Reniers *et al. Journal of Internet Services and Applications* (2017) 8:1
DOI 10.1186/s13174-016-0052-x

Journal of Internet Services and Applications

RESEARCH Open Access

Object-NoSQL Database Mappers: a benchmark study on the performance overhead

Vincent Reniers*, Ansar Rafique, Dimitri Van Landuyt and Wouter Joosen

Abstract

In recent years, the hegemony of traditional relational database management systems (RDBMSs) has declined in favour of non-relational databases (NoSQL). These database technologies are better adapted to meet the requirements of large-scale (web) infrastructures handling Big Data by providing elastic and horizontal scalability. Each NoSQL technology however is suited for specific use cases and data models. As a consequence, NoSQL adopters are faced with tremendous heterogeneity in terms of data models, database capabilities and application programming interfaces (APIs). Opting for a specific NoSQL database poses the immediate problem of vendor or technology lock-in. A solution has been proposed in the shape of Object-NoSQL Database Mappers (ONDMs), which provide a uniform abstraction interface for different NoSQL technologies.

Such ONDMs however come at a cost of increased performance overhead, which may have a significant economic impact, especially in large distributed setups involving massive volumes of data.

In this paper, we present a benchmark study quantifying and comparing the performance overhead introduced by Object-NoSQL Database Mappers, for create, read, update and search operations. Our benchmarks involve five of the most promising and industry-ready ONDMs: Impetus Kundera, Apache Gora, EclipseLink, DataNucleus and Hibernate OGM, and are executed both on a single node and a 9-node cluster setup.

Our main findings are summarised as follows: (i) the introduced overhead is substantial for database operations in-memory, however on-disk operations and high network latency result in a negligible overhead, (ii) we found fundamental mismatches between standardised ONDM APIs and the technical capabilities of the NoSQL database, (iii) search performance overhead increases linearly with the number of results, (iv) DataNucleus and Hibernate OGM's search overhead is exceptionally high in comparison to the other ONDMs.

Keywords: Object-NoSQL Database Mappers, Performance evaluation, Performance overhead, MongoDB

1 Introduction

Online systems have evolved into the large-scale web and mobile applications we see today, such as Facebook and Twitter. These systems face a new set of problems when working with a large number of concurrent users and massive data sets. Traditionally, Internet applications are supported by a relational database management system (RDBMS). However, relational databases have shown key limitations in horizontal and elastic scalability [1–3]. Additionally, enterprises employing RDBMS in a distributed setup often come at a high licensing cost, and per CPU charge scheme, which makes scaling over multiple machines an expensive endeavour.

Many large Internet companies such as Facebook, Google, LinkedIn and Amazon identified these limitations [1, 4–6] and in-house alternatives were developed, which were later called non-relational or NoSQL databases. These provide support for elastic and horizontal scalability by relaxing the traditional consistency requirements (the ACID properties of database transactions), and offering a simplified set of operations [3, 7, 8]. Each NoSQL database is tailored for a specific use case and data model, and distinction is for example commonly made between column stores, document stores, graph stores, etc. [9].

*Correspondence: vincent.reniers@cs.kuleuven.be
Department of Computer Science, KU Leuven, Celestijnenlaan 200A, B-3001 Heverlee, Belgium

This is a deviation from the traditional "one-size-fits-all" paradigm of RDBMS [2], and leads to more diversity and heterogeneity in database technology. Due to their specific nature and their increased adoption, there has been a steep rise in the creation of new NoSQL databases. In 2009, there were around 50 NoSQL databases [10], whereas today we see over 200 different NoSQL technologies [11]. As a consequence, there is currently large heterogeneity in terms of interface, data model, architecture and even terminology across NoSQL databases [7, 12]. Picking a specific NoSQL database introduces the risk of vendor or technology lock-in, as the application code has to be written exclusively to its interface [7, 13]. Vendor lock-in hinders future database migrations, which in the still recent and volatile state of NoSQL is undesirable, and additionally makes the creation of hybrid and cross-technology or cross-provider storage configurations [14] more challenging.

Fortunately, a solution has been proposed in the shape of Object-NoSQL Database Mappers (ONDM) [7, 12, 13]. ONDMs provide a uniform interface and standardised data model for different NoSQL databases or even relational databases. Even multiple databases can be used interchangeably, a characteristic called as *polyglot* or *cross-database* persistence [13, 15]. These systems support translating a common data model and operations to the native database driver. Despite these benefits, several concerns come to mind with the adoption of such middleware, and the main drawback would be the additional performance overhead associated with mapping objects and translating APIs. The performance impact potentially has serious economic consequences as NoSQL databases tend to run in large cluster environments and involve massive volumes of data. As such, even the smallest increase in performance overhead on a per-object basis can have a significant economic cost.

In this paper, we present the results of an extensive and systematic study in which we benchmark the performance overhead of five different open-source Java-based ONDMs: Impetus Kundera [16], EclipseLink [17], Apache Gora [18], DataNucleus [19] and Hibernate OGM [20]. These were selected on the basis of industry relevance, rate of ongoing development activity and comparability. We benchmarked the main operations of write/insert, read, update and a set of six distinct search queries on MongoDB. MongoDB is currently one of the most widespread adopted, and mature NoSQL document databases, in addition it is the only mutually supported database by all five ONDMs. The benchmarks presented in this paper are obtained in a single-node MongoDB setup and in a distributed MongoDB cluster consisting of nine nodes.

The main contribution of this paper is that it quantifies the performance cost associated with ONDM adoption, as such allowing practitioners and potential adopters to make informed trade-off decisions. In turn, our results inform ONDM technology providers and vendors about potential performance issues, allowing them to improve their offerings where necessary. In addition, this is to our knowledge the first study that involves an in-depth performance overhead comparison for search operations. We specifically focus on six distinct search queries of varying complexity.

In addition, the study is a partial replica study of an earlier performance study [21], which benchmarked three existing frameworks. We partially confirm the previous findings, yet in turn strengthen this study by: (i) adopting an improved measurement methodology, with the use of Yahoo!'s Cloud Serving Benchmark (YCSB) [3] —an established benchmark for NoSQL systems – and (ii) focusing on an updated set of promising ONDMs.

Our main findings first and foremost confirm that current ONMDs do introduce an additional performance overhead that may be considered substantial. As these ONDMs follow a similar design, the introduced overhead is roughly comparable: respectively the write, read and update overhead ranges between $[4-14\%]$, $[4-21\%]$ and $[60-194\%]$ (on a cluster setup). The overhead on update performance is significant due to *interface mismatches*, i.e. situations in which discrepancies between the uniform API and the NoSQL database capabilities negatively impact performance.

Regarding search, we found that query performance overhead can become substantial, especially for search queries involving many results, and secondly, that DataNucleus and Hibernate OGM's search overhead is exceptionally high in comparison to the other ONDMs.

The remainder of this paper is structured as follows: Section 2 discusses the current state and background of Object-NoSQL Database Mappers. Section 3 states the research questions of our study and Section 4 discusses the experimental setup and motivates the selection of ONDMs. Section 5 subsequently presents the results of our performance evaluation on write, read, and update operations, whereas Section 6 presents the performance results of search operations. Section 7 discusses the overall results, whereas Section 8 connects and contrasts our work to related studies. Finally, Section 9 concludes the paper and discusses our future work.

2 Object-NoSQL Database Mappers

This section provides an overview of the current state of Object-NoSQL Database Mappers (ONDMs) and motivates their relevance in the context of NoSQL technology.

2.1 Object-mapping frameworks for NoSQL

In general, object mapping frameworks convert in-memory data objects into database structures (e.g.

database rows) before persisting these objects in the database. In addition, such frameworks commonly provide a uniform, technology-independent programming interface and as such enable decoupling the application from database specifics, facilitating co-evolution of the application and the database, and supporting the migration towards other databases.

In the context of relational databases, such frameworks are commonly referred to as "Object-Relational Mapping" (ORM) tools [22], and these tools are used extensively in practice. In a NoSQL context, these frameworks are referred to as "Object-NoSQL Database Mapping" (ONDM) tools [12] or "Object-NoSQL Mapping (ONM)" tools [23].

In the context of NoSQL databases, data mapping frameworks are highly compelling because of the increased risk of vendor lock-in associated to NoSQL technology: without such platforms, the application has to be written for each specific NoSQL database and due to the heterogeneity in technology, programming interface and data model [7, 13], later migration becomes difficult. As shown in an earlier study, the use of ONDMs simplifies porting an application to another NoSQL significantly [21].

