Статья поступила в редакцию: 15.06.2026

Статья принята к публикации: 18.06.2026

Статья опубликована: 28.06.2026

УДК 004.65

COMPARATIVE PERFORMANCE ANALYSIS OF DAPPER AND ENTITY FRAMEWORK CORE IN ARCHITECTURAL DESIGN

Sychev Yegor
Senior Software Engineer,
Kaspi Bank JSC,
Kazakhstan, Almaty
E-mail: sigmadel10@gmail.com
ORCID: 0009-0001-6502-8667

Asylkhan Azat
Senior Software Engineer,
BP,
Kazakhstan, Almaty
E-mail: asylkhan.azat@gmail.com
ORCID: 0009-0005-5445-1055

СРАВНИТЕЛЬНЫЙ АНАЛИЗ ПРОИЗВОДИТЕЛЬНОСТИ DAPPER И ENTITY FRAMEWORK CORE В АРХИТЕКТУРНОМ ПРОЕКТИРОВАНИИ

Сычев Егор
старший инженер-программист,
АО Kaspi Bank,
Казахстан, г. Алматы

Асылхан Азат
старший инженер-программист, BP,
Казахстан, г. Алматы

Abstract

Selecting a data-access approach in .NET applications affects latency, memory pressure, and operational cost under load. This paper compares a full-featured object-relational mapping framework, Entity Framework Core (EF Core), with the micro-ORM (object-relational mapping (ORM)) Dapper in a controlled benchmark environment. Experiments were executed against PostgreSQL 17.6 using .NET 10 and BenchmarkDotNet, with containerized database provisioning and deterministic synthetic data generation designed to simulate complex real-world high-throughput enterprise systems. The evaluation measures end-to-end query time, managed memory allocation, and garbage collection activity for representative read and update workloads, including join-heavy retrieval of order graphs and high-concurrency selection. The results indicate that Dapper reduces managed allocations

Библиографическое описание: Sychev Y., Asylkhan A. COMPARATIVE PERFORMANCE ANALYSIS OF DAPPER AND ENTITY FRAMEWORK CORE IN ARCHITECTURAL DESIGN // Universum: технические науки : электрон. научн. журн. 2026. 6(147). URL: *https://7universum.com/ru/tech/archive/item/23040*

by bypassing complex internal state management across the tested scenarios and often improves mean latency, with the largest advantage observed for join-intensive queries and complex updates. EF Core compiled queries narrow the latency gap for some read patterns but remain closer to the baseline in memory footprint. The findings suggest that change tracking and materialization contribute measurable overhead in EF Core, while Dapper's lower abstraction level shifts responsibility to developers for query correctness, parameterization, and mapping discipline. The presented evidence supports selecting a data-access strategy that matches workload characteristics and non-functional constraints, particularly when memory pressure and latency stability are primary targets.

Аннотация

Выбор подхода к доступу к данным в .NET-приложениях влияет на задержки, нагрузку на память и операционные затраты при работе под нагрузкой. В данной статье проводится сравнение полнофункционального фреймворка объектно-реляционного отображения Entity Framework Core (EF Core) и микро-ORM Dapper в контролируемой среде бенчмаркинга. Эксперименты выполнялись с использованием PostgreSQL 17.6, .NET 10 и BenchmarkDotNet, база данных разворачивалась в контейнеризированной среде, а синтетические данные генерировались детерминированно для имитации сложных реальных высоконагруженных корпоративных систем. Оценка включает измерение полного времени выполнения запросов, объема выделяемой управляемой памяти и активности сборщика мусора для репрезентативных сценариев чтения и обновления данных, включая выборку графов заказов с большим количеством JOIN-операций и высококонкурентную выборку. Результаты показывают, что Dapper снижает объем управляемых аллокаций за счет обхода сложного внутреннего управления состоянием в протестированных сценариях и часто уменьшает среднюю задержку, наибольшее преимущество наблюдается для запросов с интенсивным использованием JOIN и для сложных операций обновления. Скомпилированные запросы EF Core сокращают разрыв по задержке для некоторых сценариев чтения, однако по объему используемой памяти остаются ближе к базовой конфигурации. Полученные результаты показывают, что отслеживание изменений и материализация вносят измеримые накладные расходы в EF Core, тогда как более низкий уровень абстракции Dapper переносит ответственность за корректность запросов, параметризацию и дисциплину отображения данных на разработчиков. Представленные данные подтверждают необходимость выбора стратегии доступа к данным с учетом характеристик рабочей нагрузки и нефункциональных требований, особенно в случаях, когда основными целями являются снижение нагрузки на память и стабильность задержек.

