Design of Data Access Architecture Using ORM Framework

Filip Majerik
University of Pardubice
Pardubice, Czech Republic
filip.majerik@student.upce.cz

Monika Borkovcova
University of Pardubice
Pardubice, Czech Republic
monika.borkovcova@upce.cz

***Abstract*—Nowadays, various Object–relational mapping frameworks are becoming a key part of computer system architecture. These frameworks provide developers with relatively easy manipulation of data stored in various database systems, even without knowledge of complex database systems. In this article, we have focused on leveraging the benefits of implementing an ORM framework while minimizing the impact on software performance. The design of an intelligent data intermediate layer is described within this paper. This provides optimized communication between the application layer and subsequently the ORM Framework. At the same time, attempts have been made to extend the layer with an additional caching layer, which however proved to be unhelpful for simple SQL queries.**

I. Introduction

Nowadays, software application developers are facing a lot of challenges. Due to the ever-increasing demands on the performance of the resulting application systems, the speed of handling gradually increasing requests, the amount of requests handled with the highest possible system response, combined with rapid agile development, it is practically not in the power of an individual to be able to know all optimization techniques at such a good level to be able to build such a system at all. [3] Of course, the development of production application systems creates many developers that collaborate on the final product. However, the benefit of working in a team is greatly influenced by the practices and standards used in each organization. Our investigation shows that many developers resort to using a variety of off-the-shelf solutions, such as ready-made complex frameworks. [1], [13], [6]

One group of these frameworks are, for example, Object-Relational Mapping (ORM) Frameworks. [6] These are focused on manipulating data that is stored in a database. The simplicity of working with these frameworks is redeemed (trade-off) very often by the performance of the application. [18] Developers using these frameworks often have no idea how to optimize the communication with the database and work with it efficiently. Thus, on the one hand we are faced with the ignorance of developers [15] and on the other hand we are faced with the limitations of ORM frameworks as such. It is necessary to realize that they are still only SQL code generators based on the given annotations and end-user actions, combining the object-oriented paradigm with the relational one and allowing faster development of computer systems. [14], [5], [17], [8], [11]

ORM frameworks have spread over time to all commonly used programming languages and there are many of them. For JAVA, Hibernate [12], the .NET Entity Framework, Python Django, GoLang GORM, NodeJS TypeORM, and PHP Doctrine. These frameworks offer basically the same thing, namely ease of working with a database system where the developer does not need to understand the internal mechanisms of relational databases in any fundamental way. [6], [5], [2]

Thus, in this paper, we focus on building an intermediate layer that will take advantage of the benefits of ORM, while trying to minimize the impact on software performance and speed. Thus, the developer will not lose the benefit of being able to continue working without the complexities of the object-oriented paradigm. Thus, this includes a description of the design of an intelligent data layer (IDL) that allows the developer to predictably define user behavior and subsequently reduce the need for database calls, thereby increasing the responsiveness of the system. The proposed intermediate layer has been tested in PHP 8.1, Symfony 6.3, Doctrine 2.6, MySQL 8.0.26 environments and the standard PHP extension Memcached has been used for caching, which works on the key-value principle and allows storing entire objects. [10], [16] The entire environment will be used in its default configuration, with no additional caching or other settings to improve performance. And it will be run using the Docker container tool within Ubuntu 23.04 [7], [9].

II. The Problem of ORM Frameworks

The fundamental problem of ORM frameworks and their subsequent implementation is that they cannot estimate in any way what data will be needed when processing a user request. Thus, the decision whether to use the so-called EAGER loading or LAZY loading when defining sessions at the Entity level can be particularly problematic. The main difference between these definitions is when the required data is retrieved. If the session is designated as LAZY, the data is loaded when it is needed. In contrast, for a session marked as EAGER, they are retrieved immediately at the moment the entity is retrieved from the database. Thus, EAGER retrieval might seem more convenient and efficient enough, but there is no way for a developer like ORM Framework to "say" that he does not need this data for an operation, until we are working with hundreds of records. To developers who have no idea about the internal processes of the ORM Framework, then, it might seem that the solution to this problem is to label the constraints as LAZY. However, the problem that arises here is that this data must be retrieved for each entity separately. Of course, if the ORM Cache for entities

is enabled, there is at least a minimization of repeated querying of the already loaded database entity.