An additional benefit is the support for multiple databases, commonly referred to as database interoperability or cross-database and polyglot persistence [13, 15]. Cross-database persistence facilitates the use of multiple NoSQL technologies, each potentially optimised for specific requirements such as fast read or write performance. For example, static data such as logs can be stored in a database that provides very fast write performance, while cached data can be stored in an in-memory key-value database. Implementing such scenarios without an object-database mapper comes at the cost of increased application complexity.

However, ONDM technology only emerged fairly recently, and its adoption in industry is rather modest. Table 1 outlines the benefits and disadvantages of using ONDM middleware. The main argument against the adoption of ONDMs is the additional performance overhead. The study presented in this paper focuses on quantifying this overhead. In the following section, we outline the current state of ONDM middleware.

2.2 Current state of ONDMs

In this paper, we focus on object-database mappers that support application portability over multiple NoSQL databases. Examples are Hibernate OGM [20], EclipseLink [17], Impetus Kundera [16] and Apache Gora [18].

Table 2 provides an overview of the main features of several ONDMs such as: application programming interfaces (APIs), support for query languages and database support.

The API is the predominant characteristic as it determines the used data model and the features that are made accessible to application developers. A number of standardised persistence interfaces exist, such as the Java Persistence API (JPA) [24], Java Data Objects (JDO) [25] and the NPersistence API [26] for .NET. Some products such as Apache Gora [18] or offer custom, non-standardised development APIs.

Many of the currently-existing ODNMs (for Java) implement JPA. Examples are EclipseLink [17], DataNucleus [19] and Impetus Kundera [16]. Some of these products support multiple interfaces. For example, DataNucleus supports JPA, JDO and REST. JPA relies extensively on annotations. Classes and attributes are annotated to indicate that their instances should be persisted to a database. The annotations can cover aspects such as the relationships, actual column name, lazy fetching of objects, predefined query statements and embedding of entities.

Associated with JPA is its uniform query language called the Java Persistence Query Language (JPQL) [24]. It is a portable query language which works regardless of the underlying database. JPQL defines queries with complex search expressions on entities, including their relationships [24].

The uniform interface (e.g. JPA) and query language (e.g. JPQL) allow the user to abstract his/her application software from the specific database. However, this abstraction comes at a performance overhead cost, which stems from translating operations and data objects to the intended native operations and data structures and vice versa. For example, on write, the object is translated to the intended data structure of the underlying NoSQL database, while on read, the query operation is translated to the native query. Once the result is retrieved, the retrieved data structure is converted back into an object.

Table 1 Advantages and disadvantages of adopting ONDM middleware

Advantages	Disadvantages
Unified interface, query language and data model for multiple databases	Performance overhead incurred from translating the uniform interface and data model to its native counterparts
Increased application maintainability	
Cross-database persistence and database portability	Potential loss of database-specific features due to the abstraction level of the ONDM
Third-party functionality (e.g. caching)	

Table 2 Features and database support for the evaluated ONDMs

	Hibernate OGM	Kundera	Apache Gora	EclipseLink	DataNucleus
Evaluated Version	4.1.1 Final	2.15	0.6	2.5.2	5.0.0.M5
Interface	JPA	JPA, REST	Gora API	JPA	JPA, JDO, REST
Query Languages	JPQL, Native Queries	JPQL, Native Queries	Query interface	JPQL, Expressions, Native Queries	JPQL, JDOQL, Native Queries
RDBMS	×	✓	×	✓	✓
NoSQL Databases	*MongoDB*, Neo4j, Ehcache, CouchDB, Infinispan	*MongoDB*, Neo4j, CouchDB, Cassandra, ElasticSearch, HBase, Redis, Oracle NoSQL	*MongoDB*, HBase, Cassandra, Apache Solr, Apache Accumulo	*MongoDB*, JMS, XML, Oracle AQ, Oracle NoSQL,	*MongoDB*, HBase, Cassandra, Neo4j, JSON, XML, Amazon S3, GoogleStorage, NeoDatis

Database support for such mapping and translation operations varies widely. For example, EclipseLink is a mature ORM framework which has introduced NoSQL support only gradually over time, and it currently only supports Oracle NoSQL and MongoDB. While Kundera was intended specifically for NoSQL databases, it now also provides RDBMS support by using Hibernate ORM. Despite the heterogeneity between RDBMS and NoSQL, a combination of both can be used.

The following section introduces our main research questions, upon which we have built this benchmark study.

3 Research questions

Our study is tailored to address the following research questions:

RQ1 What is the overhead (absolute and relative) of a write, read and update operation in the selected ONDMs?

RQ2 What is the significance of the performance overhead in a realistic database deployment?

RQ3 What is the impact of the development API on the performance overhead?

RQ4 How does the performance overhead of a JPQL search query (search on primary key) compare to that of the JPA read operation (find on primary key)?

RQ5 What is the performance overhead of JPQL query translation, and does the nature/complexity of the query play a role?

Expectations and initial hypotheses. We summarise our expectations and up-front hypotheses below:

- **RQ1:** Although earlier studies [21, 23] have yielded mixed results, in general, the performance overhead has been shown to be rather substantial: ranging between 10 and 70% depending on the operation for a single-node setup. DataNucleus in particular is shown to have tremendous overhead [23]. We expect to confirm such results and thus increase confidence in these findings.
- **RQ2:** ONDMs are by design independent of the underlying database, and therefore, we expect the absolute overhead not to be affected by the setup or the complexity of the database itself. As a consequence, we expect the absolute overhead to potentially more significant (i.e. a higher relative overhead) for low-latency setups (e.g. a single node setup or an in-memory database), in comparison to setups featuring more network latency or disk I/O (e.g. a database cluster or a disk-intensive setup).
- **RQ3:** We expect to find that the programming interface does have a certain impact on performance. For example, the JPA standard relies heavily on code annotations, we expect the extensive use of reflection on these objects and their annotations within the ONDM middleware to substantially contribute to the overall performance overhead.
- **RQ4:** This is in fact an extension to **RQ3**, focusing on which development API incurs the highest performance overhead. On the one hand, JPA is costly due to its reliance on annotation-based reflection, while on the other hand, query translation can become costly as well. To our knowledge, this is the first benchmark study directly comparing the JPA and JPQL performance overhead over NoSQL search queries.
- **RQ5:** We expect complex queries to be more costly in query translation. Additionally, queries retrieving multiple results should have increased overhead as each result has to be mapped into an object.

The following section presents the design and setup of our benchmarks that are tailored to provide answers to the above questions.

4 Benchmark setup

This section discusses the main design decisions involved in the setup of our benchmark study. Section 4.1 first

discusses the overall architecture of an ONDM framework, and then Section 4.2 discusses the measurement methodology for the performance overhead. Section 4.3 subsequently motivates our selection of Object-NoSQL Database Mapping (ONDM) platforms for this study, whereas Section 4.4 elaborates further on the benchmarks we have adopted and extended for our study. Next, Section 4.5 discusses the different deployment configurations in which we have executed these benchmarks. Finally, Section 4.6 summarises how our study is tailored to provide answers to the research questions introduced in the previous section.

4.1 ONDM Framework architecture

The left-hand side of Fig. 1 depicts the common architecture of Object-NoSQL Database Mappers (ONDMs) which is layered. As shown at the top of Fig. 1, an ONDM platform supports a `Uniform Data Model` in the application space. In the Java Persistence API (JPA) for example, these are the annotated classes. In Apache Gora however, mapping classes are generated from user specifications. An ONDM provides a `Uniform Interface` based on the `Uniform Data Model`. The `Middleware Engine` implements the operations of the `Uniform Interface` and delegates these operations to the correct `Database Mapper`.

The `Database Mapper` is a pluggable module that implements the native `Database Driver`'s API. Different `Database Mapper` modules are created for different NoSQL databases. The `Database Mapper` converts the uniform data object to the native data structure, and calls the corresponding native operation(s). The `Database Driver` executes these native operations and handles all communication with the database.

The right hand side of Fig. 1 illustrates the situation in which no ONDM framework is employed, and the application directly uses the native client API to communicate with the database.