Keywords: Dapper, Entity Framework Core, object-relational mapping, data access, benchmarking, performance, .NET platform

Ключевые слова: Dapper, Entity Framework Core, объектно-реляционное отображение, доступ к данным, бенчмаркинг, производительность, платформа .NET

Introduction

Data-access technology selection in .NET is a design decision that affects latency, throughput, and memory utilization in production systems. Object-relational mapping frameworks reduce the

effort required to implement persistence and to evolve domain models, but this convenience can introduce additional overhead through query translation, entity materialization, and change tracking. Micro-ORMs take the opposite position by keeping the mapping layer thin and delegating more responsibility to the application, which can reduce runtime overhead at the cost of higher engineering effort and stricter discipline around SQL maintenance.

Prior comparative studies report consistent performance differences between full ORMs and lightweight data-access approaches. Wiatrowski compared EF Core, Dapper, and LINQ to DB across typical CRUD scenarios and reported lower memory consumption for Dapper in most cases [12]. Zmaranda et al. showed that higher abstraction levels can increase response time and resource usage in CRUD workloads, reinforcing that technology choice should follow the operational profile of a system [23]. EF Core is widely used in enterprise .NET development as a productivity-oriented persistence layer, and its patterns and configuration options are described in detail in both practitioner and reference texts [11]. Experimental comparisons of EF-based access and lower-level approaches in C# similarly emphasize latency sensitivity to abstraction and mapping overhead [9], [1], [6].

Performance is not the only decision factor. This study evaluates EF Core and Dapper in a single, controlled PostgreSQL environment and reports time and allocation metrics for several workload classes. The paper tests four hypotheses: Dapper exhibits lower mean latency for read workloads, the gap increases with query complexity, EF Core compiled queries reduce some read-path overhead, and EF Core consumes more managed memory due to change tracking. The goal is to provide a reproducible basis for choosing an implementation strategy when the main constraints are latency stability and memory pressure, rather than developer convenience alone.

Methods and Materials

EF Core read scenarios are configured according to published performance considerations, with change tracking disabled for read-only query paths to avoid cache artifacts in repeated iterations [8]. The benchmark suite uses BenchmarkDotNet and follows established .NET benchmarking recommendations regarding warmup, iteration control, and noise reduction [2]. EF Core compiled queries are configured following official documentation [7]. The database is provisioned as a PostgreSQL 17.6 instance in a Docker container and re-initialized in a controlled manner for repeatability. The detailed hardware and software specification are summarized in “Table 1”. Database connectivity is provided through Npgsql, which is the standard .NET data provider for PostgreSQL [10]. BenchmarkDotNet configuration and reporting follow the project documentation [4]. Synthetic data are generated deterministically with a fixed seed to avoid time-dependent effects. The shared benchmark harness creates a uniform connection configuration for both technologies by constructing a DbConnection for Dapper and a DbContext for EF Core from the same base settings.

Table 1. Hardware and software configuration of the experimental testbed

Parameter	Value
.NET Version (SDK / Runtime)	.NET SDK 10.0.101 / .NET 10.0.1
Database (DBMS)	PostgreSQL 17.6
Benchmarking Library	BenchmarkDotNet v0.15.8
Operating System	Windows 11 (10.0.26200.7462, 25H2 / 2025 Update)
Processor	AMD Ryzen 7 6800H with Radeon Graphics 3.20 GHz
Cores / Threads	8 physical / 16 logical

The relational schema ‘Fig. 1’ models a minimal e-commerce domain with products, orders, and order items to capture both simple entity retrieval and join-heavy order graph queries.

