On the Performance Impact of Data Access Middleware for NoSQL Data Stores

A Study of the Trade-Off between Performance and Migration Cost

Ansar Rafique, Dimitri Van Landuyt, Bert Lagaisse, and Wouter Joosen

Abstract—The last few years have seen a drastic increase in the amount and the heterogeneity of NoSQL data stores. Consequently, exploration and comparison of these data stores have become difficult. Once chosen, it is hard to migrate to different data stores. Recently, a number of data access middleware platforms for NoSQL have emerged that provide access to different NoSQL data stores from standardized APIs. However, there are two key concerns related to: (i) the performance overhead introduced by these platforms, and (ii) the effort required to migrate between different data stores. In this paper, we present two complementary studies that provide answers to the above mentioned concerns for three of the most mature data access middleware platforms: Impetus Kundera, Playorm, and Spring Data. First, we evaluate the performance overhead introduced by these platforms for the CRUD operations. Second, we compare the cost of migration with and without these platforms. Our study shows that, despite their similarity in design, these platforms are still substantially different performance-wise. Both studies are complementary as they show the trade-off inherent in adopting a data access middleware platform for NoSQL: by allowing some performance overhead, the developer gain benefits in terms of portability and easy migration across heterogeneous data stores.

Index Terms—Data management middleware, abstraction APIs, performance evaluation, migration across heterogeneous NoSQL

1 Introduction

The cloud computing paradigm promises high availability, elastic scalability, and thus offers increased flexibility. The benefits of cloud computing are in terms of lower up-front maintenance cost with higher scalability and availability, and therefore over the last few years, many applications have been built in or migrated to cloud environments [24]. However, the storage and computational requirements of such applications have pushed centralized databases to their limits [29]. For example, most of the popular Internet-based services such as Amazon, Google, and Facebook rely on storing and processing massive amounts of data at a scale at which traditional Relational Database Management Systems (RDBMSs) fall short [35]. Therefore, to address the shortcomings of traditional RDBMSs in handling large volumes of data, a number of specialized solutions – so-called "NoSQL" or "cloud data management" systems – have emerged.

NoSQL, which stands for *not only SQL*, is an umbrella term that refers to a wide range of data stores in which ACID transaction constraints have been relaxed to allow improved horizontal scaling and performance. The NoSQL nomer combines a wide range of document databases, column family stores, key-value stores, and graph-based databases [24]. In the last couple of years, NoSQL data stores have become increasingly popular for data storage and management "in the cloud" [9].

Despite the appropriateness of NoSQL as cloud data management systems, there is, currently, a wide variety and heterogeneity among them. As stated by the CAP theorem [6], trade-offs have to be made between key concerns such as consistency and availability in case of a partition when designing a horizontally scalable system, and each NoSQL data store makes different choices when it comes to these trade-offs [47]. Also, there is a lack of standardization: NoSQL technologies differ in data models, topology (master/slave versus peer-to-peer), replication policy, and application programming interfaces (APIs) and therefore, there is no standardized query interface.

The explosive growth and heterogeneity of NoSQL data stores is problematic for organizations and application developers that are interested in the benefits of NoSQL, but lack the expertise or resources to compare different offerings in depth. Indeed, technology exploration is expensive as it involves a steep learning curve, and requires expert knowledge. Furthermore, building applications against the native APIs of a NoSQL data store introduces the significant risk of technology, vendor, or provider[1] lock-in: an application that has been developed for a specific NoSQL technology or provider requires significant effort to be migrated to another one, a migration that in many cases involves dealing with technology disruption. Finally, these systems evolve quickly, and the number of NoSQL data stores keeps

• *The authors are with iMinds-DistriNet (a research group within the Department of Computer Science), KU Leuven 3001, Leuven, Belgium. E-mail: {Ansar.Rafique, dimitri.vanlanduyt, bert.lagaisse, wouter.joosen}@cs.kuleuven.be, .*

Manuscript received 23 Apr. 2015; revised 29 Sept. 2015; accepted 4 Dec. 2015. Date of publication 23 Dec. 2015; date of current version 31 Aug. 2018.
Recommended for acceptance by R. Sitaraman.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TCC.2015.2511756

1. When the storage is used as-a-service.

growing. For example, in 2011 only 50 NoSQL solutions were available, while currently over 150 solutions exist [33]. Similarly, the development APIs evolve in parallel with NoSQL systems. For example, for a long period of time, the Thrift APIs have been the preferred choice for Cassandra [2], but these are now being replaced with the Datastax Java-driver [14], which is based on binary CQL protocol [13].

The problem of heterogeneity among NoSQL data stores has been recognized by both the industry [25], [26], [27], [40] and the research community [7], [39], [43], and this has given rise to a number of data access middleware platforms for NoSQL systems. These data access middleware platforms provide a uniform API for NoSQL solutions (many of them based on or inspired by the Java Persistence API (JPA)) and this middleware provides the translation of this uniform API into the native client APIs. The most notable examples are Impetus Kundera [27], Playorm [26], Hibernate OGM [25], Spring Data [40], and AppScale [43]. Although these middleware platforms are relatively new and evolve on a daily basis, they indeed provide a promising alternative for organizations that are interested in the benefits of NoSQL data stores. There is, however, still a limited understanding of the costs and trade-offs involved when adopting such data access middleware platforms.

We present a study of the trade-off between the performance overhead and the migration effort for heterogeneous NoSQL data stores. First, we run a series of performance benchmarks to investigate the significance of the performance overhead for the insert, read, update, and delete (CRUD) operations. Second, we perform a migration study comparing the cost of migration, in terms of the effort required (e.g., the development time, the impact on deployment) with and without using the selected data access middleware platforms. Both evaluations are applied in the context of a real application case and we compare three of the most mature open-source data access middleware platforms for NoSQL data stores: Impetus Kundera [27], Playorm [26], and Spring Data [40].

The contributions of this paper are twofold: (i) we provide an empirical comparison of alternative data access middleware platforms and show that these platforms are very different in terms of performance overhead and migration cost, and (ii) we provide insights in the essential trade-off between performance overhead and migration effort that is inherent to using a data access middleware, and we substantiate these insights with empirical data. To our knowledge, this paper reports the first study related to data access middleware platforms that combines both the performance evaluation and the cost of migration when used with heterogeneous NoSQL data stores in a real distributed environment.

The remainder of this paper is structured as follows: Section 2 discusses the current state of data access middleware platforms for NoSQL data stores, while Section 3 elaborates our motivation and further illustrates the problem statement of this paper with the realistic application case study. Section 4 presents the study of performance overhead while Section 5, presents the study of migration cost required across multiple NoSQL data stores. Section 6 connects and contrasts our work with other related research. Finally, Section 7 concludes this paper and indicates directions of future work.

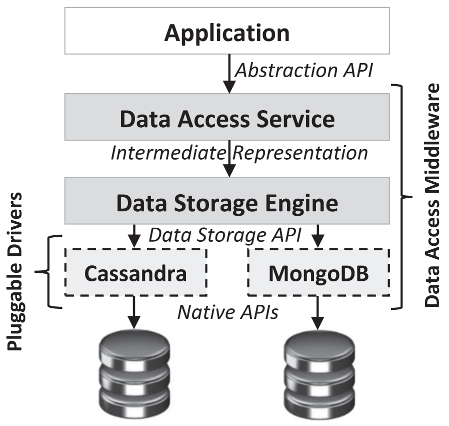


Fig. 1. Typical underlying architecture of data access middleware platforms.

2 Context: The Current State of Data Access Middleware Platforms for NoSQL

In this section, we first discuss the current state and the benefits of employing data access middleware platforms. Then, we elaborate on the typical underlying architecture of these platforms. Finally, we present the list of data access middleware platforms for NoSQL data stores and pick three middleware platforms for the study presented in this paper.

Use of data access middleware platforms. The recent advent of data access middleware for NoSQL is similar to the evolution of RDBMSs. Specifically, the current state of practice in the development of distributed systems relies heavily on the existence of data access middleware platforms and ORM tools for relational databases (e.g., Hibernate ORM, JDO, EclipseLink). These systems rely on standardized APIs and frameworks (e.g., JDBC, ODBC, JP-QL, JPA, etc.) that allow the developer to access databases programmatically via a set of well-chosen abstractions [28]. The main benefits of employing data access middleware platforms are: (a) they decouple the application from the database, improving application and database portability, (b) they help to make the system more understandable and hide some of the complexity of the underlying data storage systems, thereby reducing the time and effort to perform database operations, (c) they help developers to get rid of writing complex data stores related SQL statements and to experiment with different storage systems with minimal migration effort, and (d) they provide a uniform data query and retrieval facilities. For these reasons, they are often also called *abstraction layers*.

Typical architecture of data access middleware platforms. As shown in Fig. 1, applications are developed against a standardized, developer-friendly `Abstraction API`. The `Data Access Service` converts the incoming data to an intermediate representation and also implements middleware-specific features (e.g., caching to improve the read performance). The `Data Storage Engine` translates the intermediate representation into the `Data Storage API`, a uniform API for the different `Pluggable Drivers`. These `Pluggable Drivers` use the data store specific native client APIs to interact with each supported data store.

