International Science Journal of Engineering & Agriculture
2026; 5(4): 60-68
https://isg-journal.com/isjea
doi: 10.46299/j.isjea.20260504.05
ISSN: 2720-6319

INTERNATIONAL
SCIENCE GROUP

Порівняльний аналіз продуктивності Prisma ORM та нативного pg-драйвера при роботі з PostgreSQL у Node.js-застосунках

Ігор Андрущак
Луцький національний технічний університет, Луцьк, Україна
ORCID: 0000-0002-8751-4420

Ілля Пуршак
Луцький національний технічний університет, Луцьк, Україна
ORCID: 0009-0007-0407-2625

Анотація: Вибір інструменту доступу до бази даних є одним із базових архітектурних рішень при розробці серверних застосунків на Node.js. ORM-інструменти, зокрема Prisma, спрощують взаємодію з реляційними базами даних через типобезпечний інтерфейс, однак вносять додатковий рівень абстракції, вплив якого на продуктивність запитів залишається недостатньо дослідженим у контексті актуальних версій інструменту. У 2025 році Prisma здійснила архітектурний перехід від внутрішнього Rust-рушія до нативного TypeScript-рушія, що принципово змінило профіль продуктивності інструменту і зробило результати попередніх порівняльних досліджень некоректними для екстраполяції на поточний стан екосистеми. У дослідженні проведено контрольований бенчмарк Prisma ORM версії 7 та нативного pg-драйвера версії 8 при роботі з PostgreSQL 17 у Node.js-середовищі. Вимірювання виконувались ізольовано від HTTP-шару як автономний Node.js-процес поза контекстом веб-фреймворку на реальній схемі бази даних TypeScript-застосунку з 10 000 записів. Для кожного з чотирьох типів операцій – простого SELECT, фільтрованого SELECT із пагінацією, INSERT та JOIN-запиту з об'єднанням чотирьох таблиць – виконано 1000 ітерацій із прогрівом у 50 запитів. Зафіксовано середнє значення, медіану, мінімум, максимум та стандартне відхилення латентності. За результатами дослідження pg-драйвер демонструє нижчу середню латентність у всіх чотирьох сценаріях – від ~25% для INSERT до ~47% для JOIN-запиту. Зафіксований розрив є суттєво меншим порівняно з результатами попередніх досліджень, що підтверджує якісну зміну продуктивності Prisma після переходу на TypeScript-рушій. Абсолютна різниця між підходами становить 0.5–1.1 мс, що є статистично достовірним, але практично незначущим для переважної більшості продуктових застосунків. Стабільність результатів підтверджена повторним запуском на наборі з 100 записів. Отримані результати формують практичну основу для обґрунтованого вибору інструменту доступу до бази даних при розробці TypeScript-застосунків на Node.js із PostgreSQL.

Ключові слова: Prisma ORM; pg-драйвер; PostgreSQL; Node.js; TypeScript; продуктивність бази даних; бенчмарк; латентність запитів; ORM-абстракція; порівняльний аналіз.

1. Вступ

Сучасна розробка серверних застосунків на Node.js нерозривно пов'язана з вибором інструменту для роботи з реляційними базами даних. Серед доступних підходів особливе місце займають ORM-інструменти – бібліотеки, що забезпечують об'єктно-орієнтований інтерфейс до реляційних даних і дозволяють розробнику взаємодіяти зі схемою бази даних засобами мови програмування без написання SQL вручну. PostgreSQL залишається одною з найпопулярніших реляційних СУБД у світі, стабільно посідаючи провідні позиції серед

інструментів, що використовуються у виробничих середовищах [1]. У TypeScript-екосистемі Node.js одним із найбільш поширених ORM-інструментів є Prisma – інструмент із декларативною схемою, автоматично згенерованим типобезпечним клієнтом та вбудованою системою міграцій [2, 3]. Строга типізація запитів, яку забезпечує Prisma, дозволяє виявляти помилки на етапі компіляції, що суттєво підвищує надійність коду при роботі зі складними схемами даних [4].

