Schemaless NoSQL Data Stores – Object-NoSQL Mappers to the Rescue?

Uta Störl
Hochschule Darmstadt
uta.stoerl@h-da.de

Thomas Hauf
Hochschule Darmstadt
thomas.hauf@stud.h-da.de

Meike Klettke
Universität Rostock
meike.klettke@uni-rostock.de

Stefanie Scherzinger
OTH Regensburg
stefanie.scherzinger@oth-regensburg.de

Abstract: NoSQL data stores are becoming increasingly popular in application development. These systems are attractive for developers due to their ability to handle large volumes of data, as well as data with a high degree of structural variety. Typically, NoSQL data stores are accessed programmatically. Due to the imminent lack of standardized query languages, building applications against the native interfaces of NoSQL data stores creates an unfortunate technical lock-in. To re-gain platform independence, developers turn to object mapper libraries as an additional level of abstraction when accessing NoSQL data stores.

The current market for Java object mappers that support NoSQL data stores is still volatile, with commercial and open source products competing for adoption. In this paper, we give an overview on the state-of-the-art in Object-Relational Mappers that can handle also NoSQL data stores, as well as dedicated Object-NoSQL Mappers.

We are able to show that choosing the right object mapper library is a strategic decision with far reaching consequences: Current mappers diverge in the NoSQL data stores that they support, in their features, their robustness, their truthfulness to the documentation and query standards, and ultimately, in the runtime overhead that they introduce. Especially in web development, runtime overhead is a crucial aspect contributing to the application latency, and ultimately, the user experience. By shedding light on the current market, we intend to provide software architects with the necessary information to make informed decisions.

1 Introduction

During the last decade, we have seen radical changes in the way software is being built: Where traditional shrink-wrapped software undergoes yearly releases, new versions of cloud-based applications are released on a weekly if not daily basis (quoting Marissa Mayer in [Lig10]). This goes hand in hand with developers striving to be agile. In the spirit of lean development, design decisions are made as late as possible, a strategy that also applies to the design of the schema. Actually, the schema-flexibility of many NoSQL data stores is a driving force behind their popularity, even for applications where the expected data volume by itself does not justify using a NoSQL data store.

However, NoSQL data stores bring about their own challenges: As of today, there is no

```
@Entity
public class Profile {
    @Id
    int profileID;

    String firstname;
    String lastname;
    int year;
    String country;
    /* not showing the methods */
}
```

Figure 1: A Java class declaration of user profiles with JPA annotations.

standardized query interface. When building applications against NoSQL systems, calling the proprietary APIs condones a technical lock in. Especially in a new and volatile market such as today's NoSQL data stores, this can put the long-term success of a project at risk.

To avoid dependency on a particular system or vendor, developers commonly rely on object mapper libraries. These mappers build on the tradition of Object-Relational Mappers (ORMs) designed for interfacing with relational databases. They handle the mundane marshalling between objects in the application space and objects persisted in the data store. For instance, Figure 1 shows a Java class declaration for user profiles. The object mapper annotation `@Entity` conveniently declares that instances of this class can be persisted.

When building applications against relational databases, software architects can choose from a range of established and well-documented ORMs [MW12]. Traditionally, these ORMs handle the impedance mismatch between object-oriented programming languages and the relational model. In programming with Java, the Java Persistence API (JPA) including the Java Persistence Query Language (JPQL) has become state-of-the-art. JPA and JPQL are "standardized" within the Java Community Process [Jav09]. Using these APIs, database applications can become largely independent of ORM vendors.

It is a natural consequence that with the rising popularity of NoSQL data stores, some ORM vendors are extending their support to NoSQL data stores. Yet there are also new players in this market, offering special-purpose Object-NoSQL Mappers.

To the software architect, this raises beguiling questions:

- Which features are desirable in a mapper library when building applications against NoSQL data stores?
- Are the products offered today mature enough to be employed in *professional* application development?
- What is the runtime overhead imposed by object mapper libraries?

To answer these questions, we study a representative sample of Object-NoSQL Mappers (ONMs) for Java development as the main contribution of this article.

Structure. After reviewing the basics of NoSQL data stores and Object-NoSQL mapping in Section 2, we give an overview of the state-of-the-art and provide a comparative study of Object-NoSQL Mappers in Section 3. Besides the generic mapping capabilities, we analyze in Section 4 whether any schema management tasks are supported by Object-NoSQL Mappers beyond what the underlying NoSQL data store supports. The results of our performance evaluation are presented in Section 5. We then conclude with a summary and an outlook on future developments.

2 Foundations

We briefly recount the variety of NoSQL data stores, and proceed with an overview over Object-NoSQL mapping.

2.1 NoSQL Data Stores

NoSQL data stores vary in their data and query model, scalability, architecture, and persistence design. The most common categorization is by data model, distinguishing *key-value stores*, *document stores*, *column-family stores*, and *graph databases* [EFH$^+$11, Tiw13].

Key-value stores persist pairs of a unique key and an opaque value.

Document stores also store key-value pairs, yet the values are structured as "documents". This term connotes loosely structured sets of name-value pairs, usually in JSON (JavaScript Object Notation) [Ecm13] format or the binary representation BSON, a more type-rich format. Name-value pairs represent the properties of data objects. Names are unique within a document. Since documents are hierarchical, values may not only be scalar or appear as lists, but may contain nested documents.

Column-family stores manage records with properties. A schema declares property families, and new properties can be added to a property family ad hoc, on a per-record basis.

Graph databases provide operations on a graph data model with nodes (representing entities) and edges (representing relationships).

Object-NoSQL Mappers are available for all types of NoSQL data stores. In Section 3 we analyze selected object mapper libraries.

2.2 Object-NoSQL Mapping

From the application developer's point of view, Object-NoSQL Mappers follow the same ideas as Object-Relational Mappers, typically relying on annotations within class declarations in the application source code. We illustrate these annotation principles using a small social network example which will be extended in the next sections. Let us consider

Profile

profileID	firstname	lastname	year	country
4711	Miroslav	Klose	1978	DE

(a) A persisted object in a column-family store.

```
{
  "profileID"  : 4711,
  "firstname"  : "Miroslav",
  "lastname"   : "Klose",
  "year"       : 1978,
  "country"    : "DE"
}
```

(b) A persisted object in a document store.