III. EXPERIMENTS

A. Model for experiments

For the following sections, it is already necessary to have a model application over which the individual functionalities and principles of the intelligent data layer can be easily explained and verified. As a model application, a section from the administration system of the IPTV/OTT platform was selected and the database for the experiments, then populated with anonymized real data from this system.

The figure 1 shows the entity-relational diagram (ERD) of the model database.

The ERD was then transformed into a corresponding relational database model (figure 2) that corresponds to the actual experimental database. Both models were created using the code-first method, where the complete application code was first created and then the data models were built. [4]

The ERD thus expresses the structure of the experimental database, where Operator has the id column as the primary key, followed by the name and dataCreated columns. Subscriber has id as primary key, email column works with unique values and operator_id column is marked as foreign key referring to the primary key (PK) in Operator table for PK id. Subscriber also has non-key columns name, surname, phone and email. The Brand table is joined to the Operator and Device tables, with the Brand table having the id column as the PK, the unique column being code, and the foreign key (FK) in this table being operator_id referencing the PK in the Operator table for the id column. The Device table has the id column as the PK, and the brand_id, subscriber_id, deviceProfile_id, and deviceType_id columns as the FK. The DeviceType table has the id column as PK and the code column as unique. The DeviceProfile table has the id column selected as the PK and the code column as the unique one. All relationships are of type 1:N i.e. Operator:Subscriber, Operator:Brand, Brand:Device, Subscriber:Device, DeviceType:Device, DeviceProfile:Device.

All the entities in the ERD and the subsequent tables in the relational model have their corresponding image in the application code. Due to the extensiveness of these models, only an example (figure 3) of the most complex entity Device is given here, trimmed down so that the annotations related to the ORM in particular are clear. In the code of the Device class, we can see the work with the Device table and its FKs in the variables $brand, $subscriber, $deviceType, $deviceProfile, as well as $id as PK and the non-key columns name ($name), macAddress ($macAddress), lastStart ($lastStart), dateCreated ($dateCreated).

```php
<?php

#[ORM\Entity(repositoryClass: DeviceRepository::class)]
#[UniqueEntity(['macAddress'])]
#[ORM\Index(columns: ['brand_id'], name: 'device_brand_idx')]
#[ORM\Index(columns: ['subscriber_id'], name: 'device_subscriber_idx')]
#[ORM\Index(columns: ['device_type_id'], name: 'device_device_type_idx')]
#[ORM\Index(columns: ['device_profile_id'], name: 'device_device_profile_idx')]
class Device
{
    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column]
    private ?int $id = null;

     #[ORM\Column(length: 255)]
    private ?string $name = null;

     #[ORM\Column(length: 255)]
    private ?string $macAddress = null;

     #[ORM\Column(type: Types::DATE_MUTABLE)]
    private ?\DateTimeInterface $lastStart = null;

     #[ORM\Column(type: Types::DATE_MUTABLE)]
    private ?\DateTimeInterface $dateCreated = null;

     #[ORM\ManyToOne(targetEntity: Brand::class, fetch: 'LAZY', inversedBy: 'devices')]
    #[ORM\JoinColumn(referencedColumnName: 'id', nullable: false)]
    #[NotNull]
    Private Brand $brand;

     #[ORM\ManyToOne(targetEntity: Subscriber::class, fetch: 'LAZY', inversedBy: 'devices')]
    #[ORM\JoinColumn(referencedColumnName: 'id', nullable: false)]
    #[NotNull]
    private Subscriber $subscriber;

     #[ORM\ManyToOne(targetEntity: DeviceType::class, fetch: 'LAZY', inversedBy: 'devices')]
    #[ORM\JoinColumn(referencedColumnName: 'id', nullable: false)]
    #[NotNull]
    private DeviceType $deviceType;

     #[ORM\ManyToOne(targetEntity: DeviceProfile::class, fetch: 'LAZY', inversedBy: 'devices')]
    #[ORM\JoinColumn(referencedColumnName: 'id', nullable: false)]
    #[NotNull]
    private DeviceProfile $deviceProfile;

}
```

