Verification of IDL Architecture Performance for Larger Datasets

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

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

***Abstract*—Due to the ever increasing popularity of ORM frameworks, the IDL architecture was compiled by the authors as part of an earlier publication, Design of Data Access Architecture Using ORM Framework. This architecture allows to implement ORM frameworks with prediction of the required data. Given the size of datasets that can be managed in database systems, it was necessary to verify that this architecture is robust enough to handle similar problems. In the context of this paper, we focus on examining the performance of our defined architecture with larger datasets, which should help us determine whether such an architecture is suitable for common use. Along with the performance verification for larger datasets, performance tests were performed in case the database does not contain indexes for foreign keys.**

I. Introduction

Nowadays, with an increasing number of computer applications using ORM frameworks, developers have been faced with a performance problem. As ORM frameworks attempt to be an abstraction for a variety of database needs, many software applications are reaching a state where their performance is limited by the particular ORM framework being used. [1], [10]

Within the IDL architecture, we have tried to create an environment that allows the use of the ORM framework with all its advantages, but at the same time provides the developer with enough space to optimize the application. The IDL architecture has already been built and described within the previously published article [7].

A fundamental issue with most ORM frameworks is their general annotation of the entities and sessions they use. These annotations, by rule, do not allow for any configuration in the sense that this data is only retrieved if the user wants it or not. While it is possible to make use of the built-in Lazy Loading and Eager Loading, even this may not always be sufficient... In this paper we have thus focused on comparing the use of IDL and these built-in features for a larger dataset. [2], [3], [5]

This dataset was selected to meet the requirements of multiple entity relationships, at least one subentity existed, and complex retrieval from the database was required. Also included for comparison are results from the variant where only a native SQL query was used. For comparison, the results were measured on the same dataset using MySQL 8.0.26 and PostgreSQL 16.1 databases. The ORM used as a test ORM was Doctrine 2 with the PHP framework Symfony 6.3 combined with PHP 8.1 as the FPM without any caching or optimization extensions. Symfony Profiler, which is part of the Symfony framework, was then used to record the results. The profiler was then used for parts such as Performance and Doctrine.

The experiments were performed on the same hardware on which the IDL Architecture was built. It is a station with an Intel i7-7820X processor, 64 GB DDR4 2666MHz and equipped with a Samsung 970 EVO 500GB NVME disk. The operating system used was Ubuntu 23.10 in which the test experimental environment was virtualized using Docker.

The configuration of each software used was in default settings with two exceptions. The first is the maximum memory that can be used by a PHP process. Here the maximum was set to 16GB. The second exception was set for Nginx, where it was necessary to raise the maximum timeout for a response from PHP-fpm. Here a value of 15 minutes was used.

II. Dataset

The experiments were performed using the experimental relational database model from the article [7].

The relational database for the experiments contained the following data. This data was generated through the ORM Doctrine into MySQL and then converted into PostgreSQL using the Nmig tool. For information, we also show here the sizes that the MySQL and PostgreSQL databases displayed.

TABLE I. Data and index sizes in relational databases

Table	**Nr. of lines**	**Size**			
		MySQL		**PosgreSQL**	
			IDX		**IDX**
brand	36	0,016	0,032	0,064	0,048
device_type	3	31	0,016	0,048	0,032
device_profile	4	33	0,016	0,048	0,032
operator	10	31	0,016	0,024	0,016
subscriber	311847	29	5,783	38	9
device	1480661	3138,04	96,66	220	78

III. Experiments

A. Background of experiments

To verify the behavior of IDL within a larger dataset, two previously presented experiments were used. These experiments should sufficiently test the robustness and performance of IDL for larger datasets. Subsequently, all indexes were removed from the database. Since only indexes could not be removed for MySQL 8.0.26, the constraints on FK were also removed. These constraints were subsequently removed in PostgreSQL as well to make the results more comparable. [4] , [5]

Both experiments were then performed in the following steps:

1) ORM performance with native SQL query
2) Using ORM with Lazy loading
3) Using ORM with Eager loading
4) Using ORM with IDL

Since both of our experiments for IDL were built as worst-case, i.e., so that all data is loaded from the database, they can be directly compared with the ORM variant with Eager loading. For the IDL experiments, all ORM sessions were then set to EXTRA_LAZY so that they would not affect the functioning of IDL, but at the same time, they would not be removed from the code completely.

