МАГІСТЕРСЬКА РОБОТА

МР. ІПм - 34.00.00.000 ПЗ

Група ІПм-23-1

Гаврилишин Олександр

2024

Івано-Франківський національний технічний університет нафти і газу

Інститут інформаційних технологій

Кафедра інженерії програмного забезпечення

Гаврилишин Олександр Володимирович

(прізвище, ім'я, по батькові)

УДК 004.942

(індекс)

МАГІСТЕРСЬКА РОБОТА

Моделі та методологія виявлення антипатернів SQL

(назва роботи)

Інженерія програмного забезпечення

(назва освітньої програми)

121 - Інженерія програмного забезпечення

(шифр і назва спеціальності)

Гаврилишин О.В.

(підпис, ініціали та прізвище здобувача освітнього ступеня)

Науковий керівник Храбатин Роман Ігорович, к.ф.-м.н., доцент

(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Допущено до захисту

Завідувач кафедри

доц. Бандура В.В.

(посада) (підпис) (дата) (ініціали та прізвище)

Нормоконтроль

доц. Вовк Р.Б.

(посада) (підпис) (дата) (ініціали та прізвище)

Робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

Івано-Франківськ – 2024

Івано-Франківський національний технічний університет нафти і газу

Інститут інформаційних технологій

Кафедра інженерії програмного забезпечення

Освітній рівень магістр

Спеціальність 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедрою **ІПЗ**
доц. В.В. Бандура
“ 04 ” вересня 2024 р.

ЗАВДАННЯ

НА МАГІСТЕРСЬКУ РОБОТУ СТУДЕНТУ

Гаврилишину Олександру Володимировичу

(прізвище, ім’я, по-батькові)

1. Тема магістерської роботи “ Моделі та методологія виявлення антипатернів SQL ”

керівник проекту (роботи) Храбатин Роман Ігорович, к.ф.-м.н., доцент
затверджені наказом закладу вищої освіти від “ 22 ” листопада 2024 р. № 781/7

2. Строк подання студентом проекту (роботи) 15 грудня 2024 р.

3. Вихідні дані до проекту (роботи) Теоретичні концепції та формальні моделі побудови та функціонування програмних технологій антипаттернів

4. Зміст розрахунково - пояснювальної записки(перелік питань, які потрібно розробити)

1. Аналіз предметної області виявлення антипатернів в SQL
2. Моделі, методи та інструменти виявлення антипатернів в SQL
3. Дослідження інструментів в області виявлення антипатернів
4. Імплементація моделей та методів для виявлення антипатернів SQL

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)

1. Приклад таблиці клієнтів у базі даних із первинним ключем cID (рис. 1.1)
2. Приклад зв’язувальної таблиці, яка пов’язує таблицю клієнта з таблицею продуктів (рис. 1.2)
3. Приклад зв’язку «один-до-багатьох» між клієнтом із зовнішнім ключем aID (рис. 1.3)
4. Діаграма зв’язку сутності (ERD) у нотації, що показує зв’язок «один до багатьох» (рис. 1.4)
5. Діаграма зв’язків (рис. 1.5)

6. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата
Перевірка на плагіат	доц., к.т.н. Вовк Р.Б.	

7. Дата видачі завдання 04 вересня 2024 р.

Керівник _____
(підпис)

Завдання прийняв до виконання_____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назви етапів магістерської роботи	Строк виконання етапів роботи	Примітка
1	Підбір і вивчення літератури по темі магістерської роботи	15.09.2024	виконано
2	Аналіз концепцій та алгоритмів предметної області	29.09.2024	виконано
3	Аналіз предметної області виявлення антипатернів в SQL	15.10.2024	виконано
4	Моделі, методи та інструменти виявлення антипатернів в SQL	08.11.2024	виконано
5	Дослідження інструментів в області виявлення антипатернів	20.11.2024	виконано
6	Імплементація моделей та методів для виявлення антипатернів SQL	01.12.2024	виконано
7	Затвердження пояснювальної записки роботи завідувачем кафедри	15.12.2024	виконано

Студент – магістр _____
(підпис)

Керівник роботи _____
(підпис)

АНОТАЦІЯ

Магістерська робота: 81 с., 42 рис., 55 джерел.

Тема: Моделі та методологія виявлення антипатернів SQL

Об'єкт дослідження: процеси написання та оптимізації SQL-запитів у базах даних.

Мета роботи: розробити моделі, методи та інструмент для автоматичного виявлення антипатернів у SQL-коді з метою підвищення продуктивності, підтримуваності та якості баз даних.

Предмет дослідження: методи та інструменти для виявлення SQL-антипатернів у коді.

Результати дослідження

В роботі розроблено підхід до автоматизованого виявлення SQL-антипатернів, що включає класифікацію типових антипатернів і створення інструменту для їх автоматичного виявлення. Запропоновані методи та інструмент можуть бути адаптовані для роботи з різними СУБД, що дозволяє використовувати їх у широкому спектрі завдань, пов'язаних із оптимізацією баз даних.

Висновок

Представлено підходи до імплементації моделей і методів для виявлення антипатернів у SQL, що охоплює як класифікацію антипатернів, так і розробку інструменту для їх автоматичного виявлення.

SQL-АНТИПАТЕРНИ, БАЗИ ДАНИХ, ПРОДУКТИВНІСТЬ SQL, ВИЯВЛЕННЯ АНТИПАТЕРНІВ, ОПТИМІЗАЦІЯ БАЗ ДАНИХ, ІНСТРУМЕНТ ДЛЯ ВИЯВЛЕННЯ АНТИПАТЕРНІВ, ОПТИМІЗАЦІЯ ЗАПИТІВ

ABSTRACT

Master Thesis: 81 pp., 42 fig., 55 sources.

Thesis Subject: Models and methodology of using SQL antipatterns

Research object: processes of creation and optimization of SQL queries in databases.

The goal of the work: to develop models, methods and tools for automatic detection of anti-patterns in SQL code in order to improve performance, compatibility and quality of databases.

Research topic: methods and tools for SQL antipatterns in code.

Research results

Your work develops an approach to automated SQL antipatterns, which includes the classification of types of antipatterns and the creation of a tool for their automatic creation. The proposed methods and tool can be adapted to work with an additional DBMS, which allows them to be used in a wide range of tasks related to database optimization.

Conclusion

Approaches to the implementation of models and methods for detecting antipatterns in SQL are presented, covering both the classification of antipatterns and the development of a tool for their automatic detection.

SQL ANTI-PATTERNS, DATABASES, SQL PERFORMANCE, ANTI-PATTERN DETECTION, DATABASE OPTIMIZATION, ANTI-PATTERN DETECTION TOOL, QUERY OPTIMIZATION

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

SQL - Structured Query Language

RDBMS - Relational Database Management System

ORM - Object-Relational Mapping

ACID - Atomicity, Consistency, Isolation, Durability

OLTP - Online Transaction Processing

MVCC - Multi-Version Concurrency Control

UDF - User-Defined Function

SP - Stored Procedure

CTE - Common Table Expression

XML - eXtensible Markup Language

JSON - JavaScript Object Notation

BCNF - Boyce-Codd Normal Form

SQLi - SQL Injection

ISAM - Indexed Sequential Access Method

UUID - Universally Unique Identifier

AI - Auto-Increment

TPC - Transaction Processing Performance Council (for benchmarking)

ВСТУП

Актуальність теми.

В умовах стрімкого зростання обсягів даних і складності інформаційних систем забезпечення продуктивності та якості баз даних набуває особливої важливості. SQL-антипатерни, тобто неефективні або некоректні шаблони написання SQL-коду, призводять до зниження продуктивності, надмірного використання ресурсів та виникнення труднощів у підтримці баз даних. Антипатерни можуть впливати на загальну стабільність та швидкодію додатків, особливо в умовах високих навантажень, що стає критичним фактором для великих компаній, фінансових установ, сервісів з великими обсягами даних та інших організацій, де затримки обробки запитів є неприйнятними.

Часто виявлення і виправлення антипатернів є ручною та трудомісткою задачею, яка вимагає глибоких знань і досвіду роботи з SQL, що обмежує можливість застосування ефективних підходів до оптимізації баз даних на ранніх етапах їхнього розвитку. Сучасні технології автоматизації та штучного інтелекту можуть істотно полегшити цей процес, дозволяючи автоматично аналізувати SQL-код і виявляти потенційні антипатерни, пропонуючи більш оптимальні варіанти для їхнього виправлення.

Актуальність теми також посилюється в контексті розвитку хмарних технологій, де вимоги до продуктивності та економії ресурсів стоять на першому плані. Оскільки робота з хмарними базами даних потребує ретельного контролю використання ресурсів, виявлення та усунення антипатернів є важливим аспектом зниження вартості експлуатації таких сервісів.

Таким чином, розробка інструментів для автоматичного виявлення антипатернів у SQL та створення рекомендацій щодо їх виправлення є актуальною задачею, що сприяє підвищенню ефективності та надійності баз

даних, забезпеченню стабільної роботи програмних систем і зниженню експлуатаційних витрат.

Мета дослідження – розробити моделі, методи та інструмент для автоматичного виявлення антипатернів у SQL-коді з метою підвищення продуктивності, підтримуваності та якості баз даних.

Об'єкт дослідження – процеси написання та оптимізації SQL-запитів у базах даних.

Предмет дослідження – методи та інструменти для виявлення SQL-антипатернів у коді.

Завдання дослідження:

- Провести аналіз концепції SQL-антипатернів та їх впливу на продуктивність баз даних.

- Вивчити існуючі інструменти та методи для виявлення антипатернів.

- Розробити класифікацію типових SQL-антипатернів та їх характеристик.

- Реалізувати інструмент для виявлення антипатернів

- Оцінити ефективність запропонованого інструменту та інтерфейсу для зручної взаємодії з користувачем.

Методи дослідження

- Теоретичний аналіз — для визначення особливостей SQL-антипатернів, вивчення теоретичних основ баз даних.

- Аналіз літератури та інструментів — для огляду наявних засобів виявлення SQL-антипатернів.

- Методи програмної інженерії — для розробки та реалізації інструменту.

- Емпіричні методи — для тестування і оцінки ефективності розробленого інструменту.

Наукова новизна отриманих результатів

Розроблено підхід до автоматизованого виявлення SQL-антипатернів, що включає класифікацію типових антипатернів і створення інструменту для

їх автоматичного виявлення. Запропоновані методи та інструмент можуть бути адаптовані для роботи з різними СУБД, що дозволяє використовувати їх у широкому спектрі завдань, пов’язаних із оптимізацією баз даних.

Практичне значення результатів

Розроблений інструмент для виявлення антипатернів може бути використаний розробниками баз даних, аналітиками та адміністраторами для поліпшення продуктивності SQL-запитів і забезпечення надійності баз даних. Запропоновані моделі та методи є основою для подальшого вдосконалення інструментів автоматизованого аналізу SQL-коду.

Структура магістерської роботи. Робота складається зі вступу, трьох розділів та висновків. Загальний обсяг роботи становить 81 сторінку, і містить 42 рисунки, список використаних джерел із 55 найменувань.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ВИЯВЛЕННЯ АНТИПАТЕРНІВ В SQL

1.1. Особливості концепції антипатернів в SQL

Ми живемо в епоху інформації, коли людей бомбардують нескінченною кількістю даних, інформації та повідомлень. У природі нашого суспільства постійно отримувати цю інформацію, будь то через соціальні медіа, газети, телебачення, Інтернет або тим чи іншим способом. Великі обсяги даних, також відомі як «великі дані», щодня збираються з різних джерел і зберігаються в базах даних для використання та зберігання.

Термін «великі дані» використовується для опису збору, зберігання, аналізу та розповсюдження великих обсягів даних. Оскільки кількість даних, що генеруються та обробляються, продовжує збільшуватися, існує потреба в ефективному зберіганні цих даних. Цю потребу можна задовольнити шляхом впровадження баз даних і систем управління базами даних (СУБД). Базу даних можна описати як набір збережених даних, організованих у структурований спосіб. Його можна класифікувати на багато різних типів, серед яких реляційні (SQL, Oracle), об’єктно-орієнтовані (ObjectDatabases, ObjectStore) і NoSQL (MongoDB, Cassandra) бази даних. СУБД — це складна програмна система, яка надає можливість створювати, читати, оновлювати та видаляти дані шляхом їх упорядкованого зберігання в базі даних. Таким чином, управління базою даних.

Здебільшого ми взаємодіємо з цими базами даних, навіть не усвідомлюючи цього. З появою Інтернету важливість баз даних тільки зросла. Величезні обсяги даних генеруються різними джерелами, наприклад пристроями в домені Інтернету речей, а також такими програмами, як веб-сайти та програми для мобільних телефонів. Ці пристрої, програми, а також люди можуть отримувати та маніпулювати даними з цих баз даних. Це корисно для таких завдань, як візуалізація даних або виконання аналізу

даних. Дія отримання даних із баз даних зазвичай називається «запитом» даних із бази даних і може здійснюватися кількома способами.

Одним із найпоширеніших способів запиту до бази даних є використання мови запитів до бази даних (DBQL). DBQL – це група мов, які дозволяють користувачам створювати запити, які мають можливість доступу, зміни або видалення даних із бази даних. Однією з найпоширеніших DBQL є мова структурованих запитів (SQL), яка десятиліттями була домінуючою в індустрії баз даних, хоча впровадження інших більш сучасних, стандартизованих мов, таких як SPARQL, GraphQL, OQL (Object Query Language) або використання об'єктно-реляційних відображень зросло в останні роки.

Об’єктно-реляційне відображення (ORM) — це альтернативний метод взаємодії з базою даних. Використання ORM є звичайною практикою, яка дозволяє розробникам використовувати об’єкти, написані як структури або класи, і використовувати парадигму об’єктно-орієнтованого програмування для взаємодії з даними в базі даних. Основною перевагою використання ORM є його здатність значно зменшити обсяг роботи, необхідний для розробки додатків, здатних взаємодіяти з даними в базі даних. ORM скорочує час розробки, оскільки розробникам не потрібно писати необроблені SQL-запити з нуля, а натомість зосереджуються на дизайні програми чи логіці.

Незважаючи на це, SQL залишається однією з найпопулярніших мов пошуку даних завдяки своїй простоті, продуктивності та легкості впровадження. Згідно з опитуванням розробників 2021 року, проведеним Stack Overflow, SQL є четвертою за популярністю мовою серед усіх мов програмування, сценаріїв і розмітки, випереджаючи її лише Python, HTML/CSS і JavaScript [31].

Іншим фактором, який підкреслює важливість SQL, є те, що більшість сучасних популярних ORM і DBQL на ринку засновані на SQL. Однак важливо знати, що ORM і DBQL не є заміною SQL. Хоча ORM можуть запропонувати кращий досвід розробки, використовуючи абстракції високого

рівня, у них є свої недоліки. Накладні витрати на код, створений ORM, є поширеною скаргою серед розробників, оскільки це сповільнює продуктивність програми та ускладнює її підтримку. По-друге, ORM іноді є ненадійними щодо коду SQL, який вони генерують, що впливає на продуктивність. Отже, розробникам все ще потрібно знати SQL для випадків, коли ORM не може забезпечити необхідну продуктивність або функціональність.

З цієї причини важливо, щоб SQL все ще навчали початківців розробників. Вивчення SQL повинно супроводжуватися вивченням різних концепцій, які його оточують. Ці концепції включають базову теорію баз даних, а також більш практичні аспекти, такі як синтаксис SQL. На щастя, переважна більшість розробників знайомі з SQL у тій чи іншій формі.

Важливо добре розуміти синтаксис SQL, оскільки він є основою для всіх запитів SQL, і це не те, що більшість людей може легко зрозуміти. Перший поширений тип помилки - це синтаксична помилка. Синтаксична помилка вказує на те, що введений запит не є дійсним запитом SQL. У цьому випадку СУБД згенерує повідомлення про помилку, оскільки вона не може виконати запит. У результаті помилка виявляється під час виконання, і зазвичай її легко виправити.

Другий тип помилок - це семантичні. Ці помилки класифікуються як часті помилки, коли було введено дійсний запит SQL, але повернутий результат є неправильним для завдання. На відміну від синтаксичних помилок, розробнику не потрібно виправляти синтаксис запиту. Однак якщо вони пропустять певну деталь у запиті, це може призвести до неправильних даних.

Останній тип помилки можна вважати меншою мірою помилкою, а більшою мірою контрпродуктивною та неефективною групою запитів, які можуть бути тісно пов’язані із семантичними помилками. Це так звані запахи коду SQL або антишаблони, які є поширеними помилками в запитах SQL, які здаються хорошим і достатньо ефективним рішенням конкретного завдання.

Однак ці запити, як правило, викликають логічні або функціональні помилки, проблеми з продуктивністю або вразливі місця в безпеці. Запити, уражені антишаблонами, також можуть бути крихкими та схильними до помилок, що ускладнює підтримку коду. Набір результатів запиту, що містить антишаблон, часто правильний або здається правильним. Однак антишаблони шкідливі, оскільки вони роблять результуючу програму чи оператор неефективними, важчими для підтримки або легшими для неправильного читання.

Перша книга про антишаблони SQL була опублікована лише в 2010 році. З того часу було опубліковано лише декілька інших дослідницьких робіт на цю тему. Поточне дослідження антишаблонів головним чином зосереджено на вивченні поширеності, впливу, еволюції та проблем продуктивності, пов’язаних з антишаблонами SQL, а також виявлення антишаблонів у існуючих кодових базах. Однак ми вважаємо, що запобігання антишаблонам є більш ефективною стратегією, ніж виявлення шаблонів у робочому коді. Оскільки антишаблони SQL, здається, не є основною темою в навчанні SQL, студенти можуть не знати про існування та ефект антишаблонів. Наша мета цього дослідження полягає в тому, щоб підвищити обізнаність новачків про антишаблони SQL і надати їм інструменти для їх вирішення.

Структурована мова запитів (SQL) — це мова баз даних для взаємодії з реляційними базами даних. Попередні дослідження [2] виявили різні бар'єри у вивченні SQL. Ці бар'єри, у поєднанні з неповними знаннями, часто призводять до того, що користувачі SQL пишуть помилкові запити [20, 4, 32]. Дослідження показують, що це може призвести до антипатернів (запахів коду), які шкодять якості програмного забезпечення [17, 24]. Антипатерни також впливають на навчання, оскільки студенти можуть використовувати їх, не усвідомлюючи цього. Результат запиту, що містить антипатерн, часто є правильним або здається правильним. Однак антипатерни шкідливі, оскільки

вони роблять результуючу програму або оператор неефективними, складнішими в обслуговуванні та легшими для неправильного тлумачення.

Антипатерни для мов програмування вперше згадано у 1995 році [16]. Для баз даних загалом і SQL зокрема, антипатерни були визначені у 2010 році [15]. Раннє виявлення антипатернів може допомогти розробникам уникнути технічного боргу, оскільки написання «поганого» SQL може призвести до низької продуктивності або помилкових результатів, як показано в [17, 24]. Ось чому більшість досліджень антипатернів SQL зосереджено на їх виявленні [5, 11, 30]. Однак запобігання антипатернам є ефективнішою стратегією, ніж виявлення патернів у коді на етапі виробництва. Оскільки антипатерни SQL, здається, не є основною темою в навчанні SQL, студенти можуть не знати про існування та вплив антипатернів. Наша мета в цьому дослідженні — підвищити обізнаність про антипатерни SQL для новачків і надати їм інструменти для їх вирішення.

Концепція антипатернів в SQL має ряд особливостей, які відрізняють її від антипатернів в інших сферах програмування:

1. Прихованість.

Антипатерни в SQL часто не призводять до явних помилок. Запит може виконуватись і повертати коректні результати, але робити це неефективно або створювати проблеми в майбутньому. Це ускладнює їх виявлення, особливо для новачків, які можуть не помітити проблем з продуктивністю або складністю підтримки коду.

2. Пов'язаність з продуктивністю.

Багато антипатернів в SQL безпосередньо впливають на продуктивність запитів. Це може призвести до значного уповільнення роботи бази даних, особливо при великих обсягах даних. Наприклад, використання SELECT * замість вибору конкретних стовпців, відсутність індексів, надмірне використання підзапитів - все це може негативно вплинути на швидкість виконання запитів.

3. Складність виявлення.

Виявлення антипатернів в SQL може бути складним завданням, оскільки вони часто залежать від контексту та конкретної ситуації. Для ефективного виявлення антипатернів потрібні глибокі знання SQL, розуміння принципів роботи баз даних та досвід оптимізації запитів.

4. Вплив на навчання.

Оскільки антипатерни SQL не завжди очевидні, студенти можуть використовувати їх, не усвідомлюючи негативних наслідків. Це підкреслює важливість включення теми антипатернів в навчальні програми з SQL, щоб навчити студентів розпізнавати та уникати їх.

5. Різноманітність.