Попри очевидні переваги з точки зору зручності розробки, використання ORM-інструментів неминуче вносить додатковий шар абстракції між застосунком і базою даних. Це ставить під питання, наскільки суттєвими є накладні витрати цієї абстракції порівняно з прямим зверненням до бази даних через нативний драйвер – у випадку Node.js та PostgreSQL це бібліотека pg. Відповідь на це питання є практично значущою: у застосунках із високим навантаженням або з вимогами до мінімальної латентності вибір між ORM та нативним драйвером може безпосередньо впливати на продуктивність системи в цілому.

Додаткової актуальності цьому питанню надає архітектурна еволюція самої Prisma. До виходу версії 7 інструмент використовував внутрішній Rust-рушій, який вносив суттєві накладні витрати, задокументовані в низці досліджень. У 2025 році Prisma здійснила перехід на нативний TypeScript-рушій, що принципово змінило характеристики продуктивності інструменту. Результати попередніх порівняльних досліджень, таким чином, більше не відображають актуального стану речей і потребують переосмислення в контексті нової архітектури.

Метою цієї роботи є проведення контрольованого бенчмарку Prisma 7 та нативного pg-драйвера на рівні запиту до бази даних, аналіз отриманих результатів та формулювання практичних рекомендацій щодо вибору інструменту для роботи з PostgreSQL у Node.js-застосунках на TypeScript.

2. Об'єкт і предмет дослідження

Об'єктом дослідження є продуктивність інструментів доступу до реляційних баз даних у серверних Node.js-застосунках на TypeScript, а саме: як вибір між ORM-абстракцією та нативним драйвером впливає на латентність запитів при роботі з PostgreSQL. Сучасні серверні застосунки на Node.js взаємодіють із базою даних через один із двох принципово різних підходів: або через ORM-інструмент, що надає типобезпечний об'єктно-орієнтований інтерфейс і приховує SQL за рівнем абстракції, або через нативний драйвер, що виконує SQL-запити безпосередньо без проміжних перетворень. Вибір між цими підходами є одним із базових архітектурних рішень при проектуванні бекенду, і від нього залежить не лише продуктивність, а й зручність розробки, типова безпека та підтримуваність коду.

Prisma ORM є одним із найпоширеніших інструментів у TypeScript-екосистемі Node.js. Він надає декларативну схему бази даних, автоматично згенерований типобезпечний клієнт та вбудовану систему міграцій. Нативний pg-драйвер є низькорівневим інтерфейсом до PostgreSQL, що дозволяє виконувати довільні SQL-запити з мінімальними накладними витратами. Принципово важливим контекстом є те, що у Prisma 7 обидва підходи спираються на один і той самий низькорівневий механізм з'єднань – pg використовується як транспортний рівень через PrismaPg-адаптер, – що дозволяє ізолювати вимірювання виключно на рівні абстракції ORM.

Предметом дослідження є відмінності у латентності запитів між Prisma ORM версії 7 та нативним pg-драйвером версії 8 при виконанні чотирьох типів операцій – простого SELECT за первинним ключем, фільтрованого SELECT із пагінацією, INSERT та JOIN-запиту з об'єднанням чотирьох таблиць – на реальній схемі бази даних TypeScript-застосунку з використанням PostgreSQL 17.

Предмет включає дослідження того, як накладні витрати ORM-абстракції – маппінг результатів у типізовані TypeScript-об'єкти, генерація SQL із декларативного опису запиту,

управління пулом з'єднань – розподіляються між різними типами операцій і чи є ця різниця практично значущою в контексті реальних серверних застосунків. Окремо розглядається питання стабільності латентності: чи відрізняється варіативність результатів між підходами, і що це означає для систем із вимогами до передбачуваності часу відповіді.

У межах дослідження також розглядається версійний контекст: архітектурний перехід Prisma від Rust-рушія до нативного TypeScript-рушія у версії 7 змінив профіль продуктивності інструменту, і результати попередніх досліджень потребують переосмислення з урахуванням цієї зміни. Таким чином, предмет дослідження охоплює як кількісний аспект – числове порівняння латентності у контрольованих умовах – так і якісний: інтерпретацію отриманих результатів у контексті архітектурної еволюції інструменту та їх практичне значення для прийняття рішень при розробці TypeScript-застосунків.

