МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»
ФАКУЛЬТЕТ КОМП’ЮТЕРНИХ НАУК І ТЕХНОЛОГІЙ
(повне найменування факультету)

КАФЕДРА «СИСТЕМНИЙ АНАЛІЗ ТА ОБЧИСЛЮВАЛЬНА МАТЕМАТИКА»

Пояснювальна записка
до дипломного проєкту (роботи)
бакалавра

на тему **Створення бібліотеки для автоматищації взаємодії з реляційними базами даних**

Виконав(ла): студент(ка) 4 курсу, групи КНТ-811
Спеціальності 124 – Системний аналіз
Освітня програма (спеціалізація)
«Інтелектуальні технології та прийняття рішень в складних системах»
Бояр Є.М
Керівник РЯБЕНКО А.Є.________________
Рецензент ПИРОЖОК А.В_____________

2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет ФАКУЛЬТЕТ КОМП'ЮТЕРНИХ НАУК ТА ТЕХНОЛОГІЙ
Кафедра КАФЕДРА «СИСТАНАЛІЗ ТА ОБЧИСЛЮВАЛЬНА МАТЕМАТИКА»
Ступінь вищої освіти Бакалавріат
Спеціальність **124 Системний аналіз**
Освітня програма (спеціалізація) «Інтелектуальні технології та прийняття рішень в складних системах»

ЗАТВЕРДЖУЮ
В. о. завідувача кафедри __ТЕРЕЩЕНКО Е.В.
«_16__»_____червня____2025___року

З А В Д А Н Н Я
НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА(КИ)
Бояра Єгора Миколайовича

1. Тема проєкту (роботи) Створення програмної бібліотеки для автоматизації взаємодії з реляційними базам даних
керівник проєкту (роботи) к.ф.-м.н. доцент Доцент Рябенко Антон Євгенович, затверджені наказом закладу вищої освіти від «16» травня 2025 року № 265
2. Строк подання студентом проєкту (роботи) «16» червня 2025 року
3. Вихідні дані до проєкту (роботи) У процесі виконання роботи використовувалися офіційна документація мови Python, специфікації СУБД SQLite3, приклади реалізації ORM у фреймворках Django та SQLAlchemy, відкриті технічні ресурси щодо побудови систем об'єктно-реляційного відображення, а також типові структури моделей та CRUD-операцій для аналізу архітектурних підходів, реалізації зв'язків між об'єктами, системи міграцій та механізмів завантаження даних; додатково використовувалися результати тестування працездатності розробленої системи з різними конфігураціями моделей і баз даних.
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) Аналіз предметної області та існуючих рішень об'єктно-реляційного відображення для мови Python; огляд переваг, недоліків та архітектурних особливостей популярних ORM-систем; постановка цілей і завдань розробки власної ORM; обґрунтування вибору технологічного стеку; проєктування архітектури системи "NUZP_ORM"; опис реалізації основних компонентів (визначення моделей, CRUD-операції, система міграцій, підтримка зв'язків між моделями, механізми завантаження даних); приклади коду та аналіз згенерованих SQL-запитів; оцінка функціональності, ефективності та обмежень системи; підсумкові висновки та напрямки майбутнього розвитку.
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, кількість слайдів, плакатів)
6. Консультанти розділів проєкту (роботи)

Розділ	ПРІЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		Завдання	Прийняв виконане

		видав	завдання
1	РЯБЕНКО А.Є., к.ф.-м.н., доцент	02.02.2025	26.02.2025
2	РЯБЕНКО А.Є., к.ф.-м.н., доцент	27.02.2025	25.03.2025
3	РЯБЕНКО А.Є., к.ф.-м.н., доцент	26.03.2025	24.04.2025
4	РЯБЕНКО А.Є., к.ф.-м.н., доцент	25.04.2025	28.05.2025
Нормоконтроль	ШИРОКОРАД Д.В., к.ф.-м.н., доцент	29.05.2025	06.06.2025

7. Дата видачі завдання «__02_____» ___лютого________2025___року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назви етапів роботи	Термін виконання етапів роботи
1	Сформулювати мету та основні завдання дипломної роботи	Січень 2025
2	Опрацювати літературу та існуючі дослідження за темою роботи	Січень 2025
3	Розробка програмної реалізації для вирішення задачі	Лютий 2025
4	Розрахунки та аналіз даних	Березень–квітень 2025
5	Оформлення пояснювальної записки	Травень 2025
6	Попередній захист дипломної роботи та отримання рецензій	06.06.2025 – 15.06.2025
7	Захист дипломної роботи	16.06.2025

Студент(ка) ________Бояр_______ _____Єгор БОЯР

Керівник проєкту (роботи) ______________ ___ **Антон РЯБЕНКО**

(підпис) (Ім'я ПРІЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка: 36 с., 3 ч., 0 іл., 1 табл., 0 додат., 10 джерел.

Об’єкт дослідження – процес розробки системи об'єктно-реляційного відображення "NUZP_ORM".

Предмет дослідження – принципи проектування ОРВ, архітектурні патерни, що використовуються при їх створенні, механізми взаємодії Python з базами даних SQLite3, а також методи тестування та забезпечення якості програмних систем подібного класу.

Мета роботи – проектування, розробка та тестування легкої системи об'єктно-реляційного відображення "NUZP_ORM", оптимізованої для роботи з системою управління базами даних SQLite3, що реалізує ключовий функціонал ОРВ.

Методи дослідження: аналіз існуючих рішень у галузі об'єктно-реляційного відображення, об'єктно-орієнтоване проектування та програмування, ітеративна розробка програмного забезпечення, модульне тестування.

Актуальність проблеми зумовлена необхідністю спрощення взаємодії розробників з реляційними базами даних в об'єктно-орієнтованих мовах програмування, таких як Python, та підвищенням продуктивності шляхом абстрагування від написання SQL-запитів вручну, а також освітньою цінністю розробки подібних систем для глибокого розуміння їх функціонування.

У результаті виконання роботи розроблено систему об'єктно-реляційного відображення "NUZP_ORM" для мови Python та СУБД SQLite3. Система надає функціонал для декларативного визначення моделей даних, автоматичного створення та управління схемою бази даних, виконання операцій маніпулювання даними (CRUD) через об'єктно-орієнтований інтерфейс, підтримки основних типів зв'язків між моделями (ForeignKey, OneToOneField, ManyToManyField) та базову систему міграцій для управління змінами в структурі бази даних. Новизна роботи полягає у практичній реалізації легкої ОРВ, адаптованої для SQLite3, що

детально документує процес розробки та може слугувати як освітнім інструментом, так і потенційною основою для невеликих проектів.

Основні висновки: розроблена система "NUZP_ORM" успішно реалізує поставлені завдання, демонструючи ключові можливості об'єктно-реляційного відображення. Вона дозволяє ефективно абстрагуватися від прямої роботи з SQL, підвищуючи швидкість розробки. Визначено обмеження поточної версії, такі як використання стратегії "лінивого" завантаження даних та підтримка виключно SQLite3. Окреслено шляхи подальшого розвитку, включаючи реалізацію "жадібного" завантаження та розширення підтримки інших СУБД.

Результати роботи можуть бути використані в освітньому процесі для вивчення принципів побудови та функціонування систем об'єктно-реляційного відображення, а також як програмна основа для розробки невеликих додатків, що використовують СУБД SQLite3 та потребують простого інструменту для взаємодії з базою даних.

КЛЮЧОВІ СЛОВА: ОБ'ЄКТНО-РЕЛЯЦІЙНЕ ВІДОБРАЖЕННЯ, ОРВ, PYTHON, SQLITE3, БАЗИ ДАНИХ, CRUD, МІГРАЦІЇ ДАНИХ, МОДЕЛІ ДАНИХ, АРХІТЕКТУРА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ.

ЗМІСТ

З А В Д А Н Н Я 2
ВСТУП 8
Розділ 1: Аналітичний огляд літератури та теоретичні основи об'єктно-реляційного відображення (ОРВ) 11
1.1 Поняття, призначення та переваги ОРВ 11
1.2 Огляд існуючих ОРВ-рішень та їх ключові характеристики 12
1.3 Проблематика та виклики при розробці ОРВ 13
Розділ 2: Методологія розробки та архітектура системи "NUZP_ORM" 15
2.1 Цілі та завдання проекту 15
2.2 Обґрунтування вибору технологічного стеку 16
2.3 Архітектура системи "NUZP_ORM" 17
2.4 Процес розробки та тестування 20
2.5 Стрес-тестування та порівняльний аналіз продуктивності "NUZP_ORM" ... 22
Розділ 3: Результати розробки та функціональні можливості "NUZP_ORM" 26
3.1 Визначення моделей та типи даних 26
3.2 Автоматичне створення та управління схемою бази даних 28
3.3 Реалізація операцій маніпулювання даними (CRUD) 29
3.4 Підтримка зв'язків між моделями 32
3.5 Система міграцій бази даних 36
3.6 Оптимізація запитів та управління завантаженням даних 38
3.7.1 Аналіз результатів стрес-тестування 39
3.7.2 Аналіз результатів покриття коду 41
3.7.3 Аналіз залежності часу виконання від складності запиту 43
3.8. Додаткові можливості та утиліти 45
ВИСНОВКИ 47
Перелік посилань 51
ДОДАТКИ 53
Додаток А Файл base.py 53
Додаток Б Файл datatypes.py 80

Додаток В Файл fields.py 84
Додаток Г Файл manager.py 99
Додаток Д Файл query.py 117
Додаток Е Файл run_tests.py 137
Додаток Ж Файл test_stress.py 138
Додаток З Файл testBasic.py 154
Додаток И Файл testConnections.py 180
Додаток І Файл testDatatypes.py 212
Додаток Ї Файл testM2M.py 218
Додаток Й Файл testMigrationHist.py 234
Додаток К Файл testMigrations.py 239

ВСТУП

У сучасному світі розробки програмного забезпечення бази даних відіграють ключову роль у зберіганні, управлінні та отриманні інформації. Більшість додатків, від простих веб-сайтів до складних корпоративних систем, так чи інакше взаємодіють з базами даних. Традиційно така взаємодія відбувається за допомогою мови структурованих запитів (SQL). Однак, незважаючи на потужність та гнучкість SQL, пряма робота з ним у контексті об'єктно-орієнтованих мов програмування, таких як Python, може призводити до низки труднощів. Це включає необхідність написання великої кількості шаблонного коду для перетворення даних між об'єктним представленням у програмі та реляційним представленням у базі даних, а також ризики, пов'язані з SQL-ін'єкціями та складністю підтримки коду при зміні схеми бази даних.

Для подолання цих викликів та спрощення роботи з базами даних були розроблені технології об'єктно-реляційного відображення (Object-Relational Mapping, ORM). ОРВ є програмним шаром, що виступає посередником між об'єктно-орієнтованою моделлю програми та реляційною базою даних. Головна ідея ОРВ полягає у тому, щоб дозволити розробникам працювати з даними, використовуючи звичні об'єктно-орієнтовані концепції (класи, об'єкти, атрибути, методи), в той час як ОРВ автоматично генерує та виконує відповідні SQL-запити "за лаштунками". Це значно підвищує продуктивність розробників, покращує читабельність та підтримуваність коду, а також сприяє кращій структуризації додатків.

Незважаючи на існування багатьох потужних та популярних ОРВ-рішень для Python (наприклад, SQLAlchemy, Django ORM), створення власної ОРВ є цінним освітнім та дослідницьким завданням. Воно дозволяє глибоко зрозуміти внутрішні механізми роботи таких систем, дослідити різні архітектурні підходи та патерни проектування (зокрема, Active Record та Data Mapper), а також вивчити особливості взаємодії Python з базами даних на більш низькому рівні.

Саме ці міркування стали основною мотивацією для розробки проекту "NUZP_ORM".

Метою даної роботи є проектування, розробка та тестування легкої системи об'єктно-реляційного відображення "NUZP_ORM", оптимізованої для роботи з системою управління базами даних SQLite3. Проект спрямований на реалізацію ключового функціоналу, властивого ОРВ, включаючи визначення моделей, автоматичне створення схеми бази даних, виконання CRUD-операцій, підтримку зв'язків між моделями та базову систему міграцій.

Об'єктом дослідження є процес розробки системи об'єктно-реляційного відображення "NUZP_ORM". **Предметом дослідження** є принципи проектування ОРВ, архітектурні патерни, що використовуються при їх створенні, механізми взаємодії Python з базами даних SQLite3, а також методи тестування та забезпечення якості програмних систем подібного класу.

У **Розділі 1** ("Аналіз предметної області та існуючих рішень") буде проведено огляд основних концепцій ОРВ, переваг та недоліків їх використання, а також аналіз існуючих популярних ОРВ-рішень для мови Python. **Розділ 2** ("Методологія розробки та архітектура системи 'NUZP_ORM'") детально описує цілі та завдання проекту, обґрунтовує вибір технологічного стеку, представляє архітектуру розробленої системи та висвітлює процес її розробки та тестування. **Розділ 3** ("Результати розробки та функціональні можливості 'NUZP_ORM'") присвячений демонстрації реалізованого функціоналу, включаючи визначення моделей, управління схемою бази даних, CRUD-операції, підтримку зв'язків та систему міграцій, з наведенням прикладів коду та згенерованих SQL-запитів, та аналізом ефективності програми та її написання. У **Висновках** підсумовуються результати виконаної роботи, оцінюється досягнення поставлених цілей, зазначаються обмеження поточної реалізації та окреслюються можливі напрямки для подальшого розвитку проекту.

Розділ 1 АНАЛІТИЧНИЙ ОГЛЯД ЛІТЕРАТУРИ ТА ТЕОРЕТИЧНІ ОСНОВИ ОБ'ЄКТНО-РЕЛЯЦІЙНОГО ВІДОБРАЖЕННЯ (ОРВ)

1.1. Поняття, призначення та переваги ОРВ

Об'єктно-реляційне відображення (Object-Relational Mapping, ORM) – це технологія програмування, яка дає змогу перетворювати дані між несумісними системами типів в об'єктно-орієнтованих мовах програмування. По суті, ОРВ створює "віртуальну об'єктну базу даних", з якою розробник може взаємодіяти, використовуючи об'єкти та методи замість безпосереднього написання SQL-запитів. Основне призначення ОРВ полягає у спрощенні взаємодії з реляційними базами даних, абстрагуванні від особливостей конкретної системи управління базами даних (СУБД) та інтеграції управління даними в об'єктно-орієнтовану парадигму розробки програмного забезпечення.

Використання ОРВ надає низку суттєвих переваг. По-перше, це **підвищення продуктивності розробників**, оскільки значно зменшується обсяг шаблонного коду, необхідного для виконання операцій створення, читання, оновлення та видалення даних (CRUD-операцій). Розробники можуть зосередитися на бізнес-логіці застосунку, а не на деталях взаємодії з базою даних. По-друге, ОРВ забезпечує **абстрагування від бази даних**, що теоретично дозволяє легше змінювати використовувану СУБД з мінімальними змінами в коді застосунку. По-третє, ОРВ сприяє **підвищенню безпеки**, оскільки багато реалізацій автоматично обробляють параметризацію запитів, що допомагає запобігти атакам типу SQL-ін'єкцій. Нарешті, використання ОРВ **покращує супроводжуваність коду**: логіка доступу до даних стає більш читабельною, структурованою та відповідає об'єктно-орієнтованим принципам. Головне завдання ОРВ – автоматизувати взаємодію з базами даних, зробити її більш гнучкою, безпечною та ефективною.

1.2. Огляд існуючих ОРВ-рішень та їх ключові характеристики

В екосистемі Python існує декілька поширених та потужних ОРВ-рішень, кожне з яких має свої особливості. Серед найвідоміших можна виділити SQLAlchemy, Django ORM та Peewee.

SQLAlchemy важається однією з найпотужніших та найгнучкіших ОРВ-бібліотек для Python. Вона надає два рівні абстракції: SQLAlchemy Core, що є інструментарієм для роботи з SQL-виразами, та SQLAlchemy ORM, яка реалізує повноцінне об'єктно-реляційне відображення за патерном Data Mapper. Це дозволяє розробникам детально контролювати SQL-запити, що генеруються, та водночас користуватися перевагами високорівневої об'єктної абстракції.

Django ORM є невід'ємною частиною популярного вебфреймворку Django. Вона реалізує патерн Active Record, де клас моделі представляє таблицю в базі даних, а екземпляр класу – рядок у цій таблиці. Django ORM відома своєю простотою використання, тісною інтеграцією з іншими компонентами фреймворку та підходом "все включено" (batteries-included), що значно прискорює розробку вебзастосунків.

Peewee – це невелика, виразна та проста у вивченні ОРВ. Як і Django ORM, вона часто використовує патерн Active Record. Peewee підтримує SQLite, MySQL та PostgreSQL і є гарним вибором для невеликих та середніх проєктів, де не потрібна вся міць SQLAlchemy або інтеграція з Django.

Більшість сучасних ОРВ надають схожий набір базових функцій, таких як визначення моделей даних, автоматична генерація схеми бази даних, API для виконання запитів, управління зв'язками між моделями (один-до-одного, один-до-багатьох, багато-до-багатьох), механізми транзакцій та системи міграцій для управління змінами в схемі бази даних.

Архітектурні патерни, що лежать в основі ОРВ, переважно поділяються на **Active Record** та **Data Mapper**. Active Record (як у Django ORM та Peewee) передбачає, що об'єкт моделі "знає", як зберегти себе в базу даних та завантажити з неї. Data Mapper (як у SQLAlchemy ORM) відокремлює об'єкти в пам'яті від

бази даних за допомогою спеціального шару (мапера), що відповідає за перенесення даних. Це забезпечує більшу незалежність моделей від логіки зберігання даних. Розуміння цих підходів та можливостей існуючих рішень є важливим для позиціонування та оцінки власної розробки ОРВ.

1.3. Проблематика та виклики при розробці ОРВ

Розробка об'єктно-реляційного відображення пов'язана з низкою концептуальних та технічних викликів. Однією з фундаментальних проблем є так звана **"невідповідність об'єктно-реляційного імпедансу"** (object-relational impedance mismatch). Вона виникає через розбіжності між об'єктно-орієнтованою та реляційною парадигмами, зокрема у питаннях гранулярності даних (об'єкти можуть мати складну структуру, тоді як рядки таблиць є "пласкими"), ідентичності (ідентичність об'єкта в пам'яті проти первинних ключів у базі даних), способів представлення зв'язків (навігація за посиланнями між об'єктами проти операцій JOIN у SQL) та відмінностей у системах типів даних.

Іншим важливим аспектом є **продуктивність**. Абстракція, яку надає ОРВ, може призводити до генерації неоптимальних SQL-запитів. Класичною проблемою є **"проблема N+1 запиту"**, коли для завантаження колекції об'єктів та пов'язаних з ними даних виконується один запит для основних об'єктів, а потім N додаткових запитів для завантаження пов'язаних даних для кожного з N об'єктів. Ефективне вирішення цієї проблеми вимагає реалізації механізмів "жадібного" (eager) та "лінивого" (lazy) завантаження даних.

Складність генерації запитів також є значним викликом. Перетворення високорівневих об'єктно-орієнтованих запитів (наприклад, методів фільтрації або сортування) на ефективний та коректний SQL-код, особливо з урахуванням різноманітних діалектів SQL для різних СУБД, є нетривіальним завданням.

Управління міграціями схеми бази даних – ще одна критична область. У процесі розробки моделі даних часто змінюються: додаються нові поля, змінюються типи існуючих, видаляються застарілі. ОРВ повинна надавати надійний механізм для відстеження цих змін та автоматичного застосування їх до схеми бази даних без втрати даних.

Окрім перерахованого, розробники ОРВ стикаються з необхідністю реалізації ефективних стратегій кешування, коректного управління транзакціями для забезпечення цілісності даних, а також із тим, що складні ОРВ можуть мати досить високий поріг входження для нових користувачів. Врахування цих викликів є ключовим для створення життєздатної та корисної системи об'єктно-реляційного відображення.

Розділ 2 МЕТОДОЛОГІЯ РОЗРОБКИ ТА АРХІТЕКТУРА СИСТЕМИ "NUZP_ORM"

2.1. Цілі та завдання проекту

Розробка системи об'єктно-реляційного відображення "NUZP_ORM" мала на меті досягнення кількох ключових цілей. Перш за все, проект слугував освітньою платформою для поглибленого вивчення принципів роботи ОРВ, архітектурних патернів, що лежать в їх основі (зокрема, Active Record та елементів Data Mapper), та складнощів, пов'язаних із взаємодією між об'єктно-орієнтованою парадигмою та реляційними базами даних.

Другою важливою ціллю було створення легкої та простої у використанні ОРВ, спеціально оптимізованої для роботи з СУБД SQLite3. Акцент робився на реалізації основного функціоналу, необхідного для більшості невеликих та середніх проектів, без надлишкової складності, притаманної деяким більш потужним ОРВ-рішенням.

Основні завдання проекту включали:

– дослідження існуючих ОРВ та їх архітектурних рішень.

– Проектування власної архітектури ОРВ з урахуванням обраного стеку технологій.

– Реалізація механізму визначення моделей даних у Python.

– Розробка системи автоматичної генерації схеми бази даних на основі визначених моделей.

– Імплементація функціоналу для виконання CRUD-операцій (Create, Read, Update, Delete).

– Забезпечення підтримки основних типів зв'язків між моделями (ForeignKey, OneToOneField, ManyToManyField).

– Створення базової системи міграцій для управління змінами в структурі бази даних.

– Розробка системи тестування для забезпечення надійності та коректності роботи ОРВ.

Запланований обсяг функціональних можливостей включав визначення моделей, автоматичне створення таблиць, виконання базових CRUD-операцій, підтримку зв'язків між таблицями з каскадним видаленням для ForeignKey, а також базовий механізм міграцій та управління M2M зв'язками через спеціалізований менеджер.

2.2. Обґрунтування вибору технологічного стеку

Для реалізації проекту "NUZP_ORM" було обрано мову програмування Python та систему управління базами даних SQLite3. Цей вибір зумовлений кількома факторами.

Python було обрано завдяки його високорівневій природі, динамічній типізації та багатим можливостям метапрограмування (зокрема, використання метакласів та дескрипторів), що є надзвичайно корисними при розробці ОРВ. Простота синтаксису Python та велика кількість стандартних бібліотек дозволяють зосередитися на логіці самої ОРВ, а не на низькорівневих деталях реалізації. Крім того, Python є популярною мовою для веб-розробки та аналізу даних, де ОРВ знаходять широке застосування.

SQLite3 було обрано як цільову СУБД через її простоту, відсутність необхідності у налаштуванні окремого сервера баз даних та зберігання всієї бази даних в одному файлі. Це значно спрощує розгортання та використання ОРВ, особливо для невеликих проектів, локальних застосунків та освітніх цілей. Легковажність SQLite3 також дозволила зосередитися на ключових аспектах ОРВ, не ускладнюючи систему підтримкою безлічі діалектів SQL для різних СУБД на початковому етапі розробки. Вбудована підтримка SQLite3 у стандартній бібліотеці Python також є перевагою.

2.3. Архітектура системи "NUZP_ORM"

Архітектура "NUZP_ORM" розроблена з метою забезпечення модульності та розширюваності. Вона складається з кількох ключових компонентів, що взаємодіють між собою:

1. **Шар визначення моделей (Model Definition Layer):**

a. Користувачі визначають свої моделі даних шляхом успадкування від базового класу ORM.base.BaseModel.

б. Ключову роль в автоматизації цього процесу відіграють **метакласи** (в "NUZP_ORM" це, ймовірно, ModelMeta, вказаний для BaseModel). Коли Python обробляє визначення класу моделі (наприклад, class User(BaseModel): ...), саме метаклас перехоплює процес створення цього класу.

i. **Як саме відбувається автоматизація:**

1. **Аналіз атрибутів:** Метаклас отримує доступ до всіх атрибутів, визначених всередині класу моделі (наприклад, name = CharField(...), age = IntegerField(...)) *ще до того, як сам клас моделі повністю сформований*.

2. **Ідентифікація полів ОРВ:** Він переглядає ці атрибути та ідентифікує ті, що є екземплярами спеціальних класів полів ОРВ (CharField, IntegerField, ForeignKey тощо).

3. **Збір метаданих:** Для кожного ідентифікованого поля метаклас збирає всю необхідну інформацію: ім'я поля, його тип, передані параметри (наприклад, max_length, unique, default). Ця інформація зберігається у внутрішній структурі (часто це словник або спеціальний об'єкт, що прикріплюється до класу моделі, наприклад, як атрибут _fields).

4. **Автоматичне додавання службових елементів:** Метаклас може автоматично додавати до моделі службові поля (наприклад, первинний ключ id, якщо він не визначений явно) та генерувати ім'я таблиці бази даних на основі імені класу моделі (наприклад, клас User відображається в таблицю user).

5. **Реєстрація моделі:** Новостворений клас моделі може бути зареєстрований у глобальному реєстрі ОРВ, що дозволяє іншим

компонентам системи (наприклад, системі міграцій) знати про всі доступні моделі.

ii. Таким чином, метаклас звільняє користувача ОРВ від необхідності вручну реєструвати кожне поле або модель. Користувач просто декларативно описує структуру моделі, а метаклас "за лаштунками" виконує всю роботу зі збору метаданих та підготовки класу до взаємодії з ОРВ.

в. Автоматично додається поле id як первинний ключ для кожної моделі, якщо воно не визначене користувачем.

2. **Типи полів та валідація (Field Types and Validation):**

а. Система підтримує різноманітні типи полів, такі як CharField, IntegerField, DateTimeField (визначені в ORM/datatypes.py), а також спеціалізовані поля для зв'язків: ForeignKey, OneToOneField, ManyToManyField (визначені в ORM/fields.py).

б. Для полів передбачені опції, такі як null (чи може поле бути порожнім), unique (чи повинні значення бути унікальними), default (значення за замовчуванням). Базова валідація може виконуватися на основі цих опцій.

