Лабораторна робота №3

По дисципліні “Бази даних”

Засоби оптимізації роботи СУБД PostgreSQL

Студента

Групи КП-02

Литвиненка Артема Сергійовича

Метою роботи є здобуття практичних навичок використання засобів

оптимізації СУБД PostgreSQL.

Завдання роботи полягає у наступному:

1. Перетворити модуль “Модель” з шаблону MVC лабораторної роботи No2

у вигляд об’єктно-реляційної проекції (ORM).

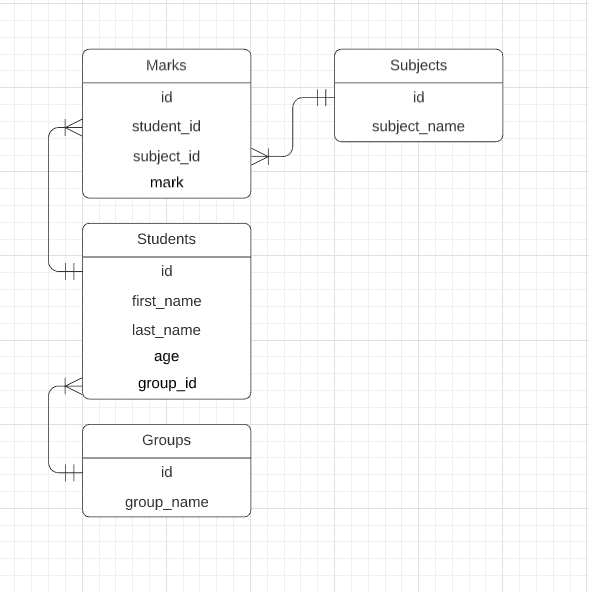
2. Створити та проаналізувати різні типи індексів у PostgreSQL.

3. Розробити тригер бази даних PostgreSQL.

|  |  |  |
| --- | --- | --- |
| № варіанта | Види індексів | Умови для тригера |
| 8 | BTree, Hash | after insert, update |

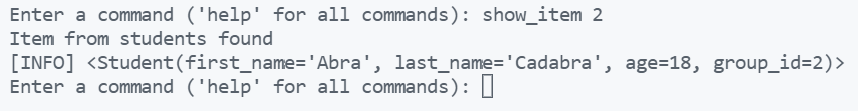
Завдання

1. Перетворити модуль “Модель” з шаблону MVC лабораторної роботи No2 у вигляд об’єктно-реляційної проекції (ORM).

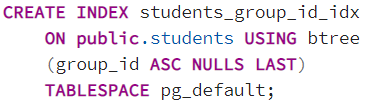
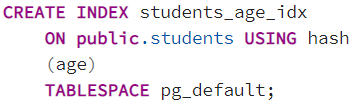


|  |
| --- |
| models.py |
| from sqlalchemy.ext.declarative import declarative\_base  from sqlalchemy import Column, Integer, String, ForeignKey  Base = declarative\_base()  class Student(Base):      \_\_tablename\_\_ = "students"      student\_id = Column(Integer, primary\_key=True)      first\_name = Column(String)      last\_name = Column(String)      age = Column(Integer)      group\_id = Column(String, ForeignKey("groups.group\_id"))      def \_\_repr\_\_(self):          return "<Student(first\_name='{}', last\_name='{}', age={}, group\_id={})>"\              .format(self.first\_name, self.last\_name, self.age, self.group\_id)  class Group(Base):      \_\_tablename\_\_ = "groups"      group\_id = Column(Integer, primary\_key=True)      title = Column(String)      def \_\_repr\_\_(self):          return "<Group(title='{}')>"\              .format(self.title)  class Subject(Base):      \_\_tablename\_\_ = "subjects"      subject\_id = Column(Integer, primary\_key=True)      title = Column(String)      def \_\_repr\_\_(self):          return "<Subject(title='{}')>"\              .format(self.title)  class Mark(Base):      \_\_tablename\_\_ = "marks"      mark\_id = Column(Integer, primary\_key=True)      student\_id = Column(Integer, ForeignKey("students.student\_id"))      subject\_id = Column(Integer, ForeignKey("subjects.subject\_id"))      mark = Column(Integer)      def \_\_repr\_\_(self):          return "<Mark(student\_id={}, subject\_id={}, mark={})>"\              .format(self.student\_id, self.subject\_id, self.mark) |