Data access middleware platforms for NoSQL data stores. Table 1 lists the most notable data access middleware

TABLE 1
Overview of Existing Data Access Middleware

Name	R/W	Advanced Search Requires	Supported Data Storage Systems
Impetus Kundera [27]	*JPA 2.0* + REST	Lucene, Elasticsearch	*Apache Cassandra*, *MongoDB*, Apache HBase, CouchDB, ElasticSearch, MySQL, Neo4j, Oracle NoSQL, Redis
Apache Gora [1]	Gora API	Lucene, Solr	*Apache Cassandra*, *MongoDB*, Apache Avro, Apache Hadoop, Apache HBase, Apache Solr, Hypertable, Voldemort
Data Nucleus Access Platform [12]	*JPA 2.1* + REST	-	*Apache Cassandra*, *MongoDB*, Amazon S3, Apache HBase, MySQL, Neo4j, Oracle, PostgreSQL
Playorm [26]	*JPA*-like	Scalable JQL(SJQL)	*Apache Cassandra*, *MongoDB*, Apache HBase
Spring Data [40]	*JPA* + REST	Lucene	*Apache Cassandra*, *MongoDB*, Apache Hadoop, Couchbase, DynamoDB, Elasticsearch, Gemfire, Neo4J, Redis, Solr
Hibernate OGM [25]	*JPA 2.0*	Hibernate Search	*MongoDB*, Ehcache, Infinispan, Neo4j
AppScale [43]	REST	-	*Apache Cassandra*, *MongoDB*, Apache HBase, Hypertable, MemcacheDB, MySQL, Voldemort
EclipseLink [20]	*JPA*	-	*MongoDB*, JMS, Oracle AQ, Oracle NoSQL, XML files

The platforms in bold are selected for our evaluation.

platforms for NoSQL data stores. The second column provides information about the `Abstraction API` (e.g., JPA, REST) provided by these platforms, showing that in a Java context, JPA has become the de-facto standard [41]. As shown in the third column, some data access middleware platforms rely on third-party libraries to implement advanced search operators (e.g., AND, OR, LIKE, IN) which are not natively supported. As an example, Kundera relies on Lucene to implement the LIKE operator in Cassandra [41].

Although they address similar goals, these platforms differ from each other in terms of (a) their support for different application technologies (e.g., AppScale [43] only supports Google App Engine (GAE) applications), (b) APIs and the abstractions provided by these middleware platforms, (c) advanced queries and the support for full-text search (e.g., Hibernate OGM [25] uses Hibernate Search for the full-text search whereas Kundera [27] uses Elasticsearch [21]), and (d) the supported data storage systems (see Table 1, fourth column): Kundera [27], Appscale [43], and Data Nucleus [12] support both relational SQL-based and NoSQL databases, whereas such support is not yet available in the other data access middleware platforms. It is important to note that these middleware platforms are under active development, and therefore, they constantly evolve and improve.

Data access middleware platforms selection. For the study presented in this paper, we picked three of most comparable data access middleware platforms: Impetus Kundera [27], Playorm [26], and Spring Data [40] (printed in bold in Table 1). These platforms are selected because they offer a JPA interface and support both Cassandra [2] and MongoDB [31].

3 Motivation and Problem Statement

The motivation for this paper is based on our experiences with a number of multi-tenant SaaS applications, obtained in the context of several research projects in collaboration with industry [10], [11], [18].

3.1 Application Case

The one such motivating case is a Log Management as a Service (LMaaS) application, a multi-tenant B2B cloud offering, that allows enterprises to maximize the value from their IT infrastructure and application logs. It provides services to monitor the entire infrastructure and perform complex analysis activities on the collected logs, e.g., detection of suspicious activity, performing log forensics, etc. Fig. 2 provides a schematic architectural overview of the LMaaS system. Integration with the on-premise infrastructure is accomplished by installing a Log Collector which aggregates the different log sources locally and communicates them to the LMaaS system. Scalable storage is a key enabler to realize the LMaaS service, i.e., scalable in terms of the sheer amount of data and the number of tenants (i.e., customer organizations) involved.

3.2 Problem Statement

Currently, most of the storage in the LMaaS system is realized with traditional relational database systems, but as outlined in Section 1, there are clear opportunities to adopt NoSQL technology to realize the LMaaS system. The selection of appropriate NoSQL technology is essential to achieve the desired level of performance, scalability, availability, and security (e.g., public cloud versus private cloud).

However, NoSQL is still a rapidly changing technology domain and due to the sheer heterogeneity involved, side-by-side comparison requires large development effort and expert knowledge, and is therefore costly. In addition, in the early prototyping phase, committing to certain technologies is also risky, as this might lead to vendor or technology

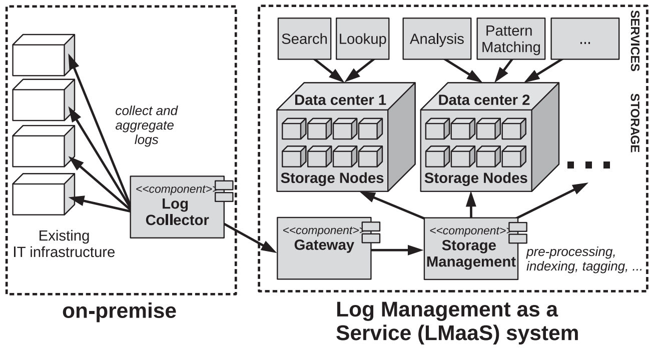


Fig. 2. Logical architecture of the log management as a service (LMaaS) system.

TABLE 2
Hardware Setup

Client Node	
Processor	Intel(R) Core(TM) i5 @ 2.60 GHz (Dual)
Memory	8 GB
Operating System	Windows 8
Server: 1 to 5 Nodes	
Processor	2 X Intel(R) Core(TM) 6,400 @ 2.13 Ghz
Memory	8 GB
Operating System	Linux/Ubuntu

TABLE 3
Software Used for the Evaluation

Name	Server	Client
JDK	Oracle 1.7.0_51	Oracle 1.7.0_51
Apache Cassandra	2.0.6	Cassandra-Thrift 2.1.3 Datastax Java driver 2.1.4 Kundera 2.15 Playorm 1.6.1 Spring Data 1.1.0
MongoDB	2.4.9	Java driver 2.10.1
Tomcat	7	Web Brower

lock-in. Furthermore, developers are still experimenting with different NoSQL systems while building and optimizing their SaaS application services. Therefore, if both the SaaS application services as well as data storage systems change rapidly, a middleware in between is necessary to limit the impact of migration changes. Moreover, such a middleware facilitates experimentation and side-by-side comparison, and provides a good starting point for hybrid setups.

Based on the current state of the middleware, as discussed in Section 2, it is clear that, in theory, data access middleware platforms can solve the problems caused by heterogeneity and limit the cost of technology disruption. Despite this clear motivation for adopting data access middleware for NoSQL data stores, two key questions remain:

1) Is the *performance impact* of such an intermediate data access middleware significant when compared to the overall performance of the NoSQL data store?
2) Are the APIs offered by these platforms sufficient to realize *easy migration* between NoSQL data stores?

The next two sections address these questions in a comparative study, conducted in the context of the LMaaS system described above.

4 Performance Impact

The first part of our study focuses on the performance impact, specifically the performance overhead introduced by the selected data access middleware platforms (Impetus Kundera, Playorm, and Spring Data). We focus specifically on answering the following questions:

Q1. *Insert/Read Overhead*: What is the performance impact of using the selected data access middleware platforms for insert and read operations?

Q2. *Update*[2]*/Delete Overhead*: What is the performance impact of using these data access middleware platforms for update and delete operations?

Q3. *Data Scale Overhead*: Does the performance overhead of these data access middleware platforms remain constant or does it increase with the increase in the data volume?

Q4. *Horizontal Scale Overhead*: Does the number of nodes in a cluster have an impact on the relative performance overhead of the selected data access middleware platforms?

Section 4.1 presents the experiments and their setup to answer the four different questions discussed above. Then, Section 4.2 presents the results of this part of the evaluation, which are then summarized in Section 4.3.

4.1 Experimental Setup

The performance overhead of the selected data access middleware platforms is investigated for the insert, random read, update and delete (CRUD) operations. As explained in Section 2, search operations are not considered in this evaluation because such operations are not directly supported by the selected data access middleware platforms and require integration of third-party libraries. Apache Cassandra [2] is selected for the back-end data storage and set to be a constant throughout the performance experiments.

Table 2 lists the hardware and Table 3 lists the software and APIs used for the evaluation on both client and server sides. As shown in Table 2, all performance benchmarks were run on the same hardware with a typical NoSQL cluster environment consisting of commodity machines in a distributed setup.

The performance benchmarks involve four different implementation setups[3]: (i) *Native*: an implementation without using any of the selected data access middleware platform, but uses the native Cassandra-Thrift API (see Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TCC.2015.2511756, for more information about the native client APIs for Cassandra), (ii) *Kundera*: an implementation that uses Kundera as a data access middleware platform, (iii) *Playorm*: an implementation that uses Playorm as a middleware platform, and (iv) *Spring Data*: an implementation that uses Spring Data as a middleware platform. The *Native* setup represents situation in which no data access middleware is used and therefore is the baseline for comparing the performance overhead introduced by the selected data access middleware platforms.

Fig. 3a represents situation in which the application uses a data access middleware platform, whereas Fig. 3b represents the situation in which the application directly uses the native client API. The overhead of each implementation setup that uses a data access middleware platform is calculated by comparing the performance of the selected data access middleware platform ($a + b + c + d$ in Fig. 3a) with

2. In practice, an update operation in NoSQL data stores is an upsert operation: insert if the record does not already exist otherwise update.

3. All the performance experiment setups are available at: http://people.cs.kuleuven.be/~ansar.rafique/TCC-Experiments.zip.