Figure 2: The class declaration from Figure 1 implies the schema of persisted objects.

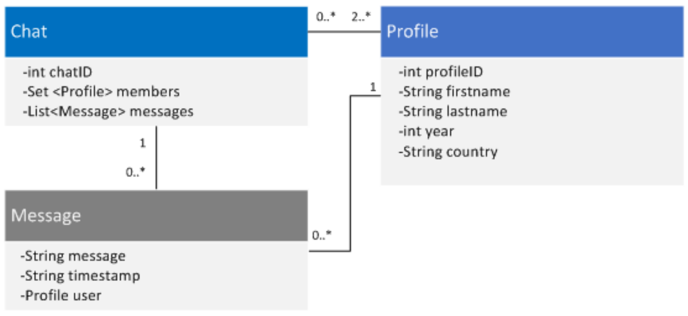


Figure 3: Class diagram for example scenario (extract).

Figure 1, illustrating the use of annotations in Java Persistence API (JPA) [Jav09]. Class `Profile` captures a user profile with a unique ID and user information (e.g., firstname and lastname). Due to annotation `@Entity` in line 1, `Profile` objects can be persisted. Annotation `@Id` in line 3 declares a class member attribute as the identifying key.

Figure 2 shows how instances of this class are persisted in different types of NoSQL data stores. Subfigure (a) shows an object persisted in a column-family store and subfigure (b) the same object persisted as JSON document in a document store. Evidently, the class declaration with the object mapper annotations also declares the schema of persisted objects.

Likewise, associations between classes can be annotated as *relationships* and carry cardinalities: Typically one-to-one, one-to-many, and many-to-many relationships are supported. To illustrate this, we extend our example by classes `Chat` and `Message`. Users can communicate together via messages within chats. Figure 3 shows the class diagram. `Message` objects have the same lifetime as the associated `Chat` object. We will come back to this point later in the discussion. Figure 4 contains the corresponding Java class declarations with JPA annotations. Line 6 of Figure 4(a) declares the `@ManyToMany` annotation for the many-to-many relationship between `Chat` and `Profile`. Analogously, the `@ManyToOne` association between `Message` and `Profile` is shown line 8 of Figure 4(b). Relationships can be unidirectional or bidirectional. Because we implement these relationships as unidirectional, no changes to class `Profile` are required (Figure 1).

```
1  @Entity
2  public class Chat {
3      @Id
4      int chatID;
5
6      @ManyToMany
7      Set<Profile> members;
8
9      @ElementCollection
10     List<Message> messages;
11 }
```

(a) Class declaration with `@ManyToMany` and `@ElementCollection` annotations.

```
1  @Embeddable
2  public class Message {
3
4
5      String message;
6
7      String timestamp;
8
9      @ManyToOne
10     Profile user;
11 }
```

(b) Embeddable class declaration with `@ManyToOne` annotation.

Figure 4: Java class declarations with different JPA annotations for relationships.

Another interesting point is the implementation of the one-to-many relationship between `Chat` and `Message`. Instead of using the `@One-To-Many` annotation, we use an ElementCollection. An ElementCollection in JPA defines a collection of instances of a basic type or embeddable class. The entities of the embeddable class have the same lifetime as their owner. Furthermore it is not necessary to manage a separate id for the instances of the embeddable class. An ElementCollection is declared via the `@ElementCollection` annotation in the owner class (see line 9 in Figure 4(a)) and the declaration `@Embeddable` for the embeddable class (see line 1 in Figure 4(b)).

Today, there is a large body of experience how to map from the object-oriented model to the relational database model, and most Object-Relational Mappers implement similar mapping strategies. In contrast, the mapping from an object-oriented class model to a NoSQL data store depends strongly on the underlying NoSQL data model. Furthermore, different mappings are possible within the same NoSQL data model (c.f. Section 2.1). For example, collections as well as one-to-many relationships can be embedded or referenced. An exhaustive discussion of mapping variants is therefore beyond the scope of this paper, and remains an understudied research area as of today. We point out interesting current limitations in mapping of some object-oriented modeling constructs in Section 3.

3 Comparison of Object-NoSQL Mappers

We now state our desiderata for Object-NoSQL Mappers (ONMs). After that, we present a selection of state-of-the-art ONM products and evaluate them w.r.t. our desiderata.

3.1 Requirements

Naturally, Object-NoSQL Mappers provide basic CRUD (Create, Read, Update, Delete) operations. In the interest of platform and vendor independence, the ONM should further offer a standardized query language. Yet for performance reasons, in particular to leverage proprietary APIs, it can be worthwhile if the ONM allows access to the query language native to the NoSQL data store.

If values are modified or objects are destroyed within the application, object mapper libraries generate the corresponding data store update respectively delete statements. Hence, updates and delete operations are usually executed on a single object. But for performance reasons it should be possible to execute update and delete operations on several objects in batch. Therefore, ONMs should support appropriated update and delete statements. Also for performance reasons, batch or bulk inserts should be provided.

NoSQL data stores can handle large volumes of data, exploiting massive parallelism, typically using MapReduce [DG04]. Therefore, ONMs should support MapReduce or some other parallel programming approach. Finally, in NoSQL application scenarios the usage of several data stores for different data within the same application become an important request. This scenario is described by the term *polyglot persistence*, coined by Martin Fowler [SF12]. Ideally, ONMs enable polyglot persistence, i.e. the usage of different NoSQL or relational data stores within the same application.

3.2 Overview over Object-NoSQL Mappers

The market for Object-NoSQL Mappers is still volatile. Various libraries are available, among them open source projects which no longer seem to be maintained. We therefore restrict our evaluation to projects where the last stable release is not older than one year.

For the programming language Java, the standardized Java Persistence API (JPA) with the Java Persistence Query Language (JPQL) has become state-of-the-art [Jav09]. Due to this fact and the popularity of Java in application development, we focus on Java ONMs.

We distinguish ONMs which support several NoSQL data stores (*Multi Data Store Mapper*) and ONMs which support only a single system (*Single Data Store Mapper*). While Single Data Store Mappers may cause a technological lock in, they offer interesting features and may display superior runtime performance.

