КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

ІМЕНІ ТАРАСА ШЕВЧЕНКА

Факультет комп'ютерних наук та кібернетики

Кафедра математичної інформатики

Кваліфікаційна робота

на здобуття ступеня бакалавра

за спеціальністю 122 Комп'ютерні науки

на тему:

РОЗРОБКА СТРУКТУРИ СИСТЕМНОГО КАТАЛОГА ДЛЯ СУБД НА ОСНОВІ РОЗШИРЕНОЇ РЕЛЯЦІЙНОЇ АЛГЕБРИ

Виконав студент 4-го курсу
ПРИНЬКО Максим ___________
(підпис)

Науковий керівник:
доцент, кандидат фізико-математичних наук
КУЛЯБКО Петро

____
(підпис)

Засвідчую, що в цій роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент ___________
(підпис)

Роботу розглянуто й допущено до захисту на засіданні кафедри математичної інформатики
«___»_______________2023 р.
протокол №____
Завідувач кафедри
Василь ТЕРЕЩЕНКО __________
(підпис)

Київ – 2023

РЕФЕРАТ

Обсяг роботи: 40 сторінок, 10 ілюстрацій, 0 таблиць, 5 використаних джерел, 0 додатків.

Об’єкт розроблення: системний каталог

Мета роботи: розробити системний каталог з урахуванням переваг та недоліків існуючих моделей СКБД

Результати: розроблено системний каталог для СКБД на основі розширеної реляційної алгебри. Цей додаток поєднує основні переваги реляційної моделі, але замість символьної адресації використовує прямі вказівники

ЗМІСТ

ВСТУП 3
РОЗДІЛ 1. Огляд та порівняння моделей баз даних 5
1.1. Підхід CODASYL 5
1.2. Реляційні бази даних 7
1.2.1 Первинні ключі 8
1.2.2 Зовнішні ключі. Типи зв’язків 10
1.3 Порівняння реляційного підходу та підходу CODASYL 14
1.4 Недоліки реляційного підходу 16
1.5 Об'єктно-орієнтований підхід 21
РОЗДІЛ 2. Особливості мови Go 26
РОЗДІЛ 3. Розробка системного каталога 31
3.1 Опис структури 31
3.2 Реалізація системного каталога 33
3.3 Робота з бінарними файлами 35
3.4 Основні функції системного каталога: 37
3.5 Функції, підтримувані системним каталогом 38
ВИСНОВКИ 39
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 40

ВСТУП

Оцінка сучасного стану. Реляційна модель баз даних є одним з найпоширеніших та найрозвиненіших підходів до організації даних. Вона використовує таблиці зі зв'язками між ними, представленням даних у вигляді реляційних відношень і використовує мову запитів SQL для маніпулювання та отримання даних. Реляційні бази даних забезпечують структурованість, цілісність та стабільність даних.

Реляційний підхід спрощує структуру даних до плоского файлу і забезпечує високий рівень маніпулювання даними, що дозволяє математично формалізувати його. Проте, перехід від складних структур до реляційних може призводити до зниження ефективності оброблення, особливо для простих запитів, через використання символьної адресації в реляційних базах, яка може бути повільнішою порівняно з відносною адресацією в базах даних з складними структурами.

З розвитком технологій великих даних (BIG DATA) та хмарних обчислень (Cloud Computing) проблема ефективності оброблення даних стає особливо актуальною. Частковим розв'язанням цієї проблеми може бути використання різних типів зв'язків між даними залежно від швидкості їх оброблення, від повільних, але гнучких (характерних для реляційних СКБД), до швидких, але менш гнучких на прямих вказівниках (характерних для дореляційних СКБД).

У другій половині 60-х років було запропоновано перехід від роботи зі структурованими файлами до СКБД зі складними структурами даних. Модель DBTG CODASYL містить мову опису даних (DDL) та мову маніпулювання даними (DML). DDL CODASYL базується на записах у стилі мови COBOL і використовує набори даних (DataSet) для швидкої локалізації необхідних даних або знаходження місця для додавання нових даних. Зазвичай, СКБД мали доступ лише до однієї позиції читання в одному файлі. Проте підхід CODASYL передбачав можливість запам’ятовувати поточну позицію читання для кожного

набору даних, що в окремих ситуаціях давало значне пришвидшення. Однак, відносно низький рівень DML CODASYL виявився занадто складним для користувачів, і цей недолік був подоланий застосуванням реляційного підходу, де мова запитів SQL стала більш зручною для більшості користувачів.

Метою запропонованої роботи є реалізація системного каталога мовою Go на основі алгебри, яка оперує реляційними відношеннями та наборами даних зі складними структурами. Це доповнення до класичної реляційної алгебри. Системний каталог буде підтримувати операції, схожі на реляційні, але з використанням відносної адресації в структурах даних для досягнення більш ефективних обчислень.

РОЗДІЛ 1. Огляд та порівняння моделей баз даних

1.1. Підхід CODASYL

CODASYL (англ. COnference on DAta SYstems Languages — Конференція по мовам систем обробки даних) була організацією, що грала важливу роль у розвитку інформаційних технологій у 60-80-их роках XX століття. Вона була створена у 1959 році з метою розробки стандартних мов програмування, які могли б використовуватися на різних комп'ютерах. CODASYL зосереджувалася на сприянні ефективному аналізу, проектуванню та впровадженню систем обробки даних і залучала представників промисловості та влади, що займалися цією сферою.

Організація видала багато специфікацій на різних мовах, використовуючи офіційні органи стандартизації, такі як ISO, ANSI та їх попередники. Однак CODASYL також став відомим як назва мережевої моделі даних та окремо як незалежно розробленої ієрархічної бази даних фірми North American Rockwell, яку пізніше IBM взяли за основу для власної БД IMS.

CODASYL закріпився за мережевою моделлю даних завдяки роботі групи Data Base Task Group (DBTG) CODASYL, яка опублікувала специфікації розширення COBOL для обробки баз даних у 1969 році. Ці специфікації включали мову опису даних (DDL) для визначення схеми бази даних і мову маніпулювання даними (DML), яка використовувалася для роботи з даними в базі даних. Хоча робота була спрямована на COBOL, з'явилася ідея незалежної мови баз даних, яка отримала розвиток завдяки компанії IBM та їхньому продукту PL/I.

У 1971 році, з метою створення незалежних мов програмування, робота над специфікаціями була розділена на дві групи: Комітет DDL продовжував розробку мови опису даних, а Комітет мови COBOL розробляв специфікації мови маніпулювання даними. Однак цей розкол призвів до того, що цим групам не вдавалось синхронізувати специфікації. Тому компаніям, які використовували цей

продукт, доводилось самостійно виправляти відмінності. Наслідком стала відсутність сумісності між їх реалізаціями.

1.2. Реляційні бази даних

Реляційні бази даних (РБД) є одним з найпоширеніших типів баз даних. Вони базуються на реляційній моделі даних, яку розробив Едгар Кодд у 1970-х роках. Такі бази даних призначені для зберігання та організації великої кількості структурованих даних, забезпечуючи ефективний та надійний доступ до інформації. Реляційні системі керування базами даних (РСКБД) широко використовуються у різних галузях та додатках, починаючи від невеликих особистих проектів і до систем корпоративного рівня.

У реляційній моделі дані представлені у вигляді таблиць, які складаються з рядків (кортежів) і стовпців (атрибутів). Рядки представляють окремі екземпляри даних, а стовпці визначають типи даних, які складаються з рядків у таблиці. Зв'язки між таблицями встановлюються за допомогою ключів - первинних та зовнішніх.