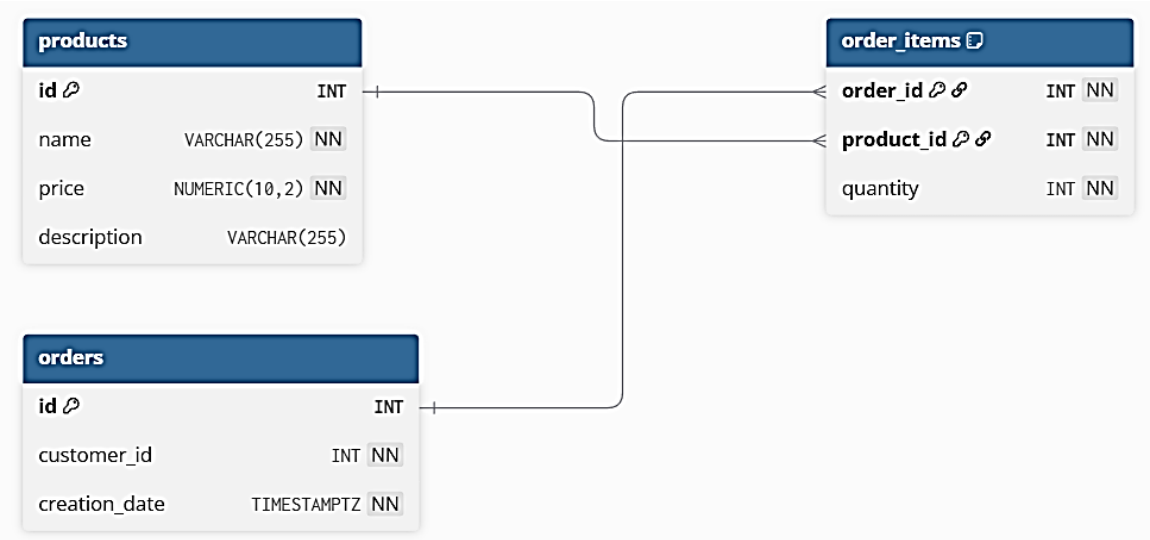

Figure 1. Database schema reflecting basic e-commerce entities

Workloads are designed to cover common access patterns without introducing application-specific business logic. They include single-row reads by primary key, filtered reads, paginated reads over increasing table sizes, join-heavy selection of orders with varying item counts, and update operations for both simple entities and nested order graphs. A high-concurrency selection scenario is included to evaluate allocation behavior under contention. EF Core configurations include a baseline setup and a compiled-query variant, with no-tracking enforced for read scenarios to prevent accidental cache effects in repeated iterations.

Data are generated deterministically with a fixed seed and loaded before measurements to avoid mixing insert costs with query execution. The dataset size is varied across benchmarks by scaling the number of products and orders, with representative points ranging from 10 to 10,000 records depending on the scenario. For join workloads, ItemsPerOrder is controlled to produce order graphs of different cardinalities, which increases the number of rows read and the amount of object materialization performed by each data-access approach.

For each workload category, the benchmark measures an end-to-end path that includes command construction, network round-trip to the containerized DBMS, row materialization, and mapping into in-memory objects. Dapper scenarios execute parameterized SQL and map rows to simple DTO-like objects, while EF Core scenarios execute equivalent LINQ queries translated to SQL and map results to entity types. Update scenarios are implemented as read-modify-write operations to reflect typical application behavior, with EF Core using *SaveChanges* and Dapper executing explicit *UPDATE* statements. Read scenarios use no-tracking for EF Core to isolate query materialization overhead from long-lived change tracking effects.

Time is reported as the mean execution time per operation in microseconds, and memory is reported as managed allocation volume per operation, as emitted by BenchmarkDotNet. Allocation-focused reporting is motivated by the fact that excess allocations translate into garbage collection

pressure, which can increase tail latency under load and amplify variability across concurrent requests. The parallel scenario uses a fixed degree of parallelism (DOP = 128) to stress allocation patterns and to expose differences that may not appear in single-threaded measurements.

To support comparison across heterogeneous scenarios, raw measurements are also transformed into two derived metrics. The performance index combines normalized time and normalized managed allocation relative to a baseline EF Core configuration and provides a single scalar that penalizes methods that are fast but allocate substantially more memory. The scalability factor summarizes how execution time grows with data volume when the query shape is held constant.

Formally, the performance index is defined as

$$PI = \frac{1}{\alpha * T_{norm} + \beta * M_{norm}}, \quad (1)$$

where T_{norm} is normalized execution time and

$$T_{norm} = \frac{T_{method}}{T_{baseline}}, \quad (2)$$

M_{norm} is normalized memory usage

$$M_{norm} = \frac{M_{method}}{M_{baseline}}. \quad (3)$$

The coefficients α and β are fixed to 0.6 and 0.4 to weight time and allocation volume. The scalability factor is defined as

$$SF = \frac{\log_2(T(N_2)/T(N_1))}{\log_2(N_2/N_1)}, \quad (4)$$

which approximates the growth rate of execution time as the number of processed records increases. Where N_n is data volumes at the first and second measurement stages and $T(N_n)$ is execution time for data volume. A near-linear trend in time-versus-volume plots corresponds to SF values close to 1, while persistent superlinear growth suggests multiplicative work in the data-access layer or in the query plan.