Fig. 3. Source code example of experimental database applicaton

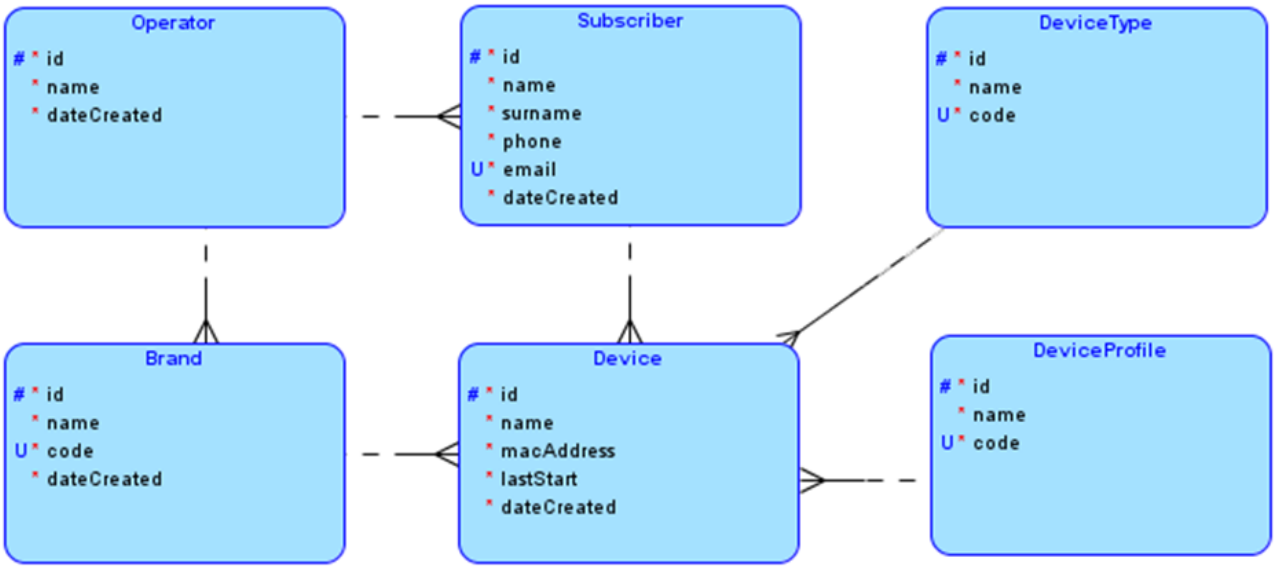

Fig. 1. ERD of the model database

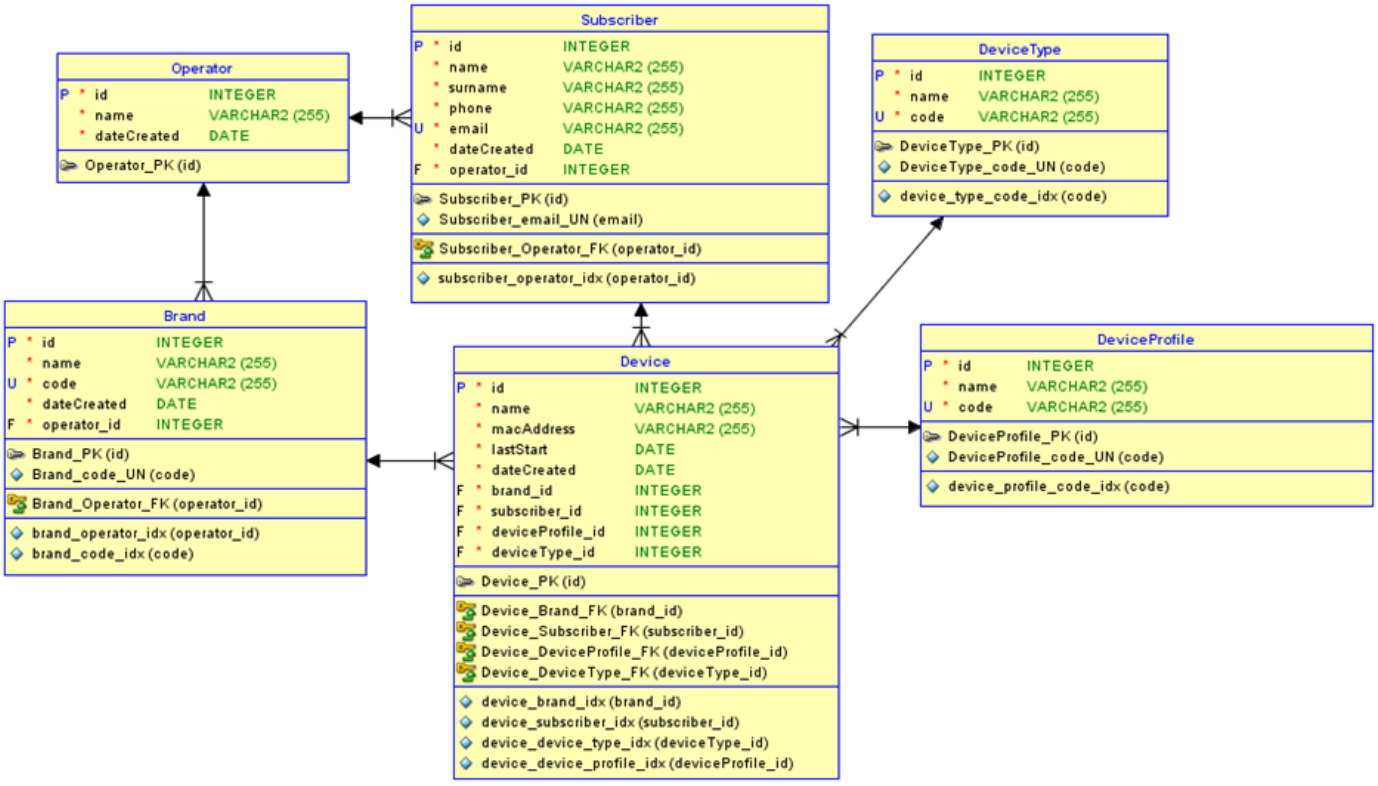

Fig. 2. Relational model of the experimental database

B. Intelligent Data Layer (IDL)

Due to the described problems and the complexity of working with the ORM Framework, a new intelligent data layer was designed. This data layer is intended to solve several problems:

- P1: Reduce the knowledge requirements for developers.
- P2: Simplify access to the data you need.
- P3: Build an interface for developers to build queries predictively.
- P4: Maintain ORM annotations so that the developer can use the standard approach.

The intelligent model layer is based on the basic knowledge of ORM Frameworks and their performance problems. Due to this knowledge, the focus of the design was to minimize the performance-impact as much as possible when using this data layer. Simply put, the working principle of this data layer is to create a tool for combining LAZY Loading and early EAGER Loading. This combination should allow the developer enough space to implement his queries so that the required data is loaded at the appropriate time, while only loading the amount of data needed for the output.

The Intelligent Model Layer contains a superstructure for the classic language used by ORM, in this case, DQL. DQL - Doctrine Query Language is the language used internally by the Doctrine ORM Framework to build queries into the database. This superstructure consists in allowing the user, thanks to a simple QueryBuilder, to define exactly which sub-entities he wants to retrieve to the main entity and which not. Alternatively, define a complete entity tree to be loaded once. This is where the combination of LAZY and EAGER loading occurs. Lazy from the point of view that the data is loaded at the moment of need and Eager from the point of view that all the data that is needed to complete the operation is loaded at once.

