Translation of article 10.22146/jnteti.v14i2.17315

Comparison Study of Object-Relational Mapping Performance Based on the Implementation of the DSAP

Muhammad Rezy Anshari[1], Redi Ratiandi Yacoub[1], Herry Sujaini[1], Bomo Wibowo Sanjaya[1], Eva Faja Ripanti[1]

[1] Electrical Engineering Study Program, Faculty of Engineering, Tanjungpura University, Pontianak, West Kalimantan 78124, Indonesia

[Submitted: 3 December 2024, Revised: 21 February 2025, Accepted: 16 April 2025]
Corresponding Author: Muhammad Rezy Anshari (email: research.murean@gmail.com)

ABSTRACT — Object-relational mapping (ORM) is a technique that maps in-memory objects and tables in the database, implementing data source architectural patterns (DSAP), namely Data Mapper and Active Record. These patterns require comparison due to performance difference indications and their significant roles in a system's business processes. This study aims to compare and analyze the execution duration and memory consumption of both patterns quantitatively, as well as the functions that influence them in the ORM. The selected ORMs were Doctrine (Data Mapper) and Eloquent (Active Record). The ORM performance was profiled as a library, not bundled in a framework. This profiling encompassed CRUD and lookup operations based on specified measurement metrics, conducted using variations in the number of database records. The profiling process was script-automated, leveraging a combination of Xdebug and Apache Benchmark. The analysis was performed using Kcachegrind and big O notation, resulting in performance graphs, relative percentage differences, and functions' contributions to the performance. Results showed that Active Record excelled in memory consumption, whereas Data Mapper was superior in execution duration in most operation and metrics combinations. Function groups of database transactions, object serialization, and record retrieval were the primary contributors to the performance. Object and database synchronizations became additional contributors to Active Record. The complexity of the largest contributor functions in Data Mapper was higher than that of Active Record. Future studies can utilize automation concepts in the profiling process and substitute Xdebug according to the requirements of the programming languages used by the ORM.

KEYWORDS — ORM, Data Mapper, Active Record, Data Source Architectural Patterns, Execution Duration, Memory Consumption.

I. INTRODUCTION

Object-relational mapping (ORM) is a technique that maps in-memory objects and tables in the database [1]. It is commonly used in developing applications that employ the object-oriented programming (OOP) paradigm and interact with database management systems (DBMS). ORM allows the execution of DBMS operations without directly writing the structured query language (SQL), which is not tied to any specific DBMS, thus improving the efficiency of the software development process [2]. It also enables developers to focus on writing program code, thereby minimizing the likelihood of syntax errors in the SQL; serves as a buffer zone in the cache [3]; and offers solutions to the semantic gap between the DBMS and OOP [4]. ORM may lead to performance degradation in an application, but this might be overlooked, considering the benefits it offers [2], [5]. Hence, performance is essential for an ORM [2]. ORM implements data source architectural patterns (DSAP) with several patterns, such as Data Mapper and Active Record. The key difference between the two lies in the separation between the domain objects and the database. Data Mapper has a layer of mappers that act as an intermediary between the domain objects and the database. As a result, both remain independent of each other. Unlike Data Mapper, Active Record closely binds domain objects to the database to make them simpler [6], [7]. Data Mapper and Active Records are related to the domain model, one of the domain logic patterns that is frequently implemented in the domain layer. The domain layer is one of the primary layers in the software architecture, containing the core domain or the business processes of a system. Domain Model can be simple or complex. Simple Domain Model usually only necessitates one domain object for each database table [6]. Given its simplicity, it is reasonable to combine the domain layer and data source layer into a single object, enabling the use of Active Record [6], [7]. A complex Domain Model can consist of inheritance and various OOP patterns; thus, it can be very different from a database design. Given its complexity, it is more appropriate to separate the domain layer from the data source layer, enabling the use of Data Mapper [6].

The differences between Data Mapper and Active Record in adjusting the interaction between the domain layer and the data source layer indicate potential performance variations when executing the same task. The presence of performance differences and their significance for the ORM, along with the significant roles of both patterns in a system's business process, underscores the necessity of conducting a comparative study of Data Mapper and Active Record.

This study aims to quantitatively compare and analyze the performances of execution duration and memory consumption, as well as functions that influence them in the ORM, utilizing Data Mapper and Active Record. The object of the study was the ORM implementing Data Mapper and Active Record. Doctrine was chosen as the Data Mapper representative, while Eloquent as the Active Record representative. Both are PHP-based popular ORMs. Their popularity owes to their frameworks, namely Symfony for Doctrine and Laravel for Eloquent.

Based on the statistical comparisons reported by packagist.org during this study, Laravel surpassed Symfony at the framework level, with 396 million installations and 33,140 stars, whereas Symfony had 82 million installations and 30,034 stars. On the other hand, at the ORM level, Doctrine outperformed Eloquent, with 233 million installations and 9,993 stars; meanwhile, Eloquent had 42 million installations and 2,703 stars.

References [8]–[10] have compared several ORMs implementing Data Mapper and Active Record. The performance profiling of these studies was done at the framework level. In other words, the ORM performances were measured under the condition that each ORM was bundled in different frameworks. In contrast to prior studies that bundled the ORM into frameworks, this study conducted performance profiling of the ORM in the form of a library so that the ORM was not bundled into frameworks. The selection of ORM in the form of a library was used as a control variable. Therefore, the difference in treatment of each ORM during the comparison process could be minimized. This was done considering that each framework has its own flow when executing the same operations. This study is also different from the previous study [11]. Although both performed ORM profiling in the form of libraries, this study distinguishes itself by comparing ORM with another ORM, namely Eloquent, which utilizes a different DSAP with Doctrine. This study utilized measurement metrics [12], [13] and variations in the number of database records [2], [9], [11], [14] in the profiling processes. This study also analyzed functions that influenced the measured performances. The performances used in this study were execution durations and memory consumption, which had been widely used in comparative studies. This election of the execution duration as the measured performance has previously been done [2], [8], [10], [11], [14]–[18]. Similarly, memory consumption has also been evaluated in earlier works [2], [8], [10], [16]–[18].