Results

“Fig. 2” presents the single-row read scenario for products and illustrates that Dapper achieves lower mean execution time and lower managed allocations than the EF Core baseline in this workload.

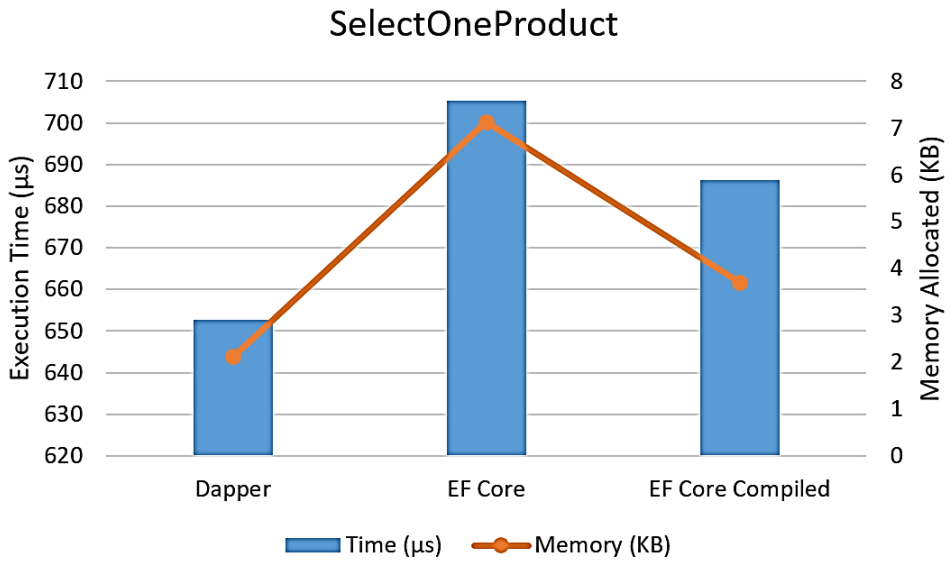

Figure 2. Comparative analysis of memory allocation and execution time for the SelectOneProduct scenario

The dependence of execution time on the number of selected records is shown in ‘Fig. 3”. In bulk selection, the relative gap between implementations narrows as absolute times grow, while the ordering between baseline EF Core, compiled queries, and Dapper remains stable in this benchmark.