IDL then uses a combination of additional query language, DQL and knowledge of ORMs and their QueryBuilders to build the final queries. Each entity must then have a corresponding entity created that can work with all of its properties. Further work to improve IDL could be to automate some features - such as defining an additional language based on existing ORM annotations. At the moment, however, this knowledge needs to be manually inserted into the IDL.

The user can think of this knowledge as "switches" that can be turned on or off when needed. It is then easy to define which sub-entities are to be loaded to the main entity and which are not. A suitable modification can then be, for example, to extend such a system to the whole entity tree, where e.g. for the main entity we define a rule that we want to load this entity, with this sub-entity and these entities to it. In an ORM framework, we would achieve this using LAZY loading, but with a significant performance impact.

C. Architectural inclusion of the intelligent data layer in the application

The intelligent data layer is generally designed as a service layer that knows the individual entities it is working with and then allows the developer to define specific operations. These operations are then used to load specific entities without having to load all unnecessary ones. And consequently also to minimize the number of database calls that would otherwise be made.

The following figure shows an example of the integration of the Intelligent Data Layer into an application architecture that uses ORM.

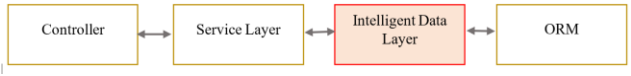

Fig. 4. Application architecture with IDL

IV. EXPERIMENTAL VERIFICATION

In order to verify the implementation of IDLs and their parameters, two specific tasks were selected from the reporting of the company that provided input to the experimental database. The following approaches will be compared in the experiment:

- native query
- DoctrineORM - Lazy loading
- DoctrineORM - Eager loading
- DoctrineORM + IDL
- DoctrineORM + IDL + Cache
- DoctrineORM + IDL + Full change Cache

Briefly described specific reporting tasks:

- Q1 - Get a JSON list of all devices with their name, mac, device profile name, device type name, last device start, brand name, full subscriber name, and the operator name of that subscriber.
- Q2 - Get a JSON list of all operators, their brands and all corresponding devices and their basic information.

At the time of the experiment, the database contained the following numbers of records, for each table:

TABLE I. CONTENTS OF THE EXPERIMENTAL DATABASE

Database table	Number of lines
brand	36
device	31623
device_profile	4
device_type	3
operator	10
subscriber	12694

The following brief description of each experiment is the same for the two specific reporting tasks.

A. Native query

Q1 - Question:

```
SELECT
   d.id AS device_id,
   d.name AS device_name,
   d.mac_address,
   dp.name AS device_profile_name,
   dt.name AS device_type_name,
   d.last_start,
   br.name AS brand_name,
   CONCAT_WS(' ', s.name, s.surname) AS subscriber_name,
   o.name AS operator_name
FROM
   device d
LEFT JOIN
   device_profile dp ON d.device_profile_id = dp.id
LEFT JOIN
   device_type dt ON d.device_type_id = dt.id
LEFT JOIN
   brand br ON br.id = d.brand_id
LEFT JOIN
   subscriber s ON d.subscriber_id = s.id
LEFT JOIN
   operator o ON s.operator_id = o.id;
```

Q2 - Question:

```
SELECT
   o.name AS operator_name,
   b.name AS brand_name,
   b.code AS brand_code,
   CONCAT_WS(' ', s.name, s.surname) AS subscriber_name,
   d.name AS device_name,
   d.mac_address AS device_mac,
   last_start device_last_start,
   dp.name AS device_profile_name,
   dt.name AS device_type_name
FROM
   brand b
LEFT JOIN
   operator o ON b.operator_id = o.id
LEFT JOIN
   subscriber s ON o.id = s.operator_id
LEFT JOIN
   device d ON d.subscriber_id = s.id
LEFT JOIN
   device_type dt ON dt.id = d.device_type_id
LEFT JOIN
   device_profile dp ON dp.id = d.device_profile_id
WHERE
   d.brand_id = b.id;
```

B. DoctrineORM - Lazy loading

For this experiment, all sessions defined at the ORM level were set as LAZY. This includes sessions that do not directly correspond to the queries from the previous section. This is to ensure that no extra data is loaded that could affect the experiment.