в. **Дескриптори** Python використовуються для реалізації логіки полів. Коли користувач звертається до атрибута моделі, що є полем ОРВ (наприклад, my_instance.name або my_instance.related_object), саме дескриптор контролює, що відбувається. Це дозволяє, наприклад, реалізувати "ліниве" завантаження для полів ForeignKey (дані з бази завантажуються лише при першому зверненні) або повертати спеціальний менеджер для полів ManyToManyField (як це видно з ORM/fields.py, де метод __get__ для ManyToManyField повертає ManyToManyRelatedManager).

3. **Механізм запитів (Query Engine):**

а. Взаємодія з базою даних для вибірки даних здійснюється через об'єкт-менеджер, доступний для кожної моделі (наприклад, Model.objects).

б. Результатом запитів є об'єкти QuerySet (клас визначено в ORM/query.py), які підтримують ланцюжкові виклики методів для побудови складних запитів (наприклад, filter(), order_by(), limit(), offset()).

в. QuerySet є ітерабельним та підтримує отримання всіх результатів (all()) або одного конкретного запису (get()).

г. Підтримуються різні типи пошукових виразів для фільтрації (__exact, __like, __gt, __gte, __lt, __lte, __in, __neq).

4. **Управління зв'язками (Relationship Management):**

а. ForeignKey та OneToOneField реалізують зв'язки "один-до-багатьох" та "один-до-одного" відповідно. OneToOneField успадковується від ForeignKey з примусовим обмеженням unique=True.

б. ManyToManyField реалізує зв'язки "багато-до-багатьох" через автоматично генеровану або користувацьку проміжну таблицю. Доступ до пов'язаних об'єктів та управління зв'язками (додавання, видалення, очищення) здійснюється через спеціальний менеджер, який повертається при доступі до M2M-поля на екземплярі моделі (наприклад, instance.m2m_field.add(related_instance)), що, як зазначено вище, реалізується за допомогою дескрипторів.

5. **Менеджер міграцій (Migration Manager):**

а. Реалізовано базову систему міграцій, керовану скриптом ORM/manager.py.

б. Підтримуються команди для генерації файлів міграцій на основі поточного стану моделей (generate --app <app_folder>), застосування міграцій до бази даних (migrate) та перегляду статусу міграцій (showmigrations).

в. Застосовані міграції відстежуються у спеціальній таблиці orm_migrations в базі даних.

Ключові можливості Python, що активно використовуються в архітектурі, включають **метакласи** для автоматизації визначення моделей (як детально описано вище). **Дескриптори** відіграють важливу роль у реалізації поведінки полів та управлінні доступом до атрибутів екземплярів моделей: вони

дозволяють перехоплювати операції читання (__get__) та запису (__set__) значень полів. Це використовується для таких завдань, як: * Реалізація "лінивого" завантаження (lazy loading) для полів зв'язків типу ForeignKey, коли пов'язаний об'єкт завантажується з бази даних лише при першому зверненні до поля. * Повернення спеціалізованих об'єктів-менеджерів для полів типу ManyToManyField при доступі до них на екземплярі моделі, що дозволяє далі використовувати методи add(), remove() тощо. * Виконання валідації даних або перетворення типів під час присвоєння значення полю. * Забезпечення того, що значення поля зберігається та витягується з правильного місця всередині екземпляра моделі (наприклад, зі словника атрибутів). Окрім цього, стандартні структури даних та об'єктно-орієнтовані принципи є основою для побудови модульної системи.

2.4. Процес розробки та тестування

Процес розробки "NUZP_ORM" можна умовно поділити на кілька етапів:

1. Дослідження та проектування: Аналіз існуючих ОРВ, визначення ключових вимог та проектування загальної архітектури системи.
2. Реалізація ядра: Створення базових класів для моделей, полів, механізму запитів та генерації SQL.
3. Розробка підтримки зв'язків: Імплементація ForeignKey, OneToOneField та ManyToManyField.
4. Створення системи міграцій: Розробка інструментів для генерації та застосування міграцій.
5. Тестування та налагодження: Написання юніт-тестів для всіх компонентів системи, виправлення помилок та рефакторинг.
6. Документування: Створення README.md та коментарів у коді.

Тестування відігравало важливу роль на всіх етапах розробки. Для забезпечення якості та надійності коду використовувався вбудований у Python модуль unittest. Основний фокус тестів був спрямований на:

– коректність визначення моделей та створення таблиць у базі даних.

– Правильність генерації SQL-запитів для CRUD-операцій.

– Функціонування фільтрації, сортування та обмеження вибірки.

– Коректну роботу всіх типів зв'язків, включаючи додавання, видалення та вибірку пов'язаних об'єктів.

– Роботу системи міграцій (генерація та застосування).

Для оцінки повноти тестового покриття використовувалася утиліта coverage. Передбачені команди для запуску тестів та генерації звітів про покриття (coverage run -m unittest discover ..., coverage report -m, coverage html). Також згадується інструмент interrogate для перевірки документованості коду. Такий підхід до тестування дозволив виявити та виправити значну кількість помилок на ранніх стадіях та забезпечити стабільну роботу основних функцій "NUZP_ORM".

2.5. Стрес-тестування та порівняльний аналіз продуктивності "NUZP_ORM"

Для оцінки продуктивності розробленої системи "NUZP_ORM" під навантаженням та для порівняння її ефективності з еталонним рішенням було проведено серію стрес-тестів. В якості еталонної системи було обрано Django ORM, як одну з найпопулярніших та оптимізованих ОРВ для Python.

Методика тестування: Тестування проводилося на однакових наборах даних, згенерованих за допомогою бібліотеки Faker. Були створені дві моделі: StressAuthor (Автори) та StressPost (Пости), зі зв'язком "один-до-багатьох" (один автор може мати багато постів). Тести включали операції масової вставки, оновлення, видалення записів, а також виконання простих та складніших запитів на вибірку даних. Кількість записів для тестів складала 1000 авторів та по 5 постів на кожного автора (загалом 5000 постів). Вимірювався час виконання кожної операції.

Результати тестування та порівняльний аналіз:

На основі проведених тестів (детальні результати часу виконання наведені у Таблиці 2.1) можна зробити наступні спостереження:

Таблиця 2.1 – Порівняння часу виконання операцій "NUZP_ORM" та Django ORM (в секундах)

Операція	**Кількість записів**	**NUZP_ORM (с)**	**Django ORM (с)**
Масова Вставка			
Вставка Авторів	1000	~0.0191	~0.0180
Вставка Постів	5000	~0.1032	~0.1377
Масове Оновлення			
Оновлення Авторів (зміна імені)	500	~8.0685	~0.0536
Оновлення Постів (зміна лічильника переглядів)	2500	~42.2089	~0.2733
Вибірка Даних			
Простий запит (один пост по автору)	1 (з 5000)	~0.0537	~0.0028
Складний запит (пости авторів на "A")	Залежить від даних	~0.0768	~0.0061
Фільтрація (пости з >5000 переглядів)	Залежить від даних	~0.0646	~0.0001
Масове Видалення			
Видалення Авторів (та пов'язаних постів)	500 (та їх пости)	~0.0000	~0.0000

Примітка до таблиці:

– *N/A (Not Available/Не зафіксовано):* Точний час для масового видалення в "NUZP_ORM" у наданому виводі не був окремо зафіксований для прямого порівняння з оптимізованим видаленням Django. Тест для "NUZP_ORM" передбачав ітеративне видалення.

– *Django ORM Масове Видалення:* Django ORM використовує оптимізовані запити DELETE та каскадне видалення на рівні бази даних, тому час виконання для цієї операції дуже малий (у виводі тестів ~0.0000 с).

1. Масова вставка (Bulk Insert):

a. У операціях масової вставки "NUZP_ORM" показала результати, цілком співставні з Django ORM (Таблиця 2.1). Це свідчить про ефективну реалізацію методу insert_entries в "NUZP_ORM" для сценаріїв початкового наповнення бази даних.

2. Масове оновлення (Bulk Update):

a. У операціях масового оновлення Django ORM демонструє значно вищу продуктивність (Таблиця 2.1). Це пояснюється тим, що Django використовує спеціалізований метод bulk_update, який оптимізує взаємодію з базою даних. В "NUZP_ORM" оновлення в поточній реалізації тесту відбувалося шляхом ітерації по записах та виконання індивідуального запиту replace_entries для кожного.

3. Вибірка даних (Querying):

a. Django ORM показує значно кращу продуктивність при виконанні запитів, особливо складних (Таблиця 2.1). Це пов'язано з тим, що Django ORM генерує більш оптимізовані SQL-запити та виконує максимальну кількість операцій фільтрації на стороні бази даних, тоді як у тестовому сценарії для "NUZP_ORM" деякі операції фільтрації виконувалися на стороні Python.

4. Масове видалення (Bulk Delete):

a. Django ORM використовує оптимізовані запити DELETE та каскадне видалення, що є значно ефективнішим, ніж ітеративне видалення, реалізоване в тестовому сценарії для "NUZP_ORM".

Висновки зі стрес-тестування: Проведене стрес-тестування показало, що "NUZP_ORM" демонструє конкурентоспроможну продуктивність при операціях масової вставки даних. Однак, у сценаріях масового оновлення та виконання складних запитів на вибірку даних, "NUZP_ORM" суттєво поступається Django ORM. Основною причиною є відсутність в "NUZP_ORM" спеціалізованих методів для масових операцій (як bulk_update в Django) та виконання частини логіки фільтрації на стороні Python замість делегування її базі даних.

Ці результати підкреслюють важливість оптимізації взаємодії з базою даних для операцій, що виконуються над великими обсягами даних. Для "NUZP_ORM" потенційними напрямками покращення продуктивності могли б стати:

- реалізація методів для масового оновлення та видалення записів, що генерують один або мінімальну кількість SQL-запитів.
- Вдосконалення механізму генерації SQL-запитів для filter(), щоб забезпечити виконання максимальної кількості умов фільтрації на стороні СУБД.

Незважаючи на виявлені аспекти продуктивності, розроблена "NUZP_ORM" виконує свої основні функції та слугує цінним інструментом для розуміння принципів роботи об'єктно-реляційних відображувачів.

Розділ 3 РЕЗУЛЬТАТИ РОЗРОБКИ ТА ФУНКЦІОНАЛЬНІ МОЖЛИВОСТІ "NUZP_ORM"

Цей розділ присвячений детальному опису функціональних можливостей реалізованої системи об'єктно-реляційного відображення "NUZP_ORM", включаючи визначення моделей, управління схемою бази даних, виконання операцій маніпулювання даними (CRUD), підтримку зв'язків між моделями, систему міграцій та інші аспекти.

3.1 Визначення моделей та типи даних

Основою будь-якої ОРВ є можливість визначати моделі даних, які відображатимуться на таблиці в реляційній базі даних. У "NUZP_ORM" цей процес є інтуїтивно зрозумілим.

Спосіб визначення моделей: Користувачі визначають свої моделі шляхом успадкування від базового класу ORM.base.BaseModel. Кожен клас, що успадковується від BaseModel, автоматично розглядається системою як модель, для якої буде створена відповідна таблиця в базі даних.

Підтримувані типи полів: "NUZP_ORM" підтримує набір базових типів полів, які визначені в модулі ORM/datatypes.py. До них належать:

– CharField: Для зберігання текстових рядків. Має обов'язковий параметр max_length та опціональні null, unique, default.

– IntegerField: Для зберігання цілих чисел. Має опціональні параметри null, unique, default.

– DateTimeField: Для зберігання дати та часу. Має опціональні параметри null, unique, default, а також auto_now (автоматично встановлює поточний час при кожному оновленні запису) та auto_now_add (автоматично встановлює поточний час при створенні запису).

– Також існують спеціальні типи полів для зв'язків: ForeignKey, OneToOneField, ManyToManyField, які детальніше розглянуті в підрозділі 3.4.

Опції та обмеження полів: Для більшості типів полів доступні такі опції:

– null (булеве значення): Якщо True, поле може мати значення NULL в базі даних. За замовчуванням False.

– unique (булеве значення): Якщо True, значення в цьому полі повинні бути унікальними в межах таблиці. За замовчуванням False.

– default: Значення за замовчуванням для поля, якщо воно не вказане при створенні запису.

Автоматичний первинний ключ: Кожна модель в "NUZP_ORM" автоматично отримує поле id типу IntegerField, яке слугує первинним ключем (PRIMARY KEY AUTOINCREMENT в SQLite). Це поле додається метакласом моделі, якщо воно не визначене користувачем явно.

Приклад визначення моделі:

```
from ORM.base import BaseModel
from ORM.datatypes import CharField, IntegerField,
DateTimeField

class Author(BaseModel):
    name = CharField(max_length=100, unique=True)
    email = CharField(max_length=150, null=True)

class Book(BaseModel):
    title = CharField(max_length=200)
    publication_year = IntegerField(null=True)
    created_at = DateTimeField(auto_now_add=True)
    # Поле для зв'язку буде додано в підрозділі 3.4
```

3.2. Автоматичне створення та управління схемою бази даних

Після визначення моделей "NUZP_ORM" бере на себе завдання трансляції цих визначень у структуру таблиць бази даних SQLite.

Трансляція моделей у таблиці:

– ім'я таблиці в базі даних автоматично генерується на основі імені класу моделі шляхом переведення його в нижній регістр (наприклад, клас Book стане таблицею book).

– Кожне поле, визначене в класі моделі, перетворюється на відповідний стовпець у таблиці. Тип даних стовпця в SQLite визначається на основі типу поля ОРВ (наприклад, CharField стає TEXT, IntegerField стає INTEGER).

– Обмеження полів (NOT NULL, UNIQUE, DEFAULT) також транслюються у відповідні обмеження на рівні SQL.

– Первинний ключ id автоматично створюється як INTEGER PRIMARY KEY AUTOINCREMENT.

Приклад згенерованого SQL для створення таблиць (для моделей з п. 3.1):

Для моделі Author:

```
CREATE TABLE IF NOT EXISTS author (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL UNIQUE,
    email TEXT
);
```

Для моделі Book:

```
CREATE TABLE IF NOT EXISTS book (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    title TEXT NOT NULL,
    publication_year INTEGER,
    created_at DATETIME NOT NULL
);
```

Ці SQL-запити генеруються та виконуються, наприклад, під час процесу міграцій (див. підрозділ 3.5).

3.3. Реалізація операцій маніпулювання даними (CRUD)

"NUZP_ORM" надає API для виконання основних операцій створення (Create), читання (Read), оновлення (Update) та видалення (Delete) записів.

Створення записів (Create): Нові записи додаються за допомогою методу Model.insert_entries([...]). Цей метод приймає список словників або екземплярів моделі.

– Приклад Python:# Створення через словники

```
author_data = [
    {"name": "George Orwell", "email":
"gorwell@example.com"},
    {"name": "J.R.R. Tolkien"}
]
Author.insert_entries(author_data)
```

Створення через екземпляри

```
new_book = Book(title="The Hobbit",
publication_year=1937)
Book.insert_entries([new_book])
```

– Відповідний SQL (концептуально для першого автора):

```
INSERT INTO author (name, email) VALUES ('George
Orwell', 'gorwell@example.com');
```

Читання записів (Read): Доступ до даних здійснюється через менеджер Model.objects.

- Отримання всіх записів: `Model.objects.all()`
  - Приклад Python:

```
all_authors = Author.objects.all()
for author in all_authors:
    print(author.name)
```

  - Відповідний SQL:

```
SELECT id, name, email FROM author;
```

- Отримання одного запису: `Model.objects.get(**kwargs)`
  - Приклад Python:

```
tolkien = Author.objects.get(name="J.R.R. Tolkien")
if tolkien:
    print(tolkien.email)
```

  - Відповідний SQL:

```
SELECT id, name, email FROM author WHERE name = 'J.R.R. Tolkien' LIMIT 1;
```

- Фільтрація записів: `Model.objects.filter(**kwargs)` Підтримуються різні типи пошукових виразів (lookups): `__exact`, `__like`, `__gt`, `__gte`, `__lt`, `__lte`, `__in`, `__neq`.
  - Приклад Python:

```
recent_books = Book.objects.filter(publication_year__gt=1950)
orwell_books = Book.objects.filter(title__like="%Animal Farm%")
```

```
# Примітка: для __like потрібна відповідна реалізація в ORM/query.py
```

– Відповідний SQL (для publication_year__gt):

```
SELECT id, title, publication_year, created_at FROM book WHERE publication_year > 1950;
```

– Сортування, обмеження, зсув: `order_by(), limit(), offset()`

– Приклад Python:

```
# Отримати 5 найновіших книг, пропустивши перші 2
latest_books = Book.objects.order_by("publication_year").offset(2).limit(5)
```

– Відповідний SQL:

```
SELECT id, title, publication_year, created_at FROM book ORDER BY publication_year DESC LIMIT 5 OFFSET 2;
```

– Ітерабельність та зрізи QuerySet: Об'єкти QuerySet, що повертаються методами all() та filter(), є ітерабельними та підтримують зрізи, що дозволяє отримувати дані частинами.

Оновлення записів (Update): Оновлення записів виконується за допомогою методу Model.replace_entries(conditions, new_values).

– Приклад Python:

```
Author.replace_entries(
conditions={"name": "George Orwell"}, new_values={"email": "eric.blair@example.com"}
```

```
)
```

– Відповідний SQL:

```
UPDATE author SET email = 'eric.blair@example.com' WHERE name = 'George Orwell';
```

Видалення записів (Delete): Видалення записів виконується за допомогою методу `Model.delete_entries(conditions).`

– Приклад Python:

```
Book.delete_entries(conditions={"title": "The Hobbit"})
```

– Відповідний SQL:

```
DELETE FROM book WHERE title = 'The Hobbit';
```

3.4. Підтримка зв'язків між моделями

"NUZP_ORM" підтримує основні типи зв'язків: ForeignKey, OneToOneField та ManyToManyField.

Визначення зв'язків:

– ForeignKey(to_model, on_delete="CASCADE"): Реалізує зв'язок "багато-до-одного". to_model – це клас моделі, на яку вказує зв'язок. on_delete="CASCADE" (як зазначено в README.md для ForeignKey) означає, що при видаленні запису з "один"-сторони, пов'язані записи з "багато"-сторони також будуть видалені.

– OneToOneField(to_model, on_delete="CASCADE"): Реалізує зв'язок "один-до-одного". Фактично, це ForeignKey з обмеженням unique=True.

– ManyToManyField(to_model, through=None): Реалізує зв'язок "багато-до-багатьох". to_model – це клас пов'язаної моделі. Якщо through не вказано, ОРВ автоматично створює проміжну таблицю.

Зберігання зв'язків у базі даних:

– для ForeignKey та OneToOneField у таблиці моделі, що визначає зв'язок, створюється стовпець із суфіксом _id (наприклад, якщо поле називається author типу ForeignKey(Author), то стовпець буде author_id), який зберігає ID пов'язаного запису.

– Для ManyToManyField створюється окрема проміжна таблиця (junction table). Ім'я таблиці зазвичай генерується на основі імен пов'язаних моделей (наприклад, book_author). Ця таблиця має два стовпці, що є зовнішніми ключами до відповідних моделей.

Приклад визначення моделей зі зв'язками:

```
from ORM.base import BaseModel
from ORM.datatypes import CharField, IntegerField
from ORM.fields import ForeignKey, ManyToManyField

class Author(BaseModel):
    name = CharField(max_length=100, unique=True)

class Book(BaseModel):
    title = CharField(max_length=200)
    # Зв'язок багато-до-одного: одна книга має одного автора (спрощення)
    # Якщо одна книга може мати багато авторів, потрібен ManyToManyField тут,
    # або ForeignKey в моделі Author, якщо автор може написати багато книг.
    # Для прикладу, припустимо, що одна книга має одного головного автора (ForeignKey)
    # і може бути у співавторстві (ManyToManyField).
```

```
main_author = ForeignKey(Author, on_delete="CASCADE")
# Для співавторства:
co_authors = ManyToManyField(Author)
```

Згенерований SQL для зв'язків (доповнення до таблиць з п. 3.2):

Таблиця book буде модифікована для main_author:

```
CREATE TABLE IF NOT EXISTS book (
id INTEGER PRIMARY KEY AUTOINCREMENT,
title TEXT NOT NULL,
main_author_id INTEGER, -- Додано для ForeignKey
FOREIGN KEY (main_author_id) REFERENCES author(id) ON
DELETE CASCADE
);
```

Автоматично створена проміжна таблиця для book.co_authors та Author (наприклад, book_co_authors_author або подібне, залежно від логіки генерації імен в ORM/fields.py):

```
CREATE TABLE IF NOT EXISTS book_author ( -- Ім'я
може відрізнятися
id INTEGER PRIMARY KEY AUTOINCREMENT,
book_id INTEGER,
author_id INTEGER,
FOREIGN KEY (book_id) REFERENCES book(id) ON DELETE
CASCADE,
FOREIGN KEY (author_id) REFERENCES author(id) ON
DELETE CASCADE,
UNIQUE (book_id, author_id) -- Зазвичай додається для
уникнення дублікатів
);
```

Використання зв'язків:

– доступ до пов'язаних об'єктів ForeignKey:

```
book = Book.objects.get(title="1984")
if book and book.main_author: # Припускаючи, що main_author завантажується
    print(f"Головний автор книги '{book.title}': {book.main_author.name}")
```

Завантаження book.main_author зазвичай є "лінивим" (lazy), тобто запит до бази даних для отримання автора виконується лише при першому зверненні до book.main_author.

– Управління ManyToManyField: Доступ до M2M-поля на екземплярі моделі повертає спеціальний менеджер (як зазначено в ORM/fields.py, це ManyToManyRelatedManager), який надає методи для управління зв'язками: add(), remove(), clear(), set(), all(), filter(), get().

```
# Припустимо, у нас є екземпляри:
book_1984 = Book.objects.get(title="1984")
author_orwell = Author.objects.get(name="George Orwell")
author_tolkien = Author.objects.get(name="J.R.R.
Tolkien")

# Додавання співавторів до книги
book_1984.co_authors.add(author_orwell, author_tolkien)
# SQL: INSERT INTO book_co_authors_author (book_id,
author_id) VALUES (...), (...);

# Видалення співавтора
book_1984.co_authors.remove(author_tolkien)
# SQL: DELETE FROM book_co_authors_author WHERE book_id =
... AND author_id = ...;

# Отримання всіх співавторів
all_coauthors = book_1984.co_authors.all()
for co_author in all_coauthors:
    print(co_author.name)
```

```
# SQL: SELECT author.* FROM author JOIN
book_co_authors_author ON ... WHERE
book_co_authors_author.book_id = ...;
```

3.5. Система міграцій бази даних

Система міграцій є важливою частиною ОРВ, оскільки дозволяє відстежувати зміни в моделях даних та застосовувати їх до схеми бази даних послідовно та контрольовано.

Призначення та важливість міграцій: Міграції вирішують проблему синхронізації структури бази даних з визначеннями моделей в коді. Коли розробник додає нове поле, змінює тип існуючого або видаляє модель, система міграцій генерує інструкції (зазвичай SQL-код) для відповідної зміни схеми БД.

Робота системи міграцій в "NUZP_ORM":

– Управління міграціями здійснюється за допомогою скрипта ORM/manager.py (або, як зазначено в README.md, через python manage.py, де manage.py використовує функціонал з ORM/manager.py).

– Команди:

– generate --app <app_folder>: Аналізує моделі у вказаному додатку (<app_folder>), порівнює їх поточний стан зі станом, зафіксованим в останній міграції, та генерує новий файл міграції, якщо виявлено зміни. Файли міграцій зазвичай містять SQL-команди для застосування змін.

– migrate: Застосовує всі ще не застосовані міграції до бази даних.

– showmigrations: Показує список усіх міграцій та їх статус (застосовано / не застосовано).

– Відстеження застосованих міграцій: "NUZP_ORM" відстежує, які міграції вже були застосовані до бази даних, у спеціальній таблиці

orm_migrations (як зазначено в README.md). Ця таблиця зазвичай зберігає імена застосованих файлів міграцій.

Приклад: додавання нового поля та процес міграції: Припустимо, ми додаємо поле pages_count до моделі Book:

```
# models.py (в якомусь додатку, наприклад, 'library_app')
class Book(BaseModel):
    title = CharField(max_length=200)
    main_author = ForeignKey(Author, on_delete="CASCADE")
    co_authors = ManyToManyField(Author)
    publication_year = IntegerField(null=True)
    pages_count = IntegerField(null=True) # Нове поле
    created_at = DateTimeField(auto_now_add=True)
```

1. Генерація міграції:

```
python manage.py generate --app library_app
```

Це створить новий файл міграції (наприклад, library_app/migrations/0002_add_pages_count_to_book.sql або подібний). Вміст цього файлу міг би бути таким:

```
-- migrations/0002_add_pages_count_to_book.sql
ALTER TABLE book ADD COLUMN pages_count INTEGER;
```

2. Застосування міграції:

```
python manage.py migrate
```

Ця команда виконає SQL з файлу 0002_add_pages_count_to_book.sql та оновить таблицю book в базі даних. Запис про застосування цієї міграції буде додано до таблиці orm_migrations.

3.6. Оптимізація запитів та управління завантаженням даних

Ефективність взаємодії з базою даних є важливим аспектом ОРВ.

Стратегія завантаження даних за замовчуванням: У "NUZP_ORM", як і в багатьох інших ОРВ, для пов'язаних об'єктів (наприклад, при доступі до book.main_author, де main_author є ForeignKey) за замовчуванням використовується "ліниве" завантаження (lazy loading). Це означає, що дані про пов'язаного автора не завантажуються разом із даними про книгу. Запит до бази даних для отримання автора виконується лише в момент першого звернення до атрибута book.main_author. Це може бути ефективним, якщо пов'язані дані потрібні не завжди, але може призвести до проблеми "N+1 запиту", якщо відбувається ітерація по колекції об'єктів і для кожного з них потрібні пов'язані дані.

Можливості "жадібного" завантаження (Eager Loading): На момент розробки та аналізу кодової бази "NUZP_ORM" станом на травень 2025 року, зокрема файлу ORM/query.py, явних механізмів "жадібного" завантаження, аналогічних select_related (для ForeignKey/OneToOne) або prefetch_related (для ManyToMany/зворотних ForeignKey) у Django, **не було реалізовано**. Поточна реалізація покладається на "ліниве" завантаження пов'язаних об'єктів. Впровадження стратегій "жадібного" завантаження є потенційним напрямком для подальшого розвитку та оптимізації ОРВ. Це дозволило б розробникам більш гнучко керувати тим, як і коли завантажуються пов'язані дані, для уникнення проблеми "N+1 запиту" та підвищення продуктивності у відповідних сценаріях.

