Efficient handling of recursive relationships in ORM frameworks using Entity Framework Core as an example

Benjamin Uwe Killisch[1] Florian Scheffler[2] Thomas Kudraß[3]

Abstract: ORM frameworks are a popular method to bridge the differences between object-oriented programming and relational data management. At the same time, recursive relationships are present in many schemas to represent tree-like or net-like structures. This paper discusses how to efficiently build and execute queries for data with recursive relationships in ORM frameworks, mainly Entity Framework Core (EF Core). Five possible solutions are conceived and implemented, while making sure that they can be used like regular LINQ queries. Next, the solutions are tested with different SQL dialects. The results of these tests are then analyzed by a variety of test parameters. This analysis shows that queries with recursive common table expressions and queries using key loading are the most efficient. Queries with auxiliary property, vertical unrolling or horizontal unrolling, are either too slow or only usable under particular circumstances. The analysis also shows that the performance of the solutions is always dependent on the circumstances, especially the SQL dialect.

Keywords: Object-Relational Mapping; Recursive; Relationships; Queries

1 Introduction

Object-oriented programming and relational databases are currently the most popular paradigms for programming and persistent data storage. Since they are different, object-relational mapping (ORM) is needed to bridge their differences when using them together. One challenge when using an ORM framework is loading data with recursive relationships from the database. The motivation for this paper was the challenge of loading article attributes in a recursive relationship for a REST API developed by the e-commerce company Relaxdays GmbH. Performance is critical, because hundreds of articles have to be updated per day. The topic of recursive queries has already been covered by Szumowska et al. [Sz11] for the framework Hibernate. This paper will cover Entity Framework Core (EF Core) for C# [Mi21]. EF Core can automatically map classes and their relationships to a database, and convert the results of database queries into objects. Its most distinguishing feature is LINQ (Language-Integrated Query) [Mi22a]. Using LINQ, one can create database queries using method chains and type-safe lambda expressions based on the classes mapped to the database. There is no need to write SQL queries as strings, and therefore, EF Core can be used independently of the underlying database.

[1] HTWK Leipzig, Fakultät Informatik und Medien, benjamin_uwe.killisch@stud.htwk-leipzig.de
[2] Relaxdays GmbH, florian.scheffler@relaxdays.de
[3] HTWK Leipzig, Fakultät Informatik und Medien, thomas.kudrass@htwk-leipzig.de

doi:10.18420/BTW2023-78

2 Solutions

There are multiple requirements for possible solutions. Solutions should be as fast as possible, while also being convenient to use. They should also be able to handle cyclical data and still terminate while returning the correct result. In the context of EF Core, the solutions should function regardless of the SQL dialect. Furthermore, they should accept two lambda expressions, one for the initial condition and one for the navigation property of the recursive relationship. Navigation properties represent relationships between two classes that are mapped to tables in the database via EF Core. The first class will have a navigation property that is a single reference or a list of the second class. An example of this is shown in Listing 1. The query will be created for the recursive relationship represented by the navigation property `employee.Subordinates`.

List. 1: Usage of the solution implemented in section 2.1.

```
context.Employees.RecursiveQuery(context, employee => employee.EmployeeId == 1, employee =>
    employee.Subordinates);
```

2.1 Recursive CTEs

Common Table Expressions (CTEs) were specified in the SQL:1999 standard and allow for queries to be named and reused again later, as described by Heuer et al. [HSS18]. CTEs can be used as recursive queries by adding the **RECURSIVE** keyword. A recursive CTE consist of the initial query, the recursive query, and the actual query. Only the recursive query can reference the CTE it is a part of. If we use this to join the CTE with the table we want to query, we can query the table recursively. The recursive CTE can then be referenced in the actual query. There are some problems with this solution, the main one being that the syntax of recursive CTEs is not standardized across the different SQL dialects [SQ22] [Or22] [Po22] [Mi22b]. Furthermore, not all SQL dialects can handle cyclical data properly. Out of the four dialects examined in this paper, Transact-SQL can not, but the other three can. The query will terminate with an error after reaching the maximum recursion depth of 1000. Furthermore, CTEs can not be used as subqueries in Transact-SQL. Recursive CTEs can be used in an ORM framework like any other query, although they are more complex. It gets more complicated when using a framework like EF Core. The following steps are executed to transform two lambda expressions into a recursive CTE:

1. The navigation property of the recursive relationship needs to be read from the first lambda expression. EF Core stores the mapping of the classes to the database schema, including the relationships between tables, so the columns of the relevant table and the corresponding primary keys and foreign keys can be loaded from there.
2. The second lambda expression, which contains the initial query, must be converted into a query string. For that purpose, EF Core provides the `ToQueryString()` method.