C. DoctrineORM - Eager loading

For this experiment, all corresponding sessions were set to EAGER. Sessions that are unnecessary for obtaining matching data were left set to LAZY.

D. DoctrineORM + IDL

For this experiment, all ORM sessions were set to EXTRA_LAZY. The reason is to avoid unintentional retrieval, for example, if the appropriate getter is called in each entity to retrieve a mere ID. Furthermore, a call with the corresponding IDL setting was made.

E. DoctrineORM + IDL + Cache

This experiment uses the same setup as the previous one. As an extension for this experiment, a cache (Memcached) was enabled that contains all normally unmodified entities. For the purpose of the experiment, the following entities were marked for caching: Brand, Operator, DeviceProfile, DeviceType. Entities are cached with an expiration of 1200 minutes.

F. DoctrineORM + IDL + Full change Cache

The last experiment contains the same settings as the previous one with IDL, but in addition all information is cached. This information is duplicated and stored persistently in the database and in the cache at the same time. Thus, the moment any entity is changed, it is stored both in the Cache and in the database. This approach replaces more modern languages in PHP that are not just "request-base" and keep the information retrieved until the information expires. In the experiment, the cache contains complete datasets from all database tables.

V. Results of Experiments

The following data were obtained from the above experiments.

TABLE II. TABLE OF EXPERIMENTAL RESULTS FOR Q1 – PART 1

Experiment	Execution time [ms]	Symfony initialization [ms]	Memory peak [MB]	Doctrine Memory [MB]
Native query	170	30	54	54
Lazy Loading	2999	31	181,5	181,5
Eager Loading	1348	33	125,5	125,5
IDL	1057	31	123,5	123,5
IDL + Cache	1049	29	105	105
IDL + Full change cache	555	31	0	0

TABLE III. TABLE OF EXPERIMENTAL RESULTS FOR Q1 – PART 2

Experiment	DB queries	Different queries	Query time [ms]	DB QT/EXT [%]
Native query	1	1	69,77	41,04
Lazy Loading	12748	6	1208,46	40,29
Eager Loading	2	2	169,02	12,54
IDL	6	6	52,37	4,95
IDL + Cache	2	2	41,37	3,94
IDL + Full change cache	0	0	0	0

TABLE IV. TABLE OF EXPERIMENTAL RESULTS FOR Q2 – PART 1

Experiment	Execution time [ms]	Symfony initialization [ms]	Memory peak [MB]	Doctrine Memory [MB]
Native query	195	30	64	44
Lazy Loading	2819	31	213,08	183,5
Eager Loading	1620	36	163,08	125,5
IDL	1037	33	161,08	129
IDL + Cache	795	31	182,58	86
IDL + Full change cache	545	30	269,58	0

TABLE V. TABLE OF EXPERIMENTAL RESULTS FOR Q2 – PART 2

Experiment	DB queries	Different queries	Query time [ms]	DB QT/EXT [%]
Native query	1	1	83,51	42,83
Lazy Loading	12748	6	1106,26	39,24
Eager Loading	4	4	89,28	5,51
IDL	6	6	49,6	4,78
IDL + Cache	1	1	20,25	2,55
IDL + Full change cache	0	0	0	0

For Lazy Loading, Eager Loading, and IDL, the best values for each of the monitored parameters, outside of Symfony initialization, were marked in the results tables. This information is included in the table to give a better idea of what the execution time contains.

The value of the database operation duration ratio (DB QT/EXT) is expressed from the ratio of Query time (QT) to total execution time (EXT) and then converted to a percentage value.

