СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 3](#_Toc170472183)

[Реализация вспомогательных функций 4](#_Toc170472184)

[Разработка ORM 6](#_Toc170472185)

[Исследования 9](#_Toc170472186)

[ЗАКЛЮЧЕНИЕ 17](#_Toc170472187)

[СПИСОК ЛИТЕРАТУРЫ 19](#_Toc170472188)

[ПРИЛОЖЕНИЕ 20](#_Toc170472189)

ВВЕДЕНИЕ

Учебная (научно-исследовательская работа) практика проходила на кафедре системного анализа и информационных технологий Института вычислительной математики и информационных технологий КФУ с 09 февраля 2024 по 07 июня 2024 года.

Целью практики является получение навыков проведения исследований и анализа полученных результатов.

Для достижения цели были поставлены следующие задачи:

1. изучить способы генерации случайных данных, методы измерения времени и построения графиков;
2. провести исследования эффективности программ, взаимодействующих с БД, на примере базы данных «AirportSystem»;
3. проанализировать полученные результаты.

# Реализация вспомогательных функций

Для упрощения работы с базой данных MySQL был разработан модуль connection, включающий класс DatabaseConnection. Этот класс выполняет роль контекстного менеджера [1], программный код которого представлен в приложении. Данный контекстный менеджер автоматически управляет подключением к базе данных, открытием и закрытием соединений, а также выполнением коммитов транзакций.

Модуль db\_methods предоставляет функции для резервного копирования и восстановления базы данных MySQL. Функция backup\_database создает бэкап базы данных и сохраняет его в файл. Для этого она открывает соединение с базой данных, извлекает имена таблиц и данные из каждой таблицы, после чего записывает их в файл. В случае ошибки, связанной с подключением к базе данных или выполнением SQL-запросов, функция выводит сообщение об ошибке.

Функция restore\_database восстанавливает базу данных из файла бэкапа. Она открывает файл бэкапа, проверяет соответствие названия базы данных, очищает существующие данные в таблицах и восстанавливает их из файла. Эта функция также обрабатывает ошибки, возникающие при подключении или выполнении запросов.

Модуль helpers включает функции, облегчающие работу с данными и взаимодействие с базой данных. Функция get\_identifiers возвращает список значений указанного идентификатора для каждого элемента в данных. Функция fetch\_existing\_pks извлекает существующие первичные ключи из указанной таблицы, что полезно для проверки уникальности данных.

Функция ensure\_unique\_identifiers гарантирует уникальность идентификаторов в данных, перегенерируя их, если они уже существуют в таблице. Это достигается путем проверки существующих и новых идентификаторов и их обновления при необходимости. Функция format\_data форматирует данные в соответствии с количеством полей в SQL-запросе.

Модуль sandbox предоставляет функции для создания копии базы данных MySQL. Функция create\_table\_sandbox создает SQL-запрос для создания таблицы с такой же структурой, как в основной базе данных. Функция create\_sandbox создает новую базу данных, копирует структуру таблиц из основной базы и создает их в песочнице. Это позволяет проводить эксперименты без риска повреждения данных в основной базе.

Модуль table\_methods включает функции для работы с таблицами в базе данных. Функция get\_table\_columns получает информацию о столбцах указанной таблицы, что необходимо для создания копий таблиц в песочнице. Функция get\_table\_names возвращает список имен таблиц из базы данных. Функция delete\_all\_data удаляет все данные из указанной таблицы, что полезно при восстановлении базы данных из бэкапа или при тестировании. Функция replace\_all\_data заменяет все данные в таблице на сгенерированные данные на основе переданного класса модели, что позволяет обновлять содержимое таблиц для тестов и экспериментов.

Для всех разработанных функций были написаны ручные тесты, которые обеспечивают корректность работы всех представленных функций.

# Разработка ORM

Разработка ORM (Object-Relational Mapping) [2] системы началась с создания модуля fields.py, который определяет различные типы полей, используемые в моделях базы данных. Этот модуль включает несколько классов, каждый из которых представляет отдельный тип поля и его специфические характеристики.