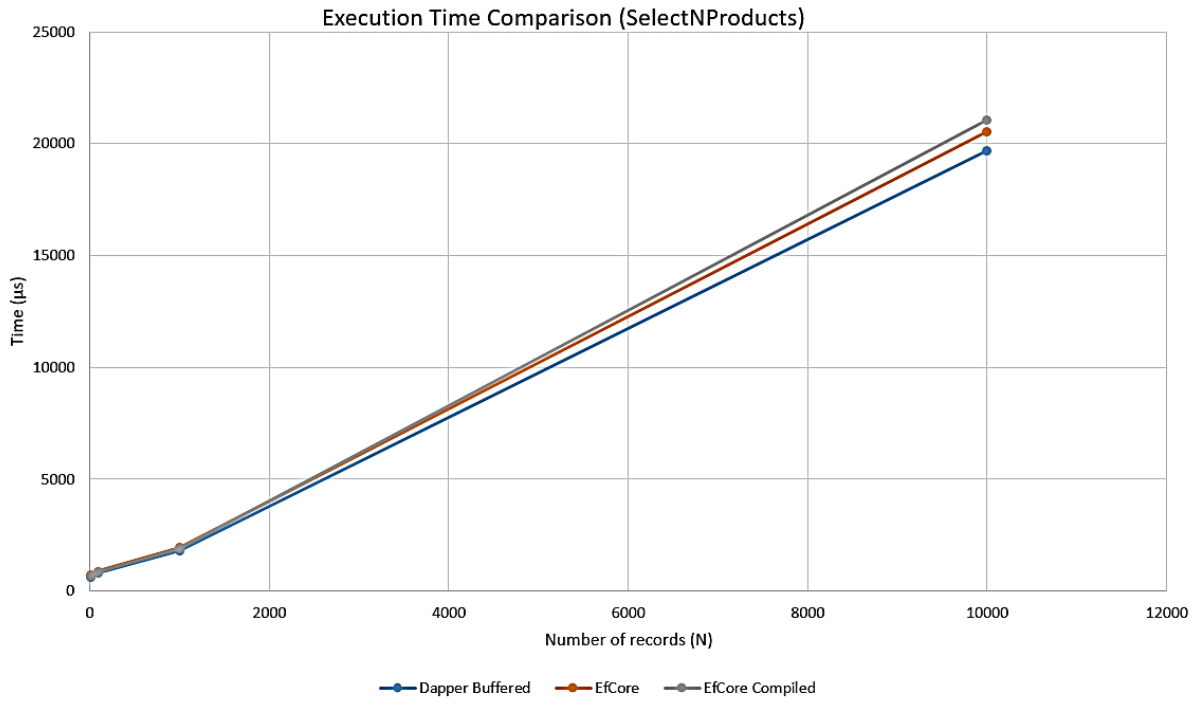

Figure 3. Execution time dependence on the number of records for the SelectNProducts scenario

Join-heavy selection of orders demonstrates the strongest relative advantage for Dapper, as shown in “Fig. 4”. This scenario increases object materialization pressure by expanding the order

graph, which also increases the amount of tracked state in EF Core configurations that do not aggressively minimize tracking overhead.