Однією з ключових переваг РСУБД є механізми для визначення правил та обмежень даних, щоб гарантувати, що зберігається лише дійсна та значуща інформація. Це дозволяє реляційним базам даних забезпечити цілісність та узгодженість.

Одним із таких механізмів є первинний ключ (Primary Key): кожна таблиця в РБД має стовпець або комбінацію стовпців, які ідентифікують унікальний запис в таблиці. Використання первинного ключа запобігає дублюванню даних і гарантує унікальність записів.

Зовнішній ключ (Foreign Key) дозволяє встановлювати зв'язок між двома таблицями в РБД. Він використовується для посилання на записи в іншій таблиці. Зовнішній ключ забезпечує цілісність даних, оскільки він гарантує, що записи в посиланій таблиці існують і не можуть бути видалені, якщо на них є посилання з інших таблиць.

Транзакції в РБД дозволяють групувати одну або декілька послідовних операцій з базою даних в один логічний блок. Якщо транзакція не може бути завершена успішно (наприклад, через помилку або виключну ситуацію), її можна відмінити, щоб забезпечити узгодженість даних. Транзакції гарантують атомарність, цілісність, узгодженість та надійність даних.

1.2.1 Первинні ключі

Первинний ключ (Primary Key) є одним із фундаментальних понять реляційної моделі даних. Він використовується для унікальної ідентифікації кожного запису в таблиці реляційної бази даних. Для кожної таблиці може бути визначений лише один первинний ключ.

Основні характеристики первинного ключа:

1. Кожне значення первинного ключа в таблиці повинно бути унікальним, не повторюватись в інших записах таблиці . Це дозволяє однозначно ідентифікувати кожен запис в таблиці.
2. Первинний ключ не повинен бути ідентичним або конфліктним з будь-яким іншим стовпцем або комбінацією стовпців в таблиці.
3. Значення первинного ключа не може бути зміненим або модифікованим після того, як запис був вставлений в таблицю. Це гарантує сталу ідентифікацію запису протягом усього життєвого циклу даних.
4. Первинний ключ повинен бути якомога більш мінімальним, тобто складатись з найменшої кількості стовпців або комбінації стовпців, необхідних для унікальної ідентифікації запису.
5. Значення первинного ключа не можуть бути порожніми або нульовими. Кожен запис в таблиці повинен мати непорожнє значення первинного ключа.

Первинний ключ встановлюється під час проектування бази даних і може визначатися наступними способами:

1. Використання автоматично збільшуваного числового значення (наприклад, цілочисельного ID) в якості первинного ключа.
2. Використання унікального символьного рядка або комбінації рядків як первинного ключа.
3. Використання комбінації двох або більше стовпців як первинного ключа. Такий ключ називається комбінованим.
4. Використання значення зовнішнього ключа з іншої таблиці як первинного ключа.

Розглянемо, наприклад, таблицю Робітник (Employee). У своїх атрибутах вона зберігає інформацію про ім’я, прізвище, адресу електронної пошти, номер телефону, дату працевлаштування, зарплатню та ID менеджера і ID відділу (останні два, дуже ймовірно, будуть зовнішніми ключами).

Employee	
EmployeeID	*int* PK
FirstName	*nvarchar(50)*
LastName	*nvarchar(50)*
EMail	*varchar(30)*
Phone	*varchar(15)*
HireDate	date
ManagerID	int
Salary	float
DepartmentID	int

Рисунок 1. Опис таблиці Робітник (Employee)

Первинний ключ грає важливу роль в реляційних базах даних, дозволяючи ефективно здійснювати пошук, оновлення та видалення даних. Він також використовується для встановлення зв'язків між таблицями за допомогою зовнішніх ключів.

1.2.2 Зовнішні ключі. Типи зв’язків

Зовнішній ключ (Foreign Key) є важливим елементом реляційної моделі даних, використовуваним для встановлення зв'язку між двома таблицями в реляційній базі даних. Він вказує на залежність між стовпцями або комбінаціями стовпців однієї таблиці від значень стовпців іншої таблиці.

Основні характеристики зовнішнього ключа:

1. Відношення між таблицями: Зовнішній ключ встановлює зв'язок між двома таблицями, який визначає, які записи з однієї таблиці посилаються на записи в іншій таблиці.
2. Цілісність даних: Зовнішній ключ допомагає забезпечити цілісність даних у базі даних. Він гарантує, що значення стовпця (або комбінації стовпців), який відповідає зовнішньому ключу в дочірній таблиці, існують у відповідному стовпці (або комбінації стовпців) первинного ключа у батьківській таблиці.
3. Заборона видалення або зміни даних: Зовнішній ключ може бути налаштований забороняти видалення або зміну записів у батьківській таблиці, якщо на них є посилання з дочірньої таблиці. Це забезпечує дотримання залежності і запобігає втраті даних або порушенню цілісності.
4. Допустимість значень NULL: Зовнішній ключ може допускати значення NULL, що означає, що стовпець (або комбінація стовпців), що посилається на первинний ключ, може не мати значення. Це дозволяє створювати необов'язкові зв'язки між таблицями.
5. Автоматичне оновлення: Зовнішній ключ може бути налаштований таким чином, що при зміні значення первинного ключа в батьківській таблиці автоматично оновлюється відповідне значення зовнішнього ключа у дочірній таблиці.

Зовнішній ключ визначається в дочірній таблиці і посилається на первинний ключ в батьківській таблиці. Це встановлює зв'язок між цими таблицями. Зовнішні ключі допомагають забезпечити цілісність даних, підтримувати зв'язки між таблицями та спрощувати операції зв'язків, такі як об'єднання та запити до пов'язаних даних.

Розглянемо типи відношень, які можуть бути встановлені за допомогою зовнішнього ключа:

1. Один до одного (One-to-One): В цьому типі відношення кожен запис в одній таблиці посилається лише на один запис в іншій таблиці, і навпаки. Зовнішній ключ використовується для зв'язку двох таблиць, забезпечуючи відношення один до одного між ними.
   Наприклад, таблиця Робітник (Employee) може бути пов'язана з таблицею ДеталіРобітника (EmployeeDetails), що зберігає інформацію про паспортні дані, відношенням Один до одного. Як бачимо, зовнішній ключ дочірньої таблиці є первісним ключем обох таблиць, не тільки батьківської. Але це не обов'язково – дуже часто дочірня таблиця також містить окремий атрибут або множину атрибутів, які утворюють власний первинний ключ дочірньої таблиці.