|  |
| --- |
| controller.py |
| from sqlalchemy.orm import sessionmaker  from sqlalchemy import create\_engine  from ast import literal\_eval  from config import DATABASE\_URI  engine = create\_engine(DATABASE\_URI)  Session = sessionmaker(bind=engine)  class Controller():      def \_\_init\_\_(self, model, view, autocommit=True):          self.model = model          self.view = view          self.session = Session(autocommit=autocommit)      def show\_items(self, bullet\_points=False):          items = self.session.query(self.model).all()          item\_name = self.model.\_\_tablename\_\_          if bullet\_points:              self.view.show\_bullet\_point\_list(item\_name, items)          else:              self.view.show\_number\_point\_list(item\_name, items)      def show\_item(self, item\_id: int):          item\_name = self.model.\_\_tablename\_\_          try:              item = self.session.query(self.model).get(item\_id)              self.view.show\_item(item, item\_name)          except Exception as \_ex:              self.view.display\_missing\_item\_error(item\_name, item\_id, \_ex)      def show\_filtered\_items(self, attrs, bullet\_points=False):          items = self.session.query(self.model).filter\_by(\*\*literal\_eval(attrs))          item\_name = self.model.\_\_tablename\_\_          if bullet\_points:              self.view.show\_bullet\_point\_list(item\_name, items)          else:              self.view.show\_number\_point\_list(item\_name, items)      def insert\_item(self, item\_data):          item\_data = literal\_eval(item\_data)          item = self.model(\*\*item\_data)          item\_name = self.model.\_\_tablename\_\_          try:              self.session.add(item)              self.view.display\_item\_insertion(item\_name)          except Exception as \_ex:              self.view.display\_insert\_item\_error(item, \_ex)      def update\_item(self, item\_data):          item\_data = literal\_eval(item\_data)          item\_id\_dict = {}          id\_column\_name = list(item\_data.keys())[0]          item\_id\_dict[id\_column\_name] = item\_data[id\_column\_name]          item = self.session.query(self.model).filter\_by(\*\*item\_id\_dict).first()          item\_type = self.model.\_\_tablename\_\_          is\_item\_found = bool(item)          try:              if not is\_item\_found:                  raise Exception('Item not found exception')              for key, value in item\_data.items():                  setattr(item, key, value)              self.view.display\_item\_updated(                  item\_type, item\_id\_dict[id\_column\_name])          except Exception as \_ex:              self.view.display\_missing\_item\_error(                  item\_type, item\_id\_dict[id\_column\_name], \_ex)      def delete\_item(self, item\_id: int):          item\_type = self.model.\_\_tablename\_\_          id\_column\_name = self.model.\_\_table\_\_.columns.keys()[0]          item = self.session.query(self.model).filter\_by(              \*\*{id\_column\_name: item\_id}          )          try:              if not item:                  raise Exception('Item not found exception')              item.delete()              self.view.display\_item\_deletion(item\_type, item\_id)          except Exception as \_ex:              self.view.display\_missing\_item\_error(item\_type, item\_id, \_ex)      def \_\_del\_\_(self):          self.session.close() |



1. Створити та проаналізувати різні типи індексів у PostgreSQL.

При порівнянні результатів було виявлено, що фільтрація без індексування виконувалася 0,19мс., у той час коли при використанні індексів ця операція зайняла 0,9мс., що приблизно у два рази швидше.

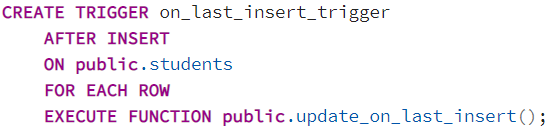
Індекси прискорюють швидкість виконання запитів оскільки для різних типів індексів використовуються різні структури даних. Ту, яку структуру даних треба вибирати залежить вид виду даних. Hash – знаходження даних по ключу, без ітерації; btree – рекурсивний пошук даних, тощо.

