ORMorpher: An Interactive Framework for ORM Translation and Optimization

Milan Abrahám
Department of Software Engineering
Charles University
Prague, Czech republic
milan.abraham882@student.cuni.cz

Pavel Koupil
Department of Software Engineering
Charles University
Prague, Czech republic
pavel.koupil@matfyz.cuni.cz

***Abstract*—Frequent changes in application requirements demand not only schema and query adaptation but also migration and optimization of the object-relational mapping (ORM) code. While database and query migration are well-studied, application-level translation across ORM frameworks remains largely overlooked. We present *ORMorpher*, a unified and extensible framework for translating and optimizing across heterogeneous ORMs. Unlike existing solutions, ORMorpher supports both structural code transformation and resource-aware framework selection under user-defined constraints. Although broadly applicable, we demonstrate its effectiveness on three widely-used .NET frameworks: Entity Framework Core, NHibernate, and Dapper over Microsoft SQL Server, enabling practical, performance-driven migration. The source code is available at: https://github.com/milan252525/orm-convertor. A demonstration video is available at: https://youtu.be/zwGGdqXtrzM.**

***Index Terms*—object-relational mapping, application-level migration, query rewriting, software evolution.**

I. Introduction

Modern software systems are subject to continuous evolution driven by shifting user requirements. These changes often affect not only application logic but also the structure of the underlying data and the queries executed over it. Consequently, developers must adapt data-access code in response to evolving schemas, altered query workloads, or transitions to different database management systems (DBMSs). Such transitions often involve data migration and require adjustments to ORM-based mapping and querying logic at the application level.

While the DBMS side of the problem has received considerable attention, particularly for data migration [1]–[3] and query synchronization [4], [5], these efforts primarily focus on rewriting queries for target DBMSs or synchronizing within a logical data model. Application-side adaptation, however, remains largely manual and error-prone, particularly within ORM layers such as entity mappings, repository classes, and query logic. This gap becomes critical when the original ORM framework, once optimal for a given workload, becomes inefficient or incompatible due to backend changes. In such cases, memory overhead, runtime performance, or even functional compatibility can be severely affected.

On the application side, the complexity is further amplified by the diversity of available ORMs. Each offers different abstractions, feature sets, and performance characteristics. However, this variety often forces developers to commit to a single framework, even when no option aligns well with all queries or the target DBMS. This mismatch can introduce overheads not due to the application logic itself but to limitations of the chosen abstraction. In such cases, a hybrid strategy – using multiple ORM frameworks for disjoint subsets of queries – can significantly improve runtime and memory efficiency.

Given recent success in applying large language models (LLMs) to tasks such as code generation and text-to-SQL translation [6]–[11], it is natural to consider their use for automating ORM framework translation. However, LLM-based techniques remain limited in practice: generated code may be incomplete, semantically incorrect, or depend on deprecated libraries. More critically, these models lack awareness of execution semantics, as they do not account for performance implications, resource constraints (e.g., memory), or generate code optimized for efficiency under specific workloads.