3. Мета та задачі дослідження

Метою дослідження є кількісне порівняння продуктивності Prisma ORM версії 7 та нативного pg-драйвера при роботі з PostgreSQL у Node.js-середовищі на основі контрольованого бенчмарку, а також формулювання практичних рекомендацій щодо вибору інструменту доступу до бази даних для TypeScript-застосунків.

Досягнення мети передбачає вирішення таких взаємопов'язаних задач:

- проектування методології бенчмарку: визначити набір операцій, що репрезентують основні сценарії взаємодії з базою даних у серверних застосунках; обґрунтувати протокол вимірювання, що забезпечує ізоляцію досліджуваної змінної від зовнішніх чинників – HTTP-шару, серіалізації та мережевої варіативності; сформулювати вимоги до статистичної достовірності результатів;
- реалізація тестового середовища: розробити автономний бенчмарк-скрипт поза контекстом веб-фреймворку; реалізувати дзеркальні імплементації чотирьох операцій через Prisma та pg з ідентичною семантикою запитів; забезпечити симетричність умов порівняння шляхом використання спільного механізму пулу з'єднань для обох підходів через PrismaPg-адаптер;
- проведення вимірювань: виконати 1000 ітерацій із прогрівом у 50 запитів для кожної операції на наборі з 10 000 записів; зафіксувати середнє значення, медіану, мінімум, максимум та стандартне відхилення латентності; підтвердити стабільність результатів повторним запуском на наборі з 100 записів;
- аналіз та інтерпретація результатів: встановити відносну різницю між підходами для кожного типу операцій; пояснити природу накладних витрат ORM-абстракції через призму архітектурних змін Prisma 7; зіставити отримані результати з висновками попередніх досліджень у контексті версійної еволюції інструменту;
- формулювання рекомендацій: визначити сценарії, у яких різниця між підходами є практично значущою, та сценарії, де вона не впливає на вибір інструменту; надати обґрунтовану рекомендацію щодо вибору між Prisma та pg для типових TypeScript-застосунків на Node.js.

4. Аналіз літератури

Питання продуктивності ORM-інструментів у порівнянні з прямими підходами до роботи з базами даних досліджувалося в різних технологічних контекстах, однак специфіка Node.js-екосистеми та TypeScript-орієнтованих застосунків залишається недостатньо висвітленою в академічній літературі.

Серед найбільш релевантних робіт слід виділити дослідження, опубліковане у збірнику наукових праць «SCIENTIA» (Гельсінкі, 2023), у якому Жихарський П. та Губарєва О. провели порівняльний аналіз підходів до роботи з базами даних у контексті API-застосунків [5]. Автори

зафіксували суттєву різницю в часі виконання запитів між нативним SQL та ORM-рішеннями, зазначивши, що накладні витрати ORM особливо відчутні при складних запитах із з'єднаннями таблиць. Разом із тим дослідження не охоплювало Node.js-стек і Prisma ORM зокрема, що залишає відкритим питання застосовності цих висновків у сучасних TypeScript-застосунках.

Загальнішу картину продуктивності ORM у середовищі .NET окреслює робота Ґуверчіна А. та Авеноґлу Б., опублікована у «Bilişim Teknolojileri Dergisi» (2022), де автори дослідили поведінку різних ORM-інструментів у середовищі .NET 6 [6]. Дослідження підтвердило, що ORM-рішення стабільно поступаються нативному SQL за абсолютними показниками швидкодії, однак ступінь цього відставання суттєво залежить від типу операції та конфігурації інструменту. Аналогічні висновки отримали Змаранда Д. та співавтори (2020), які зафіксували переваги нативного підходу при операціях CRUD у середовищі .NET, хоч і відзначили, що різниця в продуктивності скорочується з ростом складності бізнес-логіки [7]. Обидві роботи стосуються платформ, архітектурно відмінних від Node.js, що ускладнює перенесення їхніх результатів на досліджуваний контекст.