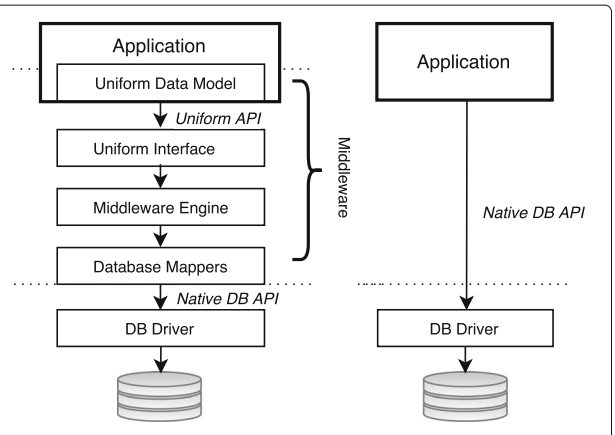

Fig. 1 Generic architecture of an Object-NoSQL Database Mapper (*left*), in comparison to a native client (*right*)

Comparing both alternatives in Fig. 1 clearly illustrates the cost of object mapping as a key contributor to the performance overhead introduced by ONDM platforms. Both write requests (which involve translating in-memory objects or API calls to native API calls) and read requests or search queries (which involve translating database objects to application objects) rely extensively on database mapping. Our benchmark study, therefore, focuses on measuring this additional performance overhead.

In addition, Fig. 1 clearly shows that an ONDM is designed to be maximally technology-agnostic: other than the `Database Mapper` which makes abstraction of a specific database technology, the inner workings of the ONDM do not take the specifics of the selected database technology into account.

4.2 Measurement methodology

In order to measure the overhead of ONDMs, we first measure t_{ONDM}, the total time it takes to perform a database operation (read, write, update, search), which is the sum of time spent by the ONDM components depicted on the left-hand side of Fig. 1.

In addition, we measure t_{DB}, the total time it takes to execute the exact same database operations using the native client API (right-hand side of Fig. 1). By subtracting both measurements, we can characterise the performance overhead introduced by the ODNM framework as $t_{Overhead} = t_{ONDM} - t_{DB}$. This is exactly the additional overhead incurred by deciding to adopt an ONDM framework instead of developing against the native client API.

To maintain comparability between different ODNMs, we must: (i) select a specific database and database version that is supported by the selected ONDM frameworks (our baseline for comparison), (ii) ensure that each ONDM framework uses the same database driver to communicate with the NoSQL database, (iii) run the exact same benchmarks in our different setups. These decisions are explained in the following sections.

4.3 ODNM selection

Our benchmark study includes the following five ONDMs: EclipseLink [17], Hibernate OGM [20], Impetus Kundera [16], DataNucleus [19] and Apache Gora [18]. Table 2 lists these ONDMs and summarises their main characteristics and features.

As mentioned above, to maintain comparability of our benchmark results, it is imperative to ensure that the selected ONDMs employ the exact same NoSQL database, and database driver version as our baseline. Driven by Table 2, we have selected MongoDB version 2.6 as the main baseline for comparison. In contrast to other

NoSQL technologies such as Cassandra for which many alternative client APIs and drivers are available, MongoDB provides only a single Java driver which is used by all of the selected frameworks. Furthermore, MongoDB can be used in various deployment configurations such as a single node or cluster setup, which will allow us to address **RQ2**.

In addition to MongoDB support as the primary selection criterion, we have also taken into account other comparability and industry relevance criteria: (i) JPA support, (ii) search support via JPQL, (iii) maturity and level of ongoing development activity. For example, we have deliberately excluded frameworks such as KO3-NoSQL [27] as their development seems to have been discontinued.

Although Apache Gora [18] is not JPA-compliant, it is included for the purpose of exploring the potential impact of the development API on the performance overhead introduced by these systems (**RQ3**).

4.4 Benchmark design

Our benchmarks are implemented and executed on top of the Yahoo! Cloud Serving Benchmark (YCSB) [3], an established benchmark framework initially developed to evaluate the performance of NoSQL databases. YCSB provides a number of facilities to accurately measure and control the benchmark execution of various workloads on NoSQL platforms.

Read, write, update. YCSB comes with a number of predefined workloads and is extensible, in the sense that different database client implementations can be added (by implementing the `com.yahoo.ycsb.DB` interface, which requires implementations for read, update, insert and delete (CRUD) operations on primary key).

Our implementation provides such extensions for the selected ONDMs (Hibernate OGM, DataNucleus EclipseLink, Kundera and Apache Gora). Especially the implementations for the JPA-compliant ONDMs are highly similar. To avoid skewing the results and to ensure comparability of the results, we did not make use of any performance optimization strategies offered by the ONDMs, such as caching, native queries and batch operations.

Furthermore, since implementations for NoSQL databases were already existing, we simply reused the client implementation for MongoDB for obtaining our baseline measurements.

Search. YCSB does not support benchmarking search queries out of the box. Therefore, we have defined a set of 6 read queries, which we execute on each platform in YCSB. These queries differ in both complexity and number of results. In support of these benchmarks, we populate our existing objects with more realistic values such as `firstName` and `lastName`, instead of YCSB's default behavior which involves generating lenghty strings of random characters.

Note that we do not benchmark query performance for Apache Gora, since it has no support for JPQL and lacks support for basic query operators such as `AND`, `OR`[1].

4.5 Deployment setup

To address **RQ2** and assess the impact of the database deployment configuration on the performance overhead introduced by ONDMs, we have executed our benchmarks over different deployment configurations. Figure 2 depicts these different configurations graphically. The client node labeled `YCSB Benchmark` runs the ONDM framework or the native driver which are driven by the YCSB benchmarks discussed above.

The single-node setup (cf. Fig. 2a) involves two commodity machines, one executing the YCSB benchmark, and the other hosting a single MongoDB database instance.

The MongoDB cluster (cf. Fig. 2b) consists of a single router server, 3 configuration servers and 5 database shards. Each database is sharded and all of the inserted entities in each database are load balanced across all 5 database shards without replication.

Each node consists of a Dell Optiplex 755 (Intel® Core™ 2 Duo E6850 3.00GHz, 4GB DDR2, 250GB hard disk). In both cases, the benchmarks were executed in a local lab setting, and the average network latency between nodes in our lab setup is quite low: around 135μs. As

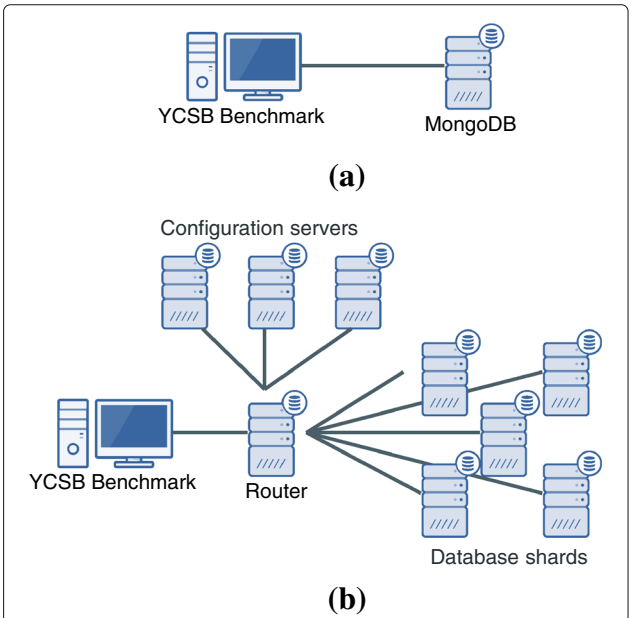

Fig. 2 Deployment setups: **a** single-node setup and **b** 9-node cluster

a consequence, our calculations of the relative overhead often represent the absolute worst case.

4.6 Setup: research questions

Below, we summarise how we address the individual research questions introduced in Section 3:

- **RQ1: Create, read, update.** We answer **RQ1** by running the benchmarks discussed above for the create, read and update operations. Our benchmarks are sequential: in the *load phase*, 20 million entities (20*GB*) are written to the database. In the *transaction phase*, the desired workload is executed on the data set (involving read and update). The inserted entity is a single object.
- **RQ2: Significance of performance overhead.** To put the absolute performance overhead measurements into perspective, we have executed our benchmarks in two different environments: (i) a remote single-node MongoDB instance, and (ii) a 9-node MongoDB cluster. These concrete setups are depicted in Fig. 2. In both cases, the actual execution of the benchmark is done on a separate machine to avoid CPU contention. The inserted data size consumes the entire memory pool of the single node and cluster shards. Read requests are not always able to find the intended record in-memory, resulting in lookup on disk. Based on the two types of responses we determine the general impact of ONDMs on overhead for deployments of varying data set sizes and memory resources.
- **RQ3: Impact of development API.** By comparing the results for the JPA middleware (Kundera, Hibernate ORM, DataNucleus and EclipseLink) to the results for Apache Gora (which offers custom, non-JPA compliant developer APIs), we can at least exploratively assess the potential performance impact of the interface.
- **RQ4: JPA vs JPQL.** To answer RQ5, we compare the basic JPA *find on primary key* (read lookup) to a JPQL *query on primary key*. By comparing both, we can assess the extra overhead cost of JPQL query translation.
- **RQ5: Search query performance overhead.** We have benchmarked queries on secondary indices in increasing order of query complexity for the ONDMs and compare the results to the benchmarks of the native MongoDB client API.