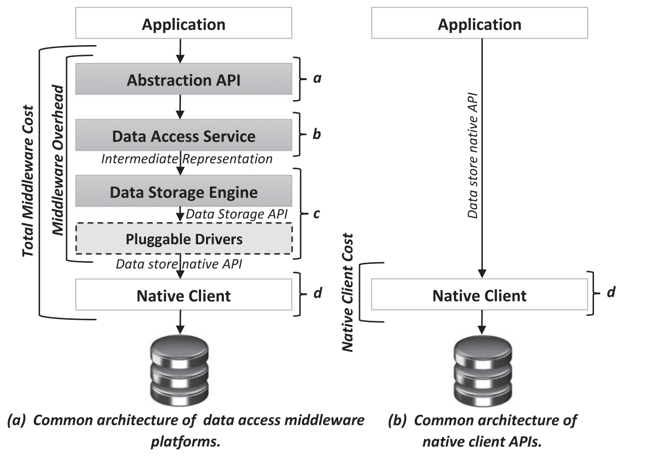


Fig. 3. Architecture of (a) data access middleware platforms, and (b) native client APIs.

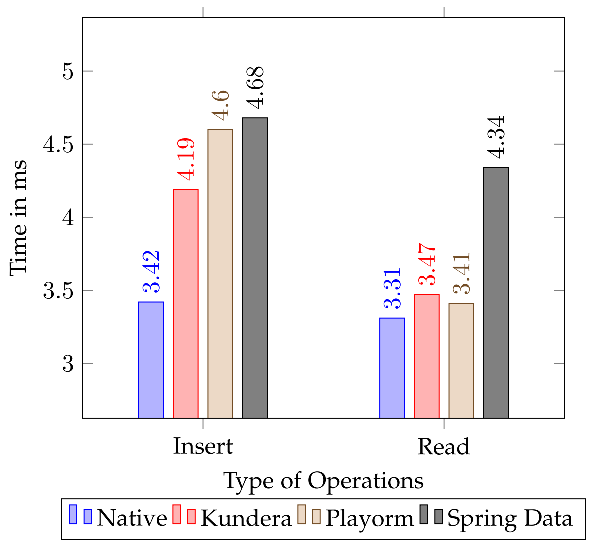


Fig. 4. Average insert and random read time for 1,280,000 `Log` entries on a five-node setup. *Native* is the baseline and the time above the *Native* is the overhead introduced by *Kundera*, *Playorm*, and *Spring Data* in ms.

the performance of the baseline that uses a native client API (*d* in Fig. 3b). We assume that the choice of a specific back-end storage system has no significant influence on the performance overhead of the selected data access middleware platforms and therefore can be factored out.[4] To avoid skewing the results, we repeat these experiments multiple times for each implementation setup and we present the average results.

To answer **Q1**, **Q2**, and **Q3**, Cassandra is deployed on a five-node cluster. For **Q1**, the performance overhead is evaluated for the insert and random read operations and for **Q2**, the performance overhead is evaluated for the update and delete operations. The overhead of these operations is evaluated on a large data volume, which involves 1,280,000 `Log` entries.

For **Q3**, the performance overhead is measured for CRUD operations by running a number of different scenarios. Each scenario considers `Log` entry volumes ranging from 10,000 to 1,280,000. The goal is to analyze whether the overhead introduced by these middleware platforms is constant regardless of the data volume.

To answer **Q4**, Cassandra is deployed on three alternative deployment setups involving a single node, a three-node, and a five-node cluster. The goal is to evaluate the relative overhead in the context of a realistic setup and analyze whether the overhead decreases when the back-end (i.e., number of nodes) is more complex. The performance overhead of the selected middleware platforms is measured for CRUD operations by running a scenario involving 1,280,000 `Log` entries.

For comparability of the results, all four implementation setups are configured to use the same Cassandra installation. The *replication_factor* is set to a constant which is 1 for all implementation setups throughout the performance experiments.

4.2 Performance Impact Results

This section presents the results of our performance benchmarks and as such shows the performance impact of the selected data access middleware platforms for NoSQL data stores.

4. We validate this assumption further in the Threats to validity section (Section 4.3.5).

4.2.1 Insert/Read Overhead

The first experiment is designed to evaluate the performance overhead of the selected data access middleware platforms for insert and random read operations on a five-node cluster (addressing **Q1**). The results of this experiment are presented in Fig. 4.

The performance overhead of selected platforms, mainly the overhead of *Spring Data* for the insert operation as well as random read operation is higher when compared to *Kundera* and *Playorm*. For the insert operation, the average overhead introduced by *Kundera* is 0.77 ms, the average overhead of *Playorm* is 1.18 ms, and the average overhead of *Spring Data* is 1.26 ms, which corresponds to 23, 35, and 37 percent overhead of the actual insert time (*Native*), respectively. In case of the random read operation, the average overhead introduced by *Kundera*, *Playorm*, and *Spring Data* is 0.16, 0.10, and 1.03 ms. That is to say, 5, 3, and 31 percent overhead of actual read time (*Native*), respectively.

4.2.2 Update/Delete Overhead

This experiment is designed to evaluate the performance overhead of the selected data access middleware platforms for update and delete operations on a five-node cluster (addressing **Q2**). The results of this experiment are presented in Fig. 5.

For the update operation, the average overhead introduced by *Kundera* is 0.73 ms, which is 23 percent overhead of an actual update time (*Native*). The average overhead of *Playorm* is 0.92 ms, and the average overhead of Spring Data is 1.02 ms which corresponds to 29 and 32 percent overhead, respectively. The performance overhead of the selected data access middleware platforms is significant for the delete operation. In case of the delete operation, the average overhead of *Kundera*, *Playorm*, and *Spring Data* is 2.98, 1.42, and 4.25 ms, that is to say, 100, 47, and 142 percent overhead respectively.

4.2.3 Data Scale Overhead

The evaluation presented in Sections 4.2.1 and 4.2.2 focused on storing, reading, updating and deleting large amounts of

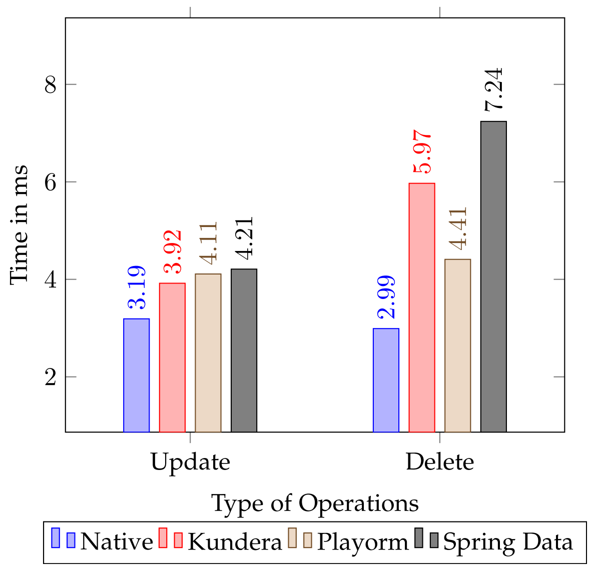
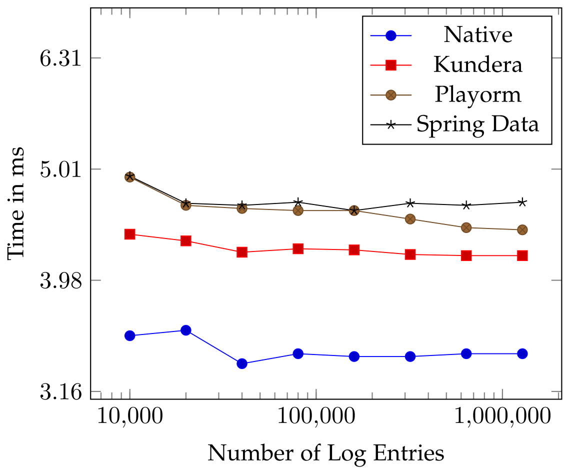


Fig. 5. Average update and delete time for 1,280,000 Log entries on a five-node Cassandra cluster. *Native* is the baseline and the time above the *Native* is the overhead introduced by the selected middleware platforms.

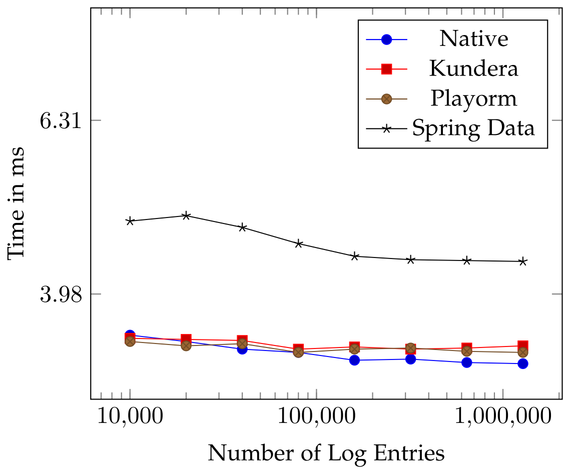
data. However, the question remains whether or not the overhead introduced by the middleware platforms is constant regardless of the data volume or if the data volume has any sort of influence on the performance overhead (addressing **Q3**). Therefore, for this evaluation, the performance impact of the selected data access middleware platforms is evaluated by inserting, reading, updating, and deleting a number of data entries in different data volumes (starting from 10 K up to 1,280 K).