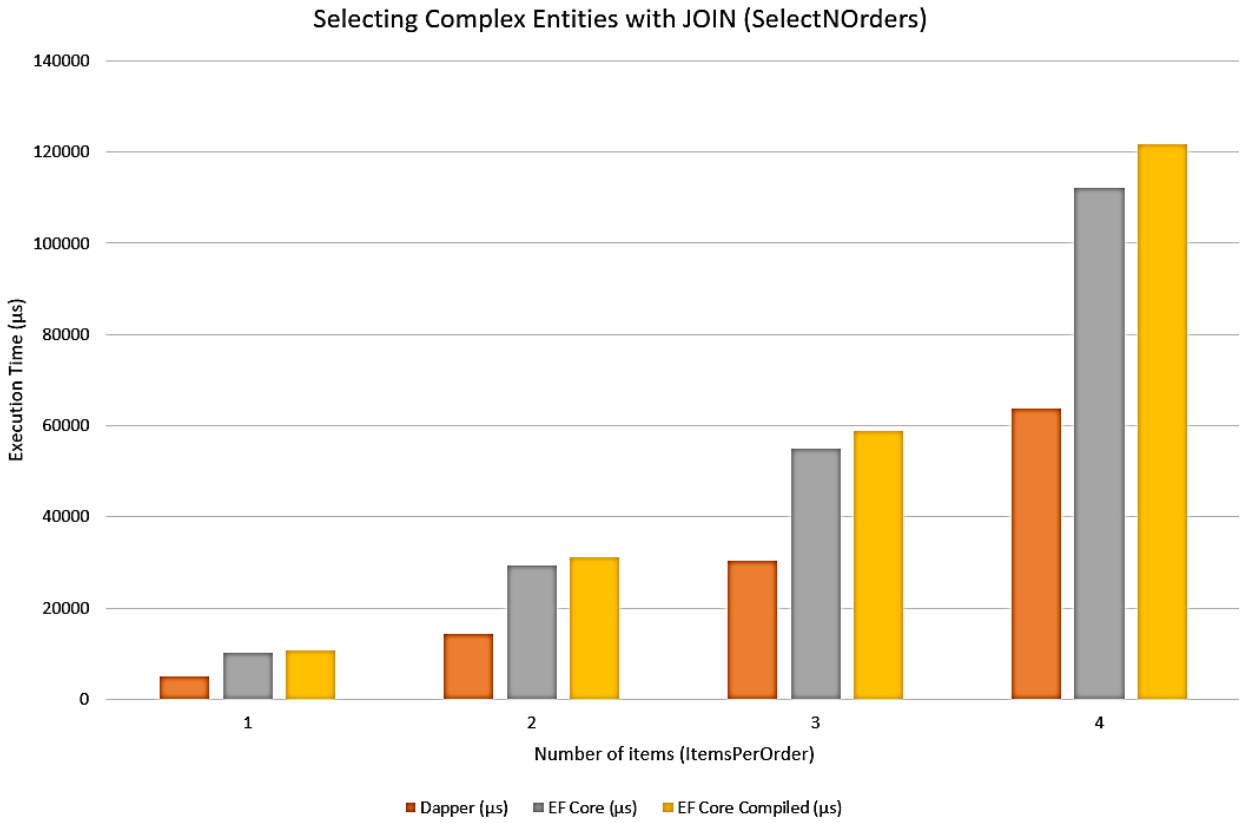

Figure 4. Selecting complex entities with JOIN (SelectNOrders) for different ItemsPerOrder values

To visualize the interaction between data volume and query complexity, “Fig. 5” provides a speedup heat map based on EF Core time divided by Dapper time.

Items per Order	N=10	N=100	N=1000
1	0.91	2.05	2
5	1.99	2.25	2.05
10	1.98	2.33	1.82
20	2.06	2.44	1.76

Figure 5. Heat map of Speedup Factor (EF Core time/Dapper time) across complexity and data volume

The high-concurrency scenario emphasizes allocation behavior at degree of parallelism 128. “Fig. 6” shows that Dapper variants allocate less memory than EF Core variants under contention, which can reduce garbage collection pressure.

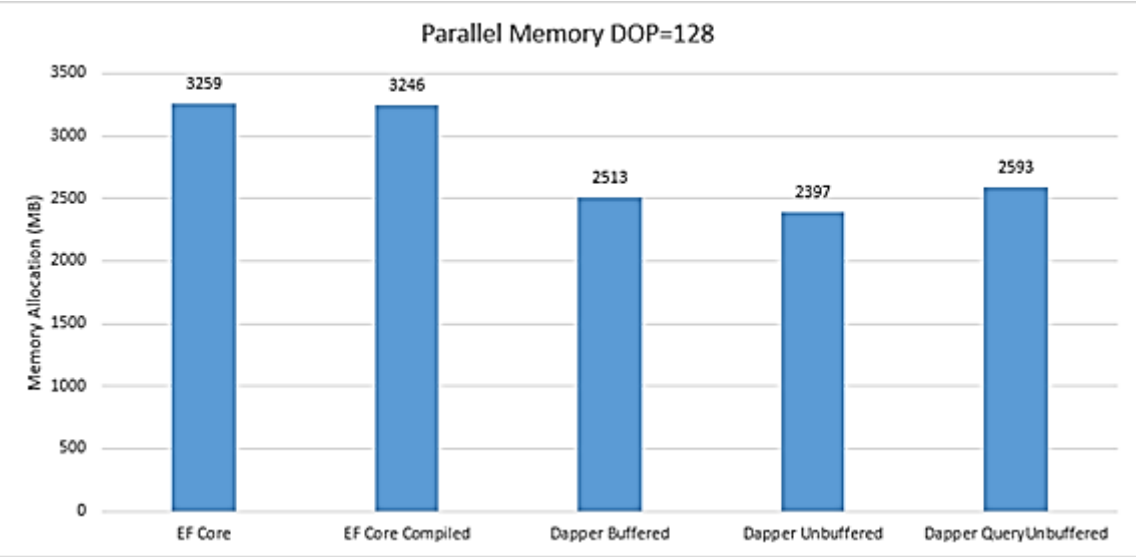

Figure 6. Memory allocation for parallel selection (SelectNProductsParallel) at DOP = 128

The difference between buffered and unbuffered Dapper configurations indicates that materialization strategy affects allocation profiles, even when the SQL query shape is held constant. This observation is consistent with the implementation of asynchronous query execution and buffering in Dapper, where buffering decisions influence transient object creation and enumeration behavior.

"Table 2" summarizes the derived performance index and time advantage for representative scenarios. The join-heavy and complex update scenarios yield the highest index values for Dapper, while compiled queries provide limited benefit outside specific read patterns.

