Object to NoSQL Database Mappers (ONDM): A systematic survey and comparison of frameworks

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

imec-DistriNet, KU Leuven
Celestijnenlaan 200A, 3001 Heverlee, Belgium

Abstract

Context: Software applications frequently interact with database systems to persist and retrieve objects. Object-mapping frameworks address (i) the bi-directional conversion of data between object and target database and (ii) provide a programmatic interface for querying and storing data. The rise of NoSQL databases poses challenges beyond object-relational mapping (ORM) frameworks to abstract from various data models and non-standardized API's, but also take into account the different database capabilities (e.g. unsupported query operators, data ordering).
Objective: A systematic survey study of existing Object-NoSQL data mapping (ONDM) frameworks. Specific focus is given to the level of abstraction of data and operations to multiple database technologies, as a means to limit vendor and technology lock-in and an enabler for multi-store and polyglot architectures. Additional attention is paid to mapping strategies that are specific to NoSQL databases (e.g. object embedding, schema flexibility).
Method: A systematic search methodology identifies all relevant mapping frameworks (in total 342 frameworks). Subsequently, a subset of ONDM frameworks is selected and systematically compared in terms of criteria of: database support, interface and query functionality, architecture and software coupling. Secondly, we provide an in-depth comparison of object-oriented mapping strategies for classes, inheritance, relationships, and attribute types to NoSQL data models.
Results: ONDM frameworks are most prevalent in Java, Ruby, Python, and overall 54 frameworks support multiple NoSQL databases. Interfaces are frequently standardized and commonly feature a uniform query language and even native DB query mapping. However, database portability may be hindered due to non-uniform abstractions. As for mapping strategies, current frameworks do not fully exploit NoSQL modeling potential, such as (i) the embedding of relationship data within referring objects' records, (ii) mapping at the individual object-level vs. class-level, and lacking (iii) collection normalization despite being supported for associations or when using relational databases.
Conclusion: The study consolidates knowledge on available ONDM frameworks, and applied object-document, object-graph, and object-column mapping patterns. The study can guide practitioners in framework selection, and pinpoints areas of future development and research in this domain, most notably towards improved support for flexible, NoSQL-aware mapping strategies.

Keywords: Object-NoSQL mapping, Object database mappers, NoSQL mapping patterns, NoSQL abstraction

1. Introduction

Many contemporary software applications involve object-oriented programming languages and a relational database. The object-oriented paradigm features concepts such as objects, classes, attributes, specialization or inheritance, and associations, whereas the relational database paradigm involves tables consisting of records, columns with data, primary keys and foreign keys. When objects are stored in a relational database, these object-oriented concepts have to be translated into the database paradigm [1, 2, 3, 4], and vice versa when they are retrieved again from the database.

This undertaking is not straightforward due to the so-called *object-relational impedance mismatch problem* [5] which is an umbrella term for a number of technical and conceptual issues of bridging both paradigms. For example, many alternative strategies exist to map associations and inheritance to database tables [3, 4], each with their own benefits and drawbacks. The simple act of querying also presents the possibility of in-

termixing database query languages within the application language, which leads to database vendor or technology lock-in, and development maintainability issues when the database schema evolves [6]. To address these issues, object-relational mapping (ORM) frameworks have been created and are used extensively in practice [4]. These frameworks (i) handle the bi-directional conversion between objects and the relational data model, (ii) manage persistence to the target database, and (iii) provide software developers with a uniform data access interface to store and query objects programmatically. As such, these frameworks aid in decoupling applications from database-specifics which benefits maintainability, and allows developers to port applications to different databases more easily.

Although it has been dominated for many years by the relational paradigm, the landscape of database systems today has evolved drastically with the rise of NoSQL databases [7]. NoSQL databases provide dedicated support for a wide range of fundamentally different data models, many of them focusing on attaining levels of horizontal and elastic scalability, and schema flexibility that can not easily be accomplished in relational databases [8, 7, 9]. NoSQL databases can be categorized according to their supported data model, and are typically categorized in document stores, graph stores, column stores and key-value stores [8]. The database functionality provided is closely tied to the data model used and ranges from simple insert and read operations on key-value pairs, to graph traversal, or even analytical queries on large data sets (e.g. MapReduce) [9]. As different databases commonly address specific use cases, there are currently over 225 NoSQL databases in existence [10].

In analogy to ORMs, Object-NoSQL database mapping (ONDM) frameworks provide a uniform interface and uniform data model for various NoSQL databases [11]. Again, software becomes more maintainable as database specifics and the risk of technology or vendor lock-in are avoided. However, given the larger discrepancy between source and destination data models, the issues of impedance mismatch in general become more stringent as it is in many cases unclear which data mapping strategies are most efficient at bridging the gap between source and destination data models [12]. The emergence of ONDM platforms furthermore is fairly recent and different platforms are at varying levels of maturity which makes the adoption of an ONDM platform for practitioners a non-trivial endeavor.

This study presents the results of an in-depth technology survey involving 342 currently-existing and emerging object-mapping frameworks (ORM, ONDM), and systematically compares the identified object-NoSQL mapping frameworks on broad aspects of: database support, programming interface (including query support), and their architectural model. Secondly, specific attention is paid to how these frameworks implement object mapping to the destination NoSQL data models by performing an in-depth analysis of the implemented object-NoSQL mapping strategies.

This study gathers and consolidates the knowledge on object-NoSQL mapping frameworks and as such its direct contributions are to (i) aid practitioners in their choice and use of a framework, (ii) further the development of these frameworks, and (iii) steer model-driven research in the domain of NoSQL mapping patterns.

In addition, recent evolutions have seen the emergence of ONDM platforms that support *multi-storage*, hybrid or federated storage architectures, i.e. storage configurations in which different database technologies and providers are combined [13, 14, 15, 16, 17, 18, 19, 20]. The success of such systems depends on the quality of the algorithms to map objects to and from different database target models efficiently, and as such the results of this survey aid in highlighting the research gaps towards this vision of integrated and adaptive multi-cloud data management.

The remainder of this paper is structured as follows. Section 2 provides the necessary background on NoSQL databases, the object-relational impedance problem, and object-NoSQL mapping patterns. Subsequently, Section 3 discusses the survey study design and applied research methodology. Sections 4 details the search results and framework selection. Next, Section 5 provides a systematic feature comparison of the frameworks, whereas Section 6 compares applied object-NoSQL mapping strategies.

Section 7 discusses the findings and provides guidelines for framework and mapping pattern selection, and also describes future research challenges. Section 8 discusses related work , whereas Section 9 concludes.

2. Background

Section 2.1 first provides the necessary background on object-relational mapper (ORM) frameworks and the object-relational impedance mismatch problem. Then, Section 2.2 shortly outlines the different NoSQL database technologies and data models. Next, Section 2.3 explains the NoSQL commonalities at the basis of novel object-mapping strategies and the challenges. Section 2.4 discusses the emergence of Object-NoSQL mapper platforms (ONDMs) and motivates this survey.

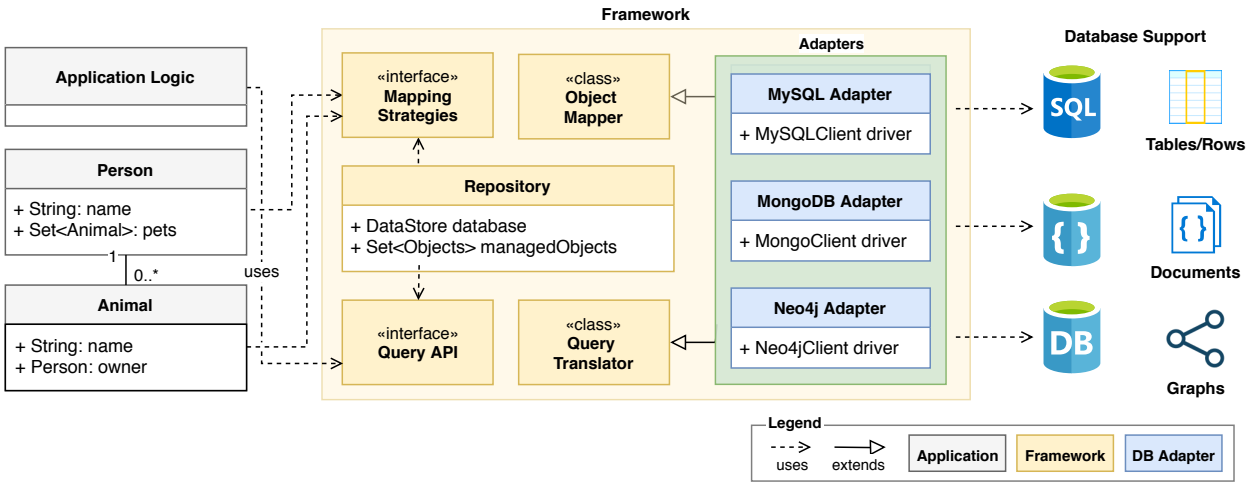

Figure 1: Object-database mapper architecture based on the repository pattern.

2.1. *Object to database mapping*

The object-oriented paradigm features concepts such as objects, classes, attributes, specialization or inheritance, and associations, whereas the relational database paradigm involves tables consisting of records, columns with data, primary keys and foreign keys. When objects are stored in a relational database, these object-oriented concepts have to be translated into the database paradigm [1, 2, 3, 4], and vice versa, when they are retrieved again from the database.

The discrepancy between source model (objects) and target model (e.g. tables) gives rise to the so-called *object-database impedance mismatch problem* [5]. Specific to relational databases, four major problems categories have been pinpointed in terms of the mapping strategy: these mappers have to deal with paradigmatic differences in terms of (i) inheritance (or specialization), (ii) relationships (or associations), (iii) complex data types, and (iv) embedded classes [1, 5].

Due to the complexity involved in this, dedicated mapper frameworks are commonly used in contemporary software systems to decouple application logic from database specifics. To this end, object-relational mapping (ORM) frameworks provide a uniform data model and interface, which shield developers from the complexity involved in persisting and retrieving objects to and from the database.

Object mapping frameworks are commonly structured according to one of two alternative architectural patterns: (i) the active record pattern and (ii) the repository pattern [21]. The active record pattern involves extending domain objects with mapping and query logic by specializing from an ORM class. As a consequence, individual objects more closely represent the database record(s). In contrast, the repository pattern separates domain classes from access logic by using a repository to persist and query objects, and thus, objects are decoupled more strongly from their representation in the database. A prevalent example of the repository pattern is the standardized Java Persistence API (JPA) [22] for data persistence and retrieval in Java that relies on an `EntityManager`.

Figure 1 depicts a generic model of a object mapping framework based on the well-known repository pattern. It depicts a client application that defines a number of domain classes. Upon persistence, domain objects are forwarded to the framework's `Repository` and then to database-specific `DB Adapters` that implement an `Object Mapper`. Similarly, query operations are exposed by the `Repository`'s uniform interface which is implemented by each supported database via an adapter. The database adapter implements a `QueryTranslator` that handles translation from the uniform query language to the native database query language.

The architecture and applied object to database mapping patterns, which are well-studied for relational databases, are challenging in the emerging and diverse context of NoSQL databases.

2.2. *NoSQL databases*

Relational databases are highly focused on structured data adhering to a strict schema, and preserving data consistency through transactional properties (ACID) [8]. In practice, application data is often too heterogeneous to model for in advance, or does not remain compliant with strict schema(s) over time.

The lack of flexibility in relational databases leads to complex issues of schema evolution and data migration. In addition, scalability is hindered by transactions and properties of Atomicity, Consistency, Isolation and Durability (ACID) [8, 7, 9].

NoSQL databases represent an alternative to relational databases and generally aim to tackle the drawbacks of relational databases in terms of schema rigidity, but also the limitations to horizontal and elastic scalability [23, 24, 25] and support specific data models (e.g. JSON documents). In practice, these database systems have non-standardized interfaces, in part due to the heterogeneity in data models.

Today, we observe over 225 distinct NoSQL database technologies, which are broadly categorized in terms of the supported data model: (i) document stores, (ii) graph stores, (iii) key-value stores and (iv) wide-column stores [10, 9]. Wide-column stores such as Apache Cassandra [26], Apache HBase [27] and Google Bigtable [28] resemble relational databases due to their table structure. Key-value and document stores allow a flexible data format such as JSON. However, key-value stores typically abstract from the type of value stored, therefore featuring simple query functionality [9]. Graph stores focus on graph data and specific query operations of traversal and clustering.

Next to the data model, the API and query language are generally non-standardized and these provisions are typically more limited than traditional query languages such as the SQL. In practice, we now have dedicated storage systems tailored to specific use cases. Time series data is for example stored into database systems which can order the data, such as the column stores Apache Cassandra [26] and HBase [27]. MongoDB [29] on the other hand is a document store that does not impose an ordering of the records and is thus unsuited for such a use case.