Similar to the prior studies, Doctrine and Eloquent were utilized in the application's profiling processes. Profiling was subsequently carried out in that application. The profiling processes in this study were executed automatically through a script leveraging the combination of Xdebug and Apache Benchmark. Xdebug has been previously utilized for profiling [19]–[21]. Similarly, the Apache Benchmark has also been used in comparative studies [15]–[17], [20]. The profiling results were records of call history between functions related to a process, which were then analyzed using Kcachegrind. Prior studies have also analyzed profiling results using Kcachegrind [20], [22], [23]. Functions with dominant contributions were further examined using big O notation to assess the complexity and algorithm performance used. The big O notation has also been discussed in [24]–[27].

This study contributes to filling the methodological gap of prior contrastive studies that have yet to discuss the contribution of ORM functions to measurable performance. This can be achieved by combining Apache Benchmark, Xdebug, Kcachegrind, and big O notation. Therefore, the percentage and complexity of the contributor functions that exert the most significant influences on performance are also obtained while acquiring quantitative performance comparisons. The results of the analysis of the contribution of these associated functions can be further employed as references for ORM optimization purposes in an effort to enhance performance.

II. METHODOLOGY

This comparative study commenced with determining measurement metrics, designing the database to meet the metric needs, preparing the study environment, and concluding with the analysis.

A. MEASUREMENT METRICS

Measurement metrics were based on a series of metrics discussed in previous studies [12], [28]. The metrics were associated with performance when handling create, read, update, delete, and lookup (CRUDL) operations. The lookup operation is related to a single datum based on id. The first measurement metric was relationship, which consisted of CRUDL performances related to inter-table relationships, namely one-to-one, one-to-many, and many-to-many. The second metric was the polymorphic query (PQ), which consisted of CRUDL performances related to tables connected by inheritance. The final metric was the additional null value (ANV), which consisted of CRUDL performances related to the presence of some table attributes that were null. In this study, change propagation and change isolation operations were excluded, as neither Doctrine nor Eloquent provides an abstraction for them.

B. DATABASE DESIGN

The predetermined measurement metrics were utilized as a guide in the process of designing the database. The design for the relationship metric pertained to employee management. Each employee (user) has a personal desk, many personal tasks, and many roles that may resemble or differ from those of others. Thus, the relation between the user entity and desk is one-to-one, between the user and tasks is one-to-many, and between users and roles is many-to-many. Each entity can be in a single table, but due to the many-to-many relation, an additional pivot table is necessary for the user and role, namely user_role. Consequently, the database design for the relationship metric consisted of five tables: user, desk, task, role, and user_role. The tables and their attributes in the database design for the relationship metric are shown in Figure 1. This design was employed to assess the performance of both ORMs in handling various inter-table relations.

The design of PQ and ANV metric databases was built based on information management for contractual and permanent employees. A permanent employee has an employee identification number (*nomor induk karyawan*, NIK), while a contractual employee has a contract duration (contract_duration). According to this context, there are three entities: employee, contract, and permanent. The employee entity represented core information of the employee and served as the parent of the Contract and Permanent entities, which denote contractual and permanent employees, respectively. This entity also had name and address attributes. The Contract entity had the contract_duration attribute, while Permanent had the nik attribute. Each ANV and PQ metric employed different mapping strategies for these entities.

The design for the PQ metric employed the single table (ST) mapping strategy [6]; hence, three tables were required: employee, contract, and permanent (Figure 2). The id attribute in the contract and permanent tables served as both primary key (PK) and foreign key (FK) of the employee table. The database design using table per class (TPC) was utilized to test the performance of both ORMs in handling inheritance relations in the related tables.

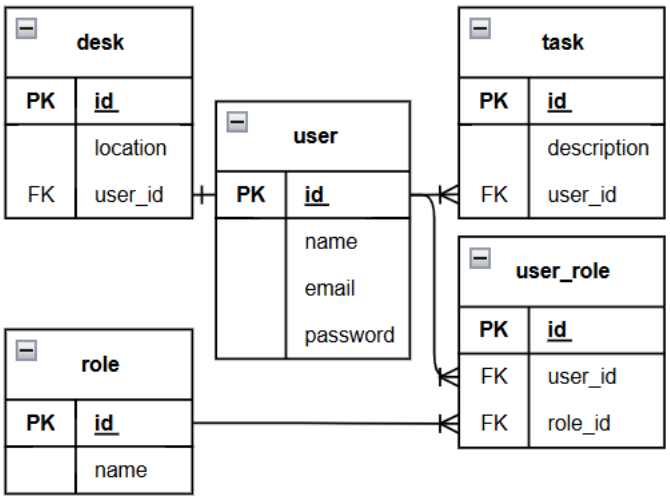

Figure 1. Design of the relationship metric database.

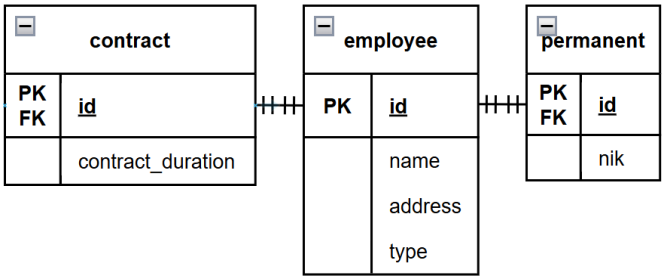

Figure 2. Design of the PQ metric database.

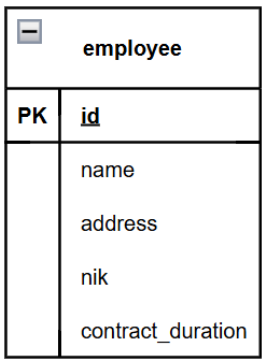

Figure 3. Design of the ANV matric database.

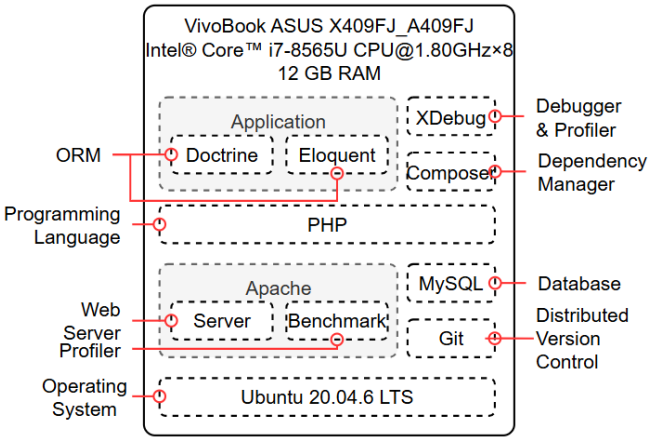

Figure 4. Software and infrastructure.