The next two sections present and discuss our findings in relation to these five research questions.

5 Write, read and update performance results

This section presents the results of our benchmarks that provide answers to questions **RQ1-3**. Research questions **RQ4-5** regarding search performance are discussed in Section 6.

The next sections first determine the overhead introduced by the selected ONDMs on the three operations (write, read, and update) in the context of the single remote node setup. In order to understand how the ONDMs introduce overhead, the default behaviour of MongoDB (our baseline for comparison) must be taken into account, which we discuss in the next Section 5.1.

5.1 Database behaviour

In our benchmarks, twenty million records (which corresponds to roughly 20GB) are inserted into the single node MongoDB database. Considering the machine only has 4GB RAM, it is clear that not all of the records will fit in-memory. As a consequence, read operations will read a record from memory around 5% of the time, but mainly require disk I/O. In-memory operations are, on average, 30 times as fast as operations requiring disk I/O. Similarly, the update operations will only be able to update a subset of objects in-memory. This, however, does not apply to the write operation: on write, the database regularly flushes records to disk, which also influences the baseline. Figure 3 shows the distribution in latency for each type of operation. We can clearly identify a bimodal distribution for read and update operations. Write operations are normally distributed, however skewed to the right, as expected.

The aim of this study is to identify the overhead introduced by ONDMs. However, the variance on latency for objects on-disk is quite high ($\pm 25ms$) and in this case, the behaviour of the ONDM frameworks may no longer be the contributing factor determining the overhead. Therefore, we have analysed the separate distributions of read and update. To alleviate this, we compare both data sets (in-memory versus on-disk) separately.

5.2 RQ1 Impact on write, read and update performance on a single node

Table 3 shows the overhead for write, read and update operations. Read and update operations are divided according to the overhead for objects in-memory and on-disk. We first discuss the results for operations in-memory. The write and read overhead of ONDMs ranges respectively between [9.9%, 36.5%] and [6.7%, 42.2%] and as such may be considered significant. However, the update operation is considerably slower and introduces twice as much latency for a single update operation in comparison to the native MongoDB driver[2]. The main reason for this is that update operations in the ODNMs frameworks first perform a read operation before actually updating a certain object. This is in contrast to the native database's capabilities: for example MongoDB can update records without requiring a read. Surprisingly

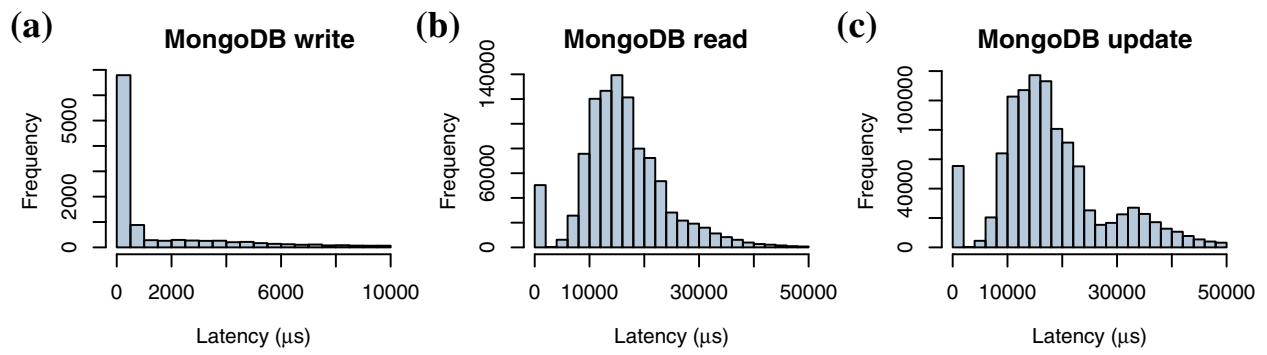

Fig. 3 MongoDB latency distribution for **a** write, **b** read and **c** update operations on a single node

enough, each of the observed frameworks require a read before update, resulting in the addition of read latency on update and thus significant overhead. Moreover, DataNucleus executes the read again, even though the object provided on update is already read, thus executing a read twice. This is a result of DataNucleus its mechanisms to ensure consistency, and local objects are verified against the database. The requirement of read on update in the ONDMs is a clear mismatch between the uniform interface and the native database's capabilities.

While operations on in-memory data structures show consistent overhead results, this is not the case for operations which trigger on-disk lookup. It may seem that the ONDM frameworks in some cases outperform the native database driver, but this is mainly due to the variance of database latency. The ordering in performance is not preserved for on-disk operations, and Kundera in particular experienced a higher latency. Considering the small overhead of around $[15\mu s, 300\mu s]$ which ONDMs introduce for operations in-memory, this is only a minimal contributor in the general time for on-disk operations. For example, MongoDB takes on average $15.9ms \pm 5.2ms$ for read on-disk. This is an increase in latency of 2 to 3 orders of magnitude. In other words, the relative overhead introduced by ONDMs is insignificant, when data needs to be searched for on-disk.

5.3 RQ2: Impact of the database topology

As shown for a single remote node, the overhead on write, read or update is significant for in-memory data. In case of the cluster, we expect the absolute overhead to be comparable to the single-node setup. Table 4 shows the results for write, read and update. As shown, the relative overhead percentages are substantially smaller in comparison to the single node. EclipseLink has only a minor write and read overhead of respectively 2.5 and 3.6%, which can be explained by considering that the absolute overhead remains more or less constant, while the baseline latency does increase. For example, EclipseLink's absolute read overhead is $15\mu s$ for the single node, and identically $15\mu s$ on the cluster. However, the write overhead decreases from $43\mu s$ to $29s$. This is attributed to the fact that MongoDB experienced more outliers, as its standard deviation for write is $12\mu s$ higher. The behaviour of each run is always slightly different, therefore the standard deviation, and thus behaviour of the database must be taken into account when interpreting these results. The ideal case is read in-memory, where the standard deviation is almost identical for all four frameworks and the native MongoDB driver. In general, the write and read overhead is still quite significant and ranges around [4%, 9%] for EclipseLink and Kundera, which are clearly more optimised than the other frameworks.

Table 3 Average latency and relative overhead for each platform on a single node

	Write		*Read in-memory*		*Read on-disk*		*Update in-memory*		*Update on-disk*	
Samples	$n = 20.000.000$		$n = 45.000$		$n = 750.000$		$n = 39.000$		$n = 750.000$	
Platform	Latency (μs)		Latency (μs)		Latency (ms)		Latency (μs)		Latency (ms)	
MongoDB	403 ± 110	-	217 ± 34	-	15.9 ± 5.2	-	298 ± 106	-	19.3 ± 9.1	-
EclipseLink	446 ± 105	10.8%	232 ± 41	6.7%	14.2 ± 5.0	−10.45%	579 ± 91	93.9%	16.9 ± 8.0	−12.0%
Kundera	442 ± 96	9.9%	256 ± 57	17.7%	17.1 ± 5.6	+8.0%	338 ± 56	13.3%	20.7 ± 9.8	+7.6%
Hibernate OGM	452 ± 72	12.3%	289 ± 42	32.8%	15.1 ± 6.5	−4.7%	620 ± 53	107.6%	16.8 ± 8.0	−12.8%
Apache Gora	495 ± 92	22.9%	282 ± 65	29.8%	14.5 ± 5.0	−8.5%	570 ± 108	91.0%	17.4 ± 8.2	−9.5%
DataNucleus	550 ± 76	36.5%	309 ± 64	42.2%	14.3 ± 5.0	−9.8%	882 ± 49	194.8%	17.7 ± 8.3	−8.0%

Table 4 Average latency and relative overhead for each platform on a cluster

	Write		*Read in memory*		*Read on disk*		*Update in memory*		*Update on disk*	
Samples	$n = 20.000.000$		$n = 360.000$		$n = 610.000$		$n = 300.000$		$n = 600.000$	
Platform	Latency (μs)		Latency (μs)		Latency (ms)		Latency (μs)		Latency (ms)	
MongoDB	694 ± 90	-	434 ± 26	-	11.7 ± 3.8	-	534 ± 122	-	14.6 ± 6.7	-
EclipseLink	723 ± 78	4.1%	449 ± 27	3.6%	11.0 ± 3.5	−5.4%	1052 ± 72	97.1%	15.2 ± 6.8	3.6%
Kundera	725 ± 79	4.4%	471 ± 27	8.7%	11.2 ± 3.5	−4.2%	858 ± 57	60.8%	15.9 ± 7.4	8.9%
Hibernate OGM	764 ± 68	10.1%	505 ± 28	16.4%	11.2 ± 3.6	−3.6%	1083 ± 67	102.9%	14.9 ± 6.6	2.1%
Apache Gora	791 ± 62	14.0%	506 ± 26	16.7%	11.5 ± 3.7	−1.2%	1034 ± 75	93.7%	15.7 ± 7.2	7.5%
DataNucleus	788 ± 54	13.6%	526 ± 27	21.2%	11.4 ± 3.6	−2.2%	1567 ± 40	193.8%	15.4 ± 6.5	5.5%

In case of update, the frameworks again introduce a substantial overhead, because they perform a read operation before an update. The cost of the additional read is even higher in the cluster context, considering that a single read takes around $434\mu s$.