2.3. Object-oriented mapping to NoSQL databases

In NoSQL databases, especially aggregate-oriented databases such as document stores, the mapping of object-oriented concepts is broadened by two common aspects [9]:

- **Aggregate data model**: The object can be mapped entirely to, for example, a single record formatted in JSON [30, 31, 32].
- **Flexible schema**: Individual records do not necessarily have to comply with a rigid table structure and its schema [9]. Objects can be mapped individually at the object-level instead of solely at the class-level. Additionally, object records from different classes can be grouped in the same table.

In each next section, we explain how these concepts go beyond traditional object-relational mapping strategies, which nonetheless remain applicable.

2.3.1. Mapping object attributes and collections

Attributes have to be mapped, together with their data type. Data types such as a `String`, `TimeStamp` and `Integer` have counterparts in database systems. However, attribute types such as `Set`, `List`, and `Array` do not have database counterparts in relational databases. As such, an array has to be either (i) **embedded** by: serialization within a single column (M_{Field}), into multiple fields ($M_{Fields+}$), or (ii) **normalized and referenced** into additional rows (M_{Rows+}), or in a new join table (M_{Table}) [33]. We will later refer to these tactics generally as embedding or referencing. Tactics vary according to the selected target database technology.

Graph nodes and edges. In the case of graph stores, there is no concept of tables, instead there are nodes and edges (i.e. relations). An object's key for example can be modeled as a node, whereas its attributes are typically node properties or alternatively modeled as edges. Typically, only relationships are modeled as edges featuring labels with additional information on the relation. Furthermore, when mapping ordered data the order can be preserved at the edge with a property or within the embedded property.

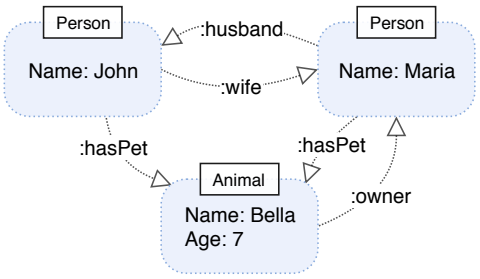

Figure 2: Graph vertices and edges, both containing properties.

Figure 2 depicts a data model in Neo4j [34] comprised of nodes and directional edges between nodes that represent relationships. The nodes, or vertices, and edges can store properties as key-value pairs.

Document stores and embedding. In aggregate-oriented databases such as document stores, the record is a flexible JSON structure. Complex data types such as a `List` can be mapped as an array and embedded

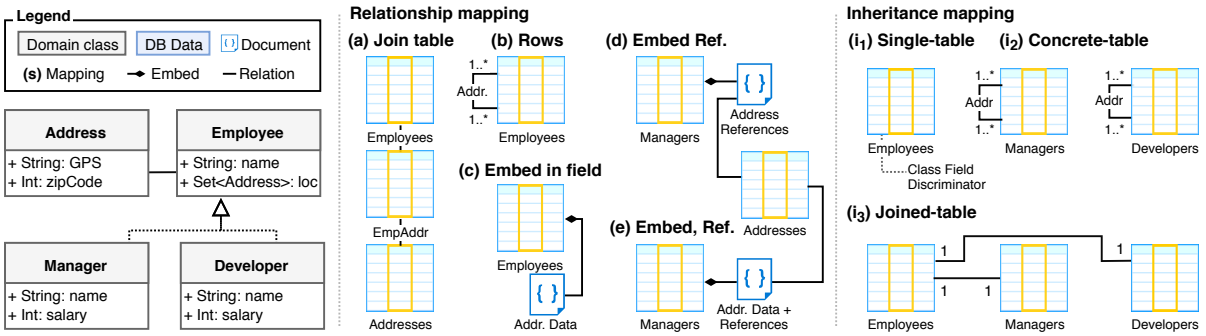

Figure 3: Object-NoSQL mapping patterns for relationships and inheritance.

within the record (M_{Embed}). The strategy is comparable to storing the array into multiple columns ($M_{Fields+}$), and naively comparable to serialization (M_{Field}). In contrast to relational databases, the schema does not need to be modeled in advance and can grow dynamically as an array or map of fields and values, without affecting other record structures already present in the same table.

Referencing and normalizing collections. However, embedding is undesirable when the array in a document grows too large or when it a subset of data is updated frequently independent of the parent. In certain database technologies, an update to a subset of record data may overwrite the entire, potentially large, record [35]. In such cases, collections are best mapped using relational tactics into additional rows (M_{Rows+}) in the same table or a join table (M_{Table}).

2.3.2. Mapping object relationships

Attribute types can also be of a (user-defined) class type, in which case there are two scenarios for the attribute object: either (i) the object is owned and used solely by the parent object, or (ii) the object exists independently and represents a relationship.

References and join tables. Relationships have no counterpart in relational databases and records are typically associated with references on foreign keys [33]. In the case of one-to-one relationships, either record can store a reference. Similarly for one-to-many relationships, the many-side can store the reference in a single field or otherwise a join table is created.

Embed references. In NoSQL databases featuring a flexible data model and schema, records can store a nested array of references at either side of the relationship. Such tactics may be beneficial for performance by avoiding a potential row scan on the one-side table for foreign keys matching its primary key.

Embed relationship data. Since references can be embedded at either entity, theoretically so can parts of the relationship data. Moreover, the referred data can still exist independently of the referring object, albeit with parts of its data copied and embedded within records that refer to it. Such tactics benefit read performance, however negatively affect consistency and update operations.

Polymorphic associations. Additionally, objects of various class types, typically similar or connected in some manner, can be mapped to the same table (i.e. record collection). On record retrieval, the framework has to determine which object type it belongs to using discriminator fields.

2.3.3. Mapping classes and inheritance

In object-oriented programming languages, classes are structured hierarchically, each class having a parent class and various child classes. Three common relational strategies exist for persisting inheritance relationships: *single-table* (an entire inheritance tree is encoded in one table), *class-table* (a table for each class), or *concrete-table* (one inheritance path is one table) [4, 22]. Figure 3 depicts these strategies applied to an object model. In case of the *single-table* strategy, various class records exist in the same table and are differentiated by a discriminator column. JDO [36] features four inheritance mapping strategies; two similar to *class-table* and *concrete-table*; and two novel strategies *superclass-table* and *subclass-table*. The latter ei-

ther maps the class information to fields in the parent's table, or the child's table.

Trade-offs exist between each tactic. For example, *joined-table* leads to costly join queries between tables. In case of the *single-table* strategy, a search query that involves a specific child type leads to a scan of the entire table. Among the inherited fields are also associations or collections, which the child class in turn can map differently than the parent.

2.3.4. Novel object mapping strategies

In case of object mapping to databases featuring a flexible data model and schema, we summarize the novel object mapping strategies as:

S1: Referencing or embedding. Relationship and collections can be aggregated and embedded within the parent document as (i) references, or (ii) embedded copies of the referred data, or (iii) subsets through partial embedding of the referred data, and this either in a uni- or bi-directional fashion [25, 31, 37, 32]. Independently, relationships can be stored in additional records and tables, and referred to using foreign keys or join tables. Embedding relationship data or either storing it independently and referencing to it are complementary tactics.

S2: Class-level or object-level mapping. Object mapping strategies can be devised on a per-object level basis versus traditionally at the class-level. For example, an object can initially embed a nested array, but when the array grows too large, that record can be refactored so that the array is stored in separate records. As a consequence of schema flexibility, a single object record can change independently of existing record structures.

2.4. Motivation: Object-NoSQL database mappers (ONDM)

Similarly to ORMs, ONDM frameworks provide a uniform interface to persist and retrieve objects from at least one or more NoSQL databases, and these platforms have been emerging to address the mapping complexities discussed in the previous section.

There is however large heterogeneity among ONDM frameworks. Some ONDMs frameworks such as EclipseLink [38] and Hibernate ORM [39] are historically ORMs that have been extended with support for some NoSQL databases, and in general, such ONDM frameworks are aligned strongly to existing reference architectures and best practices for ORMs. Other ONDMs such as Docb [40] and KEV [41] in turn focus on specific types of NoSQL databases.

From a practitioner's point of view, the selection of a suitable ONDM framework (in accordance to application requirements) is crucial. Yet to our knowledge, an extensive comparative survey of existing and emerging ONDM platforms is currently lacking and this hinders their adoption in practice. Furthermore, it as-of-yet unclear how well existing mapping frameworks have adapted to make use of the potential presented by NoSQL technologies in terms of the data model and schema flexibility (e.g. strategies **S1**, **S2**). Additionally, database functionality in NoSQL is often simplistic and features such as record ordering (e.g. MongoDB) or query operators can be missing, and this then needs to be addressed by the framework.

Secondly, a more thorough and in-depth understanding of the current state of these systems allows defining a research and development roadmap for the improvement of ONDMs and their underlying mechanisms.

3. Research methodology

This survey study is designed in function of two research objectives that are defined in Section 3.1. The study design, shown in Figure 4, is structured according to the guidelines for systematic literature review in Software Engineering by Kitchenham et al. [42, 43] and features a search activity to identify relevant frameworks, explicit inclusion and exclusion criteria to scope the study, and involves systematic, in-depth comparison. The search methodology is described in Section 3.2, whereas Section 3.3 discusses and motivates the adopted inclusion and exclusion criteria. The identified and selected ONDM frameworks are discussed in Section 4.

The in-depth comparison with respect to both research goals is presented in Sections 5 and 6 respectively.

3.1. Research objectives

The objectives of this study are twofold, as described below:

RO1 Gather and consolidate knowledge on existing Object-NoSQL database mapping (ONDM) frameworks, and compare them in terms of: database support, programming interface and query languages, architecture, and coupling between application and both framework and database.

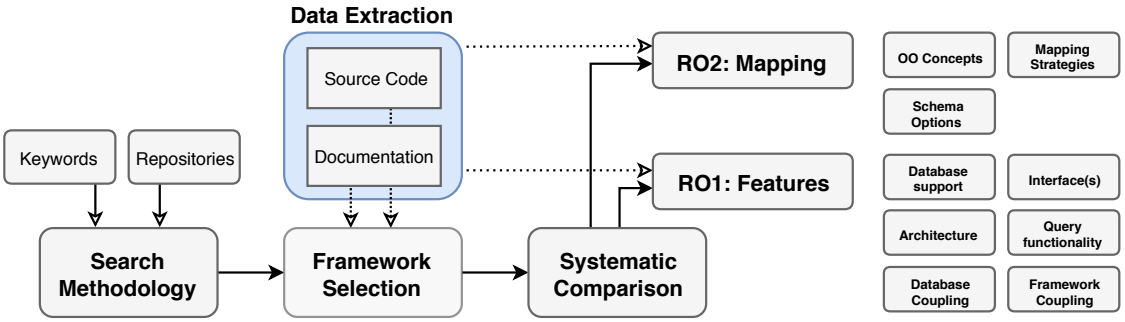

Figure 4: Graphical overview of the design of this study.

RO2 In-depth comparison of ONDM frameworks in terms of the implemented mapping strategies for object-oriented concepts of attributes and data types, relationships, classes and inheritance, with specific attention to NoSQL-specific mappings.

3.2. Search methodology

We adopt a search strategy in a similar trend to a systematic literature review [43]. In accordance with the guidelines set forward by Kitchenham et al. [42], the search methodology consists of (i) determining appropriate search keywords, (ii) the identification of resources and repositories to be searched, and (iii) establishment of framework inclusion and exclusion criteria.

3.2.1. Search keywords

In order to identify the state-of-the-art of object-mapping frameworks, we first establish a broad search query defined in Table 1 consisting of (i) common synonyms for object-mapping frameworks and (ii) a set of relevant object-oriented programming languages (OOPLs). The initial search also includes object-relational mappers (ORMs) as they may have developed support for non-relational databases over time.

In terms of synonyms, object mappers are also referred to as 'POJO mappers' (plain old Java object mappers) in the context of Java. Similarly, for the C++ program language, the term 'POCO mapper' is used. Another synonym for object-data mapper is a data access object (DAO); this is a reference to the data mapper design pattern [44]. Other variations exist such as a database abstraction layer (DAL), and persistence framework.

The search scope is limited to the twelve most popular object-oriented programming languages (OOPLs) listed in Table 2[1]. The OOPLs are ranked by an average index ranking from multiple sources such as GitHut [45] and the IEEE Spectrum Programming Language Ranking [46].

3.2.2. Resources and repositories searched

The search query of Table 1 is executed in a number of search engines and software repositories. Additionally, we manually extract the list of data access frameworks discussed in related academic literature[2].

The search also covers documentation of the most popular NoSQL vendors for mentioned object mappers and this is driven by the DB Engines ranking [50].

In terms of software repositories, each programming language typically has a number of community software repositories for package management, such as npm which is a packet manager for NodeJS [51]. Perl frameworks are listed in the "Comprehensive Perl Archive Network" [52], and Java frameworks in Maven repositories. Community members also provide their own collection of software packages, such as the Ruby Toolbox website for Ruby [53].

3.3. Framework selection criteria

To refine the scope of our study, we explicitly define a set of inclusion (*I*) and exclusion (*E*) criteria:

I1 The framework has object-mapping capabilities to relational *or* non-relational (NoSQL) databases and is publicly available.