Існує велика кількість антипатернів в SQL, які охоплюють різні аспекти написання запитів: від вибору даних до обробки NULL значень та використання агрегатних функцій. Це робить задачу виявлення та виправлення антипатернів ще більш складною.

Розуміння особливостей антипатернів в SQL є важливим для розробників, які працюють з базами даних. Це дозволяє писати більш ефективний, читабельний та підтримуваний код, уникаючи потенційних проблем з продуктивністю та складністю.

1.2. Теоретичні відомості про бази даних і системи управління базами даних

База даних, або БД, — це організований набір структурованої інформації або даних, які зазвичай зберігаються в електронному вигляді в комп’ютерній системі. Управління базою даних зазвичай покладається на окрему систему, яка називається системою керування базами даних, або скорочено СУБД, яка полегшує взаємодію з даними для своїх користувачів за допомогою набору програм, які отримують доступ до даних. Система баз даних відноситься до даних, СУБД і будь-яких програм, які з нею пов’язані. Це часто скорочується до просто бази даних. Основна мета системи баз

даних — надати користувачам загальний огляд даних. Це означає, що система приховує певні аспекти зберігання та обслуговування даних.

Сьогодні хмарні бази даних і автономні бази даних відкривають нові можливості щодо збору, зберігання, керування та використання даних. Однак, незважаючи на ці нові технології, реляційні бази даних все ще є найпопулярнішим вибором для всіляких проектів [36]. Тому в цьому дослідженні ми зосередимося лише на реляційних базах даних.

Бази даних розвиваються з часом. Це відбувається, коли дані додаються чи видаляються, або коли змінюється структура певних таблиць. Сукупність даних, що зберігаються в базі даних у певний момент часу, називається екземпляром бази даних. Структуру бази даних зазвичай називають схемою бази даних, яка визначає загальний дизайн бази даних.

Модель даних — це набір концептуальних інструментів для представлення даних, зв’язків даних, семантики даних і вимог до узгодженості, які лежать під структурою бази даних. Існує багато різних моделей даних. Наприклад, реляційні бази даних використовують реляційну модель.

1.2.1. Структура таблиці

Дані в найпоширеніших типах баз даних, що використовуються сьогодні, часто описуються в рядках і стовпцях серії таблиць, щоб полегшити обробку даних і пошук. Потім дані можна легко використовувати, керувати ними, змінювати, оновлювати, контролювати та впорядковувати. У термінології бази даних кожен рядок називається записом. Іноді запис також називають об’єктом або сутністю. Записи в таблиці — це об’єкти, які вас цікавлять , наприклад клієнти в базі даних магазину або продукти на складі продуктів. Ім'я таблиці зазвичай вибирається для представлення типу об'єкта, який вона містить. Таблиця клієнтів, наприклад, матиме назву «клієнт», тоді як таблиця замовлень матиме назву «замовлення». Правила іменування таблиць можуть відрізнятися в різних розробників. Деякі розробники

вважають за краще використовувати імена у множині, а інші – в однині. Однак більшість розробників використовують назви таблиць в однині, щоб вказати, що таблиця містить лише один тип запису. Порядок рядків усередині таблиці не має значення, тобто порядок, у якому записи з'являються в рядках таблиці, часто не має значення. Стовпець у таблиці називається полем або атрибутом і представляє одне значення для певного запису. Поле має певний тип, наприклад, ціле число, число з плаваючою точкою або текст. Іншими словами, тип поля представляє тип даних, які ви зберігаєте. Інколи поле може бути необов'язковим, що дозволяє залишати його порожнім, коли воно не потрібне. Ми називаємо таке поле полем із значенням nullable. Іноді поле має бути унікальним. Тобто два записи таблиці з абсолютно однаковим значенням для певного поля не допускаються. Підсумовуючи, таблиця — це набір записів, які, у свою чергу, складаються з полів або атрибутів, що містять певні значення. Ці значення представляють різні типи інформації для певного запису, який іноді може бути нульовим або унікальним.

customer			
cID	cName	street	city
⋮			
855	Bo	Kastelenstraat	Eindhoven
856	Vincent	Zangstraat	Weert
857	George	Bekelaan	Budel
858	Perry	Pannekoekstraat	Rotterdam
⋮			

Рис. 1.1. Приклад таблиці клієнтів у базі даних із первинним ключем cID

Кожна таблиця, що зберігається в базі даних, має первинний ключ, який служить унікальним ідентифікатором цієї таблиці. У більшості випадків це випадковий ідентифікатор, який не має значущого значення за межами бази даних. Приклад такої таблиці можна знайти на рисунку 1.1 , де показано таблицю клієнтів. Первинним ключем у цій таблиці є ідентифікатор клієнта

(cID), який є автоматично збільшеним числом, яке не має значення поза цією базою даних. Однак іноді первинний ключ є інтуїтивно вибраною властивістю об'єкта, який описується в таблиці. Наприклад, ми можемо видалити стовпець cID із рисунка 1.1 і замінити його номером соціального страхування особи. Ми знаємо, що це число унікальне для кожної людини, тому ми можемо використовувати його як первинний ключ.

У деяких випадках первинний ключ складається з кількох стовпців, це називається складеним первинним ключем. Цей тип ключа, наприклад, використовується, коли існує зв’язок «багато-до-багатьох» між двома таблицями, які об’єднані третьою таблицею, яку зазвичай називають таблицею об’єднання, з’єднання або таблицею зв’язування. Тоді ця таблиця матиме складений первинний ключ, що складається з первинних ключів пов’язаних таблиць. На рисунку 1.2 наведено приклад таблиці зв’язків. Ця таблиця пов’язує клієнта з продуктом за допомогою первинного ключа таблиці клієнтів (cID) і первинного ключа таблиці продуктів (pID). Первинним ключем цієї таблиці закупівель є пара складених ключів (cID, pID).

purchase	
cID	**pID**
⋮	
855	22
855	48
855	46
856	87
⋮	

Рис. 1.2. Приклад зв’язувальної таблиці, яка пов’язує таблицю клієнта з таблицею продуктів. Первинним ключем є пара (cID, pID)

1.2.2. Відношення між таблицями бази даних

Таблиці в реляційній базі даних можуть бути пов’язані одна з одною. Ці так звані зв’язки можна встановити за допомогою обмежень зовнішнього

ключа. Між таблицями можуть виникати три типи зв’язків. Перший – це стосунки один на один.

Відношення «один до одного» існує, коли одна сутність в одній таблиці пов’язана точно з однією сутністю в іншій таблиці. Таблиця показує взаємозв'язок один-до-одного між клієнтом і таблицею адрес. Це означає, що клієнт має одну адресу, і кожна адреса представляє одного клієнта. Відношення «один-до-одного» використовується нечасто, оскільки цей тип зв’язку рідко є природним способом моделювання ситуації.

Припустімо, що до бази даних додано іншого клієнта, який проживає за тією самою адресою, що й клієнт з ідентифікатором 855, тоді до таблиці адрес буде додано повторювану адресу, яка має лише інший ідентифікатор адреси. Одним із можливих застосувань цього зв’язку є випадки, коли частина інформації сутності, описаної в деякій таблиці, є необов’язковою.

Таким чином, замість того, щоб мати в одній великій таблиці кілька полів, що допускають значення NULL, ви можете розумно розділити її на обов’язкову таблицю та необов’язкову таблицю.

customer	
cID	cName
⋮	
855	Bo
856	Vincent
857	George
858	Perry
⋮	

address				
aID	street	city	postalcode	cID
		⋮		
10	Kastelenstraat	Eindhoven	5899CK	855
11	Zangstraat	Weert	4201OK	856
12	Bekelaan	Budel	1337AB	857
13	Pannekoekstraat	Rotterdam	4401EB	858
		⋮		

Рис. 1.3. Приклад зв’язку «один-до-багатьох» між клієнтом із зовнішнім ключем aID

Кращий спосіб моделювання відношення полягає у використанні відношення «один до багатьох». Зв’язок «один-до-багатьох» існує, коли одна сутність в одній таблиці пов’язана з однією або кількома сутностями в іншій таблиці. У нашому прикладі клієнт усе ще буде пов’язаний з однією адресою,

але тепер адреса може бути пов’язана з кількома клієнтами. Приклад цього співвідношення наведено на рисунках 1.3 і 1.4.

Рис. 1.4. Діаграма зв’язку сутності (ERD) у нотації, що показує зв’язок «один до багатьох»

1.2.3. Підвищення продуктивності SQL за допомогою індексів бази даних

Індекси в базах даних використовуються для підвищення продуктивності SQL-запитів. Індекс — це копія даних із вибраних стовпців певної таблиці, призначена для надзвичайно ефективного пошуку. Індекс можна використовувати для швидкого пошуку значення стовпця таблиці. Індекси SQL важливі, оскільки вони можуть значно покращити продуктивність запитів SQL. За відсутності індексів SQL-сервер мав би сканувати всю таблицю, щоб знайти потрібні дані. Це може зайняти багато часу, особливо для великих таблиць із великою кількістю рядків і стовпців, до яких часто звертаються. SQL-сервер може швидко знайти потрібні дані за допомогою індексів замість того, щоб сканувати всю таблицю. Більшість баз даних автоматично створюватимуть індекси для первинних і зовнішніх ключів.

Однак адміністратор бази даних відповідає за створення індексів для інших стовпців, які часто використовуються в запитах SQL. Індекси не видно користувачеві. Інша причина використання індексів полягає в тому, що ви знаєте, що впорядкуєте набір результатів за певним стовпцем. Якщо таблиця велика, операція ORDER BY буде дуже повільною. У цьому випадку корисно створити індекс для цього стовпця.

1.2.4. Альтернатива SQL: об'єктно-реляційне відображення

Object Relational Mapper (ORM) — це інструмент, який відображає об'єкти на об'єктно-орієнтованій мові програмування в таблиці реляційної бази даних. Метою ORM є забезпечення рівня абстракції між об'єктно-орієнтованою мовою програмування та реляційною базою даних. Цей рівень абстракції дозволяє використовувати об'єктно-орієнтовану мову програмування з реляційною базою даних. ORM відображає об'єкти на об'єктно-орієнтованій мові програмування в таблиці реляційної бази даних, що дозволяє нам використовувати об'єктно-орієнтовану мову для створення, читання, оновлення та видалення даних із бази даних.

ORM використовують SQL у фоновому режимі для виконання операцій з базою даних. Це означає, що коли використовується ORM, згенерований SQL зазвичай не видимий і не доступний для користувача. Однак можна побачити SQL, створений ORM. Щоб побачити SQL-запит, ми можемо скористатися функцією журналювання, яка надається більшістю ORM. Функція журналювання роздрукує SQL, згенерований ORM, на консоль.

1.3. Детальний огляд та характеристики антипатернів в SQL

У програмуванні шаблони проектування є найкращим практичним способом вирішення поширених проблем у програмуванні. Вони часто використовуються, щоб прискорити розробку, а також зробити код більш зручним для обслуговування. Прикладом шаблону проектування є шаблон Singleton, який використовується для того, щоб гарантувати створення лише однієї позиції класу. Існують і небажані шаблони, яких краще уникати. Ці так звані антишаблони є неефективними способами вирішення проблем і можуть призвести до більшої кількості проблем, ніж вони вирішують [16]. У цьому розділі ми розглянемо та обговоримо всі відомі антишаблони для SQL. Спочатку ми визначимо ці антишаблони, а потім надамо простий для розуміння приклад, у якому продемонстровано кожен антишаблони. Це

допоможе читачеві повністю зрозуміти, що ¬таке антишаблон. Ми також розглянемо деякі з найпоширеніших рішень для їх запобігання. Ми заохочуємо читачів повернутися до цього розділу, коли це необхідно.

Кожен антипаттерн ми продемонструємо на прикладі. Це робиться за допомогою попередньо визначеної схеми, яка буде представлена в цьому розділі. Схема є дещо адаптованою версією, приклад схеми наведено на рис. 1.5.

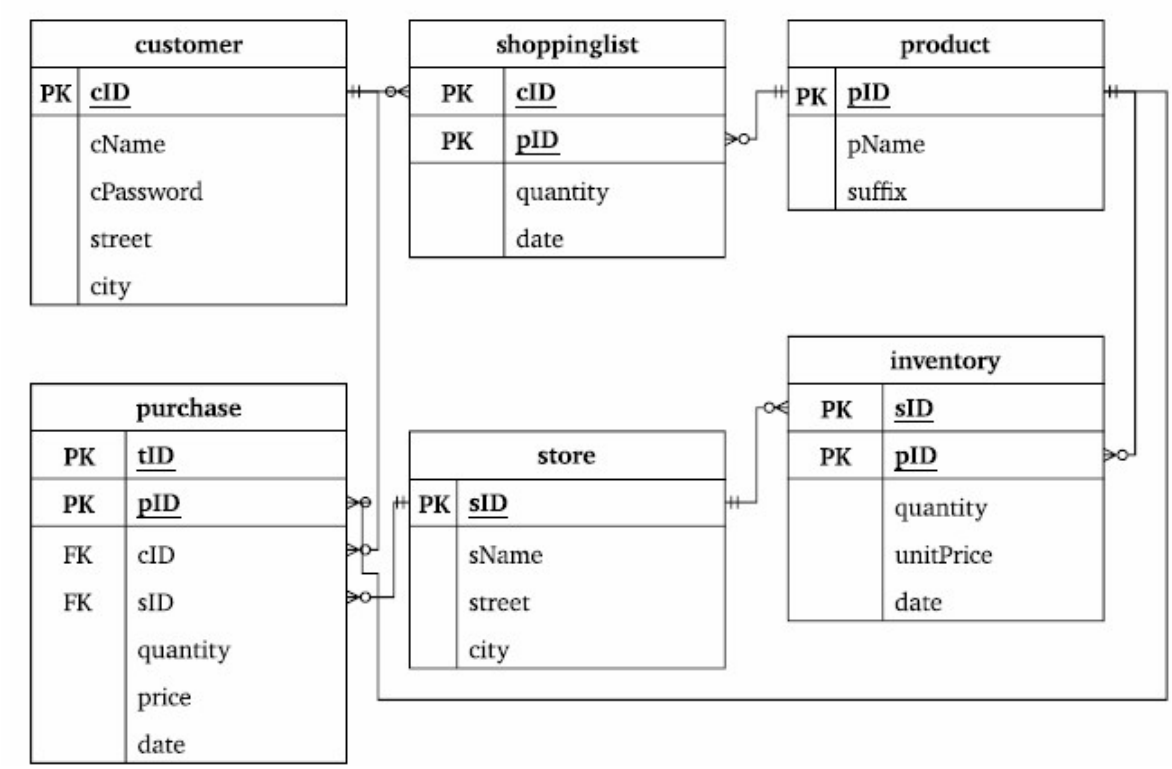


Рис. 1.5. Діаграма зв’язків

У цьому розділі наведено підсумок усіх відомих антишаблонів. Для кожного антишаблону ми надамо короткий вступ із поясненнями антишаблону, приклад коду з використанням схеми, поданої на рисунку 1.5 і, якщо це можливо, рішення для виправлення антишаблону.

1.3.1. Антипатерн “Неоднозначні групи”

Цей антипатерн виникає, коли розробники неправильно використовують команду агрегації GROUP BY. Це може призвести до

помилкових результатів. Кожен стовпець у операторі SELECT запиту повинен мати один рядок із значеннями на групу рядків, що також називається правилом одного значення. Тепер для стовпців у агрегаті GROUP BY це гарантовано, оскільки воно повертає рівно одне значення на групу, незалежно від того, скільки рядків відповідає групі. Для інших команд SQL, таких як MAX(), MIN(), AVG(), це також призведе до єдиного значення для кожної групи, тому це також гарантовано. Сервер бази даних, з іншого боку, не може бути настільки впевненим щодо будь-якого іншого поля, зазначеного в операторі SELECT. Це не завжди гарантує, що однакове значення для інших стовпців з'являється в кожному рядку групи.

```
SELECT cID, cName, pID, MAX(date)
FROM customer JOIN shoppinglist USING (cID)
GROUP BY cID, cName;
```

Рис. 1.6. Приклад антипатерну неоднозначних груп

Рисунок 1.6 показує базовий приклад цього антишаблону. У цьому прикладі, оскільки таблиця списку покупок ідентифікує численні продукти для конкретного клієнта, існує кілька різних значень ідентифікатора продукту для даного ідентифікатора клієнта. Немає способу виразити всі значення ідентифікатора продукту в запиті на групування, який зводиться до одного рядка для кожного клієнта.

Приклад вище можна легко виправити. Ми повинні гарантувати, що кожен стовпець має одне значення. Це можна зробити, включивши ідентифікатор продукту в пункт GROUP BY. На рисунку 1.17 показано, як це робиться.

```
SELECT cID, cName, pID, MAX(date)
FROM customer JOIN shoppinglist USING (cID)
GROUP BY cID, cName, pID;
```

Рис. 1.7. Рішення для антипатерну неоднозначних груп

1.3.2. Антипатерн “Страх перед невідомим”

У SQL стовпці можна залишити порожніми. Це призводить до того, що атрибут певного рядка має значення NULL. SQL вважає NULL спеціальним значенням, відмінним від нуля, false, true або порожнього рядка. Це навпаки, більшість розробників знають з інших мов, особливо новачки сприйнятливі до цього антипатерну.

Код, показаний на рисунку 1.8 запитує назву продукту та стовпці з таблиці продуктів, де суфікс не дорівнює NULL. Можна подумати, що це призведе до того, що всі рядки мають суфікс, однак це не так. Будь-яке порівняння з NULL повертає невідоме, невірне чи хибне. Таким чином, цей запит не повертає жодних даних.

```
SELECT pName, suffix
FROM product
WHERE suffix <> NULL;
```

а) Приклад порівняння NULL

```
SELECT UNIX_TIMESTAMP(date) + 86400
FROM shoppinglist;
```

б) Приклад додавання NULL

Рис. 1.8. Два приклади антипатерну «Страх перед невідомим».

Є багато інших прикладів цього антипатерну. Інший приклад наведено в запиті рис. 1.18 б), де ми перетворюємо дані у формат часу Unix і додаємо до нього число 86400. Можна було б подумати, що коли певна дата дорівнює NULL, вона просто поверне 86400. Однак це не так, оскільки дата дорівнює NULL, таким чином час Unix дорівнює NULL, а тому додавання 86400 невідоме.

Просте виправлення запиту а) було б використовувати оператор IS NOT NULL, а виправлення для запиту б) полягає у використанні спеціальної функції, яка приймає вираз і альтернативне значення та повертає значення виразу, якщо вираз не є NULL, інакше повертає альтернативне значення. Майже кожен діалект SQL має таку функцію, наприклад, функція ISNULL() у SQL Server або функція IFNULL() у MySQL. У цьому випадку ми

встановлюємо альтернативне значення 0, тобто якщо дані мають значення NULL, ми починаємо з 0. Рішення відображаються на рисунку 1.9.

```
SELECT pName, suffix
FROM product
WHERE suffix IS NOT NULL;
```

```
SELECT UNIX_TIMESTAMP(ISNULL(date,
    0)) + 86400
FROM shoppinglist;
```

Рис. 1.9. Два можливі рішення для антипатерну «Страх перед невідомим»

1.3.3. Неявні стовпці

Під час написання запиту, який потребує великої кількості стовпців, розробники часто вирішують використовувати селектор підстановки SQL (*). Це означає, що кожен стовпець із таблиці (таблиць), указаних у реченні FROM, повертається, тобто список стовпців є неявним. Багато в чому це робить запит більш лаконічним. Однак це може мати певні витрати, оскільки набір результатів може бути досить великим для великих таблиць. Зробити запит більш конкретним шляхом явного переліку стовпців може бути неінтуїтивним, але часто корисним, наприклад, для економії місця та мережевих ресурсів.

Припустімо, у нас є завдання знайти всі ідентифікатори клієнтів, ідентифікатори магазинів та ідентифікатори продуктів, а також кількість і ціну в таблиці покупок. Це означало б, що єдиними стовпцями з таблиці покупок, яких немає в цьому запиті, будуть стовпці ідентифікатора покупки та даних. Запит, показаний на рисунку 1.10 містить антипатерн неявних стовпців, оскільки використовує символ підстановки (*). Замість вибору лише стовпців, запитуваних завданням, розробник використовує символ узагальнення, повертаючи результат, який містить і ідентифікатор магазину, і стовпці дати.

```
SELECT *
FROM purchase;
```

Рис. 1.10. Приклад антипатерну неявних стовпців

Запит на рисунку 1.11 показує, що цей антипатерн зустрічається не лише в запитах SELECT, але також може з’являтися в запитах INSERT. Замість надання підмножини стовпців після назви таблиці запит застосовує значення до всіх стовпців у тому порядку, у якому вони вказані в таблиці. Багато розробників віддадуть перевагу такому способу написання запиту, оскільки він набагато коротший.

```
INSERT INTO purchase
VALUES (DEFAULT, 2, 6, 1, 3, 4.99, '2022-2-16 14:44:53');
```

Рис. 1.11. Приклад антипатерну неявних стовпців із використанням INSERT

1.3.4. Антипатерн “Неявні рядки”

Цей антипатерн виникає, коли запит зчитує більше рядків, ніж використовує програма. Це часто трапляється, коли програма застосовує фільтр після запиту даних.