When operations occur on-disk, it may seem that the frameworks outperform the baseline. Once again, this is attributed to the general behaviour of the MongoDB cluster. The standard deviation for reading on-disk for the baseline is, for example, 10% higher than the frameworks. The results of each workload execution may also vary due to records being load balanced at run-time. However, the cluster allows for a more precise determination of the overhead as there are more memory resources available, which in turn results in less variable database behaviour such as on-disk lookups. In addition, the write performance is less affected by the regular flush operation of a single node.

5.4 RQ3: Impact of the interface on performance

In contrast to the four JPA-compliant frameworks, we now include Apache Gora in our benchmarks, which offers a non-standardised, REST-based programming interface.

Tables 3 and 4 presents the average latency of Apache Gora for write, read and update on the two database topologies. Even though the interface and data model is quite different from JPA, the overhead is very similar.

Surprisingly enough, we do not see a large difference in update performance. As we actually observe the same behaviour for Apache Gora's update operation: Apache Gora's API specifies no explicit update operation, but instead uses the same write method `put(K key, T object)` for updating records. As a result, the object has to be read before updating. If an object has not yet been read and needs to be updated, it may be best to perform an update query instead.

5.5 Conclusions

In summary, the following conclusions are made from the results regarding **RQ1-3** about the performance of ONDMs:

- The write, read and update performance overhead can be considered significant. Overheads are observed between [4%, 14%] for write, [4%, 21%] for read and [60%, 194%] for update, on the cluster.
- The relative overhead becomes insignificant as the database latency increases. Examples are cases which trigger on-disk lookups or even when a higher network latency is present.
- Interface mismatches can exist between the uniform interface and the native database's capabilities which decrease performance.

The next section discusses our benchmark results regarding the performance overhead introduced by the uniform query language JPQL for the JPA ONDMs.

6 JPQL search performance

Contrary to the name, NoSQL databases often do feature a query language. In addition, ONDMs provide a uniform SQL-like query language on top of these heterogeneous languages. For example, JPA-based object-data mappers provide a standardised query language called JPQL. We have evaluated the performance of JPQL for the JPA-based platforms: EclipseLink, Kundera, DataNucleus and Hibernate OGM.

While it is clear that there can be quite some overhead attached to a create, read or update operation, the question **RQ4** still remains whether or not the JPQL search overhead is similar to JPA read. Section 6.1 therefore first compares two different ways to retrieve a single object: using a JPQL search query, or with a JPA lookup.

Then, Section 6.2 addresses **RQ5** by considering how the performance overhead of a JPQL query is affected by its nature and complexity.

6.1 RQ4: Single object search in JPA and JPQL

We compare a read for a single object using the JPA interface, to the same read in JPQL query notation. This allows us to determine the exact difference in read overhead between JPA and JPQL for **RQ4**.

In order to be able to compare the results from the earlier JPA read to the JPQL search on the same object for **RQ4**, we have re-evaluated the read performance by inserting 1 million entities (roughly 1GB of data). The data set is completely in-memory for the single-node and cluster setup, allowing for a consistent measurement of the performance overhead. More specifically, our benchmarks compare the performance overhead incurred by `Query A` (JPA code) with the overhead incurred by `Query B` (JPQL equivalent code) in Listing 1.

Listing 1 JPQL and JPA search on primary key

```
A) entityManager.find(Person.class, id);
B) SELECT * FROM Person p WHERE p.id = :id
```

Table 5 shows the average latency for a find in JPA and a search in JPQL for the same object. We can clearly see that in general, a query in JPQL comes at a higher performance overhead cost (**RQ4**). Additional observations:

- Kundera and EclipseLink both perform similarly in JPA and JPQL single entity search performance.
- Interestingly, DataNucleus and Hibernate OGM are drastically slower for JPQL queries.

In DataNucleus the additional JPQL overhead stems from the translation of the query to a generic expression tree, which is then translated to the native MongoDB query.

Additionally, DataNucleus makes use of a lazy query loading approach to avoid memory conflicts. As a result, it executes a second read call to verify if there are any records remaining.

Code inspection in Hibernate OGM revealed that this platform extensively re-uses components from the Hibernate ORM engine, which may result in additional overhead due to architectural legacy.

JPQL provides more advanced search functionality than JPA's single find on primary key. The next section discusses the performance benchmark results on a number of JPQL queries of increasing complexity.

Table 5 The average latency on single object search in JPA, JPQL, and MongoDB's native read

Native driver	1-node read		9-node read	
MongoDB	197μs		434μs	
Platform	JPA Latency	JPQL Latency	JPA s Latency	JPQL Latency
Kundera	243μs	285μs	478μs	520μs
EclipseLink	218μs	291μs	448μs	520μs
Hibernate OGM	270μs	1.804μs	521μs	2.098μs
DataNucleus	288μs	811μs	492μs	1.236μs

6.2 RQ5: Relation between the nature and complexity of the query and its overhead

This section discusses the results of our search benchmarks, and more specifically how the overhead of a search query is related to the complexity of the query for **RQ5**. Queries which retrieve multiple results incur more performance overhead, as all the results have to be mapped to objects.

The benchmarked search queries are presented in Listing 2. The respective queries are implemented in JPQL and executed in the context of all four ONDM platforms. Our baseline measurement is the equivalent MongoDB native query. The actual search arguments are chosen randomly at runtime by YCSB and are marked as `:variable`.

The queries are ordered according to the average results retrieved per query. `Query C` is a query on secondary indices using the `AND` operator and always retrieves a single result. By comparison to `Query B`, which retrieves a single object on the primary key, we can determine the impact of a more complex query text translation.

In contrast, `Queries D, E and F` retrieve respectively on average 1.35, 94 and 2864 objects. When we compare the performance of Queries D,E and F, we can assess what impact the amount of results have on the overhead. First, we evaluate the case where we retrieve a single result with a more complex query.

Listing 2 JPQL search queries

```
C) SELECT p FROM Person p WHERE
(p.email = :email) AND
(p.personalnumber = :personalnumber)

D) SELECT p FROM Person p WHERE
p.email = :email

E) SELECT p FROM Person p WHERE
(p.personalnumber < :upperBound) AND
(p.personalnumber > :lowerBound)

F) SELECT p FROM Person p WHERE
(p.firstName = :firstName) OR
(p.lastName = :lastName)
```

6.2.1 *JPQL search using the AND operator*

Table 6 presents the results for `Query C`, the JPQL search using AND on secondary indices. The query always returns a single object in our experiment. In comparison to the results from JPQL search on a primary key in Table 5, we observe an increase in baseline latency due to the use of secondary indices and the AND operator.

Additionally for the ONDMs, we observe an increase in read overhead for the more complex query on the single node for Kundera and Eclipselink. As it turns out EclipseLink is less efficient than Kundera in handling the more complex query. Furthermore, DataNucleus shows a higher increase in performance overhead, as the query is

Table 6 The average latency and overhead for `Query C`, which retrieves a single object

	1-node		*9-node*	
Native driver	Latency	Overhead	Latency	Overhead
MongoDB	281μs	-	621μs	-
Platform				
Kundera	408μs	127μs	743μs	122μs
EclipseLink	453μs	172μs	783μs	162μs
Hibernate OGM	590μs	309μs	921μs	301μs
DataNucleus	1.010μs	729μs	1.581μs	960μs

translated to a more complex expression tree first, and secondly due to the additional read from its lazy loading approach.

Surprisingly, Hibernate OGM's absolute overhead on the remote node is 309μs for the more complex `Query C`, while for the simple search (`Query B`) on primary key this was 1.607μs. Clearly, Hibernate OGM has some inefficiencies regarding its query performance.

6.2.2 JPQL search on a secondary index

`Query D` is a simple search on a secondary index of a person. The query retrieves on average 1.35 objects. Therefore, multiple records can be retrieved on search which have to be mapped into objects.

