УДК 004.43:004.65 **DOI:** 10.31673/2412-9070.2025.061204
А. С. ГАВОР, аспірант;
ORCID: 0009-0002-9705-1666
Д. О. НІЩЕМЕНКО, аспірант;
ORCID: 0009-0006-1781-4109
К. О. ГОРДІЄНКО, аспірант;
ORCID: 0009-0002-5997-9682
А. О. АРОНОВ, к.т.н. доцент,
ORCID: 0009-0000-7868-8341
Державний університет інформаційно-комунікаційних технологій, Київ

ПОРІВНЯННЯ ПРОДУКТИВНОСТІ ORM ФРЕЙМВОРКІВ PYTHON DJANGO ТА JAVA HIBERNATE

Стаття присвячена порівняльному аналізу ефективності ORM-фреймворків Django ORM (Python) та Hibernate (Java) при роботі з великомасштабними реляційними базами даних. Актуальність дослідження зумовлена потребою оптимізації взаємодії прикладного рівня з базою даних у системах із високими вимогами до продуктивності та узгодженості даних.

Метою роботи є експериментальне дослідження продуктивності, затримки, інтенсивності SQL-запитів, використання CPU та пам'яті для обох ORM у єдиному середовищі PostgreSQL. Для моделювання виконано тестові сценарії CRUD, об'єднання (joins) та агрегації з навантаженням понад 900 тисяч записів.

Результати показали, що Django ORM має перевагу в базових операціях і складних вибірках завдяки нижчому рівню абстракції та меншій кількості проміжних шарів. Hibernate, у свою чергу, демонструє стабільність і узгодженість при підвищеному транзакційному навантаженні, забезпечуючи масштабованість завдяки багаторівневій архітектурі та оптимізації JVM.

Рекомендовано використовувати Django ORM у вебзастосунках і системах швидкої розробки, а Hibernate у корпоративних і фінансових системах, де критичними є стабільність та контроль станів об'єктів.

Ключові слова: ORM, Django, Hibernate, реляційна база даних, продуктивність, латентність, транзакції, PostgreSQL, Python, Java, мови програмування.

Вступ

Розробка програмного забезпечення з використанням реляційних баз даних є ключовим елементом клієнт-серверних архітектур. Такі системи забезпечують збереження, обробку та доступ до великих обсягів інформації, становлячи основу корпоративних і веборієнтованих застосунків. Пряме використання SQL, хоча й забезпечує повний контроль над даними, вимагає значних витрат часу на ручне формування запитів і узгодження між об'єктною та реляційною моделями [1].

З метою спрощення взаємодії з базою даних застосовуються ORM-фреймворки (Object–Relational Mapping), які автоматизують відображення об'єктів у таблиці та виконання CRUD-операцій [2]. Це підвищує швидкість розробки, зменшує ризик помилок і покращує підтримуваність коду.

У цьому дослідженні буде розглянуто два поширених ORM-рішення. Django ORM орієнтований на швидку розробку з мінімальною конфігурацією [3], тоді як Hibernate є промисловим стандартом Java-екосистеми [4], оптимізованим для високонавантажених корпоративних систем [5].

Попри спільну концепцію об’єктно-реляційного відображення, фреймворки відрізняються архітектурно, що впливає на продуктивність і ефективність використання ресурсів.

Огляд технологій

Django ORM є невід’ємною частиною вебфреймворку Django та реалізує принцип «конфігурація за замовчуванням» [3]. Він дозволяє описувати структуру бази даних за допомогою Python-класів, де кожен клас відповідає таблиці, а атрибути – стовпцям. ORM автоматично формує SQL-запити для операцій створення, читання, оновлення й видалення (CRUD). Основний акцент робиться на простоту використання та швидкий старт без складних налаштувань. Django ORM також забезпечує підтримку транзакцій (автоматичних або з ручним керуванням через atomic) і внутрішнє кешування на рівні одного запиту [6]. Важливою особливістю є підтримка механізму Lazy Loading для зв’язків типу Foreign Key та Many-To-Many.