3.7. Аналіз

3.7.1 Аналіз результатів стрес-тестування

Візуалізація результатів стрес-тестування на графіку 3.1 наочно демонструє поведінку "NUZP_ORM" під зростаючим навантаженням.

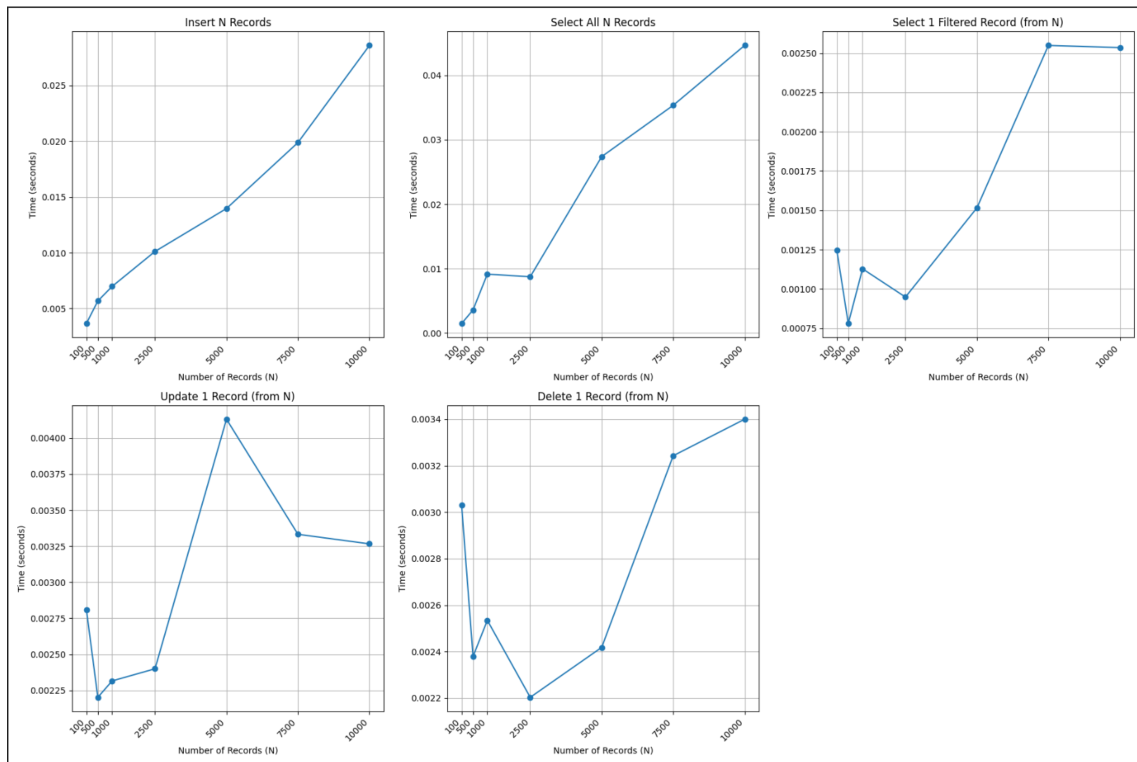

Рисунок 3.1 – Стрес Тестування

Масова Вставка та Вибірка Всіх Записів: Лінії, що відповідають операціям "Insert N Records" (Вставка N записів) та "Select All N Records" (Вибірка всіх N записів), показують чітку тенденцію до зростання часу виконання зі збільшенням кількості записів. Наприклад, вставка 100 записів займає приблизно 0.004 секунди, тоді як для 10,000 записів цей час зростає до ~0.029 секунди. Аналогічно, вибірка всіх записів масштабується від ~0.0015 секунди для 100 записів до ~0.045 секунди для 10,000. Це підтверджує очікувану залежність часу виконання операцій, що працюють з усім набором даних, від його обсягу.

Фільтрована Вибірка, Оновлення та Видалення Одного Запису: На противагу цьому, лінії для операцій "Select 1 Filtered Record" (Вибірка 1 відфільтрованого запису), "Update 1 Record" (Оновлення 1 запису) та "Delete 1 Record" (Видалення 1 запису) залишаються практично горизонтальними та розташовані близько до осі абсцис. Час їх виконання коливається в межах надзвичайно малих значень, приблизно від 0.001 до 0.004 секунди, незалежно від того, чи містить база даних 100 або 10,000 фонових записів. Це свідчить про

високу ефективність цільових операцій, яка, ймовірно, досягається за рахунок ефективного індексування, де продуктивність значною мірою не залежить від загального розміру набору даних.

Таким чином, графічне представлення ефективно підкреслює хорошу продуктивність "NUZP_ORM" для маніпуляцій з окремими записами та передбачуване масштабування для операцій з масивами даних.

3.7.2 Аналіз результатів покриття коду

Результати аналізу покриття коду для "NUZP_ORM" наведені на діаграмах (загальне покриття) та (покриття по модулях/файлах), а також у підсумковій таблиці.

Загальне покриття:
За результатами тестування, загальний рівень покриття коду складає 83% (612 з 740 рядків коду покриті тестами). Це досить високий показник, що свідчить про належну якість тестування та контроль основних функцій системи. Частка непокритих рядків становить лише 17% (128 рядків), що ілюструється на рис. 3.2.

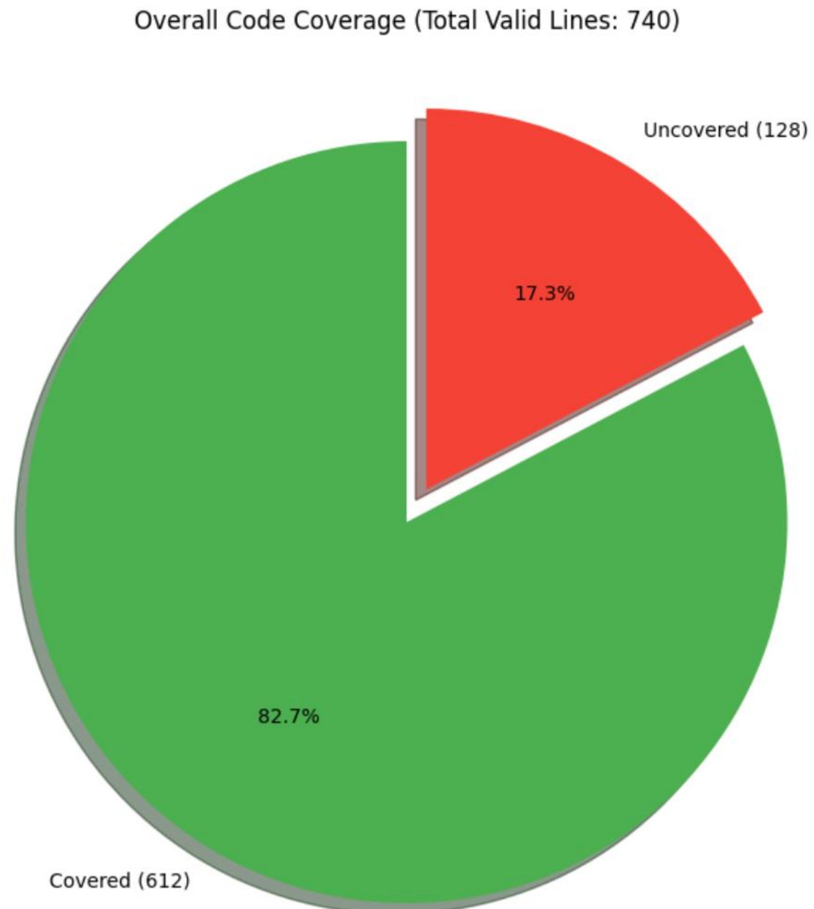

Рисунок 3.2 – Кругова діаграма покриття коду

Покриття по модулях:

Рисунок 3.3 демонструє деталізований розподіл покриття по основних модулях:

– ORM/base.py – 93%;
– ORM/datatypes.py – 97%;
– ORM/fields.py – 75%;
– ORM/manager.py – 70%;
– ORM/query.py – 87%.

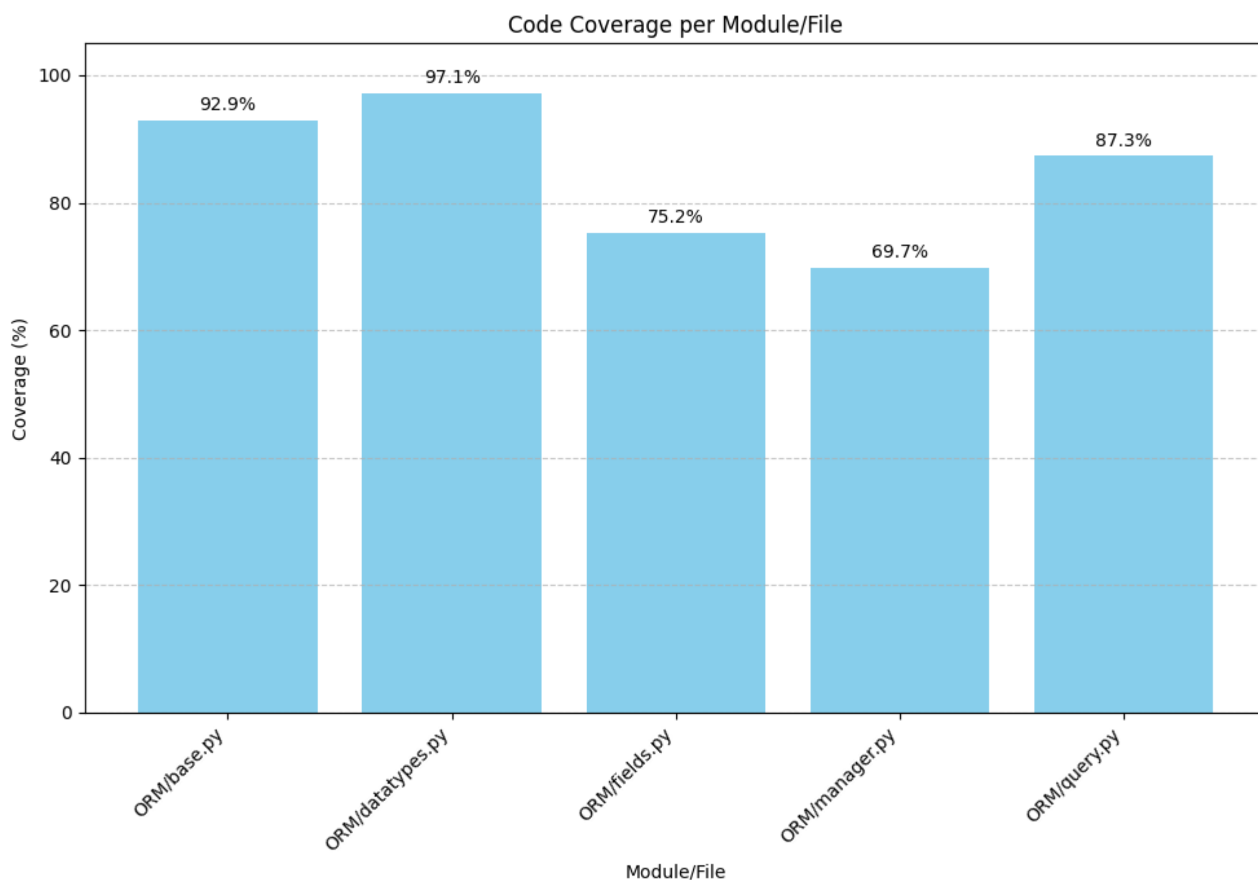

Рисунок 3.3 – Гістограма покриття коду

Найвищий рівень покриття демонструють модулі, що містять базові механізми (ORM/base.py, ORM/datatypes.py), що є критично важливим для стабільності фреймворку. Водночас, модулі ORM/fields.py та особливо ORM/manager.py мають нижчий рівень покриття (75% та 70% відповідно), що вказує на потенційні області для подальшого розширення тестів, особливо для складних гілок логіки та рідко використовуваних сценаріїв.

3.7.3 Аналіз залежності часу виконання від складності запиту

На Рисунку 3.4 показано результати вимірювання часу виконання різноманітних запитів до ORM для типового набору тестових даних (500 студентів та 20 курсів). Кожна точка на графіку відображає середній час

виконання конкретного сценарію запиту, починаючи від простого вибору по унікальному імені до складних many-to-many (M2M) операцій.

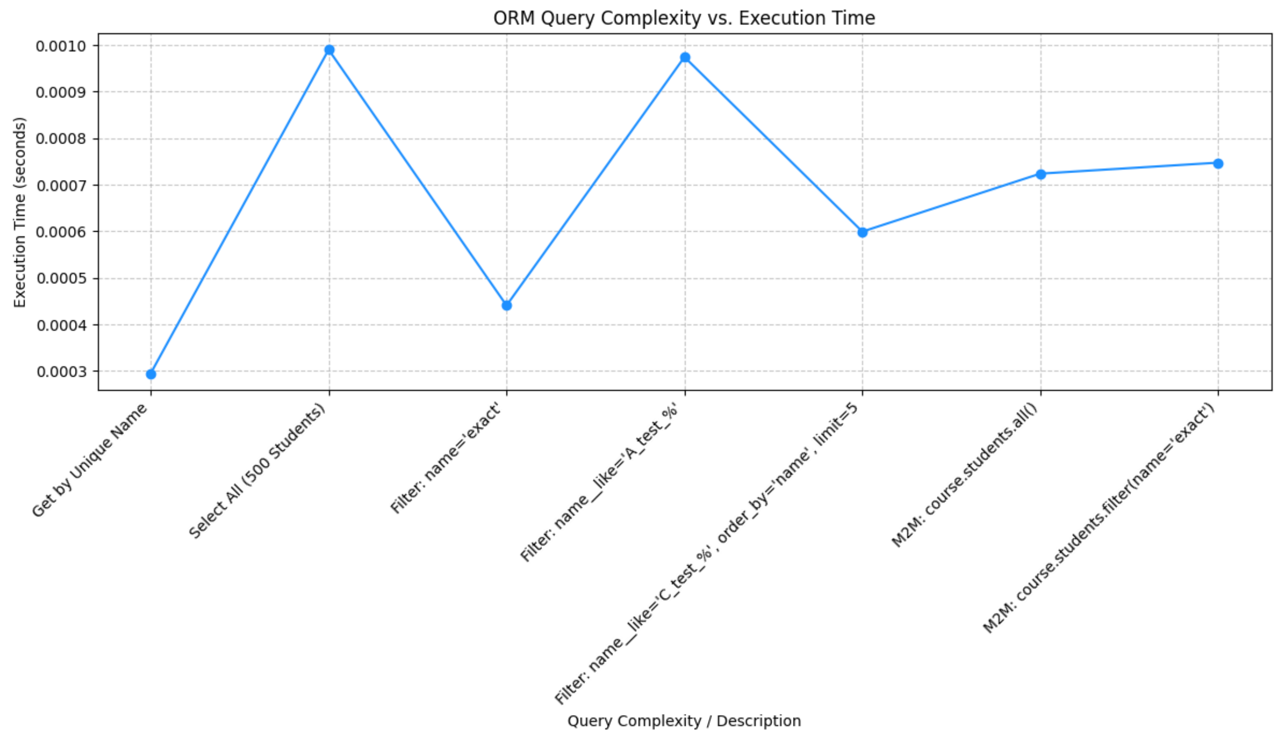

Рисунок 3.4 – результати вимірювання часу виконання різноманітних запитів

Ключові спостереження:

– прості запити виконуються найшвидше:

– Операції "Get by Unique Name" і "Filter: name='exact'" демонструють найменший час виконання (~0.0003–0.0004 с). Це очікувано, оскільки такі запити працюють по індексованих або унікальних полях.

– Вибірка великого обсягу даних ("Select All") дещо повільніша.

Час виконання запиту "Select All (500 Students)" виростає до ~0.001 с, що пов’язано з необхідністю обробки усього результатного набору.

– Складні фільтрації та сортування:

Запити з умовами типу LIKE, додатковим сортуванням чи обмеженням кількості повернутих записів мають час виконання в межах 0.0006–0.001 с.

Це свідчить про хорошу оптимізацію навіть для складніших запитів, принаймні на невеликому обсязі даних.

– Many-to-Many (M2M) запити:

Операції вибірки пов’язаних об’єктів через M2M-зв’язки ("course.students.all()" та "course.students.filter(...)") також знаходяться в діапазоні 0.0007–0.00075 с, що свідчить про відсутність значних накладних витрат при роботі з такими зв’язками в рамках поточного розміру даних.

Висновки:

Залежність часу виконання від складності запиту для "NUZP_ORM" є досить передбачуваною та лінійною на малій вибірці. Навіть складніші запити або взаємодії M2M не призводять до суттєвого зростання часу виконання. Це підкреслює добру оптимізацію базових операцій фреймворку й дозволяє очікувати стабільну продуктивність для стандартних сценаріїв використання. Однак, для подальшого аналізу доцільно виконати аналогічні тести на більших обсягах даних.

3.8. Додаткові можливості та утиліти

Окрім основного функціоналу CRUD та управління зв'язками, "NUZP_ORM" може надавати додаткові утиліти.

Серіалізація екземплярів моделей: Екземпляри моделей мають метод instance.as_dict(), який, ймовірно, повертає словникове представлення даних екземпляра. Це корисно для серіалізації об'єктів, наприклад, при підготовці відповіді для API.

– Приклад:

```
book_instance = Book.objects.get(id=1)
if book_instance:
    book_data_dict = book_instance.as_dict()
    # book_data_dict може виглядати так:
    # {'id': 1, 'title': '1984', 'main_author_id': 1, ...}
```

Розширюваність системи: Хоча деталі механізмів розширення не були глибоко розкриті, будь-яка ОРВ потенційно може бути розширена. Можливі напрямки:

– додавання власних типів полів: Якщо архітектура дозволяє, користувачі могли б створювати власні типи полів зі специфічною логікою валідації або перетворення даних.

– Кастомні методи менеджера або QuerySet: Надання можливості додавати власні методи до менеджера Model.objects або до класу QuerySet для реалізації часто використовуваних або складних запитів.

– Система сигналів/хуків: Впровадження механізму сигналів, які б спрацьовували до або після певних операцій (наприклад, до збереження, після видалення), дозволяючи користувачам додавати власну логіку.

На поточний момент, основна гнучкість забезпечується через стандартні можливості Python, такі як успадкування від BaseModel та використання наданих типів полів. Подальший розвиток міг би включати більш явні точки розширення для користувацької логіки.

ВИСНОВКИ

У ході виконання даної роботи було розроблено систему об'єктно-реляційного відображення "NUZP_ORM" для мови програмування Python з цільовою підтримкою СУБД SQLite3. Система надає розробникам інструментарій для абстрагування від безпосередньої взаємодії з базою даних через SQL, дозволяючи працювати з даними в об'єктно-орієнтованому стилі.

Основні досягнення та ключові особливості "NUZP_ORM":

– визначення моделей: Реалізовано інтуїтивно зрозумілий механізм визначення моделей даних шляхом успадкування від базового класу BaseModel, з автоматичним додаванням первинного ключа id. Підтримуються базові типи даних (CharField, IntegerField, DateTimeField) з опціями null, unique, default.

– Автоматичне управління схемою: Система здатна автоматично транслювати визначення моделей у таблиці бази даних SQLite.

– Операції CRUD: Надано повний набір операцій для маніпулювання даними: створення (insert_entries), читання (через Model.objects з методами all(), get(), filter() з різними пошуковими виразами, order_by(), limit(), offset()), оновлення (replace_entries) та видалення (delete_entries).

– Підтримка зв'язків: Реалізовано підтримку ключових типів зв'язків між моделями: ForeignKey (з каскадним видаленням), OneToOneField та ManyToManyField (з автоматичним створенням проміжної таблиці та спеціалізованим менеджером для управління зв'язками).

– Система міграцій: Розроблено базову систему міграцій, керовану через ORM/manager.py, що дозволяє генерувати, застосовувати та відстежувати зміни в схемі бази даних (generate, migrate, showmigrations).

– Управління завантаженням даних: Пов'язані об'єкти завантажуються "ліниво" (lazy loading), що є типовою стратегією для багатьох ОРВ.

– Додаткові утиліти: Наявний метод instance.as_dict() для простої серіалізації екземплярів моделей.

Співвідношення досягнень з початковими цілями: Проект успішно досягнув поставлених на початку (у підрозділі 2.1) цілей. По-перше, він слугував цінним освітнім інструментом для поглибленого вивчення принципів роботи ОРВ, архітектурних патернів та викликів, пов'язаних з об'єктно-реляційним імпедансом. По-друге, було створено функціональну, легку ОРВ, оптимізовану для SQLite3, яка реалізує основний набір можливостей, необхідних для розробки невеликих та середніх проектів. Завдання, такі як реалізація визначення моделей, CRUD-операцій, підтримки зв'язків та системи міграцій, були успішно виконані.

Обмеження поточної реалізації: Незважаючи на досягнуті результати, поточна версія "NUZP_ORM" має певні обмеження:

– відсутність "жадібного" завантаження: Як зазначено у підрозділі 3.6, система покладається виключно на "ліниве" завантаження, що може призводити до проблеми "N+1 запиту" у певних сценаріях. Відсутні механізми типу select_related або prefetch_related.

– Підтримка лише SQLite3: ОРВ розроблена спеціально для SQLite3 і не підтримує інші СУБД, що обмежує її застосування у проектах, які потребують інших баз даних.

– Базова система міграцій: Хоча система міграцій функціональна, вона може не мати розширених можливостей, таких як автоматичне визначення перейменувань полів, складні операції зі схемою або можливість відкату міграцій.

– Обмежені можливості запитів: Функціонал QuerySet хоч і покриває основні потреби, може не включати більш складні агрегатні функції, групування, підзапити або складні логічні операції безпосередньо в API.

– Управління транзакціями: Деталізоване управління транзакціями (окрім автоматичного коміту, що, ймовірно, відбувається після операцій) може бути не реалізоване або не задокументоване.

– Продуктивність: Для великих обсягів даних або високо навантажених систем продуктивність згенерованих SQL-запитів та внутрішньої логіки ОРВ може потребувати додаткового аналізу та оптимізації.

Потенційні напрямки для майбутнього розвитку та досліджень: На основі поточного стану та виявлених обмежень можна визначити такі напрямки для подальшого вдосконалення "NUZP_ORM":

1. Реалізація "жадібного" завантаження: Додавання методів select_related та prefetch_related для оптимізації запитів до пов'язаних даних.
2. Розширена підтримка СУБД: Адаптація ОРВ для роботи з іншими популярними СУБД (наприклад, PostgreSQL, MySQL) шляхом введення шару абстракції для SQL-діалектів.
3. Вдосконалення системи міграцій: Додавання функцій автоматичного визначення більш складних змін схеми, можливості відкату міграцій, а також, можливо, графічного інтерфейсу або більш детального звітування.
4. Розширення можливостей QuerySet: Впровадження підтримки агрегатних функцій (SUM, AVG, COUNT тощо), анотацій, групування та більш складних умов фільтрації.
5. Явне управління транзакціями: Надання API для явного початку, фіксації та відкату транзакцій.
6. Оптимізація продуктивності: Профілювання та оптимізація генерації SQL-запитів, а також внутрішніх механізмів ОРВ.
7. Додавання підтримки асинхронних операцій: Для використання в асинхронних Python-додатках (наприклад, з asyncio).
8. Покращення системи тестування та документації: Розширення тестового покриття, особливо для складних сценаріїв, та більш детальна документація API.

Загалом, "NUZP_ORM" є успішним проектом, що демонструє розуміння ключових концепцій об'єктно-реляційного відображення та надає робочу основу, яка може бути розширена та вдосконалена в майбутньому.

ПЕРЕЛІК ПОСИЛАНЬ

1. Лукіанов О.В. Об'єктно-орієнтоване програмування (конспект лекцій) : навч. посіб. / О.В. Лукіанов. – Харків : ХНЕУ ім. С. Кузнеця, 2015. – 176 с.
2. Приемы объектно-ориентированного проектирования. Паттерны проектирования / Э. Гамма [та ін.]. – СПб. : Питер, 2001. – 368 с.
3. Python Software Foundation. Python Language Reference, version 3.11. [Електронний ресурс]. – Режим доступу: https://docs.python.org/3/reference/index.html (дата звернення: 10.05.2025).
4. SQLite Documentation. SQL As Understood By SQLite. [Електронний ресурс]. – Режим доступу: https://www.sqlite.org/lang.html (дата звернення: 11.05.2025).
5. Django Software Foundation. Django documentation: Databases. [Електронний ресурс]. – Режим доступу: https://docs.djangoproject.com/en/stable/topics/db/ (дата звернення: 11.05.2025).
6. Fowler M. Patterns of Enterprise Application Architecture / M. Fowler. – Addison-Wesley Professional, 2002. – 560 p.
7. Martin R.C. Clean Code: A Handbook of Agile Software Craftsmanship / R.C. Martin. – Prentice Hall, 2008. – 464 p.
8. ДСТУ ГОСТ 7.1:2006. Бібліографічний запис. Бібліографічний опис. Загальні вимоги та правила складання (ГОСТ 7.1–2003, IDT). Київ : Держспоживстандарт України, 2007.
9. Coverage.py documentation. [Електронний ресурс]. – Режим доступу: https://coverage.readthedocs.io/ (дата звернення: 09.05.2025).
10. Faker documentation. [Електронний ресурс]. – Режим доступу: https://faker.readthedocs.io/ (дата звернення: 10.05.2025).
11. Бояр Є.М. Автоматизація взаємодії з базами даних за допомогою ORM у Python / Є.М. Бояр, А.Є. Рябенко // Інформаційні технології: теорія і

практика: Тези VIII (II) Міжнародної Інтернет-конференції здобувачів вищої освіти і молодих учених (Запоріжжя-Харків-Дніпро, 2-4 квітня 2025 р.), [Електронний ресурс]. – Запоріжжя : НУ «Запорізька політехніка», 2025. – 1 електрон. опт. диск (DVD-ROM); 12 см. – Назва з тит. екрана. – С.36-38.