With *EclipseLink* [Ecl14] and *DataNucleus* [Dat14a], we consider two very prominent libraries for object-relational and object-XML mapping that have been extended for object-NoSQL mapping. Hibernate, in particular, is one of the most popular ORMs. However, in contrast to EclipseLink and DataNucleus, object-NoSQL mapping has not been integrated into its main object mapper library yet. Instead, *Hibernate OGM* [Red14a] comes as its own library. *Kundera* [Imp14b] by Impetus is a ONM without "ORM-history" and was one of the earliest adopters in this market. *Morphia* [Mon14] is a proprietary Single Data Store Mapper for MongoDB. All mentioned mappers are available as open source.

	Multi Data Store Mapper				
	Hibernate OGM	Kundera	DataNucleus	EclipseLink	Morphia
Evaluated version	4.1 B 6	2.13	4.01	2.5.2	0.108
Key-value stores					
Infinispan	✓	–	–	–	–
Ehcache	✓	–	–	–	–
Redis	–	✓	–	–	–
Elasticsearch	–	✓	–	–	–
Oracle NoSQL	–	✓	–	✓	–
Document stores					
MongoDB	✓	✓	✓	✓	✓
CouchDB	✓	✓	–	–	–
Column-family stores					
Cassandra	–	✓	✓	–	–
HBase	–	✓	✓	–	–
Graph databases					
Neo4j Embedded	✓	✓	✓	–	–

Table 1: Distinguishing ONMs by the supported NoSQL data stores.

3.3 Feature Analysis

Table 1 lists the evaluated Object-NoSQL Mappers with the supported NoSQL data stores. Notably, the popular MongoDB system is supported by all evaluated ONMs.

Table 2 compares the ONMs by our desiderata. Basic **CRUD** operations are more or less supported by all evaluated products.

However, there are some modeling restrictions regarding *ElementCollections* (c.f. Section 2): DataNucleus does not support ElementCollections for CouchDB, Cassandra, HBase, and Neo4j [Dat14a]. Kundera does not support ElementCollections for CouchDB or Neo4j, yet this limitation is not evident in the current Kundera documentation. To handle this drawback, the elements of the collection have to be modeled as non-embedded entity class with own identity and a one-to-many relationship between the associated classes. Figure 5 shows the modified modeling of relationship between `Chat` and `Message` classes (c.f. Figure 4). Line 1 of Figure 5(b) contains the `@Entity` annotation instead of `@Embeddable` and line 3 and 4 the now necessary identifying attribute with its JPA annotation. The relationship between `Chat` and `Message` is now annotated with `@OneToMany` instead of `@ElementCollection` (line 9 in Figure 5(a)). As a consequence the application developer is now responsible to implement the lifetime dependencies between messages and chats in the application.

	Multi Data Store Mapper				
	Hibernate OGM	Kundera	DataNucleus	EclipseLink	Morphia
Create					
Single object	✓	✓	✓	✓	✓
Batch insert	✓	✓	✓	✓	✓
Read					
Query Language	JPQL*	JPQL*	JPQL, JDOQL	JPQL*	proprietary
Native Queries	✓	✓	–	✓	✓
MapReduce	–	–	–	–	✓
Update					
Single object	✓	✓	✓	✓	✓
Multiple objects	–	✓	–	✓	✓
Delete					
Single object	✓	✓	✓	✓	✓
Multiple objects	–	✓	✓	✓	✓
Polyglot Persistence	–	✓	(✓)	✓	–

Table 2: CRUD operations supported by Object-NoSQL Mappers.

Query Language Support Object-Relational Mappers target relational data stores, where we can rely on SQL as a standardized and well-understood query language. While not all relational databases implement the full SQL standard, and many vendors add their own extensions, there is nevertheless conformity in supporting a large, common SQL fragment represented within a object query language like JPQL. In contrast to relational databases, there is no standardized access to NoSQL data stores. Systems vary greatly, even in how they implement CRUD operations. Most systems do not support any joins. Many do not offer aggregate functions or the `LIKE` operator, again, some systems do.

This raises the question how Multi Data Store Mappers deal with this heterogeneity. Overall, there seem to be three approaches,

1. to offer only the particular subset of features that is implemented by *all* supported NoSQL data stores, i.e. the intersection of features, or
2. to distinguish by data store and to offer only the set of features implemented by a particular NoSQL data store, or
3. to offer the same set of features for all supported NoSQL data stores, possibly complementing missing features by implementing them inside the ONM library.

In the first approach, the ONM offers only the query language constructs translatable to query operators implemented by all supported NoSQL data stores. While we can write data store independent applications, the query capabilities are severely limited.

The second approach is to offer query language operators individually, depending on the functionality of the underlying NoSQL data store. This approach is chosen by Hibernate OGM, Kundera, and EclipseLink (marked with a * in Table 2). We can now make use of the full set of features implemented in each data store. However, this puts portability at

```
@Entity
public class Chat {
    @Id
    int chatID;

    @ManyToMany
    Set<Profile> members;

    @OneToMany
    List<Message> messages;
}
```

(a) Class declaration with `@OneToMany` annotation.

```
@Entity
public class Message {
    @Id
    int messageID;

    String message;
    String timestamp;

    @ManyToOne
    Profile user;
}
```

(b) Class declaration with `@Entity` instead of `@ElementCollection`.

Figure 5: Alternative modeling of `Chat-Message` relationship.

	Cassandra	HBase	MongoDB	CouchDB	Redis	Oracle-NoSQL	Neo4j
ORDER BY	–	–	✓	–	–	–	–
AND	✓	✓	✓	✓	✓	✓	✓
OR	– (✓with Lucene)	✓	✓	–	✓	✓	✓
BETWEEN	✓	✓	✓	✓	✓	✓	✓
LIKE	– (✓with Lucene)	✓	✓	–	–	–	✓
IN	–	–	–	–	–	–	–

Table 3: Kundera's support for JPQL constructs is data store specific [Imp14a].

risk: A feature supported in one data store may be not supported in another data store. For instance, in Kundera the `ORDER BY` operator is supported for MongoDB only [Imp14a]. Let us go into more detail for Kundera. As seen in Table 3, Kundera does not support the same set of JPQL constructs for all data stores. Hence, with this approach, application developers have to choose between functionality and portability.