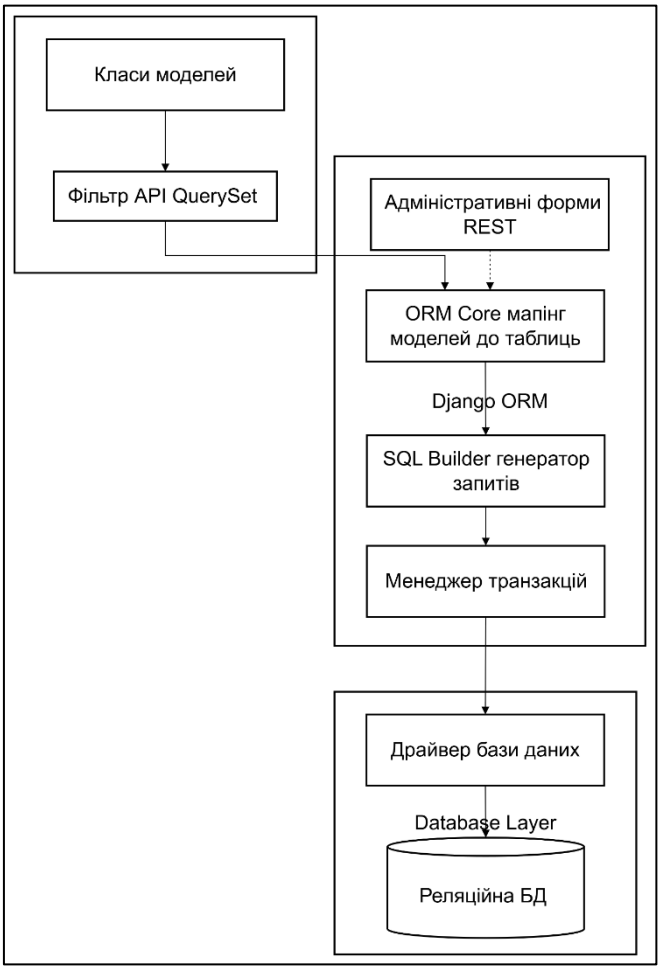

Рис 1. Архітектурна схема Django ORM

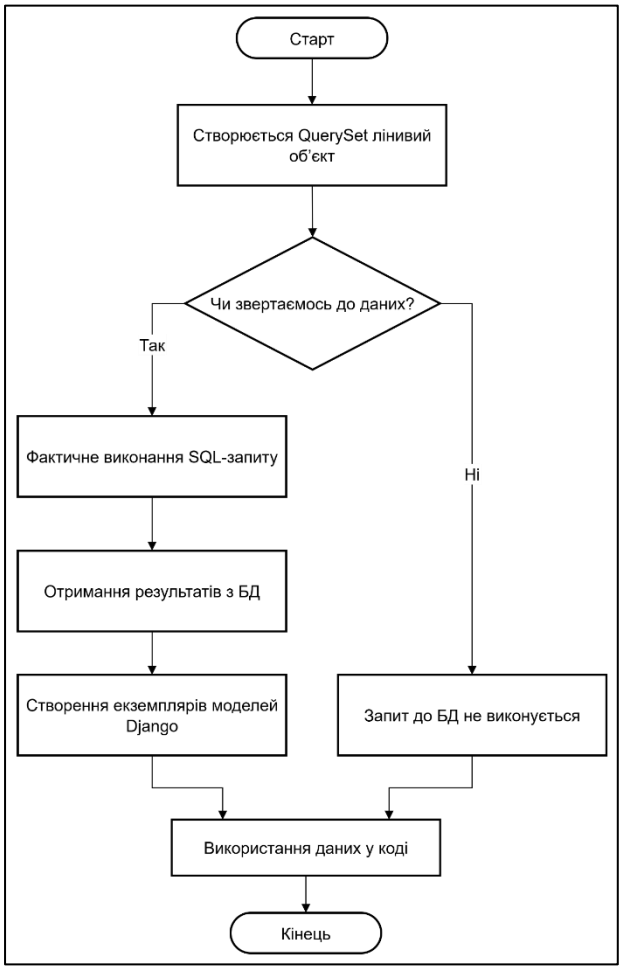

Рис 2. Блок-схема лінивого завантаження Django ORM

Hibernate – ORM-фреймворк у Java-екосистемі [4], який давно утвердився як стандарт для ентерпрайз застосунків. Його ключовою особливістю є гнучке відображення об’єктної моделі на реляційну схему за допомогою анотацій або XML-конфігурацій [4]. Hibernate підтримує складні зв’язки між сутностями, оптимізацію SQL-запитів, механізми кешування першого та другого рівнів, а також детально налаштовані транзакції з інтеграцією різних менеджерів JTA, JDBC [4]. У контексті Lazy Loading Hibernate використовує проксі-об’єкти або байткод-енхансмент, що дозволяє гнучко налаштовувати відкладене завантаження як для сутностей, так і для колекцій.