Первым был создан базовый класс Field. Этот класс служит основой для всех других типов полей. В нем определены атрибуты, такие как имя поля, возможность быть пустым, значение по умолчанию, указание на первичный ключ и дополнительные ограничения. Метод инициализации позволяет задавать все эти параметры через произвольные именованные аргументы.

Затем был разработан подклассы на основе класса Field. Класс IntegerField предназначен для целочисленных полей с дополнительными ограничениями. Класс CharField предназначен для символьных полей с ограничением в виде максимальной длины. Класс ForeignKey предназначен для полей внешнего ключа, которые ссылаются на другую таблицу. В нем определены атрибуты, включающие тип поля и имя таблицы, на которую ссылается внешний ключ. Это позволяет устанавливать связи между таблицами, обеспечивая целостность данных. Класс TimeField предназначен для полей времени. Класс ManyToManyField предназначен для полей отношения «многие ко многим». В нем определены атрибуты, включающие тип поля «MANYTOMANY» и имя таблицы, на которую ссылается отношение. Программный код модуля fields.py представлен в приложении.

Далее был разработан модуль class\_basis.py. В нем были определены классы для всех моделей, базовый класс Model и метакласс ModelMeta.

Метакласс ModelMeta обеспечивает динамическое создание моделей базы данных на основе их атрибутов [3]. Данный класс использует механизмы Python для сканирования атрибутов класса модели. Он ищет все атрибуты, которые являются экземплярами класса Field. Когда метакласс находит атрибуты, являющиеся экземплярами Field, он автоматически создает атрибуты класса модели, представляющие собой поля базы данных.

Для каждого поля метакласс анализирует документацию, связанную с классом модели. Это позволяет автоматически настраивать типы полей и их ограничения.

После создания ModelMeta был реализован класс Model, представляющий базовую модель для всех объектов базы данных. Он использует метакласс ModelMeta для управления своими полями. Включает методы для инициализации, сохранения в базу данных, создания таблицы и генерации случайных объектов модели.

Создание таблицы происходит с помощью метода create\_table. Данный Метод генерирует SQL-запрос для создания таблицы на основе полей модели. Для каждого поля определяется тип столбца в таблице, используя информацию из поля. Если был указан тип связи «многие ко многим», автоматически создается дополнительная связующая таблица.

Сохранение записи реализовано при помощи метода save. Метод сохраняет текущий объект модели в базе данных. Сначала определяется название первичного ключа модели и проверяется его значение, если в базе данных запись с таким значением первичного ключа уже существует, то происходит перегенерация. Далее создается INSERT-запрос для сохранения объекта.

Генерация данных реализована с помощью метода generate. Метод генерирует n случайных объектов данного класса. Для каждого поля класса определяется случайное значение в соответствии с его типом и ограничениями.

Используя разработанную ORM систему, были созданы классы для всех таблиц базы данных. Эти классы описывают структуру таблиц и их поля, а также ограничения предметной области. Например, для таблицы baggage был создан следующий класс:

class Baggage(Model):

"""

id: IntegerField: null=False, primary\_key=True

flight\_id: ForeignKey: to=Flights, null=False

weight: IntegerField: null=False, min=0, max=120

passenger\_id: ForeignKey: to=Passengers, null=False

"""

На примере показано, как использование классов полей и моделей в разработанной системе позволяет декларативно определять структуру и взаимосвязи таблиц базы данных.

# Исследования

Для выполнения исследований применялись различные инструменты. Основой для хранения и управления данными служила СУБД MySQL. Весь код, включая вспомогательные функции и ORM систему, был написан на языке программирования Python. Для измерения времени выполнения запросов и генерации данных использовалась библиотека timeit [4]. Для визуализации результатов применялась библиотека matplotlib [5] и созданный класс PlotBuilder.

Исследования начались с анализа времени, затрачиваемого на генерацию данных для таблиц при разных объемах строк. Для этого были разработана функция measure\_generation\_time, измеряющая время генерации данных для таблицы:

def measure\_generation\_time(table\_class, size):

"""

Измеряет время генерации данных.

Параметры

----------

table\_class : class

Класс модели таблицы.

size : int

Количество генерируемых записей.