Fig. 6a presents the performance results of the insert operation. As we can see, the average performance overhead introduced by the selected data access middleware platforms is more or less constant, regardless of the data volume. There are no big variations in terms of overhead at different data volume.

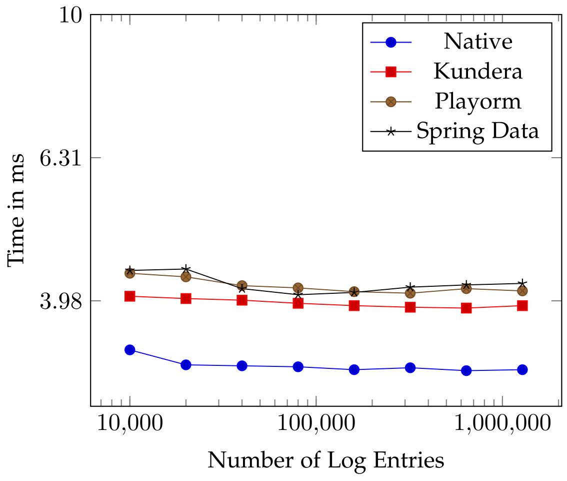
Fig. 6b shows the performance results for the random read operation. Initially, the average overhead increases slightly with the amount of data read and then more or less becomes constant again with only small variations. However, compared to *Kundera* and *Playorm*, the average overhead of *Spring Data* is significant for the read operation.



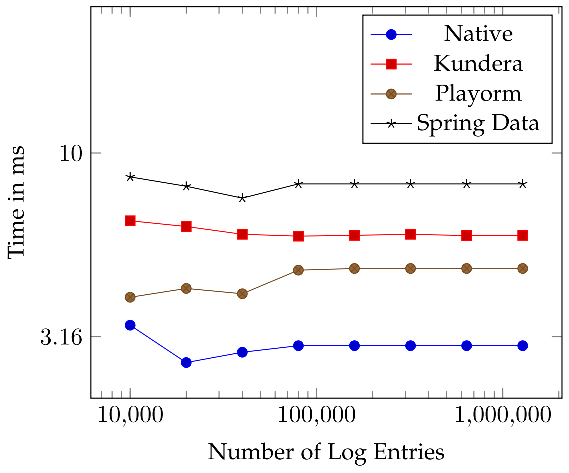
(a) Average *insert time* at increasing data volume on a five-node Cassandra Cluster.



(b) Average *read time* at increasing data volume on a five-node Cassandra cluster.

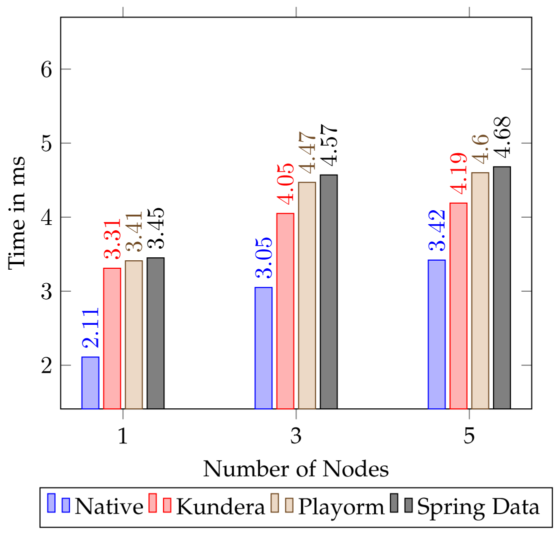


(c) Average *update time* at increasing data volume on a five-node Cassandra Cluster.

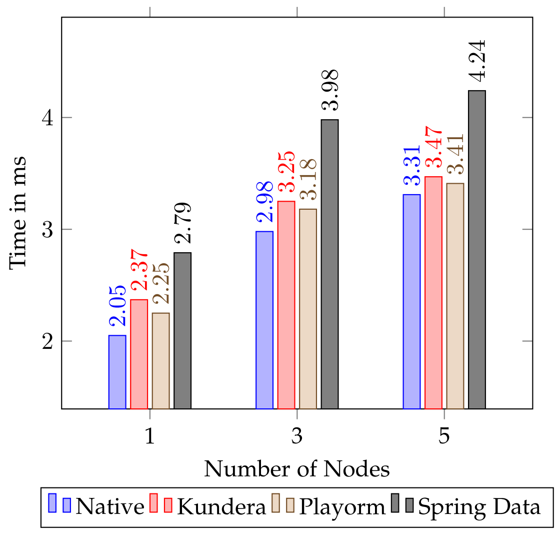


(d) Average *delete time* at increasing data volume on a five-node Cassandra Cluster.

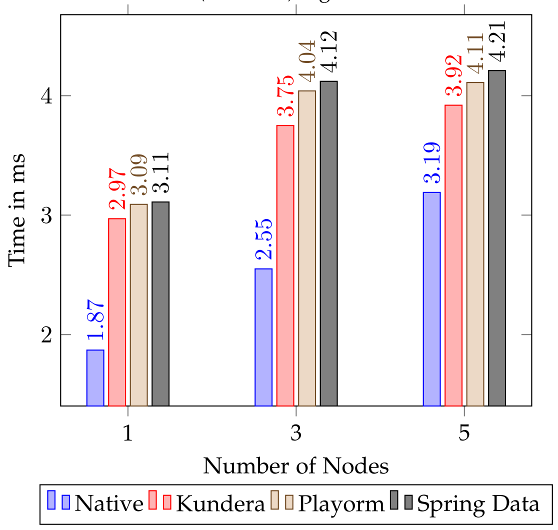
Fig. 6. Average time in ms for CRUD operations on a five-node Cassandra cluster with a minimum data volume of 10,000 and a maximum data volume of 1,280,000.



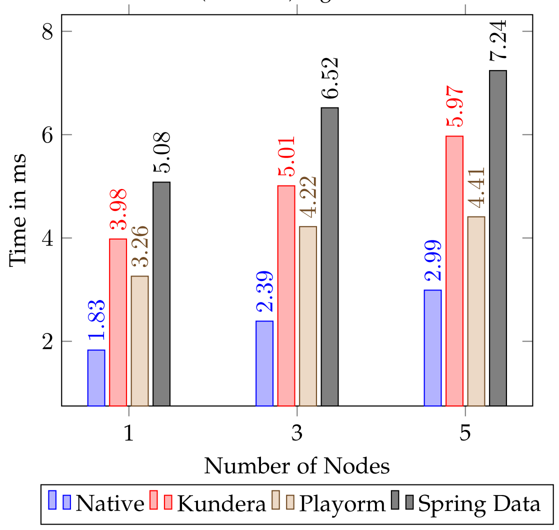
(a) Average *insert time* with varying cluster sizes from 1 node to 5 nodes for above million (1 280 000) log entries.



(b) Average *read time* with varying cluster sizes from 1 node to 5 nodes for above million (1 280 000) log entries.



(c) Average *update time* with varying cluster sizes from 1 node to 5 nodes for above million (1 280 000) log entries.



(d) Average *delete time* with varying cluster sizes from 1 node to 5 nodes for above million (1 280 000) log entries.

Fig. 7. Average time in ms for the CRUD operations at varying cluster size from 1 node to 5 nodes for above million (1,280,000) log entries.

Fig. 6c presents the performance results of the update operation. Again, there are no variations at different data volume and the average overhead is more or less constant.

Fig. 6d shows the performance overhead results for the delete operation. Again, the average overhead introduced by data access middleware platforms is more or less constant, but as we have shown in Section 4.2.2, very significant compared to the baseline.

4.2.4 Horizontal Scale Overhead

The performance overhead for CRUD operations on a five-node cluster has been discussed in Sections 4.2.1 and 4.2.2 respectively. Although NoSQL data stores are specifically designed to run in a distributed setup (i.e., multiple machines running NoSQL systems in a large data center), but a single node setup is used in practice as well (e.g., small web applications). Therefore, it is essential to evaluate the performance overhead of the selected data access middleware platforms for both a single node setup as well as a multi-node setup (addressing **Q4**). The goal is to get a preliminary idea of the overhead that can be introduced by the selected data access middleware on different deployment setups (i.e., on a single node, a three-node, and a five-node cluster). Figs. 7a, 7b, 7c, and 7d present the relative overhead introduced by the selected middleware platforms at the volume of 1,280,000 data entries for CRUD operations.

Fig. 7a presents the performance results of the insert operation. The average overhead of *Kundera* is 1.2 ms on a single node, 1.0 ms on a three-node, and 0.77 ms on a five-node cluster, which corresponds to 57 percent overhead of an actual insert operation (*Native*) on a single node, 33 percent overhead on a three-node, and 23 percent on a five-node cluster. *Playorm* introduces an average overhead of 1.3, 1.42, and 1.18, which corresponds to 62, 47, and 35 percent overhead of an actual insert operation (*Native*) on a single node, a three-node, and a five-node cluster respectively.