B. Brief description of experiments

Experiments were set up based on real Use-Case commercial companies. These are the real-use requirements that the system can handle in reporting. Although the complexity of the queries is not great at first glance, the performance of the system using ORM is significantly worse than using a simple native SQL query.

The previously tested queries Q1 and Q2 from the previous article were used for testing to maintain continuity to validate the results. [7]

Q1 - Obtain a list of devices with device, operator and subscriber information

The output combines information from the device, device_profile, device_type, brand, subscriber and operator tables. This information is then converted to JSON and returned to the user, the previously tested queries from the previous article were used for testing to maintain continuity to validate the results. [7]

Native SQL query:

```
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 - Obtaining a list of equipment for brandy operators

The output combines information from the brand, operator, subscriber, device, device_type and device_profile tables. As in the previous case, the output is formatted into JSON and then returned to the user.

Native SQL query:

```
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;
```

C. Results of experiments

The following tables record the results of the testing, where the labels of the columns are:

- ET – Execution Time [ms]
- SI – Symfony Initialization [ms]
- MP – Memory Peak [MB]
- DM - Doctrine Memory [MB]
- Q – Number of DB Queries
- DQ – Number of Different Queries
- QT – Query Time [ms]

TABLE II. RESULTS FOR Q1 EXPERIMENT – MYSQL WITH INDEXES

ET	SI	MP	DM	Q	DQ	QT	QT/ET
Native Query							
13470	32	1952,01	218,6	1	1	5205,83	38,65
Lazy Loading							
106956	32	7143,88	6378	311901	6	28076,67	26,25
Eager Loading							
161228	36	**6336,01**	4492	**2**	**2**	73061,52	45,32
IDL							
48775	31	6386,04	**4240**	6	6	**3389,21**	6,95

TABLE III. RESULTS FOR Q1 EXPERIMENT – MYSQL WITHOUT INDEXES

ET	SI	MP	DM	Q	DQ	QT	QT/ET
Native Query							
8844	42	2156,01	438,6	1	1	3296,22	37,27
Lazy Loading							
124663	32	7143,88	6378	311901	6	28372,86	22,76
Eager Loading							
71860	36	6414,01	4464	**2**	**2**	3892,99	5,42
IDL							
68217	36	**6386,04**	**4242**	6	6	**3492,98**	5,12

TABLE IV. RESULTS FOR Q1 EXPERIMENT – POSTGRESQL WITH INDEXES

ET	SI	MP	DM	Q	DQ	QT	QT/ET
Native Query							
5660	38	2262,01	656	1	1	2837,32	50,13
Lazy Loading							
154063	34	7173,87	6384	311901	6	49348,95	32,03
Eager Loading							
64395	33	**6356,02**	**4096**	**2**	**2**	4798,48	7,45
IDL (CHUNK 1000)							
67314	38	6368,02	4204	317	7	**3175,5**	4,72

TABLE V. RESULTS FOR Q1 EXPERIMENT – POSTGRESQL WITHOUT INDEXES

ET	SI	MP	DM	Q	DQ	QT	QT/ET
Native Query							
6485	36	2261,01	12	1	1	3188,32	49,16
Lazy Loading							
133142	39	7163,87	6384	311901	6	40985,21	30,78
Eager Loading							
65403	35	**6112,01**	**4106**	**2**	**2**	5524,86	8,45
IDL							
64960	35	6366,02	4204	317	7	**2771,61**	4,27

TABLE VI. RESULTS FOR Q2 EXPERIMENT – MYSQL WITH INDEXES

ET	SI	MP	DM	Q	DQ	QT	QT/ET
Native Query							
23277	39	2442,01	1104,6	1	1	20428,85	87,76