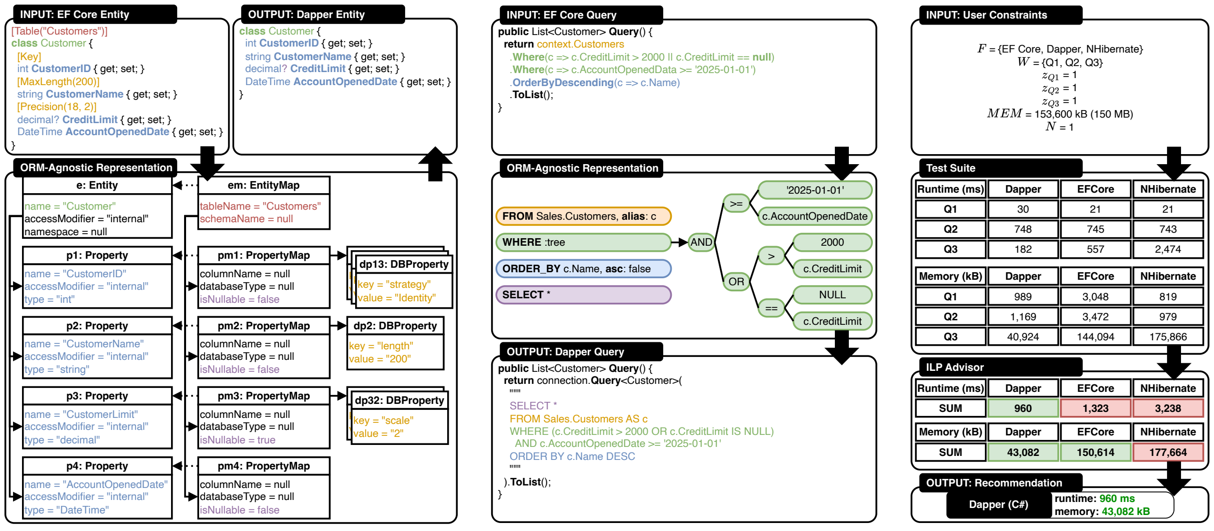
To address these challenges, we introduce *ORMorpher*, a unified, extensible framework that supports both ORM translation and performance-aware optimization under resource constraints. Our contributions are as follows:

- We introduce a unified framework for translating entity and repository classes between virtually any ORMs.
- We adopt a deterministic rule-based transformation approach instead of relying on (expensive) LLMs, which often produce incomplete code.
- We propose a unique optimization framework that selects the most efficient ORM, or combination thereof, based on queries and resource constraints, improving performance and reducing infrastructure cost.
- Since ORMorpher is designed to be framework-agnostic, we demonstrate its capabilities through a prototype implementation for three representative .NET ORMs: the full-featured Entity Framework Core (EF Core) and NHibernate, and the lightweight micro-ORM Dapper.

II. Framework ORMorpher

ORMorpher addresses the challenge of translating between ORM frameworks by unifying entities, mappings, and queries into a common abstraction. It follows a two-phase pipeline

Supported by the GAČR grant no. 23-07781S.



```
[Table("Customers")]
class Customer {
  [Key]
  int CustomerID { get; set; }
  [MaxLength(200)]
  string CustomerName { get; set; }
  [Precision(18, 2)]
  decimal? CreditLimit { get; set; }
  DateTime AccountOpenedDate { get; set; }
}
```

```
class Customer {
  int CustomerID { get; set; }
  string CustomerName { get; set; }
  decimal? CreditLimit { get; set; }
  DateTime AccountOpenedDate { get; set; }
}
```

(a)

```
public List<Customer> Query() {
  return context.Customers
    .Where(c => c.CreditLimit > 2000 || c.CreditLimit == null)
    .Where(c => c.AccountOpenedData >= '2025-01-01')
    .OrderByDescending(c => c.Name)
    .ToList();
}
```

```
public List<Customer> Query() {
  return connection.Query<Customer>(
    """
    SELECT *
    FROM Sales.Customers AS c
    WHERE (c.CreditLimit > 2000 OR c.CreditLimit IS NULL)
      AND c.AccountOpenedDate >= '2025-01-01'
    ORDER BY c.Name DESC
    """
  ).ToList();
}
```

(b)

F = {EF Core, Dapper, NHibernate}
W = {Q1, Q2, Q3}
z_{Q1} = 1
z_{Q2} = 1
z_{Q3} = 1
MEM = 153,600 kB (150 MB)
N = 1

Runtime (ms)	Dapper	EFCore	NHibernate
Q1	30	21	21
Q2	748	745	743
Q3	182	557	2,474

Memory (kB)	Dapper	EFCore	NHibernate
Q1	989	3,048	819
Q2	1,169	3,472	979
Q3	40,924	144,094	175,866

Runtime (ms)	Dapper	EFCore	NHibernate
SUM	**960**	**1,323**	**3,238**

Memory (kB)	Dapper	EFCore	NHibernate
SUM	**43,082**	**150,614**	**177,664**

(c)

Fig. 1: Overview of the interactive workflow: (a) Entity translation, (b) Query translation, and (c) ORM framework selection.

for transforming source representations into a framework-independent form and generating target-specific output, with an additional optimization phase that selects the most efficient framework configuration under user-defined constraints.