Table 7 shows the average latency and relative overhead of `Query D` for the four JPA platforms, as for the similar query implemented in MongoDB's native query language.

Again, we conclude that Kundera and EclipseLink are most efficient at handling the query.

6.2.3 JPQL search on a range of values

Table 8 shows the average latency for the JPQL search `Query E`. The performance overhead introduced by the ONDM platforms increases as on average 94 results have to be mapped into objects, and ranges between [453μs, 3.615μs] on the single node, and [473μs, 3.988μs] on the cluster.

Table 8 The average latency and overhead for `Query E`, which retrieves on average 94 objects

	1-node		*9-node*	
Native driver	Latency	Overhead	Latency	Overhead
MongoDB	943μs	-	1.901μs	-
Platform				
Kundera	1.396μs	453μs	2.374μs	473μs
EclipseLink	1.556μs	613μs	2.550μs	649μs
Hibernate OGM	4.558μs	3.615μs	5.889μs	3.988μs
DataNucleus	3.831μs	2.888μs	4.786μs	2.885μs

6.2.4 JPQL search using the OR operator

The average latency of `Query F` is presented in Table 9. Again, the performance overhead introduced by the ONDMs increases as this query involves retrieval of on average 2.864 records, to the range of [7.6ms, 56.6ms] and [10.2ms, 42ms] on the respective database topologies. These results allow us to highlight the specific object-mapping cost of each ONDM. Kundera seems to have significantly more efficient object-mapping than EclipseLink. The average overhead for each object retrieved ranges between [3μs, 17μs].

6.3 Search performance conclusion

In summary, several conclusions can be made from the results regarding **RQ4-5** about the query search performance of ONDMs:

- JPQL search on a primary key has a higher overhead than JPA's find for the same object (**RQ4**).
- The performance overhead of a JPQL query is closely related to the complexity of its translation and the amount of results retrieved (**RQ5**) and there are large differences between the ONDM in terms of the performance cost associated to search queries. Finally, the additional performance overhead per search result in general decreases for queries

Table 7 The average latency and overhead for `Query D`, which retrieves on average 1.35 objects

	1-node		*9-node*	
Native driver	Latency	Overhead	Latency	Overhead
MongoDB	250μs	-	576μs	-
Platform				
Kundera	347μs	97μs	677μs	100μs
EclipseLink	396μs	146μs	729μs	152μs
Hibernate OGM	553μs	304μs	883μs	306μs
DataNucleus	957μs	707μs	1.520μs	944μs

Table 9 The average latency and overhead for `Query F`, which retrieves on average 2.864 objects

	1-node		*9-node*	
Native driver	Latency	Overhead	Latency	Overhead
MongoDB	20.226μs	-	39.689μs	-
Platform				
Kundera	27.989μs	7.763μs	49.889μs	10.210μs
EclipseLink	33.640μs	13.414μs	56.059μs	16.370μs
Hibernate OGM	58.806μs	38.580μs	75.234μs	35.545μs
DataNucleus	77.093μs	56.587μs	81.628μs	41.993μs

involving large amounts of results, which motivates the use of JPQL for large result sets.

The next section discusses our benchmark results in further detail.

7 Discussion

First, Section 7.1 discusses the main threats to validity. Then, we provide a more in-depth discussion about some of the more surprising results of our benchmarks, more specifically about Kundera's fast update performance (Section 7.2), and the observed mismatch between standards such as JPA and NoSQL technology (Section 7.3). Finally, we discuss the significant overhead in search performance for Hibernate OGM and DataNucleus (Section 7.4).

7.1 Threats to validity

As with any benchmark study, a number of threats to validity apply. We outline the most notable topics below.

Internal validity We discuss a number of threats:

- **Throughput rate control.** A possible threat to validity is related to the method of measurement. Although YCSB allows specifying a fixed throughput rate, we did not make use of this function. Limiting the throughput ensures that no platform is constrained by the resources of the server or client. For example, the MongoDB native database driver can process create, read and update operations at a faster rate than the ONDMs, as shown. In such a case, the MongoDB driver may reach its threshold of maximum performance, as dictated by its deployment constraints. In contrast, the ONDMs work at a slower rate and are less likely to reach this threshold. Consequentially, the computing resources of the MongoDB node will not be as much of an issue. When applying throughput rate control, the possibility of reaching this threshold is excluded, and the average latency would be a more truthful depiction of the individual performance.
  To increase our confidence in the obtained results, we did run a smaller-scale additional evaluation in which we applied throughput rate control (limited to 10.000 operations per write, read and update) and did not notice any deviations from our earlier results. Furthermore, during our main experiment we have measured CPU usage, I/O wait time and memory usage. From these measurements[3] we gather that no cluster node used more than 10% CPU usage on average. Although the single-node database setup experienced the heaviest load, during workload execution, it was still idling 50% of the time. As such, we conclude that the MongoDB cluster and single-node setup did not reach their limits during our benchmarks.
- **Choice of the baseline.** In this study, we implicitly assume that the choice for MongoDB as the back-end database has no significant impact on the performance overhead of ONDMs, because we subtract the MongoDB latency in our performance overhead calculations. Furthermore, the database-specific mapper is a modularly pluggable module which is independent of the core middleware engine responsible for data mapping. Each database-specific implementation only varies in its implementation of these engine interfaces. These arguments lead us to believe that there will be minimal variation in overhead between NoSQL technologies. We can confirm this by referring to a previous study on the performance overhead [21], in which Cassandra and MongoDB were used as the baseline for comparison. The study shows similar relative overheads despite using a different database technology as the baseline for comparison.

External validity. There is a number of ways in which the results may deviate from realistic deployments of ONDM systems. Specifically, our benchmark is designed to quantify the worst-case performance overhead in a number of ways.

- **Entity relationships.** For simplicity, we chose to work with single entities containing no relationships. There are a number of different ways relationships can be persisted in NoSQL databases: denormalizing to a single entity, storing them as separate entities, etc. This may have a drastic effect on the object-data mapper's performance. A single entity containing no relationships allows us to monitor the overhead of each platform without unnecessary complexity. The performance overhead of an application that relies extensively on associations between entities may vary from the results obtained in our study.
- **Optimization strategies.** The studied ONDMs offer various caching strategies and transaction control mechanisms. EclipseLink even supports cross-application cache coordination, which may improve performance significantly. As already discussed in Section 4.4, to maximally ensure comparability of our results, we disabled these mechanisms in our benchmarks. In the case of Object-Relational Mappers (ORMs), the impact of performance optimizations has already been studied [28, 29]. A similar study can prove useful for ONDMs and should be considered future work.
- **Database deployment.** We have shown that although these frameworks introduce more or less a

constant absolute performance overhead, the significance of this performance overhead may depend highly on the nature and complexity of the overall database setup and the application case. For example, in the context of an in-memory database featuring a high-bandwidth and low-latency connection, the introduced overhead may be deemed significant. In contrast, general database deployments often read from disk and feature a higher network latency, and in such a context, the introduced overhead may be considered minimal or negligible.

It is therefore important to stress that for the above reasons, different and in many cases, better performance characteristics can be expected in realistic ONDM deployments.

7.2 Kundera's update performance

Looking at the update performance results of Impetus Kundera in Tables 3 and 4, one might conclude that Kundera significantly outperforms EclipseLink and Hibernate OGM when it comes to updating. However, upon closer inspection, we discovered that in the tested version of Kundera an implementation mistake was made.

More specifically, Kundera's implementation does not make use of the MongoDB property `WriteConcern.ACKNOWLEDGED`, which forces the client to actively wait until MongoDB acknowledges issued update requests (a default property in MongoDB since version 2.6 [30]). By not implementing this, Kundera's implementation gains an unfair advantage since some of the network latency is not included in the measurement.

We have reported this bug in the Kundera bug reporting system [31].

7.3 JPA-NoSQL interface mismatch

One remarkable result is the observation that update operations consistently introduce more performance overhead when compared to read or write operations (cf. Table 3). The main cause for this is that the JPA standard imposes that updates can only be done on *managed* entities, i.e. it forces the ONDM to read the object prior to update. This causes the update operation to be significantly costlier than a read operation[4]. As pointed out by [21], similar drawbacks are associated to delete operations (which were not benchmarked in this study).

In the context of Object-Relational Mappers (ORMs), this problem is commonly referred to as the *object-relational impedance mismatch* [32], and one may argue that in a NoSQL context, such mismatch problems may be more significant due to the technological heterogeneity among NoSQL systems and the wide range of features and data models supported in NoSQL.

Similar drawbacks apply to JPQL search operations, especially when there is a discrepancy between the native search capabilities and the features assumed by JPQL.