Яннелі Р. у роботі «Object Relational Mapping Framework Performance Impact» (2021) досліджував вплив ORM-фреймворків на продуктивність запитів у широкому класі застосунків [8]. Автор зробив висновок, що накладні витрати ORM пов'язані не стільки з абстракцією як такою, скільки з кількістю SQL-запитів, що генеруються для однієї операції на рівні застосунку – особливо у сценаріях із вибіркою пов'язаних сутностей. Цей висновок безпосередньо стосується Prisma ORM, оскільки інструмент за замовчуванням виконує окремі запити для кожного включеного відношення замість JOIN. Практичне підтвердження застосовності Prisma ORM у реальних проєктах із складною схемою бази даних представлене також у роботі Мутіари Аулії Хадіджи та співавторів (2024), де інструмент використовувався для реалізації модуля управління ресурсами у виробничій системі з кількома взаємопов'язаними сутностями [9].

З боку самої екосистеми Prisma заслуговують уваги офіційні бенчмарки, опубліковані командою розробників у 2024 році [10, 11]. У цих матеріалах Prisma ORM порівнювалася з TypeORM та Drizzle ORM на реальній інфраструктурі AWS RDS та Supabase, з використанням 500 ітерацій на кожну операцію. Результати засвідчили, що різниця в медіанній латентності між досліджуваними ORM-рішеннями є незначною для простих операцій вибірки, натомість суттєво зростає при складних запитах із вкладеними відношеннями.

Комплексне дослідження, опубліковане на ScienceDirect (2025), присвячене порівнянню Prisma ORM та оптимізованого нативного SQL на PostgreSQL у контейнеризованому середовищі, представляє найближчу до цієї роботи постановку задачі [12]. Автори зафіксували, що нативний SQL виявився до п'яти разів швидшим за часом виконання, а також споживав у 6–9 разів менше CPU та у 2–3 рази менше пам'яті порівняно з Prisma. Робота Йокубжонова Д. та Гібадулліна Р. (2024) досліджувала схожу проблематику у контексті Microsoft SQL Server і підтвердила загальну тенденцію переваги нативного підходу, вказавши водночас, що для простих CRUD-операцій різниця може бути статистично незначущою [13].

Однак критично важливим контекстом для інтерпретації наведених результатів є версійна історія Prisma ORM. До виходу версії 7 Prisma використовувала внутрішній Rust-рушій для виконання запитів, що вносило суттєві накладні витрати, особливо помітні при роботі з великими наборами результатів. У 2025 році команда Prisma здійснила повний архітектурний перехід на нативний TypeScript-рушій, офіційно зафіксувавши триразове прискорення виконання запитів та 90-відсоткове скорочення розміру бандлу порівняно з Rust-реалізацією [14]. Це означає, що результати всіх досліджень, проведених до виходу Prisma 7, характеризують інструмент у принципово іншому технічному стані, ніж той, що є актуальним на момент написання цієї роботи.

Таким чином, аналіз наявних публікацій виявляє прогалину, яку заповнює це дослідження: більшість порівняльних робіт або орієнтовані на платформи поза Node.js-екосистемою, або ґрунтуються на версіях Prisma з Rust-рушієм, результати яких не можуть

бути коректно екстрапольовані на поточний стан інструменту. Крім того, більшість існуючих бенчмарків вимірюють час відповіді на рівні HTTP-запиту, що поєднує в одному показнику затримку бази даних, серіалізацію, мережевий стек та обробку у фреймворку. Це дослідження спрямоване на заповнення зазначеної прогалини шляхом проведення ізольованого бенчмарку Prisma 7 та нативного pg-драйвера безпосередньо на рівні запиту до бази даних, на реальній схемі TypeScript-застосунку з використанням PostgreSQL

5. Методи досліджень

Для досягнення мети дослідження було розроблено контрольований бенчмарк, що дозволяє ізольовано виміряти латентність запитів до бази даних при використанні двох підходів – Prisma ORM та нативного pg-драйвера – в ідентичних умовах виконання.