Возвращает

----------

float

Время генерации данных в секундах.

"""

class\_name = table\_class.\_\_name\_\_

module\_name = table\_class.\_\_module\_\_

stmt = f"list({class\_name}.generate({size}))"

setup = f"from {module\_name} import {class\_name}"

timer = timeit.Timer(stmt, setup=setup)

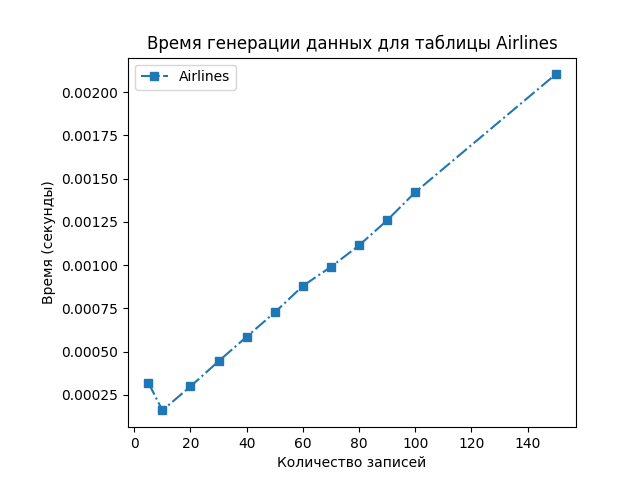
time = timer.timeit(number=5)

return time

Данная функция работает следующим образом. Для указанного класса таблицы (table\_class) и заданного количества записей (size) функция измеряет время, необходимое для генерации данных. Сначала она определяет имя класса (class\_name) и модуль, из которого этот класс импортирован (module\_name). Затем формируется строка stmt, представляющая вызов метода generate() для создания указанного количества записей, и строка setup, содержащая импорт класса из соответствующего модуля. После этого создается объект Timer из библиотеки timeit, который использует эти строки. Функция вызывает метод timeit у объекта Timer, чтобы измерить время выполнения операции, повторяя её пять раз. Результирующее время генерации данных возвращается в секундах.

Затем был построен график, где по оси X отложено количество генерируемых строк, а по оси Y — время генерации в секундах.

Результаты исследования времени генерации для таблицы Airlines представлены на рисунке 1.

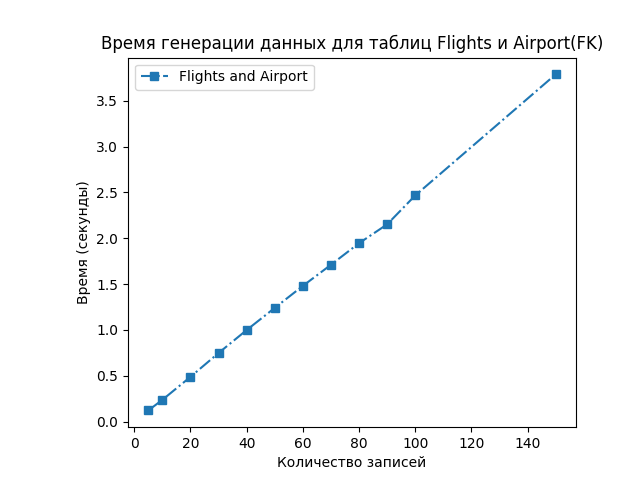


1. График времени генерации для таблицы Airlines

При анализе графика можно заметить, что время, необходимое для генерации данных, увеличивается пропорционально росту количества строк в таблицах. Например, при увеличении количества строк с 100 до 140, время выполнения возрастает с примерно на 0,00075 секунд. Это указывает на приблизительно линейную зависимость времени генерации от объема данных.

Для случая, когда таблицы имеют связи через внешние ключи (FK), используется та же функция measure\_generation\_time, но время генерации измеряется для двух связанных таблиц при различных объемах данных. Для пары таблиц выполняется замер времени генерации данных, затем эти времена складываются, что позволяет оценить полное время выполнения операции.

Результаты исследования времени генерации для двух связанных таблиц представлены на рисунке 2. Для данного случая так же наблюдается линейная зависимость времени генерации от количества данных. Время генерации при этом увеличилось, так как измеряется для двух таблиц.



