Managing Schema Evolution in NoSQL Data Stores

Stefanie Scherzinger
Regensburg University
of Applied Sciences
stefanie.scherzinger@hs-regensburg.de

Meike Klettke
University of Rostock
meike.klettke@uni-rostock.de

Uta Störl
Darmstadt University
of Applied Sciences
uta.stoerl@h-da.de

Abstract

NoSQL data stores are commonly schema-less, providing no means for globally defining or managing the schema. While this offers great flexibility in early stages of application development, developers soon can experience the heavy burden of dealing with increasingly heterogeneous data. This paper targets schema evolution for NoSQL data stores, the complex task of adapting and changing the implicit structure of the data stored. We discuss the recommendations of the developer community on handling schema changes, and introduce a simple, declarative schema evolution language. With our language, software developers and architects can systematically manage the evolution of their production data and perform typical schema maintenance tasks. We further provide a holistic NoSQL database programming language to define the semantics of our schema evolution language. Our solution does not require any modifications to the NoSQL data store, treating the data store as a black box. Thus, we want to address application developers that use NoSQL systems as database-as-a-service.

Categories and Subject Descriptors H.2.3 [*Database Management*]: Languages

General Terms NoSQL data stores, schema evolution

Keywords API for data stores, schema evolution language, schema management, eager migration, lazy migration, schema versioning

1. Introduction

The classic database textbook dedicates several chapters to schema design: Carefully crafting an abstract model, translating it into a relational schema, which is then normalized. While walking their students through the course, scholars emphasize again and again the importance of an anticipatory, holistic design, and the perils of making changes later on. Decades of experience in writing database applications have taught us this. Yet this waterfall approach no longer fits when building today's web applications.

During the last decade, we have seen radical changes in the way we build software, especially when it comes to interactive, web-based applications: Release cycles have accelerated from yearly releases to weekly or even daily, new deployments of beacon applications such as Youtube (quoting Marissa Meyer in [22]). 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. This also applies to the schema. Fields that *might* be needed in the future are not added presently, reasoning that until the next release, things might change in a way that would render the fields unnecessary after all. It is partly due to this very need for more flexibility, that schema-free NoSQL data stores have become so popular. Typically, developers need not specify a schema up front. Moreover, adding a field to a data structure can be done anytime and at ease.

Scope of this work. We study aspects of schema management for professional web applications that are backed by NoSQL data stores. Figure 1 sketches the typical architecture. All users interact with their own instance of the application, e.g. a servlet hosted by a platform-as-a-service, or any comparable web hosting service. It is established engineering practice that the application code uses an object mapper for the mapping of objects in the application space to the persisted entities.

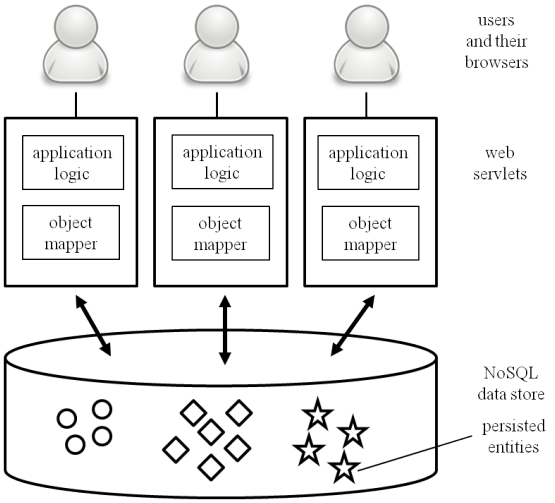

Figure 1. Architecture of an interactive web application.

We further assume that the NoSQL data store is provided as database-as-a-service, so we have no way of configuring or extending it. Our work addresses this important class of applications. Of course, there are other use cases for employing NoSQL technology, yet they are not the focus of our work.

Case Study: Blogging applications. We introduce a typical example of a professional web application: An online blog. In the spirit of shipping early and often, both the features of our application as well as the data will evolve.

