УДК 004.415.2:004.65

Низамиев А. Р., магистрант, 2 курс, кафедра программной инженерии, Казанский (Приволжский) федеральный университет, г. Казань

ЭМПИРИЧЕСКИЙ АНАЛИЗ АНТИПАТТЕРНОВ ПРОИЗВОДИТЕЛЬНОСТИ В SQLALCHEMY ORM-ПРИЛОЖЕНИЯХ НА ПРИМЕРЕ OPEN-SOURCE ПРОЕКТОВ

Аннотация

В статье представлен эмпирический анализ антипаттернов производительности в Python-приложениях, использующих SQLAlchemy ORM и FastAPI, на материале open-source проектов. Цель исследования состоит в выявлении наиболее распространенных классов неэффективного ORM-кода и оценке применимости специализированного инструмента анализа к реальным репозиториям. В качестве исследовательского инструмента использован разработанный программный комплекс `fastapi_sqlalchemy_detector`, сочетающий статический анализ исходного кода с возможностью гибридной интерпретации результатов по данным выполнения тестов. Эксперимент включал два этапа. На первом этапе был выполнен статический анализ 21 GitHub-проекта на стеке FastAPI + SQLAlchemy. На втором этапе проведена гибридная проверка 4 проектов, для которых удалось воспроизвести pytest- и интеграционные тесты. В статическом корпусе получено 76 срабатываний, причем признаки потенциально неэффективного ORM-кода обнаружены в 13 проектах из 21, а 37 находок удалось локализовать до уровня конкретного маршрута API. Наиболее частыми классами проблем стали неограниченная материализация результатов через `all()` без ограничения, риски ленивой загрузки при сериализации ответа и избыточная загрузка полной ORM-сущности. В гибридной выборке выполнено 168 тестов, собрано 230 трасс выполнения и 2242 SQL-запроса; дополнительно выявлено 116 runtime-находок. Полученные результаты показывают, что антипаттерны ORM-производительности являются систематически встречающимся явлением в реальном FastAPI + SQLAlchemy-коде, а гибридный анализ позволяет не только выявлять потенциальные риски, но и уточнять их практическую значимость.

Ключевые слова: SQLAlchemy, FastAPI, ORM, производительность, антипаттерны, эмпирический анализ, статический анализ, гибридный анализ, open-source проекты.

Empirical Analysis of Performance Anti-Patterns in SQLAlchemy ORM Applications on Open-Source Projects

Nizamiev A. R. Kazan Federal University, Kazan, Russia

Abstract

The paper presents an empirical analysis of performance anti-patterns in Python applications built with SQLAlchemy ORM and FastAPI using open-source projects as the study corpus. The goal is to identify the most common classes of inefficient ORM code and to assess the applicability of a

specialized analysis tool to real repositories. The study relies on the `fastapi_sqlalchemy_detector` tool, which combines static source-code analysis with hybrid interpretation based on runtime test data. The experiment consisted of two stages. First, a static scan was executed on 21 GitHub projects using the FastAPI + SQLAlchemy stack. Second, a hybrid evaluation was conducted on 4 projects for which pytest and integration tests could be reproduced. The static corpus produced 76 findings; signs of potentially inefficient ORM code were found in 13 of 21 projects, and 37 findings were localized to specific API routes. The most frequent problem classes were unbounded `all()` materialization, risks of lazy loading during response serialization, and overfetching of full ORM entities. In the hybrid subset, 168 tests were executed, 230 runtime traces and 2242 SQL queries were collected; 116 additional runtime findings were captured. The results show that ORM-related performance anti-patterns systematically occur in real FastAPI + SQLAlchemy codebases, while hybrid analysis helps not only detect potential risks but also refine their practical significance.

Keywords: SQLAlchemy, FastAPI, ORM, performance, anti-patterns, empirical analysis, static analysis, hybrid analysis, open-source projects.

Введение

Вопрос производительности ORM-кода остается актуальным для современных backend-приложений, поскольку объектно-реляционное отображение, упрощая разработку, одновременно скрывает реальную стоимость обращений к базе данных [1, 2, 6]. В Python-экосистеме эта проблема особенно заметна в проектах на стеке FastAPI + SQLAlchemy, где высокоуровневый код может маскировать повторяющиеся запросы, избыточную материализацию данных и транзакционные накладные расходы [9–12].

