Performance Verification of IDL Architecture for Partitioned Database

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*—With the growing popularity of ORM Frameworks and the need to efficiently manage large datasets, the deployment of various optimizations of this approach is increasingly being explored. In this paper, we focus on investigating the impact of partitioning on IDL performance, which was introduced in a previous study in the publication Design of Data Access Architecture Using ORM Framework. This IDL enables the implementation of ORM Frameworks with loaded data prediction and overall improves and speeds up database data access through ORM Frameworks. This paper extends this work and focuses on further optimizing the IDL in the form of partitioning database tables. At the same time, we have tried to analyze the effect of partitioning on the behavior and performance of IDL. Finally, it will be evaluated whether this optimization technique is applicable in a practical environment and whether an improvement in IDL performance has been achieved.**

I. Introduction

As part of our long-term research, we are trying to design a solution that would allow developers to implement ORM Frameworks efficiently. These Frameworks, as a rule, work very inefficiently with database systems and thus cause significant performance problems in computer applications. So we focused on designing our own data access layer, which we named IDL in our earlier work Design of Data Access Architecture using ORM Framework. [8], [10]

We are now exploring how to optimize access through this data layer as much as possible and take the most effective advantage of the benefits provided by ORM Frameworks. For example, maintaining the use of their own query languages, automated mapping of database data to application entities, automatic migration of database schemas, and more. [1], [2], [4]

In the previous experiments we built a dataset that roughly corresponds to a medium-sized database. This dataset was built based on the requirements of a commercial company that provided us with a database schema and example calls that are challenging to process from their perspective. We used this same dataset to measure the performance of IDL with database partitioning in the context of this paper. [3], [5], [9]

The results were measured using MySQL 8.0.26 and OracleDB 21c Express Edition 21.3.0 database for comparison. ORM Doctrine 2 with PHP framework Symfony 6.3 combined with PHP 8.1 as FPM without any caching and optimization extensions was used as the test ORM. Symfony Profiler, which is part of the Symfony framework, was then used to record the results. The Performance and Doctrine parts of the profiler were then used. [11], [12]

The experiments were performed on the same hardware on which the IDL Architecture was built and the earlier index experiments were performed. This is a workstation with an Intel i7-7820X processor, 64GB of 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. OracleDB XE was installed from a prebuilt docker image *container-registry.oracle.com/database/express:21.3.0-xe*.

II. Dataset and description of database schemas

The relational database model from the article Design of Data Access Architecture using ORM Framework was used for the experiments. [8]

The relational database for the experiments contained the following data that had been used previously for the index experiments. For the measurements in this paper, the data was converted to OracleDB using mysqldump, then reformatted using the Linux stream editor (sed) tool to match the inserts required by OracleDB. For example, it was necessary to change the formatting of the dates, where the MySQL export only contained the notation "2024-07-01", but the OracleDB insert requires the date to be converted to a date. For example, using the TO_DATE("2024-07-01", "YYYY-MM-DD") command. It was also necessary to remove all apostrophes that MySQL uses to label tables and columns. [6]

After migrating the data from MySQL to OracleDB, we converted the device and subscriber tables to new tables that were partitioned. The device table was partitioned into 300 partitions and the subscriber table was partitioned into 60 partitions. For the other tables, due to their size, partitioning was not performed. Since MySQL does not support all options and types of partitioning, Hash partitioning was chosen because both databases used support it. Furthermore, it was not possible to create partitioning for foreign keys because the MySQL

database does not support foreign key partitioning when using the InnoDB database engine. Thus, only partitioning by primary key was used.