We use a NoSQL data store, which stores data as entities. We will establish our terminology in later chapters, and make do with a hand-wavy introduction at this point. Each entity has an entity key, which is a tuple of an entity kind and an identifier. Each entity further has a value, which is a list of properties:

```
(kind, id) = { comma-separated list of properties }
```

Let us dive right in. In our first release, users publish blogs (with title and content) and guests can leave comments on blogposts. For each blogpost, information about the author and the date of the post is stored. In the example below, we use a syntax inspired

by JSON [21], a lightweight data-interchange format widely used within web-based applications.

```
(blogpost, 007) = {
 title: "NoSQL Data Modeling Techniques",
 content: "NoSQL databases are often ...",
 author: "Michael",
 date: "2013-01-22",
 comments: [
 { comment-content: "Thanks for the great article!",
   comment-date: "2013-01-24" },
 { comment-content: "I would like to mention ...",
   comment-date: "2013-01-26" } ] }
```

Soon, we realize that changes are necessary: We decide to support voting, so that users may "like" blogposts and comments. Consequently, we expand the structure of blogposts and add a "likes" counter. Since we have observed some abuse, we no longer support anonymous comments. From now on, users authenticate with a unique email address. Users may choose a username (`user`) and link from their comments to their website (`url`), as well as specify a list of interests. Accordingly, we add new fields.

We take a look at a new data store entity (`blogpost, 234708`) and the state of an older entity (`blogpost, 007`) that has been added in an earlier version of the application. For the sake of brevity, we omit the data values:

```
(blogpost, 234708) = {
 title, content, author, date, likes,
 comments [
  { comment-content, comment-date, comment-likes,
    user, email, url, interests[ ] } ] }

(blogpost, 007) = {
 title, content, author, date,
 comments [
  { comment-content, comment-date },
  { comment-content, comment-date } ] }
```

Next, we decide to reorganize our user management. We store user-related data in separate `user` entities. These entities contain the user's `login`, `passwd`, and `picture`.

During this reorganization we further rename `email` to `login` in `blogpost` entities. The `interests` are moved from `blogpost` to the `user` entities, and the `url` is removed. Below, we show blogpost (`blogpost, 331175`) of this new generation of data, along with old generation blogposts that were persisted in earlier versions of the application. The structural differences are apparent.

```
(blogpost, 331175) = {
 title, content, author, date, likes,
 comments [
  { comment-content, comment-date,
    comment-likes, user, login } ] }
(user, 42) = { login, passwd, interests[ ], picture }

(blogpost, 234708) = {
 title, content, author, date, likes,
 comments [
  { comment-content, comment-date, comment-likes,
    user, email, url, interests[ ] } ] }

(blogpost, 007) = {
 title, content, author, date,
 comments [
  { comment-content, comment-date },
  { comment-content, comment-date } ] }
```

After only three releases, we have accumulated considerable technical debt in our application code. It is now up to the developers to adapt their object mappers and the application logic so that all three versions of blogposts may co-exist.

Whenever a blogpost is read from the data store, the application logic has to account for the heterogeneity in the comments: Some comments do not have any user information, while others have information about the user identified by `email` (along with other data). A third kind of `blogpost` contains `comments` identified by the user's `login`. If new generation comments are added to old generation blogposts, we produce even a fourth class of blogposts.

Not only does this introduce additional code complexity, it also increases the testing effort. With additional case distinctions, a good code coverage in testing becomes more difficult to obtain.

In an agile setting where software is shipped early and often, developers would rather spend their time writing new features than fighting such forms of technical debt. At the same time, the NoSQL data store offers little, if any, support in evolving the data along with the application. Our main contribution in this paper is an approach to solving these kinds of problems.

Schema evolution in schema-less stores. While the sweet spot of a schema-less backend is its flexibility, this freedom rapidly manifests in ever-increasing technical debt with growing data structure entropy. Once the data structures have degenerated, a NoSQL data store provides little support for getting things straightened out.