E1 The framework has object-mapping functionality to only a single DB technology (i.e. a DB wrapper).

E2 The mapping framework support *less than* 3 NoSQL databases.

[1] This is similar to the approach taken in the ORM framework survey by Torres et al. [4].

[2] Since the academic research on ONDM frameworks is relatively scarce, the search strategy primarily focuses on identifying existing software libraries. However, academic ONDMs are discussed later in Section 7.1.1.

AND	**OR**	NoSQL, Non-relational, Relational database, RDBMS, SQL, database, `NULL`
	OR	object-relational mapping, object-data mapper, object-data mapping object-relational mapper, object mapper, data mapper, POJO mapper, POCO mapper, data access platform, data access middleware platforms, data access object, DAO, data access layer, database abstraction layer, persistence framework, ORM, ONDM, ONM
	OR	Java, Python, JavaScript, PHP, C#, C++, Ruby, Swift, Scala, Go, Objective-C, Perl, `NULL`

Table 1: Search query for object database mappers.

Programming Language	*TIOBE [47]*	*PYPL [48]*	*IEEE [46]*	*GitHut [45]*	*RedMonk [49]*	*Rank*
Java	1	1	3	3	2	1
Python	4	2	1	2	3	2
JavaScript (Node.JS)	6	4	7	1	1	3
PHP	9	3	8	6	4	4
C#	5	5	5	10	5	4
C++	3	6	4	7	6	5
C	2	7	2	8	8	6
Ruby	11	12	11	4	7	7
Swift	12	10	10	13	12	8
Shell	-	-	13	11	11	9
Scala	-	15	12	12	10	10
R	8	8	6	30	13	11
TypeScript	-	14	-	9	16	11
Go	19	19	9	5	14	12
Objective-C	16	9	21	16	9	13
Perl	10	18	14	23	15	14

Table 2: Programming languages and ranking and popularity per index from 2018. Languages printed in bold are included.

E3 Databases are only supported from a single NoSQL domain (e.g. only document stores).

E4 The framework ceased development in the last 2 years or shows insufficient functional features or minimal mapping strategies (e.g. no support for associations).

3.4. RO1 Comparison criteria

The first research objective (**RO1**) aims to consolidate knowledge on existing ONDM frameworks by a systematic comparison in terms of criteria of interest: **C1**: database support, **C2**: interface and query functionality, **C3**: ONDM architecture and **C4**: the degree of coupling to the framework or database.

The following comparison criteria are interesting but considered out-of-scope: interface and database functionality mismatches, support for specific functional aspects (e.g. transactions, caching), performance and scalability aspects, and multi-store and polyglot support. Analyzing the interface its operations versus each supported database technology's API would lead to too many dimensions for comparison. In addition, object-mapping performance is already evaluated in our previous study [11].

In the following subsections, we further define each criterion, motivate its importance, and establish a basis for comparison.

3.4.1. C1. Database support

Database support is arguably the main criterion of relevance for software developers and practitioners. Database technologies have to meet with application requirements, which are identified during the requirements elicitation and architectural design phases. Such design choices can be postponed to a later stage with the use of ONDM frameworks. However, in order to do so, one must be confident that the desired database technologies and features are *fully* supported by the framework. Database support in ONDMs can vary from a single category of NoSQL databases (documents) to multiple classes, to even relational databases.

Support for relational databases is a selling point

in a migration context, or for applications that require the transactional properties of proven existing relational database technologies. When a relational database is supported, often an entire set of different vendors are supported due to the standardization of SQL.

In certain frameworks it is possible to use relational and NoSQL database technologies interchangeably with a single interface. Such an application scenario is referred to as a *multi-store architecture* or *polyglot persistence* [54]. Generally speaking, ONDM frameworks that support many heterogeneous databases are either (a) highly extensible and mature frameworks, or (b) simply implement a very narrow abstraction.

Specifically for **C1**, we compare frameworks in terms of the number of relational and NoSQL databases supported, and we distinguish between different classes of NoSQL databases.

3.4.2. C2. Interface support and query functionality

Object-mapping frameworks provide a single or multiple programming interface(s) to store and query objects in the supported database systems. These interfaces provide access to the database and/or offer a query language for searching objects.

The abstraction should feature at minimum create, read, update and delete (CRUD) operations. In a best case scenario, the framework interface provides access to all native database operations, preferably through abstractions supported by the framework.

Standardized interfaces. Support for a standardized interface implies a large set of features according to its specification (e.g. the Java Persistence API (JPA) [22]). Standardized interfaces such as JPA reduce the learning-curve for developers and allow application migration between existing frameworks abiding by the same standards. JPA features a uniform domain model to describe classes with annotations on attributes, relationships and inheritance. These annotations describe the desired object to database mapping strategy.

Secondly, JPA provides a programmatic interface to store and search objects, for example the `findById` method. In addition, JPA features a Java Persistence Query Language (JPQL) which provides a uniform query language similar to SQL and can be implemented on top of heterogeneous database query languages. Alternatively, Java also provides the standardized persistence interface Java Data Objects (JDO) [36]. In .NET there are standardized API's such as LINQ and the Microsoft Entity Framework.

Query functionality. The query functionality in the interface is typically provided through one of the following approaches: (i) programmatic operations on object identifier (id), (ii) query object builders using selection criteria, (iii) dynamic query operations (e.g. `findByName`), (iv) abstracted query languages such as JPQL, and (v) support for native database query languages.

Specifically for **C2**, we compare the frameworks in terms of their interface support (standardized, or non-standardized), and the extent to which the aforementioned query approaches are supported.

3.4.3. C3. Architectural patterns

The architecture of the mapper framework has a substantial impact on a number of attributes, such as the mapping flexibility of the framework or its extensibility. In addition, the architectural pattern dictates to a degree the manner in which domain classes are modeled. As discussed in Section 2.1, the architecture of an object-mapping framework commonly follows either the active record or the repository pattern. Frameworks that implement the active record pattern place data access and query logic within the domain classes. In contrast, the repository pattern introduces a mediator between domain objects and DB adapters.

3.4.4. C4. Framework and database coupling

Figure 5 illustrates the implications on domain classes of different ONDM architectures. The architectural pattern is not just a stylistic choice but impacts the portability of the application code to another framework or database. This is illustrated in Figure 5b with the use of database-specific annotations.

The framework architecture further impacts the degree of coupling between domain classes in application code and the framework. For example, the active record pattern ties data access operations to the domain objects, while a repository can separate domain objects and data access logic.

We investigate the degree of coupling introduced by the ONDM framework, between (i) the application and the ONDM framework, and (ii) the application and the database. This distinction is illustrated in Table 3. As shown in the final column of Table 3, we apply a score-based ranking (ranging from $--$ to $++$) of the frameworks to express the introduced degree of coupling to the framework and database.

Framework coupling. Framework coupling ties domain classes to the framework class through inheritance, proprietary annotations, or custom interfaces.

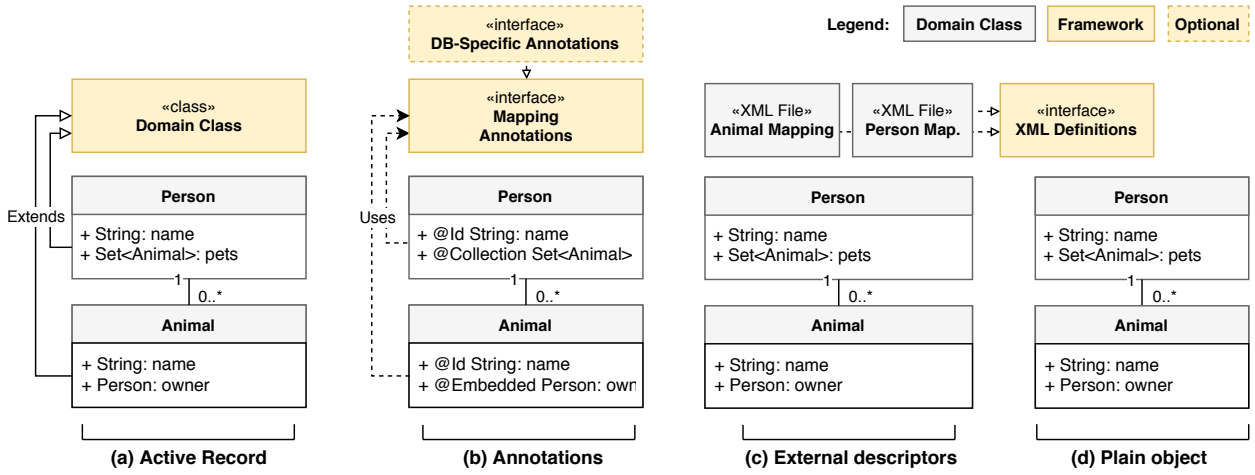

Figure 5: Mapping descriptions for domain classes, ordered by framework coupling (high → low).

	Domain	Operations	
ONDM	Plain objects	Query language	−−
	Annotations	Standardized	↓
	Inheritance ONDM	Custom interf.	++
DB	Uniform spec.	Uniform interface	−−
	Annotations per DB	Operations per DB	↓
	Map. format per DB	Interface per DB	++

Table 3: Low to high coupling between the application domain or operations, and respectively the framework or database.

Coupling between application and framework exists between domain classes and framework, and between application and the framework interface. Framework libraries have specific domain class annotations, mapping specifications (e.g. in XML), or interfaces.

Domain classes can be tied to classes from the framework, for example by inheriting functionality to deal with storing and querying data. Inheritance from framework classes introduces tight coupling with the framework. Reflection-based approaches in which specific annotations are introduced that describe how class elements should be mapped introduce a lesser degree of coupling. Frameworks that introduce the least degree of coupling are typical plain old Java object (POJO) mappers which require no annotations.

At the interface level, coupling is induced due to the framework's interface or query language which can either be custom or rely on standardization. In the best case, query languages are standardized (e.g. SQL or JPQL). Functionality can also be provided through repository classes such as a `CRUDRepository` which extends the domain class and inherits framework functionality to manipulate the object.

This type of coupling ties application code to a specific framework (lock-in) and hinders the adoption of a different ONDM later on.

Database coupling. In terms of database coupling, ONDM frameworks can offer different interfaces depending on the selected DB or even specific domain model annotations. In the best case, each framework provides the same interfaces and descriptors regardless of the selected database.

Figure 6 illustrates database coupling within a framework when subsets of the interface are provided per database technology. In the illustrated example, the `Neo4j Adapter` does not implement the `Document Query` interface. Such scenarios are typically caused by the heterogeneity between NoSQL database APIs and data models, in which case the available interface functionality may depend on the configured back-end. Therefore, when working with multiple database technologies one should be aware of the employed query operators since they might not be (natively) supported by the database and consequentially the framework.

Next to interface operations, the mapping behavior or employed annotations can differ per database technology. As such, the interface and domain coupling can introduce limitations on database interoperability and portability. When porting the application to a different database, this can lead to having to rewrite application logic pertaining to data access operations, object annotations and mapping strategies.

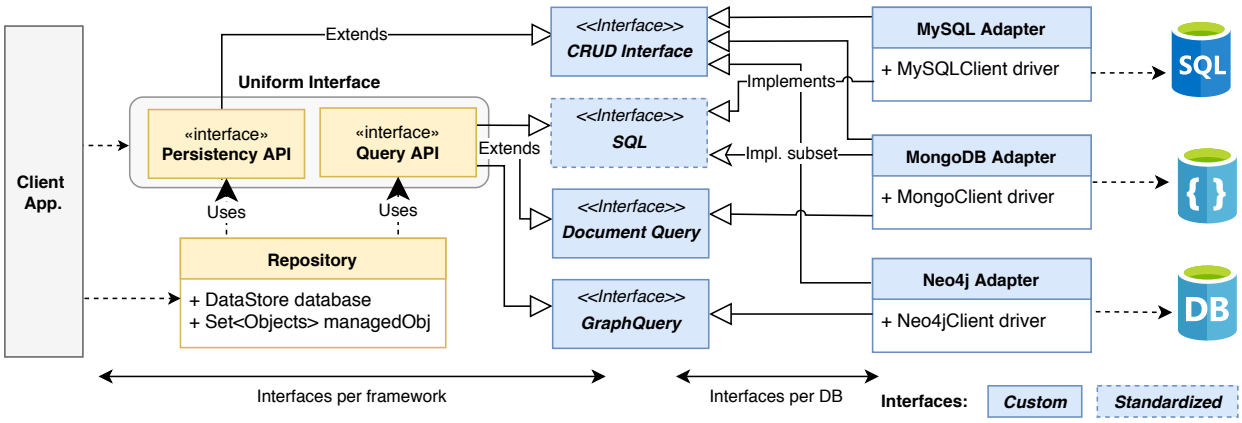

Figure 6: DB and framework coupling by respectively non-uniform and non-standardized interfaces.