На рисунку 1.12 завдання полягало в тому, щоб вибрати всі назви продуктів і ціни за одиницю, де ціна за одиницю була більшою за 2,99. Однак програма, яка використовує дані, отримані з цього запиту, також застосовує власний фільтр, який обчислює середню ціну за одиницю та відкидає всі продукти, які є нижчими.

```
SELECT pName, unitPrice
FROM products JOIN inventory USING (PID)
WHERE unitPrice > 2.99
```

Рис. 1.12. Приклад антипатерну неявних рядків

Натомість ефективніше написати один запит, який застосовує обидва фільтри, а також внутрішній фільтр із програми. Рисунок 1.13 показує

можливе рішення, у якому підзапит використовується для обчислення середньої ціни за одиницю, що перевищує порогове значення 2,99.

```
SELECT pName, unitPrice
FROM product JOIN inventory USING (pID)
WHERE unitPrice > (
  SELECT AVG(unitPrice)
  FROM inventory
  WHERE unitPrice > 2.99
)
```

Рис. 1.13. Можливе рішення для антипатерну неявних рядків

1.3.5. Антипатерн “Loop to Join”

Замість об’єднання двох таблиць і запиту з об’єднаної таблиці, цикл для об’єднання антипатерну виникає, коли ми спочатку виконуємо запит з однієї таблиці, а потім виконуємо цикл над даними, які вона повертає, щоб отримати значення з іншої таблиці.

```
product_ids = execute_sql(
    """
    SELECT pID
    FROM product
        JOIN inventory USING (pID)
    WHERE unitPrice > 2.99
    """
)

customer_ids = []

for id in product_ids:
    customer_ids.append(
        execute_sql(
            f"""
            SELECT DISTINCT cID
            FROM purchase
            WHERE pID = {id}
            """
        )
    )
```

Рис. 1.14. Код Python, який демонструє антишаблон Loop to Join

Код на рисунку 1.14 показує цей антипатерн і написаний на Python. Зауважте, що цей антипатерн можна застосувати до будь-якої мови програмування, а не лише до Python. Припустімо, що у нас є функція execute_query(), яка приймає необроблений рядок запиту SQL і повертає список. Перший запит повертає всі ідентифікатори продуктів, ціна одиниці яких перевищує 2,99 і зберігає їх у змінній під назвою products_ids. Далі код створює порожній список під назвою customer_ids. Цей список заповнюється за допомогою циклічної структури, яка обробляє всі ідентифікатори продуктів, які ми отримуємо за допомогою першого SQL-запиту. Для кожного ідентифікатора продукту ми виконуємо окремий запит, який повертає ідентифікатор клієнтів, які купили відповідний продукт.

Замість цього наведений вище код Python можна замінити одним SQL-запитом, який використовує оператор IN, який є синтаксисом SQL для зіставлення з набором значень. Ми створюємо цей набір значень ідентифікатора продукту за допомогою підзапиту.

```
SELECT DISTINCT cID
FROM purchase
WHERE pID IN (
    SELECT pID
    FROM product
      JOIN Inventory USING (pID)
    WHERE unitPrice > 2.99
  )
```

Рис. 1.15. Можливе рішення антишаблону Loop to Join

1.3.6. Антипатерн “Not Merging Projection Predicates”

Not Merging Projection Predicates — це антипатерн, який виникає, коли ми надсилаємо кілька запитів, кожен із яких читає підмножину стовпців, необхідних для певного завдання.

```
/* Query 1 */        /* Query 2 */
SELECT cName         SELECT city
FROM customer        FROM customer
```

Рис. 1.16. Приклад антипатерну “Not Merging Projection Predicates”

Припустимо, завдання полягало в тому, щоб отримати всі імена клієнтів і міста. У попередньому запиті ми бачимо два запити, які запитують стовпець із таблиці клієнтів. Натомість ці два запити можна легко об'єднати в один, який показано на рисунку 1.17.

```
SELECT cName, city
FROM customer
```

Рис. 1.17. Можливе рішення для антипатерну "Not Merging Projection Predicates"

1.3.7. Антипатерн "Not Caching"

Коли надсилається кілька синтаксично порівнянних або по суті еквівалентних запитів для отримання даних із бази даних і всі вони мають загальний підвираз, ці пошуки слід кешувати.

```
/* Query 1 */
SELECT sID
FROM products
  JOIN Inventory USING (pID)
GROUP BY pID
HAVING pID IN (
    SELECT pID
    FROM Inventory
    GROUP BY pID
    HAVING AVG(unitPrice) > 5
  )

/* Query 2 */
SELECT pID, MAX(quantity)
FROM product
  JOIN inventory USING (pID)
GROUP BY pID
HAVING pID IN (
    SELECT pID
    FROM Inventory
    GROUP BY pID
    HAVING AVG(unitPrice) > 5
  )
```

Рис. 1.18. Приклад антипатерну "Not Caching"

Запит на рисунку 1.18 показує приклад цього антипатерну. Можна помітити, що обидва запити 1 і 2 видають той самий підзапит, який повертає всі ідентифікатори продукту з ціною одиниці, вищою за 5. Припустімо, що таблиця запасів містить мільярд рядків, що зробить середнє обчислення досить повільним, особливо тому, що це обчислення є обов'язковим для обох запитів. Для зберігання результатів цих великих обчислень можна використовувати кеш. Тоді запит може отримати результати з кешу без повторного обчислення. Зазвичай кешування запитів виконується ORM, базою даних або СУБД, що робить цей антипатерн менш важливим для новачків.

1.3.8. Антипатерн "Poor Man's Search Engine"

Припустимо, ми хочемо шукати слова або речення в нашій базі даних. Перше, що спадає на думку, це використання предикату відповідності шаблону SQL, такого як ключове слово LIKE, для якого ми можемо вказати шаблон. Предикат LIKE можна використовувати із символом узагальнення (%), який відповідає нулю або більше символів. Якщо використовується перед і після ключового слова, воно відповідає будь-якому рядку, що містить це слово. SQL також підтримує регулярні вирази для зіставлення рядків, які починаються з певного шаблону, тому обидва методи виглядають як дуже хороший варіант для повного пошуку.

Однак слід знати, що основною проблемою предикатів зіставлення з шаблоном є їх низька продуктивність. Оскільки вони не можуть використовувати традиційний індекс, вони повинні сканувати кожен рядок вказаних таблиць. Загальна вартість сканування таблиці для цього пошуку дуже висока, оскільки зіставлення шаблону зі стовпцем рядків є дорогою операцією, якщо порівнювати її з іншими методами порівняння, такими як ціла рівність.

Ще одна проблема з простим зіставленням шаблону за допомогою ключового слова LIKE або регулярних виразів полягає в тому, що вони

можуть знайти ненавмисні збіги, що робить результат пошуку неточним або помилковим.

Запит на рисунку 1.19 демонструє приклад пошуку продуктів, у назві яких є слово «кішка». Ці запити слід уникати, якщо розмір таблиці товарів великий. Замість цього слід вибрати спеціалізований метод пошукової системи, деякі з яких навіть є стандартними для певних баз даних або СУБД.

```
SELECT *
FROM product
WHERE pName LIKE "%cat%"
```

Рис. 1.19. Запит, що показує відповідність шаблону

1.3.9. Антипатерн “Readable Password”

Цей антипатерн менше пов’язаний із SQL, а більше з розробкою додатків і баз даних. Небезпечно зберігати паролі та іншу конфіденційну інформацію в текстовому полі бази даних. Цей антипатерн все ще безпосередньо пов’язаний із SQL, оскільки зловмисник може перехопити та прочитати оператори SQL, які розкривають звичайні текстові паролі, як показано в запиті на рисунку 1.20. Крім того, журнали запитів можуть містити ці запити, які можуть витікати у разі порушення даних.

```
customer_id, password = 123, "secretPassword"

data = execute_sql(
    f"""
    SELECT *
    FROM customer
    WHERE cID = {customer_id} AND cPassword = {password}
    """
)
```

Рис. 1.20. Код Python, що показує антипатерн “Readable Password”

Натомість розробники повинні зберігати конфіденційну інформацію, наприклад паролі, за допомогою (надійної) криптографічної хеш-функції. Ці

функції пропонують одностороннє шифрування, тобто після того, як щось хешується, немає простого способу змінити хеш. Єдиним способом визначити значення хешу було б виконати пошук методом грубої сили, генеруючи випадкове слово, хешуючи слово та перевіряючи, чи воно відповідає хешу, що є дуже повільним і громіздким методом. Хеш-функція перетворює вхідний рядок у нерозпізнаний рядок, відомий як хеш. Щоразу, коли нам потрібно порівняти певне значення з хешованим полем, ми можемо хешувати це значення в коді програми та використовувати його в нашому запиті. Це показано на рисунку 1.21.

```
# Hash password such that it becomes unrecognizable
customer_id, password = 123, hash("secretPassword")

data = execute_sql(
    f"""
    SELECT *
    FROM customer
    WHERE cID = {customer_id} AND cPassword = {password}
    """
)
```

Рис. 1.21. Можливе рішення антипатерну “Readable Password”

1.3.10. Антипатерн “Vulnerable Query”

Уразливі запити – це запити, які містять вхідні дані, які не оброблені належним чином. Прикладом може бути введення користувача, з якого введене значення безпосередньо використовується в запиті. Тоді цей запит буде вразливим. Уразливий запит може бути використаний для виконання ін’єкції або виконання зовнішніх команд на сервері бази даних. Це відомо як SQL-ін’єкція. В запиті (рис. 1.22) ми бачимо фрагмент коду клієнта JavaScript, який запитує клієнта на основі параметра GET із поточної URL-адреси. Цей запит схильний до впровадження SQL. SQL-ін’єкція є поширеним вектором атаки. Метою зловмисника є використання слабкої сторони перевірки вхідних даних веб-додатку за допомогою SQL-атаки. За допомогою цієї методики зловмисник може отримати несанкціонований

доступ до даних у базі даних. Зловмисник створить вхідні дані, призначені для впливу на вихід SQL-запиту. Це може спричинити витік або пошкодження даних. Наприклад, якщо зловмисник змінить параметр GET таким чином: ?id="105OR1=1", тоді на сторінці відобразиться вся база даних клієнтів, оскільки запит буде проаналізовано як SELECT * FROM customers WHERE CID = 105 OR 1=1, щоб речення WHERE завжди було істинним. Параметризований запит можна використати для усунення цієї вразливості.

```
// Retrieve url param e.g. ?id=123
const urlParams = new URLSearchParams(window.location.search);

// Get value of id
const cID = urlParams.get("id")

// Use cID in query
const data = `SELECT * FROM customers WHERE cID = ${cID}`;
```

Рис. 1.22. Код JavaScript, що показує шаблон захисту від вразливого запиту

Запит на рис. 1.23 показує параметризований запит selectCustomer, який приймає як параметр ідентифікатор клієнта. Тепер ми можемо викликати цю процедуру за допомогою запиту EXEC SelectAllCustomers @CID= 105;. Зауважимо, що якщо зловмисник змінить рядок параметра GET, виконання запиту призведе до помилки, оскільки тип CID не є цілим числом.

```
CREATE PROCEDURE selectCustomer @cID int
AS
SELECT * FROM customers WHERE cID = @cID
GO;
```

Рис. 1.23. Можливе рішення антипатерну “Vulnerable Query”

Висновки до розділу

В даному розділі проведено всебічний аналіз предметної області виявлення антипатернів у SQL, що дало змогу глибше зрозуміти потенційні

недоліки і помилки, які можуть виникати під час розробки й використання реляційних баз даних. Розглянуто особливості концепції антипатернів у SQL, включно з теоретичними засадами та практичними прикладами їх впливу на продуктивність і безпеку систем управління базами даних (СУБД). Проведений огляд структури таблиць, відношень між ними та можливостей оптимізації за допомогою індексів дозволив виявити важливі аспекти побудови ефективних SQL-запитів. Було також розглянуто альтернативні підходи до реляційної моделі, зокрема об’єктно-реляційне відображення, яке може знизити складність запитів та підвищити їх ефективність у певних контекстах. Описані антипатерни в SQL охоплюють широкий спектр поширених проблем, серед яких:

- Антипатерн "Неоднозначні групи", що спричиняє плутанину при використанні групових функцій;

- Антипатерн "Страх перед невідомим", який проявляється у неефективній обробці NULL-значень;

- Неявні стовпці й рядки, що ускладнюють читабельність і підтримку бази даних;

- Проблеми, пов’язані з неправильним використанням операцій об’єднання (наприклад, "Loop to Join");

- Відсутність кешування і неврахування проекційних предикатів, що знижує продуктивність запитів;

- Антипатерн "Poor Man's Search Engine", що виникає при невдалих спробах створити функціональність пошуку без використання спеціалізованих інструментів;

- Антипатерни, що впливають на безпеку, такі як "Readable Password" і "Vulnerable Query".

Таким чином, цей розділ заклав основу для розуміння важливості виявлення та уникнення антипатернів у SQL.

РОЗДІЛ 2. МОДЕЛІ, МЕТОДИ ТА ІНСТРУМЕНТИ ВИЯВЛЕННЯ АНТИПАТЕРНІВ В SQL

2.1. Аналіз літературних джерел поширеності антипатернів в SQL

У цьому розділі ми почнемо з обговорення впливу і поширеності антипатернів SQL та деякі інструменти для виявлення цих антипатернів.

Дослідження показують нам, що помилки SQL можуть призвести до антипатернів (запахів коду), які шкодять якості програмного забезпечення [4, 17, 24]. Набір результатів запиту, що містить антипатерн, часто правильний або здається правильним. Однак антипатерни шкідливі, оскільки вони роблять результуючу програму чи оператор неефективними, важчими для підтримки та легшими для неправильного читання.

В 2019 році кількісно оцінено вплив антипатернів SQL на продуктивність мобільних додатків [17]. Це дослідження складається з огляду літератури та порівняльного дослідження для дослідження проблемних практик програмування (антипатерни) щодо використання бази даних. Для свого повторного пошуку вони використовували існуючі програми для Android, які використовують локальну базу даних SQLite. Вони зібрали результати вимірювання енергії та часу роботи з одного пристрою, Samsung Galaxy S5 з Android 5.0. Під час огляду літератури вони визначили одинадцять антипатернів SQL. На наступному етапі свого дослідження, під час проведення експерименту, вони виявили, що вісім із цих антипатернів мають значний вплив на час виконання та енергоспоживання операцій локальної бази даних.

З цих восьми два з них особливо виділялися, що навіть кінцеві користувачі не могли помітити ефект. Антипатерн, який вони називають unbatched writes, де послідовність записів до бази даних виконується окремо, а не об’єднується (пакетується) в одну транзакцію, мав такий великий вплив на час виконання, що після виправлення середнє скорочення часу виконання

перевищила 100 мс, що є звичайним порогом для сприйняття людиною затримки. Інший антипатернний позначений цикл для приєднання виникає, коли розробники використовують запит для отримання кількох значень з однієї таблиці, а потім для кожного значення роблять запит до іншої таблиці (використовуючи структуру циклу). Цей запах коду можна виправити за допомогою JOIN, який спочатку об'єднує дві таблиці, а потім надсилає запит із об'єднаної таблиці. Це в середньому також зменшить час роботи більш ніж на 100 мс.

В роботі [24] зосередили свої дослідження на поширеності, впливі та еволюції антипатернів SQL у системах з інтенсивним об'ємом даних. Вони провели емпіричне дослідження, під час якого зібрали 150 проектів з відкритим вихідним кодом і перевірили, чи були в цих проектах якісь випадки зі списку як традиційних, так і SQL-шаблонів. Під час перевірки вони подивилися на поширеність антипатернів зі списку, з'ясували, чи були якісь випадки з традиційними антипатернами та помилками, а також подивилися на еволюцію кожного антипатерну. Дослідження зосереджено на чотирьох антипаттернах, а саме на неоднозначних групах, страху перед невідомим, неявних стовпцях і випадковому виборі. Якщо інструмент перевірити на 150 проектах з відкритим вихідним кодом, стає зрозуміло, що антипатерни SQL справді часто зустрічаються в системах з інтенсивним об'ємом даних. Неявні стовпці та страх перед невідомими антипаттернами є поширеними, а два інших – ні. Додатково проаналізувавши 150 проектів і класифікувавши кожен проект як бізнес, бібліотека, мультимедіа або комунальний, автори виявили, що хоча медіана поширеності для кожної категорії дорівнює нулю, все ще існує значна кількість викидів у категоріях бібліотеки, бізнесу та комунальних послуг. Це викликає занепокоєння, оскільки бібліотеки та утиліти використовуються іншими розробниками в інших проектах як залежності, що робить ці антипатерни також поширеними в цих залежних проектах. Алгоритм асоціації Apriori m використовується, щоб побачити, чи є зв'язки між традиційними та антипатернами SQL.

Аналіз показує, що, незважаючи на те, що ці традиційні запахи незначно співвідносяться, ступінь асоціації, виміряний за допомогою тесту під назвою Cramer's V Test for Association, низький, що означає, що вони не сильно пов’язані. Той самий алгоритм використовується для визначення того, чи антипатерни SQL виникають одночасно з помилками. Показано, що статистично значущого зв’язку немає.

Нарешті, автори провели аналіз виживання, щоб визначити, як довго антипатерни SQL виживають у проекті. Для кожного проекту після 500 комітів створюється знімок із останніх комітів у зворотному напрямку. Це означає, що автор може подивитися на еволюцію проекту. Вони показують, що антипатерни SQL мають більш високу тенденцію до виживання протягом більш тривалого періоду часу порівняно з традиційними антипатернами [24]. Близько 80% антипатернів зберігаються в усіх доступних знімках і майже не привертають уваги під час рефакторингу коду розробниками. Суть цієї статті полягає в тому, що антипатерни SQL переважають серед невеликих зразків проектів, які вивчали дослідники. Незважаючи на те, що це не обов’язково відображає справжню поширеність антипатернів SQL, очевидно, що цій проблемі потрібно приділити більше уваги, щоб запобігти збереженню антипатернів SQL і потенційному впливу на проекти.

2.2. Дослідження інструментів в області виявлення антипатернів

У наведених вище джерелах зазначено, що антипатерни SQL можуть впливати на продуктивність і поширені в різних кодових базах. Щоб позбутися існуючих антипатернів, їх необхідно знайти в першу чергу. Це робиться за допомогою виявлення. Існують також різні статті про виявлення антипатернів SQL. В роботі [25] представлений інструмент для ідентифікації антипатернів у запитах SQL, знайдених у вихідному коді Java. Інструмент реалізує статичний аналіз як вбудованих запитів SQL, так і схеми бази даних

і даних у базі даних. Із шести антипатернів, перелічених у книзі Карвіна [15], чотири з них реалізовані цим інструментом.

На рисунку 2.1 представлено загальний огляд основних компонентів даного інструменту. Основним компонентом є SQL Extractor, який витягує SQL-запити, вбудовані в код Java. Вхідними даними цього компонента є вихідний код Java, а вихідними - список SQL-операторів з деякою додатковою метаінформацією, такою як ланцюжок викликів, через який формується оператор, або місцезнаходження у вихідному коді, де йвін надсилається до бази даних.