Example of the partitioned *device* table creation script. For the purposes of this article, the foreign key constraint definitions have been removed from the DDL statement.

```
CREATE TABLE device_partitioned (
id NUMBER generated BY DEFAULT AS identity PRIMARY
KEY,
brand_id NUMBER NOT NULL,
subscriber_id NUMBER NOT NULL,
device_type_id NUMBER NOT NULL,
device_profile_id NUMBER NOT NULL,
name VARCHAR2(255) NOT NULL,
mac_address VARCHAR2(255) NOT NULL,
last_start DATE NOT NULL,
date_created DATE NOT NULL
) PARTITION BY hash (id) partitions 300;
```

TABLE I. DATA SIZES AND FILE/EXTENT COUNTS

Table	Nr. of lines	Description of Databases			
		MySQL		OracleDB	
		size [MB]	Nr. of data files	size [MB]	Nr. of extents
brand	3	0,016	1	0,06	1
device_type	3	0,016	1	0,06	1
device_profile	4	0,016	1	0,06	1
operator	10	0,016	1	0,13	4
subscriber	311.847	28,89	1	26	41
Subscriber partitioned	311.847	30,47	60	480	60
device	1.480.661	138,04	1	136	88
device partitioned	1.480.661	147,45	300	2400	300

When retrieving this information, it was interesting to note that InnoDB MySQL reported an inaccurate number of rows within the database table statistics. For example, 310,632 rows were reported for the subscriber table and 316,625 rows were reported for the device_partitioned table. However, both tables contained identical data. [7]

However, partitioning has also significantly increased index sizes and increased the time for inserting data into partitioned tables. In the following table, we show the observed size through the system schema *information_schema* in MySQL and for OracleDB from the system table *dba_segments*. According to the values found, it can be assumed that OracleDB has reserved space in advance to completely populate all the partitions created. The reported data corresponds to a size of *8MB/extent*, where one extent is also one partition. For both databases, the index size is then calculated as the sum of all index sizes in a given table. [5], [9]

TABLE II. COMPARISON OF DATA AND INDEX SIZES

Table	Databases Size [MB] WITHOUT AND WITH INDEX (IDX)			
	MySQL		OracleDB	
		IDX		IDX
device	138,04	96,67	136	680
device_partitioned	147,45	115,97	2400	4800
subscriber	28,89	5,78	26	52
subscriber_partitioned	30,47	8,85	480	960

From the table above, it can be seen that the default table sizes between the databases were not significantly different. However, there are already significant differences in the size of the indexes. Compared to MySQL, OracleDB has indexes that are many times larger, so the data files take up significantly more space on the physical storage.

Another difference is that by default, OracleDB databases did not create new files for individual partitions, but created them as additional segments in a single file. In contrast, the MySQL database created a new file for each partition of the table. Unlike OracleDB, MySQL did not allocate all free space in advance, but data files were expanded only as data was incrementally inserted.

III. EXPERIMENTS

A. Background of experiments

To verify the behavior of IDL with partitioning, two previously presented experiments were used. These experiments were also used to compare the performance of IDL with and without indexes. Copies of the original device and subscriber tables with partitioning were then created as part of this work. Subsequently, the following experiments were performed:

1.) ORM performance with native SQL query

2.) Using ORM with Lazy loading

3.) Using ORM with Eager loading

4.) Using ORM with IDL (CHUNK 1000)

5.) Using ORM with IDL (OR 1000)

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 variation 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.The original annotated entities were retained in the experiments, only their source table was replaced using the following annotation:

For a table without partitioning:
#[@ORM\Table(name="subscriber")

For a table with partitioning:
#[@ORM\Table(name="subscriber_partitioned")

B. Brief description of experiments

The experiments were taken from previously published papers so that it can be better evaluated in the future whether IDL is really effective enough. These experiments were constructed based on real Use-Case commercial companies. Thus, these are real-world usage requests that the system can handle in the context of 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. [8]

The previously tested queries Q1 and Q2 from the previous article were used for testing to maintain continuity to validate the results. [8]

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. [8]

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. [8]

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 data were obtained from the above experiments. The results for MySQL without partitioning were taken from the previously experiments, and supplemented with the results of measurements with partitioned data retrieval by 1000 (Chunk 1000) and measurements with a single query and an OR clause with 1000 values in each OR part of the predicate (OR 1000). [8]

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]
- QT/ET – [%]

TABLE III. RESULTS FOR Q1 EXPERIMENT – MYSQL WITHOUT PARTITIONING

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
IDL (CHUNK 1000)							
48632	48	6340,03	**4168**	317	7	**2739,57**	5,63
IDL (OR 1000)							
67137	43	6386,04	4242	6	6	3450,72	5,14