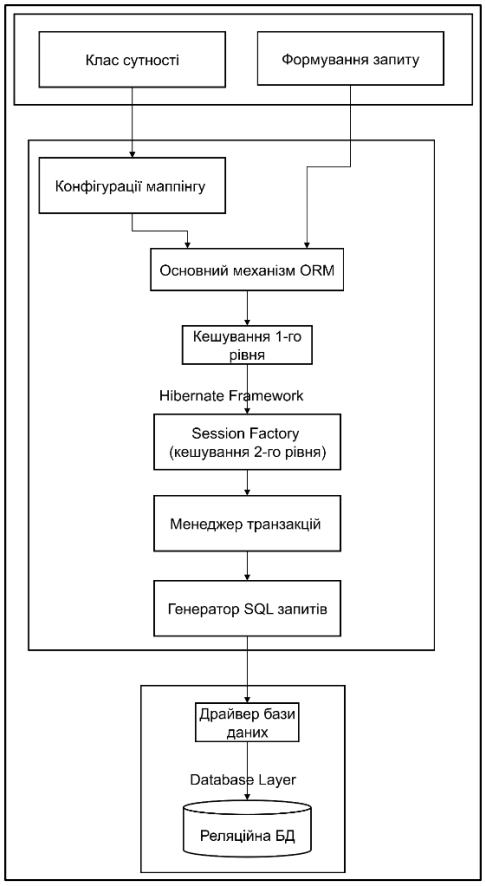

Рис 3. Архітектурна схема Hibernate

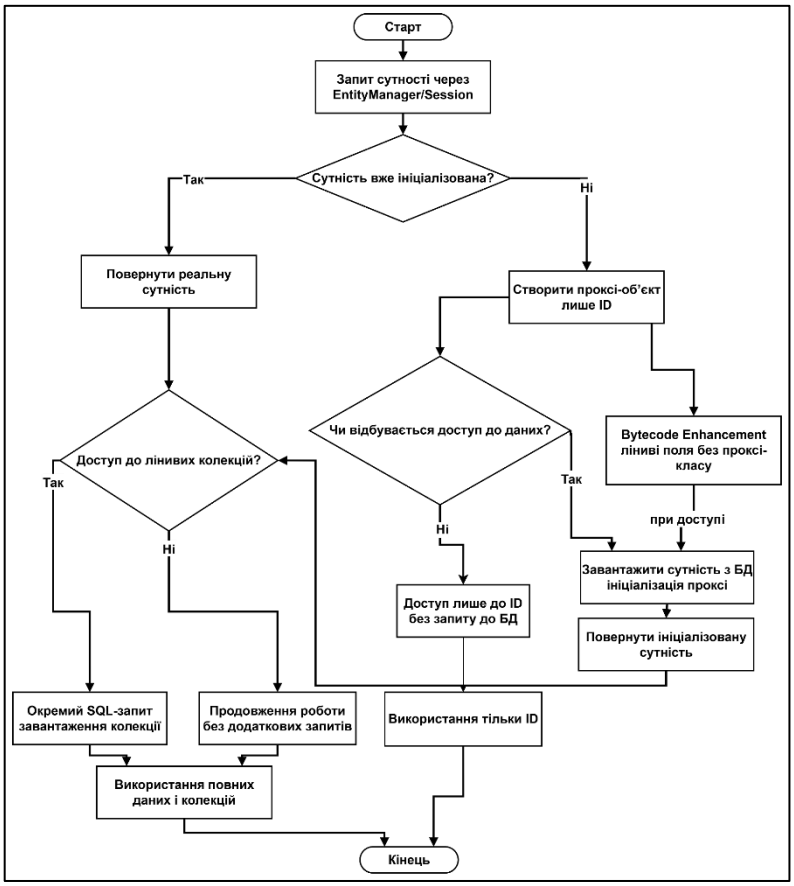

Рис 4. Блок-схема лінивого завантаження Hibernate

Огляд продемонстрував, що обидва ORM-фреймворки реалізують спільну мету, забезпечення об'єктно-реляційного відображення з мінімальними витратами розробника, проте підходять до цього через різні архітектурні принципи.