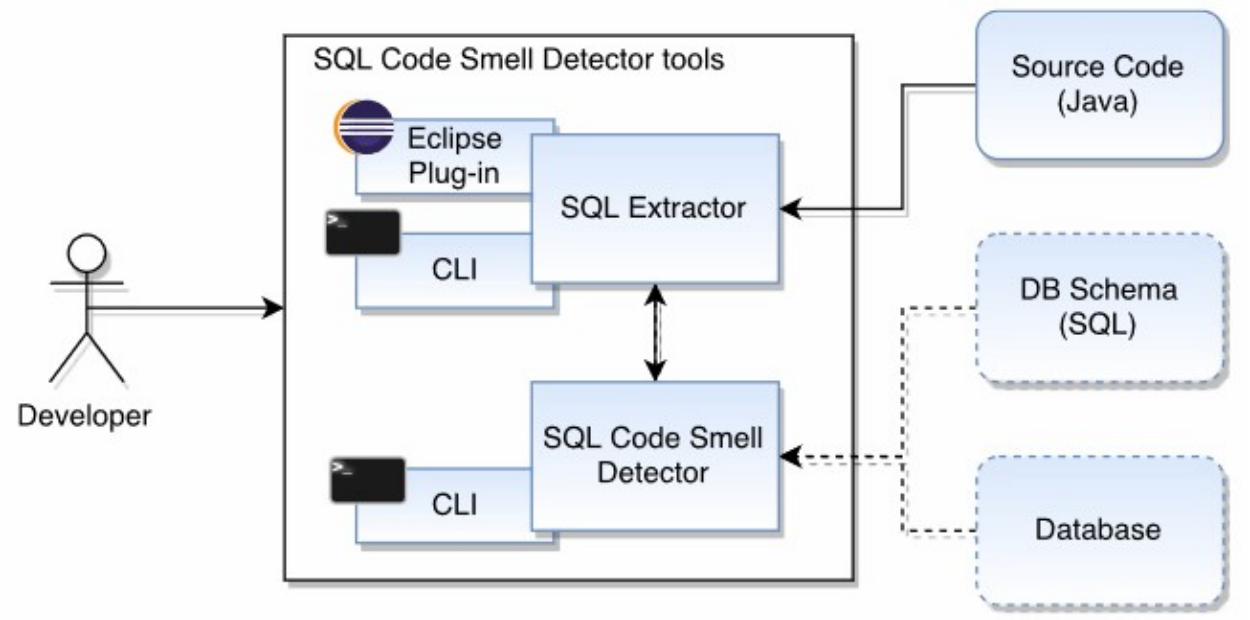


Рис. 2.1. Огляд інструментів виявлення антипатернів коду SQL

За замовчуванням інструмент може виконувати легку реалізацію екстрактора SQL, яка спирається на внутрішньопроцедурне розпізнавання рядків, що працює з AST, наданим Eclipse JDT. Оскільки інструмент постачається з плагіном Eclipse як інтерфейсом користувача, це рішення надає простий спосіб інтеграції аналізатора в IDE та виконує швидкий початковий аналіз, який може швидко представити результати розробнику. Попередні результати показують, що цей механізм може з великим успіхом ідентифікувати місця у вихідному коді, де SQL-оператори надсилаються до

бази даних, і бути здатним витягувати рядок запиту, коли його основна частина побудована в межах методу.

У зразковій програмі з відкритим кодом для управління проектами, яка використовує JDBC і має близько 130 тисяч рядків коду на Java, Plandora6, екстрактор ідентифікував 424 унікальні SQL-оператори в 62 класах DAO. Конструкція інструменту дозволяє легко замінити цей компонент іншими механізмами вилучення SQL шляхом реалізації обгортки, якщо це необхідно. Більш точний механізм може краще виконувати вилучення більшої кількості SQL-операторів, що призводить до більш точного аналізу, але може негативно вплинути на час виконання, що також може вплинути на зручність використання.

Інша стаття, в якій представлені методи виявлення різних антипаттернів це дослідження [18]. Замість того, щоб бути інструментом, як у попередній статті, автори обрали більш теоретичний підхід. SAND можна вважати більш розширюваною структурою для аналізу антипатернів з використанням нового набору абстракцій. Пропущені або майбутні антипатерни можна додати в майбутньому, якщо словниковий запас абстракцій достатній. Прототип інструменту, який реалізує SAND, є швидким, ефективним і, що найважливіше, дуже точним.

Метою статті [5] є вилучення та аналіз шаблонів із журналу запитів до бази даних з акцентом на антипатерни. Поширений метод виявлення антипатернів, як ми бачили в двох попередніх статтях, вимагає доступу до програмного забезпечення, яке генерує запити. Автори стверджують, що тоді знадобиться доступ до всіх систем, що працюють з базою даних, що практично неможливо. Натомість у статті розглядаються попередні запити, які зберігаються в журналі запитів. Важливо аналізувати журнали запитів, оскільки для зацікавлених сторін дуже важливо знати, як використовується велика база даних. Пошук шаблонів у цьому журналі може допомогти в цьому. Однак антипатерни можуть фальсифікувати такий аналіз. Тому ми хотіли б виявити їх у журналі та, якщо можливо, виправити. У статті

спочатку дається визначення поняття патерн і використовується поняття скелетного дерева в цьому визначенні. Скелетне дерево виходить із синтаксичного дерева шляхом заміни всіх змінних у листових вузлах на заповнювачі. Тоді шаблон запиту є трійкою, що містить скелетні піддерева, і, нарешті, шаблон визначається як послідовність шаблонів запиту. Потім автори використовують два антипатерни як приклади та створюють для кожного антипатерну свій шаблон запиту. Вони стверджують, що цю структуру можна розширити, щоб врахувати майбутні антипатерни.

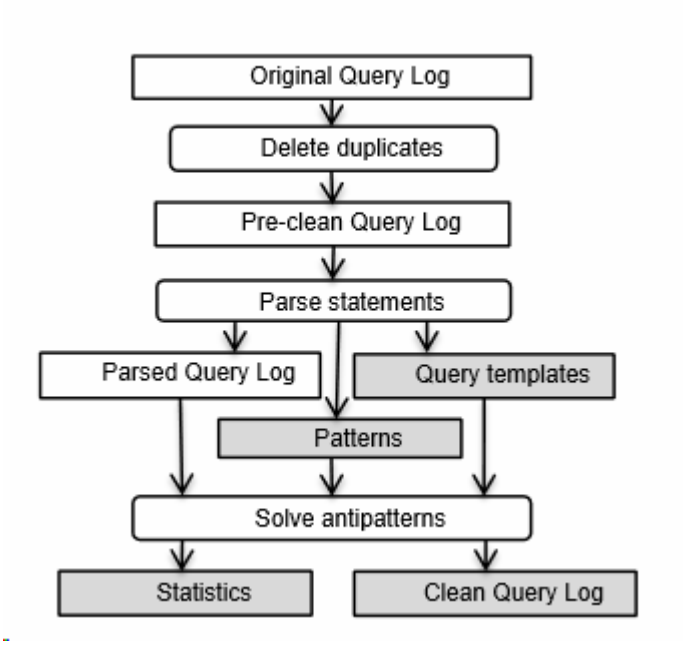


Рис. 2.2. Алгоритм виявлення антипатернів в SQL

На рисунку 2.2 зображено алгоритм виявлення антипатернів у SQL-запитах. Він складається з таких кроків:

- Original Query Log (Оригінальний журнал запитів): На цьому етапі збираються всі SQL-запити, які виконуються в системі. Це може бути лог бази даних або інше джерело інформації.

- Delete duplicates (Видалити дублікати): З журналу запитів видаляються всі повторювані запити, щоб уникнути зайвої обробки.

- Pre-clean Query Log (Попередньо очищений журнал запитів): На цьому етапі виконується попереднє очищення журналу запитів, наприклад, видалення коментарів та форматування запитів.

- Parse statements (Розбір запитів): Кожен SQL-запит розбирається на складові частини (SELECT, FROM, WHERE тощо) для подальшого аналізу.

- Parsed Query Log (Розібраний журнал запитів): Результат розбору запитів зберігається у вигляді структурованих даних.

- Query templates (Шаблони запитів): На основі розібраних запитів створюються шаблони, які представляють загальну структуру запитів.

- Patterns (Патерни): Визначаються патерни, які відповідають типовим антипатернам в SQL-запитах (наприклад, N+1 запити, використання SELECT *, відсутність індексів).

- Solve antipatterns (Вирішення антипатернів): За допомогою визначених патернів аналізується розібраний журнал запитів та виявляються потенційні антипатерни.

- Statistics (Статистика): Збирається статистика про виявлені антипатерни, така як кількість та типи антипатернів.

- Clean Query Log (Очищений журнал запитів): Журнал запитів, в якому виправлено виявлені антипатерни.

Основна ідея алгоритму полягає в тому, щоб:

1. Спочатку розібрати SQL-запити на складові частини.

2. Потім створити шаблони запитів, які дозволять виявити загальні структури.

3. Визначити патерни, які відповідають типовим антипатернам.

4. Застосувати ці патерни до розібраних запитів для виявлення потенційних проблем.

Цей алгоритм може бути використаний для автоматичного виявлення та виправлення антипатернів в SQL-запитах, що дозволить покращити продуктивність та ефективність роботи з базами даних.

Тепер, щоб отримати ці антипатерни з журналу запитів, автори спочатку видаляють дублікати та видаляють синтаксично неправильні запити за допомогою синтаксичного дерева. Далі вони витягують піддерева SELECT, FROM і WHERE та їхні скелетні форми. Тепер вони можуть використовувати формальні визначення антипатернів, щоб виявити їх із журналу запитів.

Ще один спосіб виявлення антипатернів у SQL-запитах — це методи класифікації тексту [30]. Дослідження показує як підходи до машинного навчання можна використати для пошуку антипатернів у запитах SQL, підходячи до проблеми як до проблеми класифікації тексту. Спочатку створюється великий набір даних SQL-запитів із каталогу SkyServer. З цих електронних запитів видаляються всі терміни, пов’язані зі схемою. Потім автори зіставляють кожен запит із антипатерном або відсутнім антипатерном, якщо запит не містить жодного. Потім вони використовують word2vec, групу моделей, що використовуються для створення вбудовування слів, для кодування SQL-запитів. Нарешті, згорточна нейронна мережа навчена класифікувати антипатерни із запитів SQL. Вони заявляють, що їхня модель досить точна, з точністю 83,2%, і що вона може перевершити інше програмне забезпечення.

Нарешті, ми дослідимо інструмент під назвою SQLCheck [11], ланцюжок інструментів для автоматичного виявлення та виправлення антипатернів у програмах баз даних. Метою їх техніки є поєднання аналізу коду з аналізом даних. Це означає, що інструмент поєднує аналіз необроблених SQL-запитів із базовими даними та моделями баз даних. Це означає, що, окрім антипатернів запитів, цей інструмент може перевіряти логічні, фізичні антипатерни та антипатерни даних. Інструмент починається з вилучення контексту із запитів за допомогою аналізатора запитів. Після цього він використовує аналізатор даних для вилучення контексту з таблиць бази даних. Потім інструмент використовує набір правил для визначення точок доступу в заданих запитах на основі контексту програми. Ці правила є

функціями загального призначення, які використовують переваги загального контексту програми. Потім буде повідомлено про виявлені антипатерни. Далі вони оцінюють свій інструмент і показують, що їхній інструмент має високу точність і відкликання.

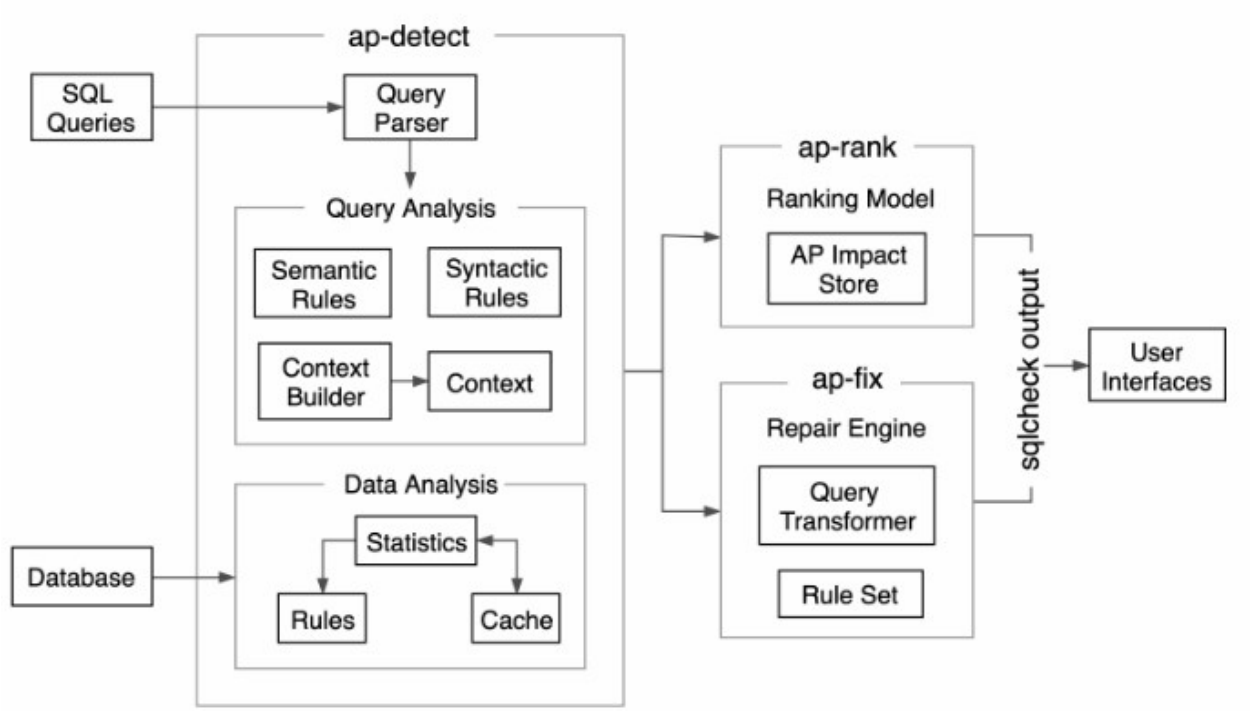


Рис. 2.3. Архітектура SQLCheck

SQLCheck приймає SQL-запит та базу даних (необов'язково) і видає впорядкований список антипатернів (АП) та пов'язані з ними виправлення. Внутрішньо sqlcheck використовує аналіз запитів та даних для виявлення АП. Потім він використовує модель ранжування та механізм виправлення запитів для створення потрібних виправлень.

Спільним у цих роботах про виявлення антипатернів є те, що всі вони є інструментами, які користувач повинен встановити, щоб ними користуватися. Дослідження показують, що розробники часто використовують веб-сайти, такі як Stack Overflow, як неформальні крауд-орієнтовані детектори шаблонів [40]. Автори припускають, що оскільки краудсорсингове виявлення враховує контекстуальні фактори, розробники йому більше довіряють, ніж автоматизованим інструментам виявлення.

Однак ми вважаємо, що це також пов’язано з тим, що більшість існуючих детекторів антипатернів є недоступними та простими у використанні. Існуючі інструменти зосереджені на виявленні антипатернів, не думаючи про досвід користувача.

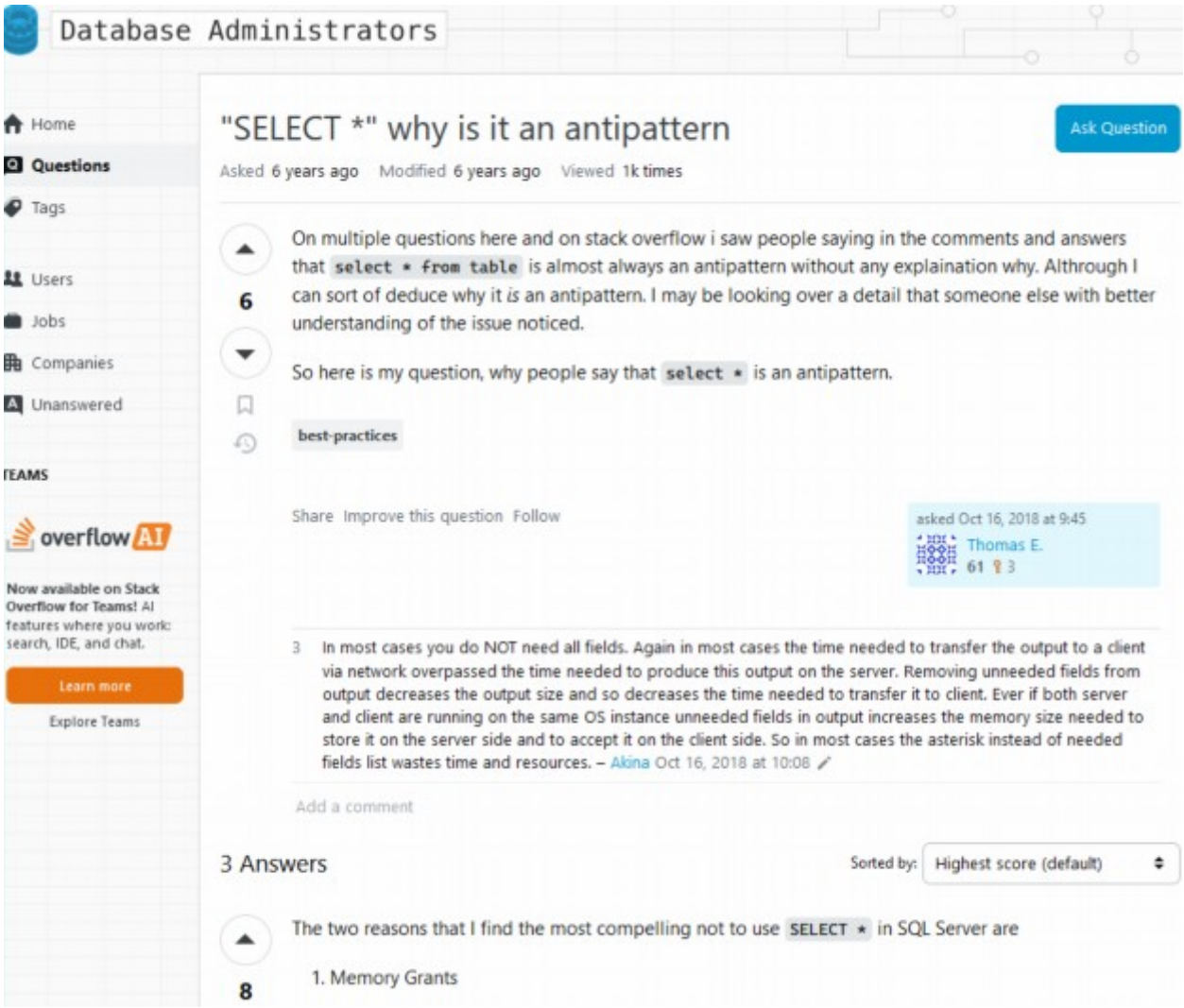


Рис. 2.4. Ресурс Stack Overflow (розділ про антипатерн SQL)

Згадані вище статті показують, що вже було проведено кілька значних досліджень на тему помилок SQL, неправильних уявлень і антипатернів. Дослідження показують нам, що є багато антипатернів (запахів коду), які шкодять якості програмного забезпечення [17 , 24] і навчанню SQL. Раннє виявлення антипатернів може допомогти розробникам уникнути технічної заборгованості, оскільки написання «поганого» SQL може призвести до

низької продуктивності або помилкових результатів. Ось чому велика частина досліджень антипатернів SQL зосереджена на їх виявленні.

2.3. Методи виявлення антипатернів в SQL

Виявлення антипатернів в SQL - важлива задача для оптимізації продуктивності баз даних та покращення якості коду. Існує ряд моделей та методів, які використовуються для цього:

1. Методи на основі правил.

Вручну визначені правила: Експерти визначають набір правил, які описують типові антипатерни. Ці правила потім використовуються для аналізу SQL-запитів та виявлення потенційних проблем. Наприклад, правило може визначати, що використання SELECT * є антипатерном.

Регулярні вирази: Використовуються для пошуку певних шаблонів в SQL-запитах, які можуть вказувати на наявність антипатернів.

2. Методи на основі машинного навчання.

Класифікація: SQL-запити класифікуються як такі, що містять антипатерни, або ні. Для цього використовуються різні алгоритми машинного навчання, такі як дерева рішень, опорні вектори та нейронні мережі.

Кластеризація: SQL-запити групуються в кластери на основі їх схожості. Антипатерни можуть бути виявлені шляхом аналізу кластерів, які містять запити з низькою продуктивністю.

3. Методи на основі аналізу статичного коду.

Аналіз потоку даних: Використовується для відстеження того, як дані передаються між різними частинами SQL-запиту. Це може допомогти виявити антипатерни, пов'язані з неефективним використанням даних.

Абстрактна інтерпретація: Спрощена версія SQL-запиту аналізується для виявлення потенційних антипатернів.

4. Гібридні методи.

- Поєднання різних методів: Наприклад, можна поєднати методи на основі правил з методами машинного навчання для досягнення кращої точності виявлення.

Приклади моделей та інструментів:

PMD: Інструмент статичного аналізу коду, який може виявляти деякі антипатерни в SQL-запитах.

FindBugs: Ще один інструмент статичного аналізу коду, який може виявляти потенційні проблеми в SQL-запитах.

JSA (Java String Analyzer): Інструмент для аналізу рядків в Java-коді, який може бути використаний для виявлення антипатернів в SQL-запитах, вбудованих в Java-код.

Вибір конкретної моделі або методу залежить від багатьох факторів, таких як тип бази даних, складність SQL-запитів та доступні ресурси.

Висновки до розділу

В даному розділі, аналіз літературних джерел показав, що антипатерни в SQL є поширеною проблемою, яка може негативно впливати на продуктивність, зрозумілість і підтримуваність баз даних. Дослідження літератури підкреслює важливість усвідомлення типових помилок і неефективних практик при розробці SQL-запитів і схем баз даних.

Дослідження інструментів для виявлення антипатернів демонструє, що існує широкий спектр засобів, які допомагають виявляти та аналізувати антипатерни в SQL-коді. Інструменти варіюються від вбудованих засобів у СУБД до окремих програмних рішень, які спеціалізуються на аналізі продуктивності та якості SQL-запитів. Кожен інструмент має свої переваги та обмеження, що слід враховувати при виборі відповідного рішення.

