* selected\_tables:

*if* table in schema\_info["tables"]:

                    selected\_schema["tables"][table] = schema\_info["tables"][table]

*# Include relevant foreign keys*

*for* fk *in* schema\_info["foreign\_keys"]:

*if* (fk["source\_table"] in selected\_tables and

                    fk["target\_table"] in selected\_tables):

                    selected\_schema["foreign\_keys"].append(fk)

*# Format selected schema*

            selected\_desc\_str, selected\_fk\_str = *self*.format\_schema(selected\_schema)

*# Log final selected schema*

            logger.info(f"Selected tables: {selected\_tables}")

            logger.info(f"Final schema length: {len(selected\_desc\_str)}")

            logger.debug(f"Final schema:\n{selected\_desc\_str}")

*# Add selection info to the message*

*message*["desc\_str"] = selected\_desc\_str

*message*["fk\_str"] = selected\_fk\_str

*message*["selection\_explanation"] = explanation

*except* Exception *as* e:

            logger.warning(f"Error parsing selection response: {e}")

            logger.exception("Detailed traceback:")

*# Fallback to full schema if parsing fails*

            logger.info("Using full schema as fallback")

*message*["desc\_str"] = desc\_str

*message*["fk\_str"] = fk\_str

*# CRITICAL: Verify that the message contains schema before sending to next agent*

*if* *message*.get("desc\_str"):

            logger.info(f"Message contains schema ({len(*message*['desc\_str'])} chars). Sending to Decomposer.")

*else*:

            logger.warning("No schema in message! This will cause Decomposer to fail.")

*message*["send\_to"] = DECOMPOSER\_NAME

*return* *message*

    def **get\_schema**(*self*, *db\_id*: str) -> Dict[str, Any]:

        """

        Get schema information for a PostgreSQL database.

        Args:

            db\_id: Database ID

        Returns:

            Dictionary with schema information

        """

*# Create a new connection to the database*

*try*:

            conn = psycopg2.connect(

*host*=*self*.pg\_host,

*port*=*self*.pg\_port,

*user*=*self*.pg\_user,

*password*=*self*.pg\_password,

*dbname*=*db\_id*

            )

*# Create cursor for executing queries*

            cursor = conn.cursor()

*# Get list of tables*

            cursor.execute("""

                SELECT table\_name

                FROM information\_schema.tables

                WHERE table\_schema = 'public'

                ORDER BY table\_name;

            """)

            tables = [row[0] *for* row *in* cursor.fetchall()]

            schema\_info = {"tables": {}}

*# Get columns for each table*

*for* table *in* tables:

                cursor.execute("""

                    SELECT column\_name, data\_type,

                           is\_nullable, column\_default,

                           (SELECT EXISTS (

                               SELECT 1 FROM information\_schema.table\_constraints tc

                               INNER JOIN information\_schema.constraint\_column\_usage ccu

                               ON tc.constraint\_name = ccu.constraint\_name

                               WHERE tc.constraint\_type = 'PRIMARY KEY'

                               AND tc.table\_name = c.table\_name

                               AND ccu.column\_name = c.column\_name

                           )) as is\_primary

                    FROM information\_schema.columns c

                    WHERE table\_name = %s AND table\_schema = 'public'

                    ORDER BY ordinal\_position;

                """, (table,))

                columns = []

*for* col *in* cursor.fetchall():

                    column\_name, data\_type, is\_nullable, default, is\_primary = col

*# Get sample values for this column (up to 5)*

*try*:

                        cursor.execute(f"""

                            SELECT "{column\_name}"

                            FROM "{table}"

                            WHERE "{column\_name}" IS NOT NULL

                            LIMIT 5

                        """)

                        sample\_values = [str(val[0]) *for* val *in* cursor.fetchall()]

*except* Exception *as* e:

*# If error getting samples, provide empty list*

                        sample\_values = []

                    columns.append({

                        "name": column\_name,

                        "type": data\_type,

                        "nullable": is\_nullable == "YES",

                        "default": default,

                        "primary": is\_primary,

                        "samples": sample\_values

                    })

                schema\_info["tables"][table] = columns

*# Get foreign keys*

            cursor.execute("""