The design for the ANV metric employed the ST mapping strategy [6]; hence, all three entities' attributes were integrated in the employee table (Figure 3). In the same record, one of the contract_duration attributes or nik was certain to be null since no employee had both contract and permanent status simultaneously. The database design using ST was utilized to test the performance of both ORMs when handling the presence of some attributes with null values.

C. STUDY ENVIRONMENT

1) DEVICES AND INFRASTRUCTURE

Figure 4 displays the devices and infrastructure used in this study. The devices used were VivoBook ASUS X409FJ_A409FJ with CPU Intel i7-8565U (8) @ 4.6 Ghz and RAM of 12 GB. The study environment infrastructures employed were Ubuntu (v20.04), PHP (v8.2), MySQL (v8), Apache Server (v2.4.41), Apache Benchmark (v2.3), Xdebug (v3.3.2), Composer (v2.5.5), Git (v2.25.1), and Kcachegrind (v0.8) operating systems. Each infrastructure's component was separated from one another, except for Apache, MySQL, and PHP, which were integrated in the same XAMPP application. The installation guide for each infrastructure component is available through their respective links, while Ubuntu can be accessed at https://ubuntu.com/tutorials/install-ubuntu-desktop. At the same time, Apache (Server dan Benchmark), MySQL, and PHP can be accessed via https://www.apachefriends.org. Xdebug is accessible at https://xdebug.org/docs/install; Composer at https://getcomposer.org /download; the installation for Git at https://git-scm.com/book/en/v2/Getting-Started-Installing-Git; and Kcachegrind at https://kcachegrind.github.io/html/Download.html.

2) COMPONENTS OF THE OBJECT OF THE STUDY

Profiler and Profiling Object (PO) are the components of the object study in the form of a PHP programming language-based application. A Profiler is an executable script that automatically conducts profiling in PO. PO consists of the ORM's implementation based on the measurement metrics. PO was developed utilizing representational state transfer (REST) architecture. This architecture was selected because it aligns with how Profiler works, which requires access to the endpoints associated with the measurement metrics. The installation for Profiler and PO involved cloning the repository and executing the Composer installation command. Detailed installation instructions, source code, and Profiler and PO structures are available in the public repository on the GitHub site: https://github.com/devmurean/orm-profiling.

In general, the Profiler and PO directory structure consisted of three basic directories, namely App, Profiler, and sql_dump, along with three directories generated automatically during the profiling process, namely doctrine_metadata, inputs, and result. The App directory is a directory for PO, which consists of two subdirectories: Doctrine and Eloquent. Each subdirectory had three documents corresponding to the measurement metrics. These documents are Relationship.php, PolymorphicQuery.php, and AdditionalNullValue.php. They consisted of the implementation of the CRUDL operation for each measurement metric. The Profiler directory contained the Profiler application, while the sql_dump directory contained the SQL documents used as seeder data for each metric. The doctrine_metadata directory contained the metadata cache in Doctrine; the input directory contained a list of inputs used by the Profiler when profiling operations needing inputs, such as

create and update. Meanwhile, the results directory contained the profiling results that could be analyzed using Kcachegrind.

The PO had five dynamic endpoints for each operation defined in the routes.php document within /{orm}/{metrik}/{operasi}/{*id*} format. Words enclosed in curly brackets are the dynamic components of the endpoint, which can be filled as required. The {orm}, {metrik}, and {operasi} components were employed to determine the ORM and measurement metric that would be profiled, whereas {id} was used to access the related records. The {id} component was exclusively employed by update, delete, and lookup operations. Writing values for {orm} and {metrik} utilized kebab-case and lowercase format without abbreviations (example: additional-null-value for the ANV metric).

The algorithm for implementing the CRUDL for each metric is simple. The create operation generated new records within the corresponding table based on the input and returned the results in the JavaScript Object Notation (JSON) format. The update operation began by finding the related records based on the id. The changes were then recorded in the corresponding tables based on the input before returning the updated record in the JSON format. The delete operation began by searching the related records based on the id and removing their existence from the corresponding tables. The create, delete, and update operations were encapsulated within database transactions. The read operation retrieved all records (without inter-table relationship), serialized them in the object collection, and returned them in the JSON format. The lookup operation began by locating the related records based on the id, followed by serializing them along with the inter-table relations in the form of an object and returning the object in the form of JSON.

D. PROFILING

The profiling process was conducted with several variations in the number of records: 100, 1,000, 10,000, and 100,000. The process was automatically executed using a script (with administrator or sudo permission) through the **`php profiler`** command. This command was executed via the command line interface (CLI), which triggered the Profiler to operate. The Profiler executed the Apache Benchmark to access every endpoint in the PO, thereby triggering Xdebug to commence the profiling process, resulting in callgrind-compatible files. All files and results were then stored in the directory (provided that no changes were made to the PO configuration). The file naming adhered to the specified format, comprising the ORM name, metric, operation, number of records, and timestamp. For instance, the name format for the Doctrine output was cachegrind.out.doctrine_polymorphic-query_create_record=100.1234.gz, representing the PQ metric, create operation, and the record number of 100.

E. ANALYSIS

The analysis was carried out in three stages, commencing by comparing the profiling results presented based on the performance comparison graph and the relative percentage difference (RPD) graph [29]. The RPD graphs exhibit the significance of the performance difference between the two ORMs in percent units. The RPD with the positive trend shows that the importance of the performance difference between the two ORMs increases as the number of records increases. Conversely, when the trend is negative, the significance of the performance difference decreases.

The subsequent stage was tabulating the DSAP implementer functions. These functions were the dominant contributors to the performance of both ORMs in every database operation and measurement metric. The tabulation was carried out based on analysis using Kcachegrind. The analysis commenced with the functions executing CRUDL operations, namely the functions of each class associated with the measurement metrics: relationship, AdditionalNullValue, and PolymorphicQuery. The criteria for determining dominance were grounded in the Pareto principle [30]; hence, only 20% of the functions contributing to 80% of the performance are deemed dominant. The final stage of the analysis involved an assessment utilizing big O notation to ascertain the complexity and algorithm performance of the functions that contribute the most to operations in each measurement metric.