Методи виявлення антипатернів охоплюють різні підходи, зокрема статичний і динамічний аналіз SQL-коду. Статичний аналіз фокусується на перевірці коду без його виконання, що дозволяє виявляти помилки на ранніх етапах розробки. Динамічний аналіз, у свою чергу, включає виконання SQL-запитів та оцінку їх продуктивності в реальному середовищі, що дає змогу виявляти антипатерни, пов’язані з продуктивністю.

Загалом, даний розділ висвітлює ключові підходи до виявлення антипатернів в SQL та визначає важливі інструменти й методи, які сприяють підвищенню ефективності та якості SQL-коду.

РОЗДІЛ 3. ІМПЛЕМЕНТАЦІЯ МОДЕЛЕЙ ТА МЕТОДІВ ДЛЯ ВИЯВЛЕННЯ АНТИПАТЕРНІВ SQL

3.1. Визначення класів та особливостей антипатернів

Якщо переглянути список антипатернів, то можна помітити, що деякі антипатерни виникають, коли хтось змішує SQL-запити з логікою програми, тобто ми не зможемо виявити ці антипатерни, коли нам надається лише необроблений запит. У тому ж порядку, що й у списку антипатернів, першим антипатерном, який використовує іншу мову поряд із SQL, є антипатерн Implicit Rows, який використовує логіку програми для додаткового фільтрування запитуваних даних. Далі йде антипатерн Loop to Join, де ми використовуємо логіку програми для запиту даних з однієї таблиці, а потім використовуємо структуру циклу для запиту іншої таблиці, використовуючи значення першого запиту. Цей антипатерн супроводжується антипатерном Readable Password і Vulnerable Query, обидва використовують логіку програми для вставки даних програми в запит. Останнім антипатерном, який використовує логіку програми, є Unbounded Query. Цей антипатерн виникає, коли набір результатів запиту може повертати необмежену кількість записів, а програма, яка використовує запит, виконує наступне обчислення над цим набором результатів. Оскільки наш інструмент виявлення повинен мати можливість аналізувати необроблені запити SQL, і ми не можемо виявити ці антипатерни в необробленому запиті без надання логіки програми, ми вважаємо ці антипатерни поза нашою сферою.

Інша група антипатернів, які не можна виявити з одного необробленого SQL-запиту, це група, яку можна виявити лише після отримання кількох запитів. Це стосується предикатів не об’єднання проекцій, предикатів вибору необ’єднання та антипатернів Unbatched Rewrites . Перші два антипатерни можна виявити, лише якщо ми побачили принаймні два запити, які або читають підмножину стовпців, підмножину рядків або обидва. Останній

анти-шаблон виникає, коли у нас є кілька запитів INSERT, які записуються окремо в базу даних, замість того, щоб об’єднуватися разом.

Останньою групою антипатернів, які не можуть бути виявлені одним необробленим SQL-запитом, є група, що містить антипатерни Not Using Parameterized Query і Not Caching. Ці два антипаттерни виникають, коли не застосовуються певні оптимізації (параметризація та кешування запитів). З необроблених запитів SQL ми не можемо спостерігати ці оптимізації, тому їх неможливо виявити.

Тепер, коли ми визначили, які антипатерни не можна виявити в необроблених запитах SQL, ми можемо створити список антипатернів, які можна виявити. Наступні антипатерни залишилися та можуть бути реалізовані в нашому інструменті виявлення статичних необроблених запитів:

- Ambiguous Groups
- Fear of the Unknown
- Implicit Columns
- Poor Man’s Search Engine
- Random Selection
- Spaghetti Query

Деякі з цих антипатернів, наприклад антипатерни Implicit Columns, можуть зустрічатися в операторах, які не починаються з SELECT, наприклад, запити INSERT. Оскільки наш інструмент буде націлений на користувачів-початківців, які здебільшого писатимуть запити SELECT, ми вирішили зосередитися на виявленні цих антипатернів лише в запитах SELECT.

3.2. Реалізація інструменту виявлення антипатернів SQL

В даній роботі, ми хочемо зробити антипаттерн-детектор якомога доступнішим. Тому ми розробимо його різними способами. Перший спосіб

— через пакет Python, який містить інтерфейс командного рядка (CLI). Другий спосіб — через RESTful API. Останній спосіб – через веб-додаток.

3.2.1. Використання пакету Python із CLI

Як згадувалося в попередньому розділі, детектор написаний мовою Python. Python — це дуже доступна мова, яку більшість користувачів-початківців встановить на своїй локальній машині. Окрім створення доступного інструменту для новачків, ще однією метою було зробити детектор максимально розширюваним. У результаті, коли виявляються нові антипатерни, до інструменту легко додати нові правила, що робить його більш універсальним інструментом. Ще одна перевага полягає в тому, що в системі також можна використовувати для виявлення більш загальних помилок і неправильних уявлень, таким чином забезпечуючи більш повну систему навчання.

Використовуючи модуль argparse, ми створили простий у використанні інтерфейс командного рядка (CLI). Ця програма CLI, а також основна програма детектора експортуються як єдиний пакет і завантажуються в індекс пакетів Python. Це означає, що користувачі-початківці можуть просто запустити pip install sqle, щоб завантажити та встановити детектор.

Після інсталяції користувач може відкрити термінал і відкрити довідковий посібник із усіма можливими параметрами та аргументами.

Користувач може вставити запит за допомогою аргументу запиту. Це робиться за допомогою прапорця -q або --query, після якого вказується запит, який потрібно проаналізувати. Детектор запускається, коли користувач натискає клавішу enter. Тепер він перегляне всі раніше розкриті антипатерни та методи виявлення. Нарешті, він покаже короткий опис антипатернів, знайдених детектором. Цей підсумок дає короткий заголовок, який повідомляє користувачам, які не знають антипатерни, що не так із запитом, тип антипатерну, достовірність появи антипатернів і місце, де виявлено антипатерни.

Припустимо, ми передаємо запит, показаний на рисунку 3.1, як вхідні дані до CLI.

```
SELECT *
FROM product
WHERE pCat <> NULL
  AND pName LIKE "%ice%"
  AND price <> NULL
  AND pSection LIKE "self%"
GROUP BY name
ORDER BY random()
LIMIT 1
```

Рис. 3.1. Приклад запиту, що містить різні антипатерн

У цьому запиті використовується селектор підстановок, що означає, що виникає антипатерн неявних стовпців. По-друге, він неправильно обробляє значення NULL, а це означає, що також присутній антипатерн Fear of the Unknown . Крім того, він використовує зіставлення шаблонів , що саме по собі є антипатерном. Нарешті, він використовує ORDER BY random() для випадкового сортування результату, це відомо як антипатерн випадкового вибору. Результат показано на рисунку 3.2.

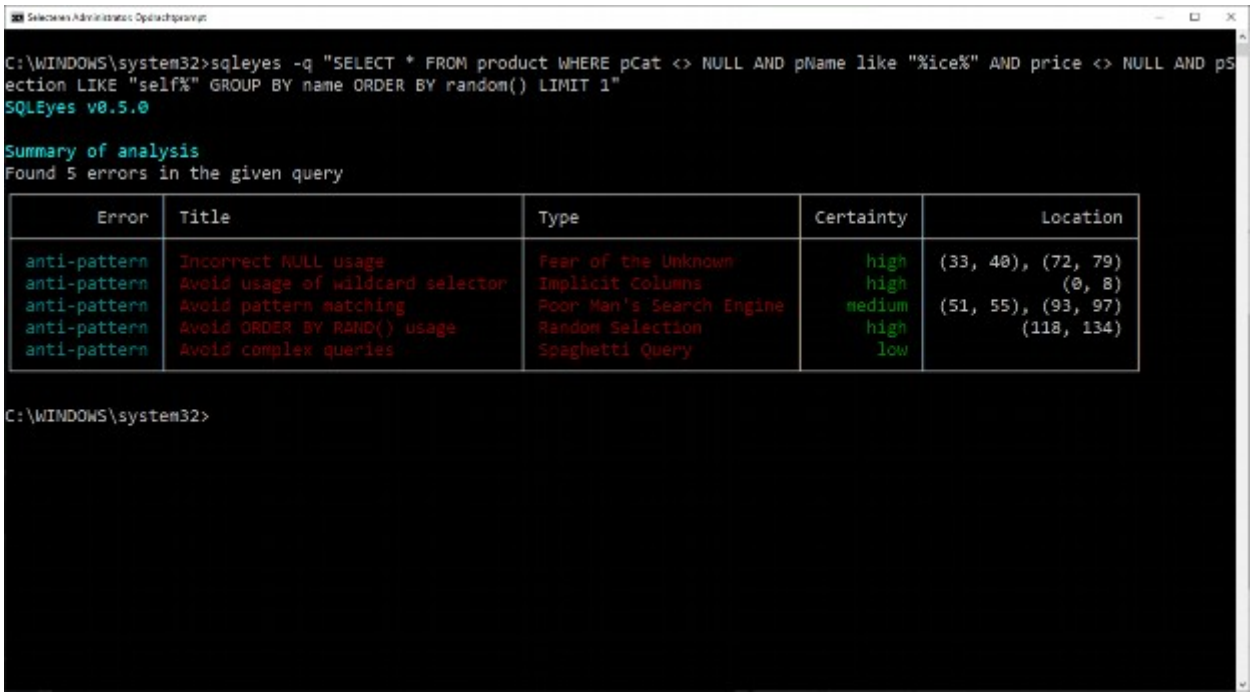


Рис. 3.2. Мінімальний результат без пояснення

Ми також можемо запустити той самий вхід із увімкненим прапорцем –d (description). Це забезпечить підсвічування помилок, а також більш детальний опис кожного антипатерну, який було виявлено, і можливе рішення для виправлення антипатерну на додаток до підсумкового виводу. Цей результат показано на рисунку 3.3.

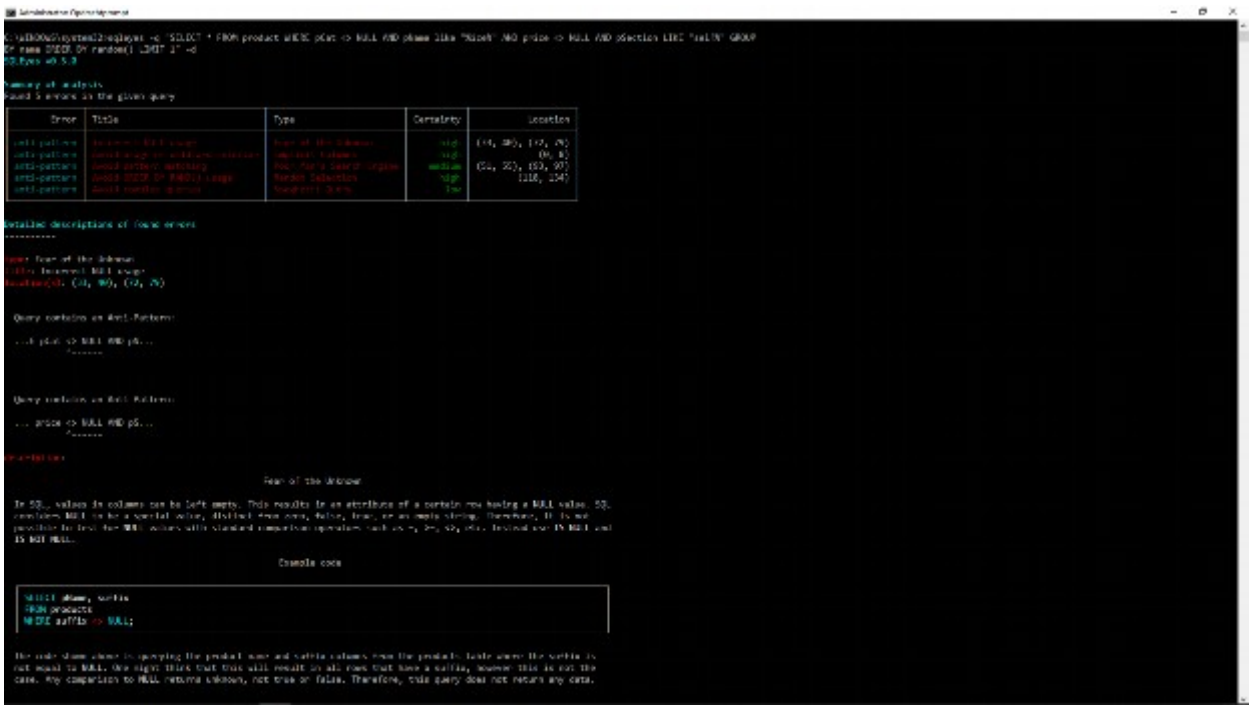


Рис. 3.3. Деталізований результат з поясненням

3.2.2. Метод з використанням RESTful API

Наступний спосіб реалізації антипаттерн-детектора — через RESTful API, який можна розміщувати самостійно. Основна перевага полягає в тому, що новачкам не потрібно встановлювати програму, їм просто потрібно виконати правильний запит до кінцевої точки API. Окрім легшого доступу, ніж до CLI, API також можна легко інтегрувати в існуючі програми, де використовуються необроблені запити SQL.

Наш API створено на основі Flask. Flask — це веб-фреймворк Python, який використовує корисні функції та пакети, які дозволяють швидко налаштувати REST API. Таким чином, це добре налагоджена і популярна структура для створення RESTful API. Користувач може зробити POST-запит

до кінцевої точки API із запитом для аналізу як корисним навантаженням. Це поверне об’єкт JSON зі списком достовірностей, описів, розташування, заголовків і типів ідентифікованих антипатернів. Лістинг коду на рисунку 3.4 показує приклад запиту HTTP POST до API, розміщеного на сервері.

```
POST /analyze HTTP/1.1
Host: api.query.com
Content-Type: application/json
Content-Length: 88

{
    "query": "SELECT pId, supplierId, AVG(price) FROM product WHERE
       price <> NULL"
}
```

Рис. 3.4. Приклад запиту з використанням API

Детектор антипатернів також можна поширювати через веб-додаток, який може виконувати та аналізувати SQL-запит. Користувачі просто запускають запит у веб-браузері, а програма аналізує його. Це, мабуть, найдоступніший спосіб використання, оскільки він надає користувачеві можливість виконувати запит без написання жодного фрагмента коду, окрім свого запиту, або завантаження будь-якого програмного забезпечення.

3.3. Особливості побудови інтерфейсу взаємодії з користувачем

Спочатку ми розглянемо кілька важливих знахідок у сфері інтерфейсу користувача/взаємодії користувача (UI/UX), які ми включили в наш дизайн. Для нашої програми ми не хотіли розробляти складний користувальницький інтерфейс із занадто великою кількістю функцій, тому що ми не хотіли перевантажувати користувача занадто великою кількістю інформації. Ми хотіли створити програму, яка проста у використанні та зрозуміла сама за себе. Користувачі повинні мати можливість використовувати систему без необхідності читати документацію. Але як створити таку програму? Розробка програми виглядає як тривіальне завдання. З іншого боку, розробити

програму з хорошим користувальницьким інтерфейсом і користувальницьким досвідом не завжди є легким завданням.

Інтерфейс користувача є першою точкою взаємодії між користувачем і програмним додатком. Взаємодія з користувачем виходить за межі інтерфейсу користувача, включаючи сприйняття та емоції користувача під час взаємодії з програмним забезпеченням. Перш ніж сформувати основні вимоги до нашої програми, ми розглянули деякі важливі роботи з дизайну інтерфейсу користувача та взаємодії з користувачем. Ми також розглянули деякі принципи дизайну, які можуть допомогти нам створити кращий інтерфейс користувача для нашої програми. Існують різні принципи проектування, спрямовані на створення ефективного інтерфейсу користувача та досвіду. При формуванні вимог до заявки ми будемо враховувати ці принципи. Оскільки не всі ці принципи дизайну застосовні до нашої програми, ми обговоримо лише ті, які ми розглядали під час розробки.

Однією з найважливіших робіт щодо проектування не лише програмних додатків, а й речей загалом є «Дизайн повсякденних речей» [27]. Автор обґрунтовує людиноорієнтований дизайн у «Дизайні повсякденних речей», описуючи, як дизайнери мають думати про користувачів та їхні проблеми. Він деконструює загальні недоліки дизайну та надає читачам основу для дизайну, орієнтованого на користувача, що означає, що дизайнери повинні проектувати в першу чергу для людей та їхніх потреб. Автор стверджує, що корисний дизайн починається з ретельних спостережень за тим, як завдання, що підтримуються, фактично виконуються, після чого йде процес проектування, який призводить до гарної відповідності фактичним способам виконання завдань. Він також вважає, що важливо полегшити людям пошук і виправлення помилок, оскільки ми ніколи не повинні звинувачувати користувача в помилці. У нашому випадку нам потрібно подивитися, як наші користувачі використовуватимуть нашу програму для аналізу необроблених запитів SQL. Ми також повинні переконатися, що коли виникає помилка, наприклад, якщо синтаксис SQL неправильний,

користувачеві буде запропоновано виправити цю помилку. Тому ми повинні гарантувати, що користувачі будуть сповіщені, коли виникають такі помилки.

В дослідженні [26] щодо інтерфейсів користувача комп'ютера та взаємодії з користувачем, вони стверджують десять важливих евристик зручності використання. Особливо важливі три евристики, а саме контроль і свобода користувача, розпізнавання, а не пригадування, а також гнучкість і ефективність використання.

Перша евристика говорить про те, що користувачі часто роблять помилки під час виконання дій. Їм потрібен чітко позначений «аварійний вихід», щоб залишити небажану дію без необхідності проходити тривалу процедуру.

Друга важлива евристика говорить нам зробити елементи, дії та опції видимими, щоб зменшити навантаження на пам'ять користувача.

Остання евристика говорить нам, що ярлики, які слід приховати від користувачів-початківців, можуть прискорити взаємодію із системою для досвідчених користувачів, зробивши для них зручнішим користування. Ми врахували ці евристики при розробці інтерфейсу користувача нашої програми: ми допомагаємо нашим користувачам позбутися помилок, натиснувши одну кнопку, ми зменшуємо навантаження на пам'ять користувача завдяки чистому та мінімалістичному інтерфейсу, де помітно видно лише основні функції та ми реалізували ярлик для кожної команди та дії, яку може виконати користувач.

Крім того, у блозі є багато публікацій і статей, у яких обговорюються різні поради та підказки щодо створення гарного дизайну інтерфейсу користувача, який приносить користь UX. Ці дописи в блозі та статті обговорюють різні питання, такі як типографіка, натхнення для дизайну, макет, теорія кольорів, темний режим тощо. Ці поради та підказки також використовуються в дизайні інтерфейсу користувача нашої програми.

3.4. Реалізація основних вимог до інструменту виявлення антипатернів

Використовуючи висновки, отримані з вищезгаданих робіт, ми тепер встановлюємо набір вимог до проектування, пріоритетність яких здійснюється за допомогою методу Must, Should, Could і Would [9]. Даний метод пріоритезації – це інструмент, який використовується для визначення важливості результатів проекту. Ось короткий огляд кожної категорії:

- Необхідно: це вимоги до проекту, які є абсолютно необхідними для успішного виконання проекту. Без цих вимог проект вважався б невдалим.

- Повинен: це важливі вимоги проекту, але вони не важливі для успіху проекту. Проект можна було б завершити без цих вимог, але він не був би таким успішним.

- Міг би: це необов'язкові вимоги до проекту, які було б добре мати, але не обов'язково.

- Буде: Це вимоги до проекту, які було б добре мати, але не є основними та навряд чи будуть виконані.

Перш ніж створити вимоги, ми повинні були встановити основний варіант використання нашої програми. Нашою метою було підвищити обізнаність новачків про антипатерни SQL. Цю мету можна досягти, створивши програму, де новачки зможуть вводити та аналізувати SQL-запит. Завдяки цьому підсвічуються потенційні антипаттерни SQL, що дозволяє новачкам вчитися на них. Потім подумав про те, як кінцевий користувач виконує завдання, а саме аналіз запитів SQL. У нас уже є два способи аналізу запитів, а саме через CLI або API. Програма – це новий підхід для аналізу SQL-запитів, оскільки користувачам не потрібно нічого встановлювати. Вони можуть аналізувати запити, просто перейшовши на веб-сайт, ввівши запит у текстове поле та натиснувши кнопку аналізу. Потім аналіз буде показано користувачеві. Виходячи з нашої мети та сценарію використання, ми створили вимоги. Виставляються такі вимоги:

Must have

M1. Система повинна містити розділ для введення запиту та розділ для відображення результатів аналізу запиту.

M2. Після введення запиту та натискання кнопки «запустити» система аналізує запит на наявність антипатернів і відображає результат.

M3. Якщо запит містить антипатерн і запит аналізується системою, результат аналізу запиту має відображати опис проблеми, антипатерн, приклад, потенційне виправлення та показати, де антипатерн знайдено в запиті.

Should have

S1. Система повинна відображати запити підсвічуванням синтаксису.