Дослідження

Основною метою дослідження є порівняння ефективності ORM-фреймворків у єдиному середовищі бази даних, щоб усунути вплив сторонніх чинників. Для цього використано PostgreSQL 15 як сучасне та стабільне рішення, що однаково підтримується в екосистемах Python (Django ORM) і Java (Hibernate) [7].

Для уніфікації середовища виконання застосовано Python 3.11 із Django ORM 5.0 та Java 17 із Hibernate 6.4.4.Final [5, 6]. Усі компоненти ізольовано за допомогою контейнеризації Docker, що гарантує повторюваність експериментів і мінімізує зовнішні впливи [8].

Керування транзакціями здійснюється локальними засобами кожного ORM. Оскільки експеримент охоплює лише одне джерело даних PostgreSQL, використання JTA є недоцільним, розподілені транзакції не потрібні, а зовнішній менеджер лише збільшує накладні витрати [4, 6]. Такий підхід спрощує конфігурацію середовища та забезпечує зіставність часових характеристик між Django ORM і Hibernate.

Для моделювання типових сценаріїв роботи з даними буде створено набір таблиць із різними видами зв'язків: «entity» (головна сутність), «entity_details» (one-to-one), «orders» (one-to-many), «tags» і «entity_tags» (many-to-many) [1]. Це дало змогу відтворити реалістичну структуру корпоративних і вебзастосунків, де поєднуються прості й складні відносини між об'єктами.

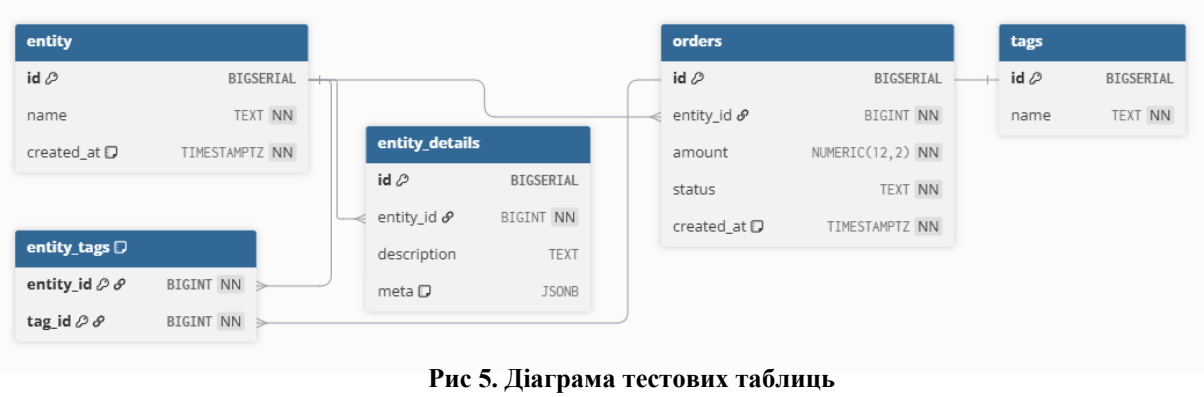

Рис 5. Діаграма тестових таблиць

Для забезпечення методологічної симетрії моделі даних в обох середовищах реалізовано єдині логічні схеми. У Django ORM мапінг здійснюється через класи моделей, де атрибути визначають поля таблиць, а зв'язки реалізовано через зовнішні ключі та відповідні типи відношень.

Код мапінгу мовою Python:

```
class Entity(models.Model):
    name = models.TextField()
    created_at = models.DateTimeField()
    class Meta:
        db_table = "entity"

class EntityDetails(models.Model):
    entity = models.OneToOneField(Entity, on_delete=models.CASCADE, related_name="details")
    description = models.TextField(null=True)
    meta = models.JSONField(default=dict)
    class Meta:
        db_table = "entity_details"

class OrderItem(models.Model):
    entity = models.ForeignKey(Entity, on_delete=models.CASCADE, related_name="orders")
    amount = models.DecimalField(max_digits=12, decimal_places=2)
    status = models.TextField()
    created_at = models.DateTimeField()
    class Meta:
        db_table = "orders"

class Tag(models.Model):
    name = models.TextField(unique=True)
    class Meta:
        db_table = "tags"

class EntityTag(models.Model):
    entity = models.ForeignKey(Entity, on_delete=models.CASCADE)
    tag = models.ForeignKey(Tag, on_delete=models.CASCADE)
    class Meta:
        db_table = "entity_tags"
        unique_together = ("entity", "tag")
```

У Hibernate структура відображена за допомогою JPA-анотацій, що встановлюють відповідність між класами та таблицями бази даних.