3. The resulting string must be slightly adjusted to account for parameterized queries. The values of query parameters must also be read from the second lambda expression.
4. Based on the type of the relationship (1:1, 1:n, m:n) and the schema data loaded in the first step, the relevant table names, column names and key comparisons are selected.
5. The final query string is built while considering the used SQL dialect.
6. The final query string and the parameter values are passed to FromSQLRaw().

FromSQLRaw() returns an instance of IQueryable (the interface used to query data sources), which can be chained to further LINQ methods to extend the query. However, EF Core only allows this if the query string passed to FromSQLRaw() starts with **SELECT**. A CTE always starts with the keyword WITH, so the query string has to be surrounded with **SELECT** $*$ **FROM** (query). But, as mentioned above, CTEs may not be used as subqueries in Transact-SQL. So, if Transact-SQL is used, the query is immediately evaluated, and an IN subquery based on the returned objects is created. Not only does this worsen the performance, but it also violates lazy evaluation, which is usually expected from instances of IQueryable in C#. An alternative would be to keep track of the search path in an additional column, and to check that column in the where clause of the recursive query. This, however, has not been implemented for this paper.

2.2 Vertical Unrolling

Boniewicz et al. [BSW12] describe alternatives to CTEs for recursive queries. One of them is vertical unrolling, which works by combining multiple **LEFT JOIN**s. The number of joins is equal to the maximum recursion depth of the table. In the best-case scenario, this depth is known. Otherwise one must make an educated guess with the risk of incomplete query results. The size of the query increases with the depth of the recursive structure, which can make this very impractical to implement when using query strings. In EF Core, however, this solution can be implemented easily with a loop and the ThenInclude() method. In EF Core, Include() and ThenInclude() are used to load related data and are translated to **JOIN**s. The query method accepts the recursion depth as an additional parameter and calls ThenInclude() accordingly.

2.3 Horizontal Unrolling

In addition to vertical unrolling, Boniewicz et al. [BSW12] also describe horizontal unrolling. Horizontal unrolling creates a temporary table for each level of the recursive structure, creating each one based on the previous table. These temporary tables are then combined using **UNION**. Similarly to vertical unrolling, this approach also requires knowledge about the maximum recursion depth. It can be implemented using a loop, and the resulting

queries will be simpler than vertical unrolling. However, temporary tables are not convenient to use in EF Core. Among other things, the name and columns of a temporary table must be known at compile time, making them impractical since one needs a specific temporary table for each entity and recursion level. Because of this, a different approach is implemented in this paper. The queries for the temporary tables are all executed immediately. The resulting entities are then used to create the next query. In the end, a final IN subquery is created based on all the loaded entities. This approach creates a lot of database roundtrips, but makes the recursion-depth parameter obsolete. Instead, a new query is only executed if the last query returned at least one unknown entity.

2.4 Using an auxiliary property

In some cases, the table with the recursive relationship has a column for which all records of the same recursive structure share the same value. Such a column can be used to create a simple recursive query. First, all the values of the aforementioned column (the auxiliary property) are loaded for the records that match the initial query. Then, an IN subquery for the auxiliary property is executed based on the loaded values. This is exactly how this approach is implemented in EF Core. While this approach is simple, it has a few drawbacks:

1. Such a column must either exist already in the current schema or be added to it. If it is added, it also needs to be maintained on every insert or update operation.
2. The auxiliary column should have an index to improve the performance.
3. This approach always loads the entire recursive structure, even if only a part of it is required.
4. To use this approach for an m:n relationship, one would have to create an additional table that links every record in the original table to the roots of the structures it is a part of. Since this would make the query more complex, and such a table would be much harder to maintain than just a column, this approach is only implemented for 1:1 and 1:n relationships.