S2. Після введення запиту та натискання кнопки «форматувати» система відформатує вставлений запит.

S3. Після введення запиту та натискання кнопки «очистити» система видаляє запит, вміст розділу вихідних даних запиту та вміст розділу вихідних даних аналізу запиту.

S4. Коли завантажено дійсну базу даних SQLite, введено запит і натиснуто кнопку запуску, система повинна містити розділ для відображення результатів запиту, виконання запиту та відображення набору результатів.

S5. Коли API недоступний або не може обробити запит і натиснуто кнопку «запустити», система має відобразити повідомлення про помилку з відповідним описом помилки в розділі результатів аналізу запиту.

S6. Коли завантажено дійсну базу даних SQLite, введено запит із принаймні однією синтаксичною помилкою та натиснуто кнопку «запустити», система має відобразити повідомлення про помилку з відповідним описом помилки в розділі виводу запиту.

Could have

C1. Після натискання кнопки «скасувати/повторити» система скасує/повторить останню дію, виконану користувачем.

C2. Коли натиснуто «перемкнути темний режим», система скасує/повторить останню дію, виконану користувачем.

C3. Система повинна містити комбінації клавіш (гарячі клавіші) для виклику певних дій і надання користувачеві негайного доступу до них.

C4. Систему можна використовувати як окрему програму, без використання веб-браузера.

Would have

W1. Коли завантажено дійсну базу даних SQLite і натиснуто кнопку «перегляд схеми», система відобразить схему бази даних.

W2. Коли в запиті знайдено антипатерн, система виділить його в режимі реального часу у полі введення запиту.

Як було сказано раніше, категорія «Must have» — це основні функції, які необхідно реалізувати. Якщо ці функції недоступні, ми вважаємо цей проект невдалим. Вимоги, які ми перерахували в категорії «Must have», — це найнеобхідніше, яке повинно мати додаток для аналізу шаблонів SQL. Для нашого сценарію використання необхідно виконати три вимоги: користувач може ввести текст (M1), користувач може проаналізувати запит, натиснувши кнопку (M2), і аналіз буде показано користувачеві (M3).

Вимоги, перелічені в категорії «Should have», є функціями, які роблять наш додаток цінним для інших користувачів. Із запровадженням цих вимог програму можна використовувати більш загально, оскільки тепер вона не лише є програмою для аналізу шаблонів SQL, але й програмою, яка може виконувати фактичні запити.

Вимоги, перелічені в розділах «Could have» та «Would have», роблять додаток більш зручним для користувача. Усі ці вимоги є невеликими вимогами, які можна побачити в порівнянних програмах, оскільки вони роблять ці програми більш привабливими для кінцевого користувача та підвищують продуктивність.

На рисунках, що подано нижче показано варіанти реалізації поданих вимог.

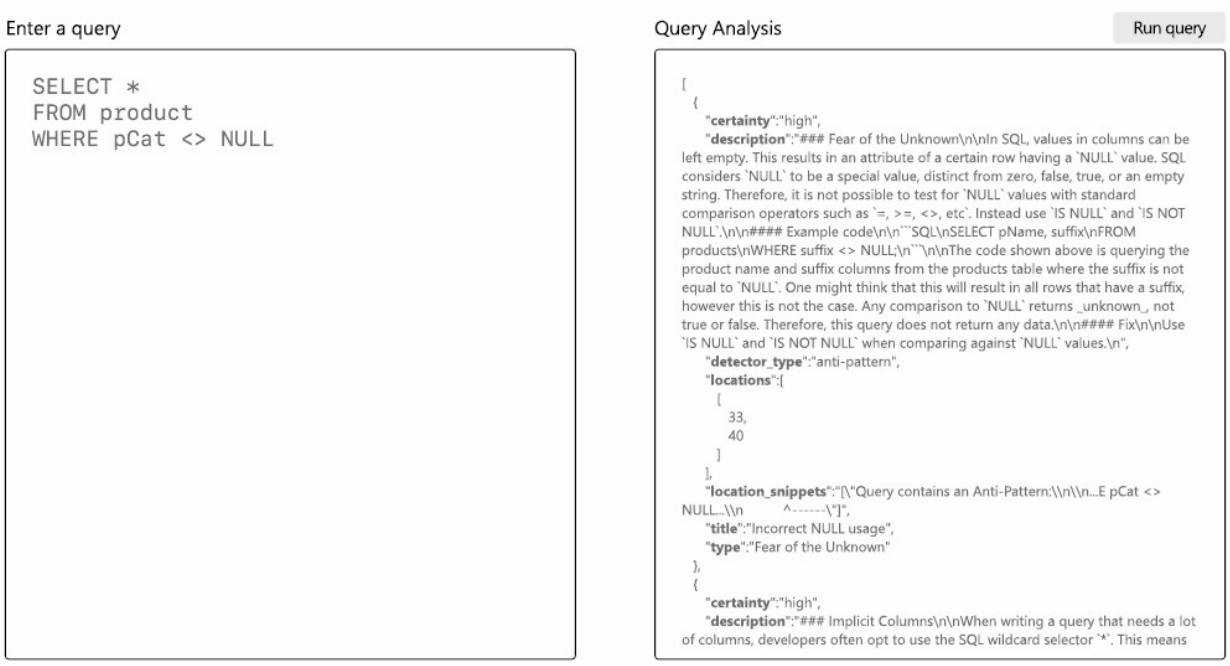


Рис. 3.5. Реалізація вимог M1 – M3

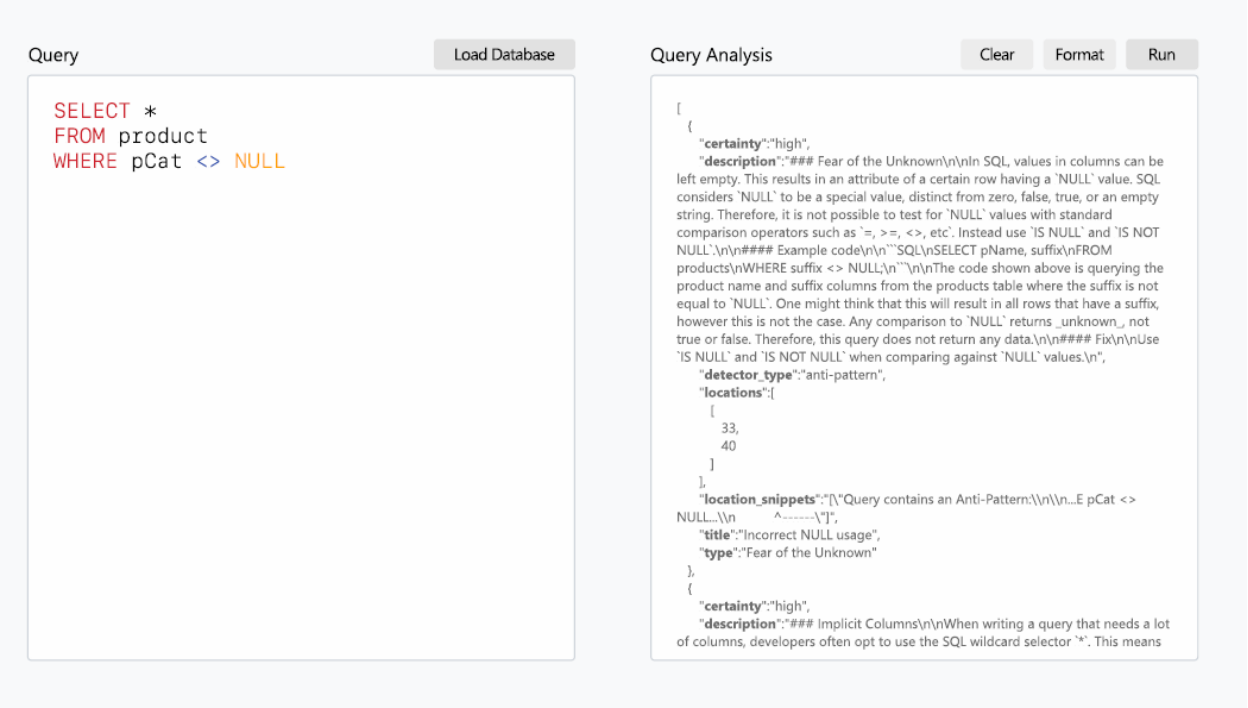


Рис. 3.6. Реалізація вимог M1 – S6

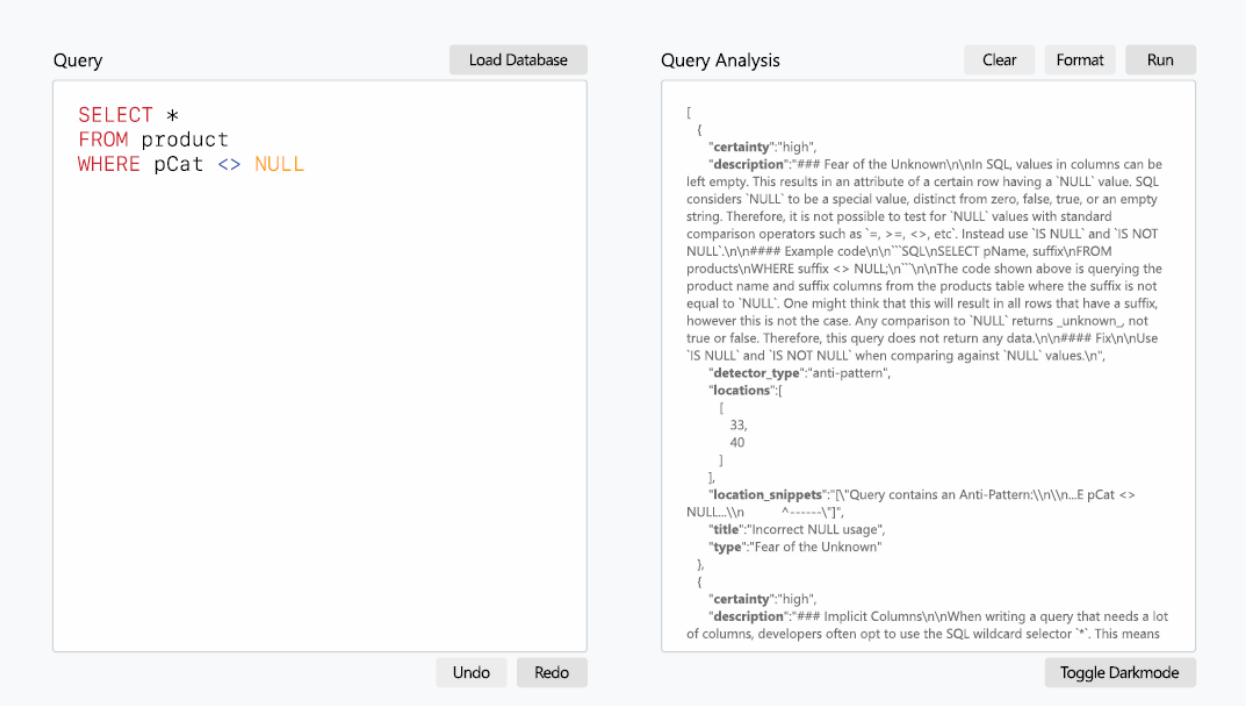


Рис. 3.7. Реалізація вимог M1 – C4

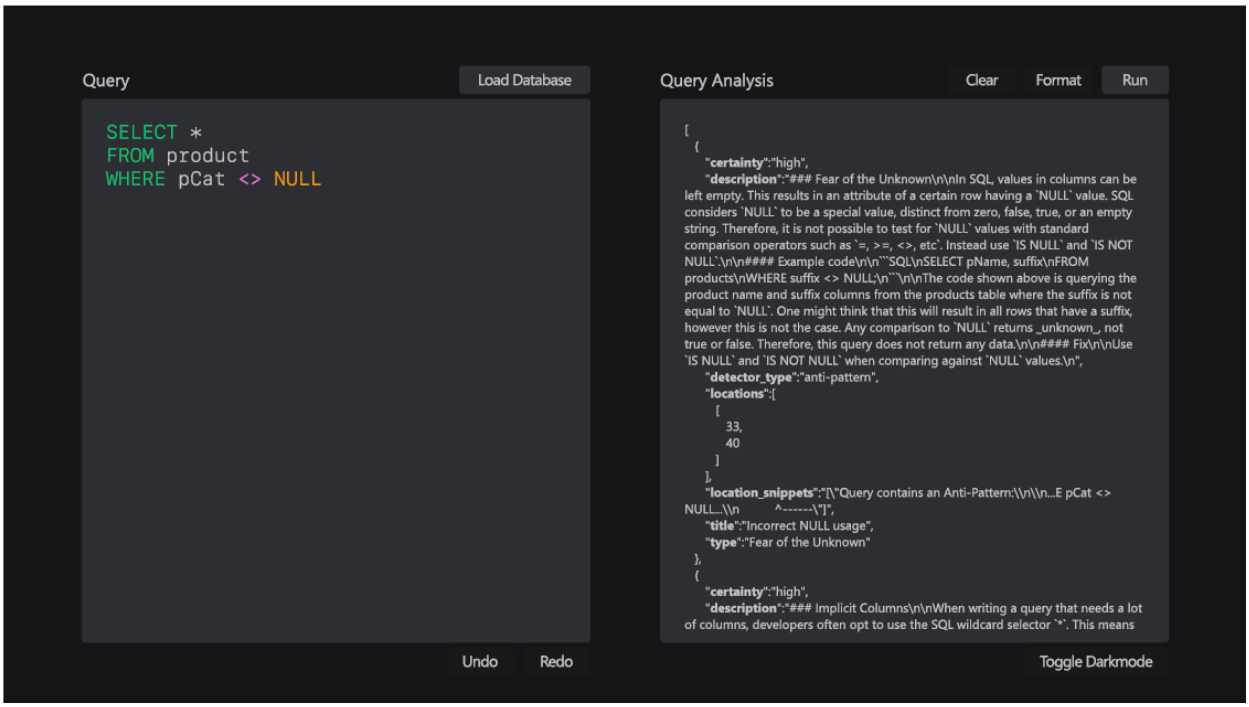


Рис. 3.8. Темна версія

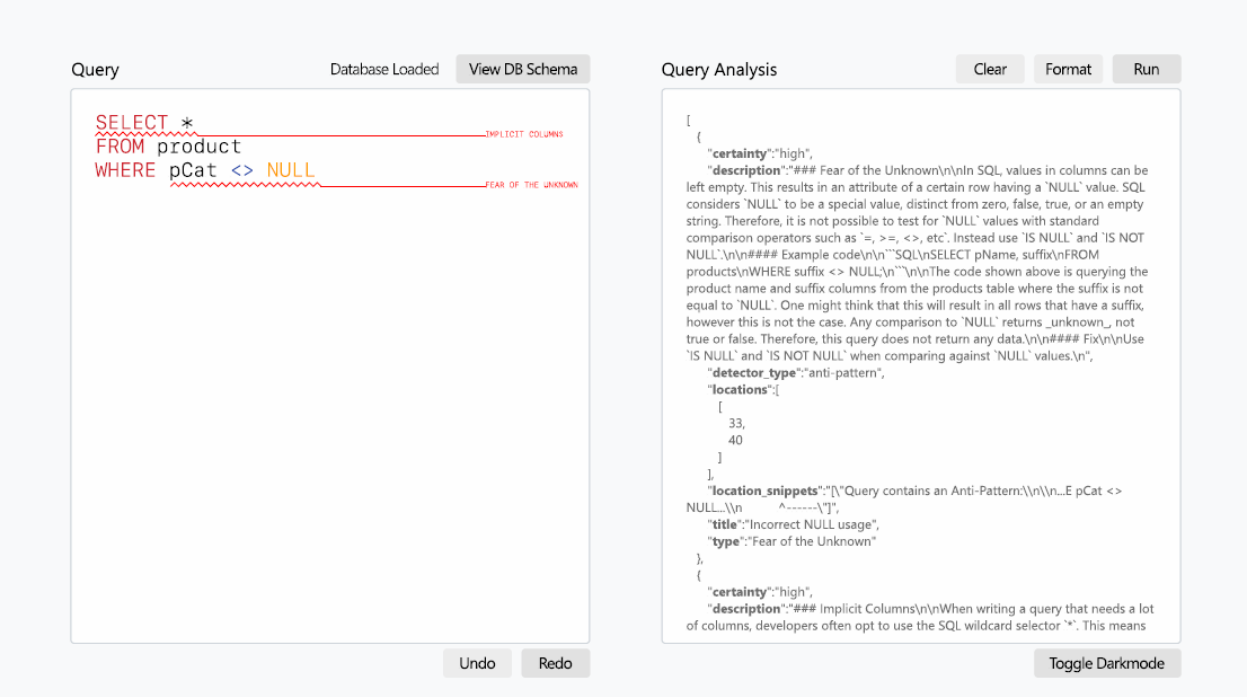


Рис. 3.9. Реалізація вимог M1 – W2

3.5. Архітектура та дизайн пропонованого інструменту

Ми почали з розробки інтерфейсу користувача після ретельного вивчення вимог до проекту. Ми створили кілька (кольорових) макетів інтерфейсу користувача, щоб допомогти нам визначити структуру системи. Після того, як ми затвердили макети, ми почали працювати над фактичним дизайном інтерфейсу користувача високої точності.

Ми почали з макета, який реалізує лише вимоги "Must have" (обов'язкові). Після цього ми використали цей макет як основу і додали елементи інтерфейсу користувача на основі вимог "Should have" (бажані). Третій макет базується на другому макеті та включає додаткові функції на основі вимог "Could have" (можливі). Остаточний макет включає всі вимоги, включаючи вимоги "Would have" (додаткові). Макети створені у Figma. Всі макети показані на рисунку 3.9.

На рисунку 3.10 представлено фінальний дизайн високої точності інструменту. Всі вимоги (як "Must have", так і "Should have") були включені в цей дизайн високої точності. Через обмеження в часі цього проекту інші

вимоги були опущені. Для загального вигляду та стилю нашої програми ми використали бібліотеку компонентів під назвою Mantine. Основною перевагою використання бібліотеки компонентів є те, що ви можете більше зосередитися на логіці системи, а не на естетиці програми. Крім того, це допомагає нам підтримувати узгодженість у дизайні нашої програми.

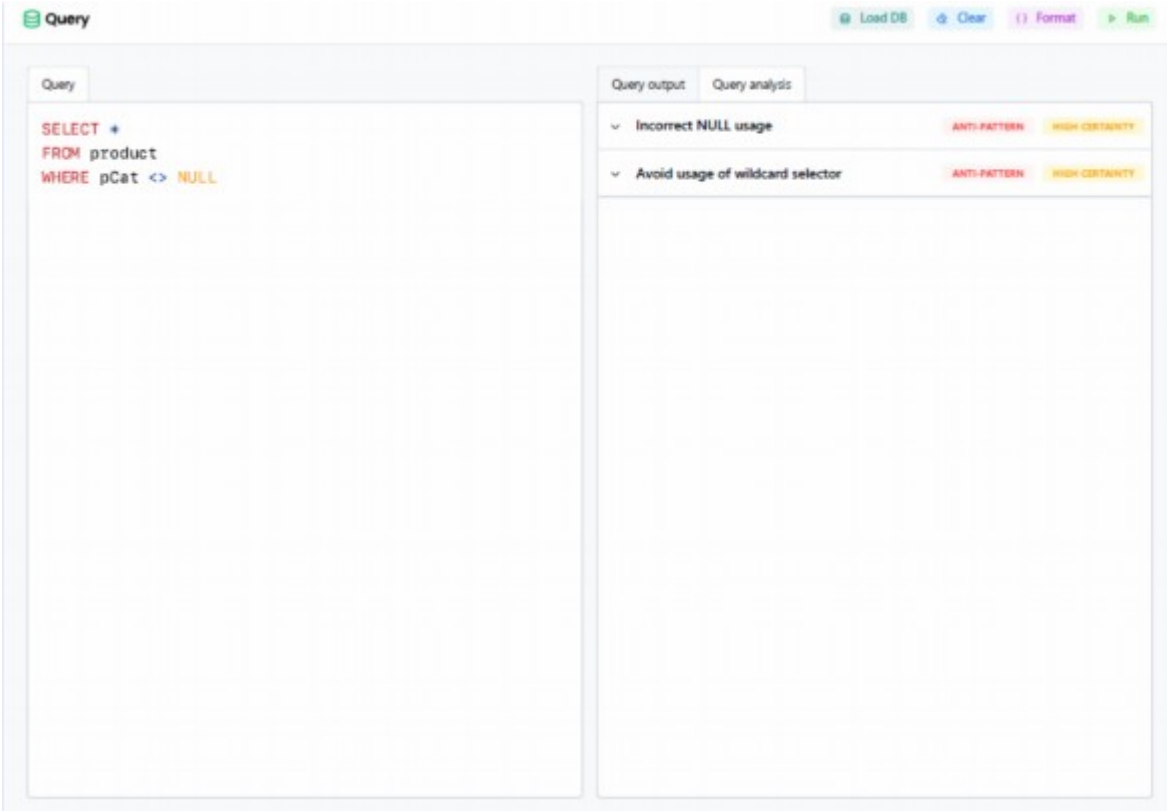


Рис. 3.10. Макет пропонованого інструменту

Бібліотека Mantine надає повний набір компонентів, таких як кнопки, текстові поля та вкладки. Mantine також має доступний файл дизайну Figma. Це дозволило нам використовувати компоненти Mantine як у нашому макеті високої точності, створеному за допомогою Figma, так і в нашому коді React, зберігаючи при цьому єдиний стиль. Ми також використали бібліотеку CSS in JS під назвою styled-components. На відміну від традиційного CSS, styled-components дозволяє використовувати JavaScript для стилізації HTML.