III. RESULTS AND DISCUSSION

A. COMPARISON OF EXECUTION DURATION PERFORMANCE

The performance of the create operation of both ORMs alternately outperformed each other on the relationship metric (Figure 5), with a negative RPD trend as the number of records increased. It was found that the performance duration of Eloquent for 100 records was higher (> 200%) than for other numbers of records. Through the Kcachegrind visualization, the cause of the increase in duration was identified as a uniform increase in duration across related functions. Doctrine demonstrated a dominant performance advantage in ANV and PQ metrics. In line with the relationship metric, the RPD ANV metric had a negative trend, whereas the PQ metric had a positive trend.

As illustrated in Figure 6, The read operation using Doctrine exhibited superior performance on the relationship metric with the negative RPD trend. In contrast, Eloquent showed excellent performance for the ANV and PQ metrics, with a positive RPD trend for both metrics. The execution duration for both ORMs in the read operation was directly proportional to the number of records added.

In the update and delete operations for the relationship metric, Eloquent had the advantage (Figure 7 and Figure 8). The RPD for update and delete operations had a positive trend. Based on the graphic, there is a spike in the execution duration for Doctrine when the record was 10,000 and 100,000. A surge of 86% occurred in the update operation and 130% occurred in the delete operation. In line with the execution duration, a similar surge was observed in the RPD under the same conditions. The RPD for the update operation demonstrated an increase of 57.6%, while it was 106.2% for the delete operation. These surges resulted from similar causes, namely the increase in the execution duration for the execute function of the PDOStatement class (internal PHP) of ±1,000% at 100,000 records compared to 10,000 records.

For the ANV and PQ metrics, in the update (Figure 7) and the delete operation (Figure 8), Doctrine outperformed Eloquent. The RPD for both metrics on the update and delete operations showed a positive trend.

In the lookup operation, Doctrine outperformed Eloquent (Figure 9) in the relationship and PQ metrics. The RPD for the lookup operation for the relationship metric had a negative trend, while the PQ metric had a positive trend. A surge in execution duration was observed for the relationship metric

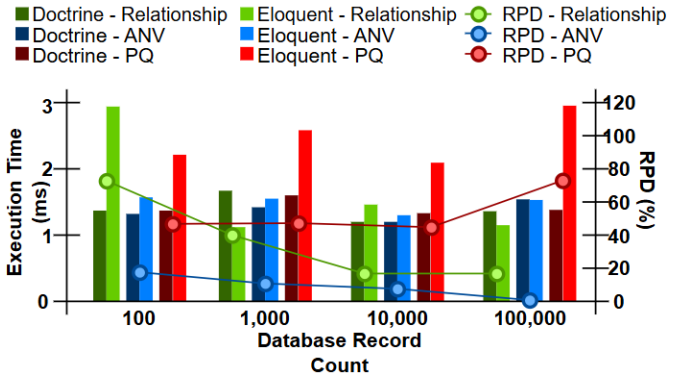

Figure 5. Execution duration and RPD in the create operation.

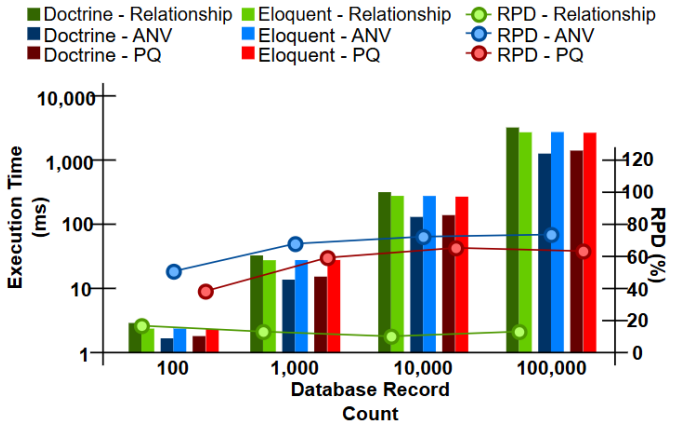

Figure 6. Execution duration and RPD in the read operation.

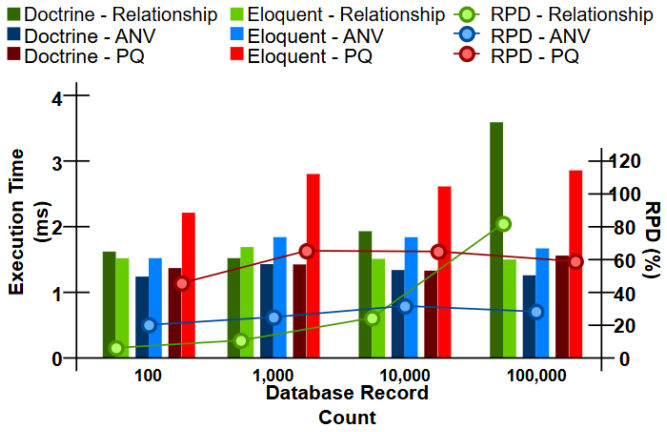

Figure 7. Execution duration and RPD in the update operation.

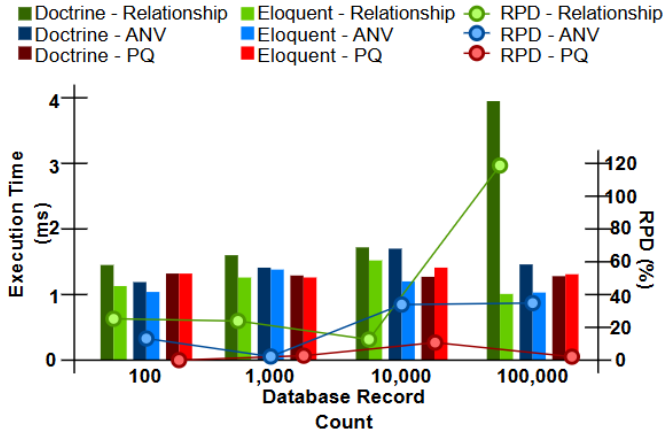

Figure 8. Execution duration and RPD in the delete operation.

when the record was 10,000 and 100,000 for Doctrine (686%) and Eloquent (571%). The cause of the surge was similar to that of the update and delete operations for the same metric. Despite experiencing a surge in execution duration for the same reason, no surge occurred at the RPD. For the ANV metric, the performance of Eloquent outperformed Doctrine, with a negative RPD trend.