Lazy Loading							
108028	34	7229,11	6376,1	311901	**6**	38184,36	35,35
Eager Loading							
Ungettable results							
IDL (Chunk 1000)							
69191	33	**6478,53**	4236,5	317	7	**3019,1**	4,36
IDL (OR 1000)							
74388	39	6524,54	**3947,9**	**6**	**6**	3559,33	4,78

TABLE VII. RESULTS FOR Q2 EXPERIMENT – MYSQL WITHOUT INDEXES

ET	SI	MP	DM	Q	DQ	QT	QT/ET
Native Query							
7051	38	2570,01	680,6	1	1	4157,32	58,96
Lazy Loading							
135674	36	7251,11	6833,1	311901	**6**	43032,98	31,72
Eager Loading							
Ungettable results							
IDL (Chunk 1000)							
57358	32	6478,53	4236,5	317	7	**2513,24**	4,38
IDL (OR 1000)							
78102	38	6526,54	**3947,9**	**6**	**6**	3855,89	4,94

TABLE VIII. RESULTS FOR Q2 EXPERIMENT – POSTGRESQL WITH INDEXES

ET	SI	MP	DM	Q	DQ	QT	QT/ET
Native Query							
5108	35	2388,01	282	1	1	2017,67	39,50
Lazy Loading							
132233	34	7241,11	6372,1	311901	**6**	44040,22	33,31
Eager Loading							
Ungettable results							
IDL (Chunk 1000)							
56294	35	**6508,52**	**4258,5**	**317**	7	**3092,27**	5,49

TABLE IX. RESULTS FOR Q2 EXPERIMENT – POSTGRESQL WITH INDEXES

ET	SI	MP	DM	Q	DQ	QT	QT/ET
Native Query							
4801	35	2388,01	200	1	1	1689,48	35,19
Lazy Loading							
119262	41	7236,08	6372,1	311901	**6**	39591,66	33,20
Eager Loading							
Ungettable results							
IDL (Chunk 1000)							
63128	34	**6506,52**	**4258,5**	**317**	7	**2811,95**	4,45

In the tables, the best values for each measured parameter, outside of Symfony initialization, have been highlighted.

Description of monitored parameters in tables:

- Execution time [ms] - the total time it took to execute a complete query on the server side.
- Symfony initialization [ms] - represents the time it took to initialize the Symfony framework. This time includes re-parsing the source files, building the Symfony Cache and then connecting to the databases. It is mainly of informative value.

- Memory peak [MB] - the maximum measured memory size on the PHP-fpm side.
- Doctrine Memory [MB] - the maximum measured size of memory occupied only by the Doctrine framework and its data.
- DB Queries - the total number of queries that have been executed on the database through Doctrine ORM.
- Different Queries - the number of unique queries that have been executed on the database through Doctrine ORM. For example, identical queries with different parameters in the predicates are considered to be unique queries.
- Query time [ms] - measured time that was needed to execute all database queries.
- DB QT/EXT [%] - the percentage of time that was required from the total http request time for processing queries by the database system. The rest of the time was used by the application layer request.

IV. Discussion of results

From the above results, it can be seen that there has been a relatively substantial reduction in the time required to process the presented experiments. This is the case for both the total processing time and the query time itself. For experiment Q2, unfortunately, we were unable to obtain results in a meaningful amount of time, and after 15 minutes of data processing on the PHP side, it was decided that these results would no longer be of any telling value. Experiment Q2 is thus unfeasible from our point of view using ORM with Eager loading.