3.5. RO2 Object mapping comparison criteria

We establish a comparison basis for **RO2** regarding the supported and implemented object-mapping strategies. These mapping strategies can include typical relational strategies but also NoSQL-specific mappings, and these may vary drastically per NoSQL technology class. Based on this reasoning, we conduct our comparison of mapping strategies on a per-data model basis, starting with aggregate-oriented databases, such as document and key-value stores, followed by object-graph mapping. Column-store mapping strategies are only briefly discussed, as the data model is highly similar to the relational database model.

For each object-oriented concept, we scan available mapping strategies from documentation and the source code of the ONDM frameworks.

Our comparison for **RO2** is mainly driven by two criteria: **C5.** mapping support for attributes, associations and inheritance relationships, and **C6.** awareness of the NoSQL data model. In the following subsections, we further define each criterion, motivate its importance, and establish a basis for comparison.

3.5.1. C5. Mapping attributes, associations and inheritance relations

We systematically compare object-mapping strategies starting with (i) attribute mapping, which can be complex data types, even user-defined classes or collections thereof, followed by (ii) relationship mapping, and (iii) inheritance mapping.

In terms of attribute and relationship mapping, we evaluate whether NoSQL-specific tactics are available such as embedding, or relational tactics such as referencing and normalizing relationship data (**S1**). We specifically investigate whether the framework offers the possibility to conceptually differentiate between relationships and embeddables (attributes uniquely owned by the object) in the domain model, and whether it can still embed relationships or reference "embeddables".

In terms of inheritance mapping, we compare support for common patterns (previously discussed in Section 2.3.3).

3.5.2. C6. Specialized mapping strategies.

This criterion assesses the degree to which the encountered mapping strategies are specific to the target database model.

We expect object-mapping strategies to be diverse for aggregate-oriented data stores, such as document and key-value stores, due to schema and data model flexibility over ORM (cfr. **S1** and **S2**).

Additionally, we expect object-mapping strategies towards graph and column stores to be similar to relational database mapping strategies due to respectively the relational nature of the data, and the tabular structure. For example, column stores such as Apache Cassandra [26] also enforce a strict schema.

In certain NoSQL databases the framework can hopefully decide between these tactics on a per-object level basis (**S2**), rather than typical at the class-level in ORM frameworks.

4. Framework search results and selection

This section discusses the results of the initial search and the selection process.

4.1. ONDM framework selection

The search strategy (Figure 7) has lead to the identification of a total 342 frameworks satisfying the inclusion criteria **I1** of supporting object-mapping functionality to a database.

The initial search set also includes frameworks originally classified as object-relational mappers (ORMs), since they may have developed NoSQL database support over time. The entire overview of all the 342 identified frameworks cataloged per programming language is available via [55, 56].

4.2. Exclusion of frameworks

We gradually apply the exclusion criteria defined in Section 3.3 to establish the main set of ONDMs for our survey.

As per **E1**, we exclude 35 frameworks that are wrappers to a single relational database (RDB mappers). Similarly, 112 frameworks that are wrappers for a single NoSQL database (NoSQL mappers) are excluded due to the **E2** exclusion criterion.

141 of the identified frameworks exclusively support object-relational mappings. These object-relational mappers (ORMs) are as such excluded from the survey study because of **E2**.

This first reduction phase yields a total of 54 frameworks. These are considered Object-NoSQL data mappers (ONDMs) since they offer support for more than one NoSQL database. We list these ONDMs per programming language in Table 4.

The second reduction phase involves assessing the extent to which the remaining 54 frameworks support at least three NoSQL databases (**E2**), and the development maturity (**E4**). As shown in Table 4, the **E2** criterion removes 18 additional frameworks, whereas **E4** removes 17 frameworks for which the main development activity has ceased since 2015. These are considered insufficiently mature or lacking of interesting functionality (e.g. no relationships). This phase yields 19 frameworks.

In the third and final reduction phase, we assess the extent to which the frameworks support databases from different database classes (**E3**). These ONDMs are considered particularly interesting as they support mappings towards fundamentally different data models.

Table 5 compares the set of 19 frameworks by supported databases to exclude those which do not support multiple NoSQL categories as per exclusion criterion **E3**. Although the framework Bass [57] supports two document databases and a single key-value database, it is still excluded per **E3** as these NoSQL categories are highly similar.

The resulting set of 11 frameworks is listed in the left-hand side of Table 5.

4.3. Selected ONDM frameworks

These 11 frameworks are heterogeneous in terms of the programming language and technology context. They either have origins in object-oriented languages such as Java, or scripting languages such as JavaScript and PHP, of which JavaScript supports objects but is considered classless, and the latter PHP which over time developed support for classes.

In terms of programming languages, Java frameworks are well-represented due to the wide-spread adoption of the Java Persistence API (JPA) [22]. JPA is implemented in for example Apache Gora [58], Impetus Kundera [59], DataNucleus [73], Hibernate OGM [60] and Spring Data [67].

In addition, frameworks exist for PHP, Node.JS, JavaScript and Ruby. However in the case of PHP, most web frameworks only have ad-hoc support for a few NoSQL databases. Many PHP frameworks rely on the well-known Doctrine ORM library [66]. For JavaScript, a single Javascript ORM named JS Data [62] was identified for front-end applications. This ORM framework supports a wide range of NoSQL databases. Back-end applications written in Node.JS can make use of frameworks such as Waterline [65]. Lastly, Ruby is represented by the Ruby Object Mapper (ROM) [64].

Although many languages such as C#, C++, Swift, and Go do not have any mature ONDM frameworks, individual NoSQL database drivers or wrappers are available. As discussed above, these are however considered out of scope of this survey study.

5. Feature comparison (RO1)

As discussed in Section 3.4, we attain the RO1 research objective by comparing the ONDMs in terms of the following comparison criteria: *database support (C1)*, *interface and query functionality (C2)*, *architecture (C3)*, and *framework and database coupling (C4)*.

5.1. Database support (C1)

Table 5 compares the frameworks in terms of database support for relational and NoSQL databases. Individual database support is listed for document, graph, key-value and column stores. Not included are NoSQL databases such as full-text search databases (e.g. ElasticSearch [74], Apache Solr [75]). Table 5

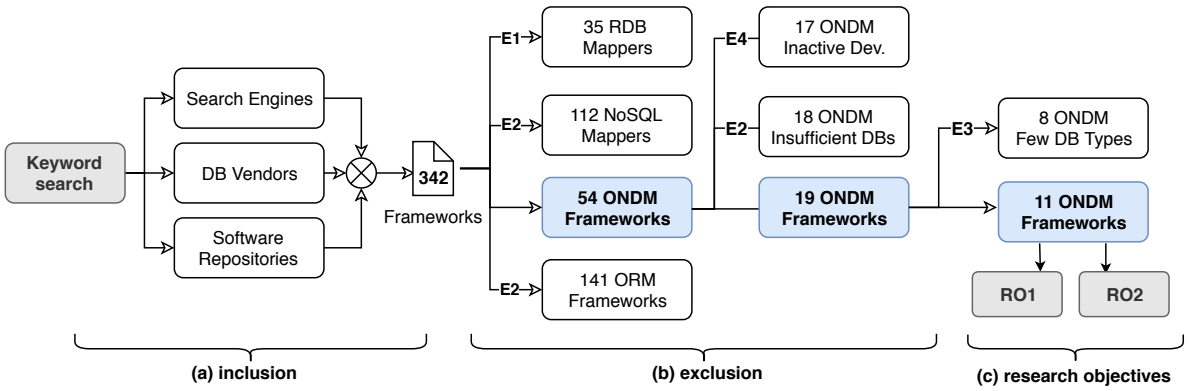

Figure 7: Framework selection process.

OOPL	**ONDM Frameworks** (n = 19)	**Excluded E2** (n = 18)	**Inactive E4** (n = 17)
Java	Apache Gora, Kundera, DataNucleus, EclipseLink, Eclipse JNoSQL, Spring Data, Hibernate OGM, GORM	Play Framework[1], Ebean[2]	Carbonado, Cumulus4j, Dasein, PlayORM, River framework
Python	KEV, pyDAL	Django[1], Docb[2]	
JavaScript	JS Data		Resourceful
Node.JS	Thinodium, Bass, Waterline, JS Data	Node ORM[2], TypeORM[2]	JugglingDB, Cleverstack, Node Document, Node NoSQL ODM, Osmos
PHP	Lithium, Yii framework, Doctrine	Phalcon, Zend framework[1], Symfony[1], CakePHP[1], Drupal[1], Kohana[1], Fat-Free Framework[2], Laravel[2]	KO3-NoSQL, Vork
C#.NET			Slazure, Charisma
Ruby	ROM	Rails[1], Hanami[1]	Ruby ORM Adapter
Swift		Fluent ORM[2]	
Scala	Lift	Play Framework[1]	Activate Framework
Go		upper[2]	

[1] ONDM has no official NoSQL database support, however third-party plugins are available.
[2] ONDM only has official support for one NoSQL database.

Table 4: ONDM frameworks per object-oriented programming language.

	Apache Gora [58]	Kundera [59]	DataNucleus	Hibernate OGM [60]	GORM [61]	JSData [62]	Eclipse JNoSQL [63]	ROM [64]	Waterline [65]	Doctrine [66]	Spring Data [67]	EclipseLink [38]	Bass [57]	Yii Framework [68]	pyDAL [69]	Lift [70]	Lithium [71]	Thinodium [72]	KEV [41]
Databases	**Included** (n = 11)										**Excluded** $E2$ (n = 6), $E3$ (n = 2)								
NoSQL	10	9	4	7	3	7	12	9	5	3	14	2	3	2	3	2	2	2	2
Document	3	3	1	2	1	5	5	4	2	3	5	1	2	1	3	2	2	2	2
Key-value	2	2	0	3	0	2	4	2	2	0	7	1	1	1	0	0	0	0	0
Graph	0	1	1	1	1	0	2	1	1	1	4	0	0	0	0	0	0	0	0
Columnar	3	3	2	0	1	0	2	1	1	0	2	0	0	0	0	0	0	0	0
RDBMS	×	✓	✓	×	✓	✓	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	×

Table 5: Overview of database support for the compared ONDMs.

shows that document and key-value stores support is often simultaneously available due their inherent similarities. Furthermore, column stores such as Apache Cassandra [26] and Apache HBase [27] are less frequently supported than document stores.

Graph stores are supported by several mappers. However, in the case of Eclipse JNoSQL [63] and Doctrine ODM [66] this is because of support for a multi-model NoSQL database such as OrientDB [76]. OrientDB lets the application developer choose between a graph, document or key-value data model. Consequentially, the ONDM can uniquely implement either data model. In contrast, object-graph mapping is specifically available where a dedicated graph store like Neo4j is supported, which is the case for Kundera [59] and GORM [61].

Finally, some ONDMs also feature persistence to on-disk storage, or an in-memory store for caching, or even to file formats such as Microsoft Word or Excel in the case of DataNucleus [73]. More recently, Impetus Kundera [59] has even implemented read support for the Ethereum blockchain.

5.2. Interface support and functionality (C2)

The supported database are accessed via non- or standardized interface(s). Standardized interfaces imply support for features compliant with its respective specification, such as JPA-compliant frameworks [22], which typically implies support for JPQL.

Standardized interfaces. Rows 3–5 in Table 6 compare the supported interfaces by each ONDM. Java frameworks typically adopt either the JPA or JDO interface. DataNucleus [73] has an implementation for both JPA and JDO, however the developers argue that JDO is designed as a more database-agnostic variant over JPA. Apache Gora [58] is a Java framework with a custom interface. In addition, Hibernate OGM [60] provides full-text search via its Hibernate API. In certain cases a REST interface is also provided which ultimately communicates with the framework's interface. Eclipse JNoSQL [63] provides four custom interfaces: one for each type of NoSQL database.

Query functionality. All of the studied ONDMs feature at minimum programmatic query operations for Create, Read, Update and Delete (CRUD). Rows 6–9 of Table 6 provide an overview of which frameworks allow constructing queries using criteria objects, and which frameworks support query languages, or provide the use of (native) query languages.

As shown in rows 7 and 8, Several ONDMs provide a programmatic abstraction to construct queries with criteria objects or a dedicated query language such as JPQL. In addition, dynamic query operations such as `findByName` or `findWhereAgeGreaterThan` can be added dynamically by the framework through inheritance or reflection.