B. COMPARISON OF MEMORY CONSUMPTION PERFORMANCE

The memory consumption performance of Doctrine and Eloquent had a tendency to be constant, except for the read operation, which was directly proportional to the number of records added (Figure 10). Eloquent absolutely outperformed the performance of Doctrine in each operation and metric. Memory consumption of both ORMs for all metrics increased consistently with the number of records. The RPD for the relationship metric had a positive trend, whereas the ANV and PQ had a negative trend.

C. CONTRIBUTION OF DSAP IMPLEMENTOR FUNCTIONS

1) DOCTRINE

Based on the Kcachegrind analysis, seven DSAP implementor functions dominantly contributed to Doctrine, which was divided into two groups: mapper and domain objects. Functions included in the Mapper group were `App\Doctrine\EM::make` (D1), `Doctrine\ORM\EntityManager->flush` (D2), `Doctrine\ORM\EntityManager->persist` (D3), `Doctrine\ORM\EntityManager->find` (D4), and `Doctrine\ORM \EntityRepository->findAll` (D5). The D1 function was the constructor for Mapper, D2 was associated with the database transaction process, D3 was associated with synchronization management between objects and databases, D4 was associated with single record retrieval, and D5 was associated with multiple record retrieval.

The functions of the domain object group were `App\Doctrine\Helper\ModelCollection->serialize` (D6), and `App\Doctrine\Models\User->serialize` (D7). The D6 function was related to the serialization for collections, and D7 was related to the serialization specific to the User object domain.

For the relationship metric, it was found that in the create operation, the contribution to the execution duration was dominated by the D1 and D2 functions. In contrast, the D2 and D3 functions dominated memory consumption. For the read operation, the contribution to execution duration and memory consumption was dominated by the D5 function. For the update and delete operations, the contribution to execution duration was dominated by the D2 and D4 functions, whereas the D4 function dominated memory consumption. For the lookup operation, the contribution to execution duration and memory consumption was dominated by D4 and D7 functions.

For the ANV metric, it was found that in the create operation, the contribution to execution duration was dominated by the D1 and D2 functions, while the D2 and D3 functions dominated memory consumption. For the read operation, the contribution to execution duration and memory consumption was dominated by the D5 function, while D5 and D6 dominated the contribution to memory consumption. For the update operation, the contribution to execution duration was dominated by D1, D2, and D4 functions, whereas D4

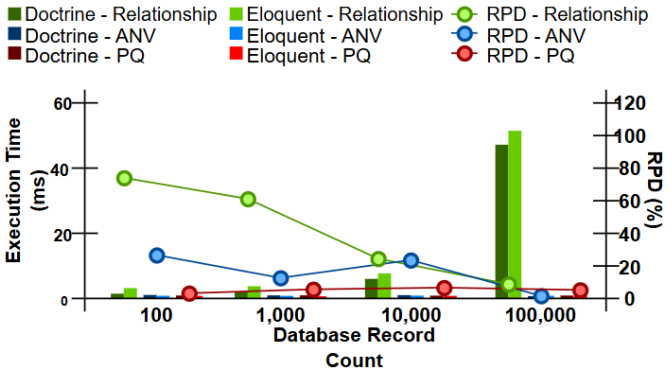

Figure 9. Execution duration and RPD in the lookup operation.

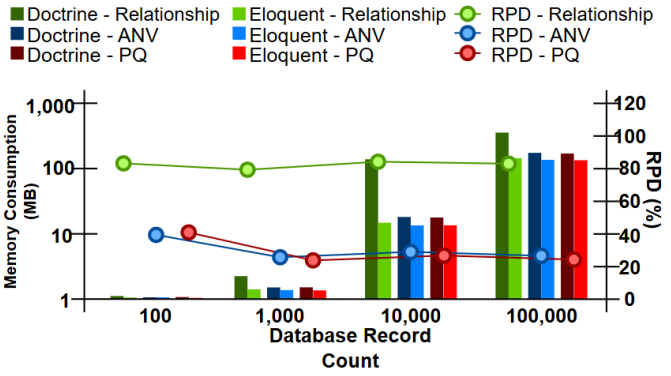

Figure 10. Execution duration and RPD in the read operation.

dominated the contribution to memory consumption. For the delete operation, the contribution to execution duration was dominated by D1, D2, and D4, while D4 dominated memory consumption. For the lookup operation, the contribution to execution duration was dominated by D1 and D4 functions, whereas the D4 function dominated memory consumption.

For the ANV metric, it was found that in the create operation, the contribution to execution duration was dominated by D1 and D2, while D2 and D3 dominated memory consumption. For the read operation, the contribution to execution duration was dominated by the D5 function, while D5 and D6 dominated the contribution to memory consumption. For the update operation, the contribution to execution duration was dominated by D1, D2, and D4 functions, whereas D4 dominated the contribution to memory consumption. For the delete operation in the PQ metric, the performance of execution duration was dominated by D1, D2, and D4, while D4 dominated memory consumption. For the lookup operation, the contribution to execution duration was dominated by D1 and D4 functions, whereas the D4 function dominated memory consumption.

An analysis of the contribution of the functions associated with the Data Mapper implementation revealed that four functions in Doctrine exhibited the most significant contribution to specified metrics and operations, namely D2, D4, D5, and D7 functions. Table I presents the percentage range and Table II presents the contribution positions of these functions to operations and metrics.

The D2 function for execution duration had the most significant contribution to each create operation across all metrics. Despite having the largest contribution percentage, its percentage tended to decrease as the number of records increased. In the delete and update operations across all metrics, this function's contribution was also dominant,

TABLE I
DOCTRINE: FUNCTIONS OF THE LARGEST CONTRIBUTORS

ORM	Code	Contribution Percentage Dominance	
		Execution Duration	*Memory Consumption*
Doctrine	D2	55.8% – 62.2%	63.4% – 66.2%
	D4	42.4% – 68.1%	68.9% – 88.5%
	D5	81.8% – 98.8%	69.5% – 86.8%
	D7	31.3% – 93.5%	–

TABLE II
DOCTRINE: CONTRIBUTION POSITION TOWARDS OPERATIONS AND METRICS

Metric	Execution Duration					Memory Consumption				
	C	*R*	*U*	*D*	*L*	*C*	*R*	*U*	*D*	*L*
Relationship	D2	D5	D4	D4	D7	D2	D5	D4	D4	D4
ANV	D2	D5	D4	D4	D4	D2	D5	D4	D4	D4
PQ	D2	D5	D4	D4	D4	D2	D5	D4	D4	D4

although not the largest, because create, update, and delete operations typically involve database transactions. Moreover, the D2 function served as the largest contributor to memory consumption performance in every create operation of all metrics.