Example 1. *Figure 1 illustrates the main components of* ORMorpher*'s translation and optimization workflow. In Subfigure 1(a), the user provides entity definitions and mappings from a source ORM (e.g., EF Core), which are translated into a universal intermediate representation. This representation enables generation of equivalent mappings in other frameworks (e.g., Dapper) and lays the foundation for future support of alternative data models, including NoSQL. Subfigure 1(b) shows query translation: user-defined queries (e.g., LINQ) are transformed into structured intermediate instructions, which can be compiled into target representations such as raw SQL, while remaining extensible to other query languages (e.g., Cypher). Finally, Subfigure 1(c) demonstrates performance-aware optimization. Given user-defined criteria, including target frameworks, memory constraints, and query weights,* ORMorpher *evaluates available ORMs and recommends the most efficient framework.* □

A. Entity and Mapping Translation

Entity and mapping translation in *ORMorpher* is structured as a two-phase transformation pipeline: from source ORM representations to an abstract intermediate model, and from this model to a target ORM. Although ORM frameworks differ in syntax and configuration style, they share common structural principles: application classes typically represent database tables, class properties correspond to columns, and object associations reflect foreign key relationships. The abstraction used in *ORMorpher* builds on the category-theoretic model proposed in [1], where tables map to products and columns to projections, supporting universal transformations across heterogeneous data models. These commonalities enable a framework-agnostic model that captures essential mapping semantics while abstracting away representational differences.

This intermediate model is explicitly implemented in *ORMorpher* through a set of modular components. *Entity* and *EntityMap* capture table-level mappings and associated metadata; *Property* and *PropertyMap* describe individual column mappings; and *DBProperty* encodes auxiliary features such as precision, length, and nullability as key-value pairs. Serving as the central abstraction layer, this representation unifies mapping semantics across ORM frameworks and lays the groundwork for future extensions beyond relational systems, including document- and graph-based data models.

The first phase of the pipeline populates the intermediate model by transforming source code using a modular parser tailored to each supported ORM. Each parser employs a variant of the Visitor design pattern to traverse entity and mapping definitions, identifying semantic roles such as tables, properties, primary keys, data types, and relational constraints. These are encoded through a unified interface of declarative methods like *addTable()*, *addProperty()*, and *addPrimaryKey()*. When source definitions lack detail, the model can be enriched by querying the underlying DBMS for inferred metadata, such as constraints omitted by lightweight frameworks.

In the second phase, the intermediate model is translated into target ORM syntax using framework-specific code generators. These generators implement the Builder design pattern, applying methods such as *buildTable()*, *buildProperty()*, and *buildPrimaryKey()* to map abstract structures into concrete syntax. A central build method coordinates this process, ensuring that the output conforms to the syntactic and semantic rules of the target framework.

B. Query Translation

Query translation in ORMorpher follows a similar abstract pipeline, transforming queries from a source ORM into a unified intermediate representation, and then generates equivalent queries in the target framework. Although ORM frameworks differ significantly in how queries are expressed, e.g., ranging from raw SQL strings to composable LINQ expressions, these approaches ultimately express the same core operations: filtering, projection, joins, grouping, ordering, and set operations. This semantic overlap makes it possible to define a framework-independent abstraction that isolates the logical intent of a query from the specifics of its API or syntax. However, unlike SQL, LINQ queries are embedded in host language code and often depend on contextual elements such as method parameters or entity mappings, requiring deeper integration with the surrounding program structure.