1. График времени генерации для связанных таблиц

Далее был проведен анализ зависимости времени выполнения различных типов запросов (INSERT, SELECT, DELETE, UPDATE) от различного количества строк. Для этого была реализована функция measure\_query\_time:

def measure\_query\_time(query, cursor):

"""

Измеряет время выполнения запроса.

Параметры

----------

query : str

SQL-запрос.

cursor : mysql.connector.cursor.MySQLCursor

Курсор для выполнения операций с базой данных.

Возвращает

----------

float

Время выполнения запроса в секундах.

"""

import timeit

def execute\_query():

cursor.execute(query)

cursor.fetchall()

timer = timeit.Timer(execute\_query)

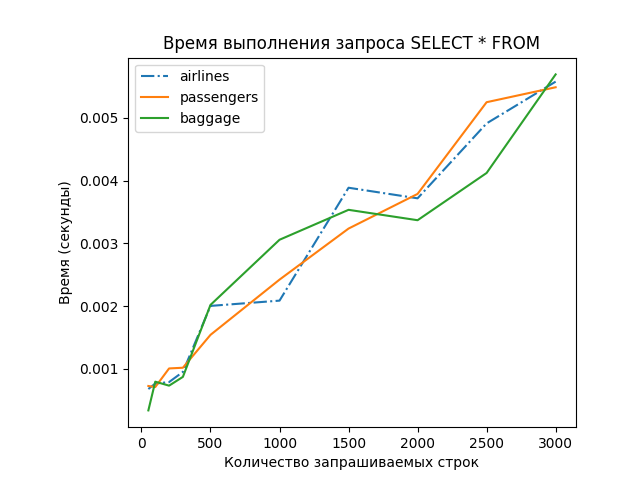
number = 1

execution\_time = timer.timeit(number=number)

return execution\_time

Данная функция работает следующим образом. Для указанного SQL-запроса (query) и курсора базы данных (cursor) функция измеряет время, необходимое для выполнения этого запроса. Сначала внутри функции определяется вложенная функция execute\_query(), которая выполняет переданный SQL-запрос с помощью метода cursor.execute() и извлекает все результаты с помощью cursor.fetchall(). Затем создается объект Timer из библиотеки timeit, который использует эту вложенную функцию. Метод timeit у объекта Timer вызывается один раз для измерения времени выполнения операции. Результирующее время выполнения запроса возвращается в секундах.

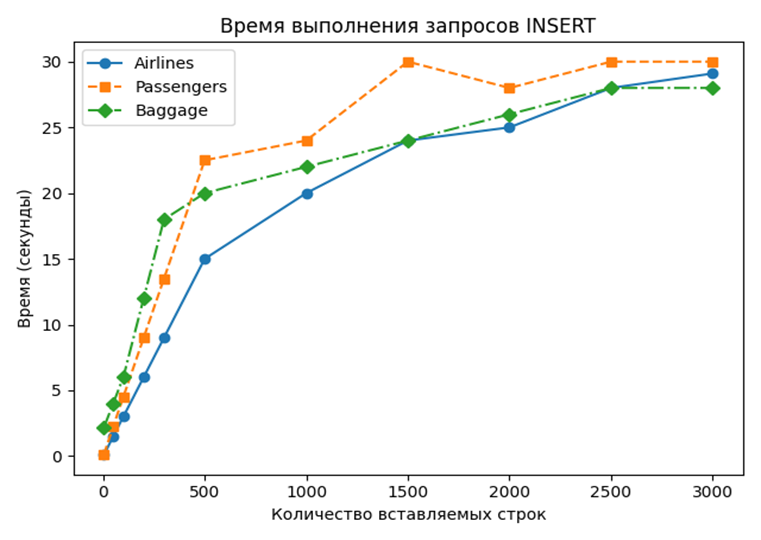
Графики зависимости времени выполнения запроса SELECT \* FROM от количества данных представлены на рисунке 3.