To overcome these limitations, some ONMs compromise between the second and the third approach. They use third-party libraries to offer more functionality for some but not for all supported NoSQL data stores. As we can also see in Table 3, Kundera provides the `LIKE` operator for Cassandra although this operator is not natively supported. To do so, Kundera relies on Apache Lucene for an implementation of the `LIKE` operator [Imp14a].

Similarly, Hibernate OGM offers Hibernate Search [Red14c] to index and query objects (entities), as well as to run full-text queries. The latter feature is driven by Apache Lucene, extended with an object oriented abstraction including an object oriented, domain-specific query language [Red14a]

The third and most advanced approach is to offer the same set of query operators for *all* NoSQL data stores. If a feature is not supported by a data store it has to be realized within the Object-NoSQL Mapper. This allows for portable applications using the full power of the standardized query language offered by the ONM. Naturally, when an operation is implemented outside of the data store, runtime performance may decrease.

In this spirit, DataNucleus evaluates JPQL or JDOQL queries in memory. Of course, DataNucleus will leverage the capabilities of the data store as much as possible, and complement the missing query evaluation steps [Dat14c, Dat14b]. Due to this query shredding technique, DataNucleus is currently the only library among the evaluated ONMs that supports aggregate functions. Hibernate OGM has announced similar functionality for future releases [Red14a, Red14b] using the Teiid[1] engine.

Unfortunately, none of the analyzed Multi Data Store Mappers offers a **MapReduce API** so far. However, Kundera as well as Hibernate OGM have announced MapReduce support for future releases [Red14b].

Finally, some words on **transactions**. Using JPA or JDO Object-NoSQL Mappers, it is possible to define transactions. However, whether this is feasible depends on the underlying NoSQL data store, since many NoSQL data stores do not support transactions. Object mappers (ORMs as well as ONMs) normally use complex caching mechanisms. As long as the data is stored in the cache only, a transaction can be rolled back. Yet once the data has been flushed to the NoSQL data store, this is no longer possible (except with data stores which support ACID transactions, such as Neo4j or Infinispan).

4 Support for Schema Management with Object-NoSQL Mappers

Most NoSQL data stores are schema-less or schema-flexible. This offers great flexibility in the early stages of application development. Yet for long-term schema management, the tools currently provided by the vendors of NoSQL data stores are too rudimentary [KSS14]. Thus, any but the most basic data management tasks (such as persisting an object after renaming a property) usually require custom coding. However, Object-NoSQL Mappers also support certain schema management tasks. In the upcoming discussion we refer to the schema management requirements proposed in [SKS13] and [KSS14].

Schema definition Unlike relational databases, schema-less data stores typically do not manage a full-blown data dictionary. Yet even then, the application source code commonly contains class declarations. When developers use Object-NoSQL Mappers, the object mapper annotations not only declare the structure of persisted entities, but also the relationships between them. (Depending on the underlying NoSQL data store, the mapping strategy will vary.) So even when the NoSQL data store itself does not manage an explicit schema, the schema is nevertheless implicit in the class annotations in the application code.

[1]teiid.jboss.org

```
 1  @Entity
 2  public class Profile {
 3      @Id
 4      int profileID;
 5      String firstname;
 6      String lastname;
 7
 8      @Column(name="year")
 9      int yearOfBirth;
10      String country;
11  }
```

(a) Renaming an attribute with `@Column`.

```
{
  "profileID"   : 4711,
  "firstname"   : "Miroslav",
  "lastname"    : "Klose",
  "year"        : 1978,
  "country"     : "DE"
}
```

(b) State of the (unchanged) persisted object in a document store.

Figure 6: Renaming attributes in Multi Data Store Mappers.

Validation To a certain degree, Object-NoSQL mappers also validate data against the implicit schema. As long as all access paths to the persisted data are using the same object mapper class definitions, the persisted data fits the class model. However, when the application code evolves, so does the class model. Thus, a persisted object may either fit the current or some historic class model, which motivates the next paragraph.

Schema evolution Schema evolution operations include adding, deleting, renaming, and updating entity classes. The latter covers all aspects of changing names or types of attributes as well as moving or copying attributes between entity classes. Such denormalization operations are essential in systems that do not support joins in query evaluation. Furthermore, relationships may be added or deleted, or the cardinality of relationships may be changed. In the following, we focus on these operations and their realization in Object-NoSQL Mappers in detail.

Adding an *attribute* works very well with Object-NoSQL Mappers. When loading a persisted object, the new attribute is added and initialized. Upon persisting the entity, the new attribute is persisted as well. This form of migrating data one entity at-a-time, at the time when it is loaded, is known as *lazy migration* [SKS13].

Removing a class member attribute from a class declaration results in *lazily deleting* the *attribute*, since it will no longer be loaded into the application space. Again, this only affects entities that are loaded at application run time, all other entities remain unchanged.

For *renaming attributes*, there are different strategies. In Multi Data Store Mappers, it is only possible to implement an aliasing approach using the `@Column` annotation [Jav09]. The new name is only available in the application space, while the persisted object itself is not changed. As an example let us rename the attribute `year` to `yearOfBirth` in our `Profile` class (c.f. Section 1, Figure 1). Figure 6 shows the JPA declaration `@Column` annotation in line 8 and the corresponding (unchanged) persisted object.

```
1  @Entity
2  public class Profile {
3      @Id
4      int profileID;
5      String firstname;
6      String lastname;
7  
8      @AlsoLoad("year")
9      int yearOfBirth;
10     String country;
11 }
```

(a) Renaming an attribute with `@AlsoLoad`.

```
{
  "profileID"  : 4711,
  "firstname"  : "Miroslav",
  "lastname"   : "Klose",
  "yearOfBirth": 1978,
  "country"    : "DE"
}
```

(b) State of the (unchanged) persisted object in MongoDB after lazy migration.

Figure 7: Lazily renaming an attribute using Morphia annotations.

In contrast, the Single Data Store Mapper Morphia provides an annotation `@AlsoLoad` which lazily migrates persisted objects. Figure 7(a) shows a class declaration with the `@AlsoLoad` annotation in line 8 and the state of the object after persisting in Figure 7(b). Thus, on loading persisted objects with the now deprecated attribute name, the attribute name is changed. The change becomes permanent upon persisting the object in MongoDB.