Most NoSQL data stores do not offer a *data definition language* for specifying a global schema (yet some systems, such as Cassandra, actually do [1]). Usually, they merely provide basic read- and write operations for manipulating single entities, delegating the manipulation of sets of entities completely to the application logic. Consequently, these systems offer no dedicated means for migrating legacy entities, and developers are referred to writing batch jobs for data migration tasks (e.g. [23]). In such batch jobs, entities are fetched one-by-one from the data store into the application space, modified, and afterwards written back to the store. Worse yet, since we consider interactive web applications, migrations happen while the application is in use. We refer to data migration in batches as *eager migration*, since entities are migrated in one go.

Alas, for a popular interactive web-application, the right moment for migrating all entities may never come. Moreover, a large-scale data store may contain legacy data that will never be accessed again, such as stale user accounts, blogposts that have become outdated, or offers that have expired. Migrating this data may be a wasted effort, and expensive, when you are billed by your database-as-a-service provider for all data store reads and writes.

As an alternative, the developer community pursues what we call a *lazy* data migration strategy. Entities of the old and new schema are allowed to co-exist. Whenever an entity is read into the application space, it can be migrated. Effectively, this will migrate only "hot" data that is still relevant to users. For instance, the Objectify object mapper [27] has such support for Google Datastore [15]. However, all structure manipulations require custom code. As of today, there is no systematic way to statically analyze manipulations before executing them. Moreover, from a database theory point-of-view, lazy migration is little understood (if at all). This makes lazy migration a venture that, if applied incorrectly on production data, poses great risks. After all, once entities have been corrupted, there may be no way to undo the changes.

Desiderata. What is missing in today's frameworks is a means to systematically manage the schema of stored data, while at the same time maintaining the flexibility that a schema-less data store provides. What we certainly cannot wish for is a rigorous corset that ultimately enforces a relational schema on NoSQL data stores.

Most systems do provide some kind of data store viewer, where single entities can be inspected, and even modified, or data can be deleted in bulk (e.g. [14]). Yet to the best of our knowledge, there is no schema management interface that would work *across* NoSQL systems from different providers, allowing application administra-

tors to manage their data's structure systematically. This entails basic operations such as adding or deleting fields, copying or moving fields from one data structure to another. From studying the discussions in developer forums, we have come to believe that these are urgently needed operations (e.g. [23, 27, 33] to list just a few references). Add, rename, and delete correspond to the capabilities of an "ALTER TABLE" statement in relational databases. Just as with relational databases, more complex data migration tasks would then have to be encoded programmatically.

Yet in the majority of NoSQL databases, *any* data structure maintenance affecting more than one entity must be coded manually [23, 27, 34]. We still lack some of the basic tooling that one would expect in a NoSQL data store *ecosystem*, so that we may professionally maintain our production data in the long run.

Contributions. The goal of this work is to address this lack of tooling. We lay the foundation for building a generic schema evolution interface to NoSQL systems. Such a tool is intended for developers, administrators, and software architects to declaratively manage the structure of their production data. To this end, we make the following contributions:

- We investigate the established field of schema evolution in the new context of schema-less NoSQL data stores.
- We contribute a declarative *NoSQL schema evolution language*. Our language consists of a set of basic yet practical operations that address the majority of the typical cases that we see discussed in developer forums.
- We introduce a generic *NoSQL database programming language* that abstracts from the APIs of the most prominent NoSQL systems. Our language clearly distinguishes the state of the persisted data from the state of the objects in the application space. This is a vital aspect, since the NoSQL data store offers a very restricted API, and data manipulation happens in the application code.
- By implementing our schema evolution operations in our NoSQL database programming language, we show that they can be implemented for a large class of NoSQL data stores.
- We investigate whether a proposed schema evolution operation is *safe* to execute.
- Apart from exploring *eager* migration, we introduce the notion of *lazy* migration and point out its potential for future research in the database community.