Dedicated query languages such as JPQL [22] and JDOQL [36] have the advantage of offering more flexibility and result in less ad-hoc queries over query objects. The downside of dedicated query languages is that it requires syntax verification and the results have to be identified and matched with classes. Type-safety is often facilitated through the use of query builders with criteria objects. Such query objects have the advantage that they can be constructed against classes instead of table

	Apache Gora [58]	Kundera [59]	DataNucleus [73]	Hibernate OGM [60]	GORM [61]	JS Data [62]	Eclipse JNoSQL [63]	Waterline [65]	Spring Mongo [67]	Doctrine ODM [66]	ROM [64]
Version	0.8	3.12	5.1	5.3	6	3	0.0.5	0.13.5	2.1	1.2	4
Interfaces		I_1	I_1,I_2	I_1					I_1		
Custom	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓
REST	×	✓	✓	×	✓	✓	×	×	✓	✓	×
Query functionality											
Query object	✓	×	✓	×	✓	✓*	✓	✓*	✓	✓*	✓*
Query lang.	×	✓	✓	✓	✓	×	✓	×	×	×	×
Native query	×	✓	✓	✓	✓	×	×	×	×	×	×
Architecture											
Repository	✓	✓	✓	✓		✓	✓		✓	✓	
Active record					✓			✓			✓
Portability and coupling											
Framework	+ +	-	- -	-	+	+	+ +	0	+	+	0
Database	+ +	-	- -	0	+	-	+	-	+	+	0

I_1 = JPA, I_2 = JDO, $*$ = Non-type safe query

Table 6: Interfaces and functionality, architectural design and coupling of compared ONDMs.

names, which are then parameterized and executed with type-safety. The frameworks that feature query builders which are not type safe are marked with an asterisk in Table 6.

Native database query languages are more efficient and functional than abstracted operations. Hibernate OGM [60] and GORM [61] both support MongoDB's and Neo4j's query languages. Kundera [59] in turn has support for Apache Cassandra's CQL [26], but lacks support for other native query languages despite supporting multiple database technologies.

In summary, there is large diversity in the ways data can be queried or stored with all of them supporting CRUD operations, and the non-Java frameworks feature solely additional query object support.

5.3. Architectural patterns (C3)

Rows 10–12 of Table 6 present the result of distinguishing between the active record pattern and the repository pattern as the dominant design paradigm of the ONDM.

Within the class of ONDMs that implement the active record pattern, we can distinguish between (i) ONDMs such as Waterline [65], GORM [61] and ROM [64] that add data access and query logic statically to the domain objects and (ii) ONDMs that introduce data access logic dynamically through reflection as is the case for GORM [61].

In the former case, these data access operations are inherited from a parent class in the ONDM framework (e.g. ROM [64]).

In ONDMs that implement the repository pattern, the mapping framework identifies persistable classes with the use of annotations or by loading XML descriptions. In JS Data [62] domain objects are instantiated by providing a schema specification as input to a mapper class. As for repository-oriented standards such as JPA and JDO, the domain access logic is largely available at for example an `EntityManager`. Eclipse JNoSQL [63] shares the notion of an `EntityManager` despite being non-JPA and non-JDO compliant.

Next to modeling descriptors for domain objects there are also plain object mappers, which do not necessarily have to follow any conventions or modeling guidelines. In such a case, the mapping behavior is defaulted. Spring Data [67] and GORM [61] can both handle plain objects.

The architectural style impacts data access logic op-

erations and domain class modeling, however, also influences the level of software coupling.

5.4. Framework and database coupling (C4)

We compare ONDMs in terms of: (i) framework coupling, and (ii) database coupling.

Framework coupling. Row 14 of Table 6 presents a qualitative ranking of the ONDM frameworks based on the degree of framework coupling.

Spring Data [67] and GORM [61] can both handle simple objects, however annotations can be added to specify behavior. However Spring Data's [67] functionality is provided for e.g. a `User` class through a `UserRepository`, while in GORM [61] the functionality is dynamically added to the domain class. The subsequent operations are then tied to these domain classes or repositories, which are however non-standardized.

JPA frameworks such as Kundera [59], DataNucleus [73], Hibernate OGM [60] use standardized mapping annotations in the domain classes. For example, Hibernate OGM has a specific Hibernate API for full-text search. In terms of mapping, DataNucleus allows the specification of mapping metadata in an external XML format.

The scripting language frameworks JS Data [62] and Waterline [65] are highly similar to each other and create the domain class through a model or schema constructor with type and property descriptors. The data access logic is created upon model instantiation for Waterline, however JS Data uses a repository (mediator) to store or query objects. Apache Gora [58] induces high database coupling as it requires the user to specify data classes in a JSON-format (Avro) that is specific to each DB technology. These class descriptions are then compiled into domain classes that extend functionality from the core framework. In addition, Apache Gora's interface is non-standardized.

Database coupling and interoperability. Row 15 of Table 6 presents a qualitative ranking of the ONDM frameworks based on the degree of database coupling. GORM Grails [61], Doctrine [66] and Spring Data [67] provide multiple framework versions or subprojects for specific database technologies, and thus deviate from the principle of a single standard interface. JS Data [62], Waterline [65], and ROM [64] feature a single interface for each supported database. This is mainly a result of the simplicity of the provided functionality (i.e. minimal abstraction).

In terms of search functionality, specific query operators can be unsupported depending on the capabilities of the configured database, which in turn may break the abstraction of the single uniform API. For example, Impetus Kundera [59] only supports the `OrderBy` query operator for MongoDB and RDBMS. In dealing with such scenarios, DataNucleus [73] provides uniform abstraction by implementing non-native database functionality in the framework itself. For example, its `OrderBy` query operator is implemented by retrieving all necessary data and then sorting within the framework. In a similar fashion, filter or conditional operations can be applied in-memory after client-side filtering. However, the use of such non-native functionality can come at a significant performance cost.

Along with database coupling at the interface level, such coupling also exists at the domain or mapping level. Apache Gora [58] requires for example specifying the mapping from object to the database structure in XML and these specifications are highly dependent on the selected database technology. Such an approach introduces a tight coupling between the application domain and the database itself.

6. Domain object mapping strategies (RO2)

In this section, we systematically compare the ONDMs in terms of the supported object mapping strategies. First, we discuss object-document mapping in Section 6.1, followed by object-graph mapping in Section 6.2, and finally, object-column mapping in Section 6.3.

6.1. Object-document mapping strategies

We start by comparing the mapping strategies for collections (e.g. `List`, `Map`) of values. Then, we compare relationship mapping strategies, and finally mapping strategies for inheritance relationships.

The frameworks can either implement a default translation for mapping object-oriented elements or the developer can specify the behavior. We survey the frameworks on default mapping behavior and the availability of other strategies.

6.1.1. Mapping object embeddables and collections

Table 7 compares the ONDMs in terms of their support for mapping of collections and arrays, specifically for document stores. In general for the frameworks, when using for example similar JPA annotations `@Embedded` or `@ElementCollection` annotations, the entire data structure will be nested for document stores as an array, or as multiple columns. However, the collection may grow rather large and consequentially it is

	Apache Gora [58]	Impetus Kundera [59]	DataNucleus [73]	Hibernate OGM [60]	GORM [61]	JSData [62]	Eclipse JNoSQL [63]	Waterline [65]	Spring Mongo [67]	Doctrine ODM [66]	ROM [64]
Collections											
Embedding	A	✓	✓	✓	✓	A	✓	×	✓	✓	×
- Ordering	×	I	×	I,O,G	I,O	×	×	-	I	I,G	-
References	×	×	✓	×	✓	×	×	✓	×	×	✓
- Embedding ref	-	-	×	-	✓	-	-	×	-	-	×
- Embed data	-	-	×	-	✓	-	-	×	-	-	×
- Join table	-	-	✓	-	×	-	-	✓	-	-	✓
- Join table + embed	-	-	✓	-	×	-	-	×	-	-	×
Relationships											
References	-	✓	✓	✓	✓*	✓*	-	✓	✓	✓	✓
- Join table	-	✓	×	✓	×	×	-	✓	×	×	✓
- Join table + embed	-	×	×	✓	×	×	-	×	×	×	×
Embedding	-	×	✓	✓	✓	×	✓	×	✓	✓	×
- Embed references	-	-	✓	✓	✓	-	-	-	✓	✓	-
- Embed data	-	-	×	×	✓	-	✓	-	×	×	-
- Ordering (def)	-	-	×	I,O,G	I,O	-	-	-	I	I,G	-

A = Array, I = Sorted index, G = Grouping, O = Ordering embedded data, * = No many-to-many relations

Table 7: Mapping collections and relationships for **MongoDB**: Embedding and referencing strategies **(S1)**

not always ideal to embed such a large array within the parent record. Ideally in such cases, the collection can then be normalized and referred to within the parent record (**S1**). However, none of the frameworks in Table 7 provide such functionality, nor mapping solutions on a per object basis (**S2**).

However, as shown in Table 7, creating references for collections is only officially supported by DataNucleus [73]. None of the frameworks support normalizing collections and embedding references within the owning entity. Consequentially, a query will have to be executed on the referred table's foreign key. The lack of support for embedding references impacts read performance. Interestingly enough, while typically join tables only store the foreign keys, in the case of DataNucleus [73], a collection's element can be stored within the row. Certain frameworks such as Waterline [65] also allow custom mappings to be defined on e.g. read or write. However, defining custom mappings will mostly break certain query functionality, since the format queried is unconventional to the framework.

Not listed in Table 7 are the relational database collection mapping strategies, which can create references for collections. Despite the availability of collection mapping strategies with normalization they are *only* supported when a relational database is selected.

Ordering and grouping collections. A potentially overlooked feature of importance is the support for ordering of collections and references stored. NoSQL databases do not always provide the full set of features common in relational databases. For example, ordered write of records are not supported in MongoDB [29]. Although data can be ordered with the use of a sorted index in several frameworks, this in turn requires additional memory and storage resources.

In addition to record ordering, the embedded data within the parent can also be ordered. Hibernate OGM [60] can order a collection as an embedded map and store these elements ordered on a specified key using `@OrderColumn`. GORM [61] simply preserves the order of collections such as lists. Kundera [59] and Apache Gora [58] can order the records stored on a clustering key in Apache Cassandra [26].

Ideally, we would also be able to partition and group certain attributes on values (e.g. boolean properties) in anticipation of group queries. This functionality is present in Doctrine ODM [66] and Hibernate

OGM [60], which can group embedded data on field values.

While it is typical to embed these owned structures in the record, strategies for embedding and referencing can also be applied to relationships.

6.1.2. Relationship mapping

Table 7 also compares the ONDMs in terms of their relationship mapping strategies for embedding (part) of the referred data, through references, and potentially embedding these references. Notably, Apache Gora [58] and Eclipse JNoSQL [63] do not provide any support for relationship mapping. Eclipse JNoSQL is however still a rather young project and under heavy development. Furthermore, GORM [61] also does not feature relationships when using Apache Cassandra [26]. JS Data [62] does not support many-to-many relationships, whereas GORM [61] allows modeling a many-to-many relationship as bi-directional one-to-many relationships.

References and join tables. As shown in Table 7, when relationships are supported, the tactic of referencing is omnipresent. This contrasts to collection mapping, which lacks strategies for referencing or join tables. Additionally, while many of the ONDMs provide relationship mappings to join tables using relational databases, this is not the case for NoSQL databases. Waterline [65] and ROM [64] do feature join tables for associations, and can embed additional data within the join table. However, when retrieving an entity, this data is not automatically retrieved in Waterline.

Embedding references. Alternatively to join tables, references can also be embedded. For example, in Hibernate OGM [60] which can store references within a document. Table 7 shows that for the embedding of relationships as references is supported more frequently than join tables. For example, Hibernate OGM embeds only the references to foreign keys.

In case of a record with many associations (e.g. several thousands of object references), it can still be beneficial to use an join table. However, the investigated mapping strategies either exclusively use join tables or embedding, despite benefits and drawbacks for both approaches.

Embedding relationship data. A different strategy involves copying relationship data and embedding it in the object, while existing independently of the referring record. As shown in Table 7, this is supported only in GORM [61] and Eclipse JNoSQL [63]. In most frameworks, if relationship data is to be embedded, it must be modeled as owned and used uniquely by a single object (i.e. embeddable), despite there being a large conceptual discrepancy between relationships and embeddables.

This is rather surprising as the feature to embed or nest relational data is a highly requested and major feature to fully use NoSQL databases. For example, Waterline [65] users have already requested this feature in 2014, and as of yet it remains unimplemented [77]. Often a workaround is suggested e (e.g. JugglingDB [78]) involving user-implemented mapping functions that override the default read and save methods. Doctrine ODM [66] applies this tactic to achieve relationship nesting, however functionality often breaks at certain points with such user-implemented mapping strategies.

Embedding relationship data partially is a more advanced mapping strategy and it is not supported by any of the frameworks from Table 7.

Ordering relationship data. Similarly to collection ordering, certain NoSQL databases lack functionality for ordering of relationships, and thus support for ordering can be implemented in the ONDM. The final row of Table 7 shows the extent to which this is supported by the investigated ONDMs.

Hibernate OGM [60] allows ordering by property on the relationship. Alternatively, an ordered index can be specified if available by the database in for example Doctrine ODM [66]. The GORM [61] framework preserves the order of object data in lists, arrays and collections except for `Bag`. Hibernate OGM and Doctrine ODM further allow to discriminate between embedded relational data on for example field type. This is potentially beneficial to "group by" query performance. Additionally, Hibernate OGM can group certain embedded relational data in separate association documents, or in a single association document.