If we look at the results through the lens of index usage, there was an interesting paradox where the processing time was reduced for both the native query for Q1 and the native query for Q2. For Q2, this happened even for both relational databases. Similar behavior can be observed for Q1 when using eager loading for Q2 and even for lazy loading.

Looking at the graphs we can also see that the required query time for Q1 processing over MySQL with indexes was 44% lower than in the case of PostgreSQL. However, the expected results were the opposite, i.e. in favor of PostgreSQL.

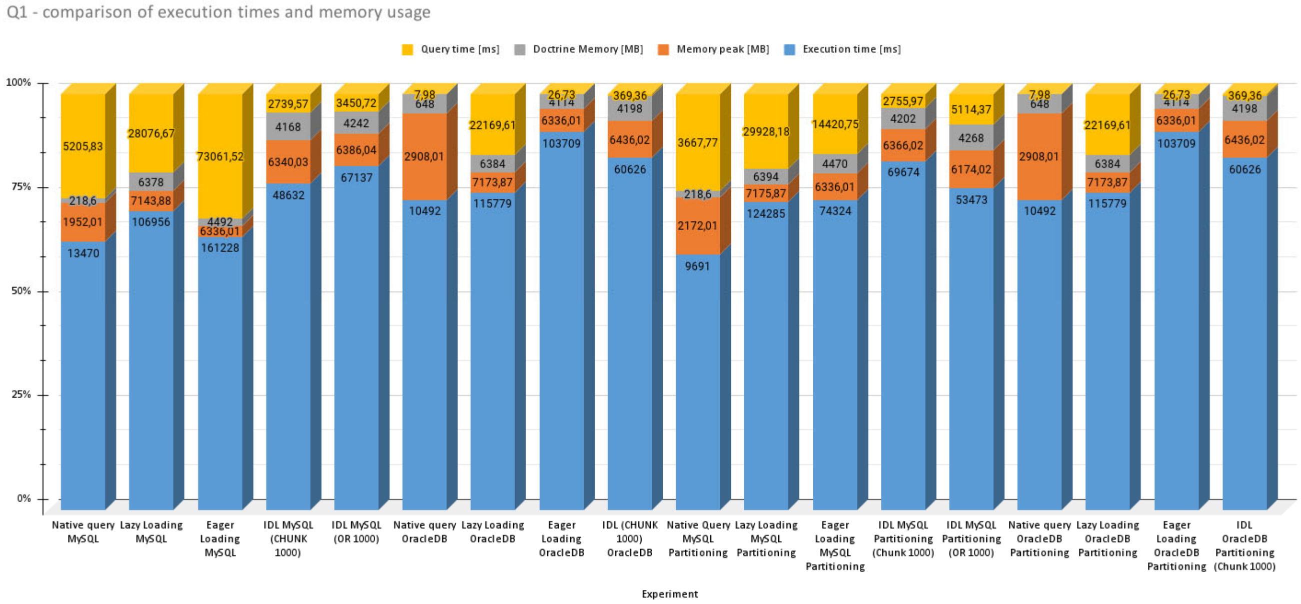

Fig. 1. Visualization of results for experiment Q1

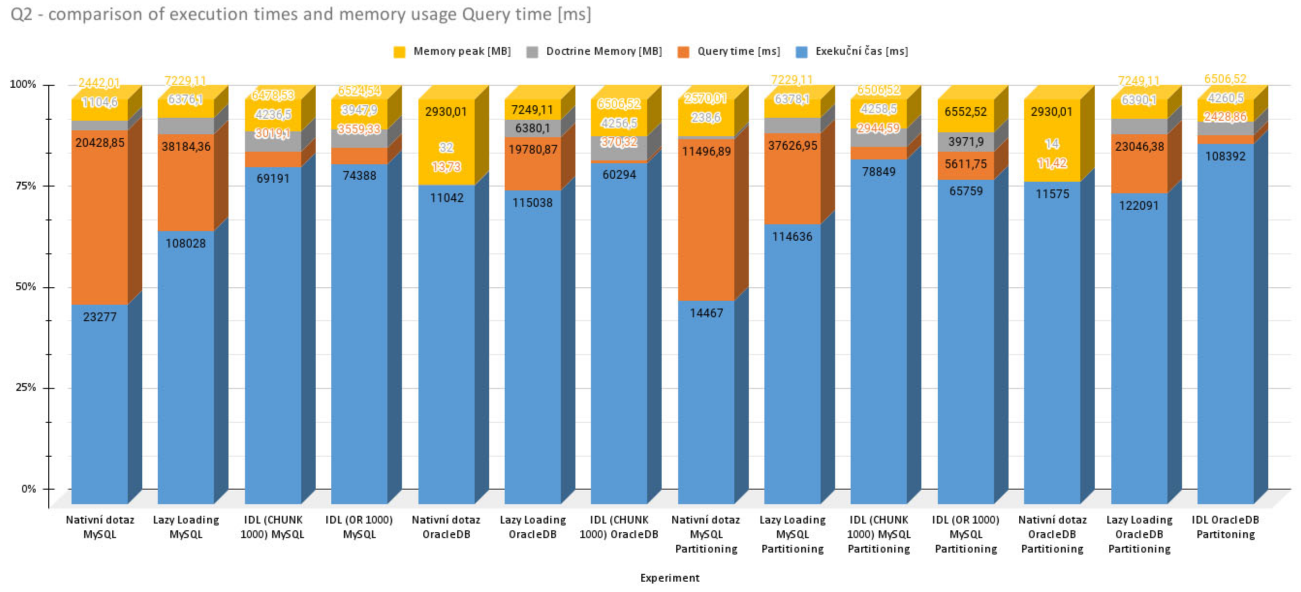

Fig. 2. Visualization of results for experiment Q2

As a general summary of the results, we can say that IDL was quite successful in testing a larger dataset in both variants of the approach and using both databases. In all cases, the use of the IDL architecture achieved the best query time, outperforming even the native query itself in some cases (e.g., Q1 MySQL + indexes).

In the case of comparing access through IDL and eager loading, we only have data for Q1. For Q2, unfortunately, we were not able to get any meaningful results even after 15 minutes of query running. These results were automatically marked as irrelevant and the experiment was deemed unavailable using Doctrine + Eager loading. However, in the case of Q1, it can be said that selecting a more appropriate solution is more difficult. Apart from using MySQL with indexes, the results are almost comparable. However, IDL still allows to define specific loaded data, which is not easily possible when using Eager loading. [5], [6]

V. Troubleshooting

In the course of the experiments, it was necessary to address various differences in the use of MySQL and PostgreSQL that were not addressed in the design of the IDL architecture. The first problematic part was data migration, where MySQL cannot easily export data for direct use in PostgreSQL. Thus, it was necessary to select a suitable tool that allowed us to convert data from MySQL database to PostgreSQL. [8], [9]

Furthermore, during the experimentation, a problem with nginx 504 Gateway timeout appeared, which was caused by long data processing on the PHP side. It was necessary to modify the server configuration so that nginx would not terminate the connection after 60 seconds. When the connection was retried, any results were not passed and php-fpm also received repeated requests and ran the experiments again every minute. The timeout value was set to 15 minutes. However, even with this increased timeout, we were unable to obtain the results of experiment Q2 for ORM with Eager loading.

The last experiment processing problem occurred when running experiments against a PostgreSQL database. Here we encountered a problem with the PostgreSQL driver, which did not allow to insert more than 65535 values into a parameter. Thus, it was necessary to modify the experiments and query the database repeatedly for a smaller number of values. Due to a previously known limitation from OracleDB, we chose a value of 1000 values in order to be able to use this modification directly with OracleDB in future tests.