Код мапінгу мовою Java:

```
@Entity @Table(name="entity_details")
public class EntityDetails {
  @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
  public Long id;
  @OneToOne @JoinColumn(name="entity_id", unique=true)
  public EntityItem entity;
  public String description;
  @Column(columnDefinition="jsonb")
  public String meta;
}

@Entity @Table(name="entity")
public class EntityItem {
  @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
  public Long id;
  public String name;
  public OffsetDateTime created_at;
  @OneToOne(mappedBy="entity", fetch=FetchType.LAZY)
  public EntityDetails details;
  @OneToMany(mappedBy="entity", fetch=FetchType.LAZY)
  @org.hibernate.annotations.BatchSize(size=100)
  public List<OrderItem> orders = new ArrayList<>();
}

@Entity @Table(name="entity_tags")
@IdClass(EntityTagId.class)
public class EntityTag {
  @Id @ManyToOne @JoinColumn(name="entity_id")
  public EntityItem entity;
  @Id @ManyToOne @JoinColumn(name="tag_id")
  public Tag tag;
}

public class EntityTagId implements Serializable {
  public Long entity; public Long tag;
  public EntityTagId(){}
  public EntityTagId(Long e, Long t){ this.entity=e; this.tag=t; }
  @Override public boolean equals(Object o){
    if (this == o) return true;
    if (!(o instanceof EntityTagId)) return false;
    EntityTagId x = (EntityTagId) o;
    return Objects.equals(entity, x.entity) && Objects.equals(tag, x.tag);
  }
  @Override public int hashCode(){ return Objects.hash(entity, tag); }
}
```

Для наповнення бази даних використано генерацію за допомогою SQL-функцій PostgreSQL, згенеровано 100 тис. сутностей, для кожної з яких створено запис у «entity_details»,

п’ять записів у «orders» та три асоціації з тегами. У підсумку база містить понад 900 тис. рядків, що забезпечує достатнє навантаження для оцінки ефективності ORM при виконанні CRUD-операцій, відкладеного та жадібного завантаження даних, а також при використанні кешування.

Оцінювання буде відбуватися за такими критеріями ефективності:

– продуктивність – як характеристика швидкості обробки транзакцій та запитів до бази даних;

– затримка виконання – як показник часу відгуку на окрему операцію.

– інтенсивність взаємодії з базою даних – як кількість SQL-запитів, що генерує ORM у типовому сценарії;

– використання процесора – як характеристика обчислювальних ресурсів, необхідних для виконання завдань;

– використання пам’яті – як показник просторової складності, тобто максимального обсягу оперативної пам’яті, який займає процес.

Продуктивність вимірюється у вигляді кількості операцій, виконаних за секунду (Transactions per Second, TPS). Цей показник використовується блокчейн подібних систем [9], але даному контекстові він дозволить оцінити наскільки швидко ORM обробляє великі обсяги CRUD-операцій у реальному навантаженні.

$$TPS = \frac{N}{T}, \quad (1)$$

де N – кількість виконаних операцій; T – загальний час виконання у секундах.

Для оцінки часу відгуку (затримки) використовується 95-й перцентиль латентності (L95). Це дозволяє врахувати не лише середнє значення, але й «довгі хвости» – найповільніші запити, які суттєво впливають на стабільність роботи систем [10]

$$L95 = Percentile95(\{t_1, t_2, \ldots, t_N\}), \quad (2)$$

де t_i – час виконання однієї операції.

Інтенсивність SQL-запитів (SQL count) – характеристика ORM яка показує кількість SQL-запитів, що генеруються для виконання бізнес-логіки.

$$SQL_{count} = \frac{SQL_{total}}{q_i}, \quad (3)$$

де q_i – кількість SQL-запитів, що виконані в ітерації i; SQL_{total} – загальна кількість згенерованих SQL запитів.

Використання CPU – ефективність роботи оцінюється через середнє використання процесора.

$$CPU = \frac{t_{wall}}{t_{cpu} \times C} \cdot \times 100, \quad (4)$$

де t_{wall} – загальний час виконання тесту; t_{cpu} – сумарний час користувацьких і системних інструкцій процесу; C – кількість логічних ядер процесора.