1. Розробити тригер бази даних PostgreSQL.

Процедура яку викликає тригер при умові

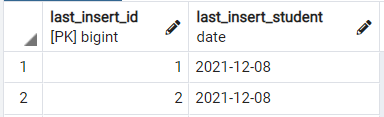


Сам тригер



Приклад





Контрольні запитання

1. Сформулювати призначення та задачі об’єктно-реляційної проекції

(ORM).

**ORM** - технологія, яка зв'язує БД з концепціями мов ООП, створюючи «віртуальну об'єктну базу даних»

Суть проблеми полягає в перетворенні таких об'єктів у форму, в якій вони можуть бути збережені у файлах або базах даних, і які легко можуть бути витягнуті в подальшому, зі збереженням властивостей об'єктів і відношень між ними.

ORM позбавляє програміста від написання великої кількості коду, часто одноманітного і схильного до помилок, тим самим значно підвищуючи швидкість розробки. Крім того, більшість сучасних реалізацій ORM дозволяє програмістові при необхідності жорстко задати код SQL-запитів, який використовуватиметься при тих чи інших діях (збереження в базу даних, завантаження, пошук тощо) з постійним об'єктом.

2. Проаналізувати основні види індексів у PostgreSQL (BTree, BRIN, GIN,

Hash): призначення, сфера застосування, переваги та недоліки.

Індекс btree, він же B-дерево, придатний для даних, які можна відсортувати. Іншими словами, для типу даних повинні бути визначені оператори «більше», «більше або одно», «менше», «менше або одно» та «рівно».

B-дерева мають кілька важливих властивостей: Вони збалансовані, тобто будь-яку листову сторінку відокремлює від кореня те саме число внутрішніх сторінок.

Ідея BRIN не в тому, щоб швидко знайти потрібні рядки, а в тому, щоб уникнути перегляду непотрібних. Це завжди неточний індекс: він взагалі не містить TID-ів табличних рядків. Спрощено кажучи, BRIN добре працює для тих стовпців, значення яких корелюють з їх фізичним розташуванням у таблиці.

GIN розшифровується як Generalized Inverted Index – це так званий зворотний індекс. Він працює з типами даних, значення яких не є атомарними, а складаються з елементів. У цьому індексуються не самі значення, а окремі елементи; кожен елемент посилається ті значення, у яких зустрічається.

Ідея хешування у тому, щоб значенню будь-якого типу даних зіставити деяке невелике число (від 0 до N−1, всього N значень). Таке зіставлення називають хеш-функцією.

Всі перераховані види індексів слугують для оптимізації алгоритмів пошуку, сортування, фільтрації, тощо. Переваги та недоліки кожного описуються в залежності від типу даних. Жоден з них не є гарним або поганим – якщо треба забити гвоздика – молоток; закрутити гайку – викрутка.

1. Пояснити призначення тригерів та функцій у базах даних.

**Тригер** – це процедура особливого типу, яку користувач не викликає явно, а використання якої обумовлено настанням визначеної події (дії) у реляційній БД:

* додаванням INSERT,
* вилученням рядка в заданій таблиці DELETE,
* або зміною даних у певному стовпці заданої таблиці UPDATE.

Тригери застосовуються для забезпечення цілісності даних і реалізації складної бізнес-логіки. Тригер запускається [сервером](https://uk.wikipedia.org/w/index.php?title=%D0%A1%D0%B5%D1%80%D0%B2%D0%B5%D1%80_%D0%B1%D0%B0%D0%B7%D0%B8_%D0%B4%D0%B0%D0%BD%D0%B8%D1%85&action=edit&redlink=1) автоматично при спробі зміни даних у таблиці, з якою він пов'язаний. Всі здійснені ним модифікації даних розглядаються як виконані в транзакції, в якій виконано дію, що викликало спрацьовування тригера. Відповідно, у разі виявлення помилки або порушення цілісності даних може відбутися відкат цієї транзакції.