ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
УФИМСКИЙ УНИВЕРСИТЕТ НАУКИ И ТЕХНОЛОГИЙ

РЕФЕРАТ

по дисциплине: «**Администрирование информационных систем**»

на тему «**Технология ORM: сравнительный анализ подходов Active Record (Django ORM) и Data Mapper (SQLAlchemy)**»

Выполнил: студент гр. ПРО-436Б
Мингазова Р. М.
Проверил:
доцент каф. ВМиК
Чернышев Е. С.

Уфа 2025 г.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Современная разработка программного обеспечения неразрывно связана с использованием баз данных для хранения и обработки информации. При этом разработчики сталкиваются с фундаментальной проблемой — несоответствием между объектно-ориентированной парадигмой программирования и реляционной моделью данных. Это несоответствие, известное как «object-relational impedance mismatch», порождает необходимость в промежуточном слое, который бы обеспечивал преобразование данных между этими двумя мирами.

Технология ORM (Object-Relational Mapping, объектно-реляционное отображение) представляет собой программную технику, позволяющую работать с реляционной базой данных, используя объектно-ориентированный подход. ORM автоматизирует преобразование данных между объектами в коде приложения и записями в таблицах базы данных, существенно упрощая разработку и сопровождение программных систем.

Согласно исследованию JetBrains State of Developer Ecosystem 2023, более 70% Python-разработчиков используют ORM в своих проектах. В экосистеме Python существуют два доминирующих ORM-решения: Django ORM и SQLAlchemy. Эти библиотеки реализуют принципиально различные архитектурные паттерны – Active Record и Data Mapper соответственно.

Актуальность темы обусловлена тем, что выбор между этими подходами оказывает существенное влияние на архитектуру приложения, его производительность, тестируемость и удобство сопровождения. Неправильный выбор может привести к техническому долгу и проблемам масштабирования.

Цель данного реферата – провести сравнительный анализ паттернов Active Record и Data Mapper на примере их реализации в Django ORM и SQLAlchemy, выявить преимущества и недостатки каждого подхода и

сформулировать рекомендации по выбору ORM для различных типов проектов.

1 Технология ORM: основные понятия и принципы

1.1 Определение и назначение ORM

ORM (Object-Relational Mapping) – это технология программирования, которая связывает базы данных с концепциями объектно-ориентированных языков программирования, создавая «виртуальную объектную базу данных» [Fowler, 2002]. По сути, ORM представляет собой слой абстракции между приложением и базой данных, который автоматически выполняет преобразование данных.

Основная задача ORM – устранение так называемого «импеданс-рассогласования» (impedance mismatch) между объектной и реляционной моделями. Объектная модель оперирует понятиями классов, объектов, наследования, инкапсуляции и полиморфизма. Реляционная модель основана на таблицах, строках, столбцах, первичных и внешних ключах, нормализации данных.

Ключевые функции ORM включают:

1) Отображение классов на таблицы базы данных (mapping) – определение соответствия между структурой класса и схемой таблицы.

2) Преобразование объектов в записи и обратно (serialization/deserialization) – автоматическое создание объектов из данных БД и сохранение объектов в БД.

3) Генерация SQL-запросов на основе операций с объектами – формирование SELECT, INSERT, UPDATE, DELETE запросов.

4) Управление связями между объектами – реализация отношений один-к-одному, один-ко-многим, многие-ко-многим.

5) Отслеживание изменений объектов (dirty checking) – определение изменённых атрибутов для оптимизации UPDATE-запросов.

1.2 Преимущества и недостатки использования ORM

Преимущества использования ORM:

1) Повышение продуктивности разработки – разработчики работают с привычными объектами вместо написания SQL-запросов. По оценкам, использование ORM сокращает объём кода для работы с БД на 30-50%.

2) Независимость от СУБД – код приложения не привязан к конкретной базе данных. Это позволяет легко мигрировать между PostgreSQL, MySQL, SQLite, Oracle и другими СУБД, меняя лишь строку подключения.

3) Безопасность – ORM автоматически защищает от SQL-инъекций, экранируя пользовательский ввод через параметризованные запросы. Это устраняет один из наиболее распространённых векторов атак (OWASP Top 10).

4) Сопровождаемость кода – объектная модель данных более наглядна и понятна, чем набор SQL-запросов. Изменения в модели данных локализованы в одном месте.

Недостатки использования ORM:

1) Потеря производительности – дополнительный слой абстракции создаёт накладные расходы. Сгенерированные SQL-запросы могут быть неоптимальными по сравнению с ручной оптимизацией.