1. Графики времени выполнения запроса SELECT \* FROM

Графики показывают, что с ростом количества запрашиваемых строк увеличивается и время выполнения запросов. Это подтверждает линейную зависимость времени выполнения операций SELECT от объема данных в таблицах. Особенно очевиден рост времени при больших объемах данных, что может быть связано с необходимостью обработки и выборки большего количества записей.

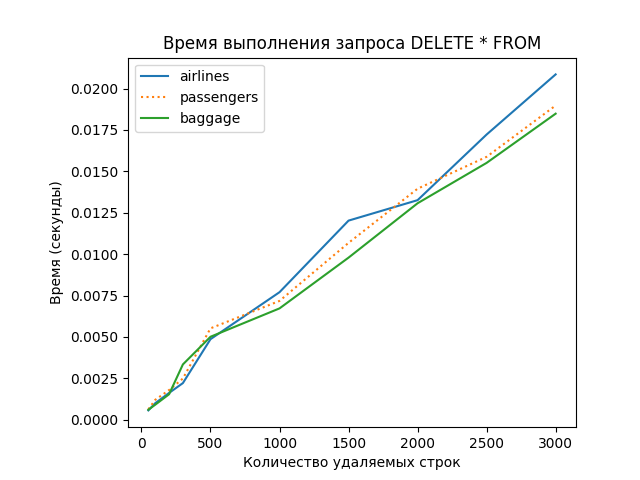
Графики других типов запросов SELECT представлены в приложении. Результаты исследования времени выполнения запроса INSERT представлены на рисунке 4.



1. Графики времени выполнения запроса INSERT

Графики для операций INSERT демонстрируют, что с увеличением числа записей возрастает и время выполнения вставок. Вставка данных занимает значительно больше времени по сравнению с выполнением SELECT-запросов. Это объясняется необходимостью обновления различных параметров и структур базы данных, что делает процесс вставки более трудоемким.

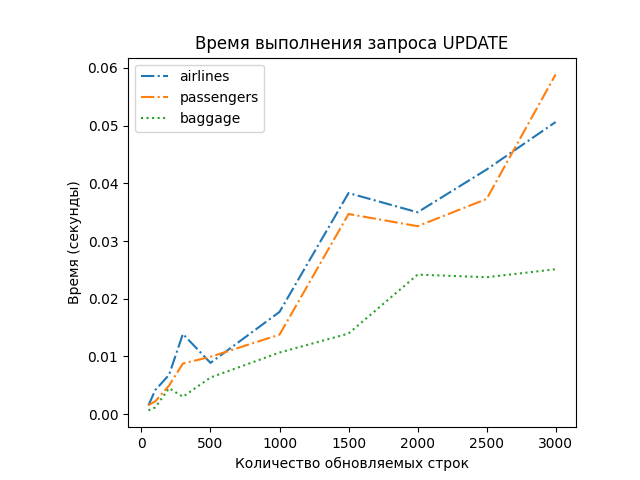
Результаты исследования времени выполнения запроса DELETE представлены на рисунке 5.



1. Графики времени выполнения запроса DELETE \* FROM

Как видно из графиков, операции DELETE требуют меньше времени по сравнению с INSERT, но больше времени по сравнению с SELECT. Это может быть обусловлено необходимостью производить операции удаления из базы данных, что является более трудоемким процессом, чем просто выборка данных.

Графики других типов запросов DELETE можно увидеть в приложении. Результаты исследования времени выполнения запроса UPDATE представлены на рисунке 6.



1. Графики времени выполнения запроса UPDATE

На графике операция UPDATE занимает больше времени, чем SELECT и DELETE, но меньше, чем INSERT. Это обусловлено необходимостью совершения как минимум двух операций – операции выборки и операции обновления и вставки данных. В целом, график имеет линейную зависимость с некоторыми отклонениями, что может быть обусловлено спецификой и структурой данных в таблицах.

Итак, проведенные исследования показали, что время выполнения операций с базой данных существенно зависит от объема данных и типа выполняемой операции. Например, операции INSERT требуют больше времени из-за обновления индексов и других структур базы данных, в то время как операции SELECT также демонстрируют увеличение времени с ростом объема данных, особенно при сложных запросах. Эти результаты подчеркивают важность оптимизации базы данных и запросов для работы с большими объемами данных.