TABLE IV. RESULTS FOR Q1 EXPERIMENT – MYSQL WITH PARTITIONING

ET	SI	MP	DM	Q	DQ	QT	QT/ET
Native Query							
9691	33	2172,01	218,6	1	1	3667,77	37,85
Lazy Loading							
124285	32	7175,87	6394	311901	6	29928,18	24,08
Eager Loading							
74324	43	6336,01	4470	**2**	**2**	14420,75	19,40
IDL (CHUNK 1000)							
69674	37	6366,02	**4202**	317	7	**2755,97**	3,96
IDL (OR 1000)							
53473	33	**6174,02**	4268	6	6	5114,37	9,56

TABLE V. RESULTS FOR Q1 EXPERIMENT – ORACLEDB WITHOUT PARTITIONING

ET	SI	MP	DM	Q	DQ	QT	QT/ET
Native Query							
10492	33	2908,01	648	1	1	7,98	0,08
Lazy Loading							
115779	31	7173,87	6384	311901	6	22169,61	19,15
Eager Loading							
103709	44	**6336,01**	**4114**	**2**	**2**	**26,73**	0,03
IDL (CHUNK 1000)							
60626	32	6436,02	4198	317	7	369,36	0,61

TABLE VI. RESULTS FOR Q1 EXPERIMENT – ORACLEDB WITH PARTITIONING

ET	SI	MP	DM	Q	DQ	QT	QT/ET
Native Query							
11203	34	2908,01	972	1	1	16,9	0,15
Lazy Loading							
126514	32	7268,01	6404	311901	6	21637,01	17,10
Eager Loading							
90458	34	**6336,01**	**4108**	**2**	**2**	**28,73**	0,03
IDL (CHUNK 1000)							
87889	33	6420,02	4200	317	7	2132,87	2,43

TABLE VII. RESULTS FOR Q2 EXPERIMENT – MYSQL WITHOUT PARTITIONING

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							
Unearned 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 VIII. RESULTS FOR Q2 EXPERIMENT – MYSQL WITH PARTITIONING

ET	SI	MP	DM	Q	DQ	QT	QT/ET
Native Query							
14467	29	2570,01	238,6	1	1	11496,89	79,47
Lazy Loading							
114636	30	7229,11	6378,1	311901	**6**	37626,95	32,82
Eager Loading							
Unearned results							
IDL (CHUNK 1000)							
78849	36	**6506,52**	4258,5	317	7	**2944,59**	3,73
IDL (OR 1000)							
65759	43	6552,52	**3971,9**	**6**	**6**	5611,75	8,53

TABLE IX. RESULTS FOR Q2 EXPERIMENT – ORACLEDB WITHOUT PARTITIONING

ET	SI	MP	DM	Q	DQ	QT	QT/ET
Native Query							
11042	30	2930,01	32	1	1	13,73	0,12
Lazy Loading							
115038	31	7249,11	6380,1	311901	**6**	19780,87	17,20
Eager Loading							
Unearned results							
IDL (CHUNK 1000)							
60294	31	**6506,52**	**4256,5**	**317**	7	**370,32**	0,61

TABLE X. RESULTS FOR Q2 EXPERIMENT – ORACLEDB WITH PARTITIONING

ET	SI	MP	DM	Q	DQ	QT	QT/ET
Native Query							
11575	32	2930,01	14	1	1	11,42	0,10
Lazy Loading							
122091	31	7249,11	6390,1	311901	**6**	23046,38	18,88
Eager Loading							
Unearned results							
IDL (CHUNK 1000)							
108392	33	**6506,52**	**4260,5**	**317**	7	**2428,86**	2,24

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 performed on the database through Doctrine ORM. For example, identical queries with different parameters in the predicates are considered as unique queries.
- Query time [ms] - measured time that was needed to execute all database queries.
- DB QT/ET [%] - the percentage of time that was needed from the total http request time to process queries by the database system. The rest of the time the request spent at the application layer.