TABLE 4
Average *Absolute Performance Overhead* in ms Introduced by the Selected Data Access Middleware on Different Deployment Setups for Insert and Read Operations

	1-Node		*3-Nodes*		*5-Nodes*	
Platform	**Insert**	**Read**	**Insert**	**Read**	**Insert**	**Read**
Kundera	1.20	0.32	1.00	0.27	0.77	0.16
Playorm	1.30	0.20	1.42	0.20	1.18	0.10
Spring Data	1.34	0.74	1.52	1.00	1.26	0.93

TABLE 5
Average *Relative Performance Overhead* Introduced by the Selected Data Access Middleware on Different Deployment Setups for Insert and Read Operations

	1-Node		*3-Nodes*		*5-Nodes*	
Platform	**Insert**	**Read**	**Insert**	**Read**	**Insert**	**Read**
Kundera	57%	16%	33%	9%	23%	5%
Playorm	62%	10%	47%	7%	35%	3%
Spring Data	64%	36%	50%	34%	37%	28%

TABLE 6
Average *Absolute Performance Overhead* in ms Introduced by the Selected Data Access Middleware on Different Deployment Setups for Update and Delete Operations

	1-Node		*3-Nodes*		*5-Nodes*	
Platform	**Upd**	**Delete**	**Upd**	**Delete**	**Upd**	**Delete**
Kundera	1.10	2.15	1.20	2.62	0.73	2.98
Playorm	1.22	1.43	1.49	1.83	0.92	1.42
Spring Data	1.24	3.25	1.57	4.13	1.02	4.25

TABLE 7
Average *Relative Performance Overhead* Introduced by the Selected Data Access Middleware on Different Deployment Setups for Update and Delete Operations

	1-Node		*3-Nodes*		*5-Nodes*	
Platform	**Upd**	**Delete**	**Upd**	**Delete**	**Upd**	**Delete**
Kundera	59%	117%	47%	110%	23%	100%
Playorm	65%	78%	58%	77%	29%	47%
Spring Data	66%	178%	62%	173%	32%	142%

The average overhead of *Spring Data* is 1.34 ms which corresponds to 64 percent overhead, 1.52, and 1.26 ms which corresponds to 50 percent overhead and 37 percent overhead of an actual insert operation (*Native*) on a single node, a three-node, and a five-node cluster respectively.

Fig. 7b presents the performance results of the random read operation. In case of the random read operation, *Kundera* introduces an average overhead of 0.32 and 0.27 ms which is 16 percent overhead and 9 percent overhead of an actual read operation (*Native*) on a single node and a three-node cluster. However, in case of a five-node cluster the average overhead of *Kundera* decreases to 0.16 ms which corresponds to 5 percent. The average overhead of *Playorm* is more or less similar to the overhead of *Kundera* with a small variation on different deployment setups. *Playorm* introduces 10 percent average overhead of an actual read operation (*Native*) on a single node, 7 percent on a three-node, and 3 percent on a five-node cluster. *Spring Data* introduces an average overhead of 0.74, 1.00, and 0.93 ms, which is 36 percent overhead of an actual read operation (*Native*) on a single node, 34 percent on a three-node cluster; however, in case of a five-node cluster, the overhead reduced to 28 percent.

For the update operation as shown in Fig. 7c, *Kundera* introduces an average overhead of 1.10, 1.20, and 0.73 ms which corresponds to 59, 47, and 23 percent overhead of an actual update operation (*Native*) on a single node, a three-node, and a five-node cluster respectively. The average overhead of *Playorm* is 1.22 ms, which is equal to 65 percent overhead of an actual update operation (*Native*) on a single node, 1.49 ms, which is 58 percent overhead on a three-node cluster, and 0.92 ms, which is 29 percent overhead on a five-node cluster. *Spring Data* introduces an average overhead of 1.24 ms on a single node, 1.57 ms on a three-node, and 1.02 ms on a five-node cluster, which corresponds to 66, 62, and 32 percent overhead.

Fig. 7d presents the performance results of the delete operation. As shown, *Kundera* introduces an average overhead of 2.15, 2.62, and 2.98 ms, that is to say, 117 percent overhead of an actual delete operation (*Native*) on a single node, 110 percent on a three-node cluster, and 100 percent on a five-node cluster. The average overhead of *Playorm* is 1.43 ms, which is 78 percent, 1.83 ms, which is 77 percent, and 1.42 ms, which is 47 percent overhead of an actual delete operation (*Native*) on a single node, a three-node, and a five-node cluster respectively. *Spring Data* introduces an average overhead of 3.25, 4.13, and 4.25 ms which corresponds to 178 percent overhead of an actual delete operation (*Native*) on a single node, 173 percent on a three-node cluster, and 142 percent on a five-node cluster.

4.3 Discussion of the Results

We examined the overhead introduced by the selected data access middleware platforms on different deployment setups. The average absolute overhead introduced by these platforms on different setups is summarized in Table 4 for the insert and random read operations and in Table 6 for the update and delete operations. Similarly, the average relative overhead introduced by the selected platforms is presented in Table 5 for the insert and random read operations and in Table 7 for the update and delete operations.

4.3.1 *Discussion on **Q1***

Overall, the obtained data presented indicates that the insert operation is more costly than the random read operation using data access middleware platforms. The random read operation introduces less performance overhead because the cost of translation from the `Abstraction API` (*a* in Fig. 3a) to the `Data Storage API` (*c* in Fig. 3a) is high for the insert operation. The main reason for this is that the insert operation involves the entire entity, whereas the random read operation only involves the identifier of the entity. Also, a cache hit[5] occurs more frequently for the random

5. A cache hit occurs when the requested data can be found in the cache which avoids reading the data from the disk.

read operation where the future requests for that data can be served faster. The performance overhead of *Spring Data* is significant for insert and random read operations when compared to the overhead of *Kundera* and *Playorm*. *Spring Data* introduces more performance overhead compared to *Kundera* and *Playorm* for the random read operation because we believe that it does not use a cache to improve the performance of random read operations. However, by looking at Table 5, we can see that the performance overhead of *Spring Data* is more or less similar to the overhead of *Playorm* for the insert operation, but is mainly significant for the random read operation on different deployment setups. *Kundera* performs better than *Playorm* and *Spring Data* for the insert operation and introduces less performance overhead. For the random read operation, both *Kundera* and *Playorm* perform better than *Spring Data* and introduce more or less the same performance overhead with small variations on different deployment setups. *Playorm* performs better than *Kundera* which in turn performs better than *Spring Data* for the random read operation.

4.3.2 Discussion on **Q2**

Table 7 summarizes the performance results of the update and delete operations on different deployment setups. As with many Big Data applications, the LMaaS application relies extensively on insert and read operations, and less on update and delete operations. However, we included benchmarks for update and delete operations in our study for the sake of completeness. We have learned that there are performance overhead differences between the selected data access middleware platforms when compared to the baseline (*Native*). These differences are significant mainly for the delete operation. The delete operation is costly and the selected data access middleware platforms introduce significant performance overhead on all deployment setups.

We believe that the use of JPA is a potential cause of this significant overhead introduced in the case of the delete operation. The traditional approach requires the database to read data before performing an update or delete operation. These operations require seek, which is unnecessarily expensive when the performance matters. One of the key features of NoSQL data stores is to improve the performance and, as such, most of the NoSQL data stores do not update and delete data in-place on disk for performance reasons. For example, Cassandra does not update or delete data in-place on disk to improve the performance. In Cassandra, SSTable files are immutable [16]. Therefore, every update and delete operation in Cassandra is a new insert operation, which does not require the reading of data first using native APIs. However, in case of a JPA where the operations are performed on entity level, data needs to be read first, before performing delete operations on an entity. This combines the cost of reading and writing data together and therefore increases performance overhead.

The overhead of *Spring Data* is significant when compared to the overhead of *Kundera*, and *Playorm* for the delete operation. This is mainly because the overhead of *Spring Data* for the read operation is significant which affects the overall performance of the delete operation.

As shown in Table 7, the performance overhead of the selected data access middleware platforms for the update operation is more or less similar to the insert operation because every update operation is an upsert operation in NoSQL data stores, meaning: insert if the record does not already exist, otherwise update it.

4.3.3 Discussion on **Q3**

We have shown that the overhead of the selected data access middleware platforms is constant and there is no correlation to the amount of data being manipulated (the data scale). The small variations, mostly visible before 80,000 data entries, are at the maximum level of 0.10 ms, which are small enough to be ignored.

4.3.4 Discussion on **Q4**

We have shown that the relative overhead of the selected data access middleware platforms is most significant on a single node setup. However, the relative overhead decreases with an increasing number of nodes as shown in Tables 5 and 7. For example, the relative overhead introduced by *Kundera* for the insert operation is 57 percent on a single node, but decreases to 23 percent with an increasing number of nodes. As discussed in Section 4.1, the *replication_factor* is set to 1 which means that there is one copy of each row. The data is equally distributed over the storage nodes. The relative overhead decreases because in case of a multi-node setup, back-end storage systems require more time to service the requests and thus the latency between the storage systems for inter-node communication comes into play. This increases the performance of the native client and therefore, decreases the relative overhead of the data access middleware platforms.

We have also noticed that there are small variations in the absolute overhead. The absolute overhead of the data access middleware platforms decreases slightly with an increasing number of nodes for the insert, random read, and update operations, but increases for the delete operation as shown in Tables 4 and 6. These variations are small enough to be ignored. However, it remains to be determined why the absolute overhead introduced by the selected middleware platforms decreases for all other operations, but increases slightly for the delete operation.

4.3.5 Threats to Validity

This section presents the threats to validity that can compromise the results of this performance study.

Internal validity. The most noteworthy threat to the validity of our conclusions relates to our performance overhead calculation for the *Spring Data* middleware. These calculations may be skewed because we employed the measurements for the *Native* implementation as the baseline for the comparison to calculate the performance overhead for all three middleware. The *Native* implementation uses the Cassandra-Thrift API underneath, while Spring Data does not use the Cassandra-Thrift API, but the DataStax Java-driver API. Other studies [3], however, show that the performance difference between these APIs is typically not very large (at the scale of 0.01 ms), and as such, the measurements for the Cassandra-Thrift API are expected to be in the same ballpark as those for the DataStax Java-driver API.