ДОДАТКИ

#ORM/base.py

```
"""

Defines the core components of the ORM, including the BaseModel, ModelMeta,

and base database interaction methods like create_table, insert, delete, update.

"""

import os

import sqlite3

from ORM.fields import ForeignKey, OneToOneField, ManyToManyField

from ORM.datatypes import Field

from ORM.query import Manager


DB_PATH = "databases/main.sqlite3"



class ModelMeta(type):

"""Metaclass for ORM models."""

def __new__(cls, name, bases, attrs):
```

```
"""
Metaclass __new__ method.
Collects field definitions from class attributes,
initializes
_fields and _many_to_many dictionaries, and sets the
Manager.
"""
fields = {}
many_to_many = {}
for attr_name, attr_value in list(attrs.items()):
if isinstance(attr_value, ManyToManyField):
many_to_many[attr_name] = attr_value
if hasattr(attr_value, '__set_name__'):
attr_value.__set_name__(None, attr_name)
elif isinstance(attr_value, Field):
fields[attr_name] = attr_value

attrs["_fields"] = fields
attrs["_many_to_many"] = many_to_many
new_class = super().__new__(cls, name, bases, attrs)
```

```
return new_class


# ====================================================

# 5. BaseModel: Create tables, insert data, etc.

# ====================================================




class BaseModel(metaclass=ModelMeta):

"""

Base class for all ORM models. Provides core ORM
functionality like

database interaction (CRUD), field handling, and instance
representation.

Subclass this to define your application models.

"""

objects = Manager()

id = None # Initialize id attribute


def __init__(self, **kwargs):

"""

Initializes a model instance.
```

```
Sets attributes based on provided keyword arguments.

Defaults fields not provided in kwargs to None.

Ensures 'id' is initialized, defaulting to None.

"""

# Initialize all defined fields (_fields comes from
metaclass)

for field_name in self._fields:

# Get the value from kwargs if provided, otherwise
default to None

value = kwargs.get(field_name, None)

setattr(self, field_name, value)


# Handle 'id' specifically - it might be in kwargs or
should be None initially

self.id = kwargs.get('id', None)


def __repr__(self):

"""Return a string representation of the model
instance."""

# Use the instance's ID if available, otherwise indicate
it's unsaved

string = ", ".join(f"{k}={value!r}" for k,
```

```
value in self.__dict__.items())
return f"<{self.__class__.__name__}: {string}>"


def as_dict(self):
"""Return a dictionary representation of the model
instance."""
data = {'id': self.id}
# Handle regular fields and FK/O2O fields
for field_name, field in self._fields.items():
if isinstance(field, (ForeignKey, OneToOneField)):
# For FK/O2O, store the related object's ID
# Check for the _id attribute first (set during loading)
fk_id_attr = field_name + '_id'
fk_id = getattr(self, fk_id_attr, None)


# Check fk_id is still None
if fk_id is None and hasattr(self, field_name):
potential_related_obj = getattr(self, field_name)
if isinstance(potential_related_obj, field.to) and
potential_related_obj.id is not None:
fk_id = potential_related_obj.id
```

```
data[fk_id_attr] = fk_id
else:
# Regular field
data[field_name] = getattr(self, field_name, None)

# Handle M2M fields
for field_name in self._many_to_many: # Iterate through
field names
# M2M relationships require the instance to have an ID
if self.id is not None:
try:
# Get the manager instance for this field on this
specific object
# Accessing self.<field_name> (e.g., self.courses)
triggers the descriptor's __get__
manager = getattr(self, field_name)

# Call the manager's all() method, which returns a
QuerySet
related_queryset = manager.all()
```

```
# Extract the IDs from the related instances in the
QuerySet

data[field_name] = [

instance.id for instance in related_queryset if
instance.id is not None]

except Exception as e:

# Handle potential errors during M2M fetch gracefully

print(

f"Warning: Could not fetch M2M field '{field_name}' for
{self}: {e}")

data[field_name] = [] # Represent as empty list on error

else:

# If the instance isn't saved, it can't have M2M
relations yet

data[field_name] = [] # Represent as empty list


return data


@classmethod

def create_table(cls):

"""
```

```
Creates the database table for this model, including
columns for

all defined fields and junction tables for
ManyToManyFields.

Drops the table if it already exists before creating.

"""

if not os.path.exists('databases'):

os.makedirs('databases')


table_name = cls.__name__.lower()

connection_obj = sqlite3.connect(DB_PATH)

cursor_obj = connection_obj.cursor()

fields_sql = ["id INTEGER PRIMARY KEY AUTOINCREMENT NOT
NULL"]


for field_name, field in cls._fields.items():

if isinstance(field, (ForeignKey, OneToOneField)):

# Store foreign keys as "<field_name>_id"

column_name = field_name + "_id"

ref_table = field.to.__name__.lower() # get referenced
table

# delete everything if id deleted
```

```
fields_sql.append(
f"{column_name} {field.db_type} REFERENCES
{ref_table}(id) ON DELETE CASCADE")
else:
fields_sql.append(f"{field_name} {field.get_db_type()}")
cursor_obj.execute(f"DROP TABLE IF EXISTS {table_name}")
cursor_obj.execute(
f"CREATE TABLE IF NOT EXISTS {table_name} ({',
'.join(fields_sql)});")


for field_name, field in cls._many_to_many.items():
junction_table = field.through or
f"{table_name}_{field.to.__name__.lower()}"
cursor_obj.execute(f"""
CREATE TABLE IF NOT EXISTS {junction_table} (
id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
{table_name}_id INTEGER REFERENCES {table_name}(id) ON
DELETE CASCADE,
{field.to.__name__.lower()}_id INTEGER REFERENCES
{field.to.__name__.lower()}(id) ON DELETE CASCADE,
UNIQUE({table_name}_id, {field.to.__name__.lower()}_id)
);
```

```
""")
connection_obj.close()


# TODO: M2M and insert entries are separate functions.
Merge them.
@classmethod
def _validate_insert_input(cls, entries) -> bool:
"""
Validate the list of entries for insertion.


Returns True if entries are dictionaries, False if they
are model instances.
Raises TypeError if entries are neither dictionaries nor
model instances.
"""
if not entries:
print("No entries provided to insert.")
return None, None # Indicate no processing needed


first_entry = entries[0]
is_dict_input = isinstance(first_entry, dict)
```

```
is_model_instance_input = isinstance(first_entry,
BaseModel)


if not is_dict_input and not is_model_instance_input:

raise TypeError(

"Entries must be a list of dictionaries or BaseModel
instances.")


# Check type consistency

for entry in entries:

if is_dict_input and not isinstance(entry, dict):

raise TypeError("All entries must be dictionaries.")

if is_model_instance_input and not isinstance(entry,
BaseModel):

raise TypeError("All entries must be BaseModel
instances.")

if is_model_instance_input and not isinstance(entry,
cls):

raise TypeError(

f"All entries must be instances of {cls.__name__}")


return is_dict_input
```

```
@classmethod
def _prepare_insert_sql(cls):
"""Prepare SQL query components for insertion."""
field_names_db = []
field_names_model = []
for field_name, field in cls._fields.items():
field_names_model.append(field_name)
if isinstance(field, (ForeignKey, OneToOneField)):
field_names_db.append(field_name + "_id")
else:
field_names_db.append(field_name)


placeholders = ", ".join(["?" for _ in field_names_db])
columns = ", ".join(field_names_db)
query = f"INSERT INTO {cls.__name__.lower()} ({columns})
VALUES ({placeholders})"
return field_names_model, field_names_db, query


@classmethod
```

```
def _extract_value_for_db(cls, entry, model_field_name,
field, is_dict_input):

"""Extract and process a single value from an entry for
DB insertion."""

value = None

if is_dict_input:

raw_value = entry.get(model_field_name)

if isinstance(field, (ForeignKey, OneToOneField)):

if isinstance(raw_value, dict):

value = raw_value.get('id')

elif isinstance(raw_value, BaseModel):

value = getattr(raw_value, 'id', None)

else: # Assume it's the ID

value = raw_value

else:

value = raw_value

else: # is_model_instance_input

raw_value = getattr(entry, model_field_name, None)

if isinstance(field, (ForeignKey, OneToOneField)):

value = getattr(raw_value, 'id', None) if raw_value else
None
```

```
else:

value = raw_value

return value


@classmethod

def _check_onetoone_constraint(cls, cursor_obj,
db_field_name, model_field_name, value):

"""Check for OneToOne constraint violation."""

check_query = f"SELECT COUNT(*) FROM
{cls.__name__.lower()} WHERE {db_field_name} = ?"

cursor_obj.execute(check_query, (value,))

if cursor_obj.fetchone()[0] > 0:

raise ValueError(

f"Duplicate entry detected for {model_field_name}
(OneToOneField) with id {value}")


@classmethod

def _process_entries_for_values(cls, entries,
is_dict_input, field_names_model, field_names_db,
cursor_obj):

"""Process all entries to generate the list of value
tuples for executemany."""
```

```
values = []

# Keep track of O2O FK values seen within this batch for
each O2O field

seen_onetoone_values = {

fn: set() for fn, f in cls._fields.items() if
isinstance(f, OneToOneField)

}


# Use enumerate for better error context

for entry_index, entry in enumerate(entries):

row = []

for model_field_name, db_field_name in
zip(field_names_model, field_names_db):

field = cls._fields[model_field_name]

value = cls._extract_value_for_db(

entry, model_field_name, field, is_dict_input)


if isinstance(field, OneToOneField) and value is not
None:

# 1. Check for duplicates within the current batch first

if value in seen_onetoone_values[model_field_name]:
```

```
# Raise ValueError immediately if duplicate found in batch
raise ValueError(
f"Duplicate entry detected within the batch for OneToOne field '{model_field_name}' with value {value} at index {entry_index}"
)
# Add the value to the set for this batch check
seen_onetoone_values[model_field_name].add(value)

# 2. Check against the database (existing check)
try:
cls._check_onetoone_constraint(
cursor_obj, db_field_name, model_field_name, value)
except ValueError as e:
# Add index context if database check fails
raise ValueError(
f"Error processing entry at index {entry_index}: {e}") from e

row.append(value)
values.append(tuple(row))
```

```
return values


@classmethod

def _execute_insert(cls, connection_obj, cursor_obj,
query, entries, values_list, is_dict_input):

"""

Execute the insert query and handle commit/rollback.

Updates instance IDs if inserting model instances one by
one.

"""

try:

if is_dict_input:

# Use executemany for dictionary inputs (bulk insert)

cursor_obj.executemany(query, values_list)

print(

f"Successfully inserted {len(values_list)} entries into
{cls.__name__}")

else:

# Insert instances one by one to get lastrowid

inserted_count = 0

for i, entry_instance in enumerate(entries):
```

```
# Get the pre-processed values for this instance
values_tuple = values_list[i]
cursor_obj.execute(query, values_tuple)
# Get the last inserted ID and update the instance
last_id = cursor_obj.lastrowid
entry_instance.id = last_id
inserted_count += 1
print(
f"Successfully inserted {inserted_count} entries into
{cls.__name__} and updated instance IDs.")

connection_obj.commit()

except Exception as e:
connection_obj.rollback()
# Log or print error
print(f"Error during insert into {cls.__name__}: {e}")
raise # Re-raise the exception after rollback

@classmethod
def insert_entries(cls, entries):
```

```
"""
Inserts multiple entries (rows) into the model's database
table.

Args:
entries (list): A list of model instances or dictionaries
representing the data to insert.

Returns:
list: The list of inserted entries (as model instances)
with
their assigned IDs updated.

Raises:
TypeError: If the input list contains mixed types or
invalid types.
ValueError: If a dictionary entry is missing required
fields or
if a unique constraint (like OneToOne) is violated.
sqlite3.IntegrityError: If a database constraint (e.g.,
unique)
is violated during insertion.
```

```
"""
is_dict_input = cls._validate_insert_input(entries)
if is_dict_input is None: # Handle case where entries
list is empty
print("No entries to insert.")
return


if not os.path.exists(DB_PATH):
raise ValueError(f"Database for {cls.__name__} does not
exist!")


connection_obj = None # Initialize to None for finally
block
try:
connection_obj = sqlite3.connect(DB_PATH)
cursor_obj = connection_obj.cursor()
cursor_obj.execute("PRAGMA foreign_keys = ON;")


field_names_model, field_names_db, query =
cls._prepare_insert_sql()
```

```
# Process entries to get values list (needed for both
dicts and instances)

values_list = cls._process_entries_for_values(

entries, is_dict_input, field_names_model,
field_names_db, cursor_obj

)



# Pass entries list along with values_list to
_execute_insert

cls._execute_insert(connection_obj, cursor_obj,

query, entries, values_list, is_dict_input)



except Exception as e:

# Catch potential errors during validation, SQL prep, or
processing

if connection_obj:

try:

connection_obj.rollback()

except Exception as rb_e:

# Log rollback error

print(f"Error during rollback: {rb_e}")

# Log or print error
```

```
print(f"Failed to insert entries into {cls.__name__}:
{e}")

# Re-raise the original exception to signal failure

raise

finally:

if connection_obj:

connection_obj.close()


@classmethod

def delete_entries(cls, conditions=None,
confirm_delete_all=False):

"""

Deletes entries from the model's table based on specified
conditions.


Args:

conditions (dict, optional): A dictionary of field-value
pairs

to filter which rows to delete.

Defaults to None (delete all).

confirm_delete_all (bool, optional): Must be set to True
to allow
```

```
deleting all entries when

conditions is None. Defaults to False.


Returns:

int: The number of rows deleted.


Raises:

ValueError: If attempting to delete all entries without
setting

confirm_delete_all=True.

"""

if not os.path.exists(DB_PATH):

raise ValueError(f"Database for {cls.__name__} does not
exist!")


connection_obj = sqlite3.connect(DB_PATH)

cursor_obj = connection_obj.cursor()

cursor_obj.execute("PRAGMA foreign_keys = ON;")


if not conditions:
```

```
if confirm_delete_all or input(f"Are you sure you want to
delete ALL records from {cls.__name__}? (yes/no):
").lower() == "yes":

query = f"DELETE FROM {cls.__name__.lower()}"

cursor_obj.execute(query)

else:

print("Deletion cancelled.")

return

else:

where_clause = " AND ".join(

[f"{field} = ?" for field in conditions.keys()])

query = f"DELETE FROM {cls.__name__.lower()} WHERE
{where_clause}"

values = tuple(conditions.values())

cursor_obj.execute(query, values)


connection_obj.commit()

print(f"Deleted entries from {cls.__name__} where
{conditions}")

connection_obj.close()


@classmethod
```

```
def replace_entries(cls, conditions, new_values):
"""
Updates entries in the model's table that match the given
conditions
with the new values provided.

Args:
conditions (dict): A dictionary of field-value pairs to
filter
which rows to update.
new_values (dict): A dictionary of field-value pairs
representing
the new data for the matched rows.

Returns:
int: The number of rows updated.

Raises:
ValueError: If conditions or new_values are empty or
invalid.
"""
if not os.path.exists(DB_PATH):
```

```
raise ValueError(f"Database for {cls.__name__} does not
exist!")

connection_obj = sqlite3.connect(DB_PATH)

cursor_obj = connection_obj.cursor()

cursor_obj.execute("PRAGMA foreign_keys = ON;")

if not conditions:

print(

"Error: You must provide at least one condition to update
specific rows.")

return

if not new_values:

print("Error: No new values provided to update.")

return

set_clause = ", ".join([f"{field} = ?" for field in
new_values.keys()])

where_clause = " AND ".join(

[f"{field} = ?" for field in conditions.keys()])

query = f"UPDATE {cls.__name__.lower()} SET {set_clause}
WHERE {where_clause}"

values = tuple(new_values.values()) +
tuple(conditions.values())

try:
```

```
cursor_obj.execute(query, values)

connection_obj.commit()

# print(

# f"Updated entries in {cls.__name__} where {conditions}
with {new_values}")

except Exception as e:

print(f"Error updating entries: {e}")

raise e

finally:

connection_obj.close()
```

#ORM/datatypes.py

```
"""
Defines the basic data types (Field, CharField,
IntegerField, DateTimeField)
used for model fields and their corresponding SQL
representation.
"""
class Field:
"""Represents a database field."""
def __init__(self, db_type, null=True, unique=False,
default=None, max_length=None):
"""
Initialize a field.

:param db_type: The database type (e.g., TEXT, INTEGER,
DATETIME).
:param null: If True, the field is nullable. If False,
the field is NOT NULL.
"""
self.db_type = db_type
self.null = null
```

```
self.unique = unique

self.default = default

self.max_length = max_length


def get_db_type(self):

"""

Returns the SQL data type string for this field,
including constraints

like NOT NULL and UNIQUE based on the field's options.

"""

parts = [self.db_type]

if not self.null:

parts.append("NOT NULL")

# Ensure UNIQUE is checked independently of NULL

if self.unique: # Changed from potential 'elif' or faulty
logic

parts.append("UNIQUE")

# Handle default value if applicable (might be in
subclass)

if hasattr(self, 'default') and self.default is not None:

# Ensure proper formatting for SQL DEFAULT clause, e.g.,
strings need quotes
```

```
default_val = self.default

if isinstance(default_val, str):

default_val = f"'{default_val}'" # Add quotes for string
defaults

parts.append(f"DEFAULT {default_val}")


return " ".join(parts)




class CharField(Field):

"""Represents a character string field (VARCHAR) in the
database."""

db_type = 'VARCHAR'

def __init__(self, null=True, unique=False, default=None,
max_length=None):

"""

Initialize a character field.


:param null: If True, the field is nullable. If False,
the field is NOT NULL.

:param unique: If True, add a UNIQUE constraint.

"""
```

```
super().__init__("TEXT", null=null, unique=unique)

self.default = default

self.max_length = max_length



class IntegerField(Field):

"""Represents an integer field (INTEGER) in the
database."""

db_type = 'INTEGER'

def __init__(self, null=True, default=0, unique=False):

"""

Initialize an integer field.


:param null: If True, the field is nullable. If False,
the field is NOT NULL.

:param default: The default value for the field.

:param unique: If True, add a UNIQUE constraint.

"""

super().__init__("INTEGER", null=null, unique=unique)

self.default = default
```

```
class DateTimeField(Field):

"""Represents a date/time field (DATETIME) in the
database."""

db_type = 'DATETIME'

def __init__(self, null=True, unique=False,
default=None):

"""

Initialize a datetime field.


:param null: If True, the field is nullable. If False,
the field is NOT NULL.

:param unique: If True, add a UNIQUE constraint.

"""

super().__init__("DATETIME", null=null, unique=unique)

self.default = default
```

#ORM/fields.py

```
"""

Defines relationship fields (ForeignKey, OneToOneField,
ManyToManyField)

and the manager for handling ManyToMany relationships.

"""
```

```
import sqlite3
from ORM.datatypes import Field
from ORM.query import QuerySet


DB_PATH = "databases/main.sqlite3"

# ====================================================
# 2. Relationship Field Types
# ====================================================


class ManyToManyRelatedManager:
"""
Manages M2M relationships for a specific instance.
Accessed via a descriptor on the model instance (e.g.,
student.courses).
"""
def __init__(self, instance, field):
"""
Initialize the manager.
```

```
Args:

instance (BaseModel): The source model instance (e.g., a
Student instance).

field (ManyToManyField): The ManyToManyField descriptor
instance.

"""

self.instance = instance

self.field = field

self.source_class = instance.__class__

self.target_class = field.to

self.source_class_name =
self.source_class.__name__.lower()

self.target_class_name =
self.target_class.__name__.lower()

self.junction_table = field.through or
f"{self.source_class_name}_{self.target_class_name}"


def _check_instance_saved(self, operation="operate"):

"""Ensure the source instance is saved before performing
relationship operations."""

if self.instance.id is None:
```

```
raise ValueError(f"Cannot {operation} on M2M relationship
for an unsaved '{self.source_class_name}' instance.")


def add(self, *target_objs):
"""
Add one or more target objects to the relationship.
"""
self._check_instance_saved("add")
connection_obj = sqlite3.connect(DB_PATH)
cursor_obj = connection_obj.cursor()
cursor_obj.execute("PRAGMA foreign_keys = ON;")
try:
for target_obj in target_objs:
if not isinstance(target_obj, self.target_class):
raise TypeError(f"Can only add
'{self.target_class.__name__}' instances.")
if target_obj.id is None:
raise ValueError(f"Cannot add unsaved
'{self.target_class_name}' instance to M2M
relationship.")
```

```
# Use INSERT OR IGNORE to handle potential UNIQUE
constraint violations gracefully

cursor_obj.execute(f"""

INSERT OR IGNORE INTO {self.junction_table}
({self.source_class_name}_id,
{self.target_class_name}_id)

VALUES (?, ?)

""", (self.instance.id, target_obj.id))

connection_obj.commit()

except sqlite3.IntegrityError as e:

# Handle FK constraint violation if target_obj.id doesn't
exist in target table

if "FOREIGN KEY constraint failed" in str(e):

connection_obj.rollback()

# Find which object caused the error (simplified check)

offending_id = next((obj.id for obj in target_objs if
obj.id is not None), 'unknown')

raise ValueError(f"Invalid target ID '{offending_id}' for
M2M relationship.") from e

else:

connection_obj.rollback()

raise e # Re-raise other IntegrityErrors
```

```
except Exception as e:

connection_obj.rollback()

raise e

finally:

connection_obj.close()



def remove(self, *target_objs):

"""

Remove one or more target objects from the relationship.

"""

self._check_instance_saved("remove")

connection_obj = sqlite3.connect(DB_PATH)

cursor_obj = connection_obj.cursor()

cursor_obj.execute("PRAGMA foreign_keys = ON;")

try:

for target_obj in target_objs:

if not isinstance(target_obj, self.target_class):

raise TypeError(f"Can only remove
'{self.target_class.__name__}' instances.")

if target_obj.id is None:

# Cannot remove relationship if target ID is unknown
```

```
print(f"Warning: Cannot remove M2M relationship for
unsaved '{self.target_class_name}' instance.")

continue



cursor_obj.execute(f"""

DELETE FROM {self.junction_table}

WHERE {self.source_class_name}_id = ? AND
{self.target_class_name}_id = ?

""", (self.instance.id, target_obj.id))

connection_obj.commit()

except Exception as e:

connection_obj.rollback()

raise e

finally:

connection_obj.close()



def clear(self):

"""Remove all relationships for this instance."""

self._check_instance_saved("clear")

connection_obj = sqlite3.connect(DB_PATH)

cursor_obj = connection_obj.cursor()
```

```
cursor_obj.execute("PRAGMA foreign_keys = ON;")

try:

cursor_obj.execute(f"""

DELETE FROM {self.junction_table}

WHERE {self.source_class_name}_id = ?

""", (self.instance.id,))

connection_obj.commit()

except Exception as e:

connection_obj.rollback()

raise e

finally:

connection_obj.close()


def set(self, target_objs):

"""

Replace the current set of related objects with the
provided ones.

"""

self._check_instance_saved("set")

self.clear()

if target_objs: # Only add if there are objects to add
```

```
self.add(*target_objs)


def all(self):
"""
Retrieve all related target objects as a QuerySet.
"""
self._check_instance_saved("retrieve")
# Construct a QuerySet for the target model, filtered by
the junction table
target_ids_query = f"""
SELECT {self.target_class_name}_id
FROM {self.junction_table}
WHERE {self.source_class_name}_id = ?
"""
connection_obj = sqlite3.connect(DB_PATH)
cursor_obj = connection_obj.cursor()
cursor_obj.execute(target_ids_query, (self.instance.id,))
target_ids = [row[0] for row in cursor_obj.fetchall()]
connection_obj.close()


if not target_ids:
```

```
# Return an empty QuerySet if no related objects

return QuerySet(self.target_class).filter(id__in=[]) #
Filter with empty list


# Use the 'id__in' filter on the target model's manager

return
QuerySet(self.target_class).filter(id__in=target_ids)


def filter(self, **kwargs):

"""Filter the set of related objects."""

return self.all().filter(**kwargs)


def get(self, **kwargs):

"""Get a single related object matching the criteria."""

return self.all().get(**kwargs)


def __iter__(self):

"""Allow iteration over related objects."""

return iter(self.all())


def __repr__(self):
```

```
"""Return a string representation of the manager."""

# Fetch related objects for representation using the
'all' method

related_objects = list(self.all()) # Convert QuerySet to
list for repr

return f"<ManyToManyRelatedManager for {self.instance} ->
{self.field.to.__name__}: {related_objects}>"




# ====================================================

# 2. Relationship Field Types

# ====================================================



class ManyToManyField(Field):

"""

Defines a many-to-many relationship between two models.



Requires a junction table to store the associations. This
table can be

automatically generated or specified using the `through`
argument.

"""
```

```
def __init__(self, to, through=None, **kwargs):
"""
Initializes the ManyToManyField.


Args:
to (class): The model class to which the relationship is
established.
through (str, optional): The name of the intermediate
junction table.
If None, a default name is generated.
**kwargs: Additional field options (inherited from Field,
but typically not used).
"""
self.to = to # Target model class
self.through = through # Optional custom junction table
# Store the name this field is assigned to on the model
self.name = None # Will be set by ModelMeta


def __set_name__(self, owner, name):
"""Called when the field is assigned to a model class
attribute."""
```

```
self.name = name # Store the attribute name (e.g.,
'courses')


def __get__(self, instance, owner):
"""
Descriptor __get__ method. Returns the manager when
accessed on an instance.
"""
if instance is None:
# Accessed on the class, maybe return a manager that
works across all instances?
# For now, let's return self or raise error, as instance
access is primary goal.
return self # Or raise AttributeError("M2M field can only
be accessed via an instance")


# Check if a manager instance is already cached on the
instance
# Use a private attribute name based on the field name
manager_attr = f"_{self.name}_manager"
if not hasattr(instance, manager_attr):
# Create and cache the manager instance
```

```
setattr(instance, manager_attr,
ManyToManyRelatedManager(instance, self))

return getattr(instance, manager_attr)



class ForeignKey(Field):

"""

Implements a one-to-many relationship (the "many" side).

In the database, we store the related object's id as an
INTEGER.

"""

def __init__(self, to, **kwargs):

"""

Initializes the ForeignKey field.


Args:

to (class): The model class that this field references.

**kwargs: Additional field options (e.g., null, unique,
default).

"""

self.to = to # the target model class
```

```
super().__init__("INTEGER", **kwargs)


class OneToOneField(ForeignKey):
"""
Implements a one-to-one relationship.
This is just a ForeignKey with a UNIQUE constraint.
"""
def __init__(self, to, **kwargs):
"""
Initializes the OneToOneField.

Ensures the relationship is unique by setting
unique=True.

Args:
to (class): The model class that this field references.
**kwargs: Additional field options (e.g., null, default).
'unique' is always forced to True.
"""
kwargs.setdefault('unique', True)
```

```
super().__init__(to, **kwargs)
```