Mapping transparency. While relational data may not be stored as embedded, some mappers automatically query all relevant data and present it as embedded to the user (e.g. ROM [64] and JS Data [62]). The drawback is that it is not always clear whether the embedded structures are composed by the framework, or if they are actually stored this way. The ROM [64] and JS Data [62] frameworks compose embedded structures in-memory while in the physical database, all data is actually stored in normalized tables. In these frameworks, embedding or nested data is merely a conceptual representation resolved by the framework as everything is

	Apache Gora [58]	Impetus Kundera [59]	DataNucleus [73]	Hibernate [60]	GORM [61]	JS Data [62]	Eclipse JNoSQL [63]	Waterline [65]	Spring Mongo [67]	Doctrine ODM [66]	ROM [64]
Inheritance	×	✓	✓	✓	✓	×	✓	×	×	✓	×
- Single table	-	✓	R	✓	✓	-	×	-	-	✓	-
- Joined table	-	×	R	×	✓	-	×	-	-	×	-
- Concrete table	-	×	✓	✓	×	-	✓	-	-	✓	-
- Superclass-table	-	×	R	×	×	-	×	-	-	×	-
- Subclass-table	-	×	R	×	×	-	×	-	-	×	-

R = Supported only for relational databases.

Table 8: Inheritance mapping strategies for document stores.

stored in a normalized fashion.

6.1.3. Class and inheritance mapping

Class objects conceptually do not have a database counterpart. Therefore, class elements have to be translated such as attributes and associations. which are potentially inherited.

Inheritance mapping. Table 8 compares the ONDM frameworks in terms of the inheritance mapping strategies discussed in Section 2.3.3, namely: *single-table*, *joined-table*, *concrete-table*, *superclass-table* and *subclass-table*.

A number of frameworks lack support for inheritance mapping altogether. Specifically, inheritance is not supported for the JavaScript framework JSData [62][3] and Waterline [65] as the inheritance concept does not exist in these languages.

As shown in the final two rows of Table 8, none of the investigated frameworks supports the novel tactics *superclass-table* and *subclass-table* that were discussed in Section 2.3.3.

As it is primarily designed to access basic functionality in the database systems, and/or apply analytics on a simple data elements, Apache Gora [58] has no mapping support for inheritance, nor relationships for that matter.

DataNucleus [73] supports all five inheritance strategies, however when using NoSQL databases, only the *concrete-table* mapping strategy is supported.

[3]Contemporary JavaScript tools make use of prototyping to model inheritance.

Additionally, although Impetus Kundera [59] claims to be JPA-compliant –which implies support for *single-table* and *joined-table*, it only supports the *single-table* strategy.

6.2. Comparison of object-graph mapping strategies

While we have primarily focused on document-oriented databases, we also evaluate the 11 ODNM frameworks in terms of their object-graph mapping.

Table 9 compares the ONDM frameworks in terms of their object-graph mapping strategies. Apache Gora [58] and JS Data [62] do not support graph databases. In addition, ROM's Neo4j adapter and Doctrine ODM's adapter for OrientDB are discontinued and lack certain mapping and query functionality. From Table 9 we can tell that attribute fields with supported data types (e.g. `String`) are persisted as properties. Domain classes are typically always mapped to nodes, except for Waterline [65] which can store an entity class as an edge, although the graph DB may require endpoints.

Embeddables. In almost all ONDM frameworks, embeddables are represented by nodes (vertices) and edges. Eclipse JNoSQL [63] has default functionality for embedding the object data as properties within the parent node, however without a readily-available DB adapter implementation. DataNucleus [73] can embed entities as properties within the parent node, although collections have to be linked.

Relationship mapping. The ONDM frameworks only map relationships using edges. Many-to-many relationships are implemented with multiple edges between the

	Apache Gora [58]	Impetus Kundera [59]	DataNucleus [73]	Hibernate [60]	GORM [61]	JS Data [62]	Eclipse JNoSQL [63]	Waterline [65]	Spring Neo4j [67]	Doctrine ODM [66]	ROM [64]
Version	0.8	3.12	5.1	5.3	6	3	0.0.5	0.13.5	4.2	1.0	4
Graph DB	×	✓	✓	✓	✓	×	×	✓	✓	✓	×~
Attribute											
- Edge(s)	-	×	×	×	×	-	×	×	×	×	×
- Property	-	✓	✓	✓	✓	-	✓	✓	✓	✓	✓
Embeddables											
- Edge(s) +Node(s)	-	✓	✓	✓	✓	-	×	✓	✓	×	✓
- Embed	-	×	✓,$\times_c$	×	×	-	✓	×	×	✓	×
Relationships											
- Edge(s)	-	✓	✓	✓	✓	-	✓	✓	✓	✓	✓
- Embed	-	×	×	×	×	-	×	×	×	×	×
- Ordering	-	I	✓	✓	✓	-	×	×	✓	×	×

I = Index. $\times_c$ = No collection mapping. L = Link edges to nodes.

Table 9: Supported object-graph mapping strategies.

vertex types, thus not requiring a join table. In Waterline [65] many-to-many relationships used to be modeled using proxy nodes, which are similar to junction tables. However, this proxy strategy was removed in favor of direct edges between the nodes.

Ordering relationship data. In certain cases, the ordering of the relationship can be preserved. This is typically implemented by adding a order property on the edge.

Inheritance mapping. Inheritance mapping support towards graph databases is generally lacking and documentation is scarce. Impetus Kundera [59] does not provide any inheritance test cases for Neo4j. GORM with Neo4j and DataNucleus [73] map inheritance using a concrete-table strategy. Hibernate OGM [60] features test cases for *single-table* and *concrete-table* inheritance in Neo4j, which simply changes a node class property (e.g. `class=User:Person`). The default object-graph converter of Eclipse JNoSQL [63] indicates that inheritance simply changes a class label value at the node and fields are most likely inherited. Spring Data for Neo4j's inheritance handling is also unclear, and presumably also adopts the *concrete-table* strategy.

6.3. Comparison of object-column mapping strategies

As a consequence of the similarity in the supported data models, object-column mappings strategies are highly reminiscent of traditional object-relational mapping strategies. The investigated object-column mappings are generally more limited when compared to the object-document mappings discussed above.

Hibernate OGM [60], JSData [62], and Doctrine [66] do not provide column store support.

GORM [61] supports simple entity with inheritance mapping without associations or collections. Impetus Kundera [59] stores relationships as tables, but also allows embedding of collections. DataNucleus [73] supports embedding of single objects, and always embeds foreign key references. For a complete reference of the object-column store mapping functionality the documentation of the ONDM frameworks can be consulted.

ROM [64] stores all data into normalized tables, but also supports conceptual transformation strategies to for example nested structures.

7. Discussion

This section discusses the main results of the survey and explores a number of interesting avenues of future

research. Section 7.1 first explores a number of frameworks and libraries that were formally excluded from this survey study, but do have some noteworthy properties. Section 7.2 then discusses the identified challenges of implementing NoSQL-specific mapping strategies in the context of ONDM frameworks, and as such provides some explanations for the current state of these systems. Finally, Section 7.3 discusses the challenges and areas of future research and development of ONDM frameworks that were identified during this survey study.

7.1. Exploration of excluded frameworks

As discussed in Section 4, we have deliberately scoped the study to allow for a focused and in-depth comparison of the ONDM platforms of interest. However, we have also explored a number of the excluded frameworks for which we present our main findings. This is an exploratory effort, and we consider a more in-depth and systematic survey of some of these classes of frameworks to be part of our future work.

7.1.1. Academic ONDM frameworks

Renesca [79] is an object-graph mapper (OGM) for Neo4j and Scala. The Renesca platform distinguishes itself from Hibernate OGM [60] and Spring Data Neo4j [67] by supporting graph-query results, hyper graphs, and multiple-inheritance.

Wolf et al. [32] implement Hibernate ORM on top of a the RIAK key-value store. This implementation was made before the initial release of Hibernate OGM [60] as an implementation and evaluation of Object-NoSQL mapping. The study primarily evaluates the performance of mapped objects in the NoSQL database RIAK versus MySQL, for example on object write and retrieval.

Atzeni et al. [80] implement a framework for uniform access to NoSQL systems, and this implementation highlights the challenges of providing a uniform interface and data model. In this framework, objects with associations are always embedded by either two patterns: multiple column(s) or field(s) creation, or map to a single column or field.

CDPort [81] is a framework for portability between NoSQL stores. It however does not specifically target object mapping. Similarly, ODBAPI [82] provides a unified REST API for relational and NoSQL data stores, but it does not handle the mapping of objects with associations transparently.

7.1.2. Excluded NoSQL Wrapper frameworks

The framework search methodology initially revealed 112 object-mappers targeted at single NoSQL technologies, which were discarded due to exclusion criterion **E1**. Yet, these mappers may still feature interesting object-NoSQL mapping strategies. To explore these frameworks further, we have sorted them in terms of their language support, additionally excluding those libraries with inactive or ceased development according to **E4** as shown in Figure 8. The remaining 39 NoSQL wrappers are analyzed in an exploratory fashion for interesting or noteworthy object-mapping features (specifically for **S1** and **S2**). We more specifically scoured the documentation and source code of the selected wrappers for object-mapping functionality. The main findings of this exercise are summarized below.

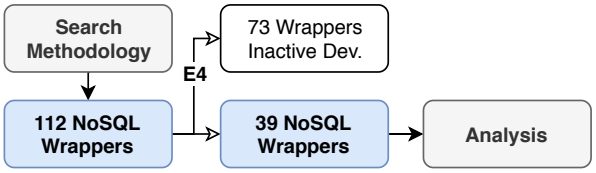

Figure 8: Further selection of excluded NoSQL wrapper frameworks.

In general, we found no use of any strategies related to mapping on a per-object basis (cfr. **S2**), for example changing the strategy based on the values of certain object attributes. Although tools exist that provide such functionality (e.g. ModelMapper [83]), to our knowledge this has not yet been used for NoSQL databases. Additionally, none of the investigated NoSQL wrappers supported partial embedding of relationship data. In the studied wrapper frameworks, relationships were either embedded as a list of references or the complete referred data structure. These findings are highly similar to the analyzed ONDM frameworks.

Apart from these broadly-applicable observations, we have identified a number of interesting mapper features. Neo4j-ogm [84] supports a `@JsonIgnore` annotation to prevent loops in graph traversal. Mongoid [85] for MongoDB has support for recursive embedding, in addition a child embedded document can be used in multiple types of parent documents. MongoKit [86] also allows class attributes to be of multiple types, such polymorphism fits well with the flexible schema nature of NoSQL document databases.

7.1.3. Excluded ONDM frameworks

Although certain ONDM frameworks were excluded due to **E3** and **E4**, some frameworks did show noteworthy features. We discuss these below.

Resourceful [87] is a JavaScript front-end ORM layer with an isomorphic resource engine, the object's at-

tributes are modeled using methods which is a distinctively different approach to common object domain modeling.

While Cumulus4j [88] is not a full-fledged ONDM framework, it serves as an encryption layer on top of DataNucleus [73] to protect application data in outsourced environments.

The Ruby ORM adapter [89] provides abstraction support for multiple ORM frameworks.

Finally, the Activate Framework [90] describes an interesting transactional architecture and polyglot persistence case.

7.2. Challenges in NoSQL mapping strategies

The specialized nature and data models inherent to NoSQL lead to best practices on how to structure and persist data that go beyond those applied in a relational database context. As discussed in Section 2.3.4, examples of NoSQL mapping strategies are embedding (**S1**) and support for per-object mappings (**S2**).

For example, data duplication (e.g. embedding data within a parent record in addition to persisting the embedded data in a separate record to optimize query performance) are not implemented in the studied frameworks. More specifically, of the studied ONDMs, data duplication is only minimally supported, e.g. at the level of embedding reference keys such as in Hibernate OGM [60], or by embedding partial data in a join table. In fact, ROM [64] even supports embedding part of the data in the join table, but this data is not automatically loaded by the framework (i.e. hydrated). JS Data [62] allows nested representations of relationships, although the data is stored in normalized tables at the database end.

In practice, when a developer requires relationship nesting or embedding they have to implement custom functions, which in turn requires specific NoSQL expertise and leads to complex, application-specific code that can not be ported easily. ONMDs that support this are Doctrine ODM [66], GORM [61], and ROM [64].

Reusable, built-in support for these NoSQL data mapping strategies are desirable at the level of ONDM platforms. However, implementing these strategies is not straightforward due to several factors, such as schema complexity and schema bookkeeping, query pathing, data duplication and consistency. We discuss each of these in the subsequent sections.

7.2.1. Schema complexity and bookkeeping

A database schema encoded in 3NF has advantages in presenting a clear and concise relational model without data redundancy, and optimized for analytical queries due its column structure. Suppose a developer wishes to optimize certain read queries, he can then create denormalized structures and embed part of the join query within the original record. In a later development stage, duplicate data structures will result in a DB schema that is difficult to maintain by the application, or its developers.

7.2.2. Data duplication, consistency and query pathing

A framework can however deal with managing data consistency automatically, and work as intended by providing abstraction from the database complexity. However, update and write operations can individually come at high latency due to unexpected consequences of the schema when badly modeled. In addition, deciding upon a query route is no easy feat since there are several options to answer a given query.