The D4, D5, and D7 functions had a dominant contribution to the operation that performed record retrieval (read, update, delete, and lookup). The D4 function had the largest contribution to the operation that performed record retrieval (update, delete, and lookup) in all metrics, except for the relationship metric in the lookup operation. In update and delete operations for the relationship metric, the percentage contribution of D4 function tended to increase in direct proportion to the increase in the number of records. In contrast, in the lookup operations, it was otherwise. The D4 function was also the primary contributor to memory consumption performance on every update, delete, and lookup operation for all metrics.

The D5 function contributed the largest percentage contribution to the metrics and operations involving multiple record retrieval, i.e., read operation in all metrics. The percentage contribution of the D5 function increased directly proportional to the increase in the number of records. In addition, this function was the largest contributor to the memory consumption performance in every operation across all metrics.

The D7 function, related to single record retrieval, had the largest contribution to the relationship metric in the lookup operation, which tended to increase as the number of records increased. However, this function was not the largest contributor to memory consumption in the lookup operation for the relationship metric. That position is occupied by function D4.

2) ELOQUENT

Ten functions dominantly contributed to Eloquent. Based on the operating scope, these functions were grouped into database transaction manager, object serialization, object and database synchronization manager, record retrieval manager, and relationship manager.

The group of database transaction managers consisted of a single function, namely `Illuminate\Database\Capsule\Manager::__callStatic` (E1). The group of object serialization consisted of a single function, namely

`Illuminate\Database\Eloquent\Model->jsonSerialize` (E2). The group of object and database synchronization manager consisted of four functions, namely `Illuminate\Database\Eloquent\Model::__callStatic` (E3), `Illuminate\Database\Eloquent\Model->saveOrFail` (E4), `Illuminate\Database\Eloquent\Model->push` (E5) dan `Illuminate\Database\Eloquent\Model->delete` (E6). Specifically, the E3 function served as a model constructor. The E4 function handled synchronization in the create and update operations; it would also throw an exception in case of errors. The E5 function handled synchronization comprising the related relationship. Meanwhile, the E6 function specifically handled the delete operation.

The group of record retrieval managers comprised two functions: `Illuminate\Database\Eloquent\Model::all` (E7) and `Illuminate\Database\Eloquent\Builder->find` (E8). The E7 function handled multiple record retrievals, whereas the single record retrieval was handled by the E8 function.

The final group was the relationship manager. This group comprised two functions: `Illuminate\Database\Eloquent\Model->loadMissing` (E9) and `Illuminate\Database\Eloquent\Model::with` (E10). Both functions were utilized to load certain relationships for the corresponding domain object, with the distinction that the E10 function was called during the initiation process of the domain object. On the other hand, the E9 function could only be called after the initiation.

For the Relationship metric, it was found that in the create operation, the contribution to execution duration was dominated by the E1, E2, and E3 functions. For the read operation, the contribution to execution duration was dominated by the E2 and E7 functions. For the update operation, the contribution to execution duration was dominated by the E1, E2, E3, and E4 functions. For the delete operation, the contribution to execution duration was dominated by E1 and E3 functions. For the lookup operation, the contribution to execution duration was dominated by the E2 and E7 functions. For the performance of the memory consumption, the E1 function dominated the create, update, and delete operations. The E7 function dominated the read operation, while the E8 function dominated the lookup operation.

For the ANV metric, it was found that in the create operation, the contribution to execution duration was dominated by the E1, E2, and E3 functions. For the read operation, the contribution to execution duration was dominated by the E2 and E7 functions. For the update operation, the contribution to execution duration was dominated by the E1, E3, and E4 functions. For the delete operation, the contribution to execution duration was dominated by E1, E3, and E7 functions. For the lookup operation, the contribution to execution duration was dominated by the E3 and E2 functions. For the performance of memory consumption, the E1 function dominated the create, update, and delete operations. The E7 function dominated the read operation, while the E3 function dominated the lookup operation.

In the PQ metric, it was found that in the create operation, the contribution to execution duration was dominated by the E1, E2, E3, and E9 functions. For the read operation, the contribution to execution duration was dominated by the E2 and E7 functions. For the update operation, the contribution to execution duration was dominated by the E1, E2, E5, and E8 functions. For the delete operation, the contribution to execution duration was dominated by E1, E3, and E6 functions. For the lookup operation, the contribution to execution duration was dominated by E8 and E10 functions. For the memory consumption performance, the E1 and E3 functions dominated the creation operation. The E7 function dominated the read operation, the E1 and E8 functions dominated the update operation, the E1 function dominated the delete operation, and the E8 function dominated the lookup operation.

Table III and Table IV present the summary of the most significant contributing functions and their contribution positions relative to specific metrics and operations, respectively. The E1, E2, E3, and E8 functions were the most significant contributors to the execution duration performance, while functions E1, E3, E7, and E8 were the largest contributors to the memory consumption performance.

For execution duration, the E1 function significantly contributed to the create, update, and delete operations for the relationship metric. For the relationship metric during the update operation, the percentage of E1 was directly proportional to the increase in the number of records. The E1 function also contributed the most to the update and delete operations for the ANV metric. The E1, functioning as the largest contributor, was also seen in the create and delete operations for the PQ metric. For memory consumption, the E1 function was the primary contributor to the create, update, and delete operations for all metrics.

For execution duration, the E3 function was the most significant contributor to the create and lookup operations for the ANV metric. For the memory consumption, the E3 function served as the largest contributor to the lookup operation for the ANV, which could be attributed to the absence of inter-table relationships—unlike the other metrics.

In the read operation across all metrics, the E2 function contributed the most to execution time, while the most significant contributor to memory consumption was the E7 function. These results suggest that, in the Active Record, the object serialization process is the primary contributor to execution duration in the read operation, whereas the record retrieval process is the largest contributor to memory consumption. The percentage contribution of the E2 function to execution duration increased proportionally with the number of records. The percentage of contribution of the E7 function to memory consumption decreased as the number of records increased.