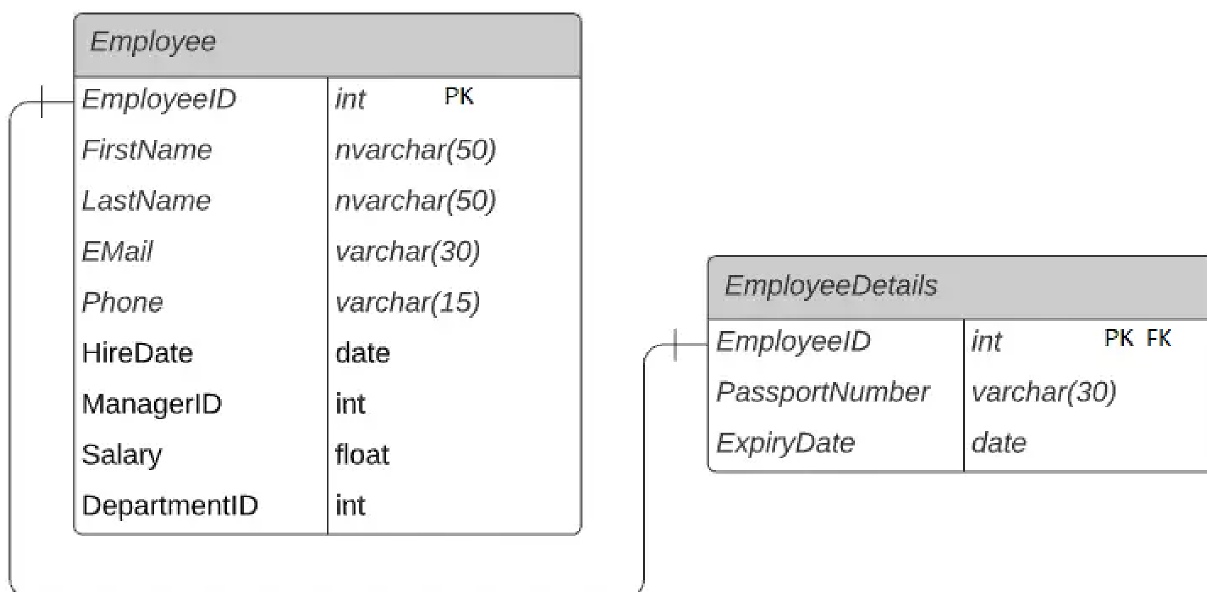

Рисунок 2. Опис таблиць зі зв'язком Один до одного

2. Один до багатьох (One-to-Many): Це найпоширеніший тип відношення, при якому кожен запис в одній таблиці може посилатись на багато записів в іншій таблиці. Також цей запис може не посилатися на жоден запис іншої таблиці. Зовнішній ключ встановлюється в таблиці, що містить багато екземплярів в цьому відношенні, щоб посилатись на первинний ключ в таблиці, що містить лише один екземпляр.
   Наприклад, таблиця Робітник (Employee) може бути пов'язана з таблицею Адреса (Address) відношенням Один до багатьох. Кожен робітник може мати декілька адрес (наприклад, домашня адреса, адреса офісу тощо). Атрибут EmployeeID є первинним ключем у батьківській таблиці і зовнішнім у дочірній. Таким чином, всі адреси одного робітника матимуть однаковий зовнішній ключ, але різні первинні ключі AddressID.

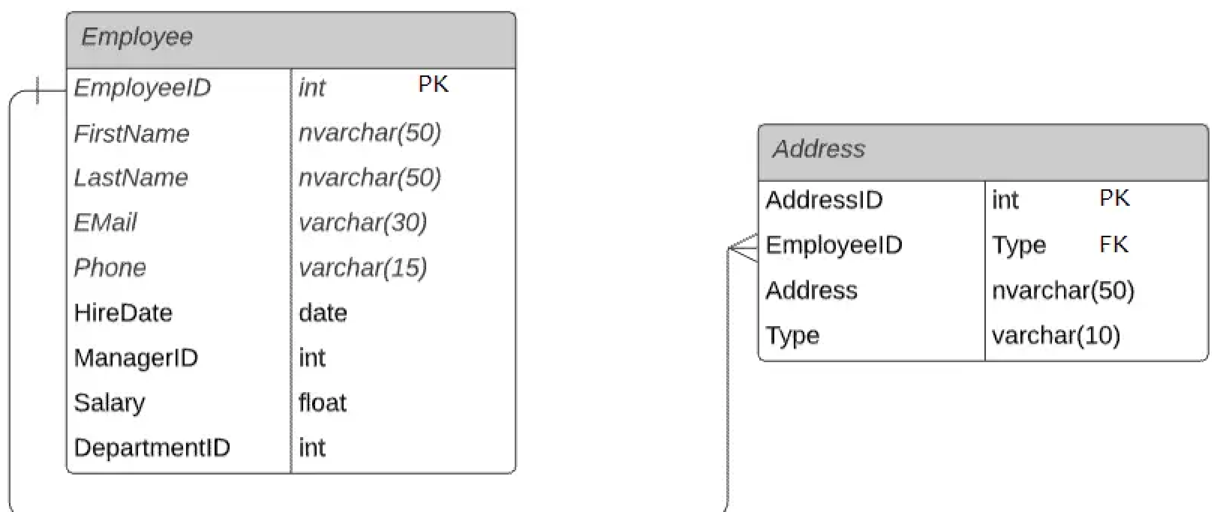

Рисунок 3. Опис таблиць зі зв'язком Один до багатьох

3. Багато до багатьох (Many-to-Many): Цей тип відношення відображає зв'язок, де багато записів в одній таблиці посилаються на багато записів в іншій таблиці. Це досягається за допомогою додаткової таблиці, яка використовується як зв'язувальна таблиця, що містить зовнішні ключі обох таблиць.

Наприклад, якщо кожен робітник може мати декілька навичок, а кожною навичкою може володіти декілька робітників, то створюється допоміжна таблиця EmployeeSkill, яка містить стовпці зовнішніх ключів EmployeeID і SkillID для формування зв’язку «багато-до-багатьох» між таблицею Employee і SkillDescription. Окремо Employee і EmployeeSkill мають відношення один до багатьох й таблиці SkillDescription і EmployeeSkill мають відношення один до багатьох. Але вони формують відношення «багато-до-багатьох» за допомогою сполучної таблиц.

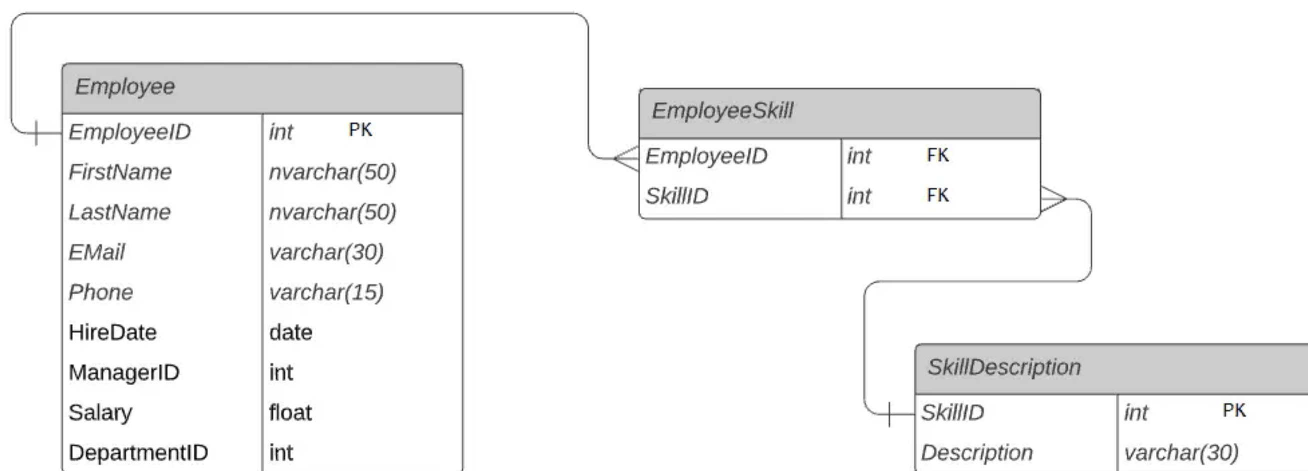

Рисунок 4. Опис таблиць зі зв’язком Багато до багатьох

1.3 Порівняння реляційного підходу та підходу CODASYL

Основна різниця між CODASYL та реляційною моделлю полягає в їхніх підходах до організації та маніпулювання даними.