Большая часть существующих исследований по ORM-антипаттернам посвящена Java/Hibernate-контексту или рассматривает проблему в более общем виде [3–5, 7]. Для SQLAlchemy-приложений с FastAPI представляют интерес не только формальные признаки кода, но и их распределение в реальных open-source репозиториях: какие классы проблем встречаются чаще, насколько они привязаны к маршрутам API и в какой степени могут быть подтверждены по данным выполнения тестов.

Целью настоящей статьи является эмпирическая оценка распространенности антипаттернов производительности в SQLAlchemy ORM-приложениях на материале open-source проектов. В отличие от статьи, фокусирующейся на самом методе анализа, в данной работе основной акцент делается на корпусе проектов, количественных результатах, типовых классах находок и практической интерпретации полученных данных.

Постановка исследования

В рамках исследования решались следующие задачи:

1. Сформировать корпус open-source проектов на стеке FastAPI + SQLAlchemy.
2. Оценить распределение статических находок по проектам, классам правил и уровням серьезности.
3. Проанализировать, какие из находок удается локализовать до конкретного маршрута API.

4. Выполнить гибридную проверку на подвыборке проектов с воспроизводимыми тестами.
5. Сформулировать выводы о типичных классах ORM-проблем и практической применимости анализа.

В качестве инструмента исследования использован `fastapi_sqlalchemy_detector`, выполняющий статический анализ Python AST и, при наличии runtime-трасс, гибридную интерпретацию находок на основе SQLAlchemy-событий во время выполнения тестов.

Корпус проектов и методика

Эксперимент был организован в два этапа.

На первом этапе использован корпус из 21 open-source GitHub-проекта на стеке FastAPI + SQLAlchemy. Для каждого проекта выполнялся статический анализ исходного кода с поиском типовых ORM-антипаттернов производительности. К числу таких антипаттернов относились запросы в циклах, неограниченная материализация через `all()`, риски ленивой загрузки при сериализации ответа, избыточная загрузка сущностей и поэлементные операции записи.

На втором этапе была выделена подвыборка из 4 проектов, для которых удалось воспроизвести существующие pytest- и интеграционные тесты. Для этих проектов собирались runtime-трассы, содержащие выполненные SQL-запросы, их типы, повторяемость SQL-шаблонов и события `commit` и `flush`. Далее эти данные сопоставлялись со статическими находками.

Такая методика позволяет одновременно рассматривать две перспективы. Первая отражает распространенность потенциально проблемных конструкций в исходном коде. Вторая показывает, какие из них проявляются в тестовом выполнении и какие маршруты остаются вне зоны подтверждения из-за отсутствия покрытия.

Результаты статического анализа

Статический анализ 21 проекта дал 76 срабатываний. При этом признаки потенциально неэффективного ORM-кода были обнаружены в 13 проектах, тогда как 8 проектов не содержали находок. Для 37 случаев удалось установить связь с конкретным маршрутом API. По уровням серьезности распределение составило 70 срабатываний уровня `medium` и 6 уровня `high`.

Ключевые количественные результаты статического этапа приведены в таблице 1.

Показатель	Значение
Проектов в корпусе	21
Проектов с находками	13
Проектов без находок	8
Статических находок	76

Показатель	Значение
Endpoint-linked findings	37
High severity	6
Medium severity	70

Таким образом, признаки потенциально неэффективного ORM-кода были обнаружены в 61,9% исследованного корпуса, а почти половина находок оказалась привязана к конкретному HTTP-маршруту. Это делает результаты анализа пригодными не только для общего аудита репозитория, но и для прикладной работы по локализации проблемных endpoint-ов.

Распределение находок по правилам показывает, что исследуемый корпус имеет выраженный перекос в сторону нескольких устойчивых классов проблем.

Правило	Количество
`unbounded_result_all`	47
`response_model_without_eager_loading`	9
`overfetching_full_entity`	7
`session_write_in_loop`	5
`query_in_loop`	3
`relationship_access_without_eager_loading`	2
`refresh_after_flush`	1
`query_helper_in_loop`	1
`all_then_slice`	1

Доминирование правила `unbounded_result_all` особенно показательно. Оно указывает, что для реальных FastAPI + SQLAlchemy-проектов одной из наиболее типичных проблем является не сложный низкоуровневый SQL-дефект, а регулярное отсутствие явного ограничения результата при materialization-операциях. Следовательно, значимая часть потенциальных проблем производительности связана с повседневными практиками использования ORM, а не с редкими экстремальными случаями.