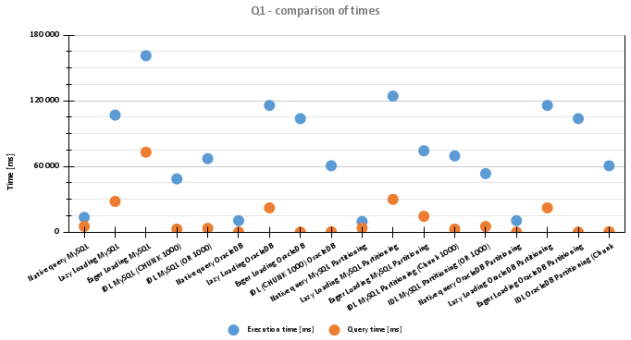

Fig. 1. Visualization of results for experiment Q1 – execution and query times

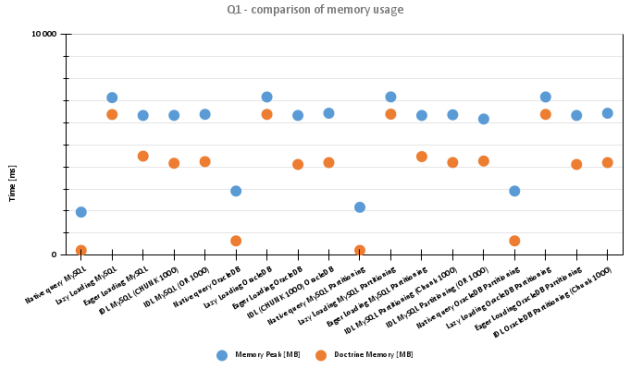

Fig. 2. Visualization of results for experiment Q1 – memory usage

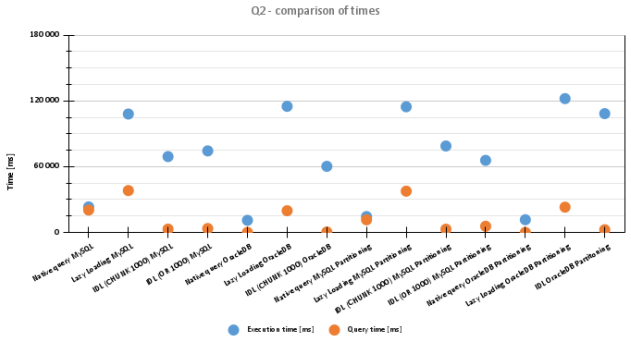

Fig. 3. Visualization of results for experiment Q2 – execution and query times

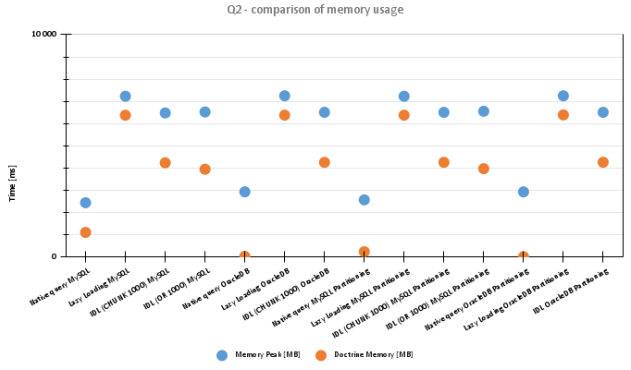

Fig. 4. Visualization of results for experiment Q2 – memory usage

IV. Discussion of Results

In the experiments, it was found that partitioning does not have a completely positive effect on the final performance of the tested experiments in all the situations we tested. In some cases it can be clearly seen that partitioning was definitely a beneficial form of optimization, however in others the opposite can be seen. As an example, the very first experiment over MySQL, where partitioning had a significant effect essentially only on the execution of native query and eager loading, where queries with multiple joins were executed. In the other cases, performance either stagnated completely or even deteriorated.

If we look at the results of experiment Q1 in terms of query time, there was a deterioration of almost 50% when using IDL with OR. On the other hand, there was a decrease of almost 30% for the native query and even 80.27% when using eager loading. Thus, eager loading became even more efficient than using Lazy loading, where query times were almost constant (+6.6% in the case of partitioning).

One cannot help but notice that the results for OracleDB were much worse for partitioning. Not only was there an increase in overall processing time in almost all cases outside of eager loading, but query times were also substantially increased. For a simple native query, we can see a 111% deterioration in query time, and an even greater deterioration of 477% when using IDL. Thus, at first glance, IDL might appear to be inefficient, but it still maintains relatively good results when comparing query time with lazy loading, however, with eager loading it is then comparable in overall execution time and unfortunately it is many times slower for query time.