To support general-purpose query translation, *ORMorpher* introduces a structured, instruction-based intermediate representation. Each query is modeled as a list of typed instructions, including *FROM*, *PROJECT*, *SELECT*, *JOIN*, *GROUP_BY*, *HAVING*, *ORDER_BY*, and *set operations*. Filtering conditions are represented as trees, capturing logical and comparison operators over columns, constants, or nested subqueries. Nested constructs are supported through *SUBQUERY* instruction. This representation builds on the instruction-based abstraction proposed in [5] and is designed for extensibility toward additional query languages common in NoSQL systems.

The translation algorithm consists of two phases. First, the parser analyzes repository methods or standalone queries in the source framework. For EF Core and NHibernate, .NET Roslyn is used to traverse the syntax tree of C# methods.[1] Visitors specialized for LINQ constructs – such as *Where*, *OrderBy*, and *Select* – interpret the method calls and emit corresponding abstract instructions. Metadata from the previously parsed entity mappings is used to resolve table and column names. Second, the builder transforms the instruction list into the target framework's syntax. For Dapper, this involves reordering and combining instructions into valid SQL using a visitor-based SQL generator. For EF Core and NHibernate, the builder reconstitutes LINQ queries as chained method calls, respecting framework-specific conventions and naming.

C. ORM Advisor

To support performance-aware migration between ORM frameworks, *ORMorpher* integrates an optimization module that selects the most suitable ORM framework, or combination thereof, for executing a given query workload under specified resource constraints. This task is inherently non-trivial, as ORM frameworks differ in their runtime and memory behavior. While syntactic translation ensures semantic equivalence across frameworks, achieving optimal performance requires systematic selection based on empirical evidence. *ORMorpher* addresses this challenge by formulating ORM selection as an instance of *Integer Linear Programming* (ILP), enabling the derivation of configurations optimized for execution time, memory usage, and user-defined limits.

[1]https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/

Prior to optimization, *ORMorpher* conducts a *what-if analysis* to empirically evaluate the performance of each candidate ORM framework. Specifically, every query $q \in W$ is executed using each framework $f \in F$, including the original ORM, against a common DBMS. For each query-framework pair (q, f), the system records runtime $\text{cost}(q, f)$ and memory usage $\text{mem}(q, f)$. These measurements, along with user-specified query frequencies z_q, a global memory limit MEM, and a cap N on the number of frameworks to use, provide the input to the optimization phase.

The ORM framework selection is then formulated as an instance of ILP. To encode the decision space, we introduce binary variables: $x_{q,f} \in 0, 1$ indicates whether query q is assigned to framework f, and $y_f \in 0, 1$ denotes whether framework f is selected globally. The objective is to minimize the total weighted execution cost:

$$\min \sum_{q \in W} \sum_{f \in F} z_q \cdot \text{cost}(q, f) \cdot x_{q,f}$$

subject to the following constraints:

$$\sum_{f \in F} y_f \leq N \tag{1}$$

$$\sum_{f \in F} x_{q,f} = 1 \qquad \forall q \in W \tag{2}$$

$$x_{q,f} \leq y_f \qquad \forall q \in W,\ \forall f \in F \tag{3}$$

$$\sum_{q \in W} \sum_{f \in F} \text{mem}(q, f) \cdot x_{q,f} \leq \text{MEM} \tag{4}$$

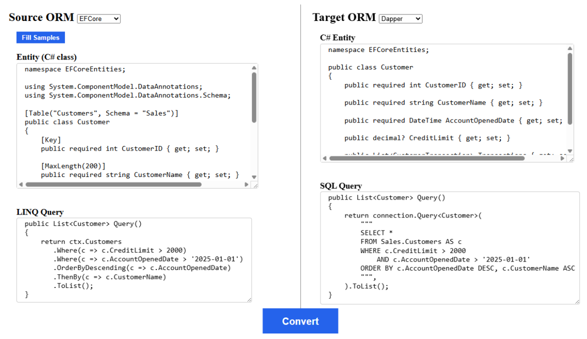
These constraints ensure that (1) the selected configuration uses no more than N frameworks, (2) assigns each query exactly once, (3) only uses frameworks that are selected globally, and (4) does not exceed the total memory budget.