Фінальний дизайн було створено за допомогою TypeScript & React 4 і він значною мірою на основі макету високої точності, показаним на рисунку

3.10. Ми обрали React, оскільки це найпопулярніша бібліотека JavaScript для створення інтерфейсів користувача, вона дозволяє створювати компоненти багаторазового використання та легко інтегрується зі сторонніми бібліотеками Javascript. Фінальний дизайн реалізує вимоги: M1, M2, M3, S1, S2, S3, S4, S5, S6, C2, C3 та C4.

TypeScript - це мова на основі JavaScript, яка додає статичні анотації типів для підвищення продуктивності розробників та якості коду. Синтаксис дуже схожий на JavaScript, фактично, це надмножина JavaScript, що робить його простим у вивченні для існуючих розробників JavaScript. TypeScript дозволяє використовувати специфічні для TypeScript функції IDE, такі як автодоповнення коду та перевірка типів, що зменшує кількість помилок часу виконання. Він також забезпечує покращену підтримку рефакторингу коду, що допомагає в обслуговуванні великих проектів.

Ми використовували Vite 5 для розробки інтерфейсу. Vite - це інструмент збірки веб-розробки, який використовує власні ES-модулі та TypeScript для забезпечення швидкого налаштування розробки. Він служить основним конкурентом Create React App. Основною перевагою Vite є низький час збірки, що пов'язано з його здатністю безпосередньо обслуговувати вихідні файли. Інші переваги Vite включають його просту конфігурацію та підтримку гарячої заміни модулів (HMR).

І інтерфейс, і серверна частина традиційної веб-програми реалізовані на сервері. Однак для нашого випадку використання це не був найефективніший метод, оскільки рендеринг на стороні сервера означає, що всю програму потрібно буде перезавантажувати щоразу, коли користувач аналізує запит. Замість цього ми використовували рендеринг на стороні клієнта, який відображає більшу частину програми на стороні клієнта (браузері користувача). Це означає, що коли користувач аналізує запит, серверу потрібно лише відповісти даними аналізу запиту. Рендеринг на стороні клієнта є більш ефективним, оскільки він зменшує навантаження на сервер, одночасно покращуючи взаємодію з користувачем, швидше

завантажуючи дані. Це призводить до швидкого завантаження односторінкової програми (SPA), яка асинхронно отримує дані.

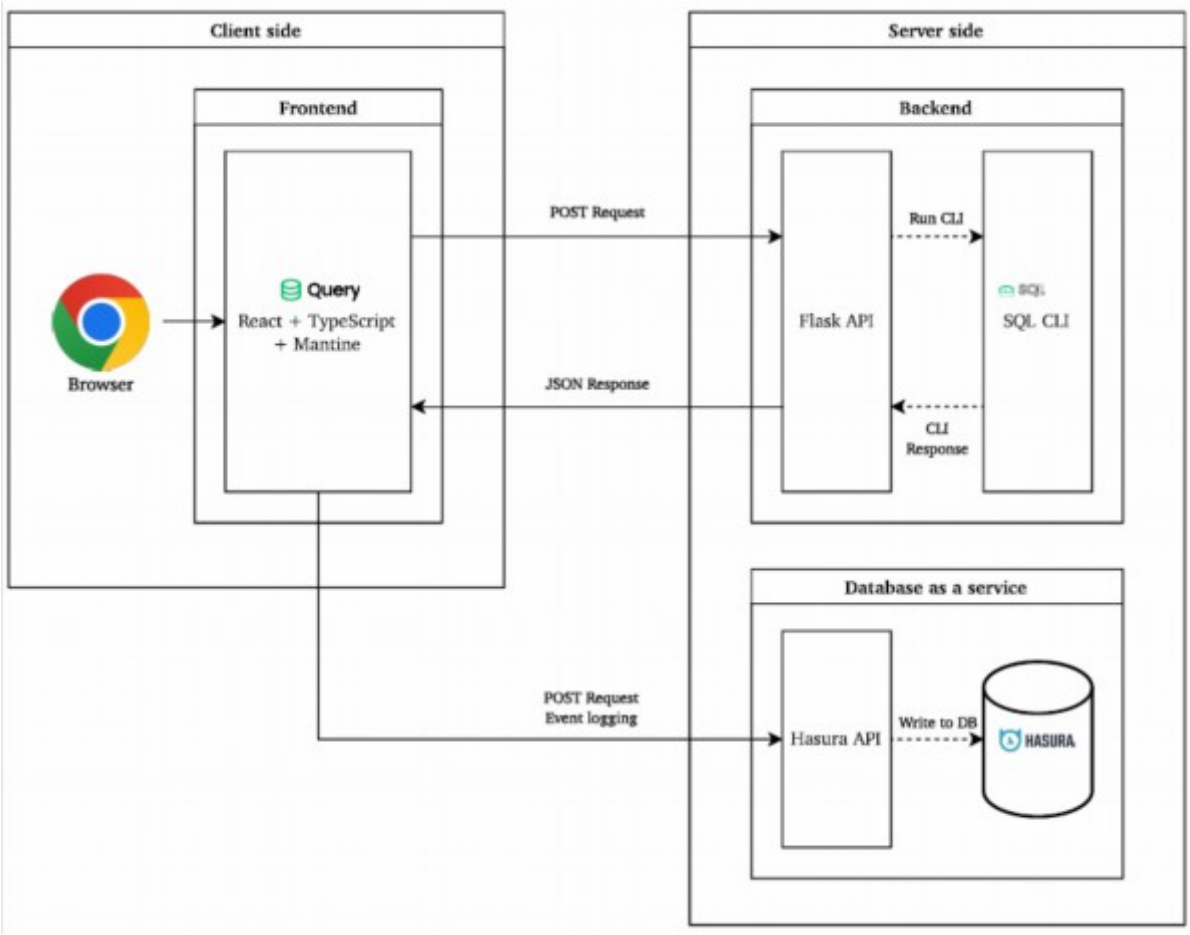


Рис. 3.11. Архітектура інструменту виявлення антипатернів

На рисунку 3.11 показано архітектуру нашої системи. З цією архітектурою весь інтерфейс завантажується одночасно на стороні клієнта. Щоразу, коли виконується дія, яка потребує сервера, у нашому випадку, коли користувач запитує аналіз запиту, POST-запит, що містить введений запит, надсилається до API SQL, що працює на сервері. Це, у свою чергу, використає SQL CLI для аналізу запиту. Потім API SQL розбирає відповідь від SQL CLI та повертає результати на сторону клієнта. Потім клієнт відображає результати та показує їх користувачеві.

Для подальшого аналізу ми також реєструємо всі дії користувача в базі даних. Ми використовуємо базу даних як послугу під назвою Hasura.

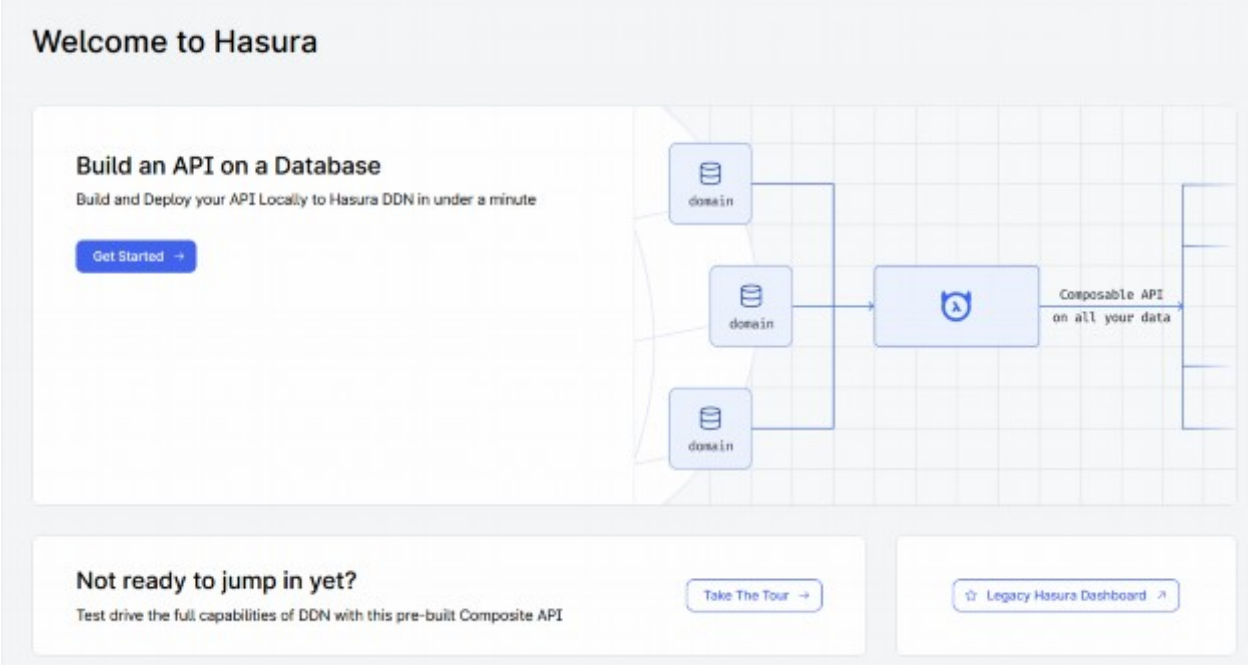


Рис. 3.12. Сервіс БД Hasura

Hasura включає базу даних PostgreSQL, яку можна використовувати для створення таблиць у середовищі Hasura. Потім Hasura надає нам кінцеву точку API GraphQL, до якої ми можемо надсилати запити, включаючи POST-запити для реєстрації дій користувача. Запити реєструються асинхронно, щоб уникнути перешкод роботі основної програми. Коли API Hasura отримує POSTзапит, він обробляє його та автоматично створює новий запис у базі даних. Ми використовуємо цю службу для реєстрації всіх дій користувача та відстеження того, що користувачі роблять у програмі. Ми не реєструємо жодних даних користувача, лише взаємодію користувача з програмою. Це дозволяє нам визначити, які функції використовуються, як часто, і відповідно покращити програму.

Було зроблено Query доступним як окрему програму, згідно з вимогою C4. Для цього ми використали JavaScript-фреймворк з відкритим кодом під назвою Electron. Electron дозволяє нам створювати кросплатформні настільні програми за допомогою TypeScript, HTML та CSS. Electron працює з механізмом Chromium для створення вікна, подібного до веб-браузера, яке називається основним процесом. Потім ми можемо відображати локальні

HTML-файли в цьому вікні. Оскільки наш інтерфейс - це односторінкова програма, ми також можемо експортувати цю програму в статичну односторінкову програму. Це призведе до набору статичних файлів HTML, CSS та JavaScript. Потім ми передаємо ці файли фреймворку Electron, що призводить до створення окремої кросплатформної настільної програми. Нарешті, ми збираємо проект, який об'єднує код і файли та створює інсталятор.

Тепер, коли ми розглянули базову архітектуру QuerySandbox, ми розглянемо реалізацію вимог. Ми розглянемо різні важливі елементи фінального дизайну, якщо можливо, пов'яжемо їх з певними вимогами та пояснимо їх призначення.

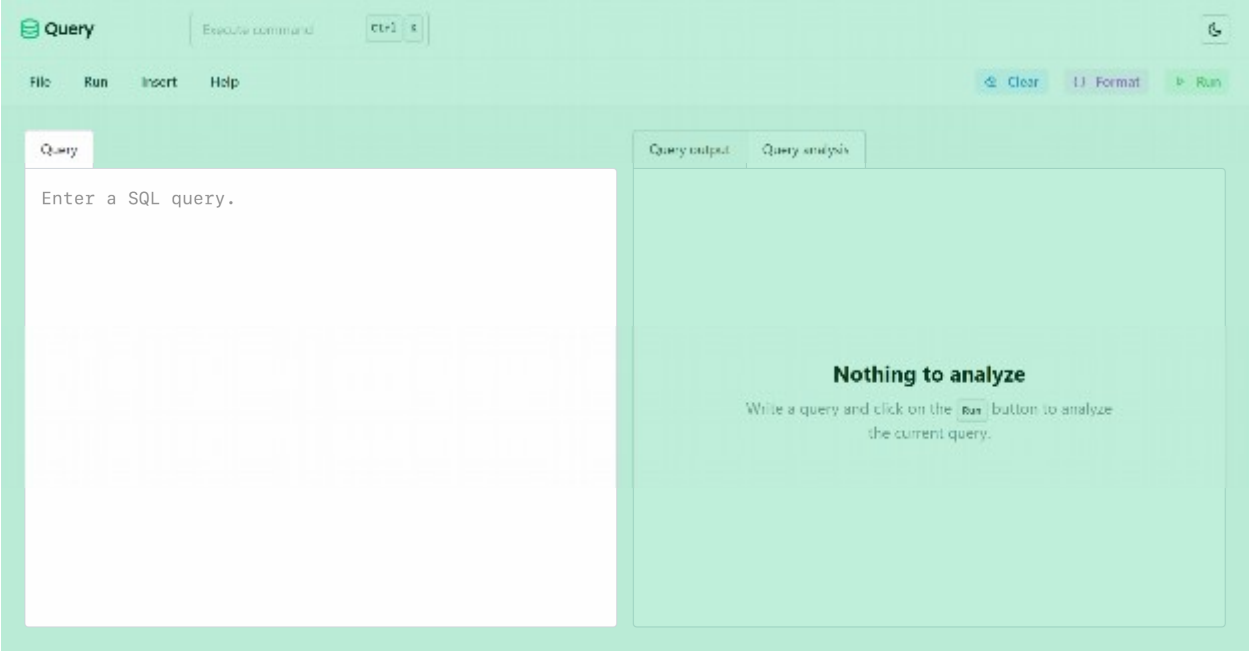
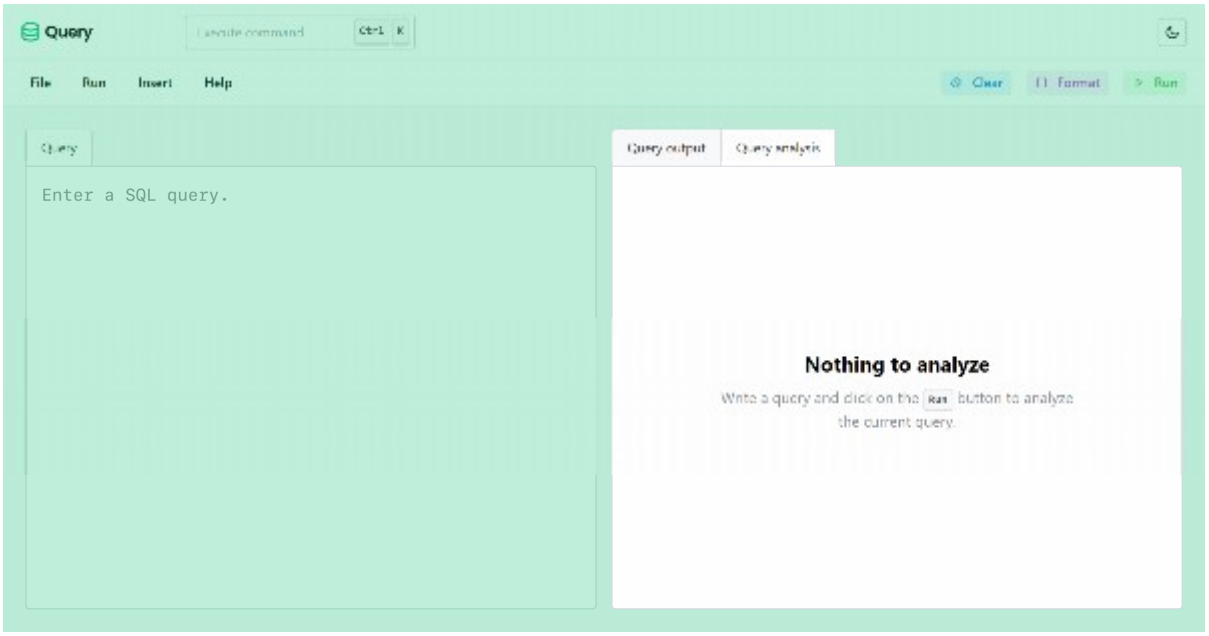


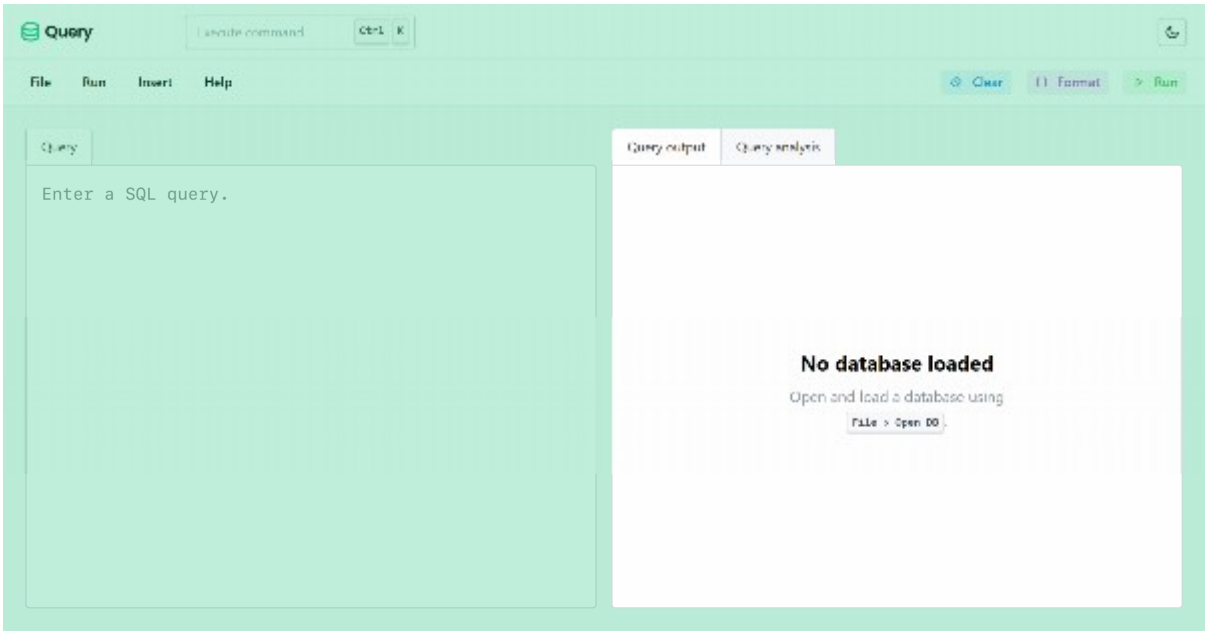
Рис. 3.13. Вкладка, де користувачі можуть писати запити

Вкладка запитів – це місце, де користувачі можуть писати запити. Він підтримує підсвічування синтаксису, щоб полегшити користувачам написання запитів. Ми використали моноширинний шрифт для вкладки запиту, оскільки він є звичайним для редагування коду. Оскільки цей компонент використовує вбудований тег введення HTML, він підтримує

комбінацію клавіш CTRL + Z для скасування та комбінацію клавіш CTRL + Y для повторення.



а)



б)

Рис. 3.14. Вкладки аналізу та результатів, де відображається поточний а) набір результатів і б) аналіз введеного запиту

Вкладки виводу, що складаються з виводу запиту та виводу аналізу запиту, є місцями, де друкуються виведення. Вихід запиту міститиме набір

результатів введеного запиту, коли завантажується база даних SQLite і виконується команда запуску запиту. Якщо база даних SQLite не завантажена, на цій вкладці буде показано «Базу даних не завантажено». Вкладка аналізу запитів виведе аналіз. Коли антипатерн буде знайдено, він відобразить його та відобразить розташування та опис антипатерну. Якщо антипатерни не знайдено, буде показано «Антипатерни не знайдено».