2) Ограниченность возможностей – сложные SQL-конструкции (оконные функции, CTE, специфичные для СУБД функции) могут быть недоступны или сложны в реализации через ORM.

3) Кривая обучения – для эффективного использования ORM необходимо понимать как его внутреннее устройство, так и принципы работы с базами данных. «Leaky abstraction» – абстракция протекает, и знание SQL остаётся необходимым.

1.3 Типичные проблемы при работе с ORM

Проблема N+1 запросов – одна из наиболее частых проблем производительности. Возникает при загрузке коллекции объектов с последующим обращением к связанным объектам: 1 запрос для получения списка + N запросов для каждого связанного объекта. Решение – использование eager loading (prefetch_related, select_related в Django; joinedload, subqueryload в SQLAlchemy).

Lazy loading и отложенная загрузка – связанные объекты загружаются только при обращении к ним. Это может приводить к неожиданным запросам к БД и ошибкам при работе вне контекста сессии.

Управление транзакциями – неправильное управление границами транзакций может приводить к проблемам целостности данных, deadlock'ам и утечкам соединений.

2 Паттерн Active Record и Django ORM

2.1 Описание паттерна Active Record

Active Record – это архитектурный паттерн, описанный Мартином Фаулером в книге «Patterns of Enterprise Application Architecture» [Fowler, 2002]. Суть паттерна заключается в том, что объект предметной области инкапсулирует как данные (атрибуты), так и поведение, связанное с их сохранением в базе данных (CRUD-операции).

В паттерне Active Record существует прямое соответствие: каждый класс модели соответствует таблице в базе данных, каждый атрибут класса – столбцу таблицы, а каждый экземпляр класса – строке в этой таблице. Класс модели содержит методы для выполнения CRUD-операций: save(), delete(), find(), all() и другие.

Ключевые характеристики паттерна Active Record:

1) Объект «знает», как сохранить себя в базу данных – методы персистенции являются частью модели.

2) Бизнес-логика и логика доступа к данным находятся в одном классе – нарушение принципа единственной ответственности (SRP).

3) Простота использования – интуитивно понятный API, минимальный boilerplate-код.

4) Тесная связь между моделью и схемой базы данных – изменения в БД требуют изменений в модели.

2.2 Архитектура Django ORM

Django ORM является классической реализацией паттерна Active Record в Python. Модели Django наследуются от класса django.db.models.Model и автоматически получают методы для работы с базой данных. Django ORM был разработан с акцентом на простоту и продуктивность в соответствии с философией Django – «The web framework for perfectionists with deadlines».

Ключевые компоненты архитектуры Django ORM:

Model – базовый класс для всех моделей. Определяет метаинформацию, методы сохранения и удаления, а также взаимодействие с Manager.

Field – описывает тип данных столбца: CharField, IntegerField, ForeignKey, ManyToManyField и др. Каждое поле знает, как конвертировать Python-значение в SQL и обратно.

Manager – интерфейс для выполнения запросов к таблице. По умолчанию доступен как Model.objects. Можно создавать кастомные менеджеры с предопределёнными фильтрами.

QuerySet – ленивый набор объектов, представляющий SQL-запрос. QuerySet не выполняется до момента итерации или вызова методов, требующих результат (list(), len(), bool()).

2.3 Работа со связями в Django ORM

Django ORM поддерживает три типа связей между моделями:

ForeignKey (один-ко-многим) – создаёт связь «многие к одному». В базе данных создаётся столбец с внешним ключом. Обратная связь доступна через related_name или имя_модели_set.

OneToOneField (один-к-одному) – аналогичен ForeignKey с unique=True. Используется для расширения моделей (например, профиль пользователя).

ManyToManyField (многие-ко-многим) – создаёт промежуточную таблицу для связи. Параметр through позволяет определить кастомную промежуточную модель с дополнительными полями.

Для оптимизации загрузки связанных объектов Django предоставляет методы select_related() для JOIN-запросов (ForeignKey, OneToOne) и prefetch_related() для отдельных запросов с последующим объединением в Python (ManyToMany, обратные связи).

3 Паттерн Data Mapper и SQLAlchemy

3.1 Описание паттерна Data Mapper

Data Mapper – это архитектурный паттерн, также описанный Мартином Фаулером [Fowler, 2002]. В отличие от Active Record, Data Mapper разделяет объекты предметной области и слой сохранения данных. Объекты предметной области «не знают» о существовании базы данных – за преобразование и сохранение данных отвечает отдельный компонент (маппер).