D. Implementation and Extensibility

ORMorpher is implemented as a modular client-server architecture, with a React-based frontend (see Figure 2) and a .NET backend. The optimization component leverages the GLPK solver. The system was designed with extensibility as a core principle, enabling future support for additional ORM frameworks, programming languages, DBMS platforms, and constraint types. At its core, the architecture employs the *Adapter design pattern*, where each ORM framework is encapsulated via a dedicated wrapper. These wrappers abstract framework-specific features and enable uniform transformation logic. The modular structure allows straightforward integration of new wrappers to support other ecosystems, such as Java-based ORMs, and other test suites beyond .NET. Furthermore, the architecture supports extending the ILP formulation with additional constraints (e.g., latency thresholds or energy budgets), and combining framework and DBMS selection.

III. Preliminary Evaluation

To evaluate the correctness of *ORMorpher*, we conducted a preliminary study using Microsoft SQL Server and three



```
namespace EFCoreEntities;

using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

[Table("Customers", Schema = "Sales")]
public class Customer
{
    [Key]
    public required int CustomerID { get; set; }

    [MaxLength(200)]
    public required string CustomerName { get; set; }
```

```
public List<Customer> Query()
{
    return ctx.Customers
       .Where(c => c.CreditLimit > 2000)
       .Where(c => c.AccountOpenedDate > '2025-01-01')
       .OrderByDescending(c => c.AccountOpenedDate)
       .ThenBy(c => c.CustomerName)
       .ToList();
}
```

```
namespace EFCoreEntities;

public class Customer
{
    public required int CustomerID { get; set; }

    public required string CustomerName { get; set; }

    public required DateTime AccountOpenedDate { get; set;

    public decimal? CreditLimit { get; set; }
```

```
public List<Customer> Query()
{
    return connection.Query<Customer>(
        """
        SELECT *
        FROM Sales.Customers AS c
        WHERE c.CreditLimit > 2000
            AND c.AccountOpenedDate > '2025-01-01'
        ORDER BY c.AccountOpenedDate DESC, c.CustomerName ASC
        """,
    ).ToList();
}
```

Fig. 2: Translation of EF Core to Dapper.

supported ORM frameworks. As a benchmark, we used the publicly available *Wide World Importers* database.[2] Representative queries[3] were selected to cover common SQL patterns: range query **Q1**,[4] text search **Q2**,[5] and optional join **Q3**.[6]

We measured runtime and memory usage for each query using BenchmarkDotNet.[7] Using EF Core entity and repository classes as input, we then ran *ORMorpher* with the ILP configured to select a single target ORM under a 150 MB memory constraint and equal query weights.

ORMorpher selected Dapper as the optimal framework. As shown in Table I, this choice aligns with empirical results and confirms improvements in execution time and memory usage.

TABLE I: Performance validation.

	Mean Time (ms)			**Allocated memory (kB)**		
Query	Dapper	NHibernate	EF Core	Dapper	NHibernate	EF Core
Q1	30	21	21	989	3,048	819
Q2	748	745	743	1,169	3,472	979
Q3	182	557	2,474	40,924	144,094	175,866
SUM	960	1,323	3,238	43,082	150,614	177,664

IV. Related Work

Prior work on managing schema and query changes has primarily focused on the database layer, including data migration [1]–[3] and query synchronization techniques [4], [5]. In contrast, adaptation at the application level – particularly within ORM frameworks – has received little attention. Recent advances in large language models (LLMs) have demonstrated potential in tasks such as code generation and text-to-SQL translation [6]–[11], but these approaches often produce incomplete or outdated code. In contrast, ORMorpher employs a direct parser-builder architecture for source-level translation and integrates an ILP-based optimization phase to select the most efficient ORM configuration under resource constraints. To the best of our knowledge, no existing work combines source code transformation with cost-aware framework selection in this manner.