TABLE 8
Average Relative Performance Overhead Introduced by the Selected Data Access Middleware for Cassandra and MongoDB Data Stores on a Single Node Setup

	Cassandra Back-end				*MongoDB Back-end*			
Platform	**Insert**	**Read**	**Update**	**Delete**	**Insert**	**Read**	**Update**	**Delete**
Kundera	57%	16%	59%	117%	55%	25%	58%	109%
Playorm	62%	10%	65%	78%	77%	18%	75%	81%
Spring Data	64%	36%	66%	178%	144%	25%	100%	145%

External validity. The main threat to external validity is the fact that we have experimented with only one (albeit representative) application, the LMaaS application which is the prototype of an actual SaaS application. It is a typical NoSQL application which stores a stream of logs to be able to analyze the suspicious activities. The scalability and availability requirements are the priority concerns of the application. All the log information is stored in a single `Log` entity and there are no associations between the entities. The performance overhead of the selected data access middleware platforms might vary if we consider an application which has associations between the entities. We expect, in such a case, the performance overhead will be higher because of the cost of translation involves multiple entities to be examined.

In this study, we made the implicit assumption that the choice of a specific back-end storage system has no significant influence on the performance impact of the selected data access middleware platforms. More specifically, we assumed that the impact of the back-end is more or less constant and therefore can be factored out of the equation. To validate this and to improve our confidence in the assumption, we ran the experiments again on a single node setup, this time using MongoDB as the back-end storage system instead of Cassandra. The results of these experiments are presented in Table 8, next to the results obtained with Cassandra as the back-end storage system.

These results indicate that the assumption is valid for *Kundera* and *Playorm*, but invalid for *Spring Data*: there are no big variations in terms of performance overhead for *Kundera* and *Playorm* using both Cassandra and MongoDB as back-end data storage systems. The small variations are negligible because even a delay of 0.10 ms results in 5—20 percent increase in the performance overhead. However, our experimental results show more significant variations in the case of *Spring Data* for which the performance overhead seems to depend on the specific back-end storage system. Although this warrants further research, we believe that *Spring Data* introduces variations mainly because it lacks a uniform API for heterogeneous back-end data stores. Therefore, in order to ran the experiments again using MongoDB as the back-end storage system, we had to adopt a different API which offers a different set of classes. We discuss in-depth why *Spring Data* is different in Section 5.3.

5 Cost of Migration

The second part of our study focuses on the effort required to migrate across heterogeneous NoSQL data stores: Cassandra [2] and MongoDB [31]. We selected these data stores because they are the most popular and representative data stores and they support very different data models (i.e., MongoDB is a document storage system, whereas Cassandra is a wide column store), programming model, and support for available queries. Consequently, migration from MongoDB to Cassandra or vice versa is a non-trivial activity. Migration usually involves both porting the application as well as migrating the data to different data stores. In this study, we focus on the former.

In Section 5.1, we discuss the prototype implementations of the LMaaS system in terms of APIs and data storage systems used for the migration study. Then, Section 5.2 presents the results of migration study, which are then summarized in Section 5.3.

5.1 Experimental Setup

Our migration study includes five prototypes of the LMaaS system: (i) *Prototype Cassandra* uses Cassandra [2] for the data storage and the Datastax Java-driver[6] [14] as the native client API to interact with Cassandra (see Appendix A, available in the online supplemental material, for more information about the native client APIs for Cassandra), (ii) *Prototype MongoDB* uses MongoDB [31] for the data storage and the Java MongoDB driver as the native client API to interact with MongoDB (see Appendix A, available in the online supplemental material, for more information about the native client APIs for MongoDB), (iii) *Prototype Kundera* uses both selected NoSQL data stores, Cassandra [2] and MongoDB [31] for the data storage and the Kundera [27] as an abstraction API to interact with these NoSQL data stores, (iv) *Prototype Playorm* uses Cassandra [2] and MongoDB [31] for the data storage and the Playorm [26] as an abstraction API to interact with the selected NoSQL data stores, and (v) *Prototype Spring Data* uses both NoSQL data stores, Cassandra [2] and MongoDB [31] for the data storage and the Spring Data [40] as an abstraction API to interact with these data stores.

All these prototype implementations run on Tomcat 7. We compared the required effort by port the prototype implementations from Cassandra to MongoDB and vice versa. We specifically focus on the lines of code (LOC) and lines of configuration (LOConfig) files that were changed.

5.2 Migration Cost Results

In this section, we discuss the migration cost results for all the prototype implementations of the LMaaS system discussed above. Section 5.2.1 discusses the migration results for the prototype implementations that directly use the native client APIs offered by the NoSQL data stores, while Section 5.2.2 discusses the migration results for the prototype implementations that use the selected data access middleware platforms.

5.2.1 Results for Native Client APIs

The migration cost results of the prototype implementations that use the native client APIs are presented in Table 9.

6. The *Datastax Java-driver* is preferred over the *Cassandra-Thrift* API because of the abstraction it provides and thus reduces the migration cost, development time, and effort.

TABLE 9
Cost of Migration for Prototype Cassandra and Prototype MongoDB using the Selected Native Client APIs

	LOC changed		*LOConfig changed*	
Prototype	**Cassandra**	**MongoDB**	**Cassandra**	**MongoDB**
Cassandra	-	120/30%	-	-
MongoDB	200/50%	-	-	-

The first row of the Table 9 shows that the *Prototype Cassandra* required 120 LOC to be changed to port the prototype implementation from Cassandra data store to the MongoDB data store, which corresponds to 30 percent of the entire prototype. Porting *Prototype MongoDB*, required changing 200 LOC to port the prototype implementation from MongoDB to Cassandra, which corresponds to 50 percent of the entire prototype, as shown in the second row of Table 9. The native APIs do not rely on configuration files.

5.2.2 Results for Data Access Middleware Platforms

The migration results of the prototype implementations that use data access midddleware platforms are presented in Table 10.

As shown in the first row of Table 10, the *Prototype Playorm* only required changing 5 LOC to port the prototype implementation from Cassandra to MongoDB and vice versa, which corresponds to 1.25 percent of the entire prototype. In Playorm, the data store specific configuration is specified in the application code as shown in Listing 1.

Listing 1. Example of a client class in Playorm that contains Cassandra-specific configurations

```
// PlayormFactory.java
public class PlayormFactory{
  public String seeds = “(...)”;
  public DbTypeEnum db = DbTypeEnum.CASSANDRA;
  public String CL = “CL_ONE”;

  public NoSqlEntityManager EM(DbTypeEnum db){
   properties.put(Bootstrap.CASSANDRA_DEFAULT
   _CONSISTENCY_LEVEL,“CL”);
   Bootstrap.createAndAddBestCassandra
   Configuration(db, “”, “”, seeds);
  }
}
```

On the other hand, the *Prototype Spring Data* required changing 85 LOC which corresponds to 21.25 percent of the entire prototype to port from Cassandra to MongoDB and 60 LOC (15 percent of the entire prototype) had to be changed to migrate the prototype implementation from MongoDB to Cassandra as shown in the second row of the Table 10.

In case of *Prototype Kundera*, only 3 LOConfig which corresponds to 0.75 percent of the entire prototype were changed to migrate from Cassandra to MongoDB and vice versa as shown in the last row of the Table 10. In Kundera, the data store specific configurations are defined in the `persistence.xml` configuration file which is the standard way of defining configurations in JPA. In this file, each data store configuration is defined as a *persistence-unit* that contains a number of properties specific to a data store as shown in Listing 2.

TABLE 10
Cost of Migration for Prototypes Playorm, SpringData, and Kundera Using the Selected Middleware Platforms

	LOC changed		*LOConfig changed*	
Prototype	**Cassandra**	**MongoDB**	**Cassandra**	**MongoDB**
Playorm	5/1.25%	5/1.25%	-	-
Spring Data	60/15%	85/21.25%	-	-
Kundera	-	-	3/0.75%	3/0.75%

5.3 Discussion on Migration Cost

We have conducted a study to examine the cost of migration across heterogeneous NoSQL data stores using the selected data access middleware platforms compared to the native client APIs. The performance impact evaluation of Section 4.3 has shown that the selected data access middleware platforms come with a performance cost. This study sheds some light on the flipside of the coin by showing that these platforms provide benefits in terms of easy migration across NoSQL data stores. Our evaluation shows that the selected data access middleware platforms significantly simplify migration across heterogeneous NoSQL data stores when compared to the native client APIs.

Listing 2. Example of a `persistence.xml` configuration file in Kundera that defines *persistence-unit* for Cassandra

```
<persistence version="2.0">
  <persistence-unit name="Cassandra">
    <properties>
      ...
       <property name="kundera.dialect" value="cassandra"/>
      <property name="kundera.client.lookup.class"
        value="(...)"/>
      ...
    </properties>
  </persistence-unit>
</persistence>
```

Furthermore, Kundera and Playorm require less development time for migration, making it easy to migrate across multiple data stores with limited effort. Moreover, an application that is developed using Kundera or Playorm requires the same LOC to be changed to migrate across supported data stores.

In the case of Spring Data, the required development effort is higher in terms of changed LOC. Although Spring Data also addresses the heterogeneity problem, it lacks a uniform API that offers a set of features implemented by a number of heterogeneous NoSQL data stores (i.e., the `Abstraction API` (*a*) depicted in Fig. 3). Instead, it provides a different module for each supported NoSQL data store which offers a different API to the developers. Hence, porting an application to different NoSQL data stores supported by Spring Data requires re-implementing against these APIs and thus involves changing substantially more LOC.