Паттерн Data Mapper реализует принцип единственной ответственности (Single Responsibility Principle) из SOLID: классы предметной области

отвечают только за бизнес-логику, а за сохранение данных отвечает отдельный слой (Repository, Unit of Work).

Ключевые характеристики паттерна Data Mapper:

1) Полное разделение бизнес-логики и логики доступа к данным – чистая архитектура.

2) Объекты предметной области являются POPO (Plain Old Python Objects) – не зависят от фреймворка.

3) Маппинг между объектами и таблицами определяется отдельно от классов моделей – гибкость конфигурации.

4) Улучшенная тестируемость – модели можно тестировать без базы данных.

3.2 Архитектура SQLAlchemy

SQLAlchemy – мощная и гибкая ORM-библиотека для Python, созданная Майком Байером в 2006 году. SQLAlchemy позиционируется как «SQL toolkit and Object-Relational Mapper» и состоит из двух основных компонентов:

SQLAlchemy Core – низкоуровневый SQL-конструктор, позволяющий создавать SQL-запросы программно с полным контролем над генерируемым SQL. Core работает напрямую с таблицами и столбцами без абстракции объектов.

SQLAlchemy ORM – высокоуровневый объектно-реляционный маппер, построенный поверх Core. Реализует паттерн Data Mapper, хотя поддерживает и декларативный стиль, похожий на Active Record.

SQLAlchemy поддерживает два стиля определения моделей: классический (императивный) маппинг, где класс и таблица определяются отдельно, и декларативный маппинг (Declarative), где определение происходит в одном классе. Начиная с версии 2.0, рекомендуется использовать Declarative с type annotations.

3.3 Unit of Work и Identity Map

SQLAlchemy реализует два важных паттерна, описанных Фаулером:

Unit of Work – паттерн, реализованный через объект Session. Session отслеживает все изменения загруженных объектов (new, dirty, deleted) и при вызове commit() генерирует и выполняет минимальный набор SQL-запросов для синхронизации состояния объектов с базой данных. Это обеспечивает транзакционность и оптимизацию количества запросов.

Identity Map – гарантирует, что для каждой записи в базе данных (идентифицируемой по первичному ключу) в рамках одной сессии существует только один объект Python. Повторный запрос записи вернёт тот же объект из кэша сессии, а не создаст новый. Это предотвращает проблемы согласованности данных.

4 Сравнительный анализ подходов

4.1 Критерии сравнения

Проведём сравнительный анализ Django ORM и SQLAlchemy по ключевым критериям:

Простота освоения. Django ORM значительно проще в освоении благодаря интуитивному API, отличной документации и принципу «batteries included». SQLAlchemy требует более глубокого понимания концепций ORM, SQL и имеет более крутую кривую обучения, но предоставляет больше контроля.

Гибкость. SQLAlchemy значительно более гибка: поддерживает работу с существующими (legacy) базами данных, сложные маппинги,

множественное наследование, композитные первичные ключи, полиморфные связи. Django ORM более ограничен и следует принципу «convention over configuration».

Производительность. SQLAlchemy предоставляет больше возможностей для оптимизации благодаря доступу к низкоуровневому Core API и возможности писать raw SQL. Django ORM генерирует достаточно эффективные запросы, но имеет меньше возможностей для тонкой настройки.

Тестируемость. Благодаря разделению ответственности, модели SQLAlchemy легче тестировать изолированно – они являются обычными Python-классами. В Django ORM модели тесно связаны с базой данных, что требует использования тестовой БД даже для unit-тестов.

Экосистема и интеграции. Django ORM является частью фреймворка Django и тесно интегрирован с Django Admin, Forms, Auth, REST Framework. SQLAlchemy – самостоятельная библиотека, используемая с Flask, FastAPI, Pyramid, Starlette и другими фреймворками.

4.2 Оптимизация запросов

Оптимизация запросов – критически важный аспект работы с ORM. Обе библиотеки предоставляют инструменты для решения проблемы N+1 и оптимизации загрузки данных:

В Django ORM: select_related() выполняет JOIN для ForeignKey и OneToOneField; prefetch_related() выполняет отдельный запрос для ManyToMany и обратных связей; Prefetch() позволяет настроить фильтрацию и сортировку при prefetch; only() и defer() для выборки только нужных полей; annotate() и aggregate() для агрегатных функций.

В SQLAlchemy: joinedload() – аналог select_related(), использует JOIN; subqueryload() – загрузка через подзапрос; selectinload() – загрузка через IN;

raiseload() – запрет lazy loading, вызывает исключение; load_only() для выбора полей; with_expression() для добавления вычисляемых полей.