Використання пам’яті – оцінки просторової складності враховується максимальний обсяг оперативної пам’яті (Resident Set Size, RSS), який реально використовував процес під час виконання тесту.

$$Mem_{max} = max(RSS_i), i = 1 \ldots N, \quad (5)$$

де RSS_i – обсяг пам’яті, зафіксований у момент виконання ітерації i.

Орієнтири для налаштувань продуктивності PostgreSQL та методики вимірювання були з офіційними рекомендаціями документації PostgreSQL та інструментом pgbench, що використовується як стандартний засіб навантажувального тестування в екосистемі PostgreSQL [11–12]

Для кожного сценарію було виконано 10000 незалежних операцій окремо для кожного ORM. Перед фіксацією показників здійснювався короткий етап попереднього прогріву з метою стабілізації JIT-компіляції, файлового кешу операційної системи та планів виконання запитів. У фінальних прогонах параметри пакетної обробки й вибірки були уніфіковані, розмір пакета для масових модифікацій встановлювався на рівні 1000, а розмір вибірки для читальних

операцій, на рівні приблизно 1000, що забезпечує методологічну симетрію між порівнюваними фреймворками. Для віртуальної машини Java фіксовано початковий і максимальний обсяг купи 512 МБ, що дозволило стабілізувати пікові значення резидентного обсягу пам'яті (RSS) та зробити порівняння споживання ресурсів коректним. Результати дослідження наведено для Django ORM (табл. 1) та Hibernate ORM (табл. 2).

Результати дослідження Django ORM

Операція	Продуктивність, TPS	Латентність, мс	Інтенсивність SQL-запитів	Використання CPU, %	Використання пам'яті, МБ
INSERT	3 249.73	334.34	1.001	6.49	57.0
SELECT	64 582.17	22.99	0.002	7.07	67.5
UPDATE	1 069.74	986.85	2.001	5.95	90.7
DELETE	585.99	1 813.51	5.001	6.46	67.1
BULK INSERT	29 963.80	47.55	0.002	7.16	67.2
BULK UPDATE	10 085.57	124.81	0.003	7.47	67.4
Simple Joins	37 134.15	41.04	0.002	5.78	66.6
Complex Joins	232 047.16	6.29	0.002	1.90	66.6
Aggregation	27 530.66	39.61	0.002	0.24	66.6
N+1 Problem	1 066.99	1 023.54	2.002	7.13	70.9

Результати дослідження Hibernate ORM

Операція	Продуктивність, TPS	Латентність, мс	Інтенсивність SQL-запитів	Використання CPU, %	Використання пам'яті, МБ
INSERT	3 490.60	412.07	1.000	13.20	228.6
SELECT	32 016.23	116.87	0.001	35.70	230.2
UPDATE	1 276.29	965.54	0.003	0.27	230.7
DELETE	586.25	1 809.80	0.006	0.34	232.5
BULK INSERT	3 924.87	274.66	1.000	12.85	279.6
BULK UPDATE	5 006.13	284.42	1.002	15.18	334.6
Simple Joins	166 890.78	7.39	0.001	8.29	330.5
Complex Joins	122 053.57	16.68	0.001	21.27	331.1
Aggregation	24 972.33	47.63	0.001	6.24	331.3
N+1 Problem	5 805.50	222.70	1.011	14.03	340.2

Виходячи з отриманих результатів експериментального бенчмарку (табл. 1-2), можна здійснити порівняльний аналіз ефективності двох ORM-фреймворків та сформулювати узагальнені висновки щодо їхньої продуктивності, архітектурних особливостей і доцільності використання в різних напрямках інформаційних систем.

Висновки

Як показали результати порівняльного бенчмарку двох ORM-фреймворків, Django ORM (Python) та Hibernate (Java), обидва рішення продемонстрували високу ефективність при роботі з великомасштабною базою даних. За показниками пропускної здатності та затримки, наведеною у табл. 1-2, Django ORM виявив перевагу у більшості базових CRUD-операцій, зокрема під час вибірки, об'єднання даних та пакетних модифікацій. Це пояснюється нижчим рівнем абстракції, за якого об'єктна модель Python безпосередньо транслюється у SQL-оператори з мінімальними накладними витратами на керування транзакціями.

Hibernate, навпаки, продемонстрував більш стабільні результати у сценаріях із високим транзакційним навантаженням, що підтверджується даними табл. 2. Його багаторівнева архітектура, побудована на принципах інверсії керування та автоматизованого життєвого циклу об'єктів, забезпечує підвищену узгодженість даних, контроль станів сутностей та підтримку складних зв'язків, хоча це супроводжується більшими витратами ресурсів.