                SELECT

                    tc.table\_name AS source\_table,

                    kcu.column\_name AS source\_column,

                    ccu.table\_name AS target\_table,

                    ccu.column\_name AS target\_column

                FROM information\_schema.table\_constraints tc

                JOIN information\_schema.key\_column\_usage kcu

                  ON tc.constraint\_name = kcu.constraint\_name

                JOIN information\_schema.constraint\_column\_usage ccu

                  ON ccu.constraint\_name = tc.constraint\_name

                WHERE tc.constraint\_type = 'FOREIGN KEY'

                AND tc.table\_schema = 'public'

                ORDER BY tc.table\_name, kcu.column\_name;

            """)

            foreign\_keys = []

*for* row *in* cursor.fetchall():

                source\_table, source\_column, target\_table, target\_column = row

                foreign\_keys.append({

                    "source\_table": source\_table,

                    "source\_column": source\_column,

                    "target\_table": target\_table,

                    "target\_column": target\_column

                })

            schema\_info["foreign\_keys"] = foreign\_keys

*# Close cursor and connection*

            cursor.close()

            conn.close()

*return* schema\_info

*except* Exception *as* e:

            logger.error(f"Error getting schema for {*db\_id*}: {e}")

*return* {"tables": {}, "foreign\_keys": []}

    def **format\_schema**(*self*, *schema\_info*: Dict[str, Any]) -> Tuple[str, str]:

        """

        Format schema information as a human-readable string.

        Args:

            schema\_info: Dictionary with schema information

        Returns:

            Tuple of (desc\_str, fk\_str)

        """

*# Format tables and columns*

        desc\_parts = []

*for* table\_name, columns *in* *schema\_info*["tables"].items():

            columns\_str = []

*for* col *in* columns:

*# Format type info with primary key if applicable*

                type\_info = col["type"]

*if* col["primary"]:

                    type\_info += " PRIMARY KEY"

*# Format sample values*

                samples\_str = ""

*if* col["samples"]:

*# Ensure samples are properly formatted for display*

                    formatted\_samples = []

*for* sample *in* col["samples"]:

*# For string types, add quotes*

*if* isinstance(sample, str) and not sample.isdigit():

                            formatted\_samples.append(f"'{sample}'")

*else*:

                            formatted\_samples.append(str(sample))

                    samples\_str = f". Value examples: [{', '.join(formatted\_samples)}]"

                columns\_str.append(f"({col['name']} {type\_info}{samples\_str})")

*# Format in a style similar to const.py*

            desc\_parts.append(f"# Table: {table\_name}\n[")

*for* i, col\_str *in* enumerate(columns\_str):

*if* i < len(columns\_str) - 1:

                    desc\_parts.append(f"  {col\_str},")

*else*:

                    desc\_parts.append(f"  {col\_str}")

            desc\_parts.append("]")

        desc\_str = "\n".join(desc\_parts)

*# Format foreign keys*

        fk\_parts = []

*for* fk *in* *schema\_info*["foreign\_keys"]:

            fk\_parts.append(

                f"{fk['source\_table']}.{fk['source\_column']} references "

                f"{fk['target\_table']}.{fk['target\_column']}"

            )

        fk\_str = "\n".join(fk\_parts)

*return* desc\_str, fk\_str

ДОДАТОК Д

*#!/usr/bin/env python*

"""

Extensions for the BIRD-UKR dataset.

Provides PostgreSQL-compatible agents for the Ukrainian BIRD dataset.

"""

*import* os

*import* logging

*import* time

*import* json

*import* psycopg2

*from* psycopg2.extras *import* RealDictCursor

*from* typing *import* Dict, List, Any, Tuple, Optional

*from* utils.pg\_selector *import* PostgreSQLSelector

*from* core.agents *import* Decomposer, Refiner, BaseAgent

*from* core.const\_ukr *import* SELECTOR\_NAME, DECOMPOSER\_NAME, REFINER\_NAME, refiner\_template\_ukr

*from* core.utils *import* parse\_sql\_from\_string

*from* core.api *import* safe\_call\_llm

*from* utils.pg\_connection *import* get\_pool\_connection, return\_connection

logger = logging.getLogger(\_\_name\_\_)

class PostgreSQLRefiner(BaseAgent):

    """

    PostgreSQL Refiner for executing SQL queries against PostgreSQL databases.