The E8 function for execution duration was the most significant contributor to the lookup operation for the relationship metric and the update and lookup operations for the PQ metric. Meanwhile, for memory consumption, the E8 function was the primary contributor to the lookup operation for the relationship and PQ metrics. The percentage contribution of the E8 function to execution duration increased proportionally with the number of records.

D. BIG O NOTATION

Big O notation for each function with the largest contribution, both to execution duration and memory consumption, has been obtained. In Doctrine, the D2 function had a complexity O(mn+o), with a variable context m denoting the number of entities in the insertion process, variable n

TABLE III
ELOQUENT: FUNCTIONS OF THE LARGEST CONTRIBUTORS

ORM	Code	Contribution Percentage Dominance	
		Execution Duration	*Memory Consumption*
Eloquent	E1	28.2% – 51.5%	68.6% – 93.3%
	E2	55.8% – 62.1%	–
	E3	30.3% – 72.5%	90.5%
	E7	–	88.0% – 94.4%
	E8	25.5% – 97.7%	82.5% – 87.2%

TABLE IV
ELOQUENT: POSITION OF CONTRIBUTION TO OPERATIONS AND METRICS

Metric	Execution Duration					Memory Consumption				
	C	*R*	*U*	*D*	*L*	*C*	*R*	*U*	*D*	*L*
Relationship	E1	E2	E1	E1	E8	E1	E7	E1	E1	E8
ANV	E3	E2	E1	E1	E3	E1	E7	E1	E1	E3
PQ	E1	E2	E8	E1	E8	E1	E7	E1	E1	E8

denoting the number of associated attributes, and variable o denoting the number of triggered events. The D4 and D5 functions had complexity O(mn), with a variable context m denoting the number of records processed (specifically for D5, it denotes the number of records in the associated table) and variable n denoting the number of attributes per record. The D7 function had complexity O(n), with a variable context n denoting the number of attributes in the record processed.

In Eloquent, the E1 and E3 functions had complexity O(1). The E2, E7, and E8 functions had the same complexity, namely O(n). The variable context for E2 was the number of records resulting from the query, E7 was the number of records in the related table, and E8 was the number of records based on the id list, which was the search parameter.

Based on the complexity comparison, functions in Doctrine exhibited a greater complexity, involving two or more variables, in contrast to Eloquent, which only involved a single variable or even a constant. This occurred because of the centralization of database operations handled by Data Mapper, which is handled by a manager called EntityManager. This aligns with the finding of function D2, which is related to database transactions, demonstrating the highest complexity compared to other functions. On the other hand, the decentralization of database operation handling in Active Record resulted in lower complexity, as evidenced by the E1 function, which handled a similar process to the D2 function but exhibited a complexity of O(1).

E. LIMITATION

This study is limited to a comparative analysis and does not address performance improvements or optimization strategies. The comparison was conducted in the PHP-based ORM. This limitation was imposed since one of the Profiler components, Xdebug, can only perform profiling in PHP-based applications. The measured performance was limited to the execution duration and memory consumption. This performance limitation was based on the feature limitation available in the Kcachegrind.

IV. CONCLUSION

The Active Record represented by Eloquent had lower memory consumption than that of the Data Mapper represented by Doctrine, with the RPD of 40–90% (relationship) and 20–60% (PQ and ANV). The performance of the Data Mapper's execution duration outperformed the Active Record, with an average RPD of 35.3%, with several exceptions in the specific operations and metrics. These exceptions occurred in the delete operation (all metrics), update (relationship), read (relationship), and lookup (ANV), with average RPDs for each being 23%, 30%, 17.4%, and 16%, respectively. Functions related to the transaction database, object serialization, and record retrieval for both DSAPs were the most significant contributors to both performances, with additional object and database synchronization, specifically for Active Record. The percentage range of the functions with the largest contribution for Data Mapper was 31–98%, while for Active Record, it was 25–97%. The highest functional complexity of Data Mapper was O(mn+o), while O(n) was for Active Record.

Future studies can leverage automation concepts in the Profiler used in this study. Profiler components, especially Xdebug, can be substituted with similar components according to the programming language of the application to be profiled.

CONFLICTS OF INTEREST

The authors declare no conflicts of interest.

AUTHORS' CONTRIBUTIONS

Conceptualization, Muhammad Rezy Anshari, Redi Ratiandi Yacoub, and Herry Sujaini; methodology, Muhammad Rezy Anshari; software, Muhammad Rezy Anshari; validation, Bomo Wibowo Sanjaya and Eva Faja Ripanti; formal analysis, Muhammad Rezy Anshari; investigation, Muhammad Rezy Anshari; resource, Muhammad Rezy Anshari; data curation, Muhammad Rezy Anshari; writing—original draft preparation, Muhammad Rezy Anshari, Redi Ratiandi Yacoub, and Herry Sujaini; writing—reviewing and editing, Muhammad Rezy Anshari, Redi Ratiandi Yacoub, Herry Sujaini, Bomo Wibowo Sanjaya, and Eva Faja Ripanti; visualization, Muhammad Rezy Anshari and Bomo Wibowo Sanjaya; supervision, Redi Ratiandi Yacoub and Herry Sujaini; project administration, Muhammad Rezy Anshari; financial acquisition, Muhammad Rezy Anshari.

REFERENCES

[1] V. Sivakumar, T. Balachander, Logu, and R. Jannali, "Object relational mapping framework performance impact," *Turk. J. Comput. Math. Educ.*, vol. 12, no. 7, pp. 2516–2519, Apr. 2021.

[2] A.E. Güvercin and B. Avenoglu, "Performance analysis of object-relational mapping (ORM) tools in .NET 6 environment," *Bilişim Teknol. Derg.*, vol. 15, no. 4, pp. 453–465, Oct. 2022, doi: 10.17671/gazibtd.1059516.

[3] G. Vial, "Lessons in persisting object data using object-relational mapping," *IEEE Softw.*, vol. 36, no. 6, pp. 43–52, Nov./Dec. 2019, doi: 10.1109/MS.2018.227105428.

[4] M. Gorodnichev *et al.*, "Exploring object-relational mapping (ORM) systems and how to effectively program a data access model," *PalArch's J. Archaeol. Egypt/Egyptol.*, vol. 17, no. 3, pp. 615–627, Nov. 2020, doi: 10.48080/jae.v17i3.141.