None of the analyzed Object-NoSQL Mappers *copy* or *move attributes between* objects. A workaround is to use JPA's lifecycle annotations [Jav09]. For instance, methods annotated with `@PostLoad` are executed after the object has been loaded from the data store into the object mapper application context. Application developers can thus implement arbitrarily complex modification operations within these methods on a per-object basis.

We next consider operations beyond modifying single attributes. *Adding* a new *entity class* is straightforward. After *deleting entities* by removing the class declaration from the application source code, the entities are no longer accessible for the application. However, any persisted objects will remain in the NoSQL data store. Unfortunately, there is no systematic support for purging this data in the NoSQL data store in the Object-NoSQL Mapper libraries evaluated by us.

Renaming entity classes works with a similar aliasing approach like renaming an attribute, and is based on the `@Table` annotation [Jav09]. Again, the new name is used in the object-oriented application context and the old entity class name is used in the data store for existing entities, as well as when new objects are added. This is illustrated in Figure 8.

Adding and *deleting relationships* is similar to adding and deleting attributes in the analyzed Object-NoSQL Mappers. *Updating relationships*, i.e. changing the cardinality of the relationship, is not support by any of the evaluated mappers.

Data migration Apart from adding and deleting attributes or relationships w.r.t. single persisted objects, the analyzed Multi Data Store Mappers do not provide further data mi-

```
@Entity
@Table(name="Profile")
public class User {
    @Id
    int profileID;
    String firstname;
    ...
}
```

(a) The Java class declaration

Profile

profileID	firstname	lastname	year	country
4711	Miroslav	Klose	1978	DE
5000	Andre	Hahn	1990	DE

(b) State of the table (table name unchanged) in a column-family store after adding a new object.

Figure 8: Renaming entity classes with the `@Table` annotation.

gration operations. For renaming operations they do not change the data in the NoSQL data store. The Single Datastore Mapper Morphia at least is able to migrate lazily, e.g., in renaming attributes.

In general, life-cycle annotations like `@PostLoad` may be used to implement lazy and complex migration tasks on a per-object basis. Nevertheless, there is no systematic and well-principled support for automated data migration in the Object-NoSQL-Mappers studied here. According to [Red14b], Hibernate has a migration engine with support for lazy migration on its roadmap.

Schema versions Object-NoSQL Mappers can handle different variants of objects. For example, after adding an attribute, objects already persisted without this attribute can still be loaded. Likewise, after deleting an attribute, objects containing the deprecated attribute can be loaded. However, there is no explicit management of different schema versions, e.g., using version numbers.

Schema extraction If data was persisted without the support of an Object-NoSQL Mapper, it can still be interesting to extract the schema from the persisted data. Apart from gaining valuable insight into the data's structure, this would allow developers to conveniently access the data from Object-NoSQL Mapper libraries. In [KSS15] we sketch out this vision in greater detail and present a schema extraction algorithm.

Conclusion Object-NoSQL Mappers extend NoSQL data stores by implicit schema definition and implicit schema validation. Basic schema evolution operations, such as adding and deleting attributes or relationships (and renaming attributes in Morphia), are currently supported. However, when several schema evolution operations are to be applied, developers need to resort to custom code. A more general support for schema evolution in Object-NoSQL-Mappers is certainly desirable.

5 Performance of Object-NoSQL Mappers

In the decision which Object-NoSQL Mapper to use, their impact on the application runtime performance is a vital criterion. After all, the response time in interactive applications is crucial for the user experience. Therefore, two questions drive our experiments:

- Will an Object-NoSQL Mapper noticeably decrease runtime performance compared to the native API, as provided by the NoSQL data store vendor?
- Do different Object-NoSQL Mappers show different runtime performance?

For investigating these questions, we have generated synthetic data for our running example and we have further defined a set of queries, since we could not employ the YCSB benchmark [CST$^+$10]: YCSB does not define an interesting data model with relationships between entities, and does not define any queries apart from basic CRUD operations.

5.1 Test Setup

We have experimentally evaluated the Object-NoSQL Mappers discussed previously against a range of NoSQL data stores. The version numbers for the ONMs are listed in Table 1. We ran each ONM against the following NoSQL data stores: MongoDB 2.4.6, CouchDB 1.0.1, Cassandra 2.0.5, and HBase 0.94.11, all running on Ubuntu 12.04, always provided that the ONM supports the system. The NoSQL data stores were used as-is, without any individual tuning.

All experiments were run on the same hardware with a typical NoSQL cluster environment consisting of commodity machines: Client and server processes ran on a single machine, each within the same local network. The Dell PowerEdge C6220 machines each have 2 Intel Xeon E5-2609 (4 Cores each), 32 GB RAM, and 4 x 1 TB SATA 7.2 k HDs. Since our focus was not to test the performance of the underlying NoSQL data stores, but the runtime overhead introduced by the Object-NoSQL Mappers, we consider it sufficient to evaluate the data stores in single-node configuration. The runtime performance was measured by profiling calls to ONM methods by manually injected statements.

5.2 Use Case and Synthetic Data

Figure 9 shows the class model on which we base our experiments. The model is an extension of our social network example and includes relationships with different types of functionalities. Due to the limitations regarding ElementCollections (c.f. Section 3), the relationships were implemented as one-to-many-relationships (c.f. Figure 5). This allows for a fair comparison of all ONMs. We generated the same test data for all experiments, ranging between 10^3 and 10^6 profiles. Each `Profile` has five `WallEntry` objects, each with two `Comments` on average. Further, each `Profile` has three `Chat` objects, and each `Chat` contains five `Message` objects on average.