Future work is required to determine whether other existing standardised interfaces such as REST-based APIs, Java Data Objects (JDO) are better suited, and more in-depth research is required toward dedicated, NoSQL-specific abstraction interfaces that can further reduce the cost inherent to database abstraction.

7.4 JPQL search performance

When comparing the results of our query benchmarks (cf. Section 6), it becomes clear that the performance overhead results for DataNucleus and Hibernate OGM are drastically worse than those of EclipseLink and Impetus Kundera: in some cases, Hibernate OGM introduces up to 383% overhead whereas the overhead introduced by the other two ONDMs never exceeds 66%.

According to the Hibernate OGM Reference Guide [20], the search implementation is a direct port of the search implementation of Hibernate's Object-Relational Mapper (ORM). Architectural legacy could therefore be one potential explanation for these surprising results.

Similarly to Hibernate OGM, DataNucleus shows a more consistent overhead of around 300%. In this case, the overhead is mainly attributed to the fact that it executes additional and unnecessary reads. Furthermore, the queries are translated first into a more generic expression tree, and then to the native database query. Various optimization strategies are provided to cache these query compilations, which might in turn provide more optimal performance. However, it is clear that the compilation of queries to generic expression trees, independent of the data store, takes a toll on performance.

8 Related work

This section addresses three domains of related work: (i) performance studies on Object-relational Mapper (ORM) frameworks, (ii) academic prototypes of Object-NoSQL Database Mappers and (iii) (performance) studies on ONDMs.

8.1 Performance studies on ORM frameworks

In the Object-relational Mapper (ORM) space, several studies have evaluated the performance of ORM frameworks, mainly focused on a direct comparison between frameworks [33–37]. Performance studies were mainly conducted on Java-based ORM frameworks, however, some studies also evaluated ORM in .NET based frameworks [38, 39]. However, few studies actually focused on the overhead, but more on the differences between the frameworks. The benchmark studies of Sembera [40] and Kalotra [35] suggest that EclipseLink is slower than Hibernate. However, a study by ObjectDB actually lists

EclipseLink as faster than Hibernate OGM [41]. The methods used in each study differ and the results are not directly applicable to NoSQL. Since none of these studies quantify the exact overhead of these ORM systems, comparison to our results is difficult.

The studies by Van Zyl et al. [42] and Kopteff [34] compare the performance of Java ORM-frameworks to the performance of Object-databases. These studies evaluate whether object databases can be used instead of ORM tools and traditional relational databases, reducing the mapping cost.

Although executed in a different technological context (.NET), the studies of Gruca et al. [38] and Cvetkovic et al. [39] seem to indicate that there is less overhead associated to translating abstraction query languages (such as Entity SQL, LINQ or Hibernate HQL) to SQL in the context of relational databases, when compared to our results. The relatively high search overhead in our results is caused by the larger abstraction gap between NoSQL query interfaces and JPQL (which is a SQL-inspired query language by origin).

8.2 Academic prototypes

Our study focused mainly on Object-NoSQL Database Mappers (ONDMs) with a certain degree of maturity and industry-readiness. Apart from these systems, a number of academic prototypes exist that provide a uniform API for NoSQL data stores. This is a very wide range of systems, and not all of them perform object-data mapping. ODBAPI, presented by Sellami et al. [13], provides a unified REST API for relational and NoSQL data stores. Dharmasiri et al. [43] have researched a uniform query implementation for NoSQL. Atzeni et al. [7] and Cabibbo [12] have presented Object-NoSQL Database Mappers which employ object entities as the uniform data model. Cabibbo [12] is the first to coin the term "Object-NoSQL Datastore Mapper".

We have excluded such systems as most of these implementations are proof-of-concepts, and few of them are readily available.

8.3 Studies on ONDMs

Three existing studies have already performed an evaluation and comparison of Object-NoSQL Database Mappers. Wolf et al. [44] extended Hibernate, the ORM framework, to support RIAK, a NoSQL Key-Value data store. In support of this endeavour, they evaluated the performance and compared it with the performance of Hibernate ORM configured to use with MySQL. The study provides valuable insights as to how NoSQL technology can be integrated into object-relational mapping frameworks.

Störl et al. [23] conducted a comparison and performance evaluation of Object-NoSQL Database Mappers (ONDMs). However, the study does not quantify the overhead directly, making a comparison difficult. Moreover, these benchmarks were obtained on a single node, and as a consequence, the results may be affected by CPU contention. Highly surprising in their results is the read performance of DataNucleus, which is shown to be at least 40 times as slow EclipseLink. We only measured similar results when *entity enhancement* was left enabled at-runtime, which recompiles entity classes to a meta model on each read. As a result, this may indicate fundamental flaws in the study's measurement methodology.

Finally, our study is a replica study of an earlier performance study by Rafique et al. [21], and we confirm many of these results. Our study differs in the sense that: (i) we adopted an improved measurement methodology, providing more insight on the correlation between the overhead and the database's behaviour and setup. Secondly, (ii) we conducted our evaluation using YCSB (an established NoSQL benchmark), (iii) we focus on a more mature set of ONDMs which have less overhead, and finally (iv) we evaluated the performance impact of ONDMs over search operations.

9 Conclusions and future work

Object-NoSQL Database Mapper (ONDM) systems have large potential: firstly, they allow NoSQL adopters to make abstraction of heterogeneous storage technology by making source code independent of specific NoSQL client APIs, and enable them to port their applications relatively easy to different storage technologies. In addition, they are key enablers for novel trends such as federated storage systems in which the storage tier of the application is composed of a combination of different heterogeneous storage technologies, potentially even hosted by different providers (cross-cloud and federated storage solutions).

There are however a number of caveats, such as the potential loss of NoSQL-specific features (due to the mismatch between APIs), and most notably, the additional performance overhead introduced by ONDM systems. The performance benchmarks presented in this paper have quantified this overhead for a standardised NoSQL benchmark, the Yahoo! Cloud Serving Benchmark (YCSB), specifically for create, read and update, and most notably search operations. In addition, we have explored the effect of a number of dimensions on the overhead: the storage architecture deployment setup, the amount of operations involved and the impact of the development API on performance.

Future work however is necessary for a survey study or gap analysis on existing ORM and ONDM framework with support for NoSQL and its features, specifically in the context of e.g. security and cross-database persistence. Additionally, we identify the need for a NoSQL

search benchmark, as we have seen YCSB used for these purposes, although it is not supported by default. In addition, we aim to provide an extended empirical validation of our results on top of additional NoSQL platform(s).

The results obtained in this study inform potential adopters of ONDM technology about the cost associated to such systems, and provides some indications as to the maturity of these technologies. Especially in the area of search, we have observed large differences among ONDMs in terms of the performance cost.

This work fits in our ongoing research on policy-based middleware for multi-storage architectures in which these ONDMs represent a core layer.

Endnotes

[1] Furthermore, Apache Gora implements most query functionality based on client-side filtering, which can be assumed quite slow.

[2] The results indicate that this is however not the case for Kundera, which is attributable to an implementation mistake in Kundera's update mechanism (see Section 7.2)

[3] Our resource measurements indicate that factors such as I/O and CPU play a negligible role in the results. For example, the utilization of ONDM platforms required only limited additional CPU usage at the client side for read (Additional file 1).

[4] Kundera's update strategy is slightly different: the `merge( object )` update operation in Kundera reads the object only when it is unmanaged, whereas in the other platforms this is explicitly done by the developer. The solution in Kundera therefore avoids the cost of mapping the result of the read operation to an object.

Additional file

Additional file 1: CPU Metric. (TXT 2 kb)

Acknowledgements
This research is partially funded by the Research Fund KU Leuven (project GOA/14/003 - ADDIS) and the DeCoMAdS project, which is supported by VLAIO (government agency for Innovation by Science and Technology).

Availability of data and materials
The datasets supporting the conclusions are included within the article. The benchmark, which is an extension of YCSB, can be found at: https://github.com/vreniers/ONDM-Benchmarker The software is distributed under the Apache 2.0 license. The project is written in Java and is therefore platform independent.

Authors' contributions
VR conducted the main part of this research with guidance from AR, who has done earlier research in this domain. DVL supervised the research and contents of the paper, and WJ conducted a final supervision. All authors read and approved the final manuscript.

Authors' information
The authors are researchers of imec-DistriNet-KU Leuven at the Department of Computer Science, KU Leuven, 3001 Heverlee, Belgium.

Competing interests
The authors declare that they have no competing interests.