[5] A. Joshi and S. Kukreti, "Object relational mapping in comparison to traditional data access techniques," *Int. J. Sci. Eng. Res.*, vol. 5, no. 6, pp. 540–543, Jun. 2014.

[6] M. Fowler, *Patterns of Enterprise Application Architecture*. Boston, MA, USA: Addison-Wesley, 2003.

[7] T. Nguyen, "Elementary event storage," Undergraduate thesis, Metropolia University of Applied Sciences, Helsinki, Finland, 2018.

[8] A. Niarman, Iswandi, and A.K. Candri, "Comparative analysis of PHP frameworks for development of academic information system using load and stress testing," *Int. J. Softw. Eng. Comput. Sci.*, vol. 3, no. 3, pp. 424–436, Dec. 2023, doi: 10.35870/ijsecs.v3i3.1850.

[9] P. Garbarz and M. Plechawska-Wójcik, "Comparative analysis of PHP frameworks on the example of Laravel and Symfony," *J. Comput. Sci. Inst.*, vol. 22, pp. 18–25, Mar. 2022, doi: 10.35784/jcsi.2781.

[10] P.R. Chavan and S. Pawar, "Comparison study between performance of Laravel and other PHP frameworks," *IJRESM*, vol. 4, no. 10, pp. 27–29, Oct. 2021.

[11] M. Choina and M. Skublewska-Paszkowska, "Performance analysis of relational databases MySQL, PostgreSQL and Oracle using Doctrine libraries," *J. Comput. Sci. Inst.*, vol. 24, pp. 250–257, Sep. 2022, doi: 10.35784/jcsi.3000.

[12] S. Holder, J. Buchan, and S.G. MacDonell, "Towards a metrics suite for object-relational mappings," in *Model-Based Softw. Data Integr.*, R.D. Kutsche dan N. Milanovic, Eds. 2008, pp. 43–54, doi: 10.1007/978-3-540-78999-4_6.

[13] M. Lorenz *et al*., "Object-relational mapping reconsidered," in *Proc. 50th Hawaii Int. Conf. Syst. Sci.*, 2017, pp. 4877–4886.

[14] U. Ibrahim, J.B. Hayfron-Acquah, and F. Twum, "Comparative analysis of CodeIgniter and Laravel in relation to object-relational mapping, load testing and stress testing," *Int. Res. J. Eng. Technol. (IRJET)*, vol. 5, no. 2, pp. 1471–1475, Feb. 2018.

[15] J.A. Yang and S.A. Aklani, "Performance analysis between interpreted language-based (Laravel) and compiled language-based (Gin) web frameworks," *Comput. Based Inf. Syst. J.*, vol. 11, no. 1, pp. 12–16, Mar. 2023, doi: 10.33884/cbis.v11i1.6583.

[16] M. Laaziri, K. Benmoussa, S. Khoulji, and M.L. Kerkeb, "A comparative study of PHP frameworks performance," *Procedia Manuf.*, vol. 32, pp. 864–871, Apr. 2019, doi: 10.1016/j.promfg.2019.02.295.

[17] H. Abutaleb, A. Tamimi, and T. Alrawashdeh, "Empirical study of most popular PHP framework," in *2021 Int. Conf. Inf. Technol. (ICIT)*, 2021, pp. 608–611, doi: 10.1109/ICIT52682.2021.9491679.

[18] D. Zmaranda *et al*., "Performance comparison of CRUD methods using NET object relational mappers: A case study," *Int. J. Adv. Comput. Sci. Appl. (IJACSA)*, vol. 11, no. 1, pp. 55–65, Jan. 2020, doi: 10.14569/IJACSA.2020.0110107.

[19] S. Selvaraj, "Performance monitoring and debugging," in *Building Real-Time Marvels with Laravel*. Berkeley, CA, USA: Apress, 2024, pp. 259–283.

[20] A. Šimec, D. Lozić, and L.T. Golubić, "Benchmarking PHP modules," *Informatologia*, vol. 50, no. 1/2, pp. 95–100, Jun. 2017.

[21] B.A. Azad, P. Laperdrix, and N. Nikiforakis, "Less is more: Quantifying the security benefits of debloating web applications," in *SEC'19, Proc. 28th USENIX Conf. Secur. Symp.*, 2019, pp. 1697–1714.

[22] A. Gocht, R. Schöne, and J. Frenzel, "Advanced Python performance monitoring with Score-P," in *Tools High Perform. Comput. 2018/2019*, H. Mix *et al*.., Eds. 2021, pp. 261–270, doi: 10.1007/978-3-030-66057-4_14.

[23] J. Buša Jr., S. Hnatič, and O.V. Rogachevsky, "Performance analysis and optimization of MPDRoot," in *Proc. 9th Int. Conf. Distrib. Comput. Grid Technol. Sci. Educ. (GRID'2021)*, 2021, pp. 75–79, doi: 10.54546/MLIT.2021.22.70.001.

[24] S. Bae, *JavaScript Data Structures and Algorithms: An Introduction to Understanding and Implementing Core Data Structure and Algorithm Fundamentals.* Berkeley, CA, USA: Apress, 2019.

[25] I. Chivers and J. Sleightholme, "An introduction to algorithms and the big O notation," in *Introduction to Programming with Fortran*. Cham, Switzerland: Springer, 2015, pp. 359–364.

[26] A.J. Lockett, "Performance analysis," in *General-Purpose Optimization Through Information Maximization.* Heidelberg, Germany: Springer, 2020, pp. 239–262.

[27] F. Shamssoolari, "The examination of analyzing data by algorithm performance," *Int. J. Comput. Sci. Mob. Comput.*, vol. 8, no. 9, pp. 167–171, Sep. 2019.

[28] Z. Xu, J. Zhu, L. Yang, and C. Zuo, "Mining the relationship between object-relational mapping performance anti-patterns and code clones," in *35th Int. Conf. Softw. Eng. Knowl. Eng.*, 2023, pp. 1–6, doi: 10.18293/SEKE2023-161.

[29] R.E. Miller, *Optimization: Foundations and Applications.* New York, NY, USA: John Wiley & Sons, 2011.

[30] J. Backhaus, "The Pareto principle," *Anal. Krit.*, vol. 2, no. 2, pp. 146–171, Nov. 1980, doi: 10.1515/auk-1980-0203.