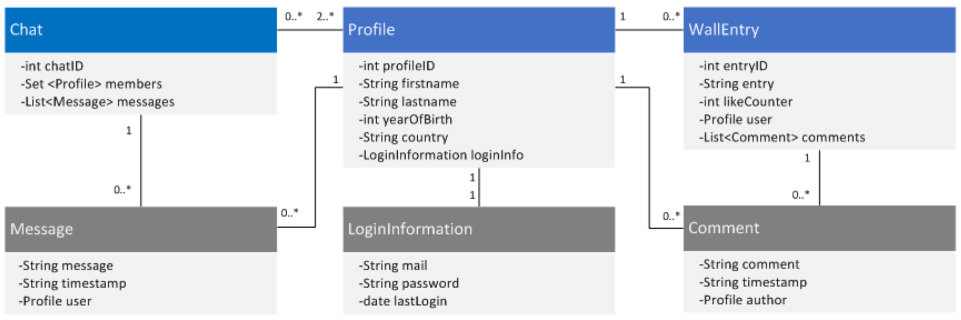


Figure 9: Class diagram for our application scenario.

Due to space limitations we can only present the highlights of our performance evaluation and refer to [Hau14] for the unabridged results. In the following, we only report the results for MongoDB, since it is the data store supported by all investigated mappers. Whenever the choice of NoSQL data store has a noticeable impact, we explicitly point this out.

5.3 Read Performance

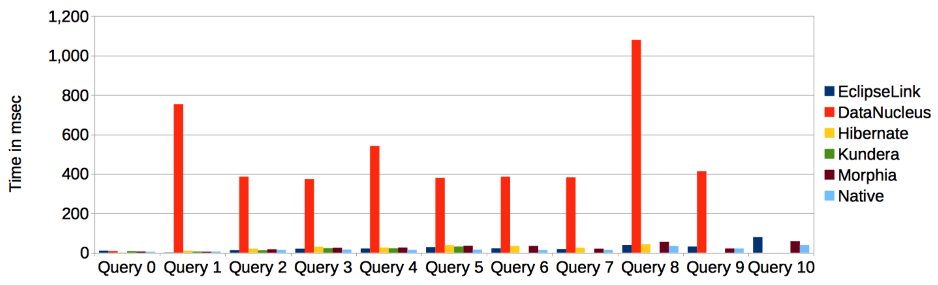
Appendix A lists the queries. Queries 0 through 9 are selections with varying predicates, e.g. atomic predicates with different data types, testing for equality and inequality, conjunction, disjunction, including basic text search functionality. Query 10 computes a join.

Differences in runtime overhead The runtime overhead of the object mappers on top of MongoDB[2] is shown in Figure 10. We refrain from listing the results for the other NoSQL data stores, since they do not contribute new insights. The runtimes in Figure 10 were collected by evaluating the queries over 10,000 Profiles. Some results are not shown, since not all ONMs support all queries.[3]

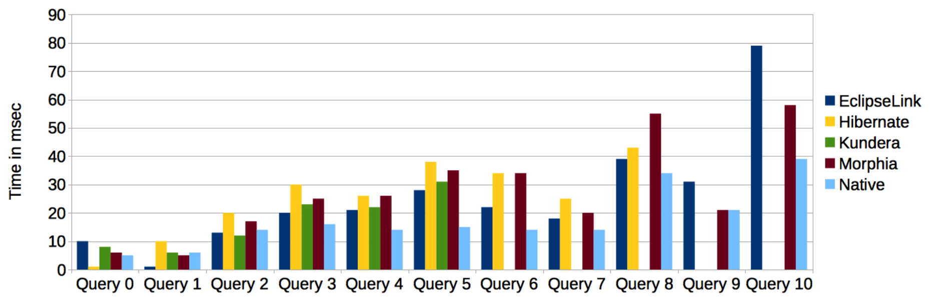
To our surprise, DataNucleus is significantly slower than its competitors on Queries 1 through 10, which all use JPQL syntax. In Figure 10(b) we omit DataNucleus to better show the differences between the remaining ONMs. The same effect occurs when using DataNucleus with other NoSQL data stores. It remains to be determined why the runtime overhead introduced by DataNucleus is considerably higher for JPQL queries. At this point it is merely a conjecture, yet DataNucleus is the only library under investigation that implements the same API for all supported NoSQL data stores, complementing missing features that are not provided by the data store inside the ONM library (c.f. Section 3.3). It may well be that this generality comes at the cost of a considerable performance overhead.

[2]The "native" access baseline for MongoDB is the MongoDB Query Builder API.

[3]DataNucleus does not support Query 10. Kundera returns the wrong number of results for Query 6 and the empty result for Queries 7-10 (for Queries 8-10 the reason is an incorrect implementation of the `LIKE` operator). Hibernate OGM throws exceptions for Queries 9 and 10, stating that they are too complex to be evaluated.



(a) Including DataNucleus.



(b) Excluding DataNucleus.

Figure 10: Query performance on 10,000 profiles (MongoDB).

Apart from DataNucleus, the outlier in this study, the overhead of ONMs for read operations is within about a factor of two when compared to the native access, the baseline in our evaluation. Thus, the other ONM libraries all show similar runtime performance.

Syntax sensitivity As an interesting finding, we point out the runtime differences between the two equivalent Queries 0 and 1. Both queries retrieve a single `Profile` based on the `profileID`. Query 0 uses the JPA interface method `find`, whereas Query 1 uses JPQL syntax. EclipseLink and Hibernate OGM differ in their runtime behavior in Figure 10(b). Hibernate is more efficient when using the `find` operator, whereas, surprisingly, EclipseLink is more efficient in using the JPQL interface. Thus, the query language is not as declarative as may be assumed, and the choice of query operator can have a noticeable impact on the runtime performance.

Scale up over larger datasets Figure 11 shows the scale up for Query 4 over larger data sets in MongoDB. Except for DataNucleus, the ONMs are about 1.4 to 2 times slower than access via the native API. The behavior for the other queries is similar.

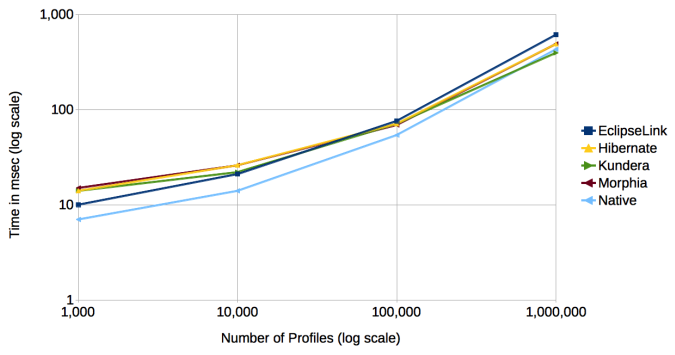


Figure 11: Scale up in evaluating Query 4 (MongoDB), not showing DataNucleus.