The framework therefore needs to make object mapping decisions based on properties such as: object and relationship size, query frequency, the existing data structures, and current performance. Ideally, such information is collected by monitoring and benchmarking within the framework.

7.3. Research challenges in ONDMs

Based on this survey study, we have identified several areas of further research and development of ONDM frameworks. We outlines these below.

7.3.1. Object-level mapping

In practice, the denormalized model is adopted because of its advantages in terms of quick read performance, over write and update complexity due to maintaining consistency. However, as discussed, a denormalized schema is not transparent or maintainable to developers as it leads to increased complexity at the level of write and update operations. As such, it is highly desirable to address such concerns independently from application code, and thus ideally within Object-NoSQL database mappers.

In the current state of ONDMs, the decision to persist a data element as either denormalized or normalized is decided (i) at development time, and (ii) at the level of data types (class level).

However, thanks to characteristics such as schema flexibility of NoSQL databases, individual objects of the same type can be mapped to the database differently. For example, many applications encounter so-called *hotspot* objects, or records which are highly clustered. For example, a small subset of Users on a Review website is typically responsible for the majority of

the content written. Suppose the reviews written by the users are embedded within the User document, a large portion of read and update requests will be targeted at a single record. Therefore, applying a normalization on this subset of the frequently-read records is highly desirable. Furthermore, the most suitable per-object mapping strategy is commonly determined on dynamic properties: what is a viral news item today, will perhaps not be as frequently read tomorrow. These properties can vary, such as the size of certain arrays, storage resources, and query activity.

As shown in this survey study, support for providing (i) fine-grained, per object mappings that also (ii) can be based on dynamic properties is entirely lacking in current ONDMs (**S2**). Supporting these levels of customization will have profound impact on (i) common ONDM architectures, (ii) development APIs and interfaces, and (iii) data management aspects.

7.3.2. Support for schema design

The problem of identifying a good mapping and data placement strategy, given a fixed set of requirements (performance, availability, consistency, security) is a complex search problem, which is exacerbated by the level of technical NoSQL expertise required to identify the best data models, and database technologies and configuration parameters.

Existing work on schema recommendation and schema design support [91, 92, 93] is promising but does not take all relevant properties and parameters into account. The optimization of mapping strategies decisions are influenced by various dimensions, for example the embedded relationship depth, attribute selection and grouping. In previous work [12], we have discussed these object NoSQL modeling dimensions and algorithms and outlined a research roadmap for schema generation and recommendation systems. These schema recommender systems are typically generative in nature: candidate schemas are generated and then evaluated in terms of projected cost or suitability criteria.

ONDM frameworks are by definition developer-centric. They decouple the application from database-specifics and allow the developer to express application abstractions without having to be aware of the database or encoding in which the data ends up. Ideally, ONDMs allow developers to use these databases without requiring a level of expertise in NoSQL database. For these reasons, integration of and extension with such development support tools is thus desirable. Hibernate OGM [60] is to our knowledge the only ONDM framework with a denormalization engine on its roadmap.

7.3.3. First-class support of mapping strategies

We have shown that some of the current ONDMs offer different mapping strategies and this level of variability is supported most visibly in the context of document databases. In these studied ONDM architectures however, these mapping strategies are typically hard-coded, and it is very difficult, even undesired to manipulate these.

However, and much in line with what is discussed above in Section 7.3.1 and throughout, allowing variations in mapping strategies is often desirable, to fine-tune aspects of performance, security, but also to allow experimenting with different encodings, etc.

Redesigning ONDM frameworks so that implementations of different mapping strategies are well-modularized, pluggable, configurable and portable across different frameworks is thus one interesting venue for further exploration, and in fact, a key enabler for researching improved mapping strategies that are tailored to specific databases or data models.

An interesting extension of this idea is the idea in which application-specific data mapping strategies can be provided by the application developer itself. This would provide a level of customization that goes beyond the instructions currently often supported in the form of code annotations.

7.3.4. Multi-store, multi-model, multi-node and multi-cloud storage architectures

Most of frameworks studied in this survey can be configured to persist data in one database at once. However, an increasingly common setup involves combining multiple database technologies (multi-store), encoded in different data models (multi-model), persisted in a complex distributed system (multi-node) and perhaps even across different providers (multi-cloud).

Of the investigated ONMDs, Impetus Kundera [59] is only with initial support for such scenarios, enabling the developer to execute a single query traversing multiple DB technologies transparently (polyglot persistence).

Many platforms exist for the management of multi-cloud storage. Examples are PERSIST [94], SCOPE [95], Hybris [19], MetaStorage [96], MCDB [97], DepSky [20], HAIL [17], IC-Store [98], SPANStore [99], RACS [16], CDPort [100], CSAL [101], and CHARM [102]. These platforms provide complex data management middleware facilities that in many cases rely extensively upon database abstraction facilities.

This however raises the key question to what extent ONDMs should provide explicit support for dealing with heterogeneity in terms of multiple data stores, data

models, and different deployment modes of the back-end storage systems. Many envisioned scenarios can benefit from tighter integration of ONDM functionality with such facilities. For example, data mapping strategies can be aware that data is to be persisted in a public cloud context and may thus decide upon data protection measures (encryption). Ideally, such complexity is again dealt with outside of the application scope and transparently to the developer.

8. Related work

The survey study presented in this paper is based upon similar studies conducted in the context of Object-Relational Mappers (ORMs). We discuss this area of related work in Section 8.1. Section 8.2 discusses related studies that focus on comparing and qualifying the state of the art in Object-NoSQL data mappers (ONDMs). Other types of related work such as academic ONDM frameworks are discussed throughout the previous Section 7.

8.1. ORM survey studies

The study presented is highly similar in some regards to the survey by Torres et al. [4] on object-relational mapping patterns. In the study, a framework is chosen for each popular object-oriented programming language and compared on mapping strategies available for concepts such as attributes, classes, associations and inheritance. Similarly, our survey analyzes the ONDM frameworks on object-relational patterns mentioned in this survey, however extended with aspects specific per NoSQL database category. We have also included additional ORM patterns such as JDO's strategies for inheritance mapping, namely the strategies: *subclass-table* and *superclass-table*. Torres et al. [4] provide a comparison of frameworks, of which only one is chosen per language on the basis of popularity. In contrast, our search methodology collects and starts from a data set of 342 frameworks, which is ultimately refined to a similar subset of frameworks. Torres et al. [4] further analyzes and discusses the framework's architectural, structural patterns, and the associated impact on coupling to which we follow a similar approach and reasoning. Coupling for ONDM frameworks is however further impacted due to the heterogeneity present in NoSQL database API's and data models.

8.2. ONDM framework comparison studies

Although the existence of a systematic ORM survey, few studies have compared object-NoSQL mapping frameworks on a systematic basis and as in-depth. Störl et al. [103] compares a set of Object-NoSQL mappers, namely Hibernate OGM [60], Impetus Kundera [59], EclipseLink [38], and a MongoDB wrapper Morphia. The study compares features of database support, interface functionality, schema management, and evaluates the performance overhead. In general, the study provides a good initial overview of the basic, and important, aspects of these frameworks. Modeling annotations or strategies and particularities are only briefly mentioned.

In previous work [11] we have conducted a comparison of ONDM frameworks on high-level features with a specific focus on the performance overhead of these frameworks. The benchmark study involves the following set of frameworks: Hibernate OGM [60], Impetus Kundera [59], Apache Gora [58], EclipseLink [38] and DataNucleus [73]. The frameworks are compared on database support and high-level features such as supported interfaces and query language. The study does not compare available object-oriented mapping strategies or for example the level of coupling with regards to application and database portability.

Similarly, the study by Rafique et al. [104] compares a slightly alternate set of ONDM frameworks on performance, and secondly focuses on portability in terms of the development effort required to migrate between databases with, or without, the use of an ONDM framework.

In contrast to these previous studies, this is to our knowledge the first study which provides an in-depth summary of object-NoSQL database mapping patterns, and furthermore provides a systematic comparison of strategies applied in practice, ranging from object-document, -column, to -graph mapping patterns.

9. Conclusion

In recent years, storage systems have evolved from relational tables to graphs, documents, key-value and wide-column stores and this causes heterogeneity in terms of database interfaces, data models and database features. As a consequence, the traditional mismatch between application domain objects and the target data model, a problem referred to as the *object-relational impedance mismatch*, has become convoluted with new dimensions in data mapping and interface translation.

This survey identifies a total of 342 frameworks relevant in this context, and evaluates in detail 11 Object-NoSQL data mappers (ONDMs). A selective and in-depth comparison (RO1) is conducted in terms of criteria of database support, query and interface function-

ality, framework and DB coupling, and in a second part on available object-mapping strategies. The systematic comparison can aid practitioners to select an ONDM framework that meet application requirements, and as steer the further advancement and development of ONDMs.

Secondly, we have investigated the ONDMs in terms of their mapping strategies for collections, relationships and inheritance (RO2). We have specifically focused on support for mapping strategies for (i) aggregate data models: embedding vs. referencing (**S1**) and (ii) exploiting schema flexibility and mapping strategies that act at the level of individual objects (**S2**). Based on this systematic comparison of available mapping strategies, we conclude that, firstly, for **S1**, collections and object owned data are typically embedded by default and support for alternate strategies is lacking. Relationships can be embedded within the referring object as a set of referred keys or identifiers. Embedding of relationship data is scarcely supported. Secondly, none of the mapping strategies are applied at the individual object-level and always at a global class-level (**S2**). While evaluating the current state of object mappings in these ONDMs, it becomes apparent that object-document mapping strategies are more sophisticated and advanced than those for column stores and graph stores.

The presented survey is an important stepping stone in our ongoing research on NoSQL abstractions, appropriate mapping strategies and schema design, and optimized configuration and deployment of highly-distributed databases. Although many of these frameworks are still under development, the results of this study should remain relevant as it examines fundamental mapping issues between objects and NoSQL databases. In addition, future studies can reuse our systematic comparison methodology.

Data and materials

The full list of identified frameworks from our survey can be found at [56], which provides an overview of ORM and ONDM frameworks cataloged per object-oriented programming language.

Acknowledgments

This research is partially funded by the Research Fund KU Leuven.

Competing interests

The authors declare that they have no competing interests.

References

[1] W. Keller, Mapping objects to tables, in: Proc. of European Conference on Pattern Languages of Programming and Computing, Kloster Irsee, Germany, Vol. 206, Citeseer, 1997, p. 207.

[2] W. R. Cook, A. H. Ibrahim, Integrating Programming Languages & Databases: Whats the Problem? DRAFT Comments welcome!

[3] L. Cabibbo, A. Carosi, Managing inheritance hierarchies in object/relational mapping tools, in: International Conference on Advanced Information Systems Engineering, Springer, 2005, pp. 135–150.

[4] A. Torres, R. Galante, M. S. Pimenta, A. J. B. Martins, Twenty years of object-relational mapping: A survey on patterns, solutions, and their implications on application design, Information and Software Technology 82 (2017) 1–18.

[5] C. Ireland, D. Bowers, M. Newton, K. Waugh, A classification of object-relational impedance mismatch, in: Advances in Databases, Knowledge, and Data Applications, 2009. DBKDA'09. First International Conference on, IEEE, 2009, pp. 36–43.

[6] L. Caruccio, G. Polese, G. Tortora, Synchronization of queries and views upon schema evolutions: A survey, ACM Transactions on Database Systems (TODS) 41 (2) (2016) 9.

[7] M. Stonebraker, SQL databases v. NoSQL databases, Communications of the ACM 53 (4) (2010) 10–11.

[8] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, R. Sears, Benchmarking cloud serving systems with YCSB, in: Proceedings of the 1st ACM symposium on Cloud computing, ACM, 2010, pp. 143–154.

[9] K. Grolinger, W. A. Higashino, A. Tiwari, M. A. Capretz, Data management in cloud environments: NoSQL and NewSQL data stores, Journal of Cloud Computing: advances, systems and applications 2 (1) (2013) 22.

[10] NoSQL-Databases, `http://nosql-databases.org/`.

[11] V. Reniers, A. Rafique, D. Van Landuyt, W. Joosen, Object-NoSQL Database Mappers: a benchmark study on the performance overhead, Journal of Internet Services and Applications 8 (1) (2017) 1.

[12] V. Reniers, D. Van Landuyt, A. Rafique, W. Joosen, Schema design support for semi-structured data: Finding the sweet spot between NF and De-NF.

[13] A. Rafique, D. Van Landuyt, B. Lagaisse, W. Joosen, Policy-driven data management middleware for multi-cloud storage in multi-tenant SaaS, in: 2nd IEEE/ACM International Symposium on Big Data Computing, IEEE, 2015, pp. 78–84. `doi:10.1109/BDC.2015.39`.
URL `https://lirias.kuleuven.be/handle/123456789/509105`

[14] M. Venkat, Enterprise cloud strategy: Applications and data in a multi-cloud environment, `https://www.ibm.com/blogs/cloud-computing/2016/12/applications-data-multi-cloud/` (December 2016).