1. Структура даних:
    a. CODASYL використовувала Network Data Model, де дані організовані у вигляді мережі записів і зв'язків між ними. Кожен запис має унікальний ідентифікатор та може мати посилання на інші записи. Зв'язки між записами встановлюються за допомогою спеціальних полів-посилань, що вказують на інші записи. Це дозволяє створювати складні структури даних з багатьма рівнями вкладеності.
    b. Реляційна модель баз даних організовує дані у вигляді таблиць (їх називають реляціями), де кожна таблиця має набір колонок і рядків. Відношення між таблицями встановлюються за допомогою первинних та зовнішніх ключів, що забезпечує структуровану та однорідну форму представлення даних.
2. Запити та мови програмування:
    a. CODASYL використовувала мову програмування DML (Data Manipulation Language) для отримання та обробки даних. DML надавала можливість робити складні запити, звертатися до зв'язків між записами та виконувати різноманітні операції.
    b. У реляційній моделі для маніпулювання даними використовується мова SQL (Structured Query Language). SQL дозволяє виконувати операції вибору, вставки, оновлення та видалення даних з реляційних таблиць. SQL також надає можливість виконувати складні запити з використанням умов, об'єднань, угрупувань та сортувань.

3. Нормалізація даних:

    a. CODASYL не надавала спеціальних правил або норм для нормалізації даних. Програмісти мали вільність організовувати дані у вигляді мережі, що може призводити до дублювання даних та проблем зі збереженням цілісності. Це вимагало від програмістів значно більшої відповідальності.

    b. Реляційна модель базується на нормальних формах, які встановлюють правила для структури даних з метою уникнення дублювання та збереження цілісності даних. Метою нормалізації є усунення недоліків структури БД, які призводять до шкідливої надмірності в даних, яка в свою чергу потенційно призводить до різних аномалій і порушень цілісності даних.

4. Складність інтеграції:

    a. Інтеграція даних між різними CODASYL-сумісними системами може бути складною через різні підходи до структури даних та мов програмування.

    b. Реляційна модель баз даних набула широкої популярності завдяки своїй стандартизації та широко підтримуваному SQL. Це спрощує інтеграцію даних між різними реляційними базами даних та додатками.

Обидва підходи мають свої переваги та недоліки і використовуються у певних контекстах. CODASYL була важливим етапом у розвитку баз даних, але реляційна модель стала більш широко прийнятою завдяки своїй простоті, стандартизації та ефективності в багатьох сценаріях.

1.4 Недоліки реляційного підходу

На сьогоднішній день реляційний підхід лежить в основі одних з найпоширеніших систем управління базами даних (СУБД), таких як MySQL, Oracle і Microsoft SQL Server. Він забезпечує ефективне та структуроване зберігання та маніпулювання даними, що робить їх незамінними для багатьох організацій та додатків. Але також реляційна модель даних має певні недоліки, які можуть бути вирішальними у певних обставинах:

1. *Слабкий зв'язок з об'єктами реального світу*: реляційна модель не завжди точно відображає складні структури або відносини між об'єктами реального світу. Наприклад, якщо в реальному світі існує об’єкт з ієрархічною структурою даних, представлення цього об’єкту за допомогою кількох реляцій може бути неефективним та складним для обробки запитів.

2. *Семантична перевантаженість:* реляційна модель використовує один тип конструкції (реляцію) як для представлення об'єктів, так і для представлення зв'язків між ними. Це особливо проблематично у випадках зв’язків типу Багато до багатьох, які відображаються з використанням додаткової реляції, що часто призводить до ускладнених структур даних. Це може ускладнити розуміння даних та підтримку моделі.

3. *Слабка підтримка обмежень цілісності і корпоративних обмежень:* Встановлення обмежень цілісності у реляційних базах даних зазвичай вимагає задання певних правил (зазвичай, предикатів), які не можна порушувати при взаємодії з базою даних. Не всі комерційні СКБД надають потужні інструменти для визначення таких обмежень. Хоча можна встановити обмеження на рівні застосування, це, звичайно, не є найкращим рішенням, оскільки створює певну загрозу для самої системи. Така ж ситуація із корпоративними обмеженнями.

4. Реляційна модель даних має вимогу до *однорідності структури даних*, яка включає горизонтальну і вертикальну однорідність. Горизонтальна

однорідність означає, що всі кортежі в реляції повинні мати однаковий набір атрибутів. Вертикальна однорідність вимагає, щоб значення в кожному стовбці реляції належали до одного домену, тобто мали однаковий тип даних. Кожне поле в таблиці повинна містити атомарне значення. Проте ця структура є надто жорсткою і неадекватною для багатьох сценаріїв. У деяких випадках, комерційні системи дозволяють зберігати великі бінарні об'єкти (Binary Large Objects, BLOB), але такими значеннями не можна оперувати або опрацьовувати окремо. Вони зберігаються як окремі файли поза базою даних, а база даних містить тільки посилання на них.

5. Іншим обмеженням реляційної моделі є *фіксований набір операцій*, які можна виконувати над множинами і кортежами. Не можна визначати нові операції, окрім розширення бібліотеки вбудованих функцій для постпошукової фази запиту. Це обмеження може бути проблематичним для деяких застосувань, наприклад, для геоінформаційних систем (GIS), які часто потребують спеціальних операцій для роботи з географічними об'єктами, такими як визначення відстані, перетини ліній, предикати включення та інші.

6. Організація *рекурсивних запитів* у реляційній моделі є складним завданням через обмеження, що накладаються на повторні групи значень. Реляційна модель не дозволяє використання запитів, що пов'язує реляцію саму з собою, тому потрібно використовувати додаткові стратегії або розширення для вирішення таких завдань. Одним з підходів є використання рекурсивних запитів з використанням спеціальних конструкцій, таких як спільна таблиця або рекурсивний CTE (Common Table Expression), які дозволяють ітеративно обробляти дані до досягнення бажаного результату.

Реляційний підхід не завжди ефективно вирішує потреби в наступних галузях:

1. Автоматизоване проектування. У базах даних для систем автоматизованого проектування (Computer-Aided Design - CAD), відповідно до особливостей проектів механічних чи електротехнічних конструкцій, необхідно враховувати наступні фактори:
    - в базі даних будуть зберігатися різноманітні типи даних, проте кількість екземплярів кожного типу може бути обмеженою;
    - проекти можуть мати значні розміри і складатися з відносно незалежних підпроектів;
    - проекти можуть змінюватися та розвиватися з часом, можуть виникати нові потреби, які не передбачалися на етапі початкового проектування;
    - оновлення даних можуть мати значний негативний вплив через топологічні залежності та функціональні зв'язки між компонентами;
    - часто одна компонента проекту може мати декілька альтернативних варіантів, тому потрібно вести облік версій та мати систему керування вибором конфігурацій для кожної компоненти;
    - робота над проектом може відбуватися в колективному режимі з великою кількістю учасників. Кінцевий продукт повинен бути згрунтованим та узгодженим, без суперечностей.
2. Автоматизоване виробництво. База даних для автоматизованого виробництва (Computer-Aided Manufacturing - CAM) містить дані, які схожі за характеристиками на CAD-системи. Також вона зберігає інформацію про результати виробництва, які можуть бути дискретними (наприклад, деталі) або неперервними (наприклад, продукти хімічного синтезу). Застосування CAM в хімічній промисловості широко використовуються для відстеження стану системи, таких як температура в реакторі, швидкість потоків та вихідний рівень продуктів реакції. Ці застосування мають працювати в режимі реального часу, щоб ефективно керувати процесами та

забезпечувати оптимальну продуктивність в умовах встановлених обмежень. Для реагування на різні умови експлуатації, такі застосування використовують стандартні процедури та параметризовані правила. Оператори можуть періодично змінювати ці правила для оптимізації продуктивності, і система повинна фіксувати ці зміни.
Системи CAM повинні зберігати великий обсяг даних з ієрархічною структурою та складними зв'язками між ними. Крім того, вони повинні мати інструменти для швидкого доступу до необхідних даних, їх перегляду та реагування на зміни.