Середовище виконання. Експеримент проводився локально на одній машині з метою виключення мережевої варіативності між застосунком і базою даних. База даних PostgreSQL запущена у Docker-контейнері. Версії компонентів: Node.js 22, Prisma 7, pg 8, PostgreSQL 17. З'єднання з базою даних в обох підходах здійснювалося через пул з'єднань – у Prisma це вбудований пул, у pg використовувався Pool з бібліотеки node-postgres [15]. Це забезпечило симетричність умов, оскільки обидва підходи виконували запити через постійні з'єднання без накладних витрат на їх встановлення при кожній ітерації.

Схема даних. Бенчмарк виконувався на реальній схемі бази даних TypeScript-застосунку, що включає таблицю користувачів із пов'язаними сутностями: обліковими даними, профілем організації та налаштуваннями двофакторної автентифікації. Схема налічує понад 18 полів у межах одного запиту з об'єднаннями. База даних попередньо заповнена фіксованим набором із 10 000 записів користувачів, згенерованих за допомогою seed-скрипту. Набір даних не змінювався між запусками.

Операції тестування. Для порівняння обрано чотири типи операцій, що охоплюють основні сценарії взаємодії з базою даних у серверних застосунках: простий SELECT за первинним ключем (отримання одного запису користувача за ідентифікатором), фільтрований SELECT із пагінацією (вибірка активних користувачів із обмеженням у 20 записів), INSERT із подальшим видаленням створеного запису (для забезпечення ізольованості між ітераціями) та JOIN-запит із об'єднанням чотирьох таблиць (користувач разом із обліковими даними, профілем та налаштуваннями двофакторної автентифікації).

Протокол вимірювання. Кожна операція виконувалася в 1000 ітерацій із попереднім прогрівом у 50 ітерацій, результати яких не враховувалися. Прогрів необхідний для того, щоб пул з'єднань встановився, а перші аномально повільні запити не впливали на статистику. Час кожної ітерації фіксувався засобами performance.now() безпосередньо навколо виклику до бази даних, без включення будь-якої іншої логіки застосунку. HTTP-шар, серіалізація відповіді та middleware фреймворку були повністю виключені з вимірювання. За результатами 1000 вимірювань для кожної операції обчислювались такі статистичні показники: середнє значення (mean), медіана (p50), мінімум, максимум та стандартне відхилення.

Реалізація. Обидва підходи реалізовані як окремі модулі в межах одного бенчмарк-скрипту, що виконується поза контекстом веб-сервера. Операції Prisma та pg чергувалися між собою в рамках кожної ітерації, а не виконувалися блоками – це унеможливлює вплив теплового стану процесора та кешу бази даних на відносні результати одного з підходів.

6. Результати досліджень

За результатами проведеного експерименту отримано статистичні показники латентності для чотирьох типів операцій при двох підходах до роботи з базою даних. Основні результати наведено в таблиці 1, що містить дані для набору з 10 000 записів.

Таблиця 1 . Результати бенчмарку Prisma 7 vs pg 8 (10 000 записів, 1000 ітерацій)

Операція	Інструмент	Mean, мс	p50, мс	Min, мс	Max, мс	Stddev, мс	Перевага pg, %
Simple SELECT	Prisma	1,449	1,344	0,877	23,322	1,159	–
Simple SELECT	pg	0,846	0,786	0,534	17,557	0,611	41,6%
Filtered SELECT	Prisma	1,493	1,445	0,988	21,206	0,954	–
Filtered SELECT	pg	0,917	0,872	0,594	17,464	0,738	38,6%
INSERT	Prisma	2,766	2,130	1,760	25,658	3,285	–
INSERT	pg	2,063	1,533	1,200	24,803	2,973	25,4%
JOIN	Prisma	2,366	2,287	1,655	20,903	1,020	–
JOIN	pg	1,248	1,176	0,880	21,844	0,863	47,3%

У всіх чотирьох операціях pg демонструє нижче середнє значення латентності порівняно з Prisma. Найбільша відносна різниця зафіксована для операції простого SELECT – середня латентність pg становить 0.846 мс проти 1.449 мс у Prisma, що відповідає перевазі у ~41%. Схожу картину демонструє JOIN-запит: 1.248 мс проти 2.366 мс, різниця ~47%. Для фільтрованого SELECT різниця за середнім значенням становить ~39%. Найменша відносна різниця зафіксована для операції INSERT – 2.063 мс проти 2.766 мс, що відповідає ~25%. При цьому стандартні відхилення для INSERT є найвищими серед усіх операцій і близькими між собою для обох підходів (3.285 мс у Prisma проти 2.973 мс у pg), що свідчить про те, що варіативність латентності INSERT визначається переважно поведінкою самої бази даних при операціях запису, а не рівнем абстракції інструменту.