Table 2. Comparative summary of performance index (PI) and relative time efficiency

Test Category	Dapper Time Advantage	PI Dapper	PI EF Compiled
SelectOne simple (TableSize = 1)	8.40%	1.5	1.3
SelectOne with filter	16.40%	1.66	1.43
SelectN (N = 1000)	6.70%	1.16	1.03
SelectN with JOIN (N = 1000, Items = 5)	51.20%	2.28	0.97
UpdateOne simple (ProductById)	31.60%	1.97	1.28
UpdateOne complex (OrderById)	48.30%	2.5	1.3
Parallel query (N = 10000, DOP = 128)	0.50%	1.12	1

Across the scenarios summarized in "Table 2", PI favors configurations that reduce allocations even when latency differences are modest. For example, the parallel query case shows near parity in time while still reflecting allocation-driven improvements in the combined index, whereas the join-heavy and complex update cases combine large time advantages with sustained allocation reductions and therefore achieve the highest PI values.

"Fig. 7" aggregates allocation metrics across benchmark categories and highlights a consistent reduction in managed allocations for Dapper relative to EF Core variants.

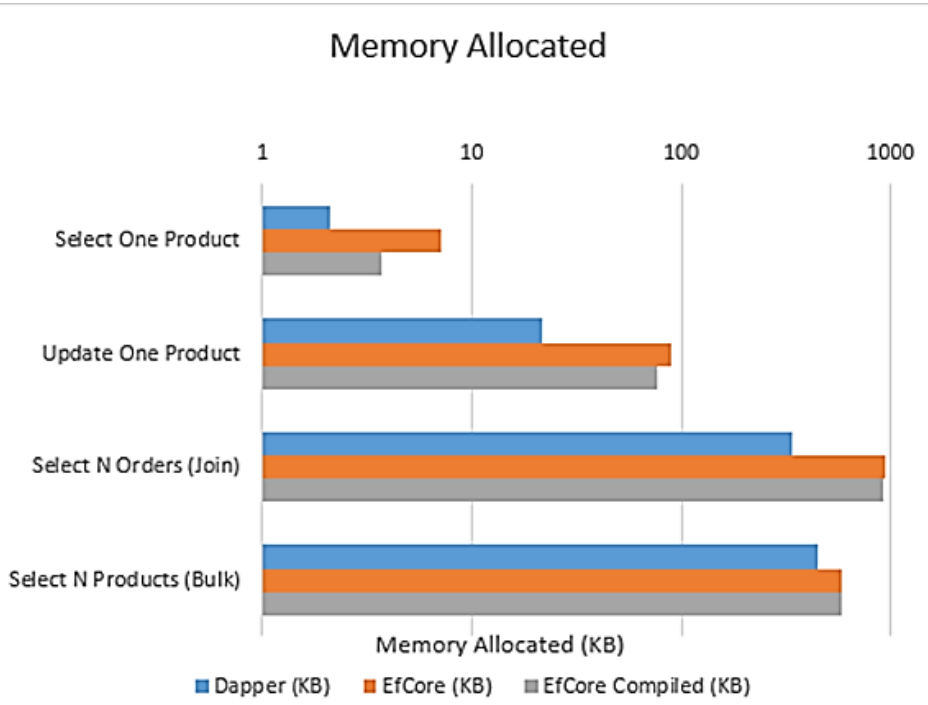

Figure 7. Managed memory allocation summary across benchmark scenarios

Discussion

The results support the hypothesis that Dapper reduces managed allocations and can improve mean latency in common read and update patterns. The strongest differences appear in join-heavy scenarios and complex updates, where EF Core pays additional costs for tracking and materialization across a larger object graph. This aligns with prior observations that ORMs provide developer productivity benefits while introducing measurable runtime overhead in some workloads. The modest impact of compiled queries in several scenarios suggests that query compilation addresses only a subset of EF Core costs, while change tracking and entity state management remain significant contributors in memory-intensive paths.

These findings should be interpreted with attention to engineering trade-offs. Dapper requires developers to manage SQL composition, parameterization, and mapping correctness, and this can increase maintenance effort and the risk of defects when schemas evolve. EF Core reduces those risks by providing richer abstractions, but the measured overhead implies that careful configuration is required in performance-constrained services. In particular, enforcing no-tracking for read-only paths and avoiding long-lived DbContext reuse patterns helps prevent caching artifacts and reduces hidden overhead in benchmarks and in production.

From an application design perspective, the results support a workload-specific configuration strategy rather than a single default for all services. For latency-sensitive read paths that return large result sets or join-expanded object graphs, a micro-ORM approach or EF Core projections that avoid tracking and minimize entity materialization can reduce both allocations and average response time [8]. For write-heavy paths, the overhead of change tracking and state management should be treated as a measurable cost, and alternative patterns such as explicit updates or narrower aggregate boundaries can be considered when the persistence layer becomes a bottleneck [12].