Matters of robustness and answer quality Unfortunately, the robustness and quality of query evaluation were not to our full satisfaction. For instance, Kundera returns correct results when processing 100,000 profiles with Query 4 on both MongoDB and Cassandra. Yet on Cassandra, the same query returns a wrong number of results when processing 1,000,000 profiles.

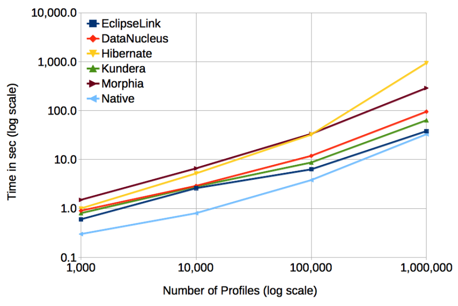
This is an alarming discovery, and shows that when choosing among ONM libraries, software architects need to test them for answer quality and reliability. Furthermore, quality assurance must go beyond simple unit tests, but be conducted on larger data sets, since some problems only reveal themselves when processing data at scale. To make matters worse, the correctness of answers depends on the underlying data store.

5.4 Write Performance

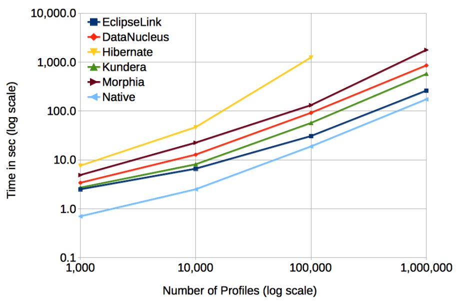
Next, we evaluate insert, update, and delete operations. It turns out that the runtime overhead for write operations is more noticeable than for read operations. Also, we now notice significant differences between libraries. Again, we only report our experiments on MongoDB, since the other NoSQL data stores do not contribute new insights.

Insertion We evaluate the insert performance for entities as well as for all relationship types of the class model shown in Figure 9. We are particularly interested in the results for inserting single entities and 1:1 relationships, as well as 1:N relationships. Figure 12(a) shows the scale up for inserting `Profiles` with `LoginInformation`. Because this is a 1:1 relationship, the matching data is stored within the same document. For 100,000 profiles, the slowest ONM is about 10 times slower than the fastest one, and about 15 times slower than access via the native API.

Figure 12(b) reports the result for inserts with a 1:N relationship, namely `Profile` objects with `WallEntrys`. The experiment for 1,000,000 `Profile` objects with Hibernate

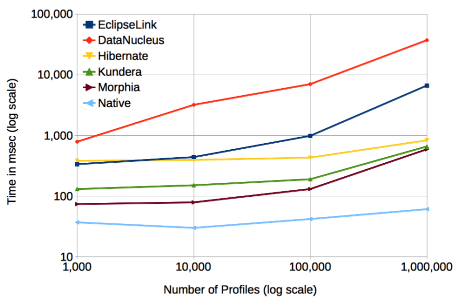


(a) Inserting Profiles.

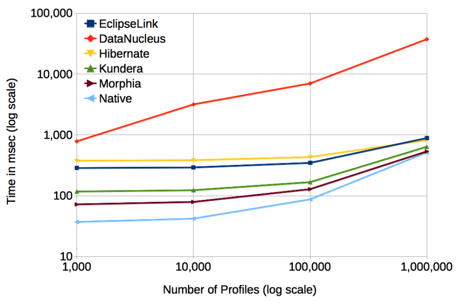


(b) Inserting 1:N relationships.

Figure 12: Scale up for insertion (MongoDB).



(a) Updating 25 Profiles.



(b) Deleting 25 Profiles.

Figure 13: Update and delete operations (MongoDB).

OGM was aborted after only 500,000 successful inserts within 10 hours. Obviously, the runtime differences between the ONMs are significant.

Updates and deletes For updates, we ran the following experiment. Query 4 returns 25 objects on our sample data, for all data sets of all sizes. These 25 objects are then updated within the application logic. Afterwards, the ONM generates the corresponding data store updates. We followed the same approach to test deletes. Figure 13(a) shows the runtime overhead for updates, while Figure 13(b) shows the results for deletes. Again, there is a significant gap between the native access and the mapper libraries. Moreover, the Object-NoSQL Mappers differ in the runtime overhead for updates and deletes.

Summary on writes In general, the effects on insert performance with data stores other than MongoDB are comparable, particularly the ranking of the different Object-NoSQL Mappers is very similar. However, the runtime differences between the slowest and the fastest ONM and especially the runtime overhead using any ONM instead the native API depends strongly on the underlying NoSQL data store (see [Hau14] for details).

6 Summary

This paper gives an overview of the state-of-the-art in Java Object-NoSQL Mappers. We have studied very prominent, popular, and sophisticated libraries. Our analysis shows that it is safe to expect that a contemporary ONM provides basic CRUD operations.

However, the supported query languages differ greatly in their expressiveness. Which query operators are offered often depends on the capabilities of the underlying NoSQL data stores. This is a fundamental limitation for application portability.

During our experiments, we encountered some unexpected glitches: We ran into cases where query operators had not yet been implemented, even though the documentation describes them in full. Also, some query operators are not implemented with the semantics described in the documentation. As a consequence, developers must not rely on the documentation alone, and should show extra care when conducting their test cases.

Nevertheless, it is indisputable that application development with ONMs has its benefits:

- ONMs provide a (currently still restricted) vendor independent query language, one of the greatest drawbacks when working with NoSQL data stores.
- Most ONMs support several NoSQL (and relational) data stores within the same application (polyglot persistence).
- ONMs extend NoSQL data stores with a form of implicit schema definition. They perform basic schema validation, as well as some basic schema evolution operations: Adding and deleting attributes or relationships are commonly supported.