#ORM/manager.py

```
"""
Provides command-line utilities for managing database
migrations,
including generating, applying, and showing migration
status.
"""
import sqlite3
import os
import sys
import importlib
import inspect
import argparse
from pathlib import Path


from ORM.base import BaseModel, DB_PATH



def find_models(project_root, models_folder='myapp'):
"""Find all model classes in the specified folder that
inherit from BaseModel."""
models = []
```

```
# Construct the path to the models folder

models_path = os.path.join(project_root, models_folder)


# Check if the folder exists

if not os.path.exists(models_path):

print(

f"The folder '{models_folder}' does not exist in the
project root.")

return models


# Walk only through the models folder

for root, dirs, files in os.walk(models_path):

for file in files:

if file.endswith('.py') and not file.startswith('__'):

file_path = os.path.join(root, file)

module_path = os.path.relpath(file_path,
project_root).replace(

'/', '.').replace('\\', '.')[:-3]
```

```
print(f"Examining {file_path} -> module path:
{module_path}")


try:

# Import the module

module = importlib.import_module(module_path)


# Find model classes in the module

classes_found = False

for name, obj in inspect.getmembers(module):

if inspect.isclass(obj):

print(f" Found class: {name}")

if issubclass(obj, BaseModel) and obj != BaseModel:

print(f" --> {name} is a model!")

models.append(obj)

classes_found = True


if not classes_found:

print(f" No model classes found in {file_path}")


except (ImportError, ModuleNotFoundError) as e:
```

```
print(f" Error importing {module_path}: {e}")
except Exception as e:
print(f" Unexpected error with {module_path}: {e}")

return models



def generate_migrations(models):
"""Generate versioned migration files for all models."""
if not models:
print("No models provided. Skipping migration
generation.")
return

migrations_dir = Path('migrations')
migrations_dir.mkdir(exist_ok=True)

# Get the next migration number
existing_migrations = [f for f in
migrations_dir.glob('????_*.py')]
next_number = 1
```

```
if existing_migrations:
latest = max(existing_migrations)
next_number = int(latest.name[:4]) + 1

# Enhanced model change detection
from hashlib import sha256

# Create detailed signatures that include field
information
model_signatures = {}
for model in models:
# Capture basic model info
model_info = f"{model.__name__}:{model.__module__}"

# Add regular fields
fields_info = []
for field_name, field in model._fields.items():
# Get field type
field_type = type(field).__name__

# Get field attributes
```

```
attrs = {}

for attr_name in ['db_type', 'null', 'unique',
'default']:

if hasattr(field, attr_name):

attrs[attr_name] = getattr(field, attr_name)


# For foreign key fields, include target model

if hasattr(field, 'to'):

attrs['to'] = field.to.__name__


# Create field signature

field_signature = f"{field_name}:{field_type}:{attrs}"

fields_info.append(field_signature)


# Add many-to-many fields if present

if hasattr(model, '_many_to_many'):

for field_name, field in model._many_to_many.items():

field_type = "ManyToManyField"

attrs = {

'to': field.to.__name__,

}
```

```
if field.through:

attrs['through'] = field.through


field_signature = f"{field_name}:{field_type}:{attrs}"

fields_info.append(field_signature)


# Create complete model signature

model_signature =
f"{model_info}:{','.join(sorted(fields_info))}"

model_signatures[model.__name__] = sha256(

model_signature.encode()).hexdigest()


# Load the last migration's signature if it exists

signature_file = migrations_dir / 'last_signature.txt'

if signature_file.exists():

with open(signature_file, 'r') as f:

last_signature = f.read().strip()

current_signature =
sha256(str(model_signatures).encode()).hexdigest()


if last_signature == current_signature:
```

```
print("No changes detected in models. Skipping migration
generation.")

return

else:

current_signature =
sha256(str(model_signatures).encode()).hexdigest()


# Create a migration file with timestamp and sequential
number

from datetime import datetime

timestamp = datetime.now().strftime("%Y%m%d_%H%M")

migration_file = migrations_dir / \

f"{next_number:04d}_migration_{timestamp}.py"


with open(migration_file, 'w') as f:

f.write(f"# Auto-generated migration file\n\n")

f.write("def migrate():\n")


for model in models:

f.write(f" # Create table for {model.__name__}\n")

f.write(f" from {model.__module__} import
{model.__name__}\n")
```

```
f.write(f" {model.__name__}.create_table()\n\n")


# Save the current signature

with open(signature_file, 'w') as f:

f.write(current_signature)


print(f"Generated migration file: {migration_file}")




def create_migrations_table():

"""Create a table to track applied migrations if it
doesn't exist."""

# Ensure databases directory exists

if not os.path.exists(os.path.dirname(DB_PATH)):

os.makedirs(os.path.dirname(DB_PATH))


connection = sqlite3.connect(DB_PATH)

cursor = connection.cursor()

cursor.execute("""

CREATE TABLE IF NOT EXISTS orm_migrations (

id INTEGER PRIMARY KEY AUTOINCREMENT,
```

```
migration_name VARCHAR(255) UNIQUE,
applied_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
)
""")
connection.commit()
connection.close()
print("Migration tracking table ensured.")



def record_migration(migration_name):
"""Record that a migration has been applied."""
connection = sqlite3.connect(DB_PATH)
cursor = connection.cursor()
try:
cursor.execute(
"INSERT INTO orm_migrations (migration_name) VALUES (?)",
(migration_name,))
connection.commit()
print(f"Recorded migration: {migration_name}")
except sqlite3.IntegrityError:
# Migration already recorded (unique constraint)
```

```
print(f"Migration {migration_name} already recorded.")

except Exception as e:

print(f"Error recording migration {migration_name}: {e}")

finally:

connection.close()



def get_applied_migrations():

"""Get a list of migrations that have already been
applied."""

try:

connection = sqlite3.connect(DB_PATH)

cursor = connection.cursor()

cursor.execute("SELECT migration_name FROM orm_migrations
ORDER BY id")

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

connection.close()

return applied

except sqlite3.OperationalError:

# Table doesn't exist yet

return []
```

```
except Exception as e:
print(f"Error retrieving applied migrations: {e}")
return []


def apply_migrations(specific_migration=None):
"""Apply all migrations or a specific migration in
sequential order."""
# Ensure migrations table exists
create_migrations_table()

# Get list of applied migrations
applied_migrations = get_applied_migrations()
print(f"Already applied migrations: {',
'.join(applied_migrations) if applied_migrations else
'None'}")

migrations_dir = Path('migrations')
if not migrations_dir.exists():
print("No migrations directory found. Run 'generate'
first.")
return
```

```
# Get all migration files in sorted order

migration_files =
sorted(migrations_dir.glob('????_*.py'))

if not migration_files:

print("No migration files found. Run 'generate' first.")

return



sys.path.insert(0, str(migrations_dir.parent))



# If applying a specific migration

if specific_migration:

migration_file = next((f for f in migration_files if
f.stem == specific_migration), None)

if not migration_file:

print(f"Migration '{specific_migration}' not found.")

return

# Skip if already applied

if specific_migration in applied_migrations:

print(f"Migration {specific_migration} already applied,
skipping.")
```

```
return

try:

print(f"Applying specific migration:
{specific_migration}")

migration_module =
importlib.import_module(f'migrations.{specific_migration}
')

migration_module.migrate()

# Record the migration as applied

record_migration(specific_migration)

print(f"Migration {specific_migration} applied
successfully!")

except Exception as e:

print(f"Error applying migration {specific_migration}:
{e}")

raise # Re-raise the exception for test cases

return


# Apply all migrations in sequence

for migration_file in migration_files:

module_name = migration_file.stem
```

```
# Skip if already applied

if module_name in applied_migrations:

print(f"Migration {module_name} already applied,
skipping.")

continue



try:

print(f"Applying migration: {module_name}")

migration_module =
importlib.import_module(f'migrations.{module_name}')

migration_module.migrate()

# Record the migration as applied

record_migration(module_name)

print(f"Migration {module_name} applied successfully!")

except Exception as e:

print(f"Error applying migration {module_name}: {e}")

raise # Re-raise the exception for test cases




def show_migrations():
```

```
"""Display migration status - which are applied and which
are pending."""

applied_migrations = get_applied_migrations()


migrations_dir = Path('migrations')

if not migrations_dir.exists():

print("No migrations directory found.")

return


migration_files =
sorted(migrations_dir.glob('????_*.py'))

if not migration_files:

print("No migration files found.")

return


print("\nMigration status:")

print("-" * 50)

for migration_file in migration_files:

name = migration_file.stem

status = "[X]" if name in applied_migrations else "[ ]"

print(f"{status} {name}")
```

```
print("-" * 50)



def main():
"""
Parses command-line arguments and executes the
corresponding
migration command (generate, migrate, showmigrations).
"""
parser = argparse.ArgumentParser(
description='ORM CLI for database management')
parser.add_argument('command', choices=['generate',
'migrate', 'showmigrations'],
help='Command to execute (generate: create migrations,
migrate: apply migrations, showmigrations: list migration
status)')
parser.add_argument('--app', default='myapp',
help='Specific app folder to search for models (default:
myapp)')

args = parser.parse_args()
```

```
# Get the project root (one level up from the ORM
package)

project_root =
os.path.dirname(os.path.dirname(os.path.abspath(__file__)
))


if args.command == 'generate':

models = find_models(project_root, args.app)

if not models:

print(f"No models found in the '{args.app}' folder.")

return


print(

f"Found {len(models)} model(s) in '{args.app}': {',
'.join(model.__name__ for model in models)}")

generate_migrations(models)


elif args.command == 'migrate':

apply_migrations()


elif args.command == 'showmigrations':

show_migrations()
```

```
if __name__ == '__main__':
main()
```

#ORM/query.py

```
"""
Defines the QuerySet and Manager classes responsible for
building
and executing database queries based on model
interactions.
"""
import sqlite3
import re


DB_PATH = "databases/main.sqlite3"
REPR_OUTPUT_SIZE = 10
```

```
class QuerySet:

"""

A QuerySet represents a collection of database queries
for a specific model.

It allows you to build and execute SQL queries in a
Pythonic way, with support

for filtering, ordering, limiting, and offsetting
results.


Attributes:

model (class): The model class associated with this
QuerySet.

where_clause (str): The SQL WHERE clause for filtering
results.

parameters (list): The parameters to be used in the WHERE
clause.

order_clause (str): The SQL ORDER BY clause for sorting
results.

limit_val (int): The maximum number of results to return.

offset_val (int): The number of results to skip before
returning.


Methods:
```

filter(**conditions): Adds conditions to the WHERE clause to filter results.

get(**conditions): Returns a single record matching the conditions or raises an exception.

order_by(*fields): Specifies the order in which results should be returned.

limit(limit_val): Limits the number of results returned.

offset(offset_val): Specifies the number of results to skip.

all(): Executes the query and returns all results.

__iter__(): Allows iteration over the results.

__getitem__(index): Retrieves a specific result or slice of results.

Example Usage:

```
# Assuming `User` is a model class representing a database table.


# Get all users

users = QuerySet(User).all()


# Filter users by age and order by name
```

```
users = QuerySet(User).filter(age=25).order_by("name")


# Get a single user by ID

user = QuerySet(User).get(id=1)


# Get the first 10 users, skipping the first 5

users = QuerySet(User).limit(10).offset(5).all()

"""



def __init__(self, model, where_clause=None,
parameters=None, order_clause=None, limit_val=None,
offset_val=None):

"""

Initializes a new QuerySet instance.


Args:

model (class): The model class associated with this
QuerySet.

where_clause (str, optional): The SQL WHERE clause for
filtering results.

parameters (list, optional): The parameters to be used in
the WHERE clause.
```

```
order_clause (str, optional): The SQL ORDER BY clause for
sorting results.

limit_val (int, optional): The maximum number of results
to return.

offset_val (int, optional): The number of results to skip
before returning.

"""

self.model = model

self.where_clause = where_clause

self.parameters = parameters if parameters is not None
else []

self.order_clause = order_clause

self.limit_val = limit_val

self.offset_val = offset_val


def _fetch_for_repr(self):

"""Fetch a limited number of results for
representation."""

# Fetch one more than the limit to check if there are
more results

qs_limited = QuerySet(

model=self.model,
```

```
where_clause=self.where_clause,
parameters=self.parameters,
order_clause=self.order_clause,
limit_val=REPR_OUTPUT_SIZE + 1, # Fetch one extra
offset_val=self.offset_val
)
# _execute now returns instances
return qs_limited._execute()


def __repr__(self):
"""Return a string representation of the QuerySet,
showing limited results."""
# _fetch_for_repr now returns instances
data = self._fetch_for_repr()
# Check if there were more results than the limit
has_more = len(data) > REPR_OUTPUT_SIZE
# Slice data to the display limit
data_to_display = data[:REPR_OUTPUT_SIZE]


# Use the __repr__ of the model instances
```

```
items_repr = ",\n ".join(repr(item) for item in
data_to_display)


if has_more:

return f"<QuerySet [\n {items_repr},\n ...\n]>"

else:

return f"<QuerySet [\n {items_repr}\n]>"




def _build_query(self):

"""

Builds the SQL query based on the current state of the
QuerySet.


Returns:

str: The constructed SQL query.

"""

# Ensure SELECT * selects all necessary columns,
including foreign key IDs

# SELECT * should be fine as FKs are stored as *_id
columns.

query = f"SELECT * FROM {self.model.__name__.lower()}"
```

```
if self.where_clause:
query += " WHERE " + self.where_clause
if self.order_clause:
query += " ORDER BY " + self.order_clause
if self.limit_val is not None:
query += f" LIMIT {self.limit_val}"
if self.offset_val is not None:
if (query.__contains__("LIMIT")):
query += f" OFFSET {self.offset_val}"
else:
query += f" LIMIT -1 OFFSET {self.offset_val}"
return query


def _execute(self):
"""
Executes the constructed SQL query and returns the
results
as a list of model instances.
"""
query = self._build_query()
connection_obj = sqlite3.connect(DB_PATH)
```

```
connection_obj.execute("PRAGMA foreign_keys = ON;")

# Set row_factory to create dictionaries directly

connection_obj.row_factory = sqlite3.Row

cursor_obj = connection_obj.cursor()

cursor_obj.execute(query, tuple(self.parameters))


# Fetch rows as dictionaries

results_as_dicts = [dict(row) for row in
cursor_obj.fetchall()]

connection_obj.close()


# Convert dictionaries to model instances

instances = []

for row_dict in results_as_dicts:

instance = self.model(**row_dict)

for column_name, value in row_dict.items():

# Directly set the attribute on the instance.

# This handles 'id', regular fields, and 'fieldname_id'
columns.

setattr(instance, column_name, value)

instances.append(instance)
```

```
return instances

def sanitize_field_name(self, field_name):
"""
Sanitizes a field name to ensure it is a valid SQL
identifier.
Raises ValueError if the field name is invalid.
"""
if not re.match(r"^[a-zA-Z_][a-zA-Z0-9_]*$", field_name):
raise ValueError(f"Invalid field name: {field_name}")
return field_name

def filter(self, **conditions):
"""
Adds conditions to the WHERE clause to filter the
results.
Supports lookup operators like __exact, __like, __gt,
etc.

Args:
```

```
**conditions: Keyword arguments representing field
lookups and values.

Returns:
QuerySet: A new QuerySet instance with the combined
filter conditions.

Example:
.filter(name__like='John%', age__gt=21)
→ WHERE name LIKE 'John%' AND age > 21
"""
clauses = []
params = []

# Parse conditions and build SQL clauses
for key, value in conditions.items():
# Split field and lookup operator
parts = key.split('__', 1)
field = parts[0]
lookup = parts[1] if len(parts) > 1 else 'exact'
```

```
try:
field = self.sanitize_field_name(field)
except ValueError as e:
raise ValueError(
f"Invalid field name in condition: {key}") from e


# Handle different lookup types
if lookup == 'exact':
clause = f"{field} = ?"
elif lookup == 'like':
clause = f"{field} LIKE ?"
elif lookup == 'gt':
clause = f"{field} > ?"
elif lookup == 'gte':
clause = f"{field} >= ?"
elif lookup == 'lt':
clause = f"{field} < ?"
elif lookup == 'lte':
clause = f"{field} <= ?"
elif lookup == 'in':
```

```
placeholders = ', '.join(['?'] * len(value))
clause = f"{field} IN ({placeholders})"
params.extend(value)
elif lookup == 'neq':
clause = f"{field} != ?"
else:
raise ValueError(f"Invalid lookup operator: {lookup}")

# Add value to params (unless handled by IN clause)
if lookup != 'in':
params.append(value)

clauses.append(clause)

# Combine with existing WHERE clause
new_where = " AND ".join(clauses)
if self.where_clause:
new_where = f"({self.where_clause}) AND ({new_where})"

# Combine parameters
```

```
new_params = self.parameters + params

return QuerySet(
model=self.model,
where_clause=new_where,
parameters=new_params,
order_clause=self.order_clause,
limit_val=self.limit_val,
offset_val=self.offset_val
)

def get(self, **conditions):
"""
Returns a single model instance matching the specified
conditions.

Args:
**conditions: Keyword arguments representing the field
names and values to filter by.

Returns:
```

```
BaseModel: A model instance representing the matching record.


Raises:
Exception: If no matching record is found or if multiple records are found.
"""
# Limit to 2 to detect multiple objects
qs = self.filter(**conditions).limit(2)
# _execute now returns instances
results = qs._execute()
if len(results) == 0:
raise Exception(f"{self.model.__name__}.DoesNotExist: No matching record found for {conditions}.")
elif len(results) > 1:
raise Exception(
f"{self.model.__name__}.MultipleObjectsReturned: More than one record found for {conditions}.")
# Return the single instance
return results[0]


def order_by(self, *fields):
```

```
"""
Specifies the order in which results should be returned.

Args:
*fields: Field names to order by. Prefix a field with "-"
to sort in descending order.

Returns:
QuerySet: A new QuerySet instance with the specified
order.
"""
order_clause = []
for field in fields:
if field.startswith("-"):
order_clause.append(f"{field[1:]} DESC")
else:
order_clause.append(f"{field} ASC")
return QuerySet(self.model, self.where_clause,
self.parameters, ", ".join(order_clause), self.limit_val,
self.offset_val)

def limit(self, limit_val):
```

```
"""
Limits the number of results returned.


Args:
limit_val (int): The maximum number of results to return.


Returns:
QuerySet: A new QuerySet instance with the specified
limit.
"""
return QuerySet(self.model, self.where_clause,
self.parameters, self.order_clause, limit_val,
self.offset_val)


def offset(self, offset_val):
"""
Specifies the number of results to skip before returning.


Args:
offset_val (int): The number of results to skip.
```

```
Returns:

QuerySet: A new QuerySet instance with the specified
offset.

"""

return QuerySet(self.model, self.where_clause,
self.parameters, self.order_clause, self.limit_val,
offset_val)


def all(self):

"""

Executes the query and returns all results as a list of
model instances.

"""

# _execute now returns instances

return self._execute()


def __iter__(self):

"""

Allows iteration over the results (model instances).

"""

# _execute now returns instances

return iter(self._execute())
```

```
def __getitem__(self, index):
"""
Retrieves a specific result (model instance) or slice of
results (list of instances).
"""
if isinstance(index, slice):
offset = index.start if index.start is not None else 0
limit_val = index.stop - offset if index.stop is not None
else None
# Create a new QuerySet for the slice
qs = QuerySet(self.model, self.where_clause,
self.parameters, self.order_clause, limit_val, offset)
# _execute returns instances
return qs._execute()
elif isinstance(index, int):
# Create a new QuerySet for the single item
qs = QuerySet(self.model, self.where_clause,
self.parameters, self.order_clause, 1, index)
# _execute returns a list of instances
result = qs._execute()
```

```
if result:
# Return the single instance
return result[0]
raise IndexError("Index out of range")
else:
raise TypeError("Invalid argument type.")



class Manager:
"""
Provides the entry point for accessing QuerySet methods
on a model class.
Acts as a descriptor to associate the Manager with a
specific model.
Delegates attribute access to a new QuerySet instance for
the model.
"""
def __get__(self, instance, owner):
"""Descriptor __get__ method. Returns the Manager
instance itself."""
self.model = owner
return self
```

```
def __getattr__(self, attr):

"""Delegates attribute access to a new QuerySet for the
associated model."""

return getattr(QuerySet(self.model), attr)


def __getitem__(self, index):

"""Allows slicing/indexing directly on the manager (e.g.,
Model.objects[0])."""

return QuerySet(self.model)[index]


def __iter__(self):

"""Allows iterating directly on the manager (e.g., for
user in User.objects)."""

return QuerySet(self.model).__iter__()
```

#tests/run_tests.py

```
import unittest


if __name__ == "__main__":
```

```
loader = unittest.TestLoader()

suite = loader.discover(start_dir=".",
pattern="test*.py")

runner = unittest.TextTestRunner(verbosity=2)

runner.run(suite)
```