[15] A. Raghavan, A. Chandra, J. Weissman, Tiera: towards flexible multi-tiered cloud storage instances, in: Middleware '14 15th International Middleware Conference, ACM, 2014, pp. 1–12. `doi:{10.1145/2663165.2663333}`.

[16] H. Abu-Libdeh, L. Princehouse, H. Weatherspoon, Racs: A case for cloud storage diversity, in: Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC '10, ACM, New York, NY, USA, 2010, pp. 229–240. doi:10.1145/1807128.1807165.
URL http://doi.acm.org/10.1145/1807128.1807165

[17] K. D. Bowers, A. Juels, A. Oprea, PHAIL: a high-availability and integrity layer for cloud storage, in: Proceedings of the 16th ACM conference on Computer and communications security, ACM, 2009. doi:10.1145/1653662.16536865.

[18] T. G. Papaioannou, N. Bonvin, K. Aberer, Scalia: an adaptive scheme for efficient multi-cloud storage, in: SC '12 Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, ACM, 2012.

[19] D. Dobre, P. Viotti, M. Vukolić, Hybris: Robust hybrid cloud storage, in: Proceedings of the ACM Symposium on Cloud Computing, SOCC '14, ACM, New York, NY, USA, 2014, pp. 12:1–12:14. doi:10.1145/2670979.2670991.
URL http://doi.acm.org/10.1145/2670979.2670991

[20] A. Bessani, M. Correia, B. Quaresma, F. André, P. Sousa, DepSky: Dependable and Secure Storage in a Cloud-of-Clouds, Trans. Storage 9 (4) (2013) 12:1–12:33. doi:10.1145/2535929.
URL http://doi.acm.org/10.1145/2535929

[21] M. Fowler, Patterns of enterprise application architecture, Addison-Wesley Longman Publishing Co., Inc., 2002.

[22] Oracle, Java Persistence 2.2, Maintenance Release (2018).

[23] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, P. Helland, 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, VLDB Endowment, 2007, pp. 1150–1160.

[24] M. Stonebraker, Stonebraker on NoSQL and enterprises., Commun. ACM 54 (8) (2011) 10–11.

[25] H. Vera, M. H. Wagner Boaventura, V. Guimaraes, F. Hondo, Data modeling for NoSQL document-oriented databases, in: CEUR Workshop Proceedings, Vol. 1478, 2015, pp. 129–135.

[26] The Apache Software Foundation, Apache Cassandra.
URL http://cassandra.apache.org/

[27] The Apache Software Foundation, Apache HBase.
URL https://hbase.apache.org/

[28] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, R. E. Gruber, Bigtable: A distributed storage system for structured data, ACM Transactions on Computer Systems (TOCS) 26 (2) (2008) 4.

[29] MongoDB, MongoDB.
URL https://www.mongodb.com/

[30] V. Jovanovic, S. Benson, Aggregate data modeling style, in: Proceedings of the Southern Association for Information Systems Conference, 2013, pp. 70–75.

[31] P. Atzeni, F. Bugiotti, L. Cabibbo, R. Torlone, Data modeling in the NoSQL world, Computer Standards & Interfaces.

[32] F. Wolf, H. Betz, F. Gropengießer, K.-U. Sattler, Hibernating in the Cloud-Implementation and Evaluation of Object-NoSQL-Mapping., in: BTW, Citeseer, 2013, pp. 327–341.

[33] S. W. Ambler, Mapping objects to relational databases: What you need to know and why, Ronin International.

[34] Neo4j Data Modeling, Neo4j Data Modeling.
URL https://neo4j.com/developer/data-modeling/

[35] Data modeling in RethinkDB, https://www.rethinkdb.com/docs/data-modeling/.

[36] Java Data Objects Expert Group, Java Data Objects 3.1 (2015).

[37] A. Kanade, A. Gopal, S. Kanade, A study of normalization and embedding in MongoDB, in: Advance Computing Conference (IACC), 2014 IEEE International, IEEE, 2014, pp. 416–421.

[38] The Eclipse Foundation, EclipseLink, http://www.eclipse.org/eclipselink/.

[39] Hibernate, Hibernate ORM, http://hibernate.org/orm/.

[40] Docb), https://github.com/capless/docb/.

[41] K.E.V. framework (Keys, Extra Stuff, and Values), https://github.com/capless/kev.

[42] B. Kitchenham, O. P. Brereton, D. Budgen, M. Turner, J. Bailey, S. Linkman, Systematic literature reviews in software engineering–a systematic literature review, Information and software technology 51 (1) (2009) 7–15.

[43] S. Keele, et al., Guidelines for performing systematic literature reviews in software engineering, in: Technical report, Ver. 2.3 EBSE Technical Report. EBSE, sn, 2007.

[44] D. Alur, D. Malks, J. Crupi, G. Booch, M. Fowler, Core J2EE Patterns (Core Design Series): Best Practices and Design Strategies, Sun Microsystems, Inc., 2003.

[45] GitHut, GitHut 2.0.
URL https://madnight.github.io/githut/

[46] IEEE Spectrum, Interactive: The Top Programming Languages 2017.
URL https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2017

[47] TIOBE, TIOBE Index.
URL https://www.tiobe.com/tiobe-index/

[48] Pierre Carbonnelle, PYPL PopularitY of Programming Language.
URL http://pypl.github.io/PYPL.html

[49] RedMonk, The RedMonk Programming Language Rankings: January 2017.
URL https://redmonk.com/sogrady/2017/03/17/language-rankings-1-17/

[50] DB-Engines, DB-Engines Ranking.
URL https://db-engines.com/en/ranking

[51] npm, The npm registry.
URL https://www.npmjs.com/

[52] CPAN, The Comprehensive Perl Archive Network.
URL https://www.cpan.org/

[53] The Ruby Toolbox, The Ruby Toolbox.
URL https://www.ruby-toolbox.com/

[54] P. J. Sadalage, M. Fowler, NoSQL distilled: a brief guide to the emerging world of polyglot persistence, Pearson Education, 2013.

[55] V. Reniers, Object-database mappers: Survey results (2018).
URL https://people.cs.kuleuven.be/~vincent.reniers/mappers/

[56] Object-relational and object-NoSQL database mapping frameworks. 1. doi:10.17632/TJ55YM9TB2.1.
URL https://data.mendeley.com/datasets/tj55ym9tb2/1

[57] Bass), https://github.com/congajs/bass.

[58] Apache, Apache Gora, http://gora.apache.org/.

[59] Impetus, Kundera: Object-datastore Mapping Library, https://github.com/impetus-opensource/Kundera/wiki.

[60] Hibernate, Hibernate OGM, http://hibernate.org/ogm/.

[61] Grails, GORM, http://gorm.grails.org/.

[62] JSData, JSData, http://www.js-data.io/.

[63] The Eclipse Foundation, Eclipse JNoSQL, http://www.jnosql.org/.

[64] ROM, Ruby Object Mapper, http://rom-rb.org/.

[65] Sails, Waterline ORM, http://waterlinejs.org/.

[66] Doctrine Project, Doctrine, http://www.doctrine-project.org/projects.html.

[67] Spring, Spring Data, http://projects.spring.io/spring-data/.

[68] Yii PHP Framework, https://www.yiiframework.com/.

[69] pyDAL framework), https://github.com/web2py/pydal/.
[70] Lift framework), https://liftweb.net/.
[71] Lithium (li3) framework), https://github.com/unionofrad/lithium.
[72] Thinodium ODM), https://thinodium.github.io.
[73] DataNucleus, DataNucleus, http://www.datanucleus.org.
[74] ElasticSearch, ElasticSearch.
URL https://www.elastic.co/
[75] The Apache Software Foundation, Apache Solr.
URL http://lucene.apache.org/solr/
[76] OrientDB, OrientDB.
URL https://orientdb.com/
[77] Waterline, Embedded documents issue, https://github.com/balderdashy/waterline/issues/658.
[78] JugglingDB, Can I nest objects?, https://github.com/1602/jugglingdb/issues/251.
[79] F. Dietze, J. Karoff, A. C. Valdez, M. Ziefle, C. Greven, U. Schroeder, An open-source object-graph-mapping framework for Neo4j and Scala: Renesca, in: International Conference on Availability, Reliability, and Security, Springer, 2016, pp. 204–218.
[80] P. Atzeni, F. Bugiotti, L. Rossi, Uniform access to NoSQL systems, Information Systems 43 (2014) 117–133.
[81] E. Alomari, A. Barnawi, S. Sakr, Cdport: A portability framework for nosql datastores, Arabian Journal for Science and Engineering 40 (9) (2015) 2531–2553.
[82] R. Sellami, S. Bhiri, B. Defude, Odbapi: a unified rest api for relational and nosql data stores, in: Big Data (BigData Congress), 2014 IEEE International Congress on, IEEE, 2014, pp. 653–660.
[83] ModelMapper, http://modelmapper.org/user-manual/property-mapping/.
[84] Neo4j-OGM), https://neo4j.com/docs/ogm-manual/current/reference/.
[85] Mongoid, https://docs.mongodb.com/mongoid/master/.
[86] Mongokit), https://github.com/namlook/mongokit/wiki/Structure.
[87] Resourceful: an isomorphic Resource engine for JavaScript, https://github.com/flatiron/resourceful.
[88] M. Huber, M. Gabel, M. Schulze, A. Bieber, Cumulus4j: A provably secure database abstraction layer, in: International Conference on Availability, Reliability, and Security, Springer, 2013, pp. 180–193.
[89] Ruby ORM Adapter, https://github.com/ianwhite/orm_adapter.
[90] Activate Framework: Architecture documentation, https://github.com/fwbrasil/activate/blob/master/activate-docs/architecture.md.
[91] M. J. Mior, K. Salem, A. Aboulnaga, R. Liu, Nose: Schema design for nosql applications, IEEE Transactions on Knowledge and Data Engineering 29 (10) (2017) 2275–2289.
[92] S. Scherzinger, M. Klettke, U. Störl, Managing schema evolution in nosql data stores, arXiv preprint arXiv:1308.0514.
[93] D. S. Ruiz, S. F. Morales, J. G. Molina, Inferring versioned schemas from nosql databases and its applications, in: International Conference on Conceptual Modeling, Springer, 2015, pp. 467–480.
[94] A. Rafique, D. Van Landuyt, E. Truyen, V. Reniers, W. Joosen, Scope: self-adaptive and policy-based data management middleware for federated clouds, Journal of Internet Services and Applications 10 (1).
[95] A. Rafique, D. Van Landuyt, W. Joosen, Persist: Policy-based data management middleware for multi-tenant saas leveraging federated cloud storage, Journal of Grid Computing 16 (2) (2018) 165–194. doi:https://doi.org/10.1007/s10723-018-9434-6.
URL https://lirias.kuleuven.be/1643052
[96] D. Bermbach, M. Klems, S. Tai, M. Menzel, Metastorage: A federated cloud storage system to manage consistency-latency tradeoffs, in: 2011 IEEE 4th International Conference on Cloud Computing, 2011, pp. 452–459. doi:10.1109/CLOUD.2011.62.
[97] M. A. Alzain, B. Soh, E. Pardede, Mcdb: Using multi-clouds to ensure security in cloud computing, in: 2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing, 2011, pp. 784–791. doi:10.1109/DASC.2011.133.
[98] C. Cachin, R. Haas, M. Vukolic, Dependable storage in the intercloud, Tech. rep., Research Report RZ, 3783 (2010).
[99] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, H. V. Madhyastha, Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services, in: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, SOSP '13, ACM, New York, NY, USA, 2013, pp. 292–308. doi:10.1145/2517349.2522730.
URL http://doi.acm.org/10.1145/2517349.2522730
[100] E. Alomari, A. Barnawi, S. Sakr, Cdport: A portability framework for nosql datastores, Arabian Journal for Science and Engineering 40 (9) (2015) 2531–2553. doi:10.1007/s13369-015-1703-0.
URL https://doi.org/10.1007/s13369-015-1703-0
[101] Z. Hill, M. Humphrey, Csal: A cloud storage abstraction layer to enable portable cloud applications, in: 2010 IEEE Second International Conference on Cloud Computing Technology and Science, 2010, pp. 504–511. doi:10.1109/CloudCom.2010.88.
[102] Q. Zhang, S. Li, Z. Li, Y. Xing, Z. Yang, Y. Dai, Charm: A cost-efficient multi-cloud data hosting scheme with high availability, IEEE Transactions on Cloud computing 3 (3) (2015) 372–386.
[103] U. Störl, T. Hauf, M. Klettke, S. Scherzinger, Schemaless NoSQL data stores-Object-NoSQL Mappers to the rescue?, Datenbanksysteme für Business, Technologie und Web (BTW 2015).
[104] A. Rafique, D. V. Landuyt, B. Lagaisse, W. Joosen, On the Performance Impact of Data Access Middleware for NoSQL Data Stores: A Study of the Trade-off between Performance and Migration Cost, IEEE Transactions on Cloud Computing (2015) 1–14.