VII. Conclusion

During the experiments, it was shown that the IDL architecture is sufficiently powerful even when used with larger databases and keeping the advantages of using ORM. Even without the use of indexes, it achieves interesting results, so it can partially eliminate the ignorance or errors of the developer who would like to implement it.

However, the results show that, like the conventional ORM approach, it requires many times more memory than the native query. This is mainly due to the fact that the native query does not use any additional complex mapping to application objects and thus accesses the actual data directly. The results also show that the vast majority of the time (>94%) in the case of IDL is used to process data in the application layer outside the database. Thus, even here there is still a lot of room for optimization of IDL on the application layer side.

Acknowledgment

It was supported by the Erasmus+ project: Project number: 2022-1-SK01-KA220-HED-000089149, Project title: Including EVERyone in GREEN Data Analysis (EVERGREEN) funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the Slovak Academic Association for International Cooperation (SAAIC). Neither the European Union nor SAAIC can be held responsible for them.

Co-funded by the European Union

EverGreen DATA Analytics

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] Z. Dong et al. 2023. "Database Deadlock Diagnosis for Large-Scale ORM-Based Web Applications," 2023 *IEEE 39th International Conference on Data Engineering (ICDE).* Anaheim, CA, USA. 2864-2877. doi: 10.1109/ICDE55515.2023.00219.

[3] C. Pitt, Pro PHP 8 MVC: „Model View Controller Architecture-Driven Application Development“, 2nd edition, Apress, 2021. ISBN: 978-1484269565.

[4] Loli, Samuel & Teixeira, Leopoldo & Cartaxo, Bruno. (2020). „A Catalog of Object-Relational Mapping Code Smells for Java“. 82-91. 10.1145/3422392.3422432.

[5] M. Kvet, "Database Index Balancing Strategy," 2021 *29th Conference of Open Innovations Association (FRUCT)*, Tampere, Finland, 2021, 214-221. doi: 10.23919/FRUCT52173.2021.9435452

[6] M. Kvet and J. Papan, "The Complexity of the Data Retrieval Process Using the Proposed Index Extension," in *IEEE Access*, vol. 10, 46187-46213, 2022. doi: 10.1109/ACCESS.2022.3170711

[7] F. Majerik and M. Borkovcova, "Design of Data Access Architecture Using ORM Framework," 2023 *34th Conference of Open Innovations Association (FRUCT)*, Riga, Latvia, 2023, 93-99. doi: 10.23919/FRUCT60429.2023.10328151.

[8] G. Vial, "Lessons in Persisting Object Data Using Object-Relational Mapping," in *IEEE Software*, vol. 36, no. 6, 43-52. Nov.-Dec. 2019, doi: 10.1109/MS.2018.227105428.

[9] Z. Xu, J. Zhu, a L. Yang, "Mining the Relationship between Object-Relational Mapping Performance Anti-patterns and Code Clones," *Proceedings of the 35th International Conference on Software Engineering*, San Francisco, 2023, 136-141. doi: 10.18293/SEKE2023-161.

[10] 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.