[2]https://learn.microsoft.com/en-us/sql/samples

[3]For space reasons, we report results on three representative queries; a broader evaluation is presented in [12].

[4]SELECT * FROM OrderLines WHERE PickingCompletedWhen BETWEEN ’2014-12-20’ AND ’2014-12-31’

[5]SELECT * FROM OrderLines WHERE Description LIKE ’%C++%’

[6]SELECT c.*, ct.* FROM Customers c LEFT JOIN CustomerTransactions ct ON c.CustomerID = ct.CustomerID ORDER BY c.CustomerID

[7]https://benchmarkdotnet.org

V. Conclusion

In the demonstration, we presented *ORMorpher*, a novel and interactive framework for exploring and optimizing ORM-layer translations across supported frameworks (EF Core, NHibernate, and Dapper) on Microsoft SQL Server backends. Users begin with custom inputs and are guided through translation, constraint specification, and optimization. The system generates ORM mappings, source code, and SQL queries, then evaluates target representations under user-defined resource limits. Results are presented in a comparative dashboard featuring performance metrics and framework recommendations, enabling users to iteratively refine queries and immediately observe the impact on translation and execution.

Future work will extend *ORMorpher* with support for dynamic schema evolution, enabling the seamless propagation of structural changes from the database to ORM mappings and query logic. This enhancement will strengthen its applicability in adaptive, long-lived systems where data models continuously evolve.

Acknowledgment

We would like to thank Roman Barták for consultations. This work is based on the thesis [12].

References

[1] P. Koupil and I. Holubová, “A unified representation and transformation of multi-model data using category theory,” *J. Big Data*, vol. 9, no. 1, p. 61, 2022.

[2] U. Störl and M. Klettke, “Darwin: A data platform for schema evolution management and data migration,” in *Proceedings of the Workshops of the EDBT/ICDT 2022 Joint Conference, Edinburgh, UK*, 2022.

[3] I. Holubová, M. Vavrek, and S. Scherzinger, “Evolution Management in Multi-Model Databases,” *Data Knowl. Eng.*, vol. 136, p. 101932, 2021.

[4] L. Caruccio, G. Polese, and G. Tortora, “Synchronization of queries and views upon schema evolutions: A survey,” *ACM Trans. Database Syst.*, vol. 41, no. 2, pp. 9:1–9:41, 2016.

[5] P. Koupil, D. Crha, and I. Holubová, “A universal approach for simplified redundancy-aware cross-model querying,” *Inf. Syst.*, vol. 127, 2025.

[6] G. Katsogiannis-Meimarakis and G. Koutrika, “A survey on deep learning approaches for text-to-sql,” *VLDB J.*, vol. 32, no. 4, 2023.

[7] M. H. Hassan, O. A. Mahmoud *et al.*, “Neural Machine Based Mobile Applications Code Translation,” in *Proc. of NILES 2020*. IEEE, 2020.

[8] B. Rozière, M. Lachaux, L. Chanussot, and G. Lample, “Unsupervised Translation of Programming Languages,” in *Proc. of NeurIPS 2020 (virtual)*, 2020.

[9] U. Alon, S. Brody, O. Levy, and E. Yahav, “code2seq: Generating Sequences from Structured Representations of Code,” in *Proc. of ICLR 2019*. OpenReview.net, 2019.

[10] B. Wang, R. Shin, X. Liu, O. Polozov, and M. Richardson, “RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers,” in *Proc. of ACL 2020*. ACL, 2020, pp. 7567–7578.

[11] C. Wang, A. Cheung, and R. Bodík, “Synthesizing Highly Expressive SQL Queries from Input-Output Examples,” in *Proc. of PLDI 2017*. ACM, 2017, pp. 452–466.

[12] M. Abrahám, “Framework-Agnostic Query Adaptation: Ensuring SQL Compatibility Across .NET Database Frameworks,” Master Thesis, Charles University in Prague, Czech Republic, 2025.