Our experiments reveal that in reading data, there is only a small gap between native access and the Object-NoSQL Mappers for the majority of the evaluated products. Yet in writing, object mappers introduce a significant overhead. This has to be considered in the decision whether to use a Object-NoSQL Mapper, and which library to choose in particular.

Despite their current limitations and drawbacks, Object-NoSQL Mapper may actually come to the rescue for many desperate application developers who find themselves struggling with schemaless and proprietary NoSQL data stores.

Exciting and new features, such as more powerful query engines, MapReduce support, and improved support for data migration have been announced. Thus, it will be interesting to watch the developments in the ONM market over time. Due to the increasing importance of scripting languages in web applications, it could be an interesting next step to evaluate the market for ONMs for languages like JavaScript, Ruby, or Python.

References

[CST$^+$10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proc. SoCC 2010, Indianapolis, Indiana, USA, June, 2010*, pages 143–154, 2010.

[Dat14a] DataNucleus. *DataNucleus AccessPlatform: Datastore Feature Support*, 2014. `www.datanucleus.org/products/accessplatform_4_0/datastores/datastore_features.html`.

[Dat14b] DataNucleus. *DataNucleus AccessPlatform: JPA : JDOQL Queries*, 2014. `www.datanucleus.org/products/accessplatform_4_0/jdo/jdoql.html`.

[Dat14c] DataNucleus. *DataNucleus AccessPlatform: JPA : JPQL Queries*, 2014. `www.datanucleus.org/products/datanucleus/jpa/jpql.html`.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. OSDI, San Francisco, California, USA, December, 2004*, pages 137–150, 2004.

[Ecl14] EclipseLink. *Understanding EclipseLink 2.6*, 2014. `http://www.eclipse.org/eclipselink/documentation/2.6/eclipselink_otlcg.pdf`.

[Ecm13] Ecma International. The JSON Data Interchange Format, 2013. `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf`.

[EFH$^+$11] Stefan Edlich, Achim Friedland, Jens Hampe, Benjamin Brauer, and Markus Brückner. *NoSQL: Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken*. Carl Hanser Verlag, 2011.

[Hau14] Thomas Hauf. Vergleich von Object to NoSQL Mappern. Master's thesis, University of Applied Sciences Darmstadt, 2014.

[Imp14a] Impetus. *Kundera JPQL*, 2014. `https://github.com/impetus-opensource/Kundera/wiki/JPQL`.

[Imp14b] Impetus. *Kundera Wiki*, 2014. `https://github.com/impetus-opensource/Kundera/wiki`.

[Jav09] Java Persistence 2.0 Expert Group. *JSR 317: Java Persistence 2.0*, 2009.

[KSS14] Meike Klettke, Stefanie Scherzinger, and Uta Störl. Datenbanken ohne Schema? Herausforderungen und Lösungs-Strategien in der agilen Anwendungsentwicklung mit schemaflexiblen NoSQL-Datenbanksystemen. *Datenbank-Spektrum*, 14(2):119–129, 2014.

[KSS15] Meike Klettke, Stefanie Scherzinger, and Uta Störl. Schema Extraction and Structural Outlier Detection for JSON-based NoSQL Data Stores. In *Proc. 16. BTW, March, 2015 Hamburg, Germany*, 2015.

[Lig10] Sam Lightstone. *Making it Big in Software*. Prentice Hall, 2010.

[Mon14] MongoDB. *Morphia Wiki*, 2014. `https://github.com/mongodb/morphia/wiki`.

[MW12] Bernd Müller and Harald Wehr. *Java Persistence API 2: Hibernate, EclipseLink, OpenJPA und Erweiterungen*. Carl Hanser Verlag, 2012.

[Red14a] Red Hat. *Hibernate OGM Reference Guide 4.1.0*, 2014. http://docs.jboss.org/hibernate/ogm/4.1/reference/en-US/pdf/hibernate_ogm_reference.pdf.

[Red14b] Red Hat. Hibernate OGM Roadmap, 2014. http://hibernate.org/ogm/roadmap/.

[Red14c] Red Hat. *Hibernate Search: Apache Lucene Integration Reference Guide*, 2014. http://docs.jboss.org/hibernate/stable/search/reference/en-US/pdf/hibernate_search_reference.pdf.

[SF12] Pramod J. Sadalage and Martin Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison Wesley, 2012.

[SKS13] Stefanie Scherzinger, Meike Klettke, and Uta Störl. Managing Schema Evolution in NoSQL Data Stores. In *Proc. 14th International Symposium on Database Programming Languages (DBPL 2013), August, 2013, Riva del Garda, Trento, Italy*, 2013.

[Tiw13] Shashank Tiwari. *Professional NoSQL*. O'Reilly, 2013.

A Appendix: Queries used in our experiments

0. `find( Profile,` *intvalue* `)`
1. `SELECT p FROM Profile p`
   `WHERE p.profileID=` *intvalue*
2. `SELECT p FROM Profile p`
   `WHERE p.firstname=` *stringvalue* `AND p.lastname=` *stringvalue*
3. `SELECT p FROM Profile p`
   `WHERE p.yearOfBirth=` *intvalue*
4. `SELECT p FROM Profile p`
   `WHERE p.lastname=` *stringvalue*
5. `SELECT p FROM Profile p`
   `WHERE p.yearOfBirth>=` *intvalue* `AND p.yearOfBirth<=` *intvalue*
6. `SELECT p FROM Profile p`
   `WHERE p.lastname=` *stringvalue* `AND`
   `p.firstname <>` *stringvalue* `AND p.firstname <>` *stringvalue*
7. `SELECT p FROM Profile p`
   `WHERE p.lastname=` *stringvalue* `AND`
   `( p.firstname =` *stringvalue* `OR p.firstname =` *stringvalue* `)`
8. `SELECT w FROM WallEntry w`
   `WHERE w.likeCounter >=` *intvalue* `AND w.entry LIKE %`*stringvalue*`%`
9. `SELECT p FROM Profile p`
   `WHERE p.loginInfo.mail LIKE %@` *stringvalue* `AND`
   `p.country IN (` *stringvalue* `,` *stringvalue* `)`
10. `SELECT w FROM WallEntry w JOIN w.comments c`
    `WHERE c.comment LIKE %`*stringvalue*`% AND w.entry LIKE %`*stringvalue*`%`