2.5 Key loading

The idea of key loading is to load the values of the primary and foreign keys of all records in the table, and then use this data to find the primary key values of the records in the recursive structures to load. For this, the primary keys of the records matching the initial query must also be loaded. Finally, the relevant records are loaded with an IN subquery. This approach uses the fact that object-oriented programming languages are turing complete, while SQL is not. A disadvantage of this approach is that a lot of data could be loaded unnecessarily if only a small amount of records of a big table is required. Usually, one can easily implement

this approach with three queries. In EF Core, it is a bit more complex, mainly because simple primary keys must be handled as well as composite primary keys. To do this, both kinds of keys are represented as value tuples. This has the advantage that C# compares value tuples by value, not by reference.

3 Performance comparison

Each one of the described approaches was tested in many different scenarios, varying in SQL dialect (MySQL, SQLite, PostgreSQL, Transact-SQL), relationship type (1:1, 1:n, m:n), recursion depth (3, 4), the amount of branches per layer (2, 5, always 1 for 1:1 relationships) and the amount of recursive structures in the database (5, 10, only one structure was loaded per query execution). Every table usually had one or two columns additionally to the key columns. Every approach was repeated and measured 150 times for each resulting scenario. For vertical unrolling, the correct recursion depth was passed to the query method. This is the ideal scenario, while in reality one might have to use a higher maximum recursion depth, to make sure to always load all the data. All tests were executed using an Intel® Core™ 5 i7-10510U Processor and 32 GB RAM. The databases for MySQL, Transact-SQL and PostgreSQL were hosted in docker containers, while the SQLite database was stored in a file. The results were then analyzed by the different parameters. Since the auxiliary property approach is not implemented for m:n relationships (cf. section 2.4), there is always one analysis for all approaches and 1:1/1:n relationships and another without the auxiliary property approach but for all relationship types. Figure 1 and 2 show the analysis results by SQL dialect with and without the auxiliary property.

Fig. 1: Average query duration, by SQL dialect, for 1:1 and 1:n relationships

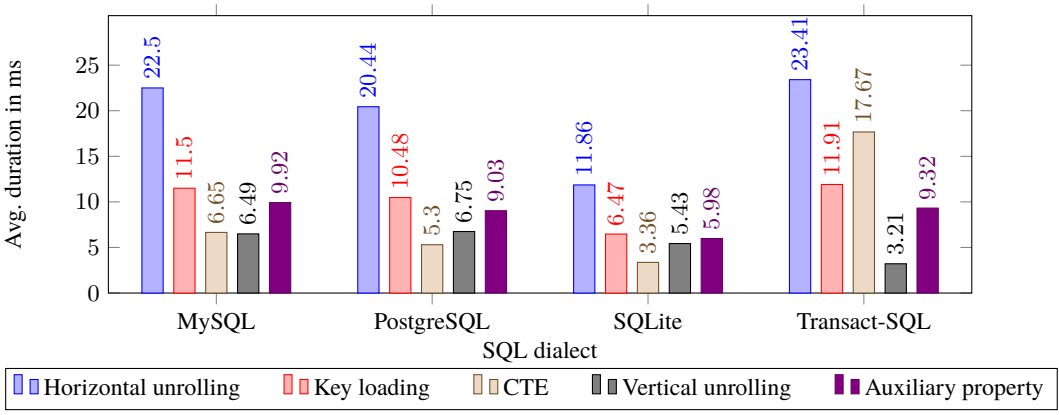
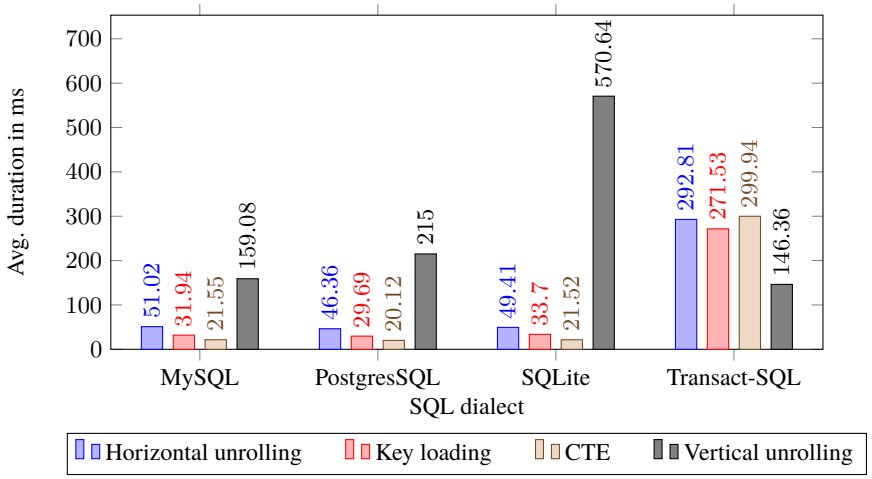