Наиболее проблемные проекты и классы находок

Отдельный интерес представляет распределение находок по проектам. Наибольшее число срабатываний было получено для проектов `Social-Media-Api` (20), `fastapi_realworld` (13), `social_network_fastAPI` (11), `fastapi-best-architecture` (10) и `fastapi-realworld-example-app` (8).

Такое распределение показывает, что антипаттерны производительности концентрируются неравномерно. Часть проектов демонстрирует сравнительно чистую картину, тогда как в другой части корпуса наблюдается накопление сразу нескольких классов проблем: неограниченная материализация, риски ленивой загрузки и поэлементные операции сессии.

С практической точки зрения это важно по двум причинам. Во-первых, наличие большого числа находок в нескольких проектах подтверждает, что исследуемые паттерны не являются искусственно сконструированными. Во-вторых, концентрация разных типов срабатываний в одном репозитории может указывать на устойчивые архитектурные или стилевые особенности, влияющие на производительность целых подсистем, а не отдельных строк кода.

Результаты гибридной проверки

На подвыборке из 4 проектов проведена гибридная проверка с использованием runtime-трасс. В сумме были выполнены 168 pytest-тестов, собраны 230 трасс выполнения и 2242 SQL-запроса. В той же выборке статический слой обнаружил 21 находку, а слой анализа во время выполнения выявил 116 runtime-находок.

Итоговые показатели гибридного этапа приведены в таблице 2.

Показатель	Значение
Проектов в гибридной выборке	4
Выполненных pytest-тестов	168
Трасс выполнения	230
SQL-запросов	2242
Статических находок	21
Runtime-находок	116
`runtime confirmed`	3
`runtime not executed`	13
`static only`	5

Наиболее существенный вывод гибридного этапа состоит в том, что он позволяет качественно уточнить интерпретацию статических сигналов. Для 16 из 21 статической находки гибридный слой дал дополнительную информацию: либо подтверждение по фактическому SQL-поведению, либо явное указание на то, что соответствующий маршрут не был покрыт тестами. В инженерной практике это важнее, чем простое деление на “подтвердилось” и “не подтвердилось”, поскольку позволяет отделить ограничение тестового покрытия от содержательной оценки самой находки.

Показательный пример

Характерный пример был получен на проекте `Social-Media-Api`. Для маршрута `GET /users/{user_id}/followers` статический анализ обнаружил вызов `all()` без `limit/offset`. Runtime-трасса, собранная во время выполнения тестов, подтвердила факт выполнения `SELECT` без ограничения результата. Тем самым одна и та же проблема была зафиксирована сразу на двух уровнях: как потенциально рискованная конструкция исходного кода и как реально проявившийся SQL-паттерн.

Этот пример показывает, что гибридный подход особенно полезен не только для поиска проблем, но и для их объяснения. Статическая находка сама по себе дает локализацию и рекомендацию, а подтверждение по данным выполнения позволяет воспринимать ее как более приоритетную в контексте оптимизации.

Обсуждение

Полученные результаты позволяют сформулировать несколько содержательных выводов.

Во-первых, антипаттерны ORM-производительности являются систематически встречающимся явлением в реальном FastAPI + SQLAlchemy-коде. Даже при сравнительно небольшом корпусе в 21 проект признаки потенциально неэффективного поведения обнаруживаются в большинстве репозиториев.

Во-вторых, преобладают не редкие узкоспециальные дефекты, а повторяющиеся паттерны повседневного использования ORM: `all()` без ограничения, отсутствие явной предварительной загрузки связей и избыточная материализация сущностей. Это означает, что значительный эффект может дать не только глубокая низкоуровневая оптимизация, но и систематическое улучшение инженерных практик на уровне application-кода.

В-третьих, гибридный этап показывает высокую ценность сочетания статического и динамического анализа. Runtime-находки дополняют картину фактическими симптомами SQL/session-поведения, а статический слой обеспечивает локализацию и объяснение. В совокупности это позволяет переходить от простого списка предупреждений к более обоснованной приоритизации найденных проблем.

Наконец, важным побочным результатом является возможность использовать гибридный анализ как индикатор покрытия тестами производственно значимых маршрутов. Статус `runtime not executed` в данном случае выступает не только ограничением исследования, но и практическим сигналом для дальнейшего развития тестового контура.