3. Автоматизована розробка програмного забезпечення. Системи автоматизованої розробки програмного забезпечення (Computer-Aided Software Engineering - CASE) зберігають інформацію, пов'язану з різними етапами життєвого циклу програмного забезпечення. Ці етапи включають планування, збір і аналіз вимог, проектування, реалізацію, тестування, супровід та розробку документації. CASE-проекти, подібно до CAD-проектів, можуть бути великими і розроблятися в колективному режимі. Інструменти керування конфігурацією програмного забезпечення зазвичай дозволяють спільне використання схеми проекту, коду та документації, а також відстежувати залежності між цими компонентами та вносити потрібні зміни. За допомогою інструментів керування проектом можна координувати різні види діяльності, такі як планування складних незалежних задач, оцінювання витрат та моніторинг ходу виконання робіт.
4. Офісні інформаційні системи та мультимедійні системи. Дані про керування бізнес-інформацією, такі як документи, рахунки, електронні листи тощо, зберігаються в офісних інформаційних системах (Office Information System - OSI). У цій сфері типи даних охоплюють ширший спектр, ніж просто імена, адреси, дати та гроші. Сучасні системи працюють з різноманітними форматами, включаючи тексти будь-якого стилю, фотографії, діаграми, аудіо- та відеозаписи. Документи можуть мати складну структуру, яка описується за допомогою мови розмітки, такої як SGML (Standartized

Generalized Markup Language) чи HTML (HyperText Markup Language). Вони підтримують об'єктно-орієнтовані моделі даних, що дозволяє зберігати складні об'єкти та їх взаємозв'язки. При цьому вони також можуть використовуватися багатьма користувачами у розподіленому середовищі.

5. Геоінформаційні системи. Використання геоінформаційних систем (Geographic Information Systems - GIS) у землеустрої та підводних дослідженнях вимагає роботи з різними типами даних, які підтримують просторову та часову орієнтацію. Багато з цих даних отримуються через геологічні дослідження та аерофотозйомку і мають значний обсяг. Для пошуку і аналізу таких даних часто використовуються складні методи розпізнання образів на основі форми, кольору, текстури тощо.

Існує безліч інших спеціалізацій, в яких корисніше використовувати інші типи СКБД: системи управління вмістом (Content Management Systems - CMS), графічні системи та комп'ютерна графіка, обробка голосу та сигналів, Інтернет речей (Internet of Things - IoT) тощо. В таких випадках доцільніше використовувати, наприклад, об'єктно-орієнтовані бази даних (ООБД), адже вони дозволяють моделювати структурно складні об'єкти та їх взаємозв'язки, забезпечуючи ефективне зберігання та аналіз великого об’єму даних.

1.5 Об'єктно-орієнтований підхід

Основні концепції ООБД:

1. *Інкапсуляція* – концепція, що дозволяє збереження даних об’єкта і методів, які з ним пов'язані, разом в межах об'єкта. Це дозволяє приховувати внутрішні деталі і забезпечує доступ до них лише через визначений інтерфейс. Тобто, зовнішній код не може безпосередньо звертатися до внутрішніх даних, а доступ до них здійснюється через публічні методи, що визначені в інтерфейсі об'єкта. Внутрішні деталі об'єкта залишаються прихованими, що дозволяє змінювати його внутрішню структуру в певних межах, не впливаючи на зовнішнє використання. Інкапсуляція допомагає забезпечити захист даних в ООБД. Вона дозволяє контролювати, які дані можуть бути змінені або отримані з об'єкта, і забезпечує ізольованість даних від несанкціонованого доступу[1].
2. *Абстракція* дозволяє виділяти найважливіші характеристики об'єкта або системи і ігнорувати менш значущі деталі. Це спрощує процес розробки і дозволяє краще розуміти складні системи. У контексті ООБД, абстракція дозволяє працювати з базою даних на вищому рівні абстракції, ігноруючи складність внутрішньої структури та деталей реалізації. Це спрощує взаємодію з базою даних і дозволяє зосередитися на важливих аспектах, таких як моделювання даних, запити та маніпуляції з даними. Абстракція в ООБД реалізується за допомогою визначення класів та використання об'єктно-орієнтованого підходу. Кожен клас представляє абстрактний тип даних, який має свої властивості (атрибути) та поведінку (методи). За допомогою цих класів можна створювати об'єкти, які є конкретними представниками цих класів.
3. *Поліморфізм* відноситься до здатності об'єктів реагувати по-різному на одну й ту саму зовнішню операцію залежно від типу об'єкта, який його викликає. У контексті ООБД, поліморфізм виражається у можливості

використовувати спеціальнізовані типи об'єктів, які успадковують властивості та методи від загального батьківського типу. Це означає, що об'єкти підкласів можуть бути використані замість об'єктів батьківського класу, і викликати методи, які мають однакову назву, але можуть мати різну реалізацію для кожного конкретного підкласу. Поліморфізм в ООБД дозволяє реалізувати гнучкі структури даних та операцій з використанням універсальних методів, які можуть працювати з різними типами об'єктів. Це спрощує розробку та підтримку коду, сприяє повторному використанню і поліпшує зрозумілість програмного забезпечення.

4. *Наслідування* дозволяє створювати ієрархію класів, де підкласи успадковують властивості та методи від батьківського класу. У контексті ООБД, наслідування дозволяє створювати спеціалізовані типи об'єктів на основі загальних типів. При цьому, підкласи успадковують всі атрибути та функціональні можливості батьківського класу, а також можуть додавати власні унікальні властивості та методи. Наслідування в ООБД дозволяє створювати більш абстрактні та універсальні схеми даних, а також полегшує розширення та модифікацію бази даних. Підкласи можуть додавати нові атрибути та функціональні можливості, які специфічні для конкретного типу об'єктів, зберігаючи при цьому загальну структуру та функціонал батьківського класу. Наслідування в ООБД є потужним інструментом для моделювання складних взаємозв'язків та створення гнучких схем даних. Воно дозволяє покращити повторне використання коду, спростити розробку та підтримку системи та забезпечити більшу зрозумілість структури бази даних.

В об'єктно-орієнтованих системах, кожен об'єкт має свій поточний стан, який описується одним чи кількома атрибутами. Атрибути можуть бути простими або складними. Прості атрибути представляють примітивні типи даних, такі як цілі числа, рядки, логічні значення тощо. Вони зберігають конкретні значення, що описують окремі характеристики об'єкта. Складний атрибут може містити

колекції або посилання на інші об'єкти. Атрибут посилання реалізує зв’язок між об’єктами і концептуально є аналогом зовнішнього ключа реляційної моделі даних.

Для ідентифікації об'єктів в об'єктно-орієнтованих системах використовується ідентифікатор об'єкта (Object Identifier - OID). OID генерується системою і унікально ідентифікує конкретний об'єкт протягом життєвого циклу програми. OID не залежить від значень атрибутів об'єкта і не може бути використаний повторно, навіть якщо об'єкт був вилучений. Зазвичай, OID приховується від користувача і використовується системою для унікальної ідентифікації об'єктів.

OID в об'єктно-орієнтованих системах є унікальними в межах всієї системи, що відрізняється від унікальності значення первинного ключа в реляційній моделі даних, яке гарантується тільки на рівні однієї таблиці. У випадку використання OID для посилань між об'єктами, система повинна гарантувати відсутність обірваних посилань. Тобто, якщо перший об'єкт посилається на другий об'єкт за допомогою OID, то цей другий об'єкт повинен існувати в системі. Щодо реалізації, іноді в об'єктно-орієнтованих мовах програмування використовують адреси в пам'яті як OID. Проте, для баз даних цей підхід не є адекватним.