#tests/test_stress.py

```
from ORM.fields import ForeignKey

from ORM.datatypes import CharField, IntegerField

from ORM.base import BaseModel, DB_PATH

from faker import Faker

import unittest
```

```
import time

import random

import os


import sys

sys.path.append(os.path.dirname(os.path.dirname(os.path.a
bspath(__file__))))



# Initialize Faker

fake = Faker()


# Define some models for stress testing




class StressAuthor(BaseModel):

name = CharField(max_length=255)

email = CharField(unique=True)




class StressPost(BaseModel):
```

```
title = CharField(max_length=255)

content = CharField() # Assuming CharField can be large
like TEXT

author = ForeignKey(to=StressAuthor)

views = IntegerField(default=0)



class TestORMStress(unittest.TestCase):

NUM_AUTHORS = 1000

NUM_POSTS_PER_AUTHOR = 5 # Creates NUM_AUTHORS *
NUM_POSTS_PER_AUTHOR posts


@classmethod

def setUpClass(cls):

"""Set up database and tables before tests."""

if not os.path.exists('databases'):

os.makedirs('databases')


# Drop tables if they exist to ensure a clean slate,

# create_table() in the ORM might already do this.
```

```
# If not, manual drop might be needed or ensure
create_table handles it.

# For simplicity, relying on create_table's potential
DROP IF EXISTS logic.

StressAuthor.create_table()

StressPost.create_table()


@classmethod

def tearDownClass(cls):

"""Clean up the database after tests."""

if os.path.exists(DB_PATH):

os.remove(DB_PATH)

if os.path.exists('databases') and not
os.listdir('databases'):

os.rmdir('databases')


def test_bulk_insert_performance(self):

"""Test performance of inserting a large number of
records."""

authors_data = []

for _ in range(self.NUM_AUTHORS):

authors_data.append({
```

```
"name": fake.name(),
"email": fake.unique.email()
})

start_time = time.time()
StressAuthor.insert_entries(authors_data)
end_time = time.time()
print(
f"\nInserted {self.NUM_AUTHORS} authors in {end_time -
start_time:.4f} seconds.")

# Retrieve inserted authors to get their IDs for posts
# Assuming insert_entries doesn't update IDs on the input
dicts,
# and we need to fetch them. If it updates model
instances, that's better.
# For this example, let's assume we can fetch them or
insert_entries handles it.
# If insert_entries updates instances, it's better to
create instances first.

# Let's try with instances for posts to get IDs easily
```

```
inserted_authors = StressAuthor.objects.all() # Fetch all
authors

self.assertEqual(len(inserted_authors), self.NUM_AUTHORS)


posts_data = []

for author_instance in inserted_authors:

for _ in range(self.NUM_POSTS_PER_AUTHOR):

posts_data.append(StressPost(

title=fake.sentence(nb_words=6),

content=fake.text(max_nb_chars=1000),

author=author_instance, # Pass the instance

views=random.randint(0, 10000)

))


start_time_posts = time.time()

StressPost.insert_entries(posts_data)

end_time_posts = time.time()

total_posts = self.NUM_AUTHORS *
self.NUM_POSTS_PER_AUTHOR

print(
```

```
f"Inserted {total_posts} posts in {end_time_posts -
start_time_posts:.4f} seconds.")


# Verification
self.assertEqual(len(StressAuthor.objects.all()),
self.NUM_AUTHORS)
self.assertEqual(len(StressPost.objects.all()),
total_posts)


# Example of a simple query
start_query_time = time.time()
# Fetch posts by a random author
if inserted_authors:
random_author = random.choice(inserted_authors)
# Ensure random_author.id is populated correctly by your
ORM after fetch or insert
if random_author.id:
authors_posts = StressPost.objects.filter(
author_id=random_author.id).all()
# print(f"Found {len(authors_posts)} posts for author ID
{random_author.id}")
# Should be NUM_POSTS_PER_AUTHOR if IDs are correct
```

```
self.assertGreaterEqual(len(authors_posts), 0)
else:
print(
"Warning: random_author.id is not populated, skipping a
query test.")

end_query_time = time.time()
print(
f"Sample query took {end_query_time -
start_query_time:.4f} seconds.")

def test_bulk_update_performance(self):
"""Test performance of updating a large number of
records."""
authors = StressAuthor.objects.all()
if not authors:
self.skipTest(
"No authors found to update. Run insert test first or add
setup here.")

updated_data_authors = []
# Prepare instances for update
```

```
authors_to_process_count = len(authors) // 2
# Ensure at least one if possible
if authors_to_process_count == 0 and len(authors) > 0:
authors_to_process_count = 1

if authors_to_process_count == 0:
self.skipTest("Not enough authors to update.")

for author in authors[:authors_to_process_count]:
author.name = fake.name() + " (Updated)"
updated_data_authors.append(author)

start_time = time.time()
for author_instance in updated_data_authors:
conditions = {'id': author_instance.id}
new_values = {'name': author_instance.name}
StressAuthor.replace_entries(conditions, new_values)
end_time = time.time()
# The replace_entries method in your ORM prints its own
messages.
print(
```

```
f"Processed {len(updated_data_authors)} author updates in
{end_time - start_time:.4f} seconds.")


# Verify an update

if updated_data_authors:

updated_author_check = StressAuthor.objects.get(

id=updated_data_authors[0].id)

self.assertTrue("(Updated)" in updated_author_check.name)


posts = StressPost.objects.all()

if not posts:

self.skipTest(

"No posts found to update. Run insert test first or add
setup here.")


updated_data_posts = []

posts_to_process_count = len(posts) // 2

# Ensure at least one if possible

if posts_to_process_count == 0 and len(posts) > 0:

posts_to_process_count = 1
```

```
if posts_to_process_count == 0:

self.skipTest("Not enough posts to update.")


for post in posts[:posts_to_process_count]:

post.views += 100

updated_data_posts.append(post)


start_time_posts = time.time()

for post_instance in updated_data_posts:

conditions = {'id': post_instance.id}

new_values = {'views': post_instance.views}

StressPost.replace_entries(conditions, new_values)

end_time_posts = time.time()

print(

f"Processed {len(updated_data_posts)} post updates
(views) in {end_time_posts - start_time_posts:.4f}
seconds.")


if updated_data_posts:

updated_post_check = StressPost.objects.get(

id=updated_data_posts[0].id)
```

```
self.assertEqual(updated_post_check.views,
updated_data_posts[0].views)


def test_complex_queries_performance(self):
"""Test performance of more complex queries."""
# Ensure data exists
if not StressAuthor.objects.all() or not
StressPost.objects.all():
self.skipTest(
"No data found for complex queries. Run insert test
first.")


start_time = time.time()
# Example: Get all posts from authors whose names start
with 'A' (case-insensitive)
# This depends on your ORM's filter capabilities (e.g.,
__startswith, __icontains)
# For simplicity, fetching all and filtering in Python if
ORM is basic:
authors_with_A = [author for author in
StressAuthor.objects.all(
) if author.name.lower().startswith('a')]
```

```
author_ids_with_A = [author.id for author in
authors_with_A]


if author_ids_with_A:

# Assuming your ORM supports an `__in` lookup or similar
for filter

# posts_from_authors_with_A =
StressPost.objects.filter(author_id__in=author_ids_with_A
).all()

# If not, you might need to iterate or fetch all and
filter

all_posts = StressPost.objects.all()

posts_from_authors_with_A = [

post for post in all_posts if post.author_id in
author_ids_with_A]

print(

f"Fetched {len(posts_from_authors_with_A)} posts from
authors whose names start with 'A'.")

else:

print("No authors found with names starting with 'A' for
complex query test.")


end_time = time.time()
```

```
print(
f"Complex query (authors with 'A', then their posts) took
{end_time - start_time:.4f} seconds.")


# Example: Posts with more than 5000 views
start_time_views = time.time()
# Assuming your ORM supports __gt or similar for filter
# high_view_posts =
StressPost.objects.filter(views__gt=5000).all()
# If not, fetch all and filter:
all_posts_for_views = StressPost.objects.all()
high_view_posts = [
post for post in all_posts_for_views if post.views >
5000]
end_time_views = time.time()
print(
f"Fetched {len(high_view_posts)} posts with >5000 views
in {end_time_views - start_time_views:.4f} seconds.")
self.assertGreaterEqual(len(high_view_posts), 0)


def test_bulk_delete_performance(self):
```

```
"""Test performance of deleting a large number of
records."""
initial_authors = StressAuthor.objects.all()
if not initial_authors:
self.skipTest("No authors found to delete. Run insert
test first.")


num_authors_to_delete = len(initial_authors) // 2
if num_authors_to_delete == 0 and len(initial_authors) >
0:
num_authors_to_delete = 1 # Delete at least one if
present


if num_authors_to_delete == 0:
self.skipTest(
"Not enough authors to perform delete test (less than
1).")


authors_to_delete_instances =
initial_authors[:num_authors_to_delete]
authors_to_delete_ids = [
author.id for author in authors_to_delete_instances]
```

```
if not authors_to_delete_ids:
self.skipTest("No author IDs selected for deletion.")

# Manually delete posts of these authors first.
# Your ORM's delete_entries enables foreign key checks.
# If StressPost.author ForeignKey doesn't have ON DELETE
CASCADE,
# deleting an author with posts will fail.
print(
f"\nDeleting posts for {len(authors_to_delete_ids)}
authors before deleting authors...")
for author_id in authors_to_delete_ids:
StressPost.delete_entries(conditions={'author_id':
author_id})
print(f"Finished deleting posts for selected authors.")

start_time = time.time()
for author_id in authors_to_delete_ids:
StressAuthor.delete_entries(conditions={'id': author_id})
end_time = time.time()
```

```
# delete_entries prints its own messages, this is a
summary.

print(

f"Attempted to delete {len(authors_to_delete_ids)}
authors individually in {end_time - start_time:.4f}
seconds.")


remaining_authors_count = len(StressAuthor.objects.all())

expected_remaining_authors = len(

initial_authors) - len(authors_to_delete_ids)

self.assertEqual(remaining_authors_count,
expected_remaining_authors)




if __name__ == '__main__':

unittest.main()
```

#tests/testBasic.py

```
import sys

import os

import unittest

import sqlite3
```

```
from unittest.mock import patch, MagicMock # Add mock

sys.path.append(os.path.dirname(os.path.dirname(os.path.a
bspath(__file__))))

from ORM import base, datatypes
from ORM.fields import ForeignKey # Add ForeignKey

DB_PATH = "databases/main.sqlite3"

# Add a simple related model for FK tests
class Department(base.BaseModel):
name = datatypes.CharField()

class Student(base.BaseModel):
name = datatypes.CharField(unique=True) # Add unique
constraint for testing errors
degree = datatypes.CharField(null=False) # Add NOT NULL
constraint for testing errors
department = ForeignKey(to=Department, null=True) # Add
FK for testing
```

```
class TestCreateTable(unittest.TestCase):

"""

A test case class to verify the creation and schema of
the 'student' table in the SQLite database.


This class contains methods to test the following:

1. Whether the 'student' table exists in the database
after creation.

2. Whether the schema of the 'student' table matches the
expected schema, including column names and data types.

3. Whether the table is correctly populated with initial
data entries.


The `setUpClass` method is used to initialize the
database and create the 'student' table before any tests
are run.

The `tearDownClass` method is used to clean up the
database after all tests have been executed.


Methods:

- `test_table_exists`: Verifies that the 'student' table
exists in the database.
```

```
- `test_table_schema`: Verifies that the schema of the
'student' table matches the expected schema.

- `test_populate_schema`: Verifies that the table is
correctly populated with the expected initial data
entries.

"""

@classmethod

def setUpClass(cls):

"""Set up the database and create the table once before
all tests."""

if not os.path.exists('databases'):

os.makedirs('databases')

# Create tables for all models used in this test file

Department.create_table()

Student.create_table()


def setUp(self):

"""Insert fresh data and reset sequence before each
test."""

# Delete from all tables used

Student.delete_entries({}, confirm_delete_all=True)

Department.delete_entries({}, confirm_delete_all=True)
```

```
connection = sqlite3.connect(DB_PATH)

cursor = connection.cursor()

try:

# Reset sequences for all tables

cursor.execute("DELETE FROM sqlite_sequence WHERE name IN
(?, ?);",

(Student.__name__.lower(), Department.__name__.lower()))

connection.commit()

except sqlite3.OperationalError as e:

print(f"Info: Could not reset sequences - {e}")

connection.rollback()

finally:

connection.close()


# Insert base data

self.dept1 = Department(name="Science")

Department.insert_entries([self.dept1])


self.student1 = Student(name="Yehor Boiar",
degree="Computer Science", department=self.dept1)
```

```
self.student2 = Student(name="Anastasia Martison",
degree="Computer Science", department=self.dept1)

Student.insert_entries([self.student1, self.student2]) #
Insert instances


def test_table_exists(self):

"""Test if the table was created in the database."""

connection = sqlite3.connect(DB_PATH)

cursor = connection.cursor()


# Check if the table exists

cursor.execute("SELECT name FROM sqlite_master WHERE
type='table' AND name='student';")

table_exists = cursor.fetchone()

self.assertIsNotNone(table_exists, "Table 'student' was
not created.")


connection.close()


def test_table_schema(self):

"""Test if the table schema matches the expected
schema."""
```

```
connection = sqlite3.connect(DB_PATH)
cursor = connection.cursor()

# Check the schema of the table
cursor.execute("PRAGMA table_info(student);")
columns = cursor.fetchall()

# Expected schema
expected_columns = [
(0, 'id', 'INTEGER', 1, None, 1), # Primary key
(1, 'name', 'TEXT', 0, None, 0),
(2, 'degree', 'TEXT', 1, None, 0),
(3, 'department_id', 'INTEGER', 0, None, 0)
]
self.assertEqual(columns, expected_columns, "Table schema
does not match expected schema.")

connection.close()

def test_populate_schema(self):
# This test now verifies the data inserted by setUp
```

```
connection = sqlite3.connect(DB_PATH)

cursor = connection.cursor()

cursor.execute("SELECT id, name, degree FROM student
ORDER BY id") # Order by ID for consistency

students = cursor.fetchall()

# Adjust expected IDs if delete/insert changes auto-
increment behavior across tests

# Assuming fresh inserts start from 1 each time due to
delete_entries in setUp

expected_entries = [(1, 'Yehor Boiar', 'Computer
Science'), (2, 'Anastasia Martison', 'Computer Science')]

self.assertEqual(students, expected_entries, "Data
inserted in setUp does not match expected.")

connection.close()


def test_slicing(self):

# Fetch instances

student0 = Student.objects[0]

students_slice = Student.objects[:2]


# Assert instance attributes for single slice item

self.assertIsInstance(student0, Student)
```

```
self.assertEqual(student0.id, 1)
self.assertEqual(student0.name, 'Yehor Boiar')
self.assertEqual(student0.degree, 'Computer Science')

# Assert instance attributes for slice range
self.assertEqual(len(students_slice), 2)
self.assertIsInstance(students_slice[0], Student)
self.assertIsInstance(students_slice[1], Student)
self.assertEqual(students_slice[0].id, 1)
self.assertEqual(students_slice[1].id, 2)
self.assertEqual(students_slice[0].name, 'Yehor Boiar')
self.assertEqual(students_slice[1].name, 'Anastasia
Martison')

def test_iter(self):
# Expected instances (or check attributes)
expected_students = [self.student1, self.student2]
fetched_students = list(Student.objects.__iter__()) #
Collect iterator results
```

```
self.assertEqual(len(fetched_students),
len(expected_students))

# Sort by ID for consistent comparison if order isn't
guaranteed by default iteration

fetched_students.sort(key=lambda s: s.id)

expected_students.sort(key=lambda s: s.id)


for i, fetched in enumerate(fetched_students):

self.assertIsInstance(fetched, Student)

self.assertEqual(fetched.id, expected_students[i].id)

self.assertEqual(fetched.name, expected_students[i].name)

self.assertEqual(fetched.degree,
expected_students[i].degree)


def test_all(self):

# Expected instances

expected_students = [self.student1, self.student2]

all_students = Student.objects.all()


self.assertEqual(len(all_students),
len(expected_students))
```

```
# Sort by ID for consistent comparison if order isn't
guaranteed

all_students.sort(key=lambda s: s.id)

expected_students.sort(key=lambda s: s.id)


for i, fetched in enumerate(all_students):

self.assertIsInstance(fetched, Student)

self.assertEqual(fetched.id, expected_students[i].id)

self.assertEqual(fetched.name, expected_students[i].name)

self.assertEqual(fetched.degree,
expected_students[i].degree)


def test_filter(self):

# Expected instance(s)

expected_student = self.student1

result = Student.objects.filter(name="Yehor Boiar").all()


self.assertEqual(len(result), 1, "Filter should return
one result")

self.assertIsInstance(result[0], Student)

self.assertEqual(result[0].id, expected_student.id)
```

```
self.assertEqual(result[0].name, expected_student.name)
self.assertEqual(result[0].degree,
expected_student.degree)


def test_get(self):
# Expected instance
expected_student = self.student2
result = Student.objects.get(id=2)


self.assertIsInstance(result, Student)
self.assertEqual(result.id, expected_student.id)
self.assertEqual(result.name, expected_student.name)
self.assertEqual(result.degree, expected_student.degree)


def test_get_no_match(self):
with self.assertRaises(Exception) as context:
Student.objects.get(id=999) # No such ID
self.assertIn("DoesNotExist", str(context.exception))


def test_get_multiple_results(self):
with self.assertRaises(Exception) as context:
```

```
Student.objects.get(degree="Computer Science") # Multiple
students have this degree

self.assertIn("MultipleObjectsReturned",
str(context.exception))


def test_order_by(self):

# Expected instances in specific order

expected_students_ordered = [self.student2,
self.student1] # Ordered by -id

result = Student.objects.order_by("-id").all()


self.assertEqual(len(result), 2)

self.assertIsInstance(result[0], Student)

self.assertIsInstance(result[1], Student)

self.assertEqual(result[0].id,
expected_students_ordered[0].id)

self.assertEqual(result[1].id,
expected_students_ordered[1].id)

self.assertEqual(result[0].name,
expected_students_ordered[0].name)

self.assertEqual(result[1].name,
expected_students_ordered[1].name)
```

```
def test_limit(self):

# Expected instance

expected_student = self.student1

result = Student.objects.limit(1).all() # Should get
student with id 1 by default ordering


self.assertEqual(len(result), 1)

self.assertIsInstance(result[0], Student)

self.assertEqual(result[0].id, expected_student.id)

self.assertEqual(result[0].name, expected_student.name)


def test_offset(self):

# Expected instance

expected_student = self.student2

result = Student.objects.offset(1).all() # Skip student
1, get student 2


self.assertEqual(len(result), 1)

self.assertIsInstance(result[0], Student)

self.assertEqual(result[0].id, expected_student.id)

self.assertEqual(result[0].name, expected_student.name)
```

```
def test_chained_operations(self):
# Expected instance
expected_student = self.student1
# Filter, Order by -id (student2, student1), limit 1
(student2), offset 1 (student1)
result = Student.objects.filter(degree="Computer
Science").order_by("-id").limit(1).offset(1).all()

self.assertEqual(len(result), 1)
self.assertIsInstance(result[0], Student)
self.assertEqual(result[0].id, expected_student.id)
self.assertEqual(result[0].name, expected_student.name)

def test_complex_filter(self):
# Expected instance
expected_student = self.student1
result = Student.objects.filter(degree="Computer
Science", name__like="Y%").all()

self.assertEqual(len(result), 1)
```

```
self.assertIsInstance(result[0], Student)
self.assertEqual(result[0].id, expected_student.id)
self.assertEqual(result[0].name, expected_student.name)
def test_limit_zero(self):
result = Student.objects.limit(0).all()
self.assertEqual(result, [])


def test_large_offset(self):
result = Student.objects.offset(100).all()
self.assertEqual(result, [])
def test_sql_injection_safety(self):
"""
Test that SQL injection attempts are safely handled.
"""
# Attempt to inject SQL via a field value
results = Student.objects.filter(name__exact="'; DROP
TABLE student; --").all()
self.assertEqual(len(results), 0) # Ensure no results are
returned


# Attempt to inject SQL via a field name
```

```
with self.assertRaises(ValueError):

Student.objects.filter(**{"invalid_field; DROP TABLE
student; --": "value"}).all()


# Attempt to use an invalid lookup operator

with self.assertRaises(ValueError):

Student.objects.filter(name__invalid_lookup="value").all(
)

def test_insert_model_instances(self):

"""Test inserting data using BaseModel instances and ID
update."""

Student.delete_entries({}, confirm_delete_all=True)


student1 = Student(name="Instance User1",
degree="Physics")

student2 = Student(name="Instance User2",
degree="Chemistry")

self.assertIsNone(student1.id) # ID should be None before
insert

self.assertIsNone(student2.id)


Student.insert_entries([student1, student2])
```

```
# Verify IDs were updated on instances
self.assertIsNotNone(student1.id)
self.assertIsNotNone(student2.id)
self.assertIsInstance(student1.id, int)
self.assertIsInstance(student2.id, int)

# Verify insertion in DB by fetching instances
fetched1 = Student.objects.get(id=student1.id)
fetched2 = Student.objects.get(id=student2.id)
self.assertIsInstance(fetched1, Student)
self.assertIsInstance(fetched2, Student)
self.assertEqual(fetched1.name, "Instance User1")
self.assertEqual(fetched2.name, "Instance User2")
self.assertEqual(fetched1.degree, "Physics")
self.assertEqual(fetched2.degree, "Chemistry")

def test_insert_mixed_types_raises_error(self):
"""Test that inserting a mix of dicts and instances
raises TypeError."""
```

```
# setUp inserted data, delete it for this specific test
scenario

Student.delete_entries({}, confirm_delete_all=True) #
Clean slate

student_instance = Student(name="Test Instance",
degree="Biology")

student_dict = {"name": "Test Dict", "degree": "Geology"}


with self.assertRaisesRegex(TypeError, "All entries must
be dictionaries."):

# The error message depends on which type is first in the
list

# If dict is first, it expects all dicts.

Student.insert_entries([student_dict, student_instance])


with self.assertRaisesRegex(TypeError, "All entries must
be BaseModel instances."):

# If instance is first, it expects all instances.

Student.insert_entries([student_instance, student_dict])


def test_insert_wrong_instance_type_raises_error(self):

"""Test that inserting instances of a different model
raises TypeError."""
```

```
# Define another simple model for testing purposes

class Course(base.BaseModel):

title = datatypes.CharField()

# Ensure Course table exists if it doesn't (idempotent)

Course.create_table()


# setUp inserted Student data, delete it for this
specific test scenario

Student.delete_entries({}, confirm_delete_all=True) #
Clean slate for Student table

wrong_instance = Course(title="Introduction to Testing")

student_instance = Student(name="Correct Student",
degree="Testing")


with self.assertRaisesRegex(TypeError, f"All entries must
be instances of {Student.__name__}"):

Student.insert_entries([student_instance,
wrong_instance])


# Clean up the Course table (optional, but good practice)

if os.path.exists(DB_PATH):

connection = sqlite3.connect(DB_PATH)
```

```
cursor = connection.cursor()

cursor.execute("DROP TABLE IF EXISTS course")

connection.commit()

connection.close()


def test_init_unexpected_kwargs(self):

"""Test initializing with unexpected keyword arguments"""

# Capture stdout/stderr to check for warning? Or just
ensure it doesn't crash.

# For now, just ensure it runs and the unexpected kwarg
is ignored.

student = Student(name="Test", degree="Test Degree",
non_existent_field="ignore_me")

self.assertEqual(student.name, "Test")

self.assertFalse(hasattr(student, "non_existent_field"))


def test_init_missing_fields(self):

"""Test initializing with missing fields defaults them to
None"""

# Student has 'name', 'degree', 'department'

student = Student(name="Only Name") # Missing degree (NOT
NULL) and department (NULL)
```

```
self.assertEqual(student.name, "Only Name")

# Check that attributes exist and are None initially

self.assertTrue(hasattr(student, 'degree'))

self.assertIsNone(getattr(student, 'degree', 'Attribute
missing')) # Default to None

self.assertTrue(hasattr(student, 'department'))

self.assertIsNone(getattr(student, 'department',
'Attribute missing')) # Default to None


def test_as_dict_fk_none(self):

"""Test as_dict when a ForeignKey field is None"""

student_no_dept = Student(name="No Dept", degree="Some
Degree", department=None)

Student.insert_entries([student_no_dept])

student_dict = student_no_dept.as_dict()

expected = {

'id': student_no_dept.id,

'name': "No Dept",

'degree': "Some Degree",

'department_id': None # Expect department_id to be None

}
```

```
self.assertDictEqual(student_dict, expected)


def test_insert_empty_list(self):

"""Test insert_entries with an empty list"""

# Should execute without error and print "No entries..."

# We can't easily capture print output in unittest
without extra libraries/setup

# So we just check it doesn't raise an error.

try:

Student.insert_entries([])

except Exception as e:

self.fail(f"insert_entries([]) raised an exception: {e}")


def test_insert_invalid_type_list(self):

"""Test insert_entries with list of invalid types (line
172)."""

with self.assertRaisesRegex(TypeError, "Entries must be a
list of dictionaries or BaseModel instances"):

Student.insert_entries([1, 2, 3])


def test_insert_constraint_violation_unique(self):
```

```
"""Test insert_entries violating UNIQUE constraint (line
220)."""

# self.student1 (Yehor Boiar) already exists from setUp

student_duplicate = Student(name="Yehor Boiar",
degree="Another Degree")

# This should raise an IntegrityError during
_execute_insert

with self.assertRaises(sqlite3.IntegrityError):

Student.insert_entries([student_duplicate])


def test_insert_constraint_violation_not_null(self):

"""Test insert_entries violating NOT NULL constraint
(line 220)."""

student_null_degree = Student(name="Null Degree Test",
degree=None) # degree is NOT NULL

# This should raise an IntegrityError during
_execute_insert

with self.assertRaises(sqlite3.IntegrityError):

Student.insert_entries([student_null_degree])


@patch('sqlite3.connect')

def test_insert_connection_error(self, mock_connect):
```

```
"""Test insert_entries with a connection error (lines
246-248)."""

# Configure the mock connection to raise an error

mock_connect.side_effect =
sqlite3.OperationalError("Cannot connect")


student_new = Student(name="Connect Fail", degree="Test")

with self.assertRaises(sqlite3.OperationalError):

Student.insert_entries([student_new])

# Verify rollback wasn't attempted (since connection
failed) - tricky without more mocks


def test_replace_no_conditions(self):

"""Test replace_entries with no conditions (lines 288-
289)."""

# Should run without error and print "Error: You must
provide..."

try:

Student.replace_entries({}, {"degree": "Updated Degree"})

except Exception as e:

self.fail(f"replace_entries with no conditions raised an
exception: {e}")
```

```
def test_replace_no_values(self):

"""Test replace_entries with no new values (line 292)."""

# Should run without error and print "Error: No new
values..."

try:

Student.replace_entries({"id": self.student1.id}, {})

except Exception as e:

self.fail(f"replace_entries with no values raised an
exception: {e}")


def test_replace_basic(self):

"""Test basic functionality of replace_entries (covers
finally block lines 363-364)."""

student_id = self.student1.id

Student.replace_entries({"id": student_id}, {"degree":
"Updated CS"})

updated_student = Student.objects.get(id=student_id)

self.assertEqual(updated_student.degree, "Updated CS")


def test_replace_constraint_violation(self):
```

```
"""Test replace_entries violating a constraint (lines
354, 359-361, 404-405)."""

# Try updating student1's name to student2's name
(violates UNIQUE)

student1_id = self.student1.id

student2_name = self.student2.name

with self.assertRaises(sqlite3.IntegrityError):

Student.replace_entries({"id": student1_id}, {"name":
student2_name})


@classmethod

def tearDownClass(cls):

"""Clean up the database after tests."""

if os.path.exists(DB_PATH):

os.remove(DB_PATH)

if os.path.exists('databases'):

os.rmdir('databases')


if __name__ == '__main__':

unittest.main()
```