Заключение

В статье представлен эмпирический анализ антипаттернов производительности в SQLAlchemy ORM-приложениях на материале open-source FastAPI + SQLAlchemy-проектов. Статический этап исследования показал, что признаки потенциально неэффективного ORM-кода обнаруживаются в 13 из 21 проекта, а наиболее частыми классами проблем являются неограниченная материализация результатов, риски ленивой загрузки и избыточная загрузка сущностей.

Гибридная проверка на подвыборке проектов с воспроизводимыми тестами показала, что специализированный анализатор способен не только выявлять потенциальные риски, но и дополнять их интерпретацию по данным выполнения тестов. Это повышает практическую ценность результатов и делает такой подход перспективным для дальнейшего применения в задачах диагностики, код-ревью и инженерного контроля качества.

Список литературы

1. Chen T.-H., Shang W., Jiang Z. M., Hassan A. E., Nasser M., Flora P. Detecting Performance Anti-patterns for Applications Developed using Object-relational Mapping // Proceedings of the 36th International Conference on Software Engineering. 2014. P. 1001-1012. DOI: 10.1145/2568225.2568259.
2. Chen T.-H., Shang W., Jiang Z. M., Hassan A. E., Nasser M. N., Flora P. Finding and Evaluating the Performance Impact of Redundant Data Access for Applications that are Developed Using Object-Relational Mapping Frameworks // IEEE Transactions on Software Engineering. 2016. Vol. 42, No. 12. P. 1148-1161. DOI: 10.1109/TSE.2016.2553039.
3. Huang Z., Shao Z., Fan G., Yu H., Yang K., Zhou Z. HBSniff: A static analysis tool for Java Hibernate object-relational mapping code smell detection // Science of Computer Programming. 2022. Vol. 217. Article 102778. DOI: 10.1016/j.scico.2022.102778.
4. Loli S., Teixeira L., Cartaxo B. A Catalog of Object-Relational Mapping Code Smells for Java // Brazilian Symposium on Software Engineering. 2020. P. 82-91. DOI: 10.1145/3422392.3422432.
5. Liu W., Chen T.-H. SLocator: Localizing the Origin of SQL Queries in Database-Backed Web Applications // IEEE Transactions on Software Engineering. 2023. Vol. 49, No. 6. P. 3376-3390. DOI: 10.1109/TSE.2023.3253700.
6. Torres A., Galante R., Pimenta M. S., Martins A. J. B. Twenty years of object-relational mapping: A survey on patterns, solutions, and their implications on application design // Information and Software Technology. 2017. Vol. 82. P. 1-18. DOI: 10.1016/j.infsof.2016.09.009.
7. Yang J., Subramaniam P., Lu S., Yan C., Cheung A. How not to structure your database-backed web applications: a study of performance bugs in the wild // Proceedings of the 40th International Conference on Software Engineering. 2018. P. 800-810.
8. Python Documentation. `ast` — Abstract Syntax Trees [Электронный ресурс]. URL: https://docs.python.org/3/library/ast.html (дата обращения: 04.06.2026).
9. SQLAlchemy Documentation. ORM Querying Guide [Электронный ресурс]. URL: https://docs.sqlalchemy.org/20/orm/queryguide/index.html (дата обращения: 04.06.2026).
10. SQLAlchemy Documentation. Performance FAQ [Электронный ресурс]. URL: https://docs.sqlalchemy.org/20/faq/performance.html (дата обращения: 04.06.2026).
11. SQLAlchemy Documentation. Relationship Loading Techniques [Электронный ресурс]. URL: https://docs.sqlalchemy.org/20/orm/queryguide/relationships.html (дата обращения: 04.06.2026).
12. SQLAlchemy Documentation. Session Basics [Электронный ресурс]. URL: https://docs.sqlalchemy.org/20/orm/session_basics.html (дата обращения: 04.06.2026).
13. SQLAlchemy Documentation. Core Events [Электронный ресурс]. URL: https://docs.sqlalchemy.org/20/core/event.html (дата обращения: 04.06.2026).
14. FastAPI Documentation. Response Model [Электронный ресурс]. URL: https://fastapi.tiangolo.com/tutorial/response-model/ (дата обращения: 04.06.2026).

15. FastAPI Documentation. Testing [Электронный ресурс]. URL: https://fastapi.tiangolo.com/tutorial/testing/ (дата обращения: 04.06.2026).