Відповідно, OID має ряд переваг, які впливають на ефективність та швидкодію системи:

- OID вимагають менше місця для зберігання порівняно з текстовими іменами, зовнішніми ключами або семантичними посиланнями. Вони зазвичай є числовими ідентифікаторами, які можуть бути збережені в малих розмірах пам'яті;
- OID надають прямий доступ до фактичної адреси або місця об'єкта в таблиці. Це дозволяє швидко знаходити об'єкти, незалежно від їхнього поточного місця зберігання, будь то в оперативній пам'яті або на диску. Замість пошуку за текстовими іменами або іншими значеннями, можна просто використовувати OID для ідентифікації і знаходження об'єктів;

- OID є інваріантом об'єкта, що означає, що їх не можна змінювати користувачем під час життєвого циклу об'єкта. Це дозволяє гарантувати стабільність ідентифікатора об'єкта і уникнути потенційних проблем, пов'язаних зі зміною ідентифікатора;
- OID не залежать від методів або атрибутів, що зберігаються в об'єкті. Це дозволяє системі бути гнучкою і незалежною від змін у структурі або властивостях об'єктів.

РОЗДІЛ 2. Особливості мови Go

Мова програмування Go (також відома як Golang) є відкритою мовою, розробленою в компанії Google. Вона була представлена у 2009 році і зарекомендувала себе як ефективний і простий засіб для розробки програмного забезпечення. Go був розроблений з метою підвищення продуктивності програмування в епоху багатоядерних мережевих машин і великих кодових баз. Розробники хотіли врахувати критику інших мов, що використовуються в Google, але зберегти їхні корисні характеристики.

Простота мови Go є однією з її ключових характеристик і була одним з головних принципів при її створенні:

1. Синтаксис Go є мінімалістичним, лаконічним і легким для читання. Він має низку простих правил і не містить складних конструкцій, що полегшує розуміння коду і сприяє швидкому освоєнню мови.
2. Мова Go має невелику кількість ключових слів, що робить її виразною і зменшує необхідність в запам'ятовуванні великої кількості концепцій.
3. Go намагається уникати зайвої складності і йде шляхом з меншим числом можливих способів виконання. Це сприяє зменшенню можливих помилок та забезпечує більш простий і передбачуваний спосіб програмування.
4. Go має статичну типізацію, що означає, що типи змінних повинні бути вказані явно. Це полегшує читання і розуміння коду, а також допомагає виявляти помилки на етапі компіляції.
5. Вбудована підтримка паралельного програмування: Мова Go має вбудовану підтримку горутин (goroutines) і каналів (channels), що дозволяють легко створювати конкурентні програми. Це робить асинхронне програмування і керування паралельними задачами простішими і зрозумілішими.[2]

Всі ці аспекти роблять мову Go простою для вивчення, розуміння і розробки програмного забезпечення. Вона спрямована на забезпечення простоти розробки і підтримки швидкого циклу розробки, що економить значні кошти. Разом з цим

простота мови Go не означає втрату потужності. Вона забезпечує багато корисних функцій і бібліотек, які допомагають розробникам реалізувати різноманітні завдання. Комбінація простоти та функціональності робить мову Go привабливою для розробників і допомагає збільшити продуктивність розробки програмного забезпечення.

Мова Go робить особливий акцент на *надійності* програмного забезпечення. Ось деякі аспекти, які сприяють стабільності мови Go:

1. Go використовує автоматичну систему збору сміття, яка відповідає за автоматичне виділення і звільнення пам'яті, яку більше не використовують програми. Це дозволяє уникнути багатьох типових помилок, пов'язаних з управлінням пам'яттю, таких як витоки пам'яті або недійсні вказівники.
2. Go має вбудовані механізми, які допомагають уникнути небезпечних станів (race conditions), які виникають коли правильність виконання програми залежить від послідовності або швидкості виконання окремих операцій при паралельному виконанні коду. Одним з способів запобігання race conditions є використання механізмів синхронізації, які дозволяють контролювати доступ до спільних ресурсів. В мові Go для цього використовуються м'ютекси (mutexes) і канали (channels), що дозволяють встановлювати блокування та передачу даних між горутинами в безпечній послідовності.
3. Go має статичну типізацію, що дозволяє виявляти багато помилок на етапі компіляції. Це допомагає зменшити можливі помилки виконання програми, зокрема, неправильні операції з типами даних, і допомагає забезпечити більш надійне виконання програм.
4. Go має чіткі правила форматування коду, які регулюють стиль написання програм. Це допомагає забезпечити єдність у вигляді коду і полегшує розуміння і підтримку кодової бази.
5. Go має вбудовану підтримку для написання тестів, що сприяє розробці надійних програм. Тестування є важливою складовою процесу розробки

програмного забезпечення і допомагає вчасно виявляти й виправляти помилки.

Не зважаючи на простоту у використанні, мова Go також відома своєю високою *швидкодією та ефективністю*:

1. Програми на Go компілюються безпосередньо в машинний код, що дозволяє їм працювати дуже ефективно. Це означає, що програми на Go не потребують інтерпретації або виконання в віртуальній машині, що знижує накладні витрати на виконання і поліпшує швидкодію.
2. Мова Go має вбудоване автоматичне управління пам'яттю (garbage collection), яке дозволяє ефективно виділяти та звільняти пам'ять. Воно визначає непотрібні об'єкти та автоматично звільняє пам'ять, що допомагає уникнути пам'яткових витоків та покращує продуктивність програміста.
3. Мова Go вбудовано підтримує паралельне виконання за допомогою горутин, що дозволяють виконувати функції паралельно, та каналів, які допомагають синхронізувати та обмінюватися даними між горутинами. Це дає можливість використовувати потужності багатоядерних систем та розподіляти завдання між ними, що поліпшує швидкодію програм.
4. Мова Go має вбудовану підтримку для мережевого програмування, що дозволяє легко створювати ефективні мережеві додатки. Завдяки стандартній бібліотеці Go для роботи з TCP/IP, HTTP, WebSocket та іншими протоколами, розробники можуть створювати швидкі та масштабовані мережеві програми.
5. Мова Go приділяє увагу ефективному використанню пам'яті та ресурсів. Її стандартна бібліотека та інструменти мають низький рівень накладних витрат та ефективно використовують ресурси системи. Крім того, простота мови Go сприяє зручності управління пам'яттю та ресурсами.

Загалом, мова Go позиціонується як мова з високою продуктивністю та швидкодією. Вона чудово підходить для розробки низькорівневих систем, веб-

серверів, мережевих додатків, паралельних програм та інших завдань, де важлива продуктивність та ефективність ресурсів.

Мова Go має декілька особливостей, що сприяють її *розширюваності*:

1. Go підтримує модульну структуру, що дозволяє розділяти код на незалежні компоненти – пакети (package) та модулі (module*)*. Кожен пакет може містити свою набір функцій, типів та методів, що дозволяє гнучко організовувати код. Модуль – це колекція пакетів, що зберігається в окремій директорії на файловій системі. Модулі можуть бути створені, оновлені та керовані за допомогою інструменту Go Modules, що спрощує керування залежностями та розширення проекту. Крім того, це полегшує повторне використання функціональності та розподілену розробку проектів.
2. Мова Go використовує інтерфейси для опису поведінки, що дозволяє створювати код, який може працювати з різними типами, не прив'язуючись до конкретної реалізації. Це дозволяє легко додавати нові реалізації і змінювати функціональність програми без необхідності змінювати багато коду.
3. Також Go дозволяє розширювати функціональність пакетів, надаючи можливість використовувати вбудовані типи (Embedded types) та обгортки (Wrappers) над існуючими типами. Розробники можуть створювати нові функції та методи, які розширюють функціональність вже існуючих пакетів без необхідності модифікувати їх вихідний код.
4. Go поставляється з багатофункціональною стандартною бібліотекою, яка включає в себе різноманітні пакети для взаємодії з мережею, обробки даних, роботи зі звуком, шифрування, тестування та багато іншого. Це спрощує розширення функціональності програми шляхом використання наявних пакетів зі стандартної бібліотеки або сторонніх бібліотек.
5. Мова Go має підтримку інтеграції з C-модулями, що дозволяє використовувати функції та типи, написаних на C. Це особливо корисно,

коли вам необхідно використати існуючий код на C або використати бібліотеки, які доступні тільки на C.