    """

    def **\_\_init\_\_**(*self*, *data\_path*: str, *model\_name*: str, *dataset\_name*: str):

        """

        Initialize the PostgreSQL Refiner.

        Args:

            data\_path: Path to the dataset

            model\_name: Name of the model to use

            dataset\_name: Name of the dataset

        """

        super().\_\_init\_\_()

*self*.name = REFINER\_NAME

*self*.data\_path = *data\_path*

*self*.model\_name = *model\_name*

*self*.dataset\_name = *dataset\_name*

*# Get PostgreSQL credentials from environment*

*self*.pg\_user = os.environ.get('PG\_USER', 'postgres')

*self*.pg\_password = os.environ.get('PG\_PASSWORD', '')

*self*.pg\_host = os.environ.get('PG\_HOST', 'localhost')

*self*.pg\_port = os.environ.get('PG\_PORT', '5432')

        logger.info("Initialized PostgreSQL Refiner")

    def **talk**(*self*, *message*: Dict[str, Any]) -> Dict[str, Any]:

        """

        Process a message, execute the SQL query, and refine if needed.

        Args:

            message: The message to process

        Returns:

            The updated message

        """

*# Get relevant data from message*

        db\_id = *message*.get("db\_id", "")

        pred\_sql = *message*.get("pred", "")

        query = *message*.get("query", "")

        desc\_str = *message*.get("desc\_str", "")

        fk\_str = *message*.get("fk\_str", "")

        try\_times = *message*.get("try\_times", 0)

*# Check if there's a final\_sql field and use it if pred\_sql is empty*

*if* not pred\_sql and "final\_sql" in *message*:

            pred\_sql = *message*.get("final\_sql", "")

*# Store the SQL in the pred field for consistency*

*message*["pred"] = pred\_sql

*# Initialize result*

*message*["try\_times"] = try\_times + 1

*# Check if we have all required information*

*if* not db\_id or not pred\_sql:

            logger.warning("Missing db\_id or SQL query")

*message*["send\_to"] = "System"

*return* *message*

*# Try to execute the SQL*

        logger.info(f"Executing SQL query against {db\_id}: {pred\_sql}")

        success, result, error = *self*.\_execute\_sql(db\_id, pred\_sql)

*# Check if we need to refine*

*if* not success and try\_times < 3:

*# SQL execution failed, need to refine*

            logger.info(f"SQL execution failed: {error}. Refining...")

            new\_sql = *self*.\_refine(

*query*=query,

*evidence*=*message*.get("evidence", ""),

*desc\_str*=desc\_str,

*fk\_str*=fk\_str,

*sql*=pred\_sql,

*sqlite\_error*=error

            )

*if* new\_sql and new\_sql != pred\_sql:

*# We got a refined SQL query, try again*

*message*["pred"] = new\_sql

*message*["fixed"] = True

*message*["send\_to"] = REFINER\_NAME  *# Send back to self for another try*

*return* *message*

*# No need to refine or refinement failed/not possible*

*message*["send\_to"] = "System"

*message*["execution\_result"] = result *if* success *else* None

*message*["execution\_error"] = error *if* not success *else* None

*return* *message*

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

        """

        Execute SQL query against a PostgreSQL database.