Received: 24 February 2016 Accepted: 2 December 2016
Published online: 05 January 2017

References

1. Băzăr C, Iosif CS, et al. The transition from rdbms to nosql. a comparative analysis of three popular non-relational solutions: Cassandra, mongodb and couchbase. Database Syst J. 2014;5(2):49–59.
2. Stonebraker M, Madden S, Abadi DJ, Harizopoulos S, Hachem N, Helland P. The end of an architectural era:(it's time for a complete rewrite). In: Proceedings of the 33rd International Conference on Very Large Data Bases. Vienna: VLDB Endowment; 2007. p. 1150–1160. http://dl.acm.org/citation.cfm?id=1325851.1325981.
3. Cooper BF, Silberstein A, Tam E, Ramakrishnan R, Sears R. Benchmarking cloud serving systems with YCSB. In: Proceedings of the 1st ACM symposium on Cloud computing - SoCC '10. Association for Computing Machinery (ACM); 2010. p. 143–154. doi:10.1145/1807128.1807152. http://dx.doi.org/10.1145/1807128.1807152.
4. Lakshman A, Malik P. Cassandra: a decentralized structured storage system. ACM SIGOPS Oper Syst Rev. 2010;44(2):35–40.
5. Chang F, Dean J, Ghemawat S, Hsieh WC, Wallach DA, Burrows M, Chandra T, Fikes A, Gruber RE. Bigtable: A distributed storage system for structured data. ACM Trans Comput Syst (TOCS). 2008;26(2):4.
6. DeCandia G, Hastorun D, Jampani M, Kakulapati G, Lakshman A, Pilchin A, Sivasubramanian S, Vosshall P, Vogels W. Dynamo. ACM SIGOPS Operating Systems Review. 2007;41(6):205–220. doi:10.1145/1323293.1294281. http://dx.doi.org/10.1145/1323293.1294281.
7. Atzeni P, Bugiotti F, Rossi L. Uniform access to nosql systems. Inform Syst. 2014;43:117–133.
8. Stonebraker M. Sql databases v. nosql databases. Commun ACM. 2010;53(4):10–11. doi:10.1145/1721654.1721659.
9. Cattell R. Scalable sql and nosql data stores. ACM SIGMOD Rec. 2011;39(4):12–27.
10. Stonebraker M. Stonebraker on nosql and enterprises. Commun ACM. 2011;54(8):10–11.
11. NoSQL databases. http://www.nosql-database.org. Accessed 22 Feb 2016.
12. Cabibbo L. Ondm: an object-nosql datastore mapper: Faculty of Engineering, Roma Tre University; 2013. Retrieved June 15th. http://cabibbo.dia.uniroma3.it/pub/ondm-demo-draft.pdf.
13. Sellami R, Bhiri S, Defude B. Odbapi: a unified rest API for relational and NoSQL data stores. In: 2014 IEEE International Congress on Big Data. IEEE; 2014. p. 653–660. doi:10.1109/bigdata.congress.2014.98. http://dx.doi.org/10.1109/bigdata.congress.2014.98.
14. Rafique A, Landuyt DV, Lagaisse B, Joosen W. Policy-driven data management middleware for multi-cloud storage in multi-tenant saas. In: 2015 IEEE/ACM 2nd International Symposium on Big Data Computing (BDC); 2015. p. 78–84. doi:10.1109/BDC.2015.39.
15. Fowler M. Polyglot Persistence. 2015. http://martinfowler.com/bliki/PolyglotPersistence.html. Accessed 22 Feb 2016.
16. Impetus: Kundera Documentation. https://github.com/impetus-opensource/Kundera/wiki. Accessed 28 May 2016.
17. Eclipselink: Understanding EclipseLink 2.6. 2016. https://www.eclipse.org/eclipselink/documentation/2.6/concepts/toc.htm. Accessed 27 May 2016.
18. Apache Gora: Apache Gora. http://gora.apache.org/. Accessed 28 May 2016.
19. DataNucleus: DataNucleus AccessPlatform. 2016. http://www.datanucleus.org/products/accessplatform_5_0/index.html. Accessed 28 May 2016.
20. Red Hat: Hibernate OGM Reference Guide. 2016. http://docs.jboss.org/hibernate/ogm/5.0/reference/en-US/pdf/hibernate_ogm_reference.pdf. Accessed 28-05-2016.
21. Rafique A, Landuyt DV, Lagaisse B, Joosen W. On the Performance Impact of Data Access Middleware for NoSQL Data Stores. IEEE Transactions on Cloud Computing. 2016;PP(99):1–1. doi:10.1109/TCC.2015.2511756.

22. Barnes JM. Object-relational mapping as a persistence mechanism for object-oriented applications: PhD thesis, Macalester College; 2007.
23. Störl U, Hauf T, Klettke M, Scherzinger S, Regensburg O. Schemaless nosql data stores-object-nosql mappers to the rescue? In: BTW; 2015. p. 579–599. http://www.informatik.uni-rostock.de/~meike/publications/stoerl_btw_2015.pdf.
24. Oracle Corporation: The Java EE6 Tutorial. 2016. http://docs.oracle.com/javaee/6/tutorial/doc/. Accessed 22 Feb 2016.
25. Apache JDO: Apache JDO. https://db.apache.org/jdo/. Accessed 22 Feb 2016.
26. NET Persistence API. http://www.npersistence.org/. Accessed 22 Feb 2016.
27. Curtis N. KO3-NoSQL. 2007. https://github.com/nichcurtis/KO3-NoSQL. Accessed 22 Feb 2016.
28. van Zyl P, Kourie DG, Coetzee L, Boake A. The influence of optimisations on the performance of an object relational mapping tool. 2009150–159. doi:10.1145/1632149.1632169.
29. Wu Q, Hu Y, Wang Y. Research on data persistence layer based on hibernate framework. 20101–4. doi:10.1109/IWISA.2010.5473662.
30. MongoDB: MongoDB Documentation. 2016. https://docs.mongodb.com/v2.6/. Accessed 22 Feb 2016.
31. Kundera bug regarding MongoDB's WriteConcern. https://github.com/impetus-opensource/Kundera/issues/830. Accessed 22 Feb 2016.
32. Ireland C, Bowers D, Newton M, Waugh K. A classification of object-relational impedance mismatch. In: Advances in Databases, Knowledge, and Data Applications, 2009. DBKDA '09. First International Conference On; 2009. p. 36–43. doi:10.1109/DBKDA.2009.11.
33. Higgins KR. An evaluation of the performance and database access strategies of java object-relational mapping frameworks. ProQuest Dissertations and Theses. 82. http://gradworks.umi.com/14/47/1447026.html.
34. Kopteff M. The Usage and Performance of Object Databases compared with ORM tools in a Java environment. Citeseer. 2008. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.205.8271&rank=1&q=kopteff&osm=&ossid=.
35. Kalotra M, Kaur K. Performance analysis of reusable software systems. 2014773–778. doi:10.1109/CONFLUENCE.2014.6949308.
36. Ghandeharizadeh S, Mutha A. An evaluation of the hibernate object-relational mapping for processing interactive social networking actions. 201464–70. doi:10.1145/2684200.2684285.
37. Yousaf H. Performance evaluation of java object-relational mapping tools. Georgia: University of Georgia; 2012.
38. Gruca A, Podsiadło P. Beyond databases, architectures, and structures: 10th international conference, bdas 2014, ustron, poland, may 27–30, 2014. proceedings. 201440–49. Chap. Performance Analysis of .NET Based Object–Relational Mapping Frameworks. doi:10.1007/978-3-319-06932-6_5.
39. Cvetković S, Janković D. Objects and databases: Third international conference, icoodb 2010, frankfurt/main, germany, september 28–30, 2010. proceedings. 2010147–158. Chap. A Comparative Study of the Features and Performance of ORM Tools in a .NET Environment. doi:10.1007/978-3-642-16092-9_14.
40. Šembera L. Comparison of jpa providers and issues with migration. Masarykova univerzita, Fakulta informatiky. 2012. http://is.muni.cz/th/365414/fi_m/.
41. JPA Performance Benchmark (JPAB). http://www.jpab.org/. Accessed 22 Feb 2016.
42. Van Zyl P, Kourie DG, Boake A. Comparing the performance of object databases and ORM tools. In: Proceedings of the 2006 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing couuntries - SAICSIT '06; 2006. p. 1–11. doi:10.1145/1216262.1216263.
43. Dharmasiri HML, Goonetillake MDJS. A federated approach on heterogeneous nosql data stores. 2013234–23. doi:10.1109/ICTer.2013.6761184.
44. Wolf F, Betz H, Gropengießer F, Sattler KU. Hibernating in the cloud-implementation and evaluation of object-nosql-mapping. Citeseer.