Another important factor to consider is the type of change required to port across the heterogeneous NoSQL systems which reflects the (re)deployment of an application. In *Prototype Playorm* and *Prototype Spring Data*, the application code needs to be changed, and therefore the application has to be (re)compiled and (re)deployed. In case of *Prototype Kundera* however, this is done in the configuration file which is the standard way of defining a data store specific information. In Kundera, neither the code of the prototype implementation needs to be changed, nor the application code needs to be (re)compiled and (re)deployed as the data stores can be (re)configured at run time. Migrating to and from another SQL or NoSQL data store (e.g., HBase, CouchDB, Redis, MySQL) that is supported by Kundera only requires 3 LOConfig changing in the `persistence.xml` configuration file.

In the case of native client APIs, in which both prototypes use native libraries, *Prototype Cassandra* and the *Prototype MongoDB* require the same operations to be supported for Cassandra [2] and MongoDB [31]. As a consequence, this requires substantial (re)writing of the application code. Additionally, both native prototypes involve different data models to be adopted, hence requiring significant learning curve to deal with this heterogeneity.

5.3.1 Threats to Validity

The most obvious threats to the validity relates to the prototype implementation used for the migration study. The study is conducted on a small prototype implementation. The main objective of this study is to highlight the benefits of using data access middleware platforms. More specifically, the benefits offered by data access middleware platforms in terms of easy migration. However, we expect in larger applications the cost of porting across heterogeneous NoSQL data stores when used with different data access middleware platforms (e.g., Kundera, Playorm) will be the same.

6 Related Work

This section discusses three domains of related work: (i) heterogeneity support, (ii) performance studies, and (iii) migration studies.

6.1 Addressing Cloud Heterogeneity

The increasing popularity of NoSQL data stores has spawned the interesting research on middleware dealing with heterogeneity and support interoperability and portability. The problem of heterogeneity among the NoSQL systems has been recognized by both the industry [25], [26], [27] and the research community [4], [7], [8], [17], [42]. Similarly, there are a number of other research works [5], [19] that provide a federated cloud storage system to integrate diverse public cloud storage providers. These works aim to: (i) hide the complexity of using different interfaces provided by these public cloud storage providers, and (ii) use multiple public cloud storage systems to get composite benefits as well as avoid vendor and technology lock-in.

In our previous work [37], we investigated the heterogeneity problem across PaaS platforms by building an abstraction API, supporting different NoSQL systems for a common PaaS service in a hybrid cloud environment. The work addresses the challenges of heterogeneity and support application portability across the PaaS platforms in a hybrid setup; however, we have learned that this heterogeneity, in terms of different data storage systems, exists even within a single cloud environment (e.g., private cloud). The work clearly indicates the need to address the heterogeneity problem, improve database portability, and tackle vendor/technology lock-in.

6.2 Performance Studies

The applicability and the performance evaluation of NoSQL data stores has been studied before [9], [15], [22], [36], [44], [45]. However, to the best of our knowledge, there is no systematic study that characterizes both the performance overhead as well as the cost of migration in data access middleware platforms when used with NoSQL data stores.

Storl et al. [41] studied the state-of-the-art in object-relational mappers and dedicated object-NoSQL mappers that can handle heterogeneous NoSQL data stores. The authors study the performance of the abstraction layers for NoSQL data stores with a focus on the run-time performance which is comparable to the first part of our study. The main difference to our work is that we perform a more extensive (e.g., a multi-node setup) performance evaluation and consider other popular data access middleware platforms (e.g., Spring Data, Playorm). Their research states several interesting conclusions worth discussing in relation to our study: (i) when reading the data, there are small run-time performance differences between the native and the abstraction APIs, whereas we have learned that the conclusion is not valid for all abstraction APIs (i.e., the overhead of Spring Data is significant for the read operation), (ii) they conclude that the overhead of abstraction APIs is significant for the write operation, which we confirmed in our performance study, and (iii) their research concludes that the back-end storage system has an influence on the performance overhead of these abstraction APIs. In Section 4.3.5, we investigated further to analyze the impact of the back-end on the performance of data access middleware platforms and learned that this conclusion does not hold for all the abstraction APIs.

Another related work is the research conducted by Bunch et al. [8]. The authors present AppScale: a Platform-as-a-Service (PaaS) cloud infrastructure. AppScale is a platform that provides a uniform API, enabling different NoSQL systems to be evaluated. In their research, the focus is on the performance evaluation of heterogeneous NoSQL systems using AppScale platform, but they don't quantify the performance overhead of AppScale, which is the focus of our research.

In another similar study, the research conducted by van Zyl et al. [46] gain insight into the performance differences of using two persistence mechanism such as object databases (i.e., db4o) and object-relational mapping (ORM) tools (i.e., Hibernate ORM) to communicate with relational databases. In their research, the focus is only on the relational model, whereas our research focus on broader aspect, address heterogeneity, and cover NoSQL data stores which follow different data models (e.g., document oriented data model and wide column).

6.3 Migration Studies

Data migration across heterogeneous NoSQL systems is a non-trivial activity due to the heterogeneity in these technologies. Some Database-as-a-Service (DBaaS) providers provide tools to import and export data from their data stores. However, the usage of such a tool is limited to a specific NoSQL storage system and does not address the vendor lock-in.

Another research conducted by Scavuzzo et al. [38] focuses on data migration across columnar NoSQL data stores. The authors proposed an extensible system that can be used to migrate data across different NoSQL data stores. The system allows developers to easily add a support for the new data store. The goal of their research is to provide a platform for easier migration across columnar NoSQL data stores. However, the key differences is that they have only focused on data migration in terms of the performance cost while our research focuses on the development effort required to migrate across multiple NoSQL data stores.

7 Conclusion and Future Work

We have conducted an in-depth study of the trade-off between the performance overhead and the cost of migration, inherent to the decision of using data access middleware platforms. We present two complementary studies in which we compare three Java-based data access middleware platforms for NoSQL systems: Impetus Kundera, Playorm, and Spring Data. These platforms are inevitable to achieve portability, interoperability, and easy migration across NoSQL storage systems. Due to the heterogeneity and increasing popularity of NoSQL data stores, we believe that such a middleware platform will become increasingly useful in the near future.

Our performance study shows that, despite these platforms are similar in design, there are still substantial differences in terms of performance which indicates different levels of maturity. In general, the extent of the performance overhead depends highly on the nature of the operation, the delete operation being the most costly. In particular, we have demonstrated that by allowing some performance overhead, the developer gain benefits in terms of portability and easy migration across heterogeneous data stores. Our studies show that Kundera is ahead of the studied systems, introduces less performance overhead, and requires the least migration effort. This study also highlights the limitations of using JPA in case of NoSQL data stores, notably for the delete operation and raises the question if this standard is suitable for NoSQL systems.

The work presented in this paper forms the foundation of our ongoing research. Based on our experience with the initial evaluation, we have several short and long-term goals. In the future, we first plan to extend our evaluation to include search operations. Second, we plan to consider other data access middleware platforms and conduct our benchmark in extended setups, e.g., spanning multiple availability zones on the server side, to evaluate the overhead of these platforms. In another line of work, we want to confirm our findings with other benchmarks (e.g., YCSB [9]). Finally, we intend to extend our migration study by also taking into account the cost to migrate data across heterogeneous data storage systems.

Acknowledgments

This research is partially funded by the Research Fund KU Leuven (project GOA/14/003 - ADDIS) and the iMinds DMS2 project, which is co-funded by iMinds (Interdisciplinary institute for Technology), a research institute founded by the Flemish Government. Companies and organizations involved in the project are Agfa Healthcare, Luciad, UP-nxt, and Verizon Terremark, with project support of IWT (government agency for Innovation by Science and Technology).

References

[1] Apache. (2014, Oct. 13). Apache gora [Online]. Available: http://gora.apache.org/

[2] Apache. (2015, Mar. 11). Cassandra [Online]. Available: http://cassandra.apache.org/

[3] Apache. (2014, Oct. 18). Compare string vs. binary prepared statement parameters [Online]. Available: https://issues.apache.org/jira/browse/CASSANDRA-3634

[4] P. Atzeni, F. Bugiotti, and L. Rossi, "Uniform access to non-relational database systems: The sos platform," *Proc. 24th Int. Conf. Adv. Inf. Syst. Eng.*, 2012, vol. 7328, pp. 160–174.

[5] D. Bermbach, M. Klems, S. Tai, and M. Menzel, "Metastorage: A federated cloud storage system to manage consistency-latency tradeoffs," in *Proc. 4th Int. Conf. Cloud Comput.*, 2011, pp. 452–459.

[6] E. Brewer, "Pushing the cap: Strategies for consistency and availability," *J. Comput.*, vol. 45, no. 2, pp. 23–29, 2012.

[7] F. Bugiotti and L. Cabibbo, "An object-datastore mapper supporting NoSQL database design (ONDM)," [Last visited on Jun. 30, 2014].

[8] C. Bunch, N. Chohan, C. Krintz, J. Chohan, J. Kupferman, P. Lakhina, Y. Li, and Y. Nomura, "An evaluation of distributed datastores using the appscale cloud platform," in *Proc. IEEE 3rd Int. Conf. Cloud Comput.*, 2010, pp. 305–312.