        Args:

            db\_id: Database identifier

            sql: SQL query to execute

        Returns:

            Tuple of (success, result, error\_message)

        """

*try*:

*# Create connection*

            conn = psycopg2.connect(

*dbname*=*db\_id*,

*user*=*self*.pg\_user,

*password*=*self*.pg\_password,

*host*=*self*.pg\_host,

*port*=*self*.pg\_port

            )

*# Create cursor*

            cursor = conn.cursor(*cursor\_factory*=RealDictCursor)

*# Set timeout (30 seconds)*

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

*# Execute query*

            start\_time = time.time()

            cursor.execute(*sql*)

            result = cursor.fetchall()

            execution\_time = time.time() - start\_time

*# Convert result to list of dictionaries*

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

*# Close cursor and connection*

            cursor.close()

            conn.close()

            logger.info(f"SQL executed successfully in {execution\_time:.2f}s. Rows: {len(result\_list)}")

*return* True, result\_list, None

*except* Exception *as* e:

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

*return* False, None, str(e)

    def **\_refine**(*self*, *query*: str, *evidence*: str, *desc\_str*: str,

*fk\_str*: str, *sql*: str, *sqlite\_error*: str) -> str:

        """

        Refine the SQL query based on the error.

        Args:

            query: Natural language query

            evidence: Additional evidence

            desc\_str: Database schema description

            fk\_str: Foreign key description

            sql: Original SQL query that failed

            sqlite\_error: Error message from PostgreSQL

        Returns:

            Refined SQL query

        """

*try*:

*# Create prompt for the LLM*

            prompt = refiner\_template\_ukr.format(

*question*=*query*,

*evidence*=*evidence*,

*desc\_str*=*desc\_str*,

*fk\_str*=*fk\_str*,

*sql*=*sql*,

*sqlite\_error*=*sqlite\_error*,

*exception\_class*="PostgreSQLError"

            )

*# Call the LLM*

            response = safe\_call\_llm(prompt)

*# Parse the SQL from the response*

            new\_sql = parse\_sql\_from\_string(response)

*if* new\_sql and new\_sql != *sql*:

                logger.info(f"Refined SQL: {new\_sql}")

*return* new\_sql

*else*:

                logger.warning("Failed to extract valid refined SQL")

*return* *sql*

*except* Exception *as* e:

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

*return* *sql*

def **load\_pg\_selector**(\**args*, \*\**kwargs*) -> PostgreSQLSelector:

    """

    Create and return a PostgreSQL-compatible Selector for the BIRD-UKR dataset.

    Args:

        \*args: Additional positional arguments

        \*\*kwargs: Additional keyword arguments

    Returns:

        PostgreSQLSelector instance

    """

    logger.info("Loading PostgreSQL Selector for BIRD-UKR dataset")

    selector = PostgreSQLSelector(\**args*, \*\**kwargs*)

*return* selector

def **load\_bird\_ukr\_extensions**(*data\_path*: str, *model\_name*: str, \*\**kwargs*) -> Dict[str, Any]:

    """

    Load the BIRD-UKR extension agents.

    Args:

        data\_path: Path to the dataset

        model\_name: Name of the model to use

        \*\*kwargs: Additional keyword arguments

    Returns:

        Dictionary mapping agent names to agent instances

    """

    logger.info("Loading BIRD-UKR extensions")

*# Create the agents*

    selector = load\_pg\_selector(

*data\_path*=*data\_path*,

*tables\_json\_path*=*kwargs*.get('tables\_json\_path'),

*model\_name*=*model\_name*,

*dataset\_name*="bird-ukr"

    )

    decomposer = Decomposer(

*model\_name*=*model\_name*,

*dataset\_name*="bird" *# Use standard BIRD for decomposer as the prompt is similar*

    )

    decomposer.name = DECOMPOSER\_NAME

    refiner = PostgreSQLRefiner(

*data\_path*=*data\_path*,

*model\_name*=*model\_name*,

*dataset\_name*="bird-ukr"

    )

*# Create dictionary of agents*

    agents = {

        SELECTOR\_NAME: selector,

        DECOMPOSER\_NAME: decomposer,

        REFINER\_NAME: refiner

    }

*return* agents

ДОДАТОК Е

*#!/usr/bin/env python*

"""

Ukrainian constants and prompts for BIRD-UKR dataset.