#tests/testConnections.py

```
from ORM.fields import ManyToManyRelatedManager
```

```
import sys

import os

import unittest

import sqlite3

from unittest.mock import patch


sys.path.append(os.path.dirname(os.path.dirname(os.path.a
bspath(__file__))))


from ORM import base, datatypes, fields


DB_PATH = "databases/main.sqlite3"


class Customers(base.BaseModel):

name = datatypes.CharField(unique=True)

age = datatypes.IntegerField()

class ContactInfo(base.BaseModel):

phone = datatypes.CharField()

city = datatypes.CharField()

customer = base.OneToOneField(Customers)
```

```
class Orders(base.BaseModel):

item = datatypes.CharField()

customer = base.ForeignKey(to=Customers)


class Author(base.BaseModel):

name = datatypes.CharField()


class Book(base.BaseModel):

title = datatypes.CharField()

authors = base.ManyToManyField(to=Author)



class TestOneToManyRelationship(unittest.TestCase):

@classmethod

def setUpClass(cls):

# Create tables

Customers.create_table()

Orders.create_table()

# Insert a customer

Customers.insert_entries([{"name": "Yehor", "age": 18}])
```

```
customer = Customers.objects.get(name="Yehor")

# Insert multiple orders for the customer

Orders.insert_entries([

{"item": "item1", "customer": customer.id},

{"item": "item2", "customer": customer.id},

{"item": "item3", "customer": customer.id},

{"item": "item4", "customer": customer.id}

])


def test_customer_orders(self):

# Fetch the customer

customer = Customers.objects.get(id=1)


# Fetch all orders for the customer

orders =
Orders.objects.filter(customer_id=customer.id).all()


# Assert that the customer has 4 orders

self.assertEqual(len(orders), 4)

self.assertEqual(orders[0].item, "item1")
```

```
self.assertEqual(orders[1].item, "item2")

self.assertEqual(orders[2].item, "item3")

self.assertEqual(orders[3].item, "item4")


def test_as_dict_foreign_key(self):

"""Test as_dict() for a model with a ForeignKey."""

# Fetch the customer and one of their orders

customer = Customers.objects.get(name="Yehor")

order =
Orders.objects.filter(customer_id=customer.id).all()[0]


# Get dict representation of the order

order_dict = order.as_dict()


# Expected dict for the order

expected_order_dict = {

'id': order.id,

'item': order.item, # e.g., "item1"

'customer_id': customer.id # Should contain the related
customer's ID

}
```

```
self.assertDictEqual(order_dict, expected_order_dict)


# Test as_dict on the customer (which has no outgoing
FK/O2O in this test)

customer_dict = customer.as_dict()

expected_customer_dict = {

'id': customer.id,

'name': "Yehor",

'age': 18

}

self.assertDictEqual(customer_dict,
expected_customer_dict)


@classmethod

def tearDownClass(cls):

"""Clean up the database after tests."""

if os.path.exists(DB_PATH):

os.remove(DB_PATH)

if os.path.exists('databases'):

os.rmdir('databases')
```

```
class
TestOneToOneRelationshipEdgeCases(unittest.TestCase):

def setUp(self):

# Ensure tables exist before deleting entries

Customers.create_table()

ContactInfo.create_table()


# Skip confirmation prompt in tests

Customers.delete_entries({}, confirm_delete_all=True)

ContactInfo.delete_entries({}, confirm_delete_all=True)


# Reset sequences

connection = sqlite3.connect(DB_PATH)

cursor = connection.cursor()

try:

cursor.execute("DELETE FROM sqlite_sequence WHERE name IN
(?, ?);",

(Customers.__name__.lower(),
ContactInfo.__name__.lower()))

connection.commit()
```

```
except sqlite3.OperationalError as e:

print(f"Info: Could not reset sequences - {e}")

connection.rollback()

finally:

connection.close()



# Insert test data

self.cust1 = Customers(name="Yehor", age=18)

self.cust2 = Customers(name="Alice", age=25)

self.cust3 = Customers(name="Bob", age=30)

Customers.insert_entries([self.cust1, self.cust2,
self.cust3])


# Use instances for FK assignment

self.contact1 = ContactInfo(phone="123-456-7890",
city="New York", customer=self.cust1)

self.contact2 = ContactInfo(phone="987-654-3210",
city="Los Angeles", customer=self.cust2)

ContactInfo.insert_entries([self.contact1,
self.contact2])
```

```
def test_multiple_customers_with_contact_info(self):
# Fetch all customers and their contact info
yehor = self.cust1
alice = self.cust2
bob = self.cust3

yehor_contact =
ContactInfo.objects.get(customer_id=yehor.id)
alice_contact =
ContactInfo.objects.get(customer_id=alice.id)

# Assert contact info matches
self.assertEqual(yehor_contact.phone, "123-456-7890")
self.assertEqual(alice_contact.city, "Los Angeles")

# Bob has no contact info
with self.assertRaises(Exception):
ContactInfo.objects.get(customer_id=bob["id"])

def test_customer_without_contact_info(self):
```

```
# Fetch Bob, who has no contact info

bob = self.cust3


# Attempt to fetch contact info for Bob

with self.assertRaises(Exception):

ContactInfo.objects.get(customer_id=bob["id"])


def test_contact_info_without_customer(self):

# Attempt to insert contact info without a valid customer

with self.assertRaises(Exception): # Replace with the
specific exception your ORM raises

ContactInfo.insert_entries([{"phone": "555-555-5555",
"city": "Chicago", "customer": 999}]) # Invalid customer
ID


def test_duplicate_contact_info_for_customer(self):

# Fetch Yehor

yehor = self.cust1

# Attempt to insert another contact info for Yehor

with self.assertRaises(Exception): # Replace with the
specific exception your ORM raises
```

```
ContactInfo.insert_entries([{"phone": "111-222-3333",
"city": "San Francisco", "customer": yehor["id"]}])


def test_updating_contact_info(self):

# Fetch Yehor and his contact info

yehor = self.cust1

yehor_contact =
ContactInfo.objects.get(customer_id=yehor.id)


# Update Yehor's contact info

ContactInfo.replace_entries({"id": yehor_contact.id},
{"phone": "999-999-9999", "city": "Boston"})


# Fetch updated contact info

updated_contact =
ContactInfo.objects.get(customer_id=yehor.id)

self.assertEqual(updated_contact.phone, "999-999-9999")

self.assertEqual(updated_contact.city, "Boston")


def
test_deleting_customer_cascades_to_contact_info(self):

# Fetch Alice and her contact info
```

```
alice = self.cust2

alice_contact =
ContactInfo.objects.get(customer_id=alice.id)


# Delete Alice

Customers.delete_entries({'id':alice.id})

# fix : passing a dictionnary


# Ensure Alice's contact info is also deleted

with self.assertRaises(Exception): # Replace with the
specific exception your ORM raises

ContactInfo.objects.get(id=alice_contact.id)


def test_as_dict_one_to_one(self):

"""Test as_dict() for models involved in a OneToOne
relationship."""

# Use instances from setUp

yehor = self.cust1

alice = self.cust2

bob = self.cust3 # Bob has no contact info
```

```
# Fetch contact info for Yehor
yehor_contact =
ContactInfo.objects.get(customer_id=yehor.id)

# Get dict representation of Yehor's contact info
contact_dict = yehor_contact.as_dict()
expected_contact_dict = {
'id': yehor_contact.id,
'phone': "123-456-7890",
'city': "New York",
'customer_id': yehor.id # Should contain the related
customer's ID
}
self.assertDictEqual(contact_dict, expected_contact_dict)

# Get dict representation of Yehor (Customer)
# Customer model itself doesn't define the O2O field
pointing *to* ContactInfo
# So its dict should only contain its own fields.
yehor_dict = yehor.as_dict()
expected_yehor_dict = {
```

```
'id': yehor.id,

'name': "Yehor",

'age': 18

}

self.assertDictEqual(yehor_dict, expected_yehor_dict)


# Get dict representation of Bob (no related ContactInfo)

bob_dict = bob.as_dict()

expected_bob_dict = {

'id': bob.id,

'name': "Bob",

'age': 30

}

self.assertDictEqual(bob_dict, expected_bob_dict)


def test_as_dict_unsaved_one_to_one(self):

"""Test as_dict() on unsaved instances with FK/O2O
fields."""

# Unsaved ContactInfo (customer_id should be None)

unsaved_contact = ContactInfo(phone="111-000",
city="Nowhere")
```

```
contact_dict = unsaved_contact.as_dict()

expected_contact_dict = {

'id': None,

'phone': "111-000",

'city': "Nowhere",

'customer_id': None # FK/O2O ID should be None if
instance/relation not set

}

self.assertDictEqual(contact_dict, expected_contact_dict)


# Unsaved ContactInfo with unsaved Customer assigned

unsaved_customer = Customers(name="Temp", age=1)

unsaved_contact_with_rel = ContactInfo(phone="222-000",
city="Somewhere", customer=unsaved_customer)

contact_dict_rel = unsaved_contact_with_rel.as_dict()

expected_contact_dict_rel = {

'id': None,

'phone': "222-000",

'city': "Somewhere",

'customer_id': None # Related customer has no ID yet

}
```

```
self.assertDictEqual(contact_dict_rel,
expected_contact_dict_rel)


def test_insert_onetoone_violation_in_batch(self):
"""Test O2O violation check within
_process_entries_for_values (line 210)."""
# Try inserting two ContactInfo entries for self.cust3 in
the same batch
contact_batch = [
ContactInfo(phone="111", city="CityA",
customer=self.cust3),
ContactInfo(phone="222", city="CityB",
customer=self.cust3) # Duplicate customer FK
]
with self.assertRaisesRegex(ValueError, "Duplicate entry
detected within the batch for OneToOne field 'customer'
with value 3 at index 1"):
ContactInfo.insert_entries(contact_batch)


@classmethod
def tearDownClass(cls):
"""Clean up the database after tests."""
if os.path.exists(DB_PATH):
```

```
os.remove(DB_PATH)

if os.path.exists('databases'):

os.rmdir('databases')


# Add test for M2M as_dict error

class TestM2MAsDictError(unittest.TestCase):

@classmethod

def setUpClass(cls):

if not os.path.exists('databases'):

os.makedirs('databases')

Author.create_table()

Book.create_table()


def setUp(self):

Author.delete_entries({}, confirm_delete_all=True)

Book.delete_entries({}, confirm_delete_all=True)

# Clear junction table

connection_obj = sqlite3.connect(DB_PATH)

cursor_obj = connection_obj.cursor()

try: cursor_obj.execute("DELETE FROM book_author")
```

```
except sqlite3.OperationalError: pass

try: cursor_obj.execute("DELETE FROM sqlite_sequence
WHERE name IN ('author', 'book')")

except sqlite3.OperationalError: pass

connection_obj.commit()

connection_obj.close()


@patch('ORM.fields.ManyToManyRelatedManager.all')

def test_as_dict_m2m_error(self, mock_m2m_all):

"""Test as_dict M2M error handling (lines 108-111)."""

# Setup data

author = Author(name="Test Author")

Author.insert_entries([author])

book = Book(title="Test Book")

Book.insert_entries([book])

book.authors.add(author) # Add relationship


# Configure mock to raise error when .all() is called
within as_dict

mock_m2m_all.side_effect = Exception("Simulated M2M fetch
error")
```

```
# Call as_dict and check output
book_dict = book.as_dict()

# Expect 'authors' to be an empty list due to error
handling
expected_dict = {
'id': book.id,
'title': "Test Book",
'authors': [] # Should default to empty list on error
}
self.assertDictEqual(book_dict, expected_dict)
# Optionally check if the warning was printed (requires
more setup)

@classmethod
def tearDownClass(cls):
if os.path.exists(DB_PATH):
os.remove(DB_PATH)
if os.path.exists('databases'):
try: os.rmdir('databases')
```

```
except OSError: pass


class TestFieldFeatures(unittest.TestCase):
"""Tests for basic Field class features like default
values."""
@classmethod
def setUpClass(cls):
# Define a simple model for testing field defaults
class DefaultModel(base.BaseModel):
name = datatypes.CharField(default="DefaultName")
value = datatypes.IntegerField(default=100)
nullable_int = datatypes.IntegerField(null=True,
default=None) # Test None default


cls.DefaultModel = DefaultModel
# No table creation needed as we only test instance
initialization


def test_field_default_init(self):
"""Test Field __init__ default handling (line 37 in
base.py)."""
```

```
# BaseModel.__init__ does NOT apply defaults from Field
definition

# It defaults to None if kwarg is missing.

instance = self.DefaultModel()

self.assertIsNone(instance.name, "CharField default
should not be applied by __init__")

self.assertIsNone(instance.value, "IntegerField default
should not be applied by __init__")

self.assertIsNone(instance.nullable_int,
"IntegerField(default=None) should be None")



# Test providing values overrides the None default from
__init__

instance_override =
self.DefaultModel(name="SpecificName", value=50,
nullable_int=5)

self.assertEqual(instance_override.name, "SpecificName")

self.assertEqual(instance_override.value, 50)

self.assertEqual(instance_override.nullable_int, 5)


def test_charfield_init(self):

"""Test CharField __init__ (line 68 in datatypes.py)."""
```

```
# Primarily testing instantiation works and attributes
are stored

field = datatypes.CharField(max_length=10, unique=True,
default="test", null=False)

self.assertEqual(field.max_length, 10)

self.assertTrue(field.unique)

self.assertEqual(field.default, "test")

self.assertFalse(field.null)

self.assertEqual(field.db_type, "TEXT")

# Note: max_length validation isn't typically done on
assignment in simple ORMs

# It's usually a database constraint.


def test_integerfield_init(self):

"""Test IntegerField __init__ (line 87 in
datatypes.py)."""

instance = datatypes.IntegerField(default=0, null=False,
unique=True)

self.assertEqual(instance.default, 0)

self.assertFalse(instance.null)

self.assertTrue(instance.unique)

self.assertEqual(instance.db_type, "INTEGER")
```

```
# No specific validation on assignment is implemented in
the Field base class


class TestForeignKeyFeatures(unittest.TestCase):

"""Tests specific ForeignKey features."""

@classmethod

def setUpClass(cls):

class Country(base.BaseModel):

name = datatypes.CharField(unique=True)

# Removed 'cities' CharField - reverse relations aren't
automatic


class City(base.BaseModel):

name = datatypes.CharField()

# ForeignKey definition

country = fields.ForeignKey(to=Country, null=False) #
Make it non-nullable for testing


cls.Country = Country

cls.City = City

if not os.path.exists('databases'):
```

```
os.makedirs('databases')

cls.Country.create_table()

cls.City.create_table()


def setUp(self):

# Clear tables before each test

self.City.delete_entries({}, confirm_delete_all=True) #
Delete dependent table first

self.Country.delete_entries({}, confirm_delete_all=True)


connection = sqlite3.connect(DB_PATH)

cursor = connection.cursor()

try:

cursor.execute("DELETE FROM sqlite_sequence WHERE name IN
(?, ?);",

(self.Country.__name__.lower(),
self.City.__name__.lower()))

connection.commit()

except sqlite3.OperationalError: pass # Ignore if
sequence table doesn't exist

finally: connection.close()
```

```
# Insert a country instance

self.country1 = self.Country(name="Testland")

self.Country.insert_entries([self.country1]) # ID gets
updated


def test_foreignkey_reverse_access_not_implemented(self):

"""Test that reverse ForeignKey access is not
automatically created."""

city1 = self.City(name="Capital", country=self.country1)

self.City.insert_entries([city1])


# Accessing 'city_set' (or similar) should fail as it's
not defined

self.assertFalse(hasattr(self.country1, 'city_set'))

self.assertFalse(hasattr(self.country1, 'cities')) #
Check original name too

with self.assertRaises(AttributeError):

_ = self.country1.city_set # Or whatever default name
might be expected


# To get related cities, query the City model directly
```

```
related_cities_qs =
self.City.objects.filter(country_id=self.country1.id)
related_cities = list(related_cities_qs)
self.assertEqual(len(related_cities), 1)
self.assertEqual(related_cities[0].name, "Capital")



@classmethod
def tearDownClass(cls):
if os.path.exists(DB_PATH):
os.remove(DB_PATH)
if os.path.exists('databases'):
try: os.rmdir('databases')
except OSError: pass



class TestManyToManyFieldFeatures(unittest.TestCase):
"""Tests specific ManyToManyField features."""
@classmethod
def setUpClass(cls):
class Tag(base.BaseModel):
```

```
name = datatypes.CharField(unique=True)

# No automatic reverse relation 'posts'


class Post(base.BaseModel):

title = datatypes.CharField()

tags = fields.ManyToManyField(to=Tag) # Use
fields.ManyToManyField


cls.Tag = Tag

cls.Post = Post

if not os.path.exists('databases'):

os.makedirs('databases')

cls.Tag.create_table()

cls.Post.create_table() # This should also create the
junction table


def setUp(self):

# Determine junction table name dynamically

m2m_field = self.Post._many_to_many['tags']

self.junction_table = m2m_field.through or
f"{self.Post.__name__.lower()}_{self.Tag.__name__.lower()
}"
```

```
# Clear tables and junction table

connection_obj = sqlite3.connect(DB_PATH)

cursor_obj = connection_obj.cursor()

try: cursor_obj.execute(f"DELETE FROM
{self.junction_table}")

except sqlite3.OperationalError: pass # Ignore if table
doesn't exist yet

connection_obj.commit() # Commit deletion before deleting
main tables


self.Post.delete_entries({}, confirm_delete_all=True) #
Delete Post first if Tag has FKs to it (it doesn't here)

self.Tag.delete_entries({}, confirm_delete_all=True)


try:

cursor_obj.execute("DELETE FROM sqlite_sequence WHERE
name IN (?, ?, ?);",

(self.Tag.__name__.lower(), self.Post.__name__.lower(),
self.junction_table)) # Include junction table if it has
own sequence

connection_obj.commit()
```

```
except sqlite3.OperationalError: pass # Ignore if
sequence table doesn't exist

finally: connection_obj.close()


# Insert base data

self.tag1 = self.Tag(name="Tech")

self.tag2 = self.Tag(name="News")

self.Tag.insert_entries([self.tag1, self.tag2])

self.post1 = self.Post(title="Post 1")

self.Post.insert_entries([self.post1])


def test_manytomanyfield_init_no_related_name(self):

"""Test ManyToManyField __init__ doesn't store
related_name."""

# The implementation in fields.py doesn't accept/store
related_name

m2m_field = fields.ManyToManyField(self.Tag) # No
related_name arg

self.assertFalse(hasattr(m2m_field, 'related_name'))


def test_manytomany_get_manager(self):

"""Test ManyToManyField __get__ returns manager."""
```

```
manager = self.post1.tags

self.assertIsInstance(manager, ManyToManyRelatedManager)

# Check manager attributes based on fields.py
implementation

self.assertIs(manager.instance, self.post1)

self.assertIsInstance(manager.field,
fields.ManyToManyField)

self.assertIs(manager.source_class, self.Post)

self.assertIs(manager.target_class, self.Tag)

self.assertEqual(manager.junction_table,
self.junction_table)


def test_manytomany_direct_assignment_possible(self):

"""Test that direct assignment to M2M field replaces the
manager (not recommended)."""

# The descriptor doesn't define __set__, so assignment
replaces the manager

original_manager = self.post1.tags

self.post1.tags = [self.tag1] # Assign a list

self.assertNotIsInstance(self.post1.tags,
ManyToManyRelatedManager)

self.assertEqual(self.post1.tags, [self.tag1])
```

```
# Restore manager for subsequent tests if needed (though
setUp handles it)
setattr(self.post1, '_tags_manager', original_manager)



def test_manytomany_reverse_access_not_implemented(self):
"""Test accessing reverse M2M relationship is not
automatic."""
self.post1.tags.add(self.tag1)
post2 = self.Post(title="Post 2")
self.Post.insert_entries([post2])
post2.tags.add(self.tag1)

# Accessing tag1.posts should fail
self.assertFalse(hasattr(self.tag1, 'posts'))
with self.assertRaises(AttributeError):
_ = self.tag1.posts

# To get related posts, query the Post model and filter
via the junction table
# This requires a more complex query or a helper method
not shown in the base ORM
```

```
# Simplest is to iterate through posts and check their
tags

related_posts = []

for post in self.Post.objects.all():

if self.tag1.id in [tag.id for tag in post.tags.all()]:

related_posts.append(post)


self.assertEqual(len(related_posts), 2)

post_titles = {p.title for p in related_posts}

self.assertEqual(post_titles, {"Post 1", "Post 2"})




def test_m2m_manager_add_invalid_type(self):

"""Test ManyToManyRelatedManager add() type validation
(line 193 in fields.py)."""

# The add method checks isinstance(target_obj,
self.target_class)

with self.assertRaisesRegex(TypeError, f"Can only add
'{self.Tag.__name__}' instances."):

self.post1.tags.add("not a tag instance")


@classmethod
```

```
def tearDownClass(cls):

if os.path.exists(DB_PATH):

os.remove(DB_PATH)

if os.path.exists('databases'):

try: os.rmdir('databases')

except OSError: pass


if __name__ == '__main__':

unittest.main()
```

#tests/testDatatypes.py

```
import unittest

import sys

import os


# Add the project root to the Python path

sys.path.append(os.path.dirname(os.path.dirname(os.path.a
bspath(__file__))))


from ORM.datatypes import Field, CharField, IntegerField,
DateTimeField
```

```
class TestFieldTypes(unittest.TestCase):


def test_field_get_db_type(self):

"""Test the get_db_type method of the base Field
class."""

# Default (null=True, unique=False)

field_default = Field("TEXT")

self.assertEqual(field_default.get_db_type(), "TEXT")


# Not Null (null=False, unique=False)

field_not_null = Field("INTEGER", null=False)

self.assertEqual(field_not_null.get_db_type(), "INTEGER
NOT NULL")


# Unique and Null (null=True, unique=True)

field_unique_null = Field("REAL", unique=True)

self.assertEqual(field_unique_null.get_db_type(), "REAL
UNIQUE")


# Unique and Not Null (null=False, unique=True)

field_unique_not_null = Field("BLOB", null=False,
unique=True)
```

```
self.assertEqual(field_unique_not_null.get_db_type(),
"BLOB NOT NULL UNIQUE") # This line fails


def test_char_field(self):
"""Test CharField initialization and db_type."""
# Default (null=True)
char_default = CharField()
self.assertEqual(char_default.db_type, "TEXT")
self.assertTrue(char_default.null)
self.assertEqual(char_default.get_db_type(), "TEXT")


# Not Null (null=False)
char_not_null = CharField(null=False)
self.assertEqual(char_not_null.db_type, "TEXT")
self.assertFalse(char_not_null.null)
self.assertEqual(char_not_null.get_db_type(), "TEXT NOT
NULL")


def test_integer_field(self):
"""Test IntegerField initialization and db_type including
default."""
```

```
# Default (null=True, default=0)

int_default = IntegerField()

self.assertEqual(int_default.db_type, "INTEGER")

self.assertTrue(int_default.null)

self.assertEqual(int_default.default, 0)

# Check get_db_type includes default

self.assertEqual(int_default.get_db_type(), "INTEGER
DEFAULT 0")


# Not Null with different default

int_not_null_default_5 = IntegerField(null=False,
default=5)

self.assertEqual(int_not_null_default_5.db_type,
"INTEGER")

self.assertFalse(int_not_null_default_5.null)

self.assertEqual(int_not_null_default_5.default, 5)

self.assertEqual(int_not_null_default_5.get_db_type(),
"INTEGER NOT NULL DEFAULT 5")


# Nullable without explicit default (should still have
default=0 from init)

int_nullable = IntegerField(null=True)
```

```
self.assertEqual(int_nullable.get_db_type(), "INTEGER
DEFAULT 0")


# Nullable with None default (should not add DEFAULT
clause)

# Note: The current implementation correctly omits
DEFAULT when the value is None.

int_none_default = IntegerField(null=True, default=None)

self.assertEqual(int_none_default.default, None)

# Assert that no DEFAULT clause is added when default is
None

self.assertEqual(int_none_default.get_db_type(),
"INTEGER") # Changed expected value


def test_datetime_field(self):

"""Test DateTimeField initialization and db_type."""

# Default (null=True)

dt_default = DateTimeField()

self.assertEqual(dt_default.db_type, "DATETIME")

self.assertTrue(dt_default.null)

self.assertEqual(dt_default.get_db_type(), "DATETIME")
```

```
# Not Null (null=False)

dt_not_null = DateTimeField(null=False)

self.assertEqual(dt_not_null.db_type, "DATETIME")

self.assertFalse(dt_not_null.null)

self.assertEqual(dt_not_null.get_db_type(), "DATETIME NOT
NULL")


if __name__ == '__main__':

unittest.main()
```