Окремої уваги заслуговує порівняння стабільності результатів. Незважаючи на те, що pg стабільно швидший за середнім значенням, Prisma демонструє зіставне або менше стандартне відхилення в більшості операцій – зокрема для простого SELECT (1.159 мс проти 0.611 мс у pg) та JOIN (1.020 мс проти 0.863 мс у pg). Це означає що перевага pg за швидкістю супроводжується дещо вищою варіативністю латентності в окремих сценаріях.

З метою перевірки стабільності отриманих результатів при різних розмірах набору даних експеримент було повторено на базі з 100 записів за ідентичним протоколом. Скорочені результати наведено в таблиці 2.

Таблиця 2. Порівняння mean та p50 при 100 та 10 000 записах

Операція	Інструмент	100 записів		10 000 записів	
		Mean, мс	p50, мс	Mean, мс	p50, мс
Simple SELECT	Prisma	1,399	1,366	1,449	1,344
Simple SELECT	pg	0,814	0,772	0,846	0,786
Filtered SELECT	Prisma	1,342	1,207	1,493	1,445
Filtered SELECT	pg	0,868	0,732	0,917	0,872
INSERT	Prisma	2,649	2,025	2,766	2,130
INSERT	pg	2,080	1,457	2,063	1,533
JOIN	Prisma	2,302	2,144	2,366	2,287
JOIN	pg	1,210	1,040	1,248	1,176

Результати двох запусків є практично ідентичними – різниця між середніми значеннями не перевищує 10% для жодної з операцій. Відносна перевага pg над Prisma залишилась незмінною в обох конфігураціях, що підтверджує стабільність і відтворюваність отриманих результатів незалежно від розміру набору даних у дослідженому діапазоні. Ця стабільність пояснюється тим, що всі запити спираються на індекси – пошук за первинним ключем та

фільтрація за індексованим полем – тому планувальник PostgreSQL обирає однаковий план виконання незалежно від кількості рядків у таблиці.

7. Перспективи подальшого розвитку досліджень

Проведене дослідження охоплює чотири типи операцій на схемі з 10 000 записів в умовах локального підключення без мережевої затримки між застосунком і базою даних, що визначає межі його узагальнення та окреслює напрями подальшої роботи. Першочерговим напрямом є дослідження поведінки обох підходів в умовах конкурентного навантаження з паралельними запитами. Поточний бенчмарк виконує операції послідовно, що не відображає реального профілю навантаження серверного застосунку, де пул з'єднань обслуговує одночасні запити від різних клієнтів. В умовах конкуренції за з'єднання накладні витрати управління пулом можуть по-різному проявлятися у Prisma та pg, і відносна різниця в латентності може суттєво змінитись порівняно з результатами, отриманими при послідовному виконанні.

Другим перспективним напрямом є повторення вимірювань в умовах реальної мережевої затримки між застосунком і базою даних – коли PostgreSQL розгорнуто на окремому сервері, а не в Docker-контейнері на тій самій машині. У поточному експерименті абсолютні значення латентності є нижчими, ніж у типових виробничих умовах, саме через відсутність мережевого round-trip. При збільшенні базової мережевої затримки абсолютна різниця 0.5–1.1 мс між підходами може набувати іншого практичного значення: відносна перевага pg залишиться незмінною, але питома вага ORM-накладних витрат у загальному часі відповіді зменшиться, що додатково обґрунтовує вибір Prisma для більшості продуктових сценаріїв.