6. У Go можна використовувати рефлексію (reflection), що дозволяє аналізувати та маніпулювати структурами даних та функціями під час виконання програми. Це дозволяє створювати загальні алгоритми, які можуть працювати з різними типами даних і функціями.
7. Мова Go має широку спільноту розробників, що призводить до активного розвитку екосистеми сторонніх бібліотек і пакетів. Розробники можуть легко використовувати сторонні бібліотеки, що дозволяє розширювати функціональність мови шляхом використання готових рішень.

Мова Go спроектована з орієнтацією на розширюваність та підтримку масштабування проектів. Це дозволяє розробникам легко додавати нову функціональність, розширювати існуючі пакети та створювати великі проекти з ефективним управлінням залежностями.

Я обрав саме мову Go для розробки системного каталога СКБД через її швидкодію, що є важливо для систем, що обробляють великі обсяги даних, через простоту розробки та читабельність коду, через наявність garbage collector і через звичний для мене C-подібний синтаксис.

РОЗДІЛ 3. Розробка системного каталога

3.1 Опис структури

Мій системний каталог має оперувати двома основними сутностями: таблицями (Table) і датасетами (DataSet). Причому не тільки схемами (описами), але й їх наповненнями.

Таблиці, аналогічно до таблиць з реляційного підходу, відіграють роль основної структурної одиниці для збереження даних сутностей або об'єктів, які взаємодіють між собою.

Кожна таблиця має своє унікальне ім’я, список атрибутів (стовпців) та записів (рядків). Кожен стовпець має назву і тип даних, що визначає, який тип інформації може бути збережений у цьому стовпці. Рядки таблиці містять конкретні значення для кожного поля. Кожен рядок є екземпляром цієї сутності. Відповідно, назва таблиці й атрибути складають схему таблиці, а рядки – її наповнення.

Кожен датасет відіграє роль зв’язку між двома таблицями. Датасет зберігає множину пар вигляду (o, m), де o – посилання на запис-власник (OWNER), а m – посилання на запис-член (MEMBER). Всі записи-власники в межах одного датасету повинні бути однотипними, що означає, що вони відповідають одній і тій же таблиці. Так само, всі записи-члени, які належать до цих записів-власників, також повинні бути одного певного типу, який може відрізнятися від типу об'єктів-власників. Тут терміни "запис" і "однотипний" використовуються в реляційному сенсі. Також кожен датасет може містити свою власну таблицю з атрибутів (стовпців) та записів (рядків). Ця таблиця може зберігати додаткову інформацію щодо зв’зку між таблицями, який відображається даним датасетом.

У загальному випадку, кожному запису-власнику може відповідати будь-яка кількість записів-членів, і навпаки. Це відрізняється від пропозиції DBTG CODASYL, де кожному запису-члену мав відповідати лише один запис-власник[3]. Однак, для досягнення основної мети, яка полягає у швидкому виконанні запитів, використовуємо базову структуру зберігання даних –

дворівневе дерево. Так само, як і в CODASYL, датасет зберігається у вигляді множини дворівневих дерев. Кожне дерево має свій корінь, який відповідає певному запису-власнику, і листя, які підпорядковані цьому запису-власнику. Однак, відрізняємось від пропозиції DBTG CODASYL тим, що кілька коренів дерев можуть посилатися на один лист. Таким чином, ця структура даних реалізує зв'язок "багато-до-багатьох" з швидким доступом в одному напрямку: від записів-власників до підлеглих записів-членів. Часто під час проєктування бази даних виникає потреба реалізувати інші типи зв'язків, такі як "один-до-багатьох" зі швидким доступом від "один" до "багато", "багато-до-багатьох" зі швидким доступом в обох напрямках, "один-до-одного", слабкі сутності та інші.

Специфікатор single_owner означає, що не більше одного запису-власника може посилатися на кожен запис-член. Аналогічно з single_member. Таким чином можна визначити всі типи зв’язків: і "один-до-одного", і "один-до-багатьох", і "багато-до-багатьох".

Для створення слабких сутностей визначаємо специфікатор unique та спеціальну додаткову таблицю.

Отже, до схеми датасету належать ім’я датасету (унікальне в межах конкретної бази даних), посилання на таблицю-власник і таблицю-член (які задають типи записів таблиць на які можуть посилатися записи датасету), опис додаткової таблиці (тобто, її атрибути), специфікатори single_owner, single_member, unique та опис спеціальної додаткової таблиці для слабких сутностей.

До наповнення датасету відносяться його рядки, кожен з яких містить посилання на запис-власник з таблиці-власника, посилання на запис-член з таблиці-члена і рядок з додаткової таблиці та рядок з додаткової таблиці для слабких сутностей, якщо вони є.

3.2 Реалізація системного каталога

Почнемо з записів таблиць та датасетів, які складають наповнення відповідних структур. Всі поля матимуть тип string.

```go
type TableEntry struct {
    fields []string
}

type DSEntry struct {
    own           *TableEntry
    mem           *TableEntry
    fields        []string
    unique_fields []string
}
```

Рисунок 5. Структури записів таблиць і записів датасетів

Допоміжна структура Tuple потрібна для опису кожного атрибута. Відповідно, кожен атрибут має ім’я (унікальне в межах таблиці), тип та розмір.
Table також зберігає ім’я (унікальне на рівні БД), масив атрибутів, масив, що вказує які атрибути ключові (мова про первинні ключі, звичайно), розмір структури в байтах при записі на зовнішню пам’ять (тобто, трошки більша за фактичний розмір структури через необхідність в службових полях) і масив посилань на записи таблиці. Як видно, лише останнє поле відповідає за наповнення таблиці.

```go
type Tuple struct {
    Name string
    Type string
    Size int8
}

type Table struct {
    name       string
    attributes []Tuple
    key        []int8
    size       int32
    rows       []*TableEntry
}
```

Рисунок 6. Структури таблиць

DataSet зберігає ім’я, посилання на таблицю-власника і на таблицю-члена, відповідні специфікатори, два масиви атрибутів (тобто, таблиці без імен), розмір в байтах, посилання на обернений датасет і масив посилань на записи таблиці. Лише останнє поле відповідає за наповнення датасету.

```
type DataSet struct {
    name          string
    owner         *Table
    member        *Table
    single_member bool
    single_owmer  bool
    unique        bool
    unique_attr   []Tuple
    attributes    []Tuple
    reverse       *DataSet
    size          int32
    rows          []*DSEntry
}
```

Рисунок 7. Структура датасетів

3.3 Робота з бінарними файлами

Для цього я використовував пакет bytes і encoding/binary[4][5].

Buffer – основний тип даних з bytes, який я застосував, – це змінний буфер, який зберігає послідовність байтів і може використовуватись для читання та запису даних.