"""

*# Database settings - EXPLICIT CONFIGURATION*

DB\_TYPE = "postgresql"  *# THIS IS A POSTGRESQL DATABASE - NOT SQLITE*

USE\_POSTGRESQL = True

USE\_UKRAINIAN = True   *# ALL TABLES AND COLUMNS ARE IN UKRAINIAN*

*# PostgreSQL specific syntax*

PG\_TRUE = "TRUE"       *# PostgreSQL uses TRUE not 1*

PG\_FALSE = "FALSE"     *# PostgreSQL uses FALSE not 0*

*# Agent names - keeping English names for compatibility*

SYSTEM\_NAME = "System"

SELECTOR\_NAME = "Selector"

DECOMPOSER\_NAME = "Decomposer"

REFINER\_NAME = "Refiner"

*# Engine names*

ENGINE\_TOGETHER = "meta-llama/Llama-3.3-70B-Instruct-Turbo"

ENGINE\_DEFAULT = ENGINE\_TOGETHER

*# Templates for agents working with Ukrainian PostgreSQL database (with English Prompts)*

*# --- Enhanced Selector Prompt for BIRD-UKR (English Instructions) ---*

selector\_template\_ukr = """

You are an experienced PostgreSQL database administrator. Your task is to analyze a user question (in Ukrainian) and a PostgreSQL database schema (with Ukrainian table/column names) to determine the relevant tables and columns.

[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

[Input Schema Format]

You will receive the database schema in the following format:

- `# Table: <table\_name>` (Ukrainian table name)

- `Columns:`

  - `<column\_name> (<data\_type>) [PRIMARY KEY] [Value examples: ...]` (Ukrainian column name)

- Additionally, there might be a `Foreign Keys:` section listing relationships.

[Instructions]

1.  Carefully analyze the user question (provided in Ukrainian) to understand what information is needed.

2.  Identify \*only the tables and columns\* that are directly necessary to answer the question. Be precise and minimalist.

3.  If two tables are linked by a foreign key and both are needed, include both.

4.  Prioritize tables containing columns that match keywords in the Ukrainian question.

5.  Consider the `Value examples` to determine relevance.

6.  If a table has 10 or fewer columns and is relevant, mark it as `"keep\_all"`.

7.  If a table is completely irrelevant, mark it as `"drop\_all"`.

8.  For relevant tables with more than 10 columns, list \*only\* the necessary columns (including primary and foreign keys if needed for joins).

9.  Return the result in JSON format, using the original Ukrainian table and column names as keys/values where appropriate.

[Example]

==========

【DB\_ID】 університет

【Schema】

# Таблиця: студенти

Стовпці:

  id\_студента (INTEGER) PRIMARY KEY

  імʼя (VARCHAR)

  прізвище (VARCHAR)

  дата\_народження (DATE)

  id\_факультету (INTEGER) Value examples: [1, 2, 3]

# Таблиця: факультети

Стовпці:

  id\_факультету (INTEGER) PRIMARY KEY

  назва\_факультету (VARCHAR) Value examples: ["Комп'ютерних наук", "Економічний"]

  декан (VARCHAR)

  рік\_заснування (INTEGER)

# Таблиця: курси

Стовпці:

  id\_курсу (INTEGER) PRIMARY KEY

  назва\_курсу (VARCHAR)

  кредити (INTEGER)

  id\_викладача (INTEGER)

【Foreign keys】

студенти.id\_факультету = факультети.id\_факультету

【Question】

Скільки студентів навчається на факультеті Комп'ютерних наук? (How many students are studying in the Computer Science faculty?)

【Evidence】

(none)

【Answer】

```json

{{

  "студенти": ["id\_студента", "id\_факультету"],

  "факультети": ["id\_факультету", "назва\_факультету"],

  "курси": "drop\_all"

}}

```

==========

Now your turn:

【DB\_ID】 {db\_id}

【Schema】

{desc\_str}

【Foreign keys】

{fk\_str}

【Question】

{question}

【Evidence】

{evidence}

【Answer】

"""

*# --- Enhanced Decomposer Prompt for BIRD-UKR (English Instructions) ---*

decomposer\_template\_ukr = """

Given a PostgreSQL database schema (with Ukrainian names), additional 【Evidence】, and a 【Question】 in Ukrainian, your task is to decompose the question into logical sub-queries (if necessary) and generate a single, correct PostgreSQL SQL query that answers the question.