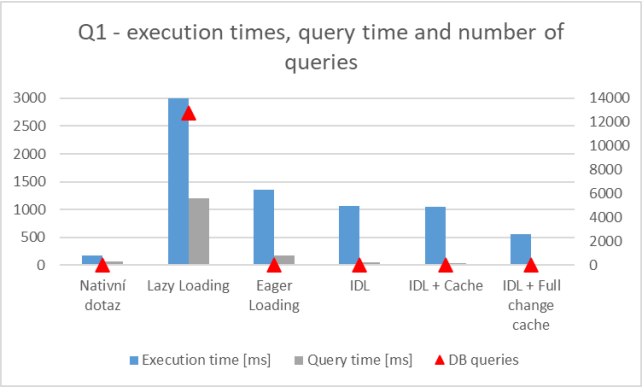

Fig. 5. Q1 - Graph of execution time, query time and number of queries

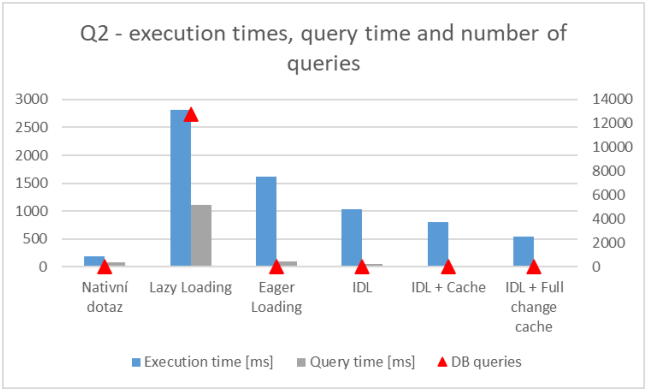

Fig. 6. Q2 - Graph of execution time, query time and number of queries

VII. CONCLUSION

From the above results, it can be seen that in both experimental treatments, some of the observed parameters were reduced. To give an example, the execution time was reduced in both cases. Furthermore, a significant reduction in query time, which was even reduced to below the duration of the native query in both cases. And it is certainly worth pointing out the DB QT/EXT parameter, which shows that a large part of the performance impact has been moved from the ORM, directly to the application layers, including the IDL.

In terms of the defined criteria and the problems to be solved by the IDL, the IDL design was successful. The IDL introduced an interface for intelligent loading of entities and their sub-entities, which still causes problems for conventional ORM frameworks. By minimizing query time even against native SQL queries, the IDL design can be considered a very successful form of data access optimization using an ORM framework.

In the process of working on the experiments and evaluating them, several other suggestions for further work emerged, e.g., automated creation of factual information about entities, extensions by projection, paging of entities or subentities and their sequential loading. Another possible direction of exploration could be the data structures

REFERENCES

[1] Arzamasova, Natalia, Martin Schäler, a Klemens Böhm. 2017. „Cleaning Antipatterns in an SQL Query Log.“ *IEEE Transactions on Knowledge and Data Engineering.* IEEE. 421-434. doi:10.1109/TKDE.2017.2772252.

[2] Boniewicz, Aleksandra, Piotr Wiśniewski, a Krzysztof Stencel. 2013. „On redundant data for faster recursive querying via ORM systems.“ *2013 Federated Conference on Computer Science and Information Systems.* Krakow. 1451-1458.

[3] Čerešňák, Roman, a Michal Kvet. 2019. „Comparison of query performance in relational a non-relation databases.“ *Transportation Research Procedia.* Padova. 170-177. doi:10.1016/j.trpro.2019.07.027.

[4] Fertalj, Kresimir, Nikica Hlupic, a Lidia Rovan. 2006. „Why (not) ORM?“ *28th International Conference on Information Technology Interfaces, 2006.* Cavtat: IEEE. 683-688. doi:10.1109/ITI.2006.1708563.

[5] Chen, Te-Hsun, Shang Weiyi, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, a Flora Parminder. 2014. „Detecting performance anti-patterns for applications developed using object-relational mapping.“ *Proceedings of the 36th International Conference on*

Software Engineering. Hyderabad: Association for Computing Machinery. 1001-1012. doi:10.1145/2568225.2568259.