Пакет encoding/binary в мові програмування Go (Golang) надає функціонал для серіалізації та десеріалізації даних у байтовому вигляді. Він дозволяє працювати з бінарними форматами даних, такими як цілі числа, рядки, числа з рухомою комою тощо.

Наприклад, для запису опису таблиці у файл я спочатку записую довжину рядка table.name у байтовий буфер buffer, вказавши порядок байтів binary.LittleEndian, де менш значущі байти записуються першими. Однобайтового uint цілком вистачить для збереження довжини імені таблиці. І тільки після цього я записую в буфер саме ім'я таблиці. Звичайно ж, перед серіалізацією перетворюємо масиви рядків на байтові масиви.

```go
func write_table(table *Table) {
    buffer := new(bytes.Buffer)

    err := binary.Write(buffer, binary.LittleEndian, uint8(len([]byte(table.name))))
    if err != nil {
        panic(err)
    }
    _, err = buffer.Write([]byte(table.name))
    if err != nil {
        panic(err)
    }
```

Рисунок 8. Серіалізація полів схеми таблиці

Після цього я записую в буфер кількість атрибутів (також у однобайтовий uint), і для кожного атрибута зберігаю довжину імені атрибута, ім'я атрибута, довжину типу атрибута, тип атрибута і, нарешті, розмір атрибута.

Вкінці записую буфер у файл.

Ось так, наприклад, виглядає запис таблиці з назвою Example, атрибутами attribute1, attribute2, attribute3 типів string, int, bool і розмірами 10, 4, 1 байт відповідно.

Address	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	Dump
00000000	07	45	78	61	6d	70	6c	65	03	0a	61	74	74	72	69	62	.Example..attrib
00000010	75	74	65	31	06	73	74	72	69	6e	67	0a	0a	61	74	74	ute1.string..att
00000020	72	69	62	75	74	65	32	03	69	6e	74	04	0a	61	74	74	ribute2.int..att
00000030	72	69	62	75	74	65	33	04	62	6f	6f	6c	01				ribute3.bool.

Рисунок 9. Записаний опис таблиці

Такий запис дозволяє легко зчитувати навіть великі структури даних.

У випадку запису опису датасета все аналогічно, за винятком того, що замість полів owner та member (які є вказівниками на Table) треба зберігати назви таблиць, на які вони вказують. Для цього створюю мапу з назви Таблиць у вказівник на таблицю. Нові записи у цю мапу додаються при зчитувані таблиць з файлу чи створенні нової таблиці.

```
var TableIdx map[string]*Table
```

Рисунок 10. Мапа

3.4 Основні функції системного каталога:

1. WriteDescrR(ім'я таблиці) – зберегти опис таблиці у файл;
2. WriteContR(ім'я таблиці) – зберегти вміст таблиці у файл;
3. WriteDescrDS(ім'я датасету) – зберегти опис датасету у файл;
4. WriteContDS(ім'я датасету) – зберегти вміст датасету у файл;
5. ReadDescrR(ім'я таблиці) – отримати опис таблиці з файлу;
6. ReadContR(ім'я таблиці) – отримати вміст таблиці з файлу;
7. ReadDescrDS(ім'я датасету) – отримати опис датасету з файлу;
8. ReadContDS(ім'я датасету) – отримати вміст датасету з файлу;
9. SyntReadDescrR(ім'я текстового файлу) – внесення нового опису таблиці в результаті синтаксичного аналізу вмісту відповідного файлу;
10. SyntReadContR(ім'я текстового файлу) – внесення нового вмісту таблиці в результаті синтаксичного аналізу вмісту відповідного файлу;
11. SyntReadDescrDS(ім'я текстового файлу) – внесення нового опису датасета в результаті синтаксичного аналізу вмісту відповідного файлу;
12. SyntReadContDS(ім'я текстового файлу) – внесення нового опису датасета в результаті синтаксичного аналізу вмісту відповідного файлу;
13. FInsertR(вказівник на відповідний опис реляції) – внесення нового опису таблиці в результаті відпрацювання функції, яка створює новий опис;
14. FInsertDS(вказівник на відповідний опис DS) – внесення нового опису датасета в результаті відпрацювання функції, яка створює новий опис;
15. DeleteR(ім'я відповідної таблиці) – вилучити таблицю;
16. DeleteDS(ім'я датасету) – вилучити таблицю.

3.5 Функції, підтримувані системним каталогом

Аргументи багатьох операцій можуть бути не довільні, а лише однотипні датасети. Розглянемо визначення однотипності. Два набори даних є однотипними, якщо однотипні і їхні записи-власники, і записи-члени з обох датасетів. Множина однотипних наборів даних формує тип набору даних.

Наведемо операції вибірки даних, аргументами і результатами яких є датасети:

1. UnionDS(DS1, DS2) –> DS3
2. IntersectDS(DS1, DS2) –> DS3
3. TimesDS(DS1, DS2) –> DS3
4. Project DS(DS1, список атрибутів) –> DS3
5. BFilter(DS1, умова) –>DS2
6. OnlyFilter (DS1, умова) –>DS2
7. EveryFilter (DS1, умова) –>DS2
8. Join (DS1, DS2, умова) –> DS3
9. AddElement (DS, REC1, REC2) –>DS
10. Compose (REL1, REL2, умова) –> DS1
11. REVERSE (DS1) –> DS2

ВИСНОВКИ

У даній роботі були розглянуті основні моделі баз даних. Реляційний підхід є одним із найпоширеніших. Реляційні бази даних забезпечують структурованість, цілісність та стабільність даних.

Проте і реляційний метод має ряд недоліків, які можуть бути вирішальними у певних обставинах. Зокрема, реляційна модель не завжди точно відображає зв’язки між об'єктами реального світу. Наприклад, якщо у реальному світі існує об'єкт з ієрархічною структурою даних, спроба представити його за допомогою кількох реляцій призведе до ускладнення обробки запитів. Також реляційна модель використовує тільки один тип конструкції (таблицю) і для представлення об'єктів, і для представлення зв'язків між ними. Це особливо незручно у зв’язках ‘багато до багатьох‘, в яких доводиться створювати додаткову реляцію, що може сильно заплутати користувача і ускладнити підтримку моделі.

Не мало важливим фактором є використання символьної адресації в реляційних базах даних, яка значно повільніша відносної адресації чи прямих вказівників. Особливо це помітно у базах даних зі складними структурами.

Тому у даній роботі було запропоновано розв'язання цієї проблеми шляхом розробки системного каталогу для СКБД на основі розширеної реляційної алгебри. Я обрав реляції в якості основної структурної одиниці для збереження даних сутностей чи об'єктів. А від підходу, що запропонували DBTG CODASYL я взяв концепцію датасету, що являє собою множину дворівневих дерев, коренем кожного з яких є певний запис-власник, а листями – підпорядковані йому записи-члени. Відмінність в тому, що в моєму датасеті багато коренів можуть посилатися на один лист, що дає можливість реалізовувати зв’язки ‘багато до багатьох’, а не тільки ‘один до багатьох’ чи ‘один до одного’.

Таким чином, мені вдалося позбавитися від деяких негативних аспектів реляційної моделі і реалізувати деякі переваги вже дещо застарілого підходу CODASYL.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

[1] “Object-Oriented Database {Concepts, Examples, Pros and Cons}” by phoenixNAP

[2] “The Go Programming Language and Environment” by Russ Cox, Robert Griesemer, Rob Pike, Ian Lance Taylor, and Ken Thompson in Communications of the ACM

[3] Olle T.W. The CODASYL approach to data base management. NY: John Wiley&Sons, 1978

[4] Документація мови Go, package encoding/binary https://pkg.go.dev/encoding/binary

[5] Документація мови Go, package bytes https://pkg.go.dev/bytes#Buffer