4.3 Рекомендации по выбору подхода

Выбор между Django ORM и SQLAlchemy зависит от требований и контекста проекта:

Django ORM рекомендуется для:

1) Проектов, использующих Django-фреймворк — максимальная интеграция.

2) Быстрого прототипирования и MVP – минимум конфигурации.

3) CRUD-приложений с простой или средней предметной областью.

4) Команд с небольшим опытом работы с ORM и SQL.

SQLAlchemy рекомендуется для:

1) Проектов со сложной предметной областью, применяющих DDD (Domain-Driven Design).

2) Работы с существующими (legacy) базами данных со сложной схемой.

3) Микросервисной архитектуры с Flask, FastAPI или асинхронными фреймворками.

4) Проектов, требующих максимальной гибкости, контроля над SQL и высокой производительности.

ЗАКЛЮЧЕНИЕ

В ходе написания реферата был проведён сравнительный анализ двух фундаментальных подходов к реализации ORM – паттернов Active Record и Data Mapper – на примере их реализации в Django ORM и SQLAlchemy соответственно.

Показано, что паттерн Active Record, реализованный в Django ORM, обеспечивает простоту и скорость разработки за счёт объединения логики данных и бизнес-логики в одном классе. Этот подход идеально подходит для CRUD-приложений, прототипов и проектов с простой предметной областью, где скорость разработки важнее гибкости архитектуры.

Паттерн Data Mapper, реализованный в SQLAlchemy, обеспечивает строгое разделение ответственности между объектами предметной области и слоем сохранения данных. Реализация паттернов Unit of Work и Identity Map обеспечивает транзакционность и согласованность данных. Это делает SQLAlchemy предпочтительным выбором для сложных проектов, требующих гибкости, тестируемости и следования принципам чистой архитектуры.

Были детально рассмотрены типичные проблемы при работе с ORM, включая проблему N+1 запросов, и методы их решения в обеих библиотеках. Проанализированы критерии сравнения: простота освоения, гибкость, производительность, тестируемость и экосистема.

Сформулированы практические рекомендации по выбору ORM-решения в зависимости от требований проекта. Таким образом, выбор между Active Record и Data Mapper – это не вопрос «лучше или хуже», а вопрос соответствия инструмента поставленной задаче. Понимание принципов работы обоих паттернов позволяет разработчику принимать обоснованные архитектурные решения.

СПИСОК ЛИТЕРАТУРЫ

1. Fowler, M. Patterns of Enterprise Application Architecture / M. Fowler. – Addison-Wesley Professional, 2002. – 560 p. ISBN 978-0321127426.
2. Django Software Foundation. Django Documentation [Электронный ресурс]. – URL: https://docs.djangoproject.com/ (дата обращения: 15.12.2025).
3. SQLAlchemy. SQLAlchemy Documentation [Электронный ресурс]. – URL: https://docs.sqlalchemy.org/ (дата обращения: 15.12.2025).
4. Copeland, R. Essential SQLAlchemy: Mapping Python to Databases / R. Copeland. – O'Reilly Media, 2015. – 232 p. ISBN 978-1491916544.
5. Percival, H. Architecture Patterns with Python / H. Percival, B. Gregory. – O'Reilly Media, 2020. – 304 p. ISBN 978-1492052203.
6. Martin, R. Clean Architecture: A Craftsman's Guide to Software Structure and Design / R. Martin. – Prentice Hall, 2017. – 432 p. ISBN 978-0134494166.
7. Evans, E. Domain-Driven Design: Tackling Complexity in the Heart of Software / E. Evans. – Addison-Wesley Professional, 2003. – 560 p. ISBN 978-0321125217.
8. JetBrains. State of Developer Ecosystem 2023 [Электронный ресурс]. – URL: https://www.jetbrains.com/lp/devecosystem-2023/ (дата обращения: 15.12.2025).
9. Real Python. Django ORM vs SQLAlchemy [Электронный ресурс]. – URL: https://realpython.com/tutorials/databases/ (дата обращения: 15.12.2025).
10. Python Software Foundation. PEP 249 — Python Database API Specification [Электронный ресурс]. – URL: https://peps.python.org/pep-0249/ (дата обращения: 15.12.2025).
11. PostgreSQL Global Development Group. PostgreSQL Documentation [Электронный ресурс]. – URL: https://www.postgresql.org/docs/ (дата обращения: 15.12.2025).
12. Alembic. Alembic Documentation [Электронный ресурс]. – URL: https://alembic.sqlalchemy.org/ (дата обращения: 15.12.2025).

Ссылка на Github: https://github.com/AIS-436/Mingazova_Regina_Maratovna_11