The focus on allocations is also motivated by operational considerations in managed runtimes. Under sustained load, elevated allocation rates can increase garbage collection frequency and, in turn, increase response time variance, which is often more disruptive to user-facing systems than changes in mean latency alone [2]. The derived performance index is intended to make this joint behavior visible when comparing implementations, while still allowing authors to choose weights that reflect their service-level objectives.

Threats to validity include the use of synthetic data, the reliance on a simplified schema, and the focus on one database system. Results may differ under alternative indexing strategies, network latencies, or domain-specific query shapes.

Conclusion

This paper presents a controlled comparison of Dapper and EF Core on PostgreSQL 17.6 across representative read, join-heavy, update, and high-concurrency workloads. Dapper demonstrates consistently lower managed allocations and often lower mean latency, with the largest benefits observed in complex object graph retrieval and update scenarios. EF Core compiled queries reduce latency for some reads but do not materially change allocation behavior in the tested configuration. The evidence supports a workload-driven approach to selecting a data-access technology, with Dapper favored when memory pressure and latency stability dominate and EF Core favored when development speed, maintainability, and richer persistence abstractions outweigh the measured overhead.

The benchmark source code and BenchmarkDotNet reports are published in GitHub repository [3].

During the source-code review performed for this study, the authors also identified and reported a redundant nullable-conversion operation in Dapper's asynchronous query path. This finding is mentioned as an implementation-level observation and is not included in the benchmark evaluation [5].

References:

1. Abdasameea A.A., Daduch T.S., Isbeeqah H.M., Shakeeb H.M. Comparative analysis of ADO.NET and Entity Framework performance evaluation in database operations // African Journal of Advanced Pure and Applied Sciences. — 2024. — Vol. 3, No. 1. — P. 11–15.
2. Akinshin A. Pro .NET Benchmarking: The Art of Performance Measurement. — Apress, 2019.
3. Asylkhan A., Sychev Y. efcore-vs-dapper: Version 1.0.0 [Electronic resource]. — GitHub, 2026. — URL: https://github.com/asylkhan-azat/efcore-vs-dapper/releases/tag/v1.0.0 (accessed: 11.06.2026).
4. BenchmarkDotNet contributors. BenchmarkDotNet documentation [Electronic resource]. — 2026. — URL: https://benchmarkdotnet.org (accessed: 11.06.2026).
5. Dapper GitHub Issue #2184: Unused value in QueryAsync<T> [Electronic resource]. — GitHub, 2025.— URL: https://github.com/DapperLib/Dapper/issues/2184 (accessed: 11.06.2026).
6. Ilić M., Denić N., Zlatković D., Stojanović J., Spasić B. The difference between ADO.NET and Entity Framework in software development // Annals of the University of Oradea. Fascicle of Management and Technological Engineering. — 2021. — Vol. 30, No. 2.
7. Microsoft. Entity Framework Core documentation [Electronic resource]. — 2026. — URL: https://learn.microsoft.com/ef/core (accessed: 11.06.2026).
8. Microsoft. Performance considerations (Entity Framework) [Electronic resource]. — 2026. — URL: https://learn.microsoft.com/en-us/ef/core/performance (accessed: 11.06.2026).
9. Nowicki T., Tomczak S., Kozieł G. Comparative analysis of the time performance of database queries in C# language // Journal of Computer Sciences Institute. — 2022. — Vol. 22. — P. 8–12. — DOI: 10.35784/jcsi.2772.

10. Npgsql contributors. Npgsql documentation [Electronic resource]. — 2026. — URL: https://www.npgsql.org/doc (accessed: 11.06.2026).
11. Smith J.P. Entity Framework Core in Action. — 2nd ed. — Manning Publications, 2021.
12. Wiatrowski T. Comparative analysis of ORM systems for the .NET platform // Journal of Computer Sciences Institute. — 2024. — Vol. 31. — P. 97–102. — DOI: 10.35784/jcsi.6012.
13. Zmaranda D., Pop-Fele L.-L., Győrödi C., Győrödi R., Pecherle G. Performance comparison of CRUD methods using .NET object relational mappers: a case study // International Journal of Advanced Computer Science and Applications. — 2020. — Vol. 11, No. 1. — P. 55–65. — DOI: 10.14569/ijacsa.2020.0110107.