ЗАКЛЮЧЕНИЕ

Во время работы были разработаны функции для взаимодействия с базой данных, такие как создание таблиц, генерация данных, их сохранение, удаление и создание резервных копий. Самым главным объектом разработки являлась ORM система, которая обеспечивает удобное взаимодействие с данными через классы полей и моделей, позволяя декларативно описывать структуру данных и операции с ними.

Оценка производительности охватывала измерение времени, требуемого для генерации данных и выполнения различных запросов (SELECT, INSERT, UPDATE и DELETE) в зависимости от объема данных. Визуализация и углубленный анализ результатов позволили выявить зависимость производительности базы данных от объема данных и типа операций. Дополнительно были рассмотрены различные вариации запросов для более широкого представления влияния различных факторов на производительность и поиска оптимальных решений для повышения эффективности работы базы данных.

За период практики были приобретены следующие компетенции ().

1. Компетенции

|  |  |  |
| --- | --- | --- |
| Компетенция | Расшифровка компетенции | Описание приобретенных знаний, умений и навыков |
| ПК-4 | Оценка и выбор варианта архитектуры программного средства | - умения и навыки проектирования программных средств проведения исследований |
| ПК-5 | Разработка тестовых случаев, проведение тестирования и исследование результатов | - умения и навыки проведения тестирования |

В ходе исследования были выявлены важные связи между временем выполнения операций и объемом данных, а также структурными особенностями таблиц. Основные результаты показали, что увеличение числа обрабатываемых строк приводит к линейному росту времени выполнения таких операций, как генерация данных и выполнение запросов. Кроме того, исследование продемонстрировало, что наличие внешних ключей также влияет на производительность. Генерация данных и выполнение запросов в таблицах, связанных через внешние ключи, требуют дополнительного времени из-за необходимости поддержания целостности данных и проведения дополнительных проверок.

Исследование также рассмотрело влияние различных типов запросов (SELECT, INSERT, UPDATE и DELETE) на производительность базы данных в зависимости от объема данных. Полученные результаты были визуализированы и подробно проанализированы, что позволило сделать выводы о зависимости производительности базы данных от объема данных и типа операций. Эти выводы помогут оптимизировать работу с базой данных, учитывая специфические особенности и требования каждого типа операции.

СПИСОК ЛИТЕРАТУРЫ

1. What are context managers in Python? [Электронный ресурс]. —2023. — URL: https://www.freecodecamp.org/news/context-managers-in-python/ (дата обращения 21.05.2024).
2. Object-relational Mappers (ORMs) [Электронный ресурс]. —2024. — URL: https://www.fullstackpython.com/object-relational-mappers-orms.html (дата обращения 07.06.2024).
3. [Метаклассы](https://matplotlib.org/) в Python [Электронный ресурс]. —2021. — URL: <https://habr.com/ru/companies/piter/articles/592127/> (дата обращения 19.05.2024).
4. [Модуль timeit в Python](https://ru.m.wikipedia.org/wiki/ORM) [Электронный ресурс]. —2022. — URL: https://docs-python.ru/standart-library/modul-timeit-python/ (дата обращения 19.05.2024).
5. Matplotlib – Visualisation with Python [Электронный ресурс]. —2023. — URL: https://matplotlib.org/ (дата обращения 19.05.2024).

ПРИЛОЖЕНИЕ

Программный код класса DatabaseConnection

class DatabaseConnection:

def \_\_init\_\_(self, host, user, password, database):

self.host = host

self.user = user

self.password = password

self.database = database

self.connection = None

self.cursor = None

def \_\_enter\_\_(self):

self.connection = mysql.connector.connect(

host=self.host,

user=self.user,

password=self.password,

database=self.database

)

self.cursor = self.connection.cursor()

return self.cursor, self.connection

def \_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb):

if self.cursor:

self.cursor.close()

if self.connection:

self.connection.commit()

self.connection.close()

Программный код модуля fields.py

class Field:

"""

Базовый класс для определения поля базы данных.