#tests/testM2M.py

```
import unittest

import sqlite3

import sys

import os


sys.path.append(os.path.dirname(os.path.dirname(os.path.a
bspath(__file__))))


from ORM import base, datatypes

# Import QuerySet to check return types if needed

from ORM.query import QuerySet


DB_PATH = "databases/main.sqlite3"


class Author(base.BaseModel):

name = datatypes.CharField()


class Book(base.BaseModel):

title = datatypes.CharField()
```

```
authors = base.ManyToManyField(to=Author)


class CustomBook(base.BaseModel):

title = datatypes.CharField()

authors = base.ManyToManyField(Author,
through="customjunction")




class TestManyToManyRelationships(unittest.TestCase):

@classmethod

def setUpClass(cls):

# Create tables only once

if not os.path.exists('databases'):

os.makedirs('databases')

Author.create_table()

Book.create_table()

CustomBook.create_table() # Ensure custom junction table
is created


def setUp(self):
```

```
"""Clean up tables and insert fresh base data using
instances before each test."""

connection_obj = sqlite3.connect(DB_PATH)

cursor_obj = connection_obj.cursor()

cursor_obj.execute("PRAGMA foreign_keys = ON;")

# Clear junction tables first

try: cursor_obj.execute("DELETE FROM book_author")

except sqlite3.OperationalError: pass # Ignore if table
doesn't exist yet

try: cursor_obj.execute("DELETE FROM customjunction")

except sqlite3.OperationalError: pass # Ignore if table
doesn't exist yet

# Clear main tables

cursor_obj.execute("DELETE FROM author")

cursor_obj.execute("DELETE FROM book")

cursor_obj.execute("DELETE FROM custombook")

# Reset sequences

try:

cursor_obj.execute("DELETE FROM sqlite_sequence WHERE
name IN ('author', 'book', 'custombook')")

except sqlite3.OperationalError: pass # Ignore if
sequence table doesn't exist
```

```
connection_obj.commit()

connection_obj.close()


# Reinsert base data using instances (IDs will be
updated)

self.rowling = Author(name="J.K. Rowling")

self.orwell = Author(name="George Orwell")

self.christie = Author(name="Agatha Christie")

Author.insert_entries([self.rowling, self.orwell,
self.christie])


self.harry_potter = Book(title="Harry Potter")

self.nineteen_eighty_four = Book(title="1984")

Book.insert_entries([self.harry_potter,
self.nineteen_eighty_four])


def test_add_m2m_relationship(self):

"""Test adding authors to a book using instance
manager."""

# Use instances from setUp

rowling = self.rowling

harry_potter = self.harry_potter
```

```
# Add J.K. Rowling to Harry Potter using the instance
manager

harry_potter.authors.add(rowling)


# Retrieve authors for Harry Potter (should return Author
instances via QuerySet)

harry_authors_qs = harry_potter.authors.all()

self.assertIsInstance(harry_authors_qs, QuerySet)

harry_authors = list(harry_authors_qs) # Execute QuerySet

self.assertEqual(len(harry_authors), 1)

self.assertIsInstance(harry_authors[0], Author)

self.assertEqual(harry_authors[0].name, "J.K. Rowling")

self.assertEqual(harry_authors[0].id, rowling.id)


def test_remove_m2m_relationship(self):

"""Test removing an author from a book using instance
manager."""

# Use instances from setUp

rowling = self.rowling

harry_potter = self.harry_potter
```

```
# Add and then remove J.K. Rowling from Harry Potter
harry_potter.authors.add(rowling)
harry_potter.authors.remove(rowling)

# Retrieve authors for Harry Potter
harry_authors = list(harry_potter.authors.all())
self.assertEqual(len(harry_authors), 0)

def test_m2m_relationship_uniqueness(self):
"""Test that the same relationship cannot be added twice
via manager."""
# Use instances from setUp
rowling = self.rowling
harry_potter = self.harry_potter

# Add J.K. Rowling to Harry Potter twice (second add
should be ignored)
harry_potter.authors.add(rowling)
harry_potter.authors.add(rowling) # Should be ignored due
to INSERT OR IGNORE
```

```
# Retrieve authors for Harry Potter

harry_authors = list(harry_potter.authors.all())

self.assertEqual(len(harry_authors), 1) # Should only
have one entry


def test_m2m_cascade_delete_source(self):

"""Test M2M relationships are deleted when source is
deleted."""

# Use instances from setUp

rowling = self.rowling

harry_potter = self.harry_potter

harry_potter.authors.add(rowling)

hp_id = harry_potter.id # Store ID before deleting


# Delete source record (Book instance)

Book.delete_entries({'id': hp_id}) # Pass condition dict


# Verify relationships are gone from junction table

connection_obj = sqlite3.connect(DB_PATH)

cursor_obj = connection_obj.cursor()
```

```
cursor_obj.execute("SELECT * FROM book_author WHERE
book_id = ?", (hp_id,))

self.assertEqual(len(cursor_obj.fetchall()), 0)

connection_obj.close()


def test_m2m_cascade_delete_target(self):

"""Test M2M relationships are deleted when target is
deleted."""

# Use instances from setUp

rowling = self.rowling

harry_potter = self.harry_potter

harry_potter.authors.add(rowling)

rowling_id = rowling.id # Store ID before deleting


# Delete target record (Author instance)

Author.delete_entries({'id': rowling_id}) # Pass
condition dict


# Verify relationships are gone from junction table

connection_obj = sqlite3.connect(DB_PATH)

cursor_obj = connection_obj.cursor()
```

```
cursor_obj.execute("SELECT * FROM book_author WHERE
author_id = ?", (rowling_id,))

self.assertEqual(len(cursor_obj.fetchall()), 0)

connection_obj.close()


# Also verify trying to access via manager reflects the
deletion

# Re-fetch harry_potter as the original instance might be
stale if caching were involved

harry_potter_refetched =
Book.objects.get(id=harry_potter.id)

remaining_authors =
list(harry_potter_refetched.authors.all())

self.assertEqual(len(remaining_authors), 0)



def test_m2m_custom_junction_table(self):

"""Test M2M relationships with custom junction table
using manager."""

# Use instance from setUp

rowling = self.rowling

# Create CustomBook instance
```

```
custom_book_inst = CustomBook(title="Custom Book")

CustomBook.insert_entries([custom_book_inst]) # Insert
and update ID


# Add relationship using instance manager

custom_book_inst.authors.add(rowling)


# Verify relationship exists in custom table

connection_obj = sqlite3.connect(DB_PATH)

cursor_obj = connection_obj.cursor()

cursor_obj.execute("SELECT * FROM customjunction WHERE
custombook_id = ? AND author_id = ?",
(custom_book_inst.id, rowling.id))

self.assertEqual(len(cursor_obj.fetchall()), 1)

connection_obj.close()


# Verify retrieval via manager's all()

authors = list(custom_book_inst.authors.all())

self.assertEqual(len(authors), 1)

self.assertEqual(authors[0].id, rowling.id)

self.assertEqual(authors[0].name, rowling.name)
```

```
def test_m2m_invalid_relationship(self):

"""Test adding relationship with non-existent target ID
using manager."""

# Create an Author instance but don't save it (no ID)

unsaved_author = Author(name="Unsaved Author")

harry_potter = self.harry_potter # Use instance from
setUp


# Adding unsaved instance should raise ValueError

with self.assertRaisesRegex(ValueError, "Cannot add
unsaved 'author' instance"):

harry_potter.authors.add(unsaved_author)


# Create an Author instance with a fake ID that doesn't
exist in DB

invalid_author = Author(id=999, name="Invalid Author")

# Adding instance with non-existent ID should raise
ValueError (due to FK constraint)

with self.assertRaisesRegex(ValueError, "Invalid target
ID"):

harry_potter.authors.add(invalid_author)
```

```
def test_remove_nonexistent_relationship(self):
"""Test removing a relationship that doesn't exist using
manager."""
# Use instances from setUp
rowling = self.rowling
harry_potter = self.harry_potter

# Should complete without errors
harry_potter.authors.remove(rowling)
harry_authors = list(harry_potter.authors.all())
self.assertEqual(len(harry_authors), 0)

def test_m2m_multiple_operations(self):
"""Test complex add/remove sequences using manager."""
# Use instances from setUp
rowling = self.rowling
orwell = self.orwell
harry_potter = self.harry_potter
```

```
# Add two authors (can add multiple at once)

harry_potter.authors.add(rowling, orwell)

authors = list(harry_potter.authors.all())

self.assertEqual(len(authors), 2)

author_ids = {a.id for a in authors}

self.assertIn(rowling.id, author_ids)

self.assertIn(orwell.id, author_ids)


# Remove one author

harry_potter.authors.remove(rowling)

authors = list(harry_potter.authors.all())

self.assertEqual(len(authors), 1)

self.assertEqual(authors[0].id, orwell.id)

self.assertEqual(authors[0].name, "George Orwell")


def test_empty_relationships(self):

"""Test retrieving relationships when none exist using
manager."""

# Use instance from setUp

harry_potter = self.harry_potter

authors_qs = harry_potter.authors.all()
```

```
self.assertIsInstance(authors_qs, QuerySet)

authors = list(authors_qs)

self.assertEqual(len(authors), 0)


def test_as_dict_with_m2m(self):

"""Test the as_dict() method includes M2M
relationships."""

# Use instances from setUp

rowling = self.rowling

orwell = self.orwell

harry_potter = self.harry_potter

nineteen_eighty_four = self.nineteen_eighty_four


# Add authors to Harry Potter

harry_potter.authors.add(rowling)


# Get dict representation of Harry Potter

hp_dict = harry_potter.as_dict()


# Expected dict for Harry Potter

expected_hp_dict = {
```

```
'id': harry_potter.id,

'title': "Harry Potter",

'authors': [rowling.id] # Should contain list of related
IDs

}

self.assertDictEqual(hp_dict, expected_hp_dict)


# Add another author to Harry Potter

harry_potter.authors.add(orwell)

hp_dict_updated = harry_potter.as_dict()

# Order of IDs might not be guaranteed, so compare sets

self.assertEqual(hp_dict_updated['id'], harry_potter.id)

self.assertEqual(hp_dict_updated['title'], "Harry
Potter")

self.assertIsInstance(hp_dict_updated['authors'], list)

self.assertSetEqual(set(hp_dict_updated['authors']),
{rowling.id, orwell.id})


# Get dict representation of 1984 (no authors added yet)

nef_dict = nineteen_eighty_four.as_dict()

expected_nef_dict = {
```

```
'id': nineteen_eighty_four.id,

'title': "1984",

'authors': [] # Should be an empty list

}

self.assertDictEqual(nef_dict, expected_nef_dict)


def test_as_dict_unsaved_instance(self):

"""Test as_dict() on an unsaved instance with M2M
fields."""

unsaved_book = Book(title="Unsaved Book")

book_dict = unsaved_book.as_dict()


expected_dict = {

'id': None,

'title': "Unsaved Book",

'authors': [] # M2M should be empty list for unsaved
instances

}

self.assertDictEqual(book_dict, expected_dict)


@classmethod
```

```
def tearDownClass(cls):

"""Clean up the database file after all tests."""

if os.path.exists(DB_PATH):

os.remove(DB_PATH)

# Attempt to remove directory if empty

if os.path.exists('databases'):

try:

os.rmdir('databases')

except OSError:

pass # Ignore if not empty


if __name__ == '__main__':

unittest.main()
```

#tests/testMigrationHist.py

```
import unittest

import os

import shutil

from pathlib import Path
```

```
class TestMigrationHistory(unittest.TestCase):

def setUp(self):

"""Set up temporary migrations directory and database."""

self.migrations_dir = Path("migrations")

if self.migrations_dir.exists():

shutil.rmtree(self.migrations_dir)

self.migrations_dir.mkdir()


# Create a migration file

migration_file = self.migrations_dir /
"0001_initial_migration.py"

with open(migration_file, "w") as f:

f.write("""

from ORM.base import BaseModel

from ORM.datatypes import CharField


class TestModel(BaseModel):

name = CharField()
```

```
def migrate():

TestModel.create_table()

""")


def test_migration_tracking(self):

"""Test that migrations are properly tracked once
applied."""

from ORM.manager import apply_migrations,
get_applied_migrations


# Apply the migration

apply_migrations()


# Check that migration is recorded as applied

applied = get_applied_migrations()

self.assertIn("0001_initial_migration", applied,

"Migration should be recorded in tracking table")


# Reapplying migrations should not error and should skip
already applied ones

apply_migrations() # This should run without errors
```

```
# Verify table exists in database
import sqlite3
connection = sqlite3.connect("databases/main.sqlite3")
cursor = connection.cursor()

# Check the model table was created
cursor.execute(
"SELECT name FROM sqlite_master WHERE type='table' AND
name='testmodel';")
table_exists = cursor.fetchone()
self.assertIsNotNone(
table_exists, "The model table should be created.")

# Check the migrations table was created
cursor.execute(
"SELECT name FROM sqlite_master WHERE type='table' AND
name='orm_migrations';")
migrations_table_exists = cursor.fetchone()
self.assertIsNotNone(migrations_table_exists,
"The migrations tracking table should be created.")
```

```
# Check the migration is recorded in the table
cursor.execute("SELECT migration_name FROM
orm_migrations;")
recorded_migrations = cursor.fetchall()
self.assertEqual(len(recorded_migrations), 1,
"One migration should be recorded")
self.assertEqual(recorded_migrations[0][0],
"0001_initial_migration",
"The correct migration name should be recorded")


connection.close()


def tearDown(self):
"""Clean up the migrations directory and database."""
if self.migrations_dir.exists():
shutil.rmtree(self.migrations_dir)
if os.path.exists("databases/main.sqlite3"):
os.remove("databases/main.sqlite3")
if os.path.exists("databases"):
os.rmdir("databases")
```

```
if __name__ == "__main__":
unittest.main()
```

#tests/testMigrations.py

```
import unittest
import os
import shutil
from pathlib import Path
from ORM.manager import find_models, generate_migrations,
apply_migrations
from ORM.base import BaseModel
from ORM.datatypes import CharField

# Temporary test directory for models
TEST_APP_DIR = "test_app"



class TestModelDiscovery(unittest.TestCase):
"""
```

```
Test case for discovering models in a specified
directory.

This test case creates a temporary directory with a test
model

and verifies that the model is correctly discovered by
the find_models function.

"""

@classmethod

def setUpClass(cls):

"""Set up a temporary app directory with test models."""

if not os.path.exists(TEST_APP_DIR):

os.makedirs(TEST_APP_DIR)


# Create a test model file

with open(os.path.join(TEST_APP_DIR, "test_model.py"),
"w") as f:

f.write("""

from ORM.base import BaseModel

from ORM.datatypes import CharField


class TestModel(BaseModel):

name = CharField()
```

```
""")

def test_find_models(self):
"""Test that find_models correctly identifies models
inheriting from BaseModel."""
project_root = os.getcwd()
models = find_models(project_root,
models_folder=TEST_APP_DIR)
self.assertEqual(
len(models), 1, "find_models should discover one model.")
self.assertEqual(models[0].__name__, "TestModel",
"The discovered model should be 'TestModel'.")

@classmethod
def tearDownClass(cls):
"""Clean up the temporary app directory."""
if os.path.exists(TEST_APP_DIR):
shutil.rmtree(TEST_APP_DIR)


class TestMigrationGeneration(unittest.TestCase):
```

"""

TestMigrationGeneration is a test suite for verifying the behavior of the migration generation system.

It ensures that migrations are generated correctly based on model changes, handles edge cases, and validates

expected outcomes.

Test cases included:

1. `test_generate_migrations`:

Verifies that migrations are generated correctly for new models and that no duplicate migrations

are created when models remain unchanged.

2. `test_field_modification`:

Ensures that modifying field attributes (e.g., nullability) generates a new migration.

3. `test_removing_field`:

Tests that removing a field from a model generates a new migration.

4. `test_multiple_models`:

Verifies that migrations can handle multiple models and that changes to one model generate a new migration.

5. `test_consecutive_changes`:

Tests the behavior of consecutive changes to the same model, ensuring each change generates a new migration.

6. `test_empty_models_list`:

Ensures that no migrations are generated when the models list is empty.

7. `test_unchanged_migration_signature`:

Verifies that non-model-changing updates (e.g., comments) do not trigger a new migration.

Each test case sets up a controlled environment, generates migrations, and verifies the expected migration files

and their contents.

"""

```
def setUp(self):

"""Set up a temporary migrations directory."""

self.migrations_dir = Path("migrations")

if self.migrations_dir.exists():

shutil.rmtree(self.migrations_dir)

self.migrations_dir.mkdir()
```

```
def test_generate_migrations(self):

"""Test that generate_migrations creates a valid
migration file."""

class TestModel(BaseModel):

name = CharField()


# First migration should be generated

generate_migrations([TestModel])


# Find the generated migration file (should have format
like 0001_migration_*.py)

migration_files =
list(self.migrations_dir.glob("????_*.py"))

self.assertEqual(len(migration_files), 1,

"One migration file should be created")


migration_file = migration_files[0]

self.assertTrue(migration_file.exists(),

"Migration file should be created.")


with open(migration_file, "r") as f:
```

```
content = f.read()

self.assertIn("def migrate():", content,

"Migration file should contain a migrate function.")

self.assertIn("TestModel.create_table()", content,

"Migration file should include table creation for
TestModel.")


# Capture the files before the second generation attempt

files_before = set(self.migrations_dir.glob("*.py"))


# Running again with the same model should NOT generate a
new migration

generate_migrations([TestModel])


# Verify no new migrations were created

files_after = set(self.migrations_dir.glob("*.py"))

self.assertEqual(files_before, files_after,

"No new migration should be generated when models haven't
changed")


# Now, modify the model
```

```
class TestModel(BaseModel):

name = CharField()

description = CharField() # Added field


# This should generate a new migration

generate_migrations([TestModel])


# There should now be two migration files

migration_files =
list(self.migrations_dir.glob("????_*.py"))

self.assertEqual(len(migration_files), 2,

"A second migration should be created when models
change")


def test_field_modification(self):

"""Test that changing field attributes generates a new
migration."""

class TestModel(BaseModel):

name = CharField(null=False)


# Generate initial migration
```

```
generate_migrations([TestModel])

initial_migrations =
list(self.migrations_dir.glob("????_*.py"))

self.assertEqual(len(initial_migrations), 1)


# Modify field attribute

class TestModel(BaseModel):

name = CharField(null=True) # Changed null attribute


# Generate another migration

generate_migrations([TestModel])


# Should have a new migration

updated_migrations =
list(self.migrations_dir.glob("????_*.py"))

self.assertEqual(len(updated_migrations), 2,

"Changing field attributes should create a new
migration")


def test_removing_field(self):

"""Test that removing a field generates a new
migration."""
```

```
class TestModel(BaseModel):

name = CharField()

age = CharField()


# Generate initial migration

generate_migrations([TestModel])


# Remove a field

class TestModel(BaseModel):

name = CharField() # age field removed


# Generate another migration

generate_migrations([TestModel])


# Should have a new migration

migrations = list(self.migrations_dir.glob("????_*.py"))

self.assertEqual(len(migrations), 2,

"Removing a field should create a new migration")


def test_multiple_models(self):
```

```
"""Test handling multiple models at once."""

class FirstModel(BaseModel):

title = CharField()


class SecondModel(BaseModel):

name = CharField()


# Generate migration with two models

generate_migrations([FirstModel, SecondModel])


# Check that one migration file is created

migration_files =
list(self.migrations_dir.glob("????_*.py"))

self.assertEqual(len(migration_files), 1)


# Check both models are in the migration

with open(migration_files[0], "r") as f:

content = f.read()

self.assertIn("FirstModel.create_table()", content)

self.assertIn("SecondModel.create_table()", content)
```

```
# Change only one model

class FirstModel(BaseModel):

title = CharField()

content = CharField() # Added field


class SecondModel(BaseModel):

name = CharField() # Unchanged


# Generate a new migration

generate_migrations([FirstModel, SecondModel])


# Should have a new migration

migration_files =
list(self.migrations_dir.glob("????_*.py"))

self.assertEqual(len(migration_files), 2,

"Changing one model should create a new migration")


def test_consecutive_changes(self):

"""Test multiple consecutive changes to the same
model."""

class TestModel(BaseModel):
```

```
name = CharField()


# Initial migration

generate_migrations([TestModel])


# First change - add a field

class TestModel(BaseModel):

name = CharField()

description = CharField()


generate_migrations([TestModel])


# Second change - add another field

class TestModel(BaseModel):

name = CharField()

description = CharField()

created_at = CharField()


generate_migrations([TestModel])
```

```
# Third change - remove a field
class TestModel(BaseModel):
name = CharField()
created_at = CharField() # description removed

generate_migrations([TestModel])

# Should have four migrations total
migration_files =
list(self.migrations_dir.glob("????_*.py"))
self.assertEqual(len(migration_files), 4,
"Each model change should create a new migration")

def test_empty_models_list(self):
"""Test behavior with an empty models list."""
# Generate with empty list
generate_migrations([])

# Should not create any migrations
migration_files =
list(self.migrations_dir.glob("????_*.py"))
```

```
self.assertEqual(len(migration_files), 0,
"No migrations should be created for empty models list")


def test_unchanged_migration_signature(self):
"""Test that adding a non-model changing comment doesn't
trigger a migration."""
# Define a model with a comment
class TestModel(BaseModel):
name = CharField()
# This is a comment that doesn't affect the model


# Generate initial migration
generate_migrations([TestModel])


# Update the comment only
class TestModel(BaseModel):
name = CharField()
# This is a different comment that still doesn't affect
the model


# This shouldn't generate a new migration
```

```
generate_migrations([TestModel])


# Should still have only one migration
migration_files =
list(self.migrations_dir.glob("????_*.py"))
self.assertEqual(len(migration_files), 1,
"Comments and whitespace shouldn't trigger new
migrations")


def tearDown(self):
"""Clean up the migrations directory."""
if self.migrations_dir.exists():
shutil.rmtree(self.migrations_dir)


class TestMigrationApplication(unittest.TestCase):
"""
TestMigrationApplication is a test suite for verifying
the behavior of a database migration system.
It ensures that migrations are applied correctly, handles
edge cases, and validates expected outcomes.
Test cases included:
1. `test_apply_migrations`:
```

Verifies that migrations are applied successfully and the corresponding database tables are created.

2. `test_empty_migrations_directory`:

Tests the behavior when the migrations directory is empty, ensuring no errors occur.

3. `test_failed_migration`:

Ensures that failed migrations are handled gracefully and are not recorded in the database.

4. `test_duplicate_application`:

Confirms that applying migrations multiple times does not result in duplicate entries or errors.

5. `test_out_of_order_migrations`:

Tests that migrations are applied in the correct numerical order, even if the files are out of order.

6. `test_migration_with_dependencies`:

Verifies that migrations with dependencies on previous migrations are applied correctly.

7. `test_non_existent_migrations_dir`:

Ensures that the system handles the absence of a migrations directory gracefully.

8. `test_apply_specific_migration`:

Tests the ability to apply a specific migration by name without affecting other migrations.

```
Each test case sets up a controlled environment, applies
migrations, and verifies the expected database state or
behavior.
"""


def setUp(self):
"""Set up a temporary migrations directory and
database."""
self.migrations_dir = Path("migrations")
if self.migrations_dir.exists():
shutil.rmtree(self.migrations_dir)
self.migrations_dir.mkdir()


# Create a migration file
migration_file = self.migrations_dir /
"0001_initial_migration.py"
with open(migration_file, "w") as f:
f.write("""
from ORM.base import BaseModel
from ORM.datatypes import CharField


class TestModel(BaseModel):
```

```
name = CharField()


def migrate():

TestModel.create_table()

""")


def test_apply_migrations(self):

"""Test that apply_migrations successfully applies
migrations."""

apply_migrations()


# Verify that the table was created

import sqlite3

connection = sqlite3.connect("databases/main.sqlite3")

cursor = connection.cursor()

cursor.execute(

"SELECT name FROM sqlite_master WHERE type='table' AND
name='testmodel';")

table_exists = cursor.fetchone()

self.assertIsNotNone(

table_exists, "The 'testmodel' table should be created.")
```

```
connection.close()


def test_empty_migrations_directory(self):
"""Test behavior when migrations directory is empty."""
# Remove any migration files
for file in self.migrations_dir.glob("*.py"):
os.remove(file)


# Apply migrations with empty directory
apply_migrations()


# This should not error and should simply report no
migrations to apply
# We just verify that the function returns without error


def test_failed_migration(self):
"""Test handling of a failed migration."""
# Create a migration file with an error
bad_migration = self.migrations_dir /
"0002_bad_migration.py"
with open(bad_migration, "w") as f:
```

```
f.write("""
def migrate():
# This will raise a NameError
undefined_variable + 1
""")

# Apply migrations
with self.assertRaises(Exception):
apply_migrations()

# Check that no record of the bad migration exists
import sqlite3
connection = sqlite3.connect("databases/main.sqlite3")
cursor = connection.cursor()
cursor.execute("SELECT migration_name FROM
orm_migrations;")
recorded_migrations = [row[0] for row in
cursor.fetchall()]
self.assertNotIn("0002_bad_migration",
recorded_migrations,
"Failed migrations should not be recorded")
```

```
connection.close()


def test_duplicate_application(self):
"""Test that applying migrations multiple times is
safe."""
# First application
apply_migrations()


# Second application should skip already applied
migrations
apply_migrations()


# Third application still shouldn't error
apply_migrations()


# Check that the migration is only recorded once
import sqlite3
connection = sqlite3.connect("databases/main.sqlite3")
cursor = connection.cursor()
cursor.execute(
```

```
"SELECT migration_name, COUNT(*) FROM orm_migrations
GROUP BY migration_name;")

counts = cursor.fetchall()

for migration, count in counts:

self.assertEqual(

count, 1, f"Migration {migration} should only be recorded
once")

connection.close()


def test_out_of_order_migrations(self):

"""Test behavior with out-of-order migration files."""

# Create migrations out of numerical order

third_migration = self.migrations_dir /
"0003_third_migration.py"

with open(third_migration, "w") as f:

f.write("""

from ORM.base import BaseModel

from ORM.datatypes import CharField


class ThirdModel(BaseModel):

content = CharField()
```

```
def migrate():
ThirdModel.create_table()
""")

# Apply migrations - they should be applied in numerical order
apply_migrations()

# Verify tables exist in expected order
import sqlite3
connection = sqlite3.connect("databases/main.sqlite3")
cursor = connection.cursor()

# Check all tables were created
cursor.execute(
"SELECT name FROM sqlite_master WHERE type='table' AND name='testmodel';")
self.assertIsNotNone(
cursor.fetchone(), "First migration table should be created")
```

```
cursor.execute(
"SELECT name FROM sqlite_master WHERE type='table' AND
name='thirdmodel';")
self.assertIsNotNone(
cursor.fetchone(), "Third migration table should be
created")

# Check order in which migrations were applied
cursor.execute(
"SELECT migration_name FROM orm_migrations ORDER BY id;")
migration_order = [row[0] for row in cursor.fetchall()]
self.assertEqual(migration_order[0],
"0001_initial_migration",
"First migration should be applied first")
self.assertEqual(migration_order[1],
"0003_third_migration",
"Third migration should be applied next")

connection.close()

def test_migration_with_dependencies(self):
```

```
"""Test migrations that depend on previous migrations."""

# Create a migration that depends on a previous migration

second_migration = self.migrations_dir /
"0002_dependent_migration.py"

with open(second_migration, "w") as f:

f.write("""

from ORM.base import BaseModel

from ORM.datatypes import CharField


class TestModel(BaseModel):

# This model is defined in the first migration

# We're adding a method that depends on the table
existing

pass


def migrate():

# This migration only works if TestModel table already
exists

import sqlite3

connection = sqlite3.connect("databases/main.sqlite3")

cursor = connection.cursor()
```

```
cursor.execute("ALTER TABLE testmodel ADD COLUMN
description TEXT;")

connection.commit()

connection.close()

""")


# Apply migrations

apply_migrations()


# Check that the column was added

import sqlite3

connection = sqlite3.connect("databases/main.sqlite3")

cursor = connection.cursor()

cursor.execute("PRAGMA table_info(testmodel);")

columns = [row[1] for row in cursor.fetchall()]

self.assertIn("description", columns,

"The dependent migration should add a column")

connection.close()


def test_non_existent_migrations_dir(self):
```

```
"""Test behavior when migrations directory doesn't
exist."""

# Remove migrations directory

shutil.rmtree(self.migrations_dir)


# Apply migrations should handle this gracefully

apply_migrations()

# We just verify that no exception is raised


def test_apply_specific_migration(self):

"""Test applying a specific migration by name."""

# Create a second migration

second_migration = self.migrations_dir /
"0002_second_migration.py"

with open(second_migration, "w") as f:

f.write("""

from ORM.base import BaseModel

from ORM.datatypes import CharField


class SecondModel(BaseModel):

title = CharField()
```

```
def migrate():
SecondModel.create_table()
""")

# Apply only the second migration directly
apply_migrations(specific_migration="0002_second_migratio
n")

# Verify only the second model table exists
import sqlite3
connection = sqlite3.connect("databases/main.sqlite3")
cursor = connection.cursor()

cursor.execute(
"SELECT name FROM sqlite_master WHERE type='table' AND
name='secondmodel';")
self.assertIsNotNone(
cursor.fetchone(), "Second model table should be
created")
```

```
cursor.execute(
"SELECT name FROM sqlite_master WHERE type='table' AND
name='testmodel';")
self.assertIsNone(cursor.fetchone(),
"First model table should not be created")

# Check that only the specific migration is recorded
cursor.execute("SELECT migration_name FROM
orm_migrations;")
recorded_migrations = [row[0] for row in
cursor.fetchall()]
self.assertEqual(len(recorded_migrations), 1,
"Only one migration should be recorded")
self.assertEqual(recorded_migrations[0],
"0002_second_migration",
"The specific migration should be recorded")

connection.close()

def tearDown(self):
"""Clean up the migrations directory and database."""
if self.migrations_dir.exists():
```

```
shutil.rmtree(self.migrations_dir)

if os.path.exists("databases/main.sqlite3"):

os.remove("databases/main.sqlite3")

if os.path.exists("databases"):

os.rmdir("databases")




if __name__ == "__main__":

unittest.main()
```