[9] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proc. 1st ACM Symp. Cloud Comput.*, 2012, pp. 25–36.

[10] CUSTOMSS. (2014, Jun. 27). CUSTOMization of Software Services in the cloud (iMinds ICON project [Online]. Available: http://www.iminds.be/en/research/overview-projects/p/detail/customss

[11] D-Base. (2014, Jun. 27). Decentralized support for Business processes in Application Services [Online]. Available: http://www.iminds.be/en/research/overview-projects/p/detail/d-base

[12] DataNucleus. (2015, Mar. 17). DataNucleus, tags = "data access middleware" [Online]. Available: http://datanucleus.org/

[13] Datastax. (2014, Jun. 29). DataStax Java Driver: A new face for Cassandra [Online]. Available: http://www.datastax.com/dev/blog/new-datastax-drivers-a-new-face-for-cassandra

[14] Datastax. (2015, Mar. 12). DataStax Java Driver for Apache Cassandra [Online]. Available: https://github.com/datastax/java-driver

[15] E. Dede, M. Govindaraju, R. S. Canon, and L. Ramakrishnan, "Performance evaluation of a mongodb and hadoop platform for scientific data analysis," in *Proc. 4th ACM Workshop Sci. Cloud Comput.*, 2013, pp. 13–20.

[16] E. Dede, B. Sendir, P. Kuzlu, J. Hartog, and M. Govindaraju, "An evaluation of cassandra for hadoop," in *Proc. IEEE 6th Int. Conf. Cloud Comput.*, 2013, pp. 494–501.

[17] A. Dey, A. Fekete, and U. Rohm, "Scalable transactions across heterogeneous nosql key-value data stores," *Proc. VLDB Endowment*, vol. 6, pp. 1434–1439, 2013.

[18] [DMS]2. (2014, Jun. 18). Decentralized Data Management and Migration for SaaS (iMinds ICON project [Online]. Available: http://www.iminds.be/en/research/overview-projects/p/detail/dms2

[19] D. Dobre, P. Viotti, and M. Vukolic, "Hybris: Robust hybrid cloud storage," in *Proc. ACM Symp. Cloud Comput.*, 2014, pp. 1–14.

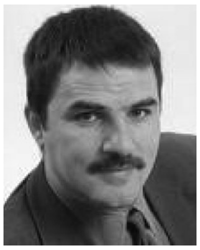
[20] eclipse. (2015, Mar. 12). Eclipselink [Online]. Available: https://eclipse.org/eclipselink/

[21] Elasticsearch. (2014, May 24). Elasticsearch [Online]. Available: http://www.elasticsearch.org/
[22] A. Floratou, N. Teletia, D. J. DeWitt, J. M. Patel, and D. Zhang, "Can the elephants handle the NoSQL onslaught? " *Proc. VLDB Endowment*, vol. 5, pp. 1712–1723, 2012.
[23] GehrigKunz. (2014, Jun. 29). ClientOptions Thrift [Online]. Available: https://wiki.apache.org/cassandra/ClientOptionsThrift
[24] K. Grolinger, W. A. Higashino, A. Tiwari, and M. A. M. Capretz, "Data management in cloud environments: NoSQL and newsql data stores," *J. Cloud Comput.: Adv., Syst. Appl.*, vol. 2, no. 1, p. 22, 2013.
[25] Hibernate. (2014, Jun. 29). Object/grid mapper (OGM) [Online]. Available: http://hibernate.org/ogm/
[26] D. Hiller. (2014, Jun. 28). Playorm: Orm for NoSQL with scalable SQL [Online]. Available: https://github.com/deanhiller/playorm
[27] Impetus. (2014, Jun. 28). Kundera: Object-datastore mapping library for NoSQL datastores [Online]. Available: https://github.com/impetus-opensource/Kundera
[28] E. J. O'Neil, "Object/relational mapping 2008: Hibernate and the entity data model (edm)," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2008, pp. 1351–1356.
[29] I. Konstantinou, E. Angelou, C. Boumpouka, D. Tsoumakos, and N. Koziris, "On the elasticity of NoSQL databases over cloud management platforms," in *Proc. 20th ACM Int. Conf. Inf. Knowl. Manage.*, 2011, pp. 2385–2388.
[30] mongoDB. (2015, Mar. 12). Java MongoDB Driver [Online]. Available: http://docs.mongodb.org/ecosystem/drivers/java/
[31] INC MongoDB. (2014, Jul. 7). Mongodb [Online]. Available: http://www.mongodb.org/
[32] Netflix. (2014, Jun. 28). astyanax - cassandra java client [Online]. Available: https://github.com/Netflix/astyanaxs
[33] NoSQL. (2014, Jun. 29). Nosql your ultimate guide to the non-relational universe [Online]. Available: http://nosql-database.org/
[34] Nate McCall (zznate) Patricio Echague (patricioe). (2014, Jun. 29). Hector - a high level java client for apache cassandra [Online]. Available: http://hector-client.github.io/hector/build/html/
[35] J. Pokorny, "NoSQL databases: A step to database scalability in web environment," *Int. J. Web Inf. Syst.*, vol. 9, pp. 66–82, 2013.
[36] T. Rabl, S. G. Villamor, M. Sadoghi, V. M. Mulero, H.-A. Jacobsen, and S. Mankovskii, "Solving big data challenges for enterprise application performance management," *Proc. VLDB Endowment*, vol. 5, no. 12, pp. 1724–1735, 2012.
[37] A. Rafique, S. Walraven, B. Lagaisse, T. Desair, and W. Joosen, "Towards portability and interoperability support in middleware for hybrid clouds," in *Proc. IEEE INFOCOM CrossCloud Workshop*, 2014, pp. 7–12.
[38] M. Scavuzzo, E. Di Nitto, and S. Ceri, "Interoperable data migration between NoSQL columnar databases," in *Proc. 18th Int. Enterprise Distrib. Object Comput. Conf. Workshops Demonstrations*, 2014, pp. 154–162.
[39] R. Sellami, S. Bhiri, and B. Defude, "Odbapi: A unified rest API for relational and NoSQL data stores," in *Proc. IEEE Int. Congress Big Data*, 2014, pp. 653–660.
[40] Spring. (2015, Jun. 29). Spring data [Online]. Available: http://projects.spring.io/spring-data/
[41] U. Storl, T. Hauf, M. Klettke, and S. Scherzinger, "Schemaless nosql data stores object-NoSQL mappers to the rescue?" in *Proc. 16th Conf. Database Syst. Bus., Technol. Web*, 2015, pp. 579–600.
[42] T. Sun and X. Wang, "Research on heterogeneous data resource management model in cloud environment," *Int. J. Database Theory Appl.*, vol. 6, no. 1, pp. 141–152, 2013.
[43] AppScale Systems. (2014, Jun. 30). The open source implementation of google app engine - take your apps everywhere [Online]. Available: https://github.com/AppScale/appscale
[44] B. G. Tudorica and C. Bucur, "A comparison between several NoSQL databases with comments and notes," in *Proc. IEEE 10th Roedunet Int. Conf.*, 2011, pp. 1–5.
[45] J. S. van der Veen, B. van der Waaij, and R. J. Meijer, "Sensor data storage performance: SQL or NoSQL, physical or virtual," in *Proc. IEEE 5th Int. Conf. Cloud Comput.*, 2012, pp. 431–438.
[46] P. van Zyl, D. G. Kourie, and A. Boake, "Comparing the performance of object databases and ORM tools," in *Proc. Annu. Res. Conf. South African Instit. Comput. Scientists Inf. Technol. IT Res. Develop. Countries*, 2006, pp. 1–11.
[47] W. Vogels, "Eventually consistent," *Commun. ACM*, vol. 52, no. 1, pp. 41–44, Jan. 2009.
[48] D. Washusen. (2014, Jun. 29). A java library for accessing the cassandra database [Online]. Available: https://github.com/s7/scale7-pelops

Ansar Rafique is a PhD researcher at the iMinds-DistriNet, a research group within the Department of Computer Science of KU Leuven, Belgium. In 2013, he obtained MS degree in Computer Science from Uppsala University, Sweden. His research interests include cloud computing, distributed systems, big data, and NoSQL systems. He is particularly interested in issues concerning cloud data storage and efficient data management.

Dimitri Van Landuyt is a research expert in Software Engineering at iMinds-DistriNet, the Distributed Systems research group of the Department of Computer Science of KU Leuven, Belgium. Dimitri obtained his PhD degree in 2011 and focuses his current research on applying and validating established software engineering principles to cloud computing, more specifically in the context of Software-as-a-Service applications.

Bert Lagaisse is industrial research manager at the iMinds-DistriNet research group in which he manages a portfolio of applied research projects on cloud and security middleware in close collaboration with industrial partners. He obtained his MS and PhD in 2003 and 2009 from KU Leuven, Belgium. Bert is experienced with industrial valorization of research as well as the cross-fertilization between academic know-how and industrial expertise in multi-partner industrial projects. He has a strong interest in distributed systems, enterprise middleware, cloud platforms, and security services.



Wouter Joosen is full professor in distributed software systems at the Department of Computer Science of KU Leuven, Belgium. He obtained a PhD degree from KU Leuven in 1996. He has also co-founded spin-off companies of KU Leuven: Luciad, a company specializing in software components for Geographical Information Systems, and Ubizen (now part of Verizon Business Solutions), where he has been the CTO from 1996 till 2000, and COO from 2000 till 2002. His research interests are in cloud computing, focusing on software architecture and middleware, and in security aspects of software, including security in component frameworks and security architectures.