Виявлені відмінності у продуктивності зумовлені мовними та архітектурними особливостями фреймворків. Django ORM, реалізований поверх інтерпретованої мови Python, характеризується гнучкістю, простотою інтеграції та високою швидкодією у типових запитах, але поступається ефективністю при обробці великих транзакцій. Hibernate, що використовує статичну типізацію Java і оптимізації віртуальної машини JVM, забезпечує кращу масштабованість і передбачувану продуктивність при довготривалих навантаженнях.

Узагальнюючи, можна стверджувати, що Django ORM доцільно застосовувати у веб-проєктах, аналітичних системах і середовищах швидкої розробки, де пріоритетом є адаптивність та швидке впровадження змін. Hibernate, своєю чергою, є оптимальним вибором для корпоративних, фінансових та промислових систем, орієнтованих на високу узгодженість даних, стабільність і масштабованість архітектури.

Список літератури

1. Kleppmann, M. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. O'Reilly Media, 2017. URL: https://www.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/

2. What is Object-Relational Mapping (ORM)? - Object-Relational Mapping Explained - AWS. Amazon Web Services, Inc. URL: https://aws.amazon.com/what-is/object-relational-mapping/ (дата звернення: 12.10.2025).

3. Django documentation | Django documentation. Django Project. URL: https://docs.djangoproject.com/en/5.0/ (дата звернення: 12.10.2025).

4. Jakarta Persistence 3.1 | Jakarta EE | The Eclipse Foundation. Jakarta® EE: The New Home of Cloud Native Java. URL: https://jakarta.ee/specifications/persistence/3.1/ (дата звернення: 12.10.2025).

5. Python 3.11 documentation. Python documentation. URL: https://docs.python.org/3.11/ (дата звернення: 12.10.2025).

6. Java Platform, Standard Edition Documentation - Releases. Oracle Help Center. URL: https://docs.oracle.com/en/java/javase/ (дата звернення: 12.10.2025).

7. PostgreSQL 15.14 Documentation. PostgreSQL Documentation. URL: https://www.postgresql.org/docs/15/ (date of access: 12.10.2025).

8. Home. Docker Documentation. URL: https://docs.docker.com/ (дата звернення: 12.10.2025).

9. Academy B. Transactions Per Second (TPS) | Binance Academy. Binance Academy. URL: https://academy.binance.com/en/glossary/transactions-per-second-tps (дата звернення: 12.10.2025).

10. 95th Percentile Calculations in the SolarWinds Platform. Documentation for SolarWinds. URL:https://documentation.solarwinds.com/en/success_center/orionplatform/content/core-95th-percentile-calculations-sw80.html (дата звернення: 12.10.2025).

11.Chapter 14. Performance Tips. PostgreSQL Documentation. URL:

https://www.postgresql.org/docs/current/performance-tips.html (дата звернення: 12.10.2025).

12.pgbench. PostgreSQL Documentation. URL: https://www.postgresql.org/docs/current/pgbench.html (дата звернення: 12.10.2025).

A. Havor, D. Nishchemenko, K. Hordiienko, A. Aronov

COMPARING THE PERFORMANCE OF PYTHON DJANGO AND JAVA HIBERNATE ORM FRAMEWORKS

The article presents a comparative analysis of the performance of ORM frameworks Django ORM (Python) and Hibernate (Java) in large-scale relational database environments. The relevance of the study is determined by the need to optimize the interaction between the application layer and the database in systems with high demands for performance and data consistency.

The purpose of the work is to experimentally evaluate performance, latency, SQL query intensity, CPU usage, and memory consumption for both ORM frameworks in a unified PostgreSQL environment. The modeling included CRUD operations, join queries, and aggregation scenarios with a dataset exceeding 900,000 records.

The results showed that Django ORM outperforms in basic operations and complex data selections due to its lower level of abstraction and reduced number of intermediate layers. Hibernate, in contrast, demonstrates stability and consistency under high transactional load, ensuring scalability through its multi-layered architecture and JVM-level optimizations.

It is recommended to use Django ORM in web applications and rapid development systems, while Hibernate is more suitable for corporate and financial systems where stability and object state control are critical.

Keywords: ORM, Django, Hibernate, relational database, performance, latency, transactions, PostgreSQL, Python, Java, programming languages.