Fig. 2: Average query duration, by SQL dialect, for 1:1, 1:n and m:n relationships



The figures show that recursive CTEs are a lot slower when using Transact-SQL, as it was anticipated. They also show a pattern consistent across all results: When there is no join table (1:1 and 1:n relationships), and the recursion depth is low, vertical unrolling is quite fast. But vertical unrolling becomes very slow when there is a join table (m:n relationship), especially when using SQLite. The same goes for higher recursion depth, although the corresponding analysis is not shown here for brevity. This slowdown happens because the number of join s increases in both cases. Also, horizontal unrolling and queries with auxiliary property are usually slower than at least one other approach, while key loading is usually faster than those two methods but slower than recursive CTEs.

4 Conclusion and further research

Recursive CTEs are the fastest approach in many scenarios; even when they are not, they are usually not far behind. They also scale well with larger amounts of data. The same goes for key loading, but it is a bit slower. For these reasons, recursive CTEs are the most efficient solution, but depend on the used SQL dialect. They should not be used with Transact-SQL, because of the slow performance and the missing handling of cyclical data. One should use key loading instead if one wants to avoid these disadvantages. If smaller amounts of data and a low recursion depth can be assumed, one can also use vertical unrolling, since it performs well in these situations. The use of horizontal unrolling can not be recommended since there is no scenario where it is the fastest. Queries with auxiliary property are also not recommended since this approach can only be used in specific situations and is also slower than vertical unrolling in many of them. Further research could be done using further SQL dialects like PL/SQL, and/or using another ORM Framework. More extensive testing could also be done with higher recursion depths and more entities.

References

[BSW12] Boniewicz, A.; Stencel, K.; Wiśniewski, P.: Unrolling SQL: 1999 Recursive Queries. In (Kim, T.-h.; Ma, J.; Fang, W.-c.; Zhang, Y.; Cuzzocrea, A., eds.): Computer Applications for Database, Education and Ubiquitous Computing. Springer, 2012.

[HSS18] Heuer, A.; Saake, G.; Sattler, K.-U.: Datenbanken: Konzepte und Sprachen. mitp, 2018.

[Mi21] Microsoft: Entity Framework Core, 2021, URL: https://learn.microsoft.com/en-us/ef/core/, visited on: 12/15/2022.

[Mi22a] Microsoft: Language Integrated Query (LINQ) (C#), 2022, URL: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/, visited on: 12/15/2022.

[Mi22b] Microsoft: WITH common_table_expression (Transact-SQL), 2022, URL: https://docs.microsoft.com/en-us/sql/t-sql/queries/with-common-table-expression-transact-sql?view=sql-server-ver16, visited on: 12/15/2022.

[Or22] Oracle: 13.2.15 WITH (Common Table Expressions), 2022, URL: https://dev.mysql.com/doc/refman/8.0/en/with.html, visited on: 12/15/2022.

[Po22] PostgreSQL-Global-Development-Group: 7.8. WITH Queries (Common Table Expressions), 2022, URL: https://www.postgresql.org/docs/14/queries-with.html, visited on: 12/15/2022.

[SQ22] SQLite-Team: The WITH Clause, 2022, URL: https://www.sqlite.org/lang_with.html, visited on: 12/15/2022.

[Sz11] Szumowska, A.; Boniewicz, A.; Burzańska, M.; Wiśniewski, P.: Hibernate the Recursive Queries - Defining the Recursive Queries Using Hibernate ORM. In (Eder, J.; Bielikova, M.; Tjoa, A. M., eds.): 15th East-European Conference on Advances in Databases and Information Systems. Faculty of Mathematics and Computer Science, Nicolaus Copernicus University, Toruń, Poland, 2011.