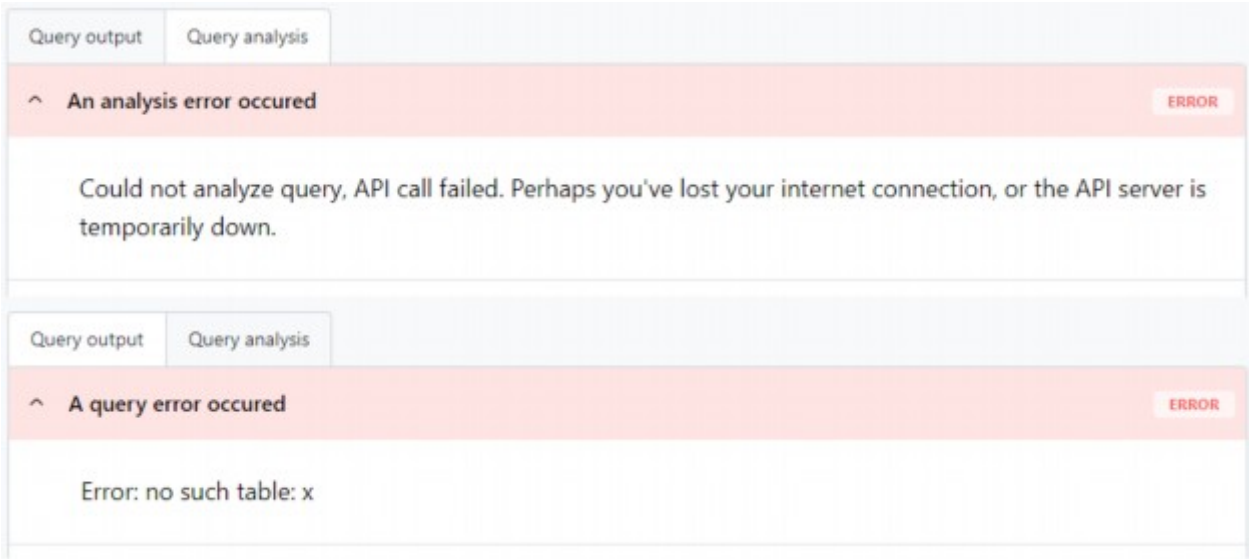


Рис. 3.15. Помилки, які відображаються на вкладках виводу, коли щось пішло не так

Обидві вкладки відображатимуть помилки відповідно до вимог S5 та S6, як показано на рисунку 3.15 . Якщо введений запит містить синтаксичну помилку, на вкладці виводу запиту буде відображено повідомлення про помилку. Це повідомлення про помилку пояснює помилку та місце її виникнення. Інше повідомлення про помилку відображатиметься на вкладці аналізу запитів, якщо запит до серверного API не вдасться. Це також може бути спричинено синтаксичною помилкою в запиті або якщо серверна частина не працює. Ми вважаємо, що ці повідомлення про помилки є важливими функціями бо користувачеві має бути легко виявляти та виправляти свої помилки. Без цих помилок користувач не отримує жодного

візуального зворотного зв’язку щодо того, чому їхній запит чи аналіз не вдаються.

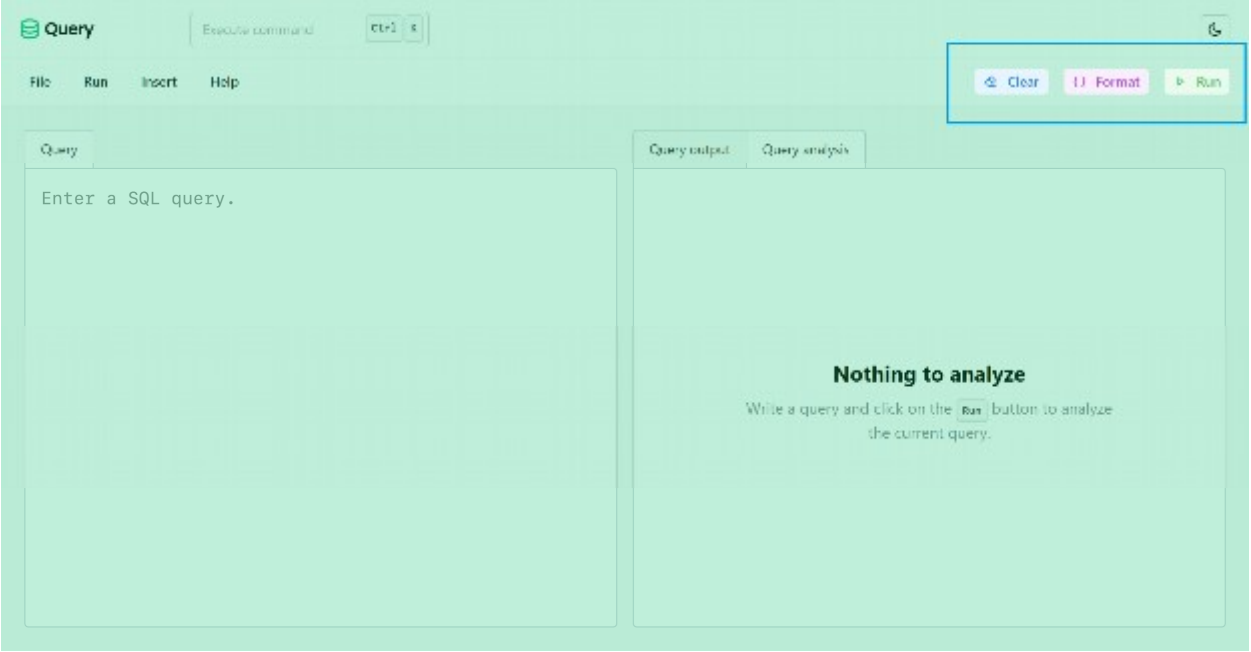


Рис. 3.16. Кнопки дій, які виконують дії запиту очищення, форматування та виконання відповідно

Кнопка «Очистити» використовується для видалення всіх вхідних і вихідних даних. Це означає, що буде видалено не лише запит, записаний на вкладці запиту, але й наявні результати на вкладках виводу запиту та аналізу запиту. Як зазначено в літературі, користувачі часто допускають помилки при виконанні дій. Чітка дія забезпечує «аварійний вихід», або іншими словами, спосіб швидко скасувати помилку та почати все заново. Кнопка «формат» використовується для форматування поточного запиту, що дозволяє користувачеві переглядати запит у більш зручному для читання форматі. Кнопка «бігти» виконує дві послідовні дії. Якщо базу даних завантажено, вона візьме поточний введений запит, запустить його з базою даних і відобразить набір результатів на вкладці виводу запиту. По-друге, він зробить запит POST до API SQL для аналізу запиту. Потім він виведе відповідь на вкладці аналізу запиту. Якщо база даних не завантажена, ця дія лише аналізуватиме запит. Кнопки дій відрізняються від решти інтерфейсу,

оскільки вони мають помітний колір. Через це ми вважаємо, що помітні кнопки дій допоможуть зменшити навантаження на пам’ять користувача.

Усі три дії зіставляються з комбінацією клавіш. У випадку операційних систем Windows і Linux CTRL + U використовується для запуску команди очищення, CTRL + B використовується для запуску команди форматування, а CTRL + Enter використовується для запуску запиту. У Mac OS відповідними комбінаціями клавіш є Command + U, Command + B і Command + Enter. Вони надають користувачеві механізм для швидкого й ефективного виконання дій без використання вказівного пристрою для вибору дії з меню.

Як було зазначено в попередньому розділі, користувач може завантажити базу даних за допомогою спадного меню файлів на панелі інструментів програми.

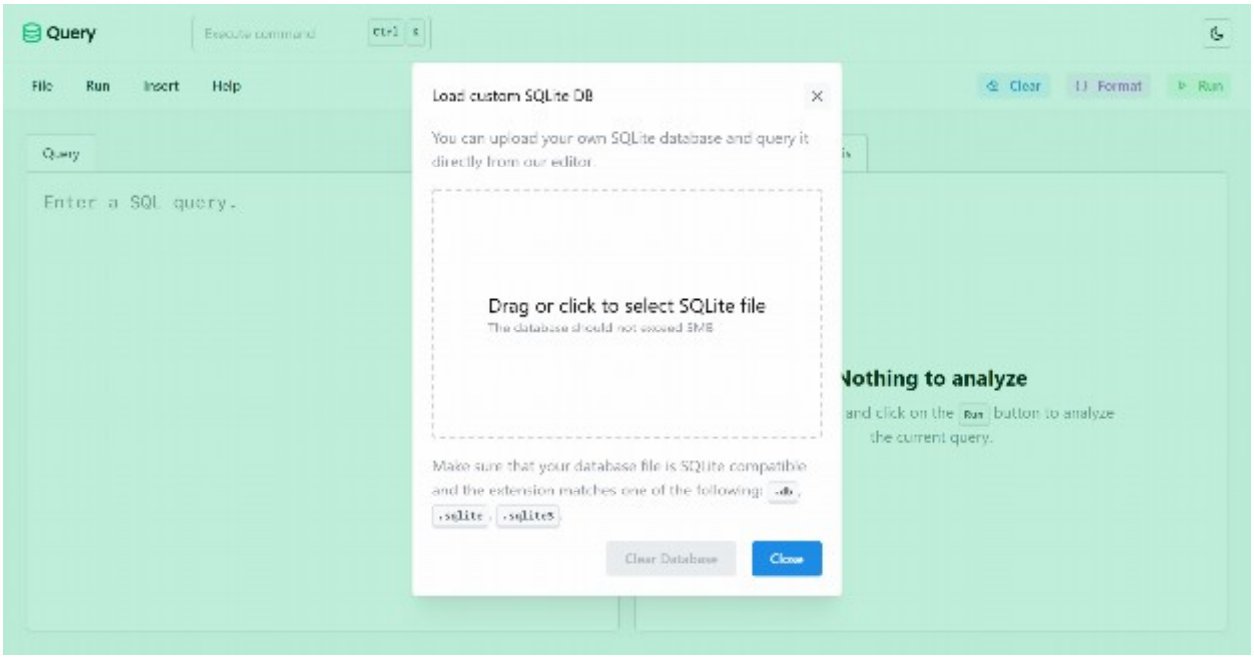


Рис. 3.17. Модальне вікно, що дозволяє користувачеві завантажувати базу даних SQLite

Модальний режим, схожий на спливаюче діалогове вікно, яке відображається в центрі екрана, використовується для того, щоб користувач міг вибрати базу даних для завантаження. Модальний показує назву файлу та назву бази даних, а також розмір бази даних. Користувач також має

можливість очистити поточну базу даних. У цьому випадку база даних видаляється з програми, а користувач повертається до головного перегляду. Ми вирішили дозволити користувачеві виконувати завантаження та очищення баз даних у модальному режимі, оскільки фокус зміщується на модальний режим і користувач може виконувати лише дії, пов’язані з базою даних. Ми вирішили використовувати модальні над спливаючими вікнами, оскільки модальні зазвичай не відхиляються автоматично, і таким чином користувач все одно може скасувати дію.

Отже, ми представили дизайн інтерфейсу користувача та UX, а потім надали деякі важливі принципи та евристики. Потім ми представили наші вимоги, які були частково сформовані з використанням принципів дизайну. Потім ми представили архітектуру програми та основні функції.

Висновки до розділу

Отже, в цьому розділі представлено підходи до імплементації моделей і методів для виявлення антипатернів у SQL, що охоплює як класифікацію антипатернів, так і розробку інструменту для їх автоматичного виявлення. Здійснено визначення типів та особливостей антипатернів, що дозволило систематизувати їх і підготуватися до їхньої практичної реалізації. Описано процес розробки інструменту, включаючи використання Python-пакетів для командного рядка та інтеграцію з RESTful API, що розширює функціональні можливості і забезпечує гнучкість застосування. Особливу увагу приділено побудові зручного інтерфейсу для користувача, що спрощує взаємодію з інструментом. Зрештою, визначено основні вимоги до функціонування інструменту, які було враховано при розробці архітектури та дизайну, що робить його придатним для практичного використання в різних середовищах і підвищує його ефективність у виявленні антипатернів SQL.

ВИСНОВКИ

У магістерській роботі було проведено дослідження виявлення антипатернів у SQL, зокрема аналіз теоретичних основ, розробку моделей і методів, а також імплементацію інструменту для їх автоматичного виявлення. Основні результати дослідження можна сформулювати таким чином:

Аналіз предметної області дозволив визначити поняття антипатернів в SQL, охарактеризувати їхні особливості, а також дослідити основи баз даних і систем управління ними. Вивчення структури таблиць, відносин між таблицями, індексів та об'єктно-реляційного відображення створило необхідне підґрунтя для подальшого виявлення типових SQL-антипатернів, таких як "Неоднозначні групи", "Loop to Join", "Неявні рядки" тощо. Це дало можливість систематизувати антипатерни, їх характеристики та наслідки використання, що стало основою для подальших досліджень.

Аналіз існуючих моделей, методів та інструментів продемонстрував, що поширеність антипатернів у SQL-коді є суттєвою проблемою. Було проведено огляд літературних джерел, які висвітлюють питання продуктивності та ефективності SQL, а також вивчено інструменти для виявлення антипатернів, що охоплюють як вбудовані засоби СУБД, так і спеціалізовані аналізатори коду. Також були розглянуті методи виявлення антипатернів, які включають статичний та динамічний аналіз SQL-коду. Це дозволило обрати оптимальні методи для розробки власного інструменту.

Розробка та імплементація інструменту для виявлення антипатернів стали основним практичним результатом роботи. Було визначено класи та особливості антипатернів, реалізовано інструмент із використанням Python-пакету для командного рядка та RESTful API, що забезпечує гнучкість та зручність використання. Було створено інтуїтивно зрозумілий інтерфейс, який полегшує взаємодію користувача з інструментом. Також розроблено архітектуру інструменту з урахуванням основних вимог до його

функціонування, що робить його ефективним для виявлення антипатернів у SQL-коді.

Таким чином, результати роботи вносять вагомий внесок у підвищення якості та продуктивності SQL-запитів шляхом розробки інструменту для автоматичного виявлення антипатернів, що може бути корисним для розробників, аналітиків і адміністраторів баз даних.

ПЕРЕЛІК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Brown, W. H., Malveau, R. C., McCormick III, T. J., & Mowbray, T. J. AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. Wiley, 1998.
2. Karwin, B. SQL Antipatterns: Avoiding the Pitfalls of Database Programming. Pragmatic Bookshelf, 2010.
3. Celko, J. SQL for Smarties: Advanced SQL Programming. Morgan Kaufmann, 2014.
4. Celko, J. Joe Celko's Thinking in Sets: Auxiliary, Temporal, and Virtual Tables in SQL. Morgan Kaufmann, 2008.
5. SQLator - An online SQL learning workbench. pages 223–227, 2004.
6. Albrecht A. sqlparse. Available at https://pypi.org/project/sqlparse/. 35
7. Alireza Ahadi, Julia Prior, Vahid Behbood, and Raymond Lister. Students' semantic mistakes in writing seven different types of sql queries. In Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education, ITiCSE '16, page 272–277, New York, NY, USA, 2016. Association for Computing Machinery
8. Stonebraker, M., & Hellerstein, J. M. Readings in Database Systems. MIT Press, 2005.
9. Fehily, C. SQL: Visual QuickStart Guide. Peachpit Press, 2008.
10. Molinaro, A. SQL Cookbook: Query Solutions and Techniques for Database Developers. O'Reilly Media, 2005.
11. Date, C. J. An Introduction to Database Systems. Addison-Wesley, 2003.
12. Elmasri, R., & Navathe, S. B. Fundamentals of Database Systems. Pearson, 2016.
13. Connolly, T., & Begg, C. Database Systems: A Practical Approach to Design, Implementation, and Management. Pearson, 2014.

14. Natalia Arzamasova, Martin Schäler, and Klemens Böhm. Cleaning antipatterns in an sql query log. IEEE Transactions on Knowledge and Data Engineering, 30(3):421–434, 2018.
15. Stefan Brass and C. Goldberg. Semantic errors in sql queries: A quite complete list. Pages 250–257, 10 2004.
16. Fowler, M. Patterns of Enterprise Application Architecture. Addison-Wesley, 2002.
17. Hellerstein, J. M., Stonebraker, M., & Hamilton, J. Architecture of a Database System. Foundations and Trends in Databases, 2007.
18. Redgate. SQL Server Execution Plans. Redgate, 2013.
19. Lewis, C., & Barroso, L. A. SQL Performance Explained. Markus Winand, 2012.
20. Gulutzan, P., & Pelzer, T. SQL Performance Tuning. Addison-Wesley, 2003.
21. John Brooke. ”SUS-A quick and dirty usability scale.” Usability evaluation in industry. CRC Press, June 1996. ISBN: 9780748404605. 62, 65
22. Donald D. Chamberlin and Raymond F. Boyce. Sequel: A structured english query language. In Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control, SIGFIDET ’74, page 249–264, New York, NY, USA, 1974. Association for Computing Machinery.
23. Lahdenmäki, H., & Leach, M. Relational Database Index Design and the Optimizers. Wiley, 2005.
24. Kimball, R., & Ross, M. The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling. Wiley, 2013.
25. Linwood, J., & Minter, R. Professional SQL Server Analysis Services 2005 with MDX. Wiley, 2006.
26. Melton, J., & Simon, A. R. Understanding the New SQL: A Complete Guide. Morgan Kaufmann, 2001.

27. Ramakrishnan, R., & Gehrke, J. Database Management Systems. McGraw-Hill, 2003.
28. Dai Clegg and Richard Barker. Case Method Fast-Track: A Rad Approach. Addison-Wesley Longman Publishing Co., Inc., USA, 2004.
29. Jakob Nielsen. 10 usability heuristics for user interface design, Apr 1994.
30. D. Norman. The Design of Everyday Things: Revised and Expanded Edition. Basic Books, 2013.
31. William C. Ogden and Susan R. Brooks. Query languages for the casual user. In Proceedings of the SIGCHI conference on Human Factors in Computing Systems - CHI '83, pages 161–165, New York, New York, USA, 1983. ACM Press.
32. E. F. Codd. A relational model of data for large shared data banks. Commun. ACM, 13(6):377–387, jun 1970.
33. Prashanth Dintyala, Arpit Narechania, and Joy Arulraj. SQLCheck: Automated detection and diagnosis of SQL anti-patterns. In Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. ACM, jun 2020.
34. Haerder, T., & Reuter, A. Principles of Transaction-Oriented Database Recovery. ACM Computing Surveys, 1983.
35. Kimball, R. The Data Warehouse ETL Toolkit: Practical Techniques for Extracting, Cleaning, Conforming, and Delivering Data. Wiley, 2004.
36. O'Neil, P., & O'Neil, E. Database: Principles, Programming, and Performance. Morgan Kaufmann, 2001.
37. Russell, S. J., & Norvig, P. Artificial Intelligence: A Modern Approach. Pearson, 2016. (Relevant for query optimization methods).
38. Silberschatz, A., Korth, H. F., & Sudarshan, S. Database System Concepts. McGraw-Hill, 2011.
39. Jeffrey E. F. Friedl. Mastering Regular Expressions. O'Reilly, Beijing, 3 edition, 2006.

40. Peter M. D. Gray. OQL, pages 2003–2004. Springer US, Boston, MA, 2009.
41. Jeffries, R., & Anderson, A. Effective SQL: 61 Specific Ways to Improve Your SQL. Addison-Wesley, 2016.
42. Tiwari, M. Mastering SQL Queries for SAP Business One. Packt Publishing, 2011.
43. Adams, P., & Vasic, L. SQL Server Query Performance Tuning. Apress, 2019.
44. Lavin, P. SQL for Data Scientists: A Beginner's Guide for Building Datasets for Analysis. O'Reilly Media, 2020.
45. Maurice H. Halstead. Elements of Software Science (Operating and Programming Systems Series). Elsevier Science Inc., USA, 1977.
46. Bill Karwin. SQL Antipatterns: Avoiding the Pitfalls of Database Programming. Pragmatic Bookshelf, 1st edition, 2010.
47. Andrew Koenig. Patterns and antipatterns. J. Object Oriented Program., 8(1):46–48, 1995.
48. Yingjun Lyu, Ali Alotaibi, and William G. J. Halfond. Quantifying the performance impact of sql antipatterns on mobile applications. In 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 53–64, 2019.
49. Yingjun Lyu, Sasha Volokh, William G. J. Halfond, and Omer Tripp. Sand: A static analysis approach for detecting sql antipatterns. In Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2021
50. Viescas, J. R., & Hernandez, M. SQL Queries for Mere Mortals: A Hands-On Guide to Data Manipulation in SQL. Addison-Wesley, 2014.
51. Karwin, B. SQL Performance Explained. Markus Winand, 2013.
52. Sadalage, P. J., & Fowler, M. NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence. Addison-Wesley, 2012.

53. Neward, T. The Vietnam of Computer Science. Communications of the ACM, 2008. (Explores database antipatterns in a critical context).
54. McLaughlin, B. Java and XML Data Binding. O'Reilly Media, 2002. (Covers XML antipatterns that can intersect with SQL use cases).
55. Csaba Nagy and Anthony Cleve. A static code smell detector for sql queries embedded in java code. In 2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM), pages 147–152, 2017