Третім напрямом є оцінка впливу Prisma Accelerate – хмарного рівня кешування та пулу з'єднань від команди Prisma – на продуктивність у виробничому середовищі. Поточне дослідження порівнює Prisma та pg за умови прямого підключення до бази даних; введення Prisma Accelerate як проміжного рівня може суттєво змінити профіль латентності для операцій читання, де кешування є найбільш ефективним. Нарешті, розширення набору операцій на сценарії з вкладеними відношеннями – де Prisma за замовчуванням виконує окремі запити замість JOIN – дозволить кількісно оцінити проблему N+1 запитів у контексті нового TypeScript-рушія і перевірити, чи зберігається ця поведінка після архітектурного переходу від Rust-реалізації.

8. Висновки

У дослідженні проведено контрольований бенчмарк Prisma 7 та нативного pg-драйвера на рівні запиту до бази даних PostgreSQL у Node.js-середовищі. Вимірювання виконувались ізольовано від HTTP-шару на реальній схемі TypeScript-застосунку з використанням 1000 ітерацій та прогрівом у 50 запитів для кожної з чотирьох операцій. Стабільність результатів підтверджена повторним запуском на наборі з 100 записів, який показав відхилення не більше 10% від основних значень при незмінній відносній різниці між підходами.

На основі отриманих даних встановлено:

1. pg-драйвер демонструє нижчу середню латентність у всіх чотирьох сценаріях – від ~25% для INSERT до ~47% для JOIN-запиту. Найменша різниця зафіксована для операції INSERT, де стандартні відхилення обох підходів практично збігаються (3.285 мс у Prisma проти 2.973 мс у pg), що свідчить про домінування затримок на рівні самої бази даних при операціях запису. Найбільша відносна різниця зафіксована для JOIN-запиту та простого SELECT, де накладні витрати маппінгу результатів у типізовані TypeScript-об'єкти є найбільш відчутними.

2. Зафіксований розрив між підходами є суттєво меншим порівняно з результатами попередніх досліджень, де перевага нативного SQL над Prisma сягала п'яти разів. Це підтверджує що архітектурний перехід Prisma 7 від Rust-рушія до нативного TypeScript-рушія

принципово змінив профіль продуктивності інструменту – і попередні порівняльні результати не можуть бути коректно екстрапольовані на поточний стан екосистеми.

3. Абсолютна різниця між підходами становить 0.5–1.1 мс залежно від типу операції. У контексті реального серверного застосунку, де загальний час обробки HTTP-запиту вимірюється десятками мілісекунд, ця різниця є статистично достовірною, але практично незначущою для переважної більшості продуктових сценаріїв. Prisma 7 є обґрунтованим вибором за замовчуванням для TypeScript-застосунків на Node.js – перехід на нативний pg виправданий лише тоді, коли профілювання підтверджує що рівень ORM-абстракції є вузьким місцем продуктивності, а не мережа, серіалізація чи бізнес-логіка.

4. Межі узагальнення: експеримент проводився локально з підключенням до бази даних через Docker без мережевої затримки між застосунком і сервером бази даних, що дає нижчі абсолютні значення латентності порівняно з виробничими умовами, хоча відносне порівняння між підходами залишається дійсним. Результати отримані на схемі з 10 000 записів – поведінка при значно більших наборах даних зі складними індексами не досліджувалась. Перспективні напрями: дослідити поведінку при конкурентному навантаженні з паралельними запитами; порівняти результати в умовах реальної мережевої затримки між застосунком і базою даних; оцінити вплив Prisma Accelerate як рівня кешування на продуктивність у виробничому середовищі.

Список літератури:

1) Akhtar, A. (2023). Popularity ranking of database management systems. https://arxiv.org/abs/2301.00847

2) Prisma. (n.d.). Prisma ORM | Type-safe ORM for Node.js and TypeScript. https://www.prisma.io/orm

3) Strapi. (2024). Top 5 ORMs for developers in 2025. https://strapi.io/blog/orms-for-developers

4) Prisma. (n.d.). Type safety. https://www.prisma.io/docs/orm/prisma-client/type-safety

5) Zhykharskyi, P., & Gubaryeva, O. (2023). Pure SQL vs ORM libraries: A comparative analysis for API applications. Collection of Scientific Papers «SCIENTIA» (June 30, 2023; Helsinki, Finland), 97–98.