[6] Chen, Tse-Hsun, Weiyi Shang, Zhen Ming Jiang, Hassan E. Ahmed, Mohamed Nasser, a Flora Parmider. 2016. „Finding and Evaluating the Performance Impact of Redundant Data Access for Applications that are Developed Using Object-Relational Mapping Frameworks." *IEEE Transactions on Software Engineering*. IEEE. 1148-1161. doi:10.1109/TSE.2016.2553039.

[7] Choina, Marcin, a Maria Skublewska-Paszowska. 2022. „Performance analysis of relational databases MySQL, PostgreSQL and Oracle using Doctrine libraries." *Journal of Computer Sciences Institute*. Wydawnictwo Politechniki Lubelskiej. 250-257. doi:10.35784/jcsi.3000.

[8] Ireland, Christopher, David Bowers, Michael Newton, a Kevin Waugh. 2009. „A Classification of Object-Relational Impedance Mismatch." *2009 First International Confernce on Advances in Databases*, Knowledge, and Data Applications. Gosier: IEEE. 36-43. doi:10.1109/DBKDA.2009.11.

[9] Khelifi, Nassima Yamouni, Michał Śmiałek, a Rachida Mekki. 2015. „Generating Database Access Code From Domain Models." *2015 Federated Conference on Computer Science and Information Systems*. Lodz: IEEE. 991-996. doi:10.13140/RG.2.1.1476.4564.

[10] Ma, Kun, a Yang Bo. *2015. „Access-Aware In-memory Data Cache Middleware for Relational Databases." 2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*. New York: IEEE. 1506-1511. doi:10.1109/HPCC-CSS-ICESS.2015.186.

[11] Martin, Lorenz, Hesse Guenter, a Rudolph Jan-Peer. 2016. „Object-relational Mapping Revised - A Guideline Review and Consolidation." *11th International Conference on Software Engineering and Applications*. Lisabon: SciTePress. 157-168. doi:10.5220/0005974201570168.

[12] Nazário, Marcos Felipe Carvalho, Eduardo Guerra, Rodrigo Bonifácio, a Gustavo Pinto. 2019. „Detecting and Reporting Object-Relational Mapping Problems: An Industrial Report." 2019 *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. Proto de Galinhas: IEEE. 1-6. doi:10.1109/ESEM.2019.8870163.

[13] Scully, Ziv, a Adam Chlipala. 2017. „A program optimization for automatic database result caching." *ACM SIGPLAN Notices*. Association for Computing Machinery: New York. 271-284. doi:10.1145/3093333.3009891.

[14] Shao, Shudi, Qui Zhengyi, Yu Xiao, a Yang Wei. 2020. „Database-Access Performance Antipatterns in Database-Backed Web Applications." *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Adelaide: IEEE. 58-69.

[15] Sharma, Tushar, Marios Fragkoulis, Stamatia Rizou, Magiel Bruntink, a Diomidis Spinellis. 2018. „Smelly Relations: Measuring and Understanding Database Schema Quality." *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. Gothenburg: IEEE. 55-64.

[16] Vaja, Dhaval Dhirajlal, a Rahevar Mrugendrasinh. 2016. „Improve performance of ORM caching using In-Memory caching." *2016 International Conference on Computing, Analytics and Security Trends (CAST)*. Pune: IEEE. 112-115. doi:10.1109/CAST.2016.7914950.

[17] Węgrzynowicz, Patrycja. 2013. „Performance antipatterns of one to many association in hibernate." *2013 Federated Conference on Computer Science and Information Systems*. Krakow: IEEE. 1475-1481.

[18] Yan, Cong, Alvin Cheung, Junwen Yang, a Shan Lu. 2017. „Understanding Database Performance Inefficiencies in Real-world Web Applications." *ACM Conference on Information and Knowledge Management*. New York: Association for Computing Machinery. 1299-1308. doi:10.1145/3132847.3132954.