[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

[Input Schema Format]

The schema is provided in a text format including table names (Ukrainian), columns (Ukrainian), their types, primary keys (PK), and value examples. Foreign keys (FK) are provided separately.

[Important PostgreSQL Constraints]

1.  \*\*`SELECT` Only:\*\* Generate only `SELECT` queries. \*Do not\* include `CREATE TABLE`, `INSERT`, `UPDATE`, `DELETE`.

2.  \*\*Quoting Identifiers:\*\* Use double quotes (`"`) for Ukrainian table and column names \*if\* they contain spaces, special characters, or are PostgreSQL reserved words. Simple names (e.g., `ід\_студента`) do not require quotes, but quoted names (e.g., `"Ім'я Студента"`) are safer if unsure.

3.  \*\*Syntax:\*\* Use standard PostgreSQL syntax for functions (date, string, aggregation) and operators.

4.  \*\*Precision:\*\* Select only the columns truly needed for the answer.

5.  \*\*Aggregation:\*\* If using `MAX`, `MIN`, `AVG`, `SUM`, `COUNT`, ensure `GROUP BY` is used correctly.

6.  \*\*String Literals:\*\* Use single quotes (`'`) for string literals. To escape apostrophes within strings, double them (`''`) as in: WHERE назва = 'Не з''явився'

7.  \*\*Boolean Literals:\*\* Use TRUE/FALSE for boolean values, not 1/0.

[Example]

==========

【DB\_ID】 університет

【Schema】

# Таблиця: студенти

Стовпці:

  id\_студента (INTEGER) PRIMARY KEY

  імʼя (VARCHAR)

  прізвище (VARCHAR)

  дата\_народження (DATE)

  id\_факультету (INTEGER) Value examples: [1, 2, 3]

# Таблиця: факультети

Стовпці:

  id\_факультету (INTEGER) PRIMARY KEY

  назва\_факультету (VARCHAR) Value examples: ["Комп'ютерних наук", "Економічний"]

  декан (VARCHAR)

  рік\_заснування (INTEGER)

# Таблиця: курси

Стовпці:

  id\_курсу (INTEGER) PRIMARY KEY

  назва\_курсу (VARCHAR)

  кредити (INTEGER)

  id\_викладача (INTEGER)

【Foreign keys】

студенти.id\_факультету = факультети.id\_факультету

【Question】

Скільки студентів навчається на факультеті Комп'ютерних наук? (How many students are studying in the Computer Science faculty?)

【Evidence】

(none)

【Answer】

```sql

SELECT COUNT(\*) FROM студенти WHERE id\_факультету = 1

```

==========

Now your turn:

【DB\_ID】 {db\_id}

【Schema】

{desc\_str}

【Foreign keys】

{fk\_str}

【Question】

{question}

【Evidence】

{evidence}

【Answer】

"""

refiner\_template\_ukr = """

You are given a PostgreSQL database schema with Ukrainian table and column names, a question in Ukrainian, and a SQL query with an error. Your task is to fix the SQL query.

[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: This is a PostgreSQL database where tables already exist - DO NOT include any CREATE TABLE or INSERT statements in your response.

【Database schema】

{desc\_str}

【Foreign keys】

{fk\_str}

【Question】

{question}

【Evidence】

{evidence}

SQL query with error:

```sql

{sql}

```

PostgreSQL error message:

```

{sqlite\_error}

```

Please analyze the error and provide a corrected SQL query that follows PostgreSQL syntax.

Common issues to check:

1. Identifier quoting - PostgreSQL uses double quotes for Ukrainian identifiers, not backticks

2. Character encoding - Ensure proper handling of Ukrainian characters in string literals

3. PostgreSQL-specific syntax for functions, operators, and aggregations

4. Case sensitivity - PostgreSQL identifiers are case-sensitive when quoted

5. Boolean literals - PostgreSQL uses TRUE/FALSE, not 1/0

6. Table and column names - Use Ukrainian names, not English translations

Write only the corrected SELECT query. Do not include any table creation or data insertion statements."""

*# Keep the English versions available as well*

selector\_template = selector\_template\_ukr

decomposer\_template = decomposer\_template\_ukr

refiner\_template = refiner\_template\_ukr