6) Güvercin, A. E., & Avenoğlu, B. (2022). Performance analysis of object-relational mapping (ORM) tools in .NET 6 environment. Bilişim Teknolojileri Dergisi, 15(4), 453–465. https://doi.org/10.17671/gazibtd.1059516

7) Zmăranda, D., Pop-Fele, L.-L., Győrödi, C., Győrödi, R., & Pecherle, G. (2020). Performance comparison of CRUD methods using .NET object relational mappers: A case study. International Journal of Advanced Computer Science and Applications, 11(1). https://www.ijacsa.thesai.org

8) Jannali, R. (2021). Object relational mapping framework performance impact. Turkish Journal of Computer and Mathematics Education, 12(7).

9) Mutiara Auliya Khadija et al. (2024). Backend programming techniques in the development of resource manager features in VRMS systems based on ORM Prisma. Decode: Jurnal Pendidikan Teknologi Informasi, 4(3), 830–843. https://doi.org/10.51454/decode.v4i3.792

10) Prisma. (2024). Performance benchmarks: Comparing query latency across TypeScript ORMs & databases. https://www.prisma.io/blog/performance-benchmarks-comparing-query-latency-across-typescript-orms-and-databases

11) Prisma. (n.d.). Query performance benchmarks. https://benchmarks.prisma.io/

12) Yusmita, J. C., Arya, R., Wijaya, J. M., Suryaningrum, K. M., & Siswanto, R. R. (2025). Optimizing database access strategy: A performance analysis comparison of raw SQL and Prisma ORM. Procedia Computer Science. https://www.sciencedirect.com/science/article/pii/S187705092502722X

13) Yoqubjonov, D. I., & Gibadullin, R. F. (2024). A performance study of object-relational mapping technologies when interacting with Microsoft SQL Server. International Research Journal.

14) Prisma. (2025). Prisma ORM without Rust: Latest performance benchmarks. https://www.prisma.io/blog/prisma-orm-without-rust-latest-performance-benchmarks-node-postgres. (n.d.). Connection pooling. https://node-postgres.com/features/pooling

Comparative Analysis of the Performance of Prisma ORM and the Native pg Driver When Working with PostgreSQL in Node.js Applications

Ihor Andrushchak
Lutsk National Technical University, Lutsk, Ukraine
ORCID: 0000-0002-8751-4420

Illa Purshak
Lutsk National Technical University, Lutsk, Ukraine
ORCID: 0009-0007-0407-2625

Abstract: The choice of database access tool is one of the fundamental architectural decisions in Node.js server-side application development. ORM tools, including Prisma, simplify interaction with relational databases through a type-safe interface, yet introduce an additional abstraction layer whose impact on query performance remains insufficiently studied in the context of current tool versions. In 2025, Prisma underwent an architectural transition from an internal Rust engine to a native TypeScript engine, which fundamentally changed the performance profile of the tool and rendered the results of previous comparative studies invalid for extrapolation to the current state of the ecosystem. This study presents a controlled benchmark of Prisma ORM version 7 and the native pg driver version 8 operating against PostgreSQL 17 in a Node.js environment. Measurements were performed in isolation from the HTTP layer as a standalone Node.js process outside the context of any web framework, on a real TypeScript application database schema with 10,000 records. For each of four operation types – simple SELECT, filtered SELECT with pagination, INSERT, and a JOIN query across four tables – 1,000 iterations were executed with a 50-query warm-up. Mean, median, minimum, maximum, and standard deviation of latency were recorded. The results show that the pg driver demonstrates lower mean latency across all four scenarios – ranging from ~25% for INSERT to ~47% for the JOIN query. The observed gap is substantially smaller than reported in previous studies, confirming a qualitative shift in Prisma's performance following the transition to the TypeScript engine. The absolute difference between the approaches is 0.5–1.1 ms, which is statistically significant but practically negligible for the vast majority of production applications. Result stability was confirmed by a repeated run on a dataset of 100 records. The findings provide a practical basis for informed selection of a database access tool when developing TypeScript applications on Node.js with PostgreSQL.

Keywords: Prisma ORM; pg driver; PostgreSQL; Node.js; TypeScript; database performance; benchmark; query latency; ORM abstraction; comparative analysis.