Атрибуты

--------

name : str

Имя поля.

null : bool

Указывает, может ли поле быть пустым (по умолчанию True).

default : any

Значение по умолчанию для поля (по умолчанию None).

primary\_key : bool

Указывает, является ли поле первичным ключом (по умолчанию False).

constraints : dict

Дополнительные ограничения для поля.

"""

def \_\_init\_\_(self, \*\*kwargs):

"""

Параметры

---------

\*\*kwargs : dict

Произвольные именованные аргументы для установки атрибутов поля.

"""

self.name = kwargs.get('name')

self.null = kwargs.get('null', True)

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

self.primary\_key = kwargs.get('primary\_key', False)

self.constraints = kwargs

class IntegerField(Field):

super().\_\_init\_\_(\*\*kwargs)

self.type = 'INT'

self.min = int(min) if min is not None else None

self.max = int(max) if max is not None else None

def validate(self, value):

if self.min is not None and value < self.min:

return False

if self.max is not None and value > self.max:

return False

return True

class CharField(Field):

def \_\_init\_\_(self, max\_length=45, words\_count=1, \*\*kwargs):

super().\_\_init\_\_(\*\*kwargs)

self.type = f'VARCHAR({max\_length})'

self.max\_length = int(max\_length)

self.words\_count = int(words\_count)

def validate(self, value):

if self.constraints.get(' words\_count') is not None:

if int(self.constraints.get(' words\_count')) > 1:

return True # В случае, если words\_count > 1, длина слов не будет проверяться

return True if len(str(value)) <= self.max\_length else False

class ForeignKey(Field):

super().\_\_init\_\_(\*\*kwargs)

self.type = 'INT'

self.to = to

class TimeField(Field):

def \_\_init\_\_(self, \*\*kwargs):

super().\_\_init\_\_(\*\*kwargs)

self.type = 'TIME'

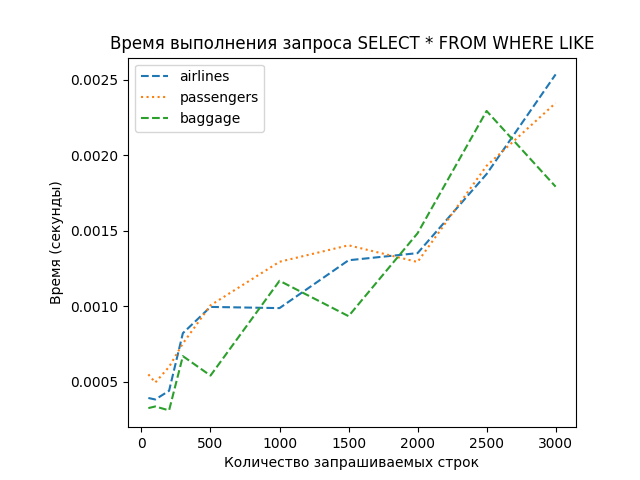
class ManyToManyField(Field):

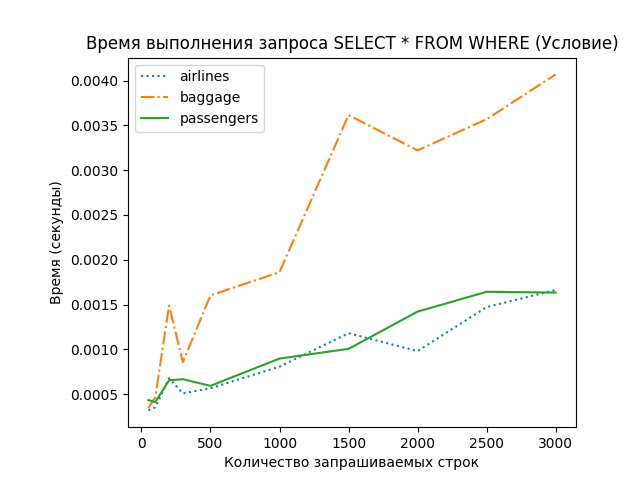
super().\_\_init\_\_(\*\*kwargs)

self.type = 'MANYTOMANY'

self.to = to

Графики времени выполнения запросов SELECT с различными условиями





Графики времени выполнения запросов DELETE с различными условиями