For the Q2 experiment, we can then see a very similar trend. Unfortunately, here again we cannot make comparisons for eager loading, as we were not able to complete the experiment. Even using OracleDB. Thus, from this perspective, we can easily say that IDL is more efficient than eager loading because it is able to complete the query. Looking at the results tables, it is easy to see that essentially the same evaluation as Q1 holds. For MySQL, again, only the native query was optimized. In other cases, the results are worse using partitioning.

For OracleDB, we can see again a significant deterioration of query time for IDL use by 556%. Similarly, the total query execution time has also increased substantially by 79%. Fundamentally, the total execution time has thus approached that of using Doctrine alone without any optimization. Although still query time is 89.5% lower in comparison. For the second experiment, we can then observe a 17% reduction in query time with partitioning versus the variant without partitioning.

One cannot help but notice an interesting phenomenon in MySQL, where in all cases the CHUNK 1000 variant, i.e., query segmentation after queries with 1000 parameters, required less query time than the alternative OR 1000 variant, which performed significantly fewer queries. For OracleDB, this comparison was not possible because OracleDB does not support more than 1000 parameters in a single database query.

The Doctrine memory usage and the total peak memory required are not evaluated here, as there was no significant change due to partitioning. Minor differences can only be seen for native queries, where in the case of Q2 the required memory for Doctrine was reduced, while for Q1 it remained the same for MySQL and for OracleDB the required memory even increased. For the other parts of the experiments, the memory is essentially comparable between the variants with and without partitioning. The same is true for the MySQL and OracleDB variants of the experiments. The only major difference was system memory, with OracleDB requiring almost 2GB of memory and MySQL making do with 430MB.

VII. Conclusion

From experimentation, measurement and subsequent discussion of the results, it is clear that OracleDB without and with partitioning is fundamentally more powerful than MySQL. However, it cannot be clearly recommended that using OracleDB is the solution to the ORM implementation problem. As can be seen from the results, the overall processing times are in many cases worse than the request processing times with MySQL database. This could be due to, for example, poorer OracleDB support in PHP, or an overall poor implementation of the OCI8 driver in PHP. Thus, it can be easily deduced that in terms of database load, OracleDB is definitely preferable, with the entire request spending an average of 13% of the time querying the database. For MySQL, it was 25.24% of the time. On the other hand, in terms of total execution time, it was an average time of 69802ms for MySQL and 73935ms for OracleDB. Thus, there is a 6.6% disadvantage for OracleDB in this parameter.

In conclusion, the experiments did not show that partitioning had a significant positive effect on the experiments. And that it could be recommended as an optimization technique for IDL, or in general as an optimization technique for using an ORM framework.

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.

References

[1] M. Bandle, J. Giceva, T. Neumann. 2021. „To Partition, or Not to Partition, That is the Join Question in a Real Systém“. *International Conference on Management of Data, SIGMOD 2021* 168-180. Online: Association for Computing Machinery. doi:10.1145/3448016.3452831.

[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] W. Khan, C. Zhang, B. Luo, T. Kumar, E. Ahmed. 2021. Robust Partitioning Scheme for Accelerating SQL Database. *IEEE International Conference on Emergency Science and Information Technology, ICESIT 2021*. Chongqing: Institute of Electrical and Electronics Engineers Inc. 369-376. doi:10.1109/ICESIT53460.2021.9696761

[5] S. Kläbe, K. Sattler. 2023. Patched Multi-Key Partitioning for Robust Query Performance. 26th International Conference on Extending Database Technology, EDBT 2023. Ioannina: OpenProceedings.org. 324-336. doi:10.48786/edbt.2023.26

[6] 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

[7] 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

[8] 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.

[9] V. Salgova, K. Matiasko. 2021. The Effect of Partitioning and Indexing on Data Access Time. *29th Conference of Open Innovations Association FRUCT, FRUCT 2021*. Tampere: IEEE Computer Society. 301-306. doi:10.23919/FRUCT52173.2021.9435500

[10] 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.

[11] 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.

[12] 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.