Structure. In the next section, we start with an overview on the state-of-the-art in NoSQL data stores. Section 3 introduces our declarative language for evolving the data and its structure. In Section 4, we define an abstract and generic NoSQL database programming language for accessing NoSQL data stores. The operations of our language are available in many popular NoSQL systems. With this formal basis, we can implement our schema evolution operations eagerly, see Section 5. Alternatively, schema evolution can be handled lazily. We sketch the capabilities of object mappers that allow lazy migration in Section 6. In Section 7, we discuss related work on schema evolution in relational databases, XML applications, and NoSQL data stores. We then conclude with a summary and an outlook on our future work.

2. NoSQL Data Stores

We focus on NoSQL data stores hosted in a cloud environment. Typically, such systems scale to large amounts of data, and are schema-less or schema-flexible. We begin with a categorization of popular systems, discussing their commonalities and differences.

We then point out the NoSQL data stores that we consider in this paper with their core characteristics. In doing so, we generalize from proprietary details and introduce a common terminology.

2.1 State of the art

NoSQL data stores vary hugely in terms of data model, query model, scalability, architecture, and persistence design. Several taxonomies for NoSQL data stores have been proposed. Since we focus on schema evolution, a categorization of systems by data model is most natural for our purposes. We thus resort to a (very common) classification [8, 34] into (1) key-value stores, (2) document stores, and (3) extensible record stores. Often, extensible record stores are also called wide column stores or column family stores.

(1) Key-value stores. Systems like Redis [30, Chapter 8] or Riak [4] store data in pairs of a unique key and a value. Key-value stores do not manage the structure of these values. There is no concept of schema beyond distinguishing keys and values. Accordingly, the query model is very basic: Only inserts, updates, and deletes by key are supported, yet no query predicates on values. Since key-value stores do not manage the schema of values, schema evolution is the responsibility of the application.

(2) Document stores. Systems such as MongoDB [10] or Couchbase [7] also store key-value pairs. However, they store "documents" in the value part. The term "document" connotes loosely structured sets of name-value pairs, typically in JSON (JavaScript Object Notation) format or the binary representation BSON, a more type-rich format of JSON. Name-value pairs represent the properties of data objects. Names are unique, and name-value pairs are sometimes even referred to as key-value pairs. The document format is hierarchical, so values may be scalar, lists, or even nested documents. Documents within the same document store may differ in their structure, since there is no fixed schema.

Queries in document stores are more expressive than in key-value stores. Apart from inserting, updating, and deleting documents based on the document key, we may query documents based on their properties. The query languages differ from system to system. Some systems, such as MongoDB, have an integrated query language for ad-hoc queries, whereas other systems, such as CouchDB [30, Chapter 6] and Couchbase, do not. There, the user predefines views in form of MapReduce functions [12, 34].

An interesting and orthogonal point is the behavior in evaluating predicate queries: When a document does not contain a property mentioned in a query predicate, then this property is not even considered in query evaluation.

Document stores are schema-less, so documents may effortlessly evolve in structure: Properties can be added or removed from a particular document without affecting the remaining documents. Typically, there is no schema definition language that would allow the application developer to manage the structure of documents globally, across all documents.

(3) Extensible record stores. Extensible record stores such as BigTable [9] or HBase [13] actually provide a loosely defined schema. Data is stored as records. A schema defines families of properties, and new properties can be added within a property family on a per-record basis. (Properties and property families are often also referred to as *columns* and *column families*.) Typically, the schema cannot be defined up front and extensible record stores allow the ad-hoc creation of new properties. However, properties cannot be renamed or easily re-assigned from one property family to the other. So certain challenges from schema evolution in relational database systems carry over to extensible record stores.

Google Datastore [15] is built on top of Megastore [3] and BigTable, and is very flexible and comfortable to use. For instance, it very effectively implements multitenancy for all its users.

The Cassandra system [1] is an exception among extensible record stores, since it is much more restrictive regarding schema. Properties are actually defined up front, even with a "CREATE TABLE" statement, and the schema is altered globally with an "ALTER TABLE" statement. So while Cassandra is an extensible record store [8, 34], it is not schema-less or schema-flexible. In this work, we will exclusively consider schema-less data stores.

A word on NULL values. The handling of NULL values in NoSQL data stores deserves attention, as the treatment of unknown values is a factor in schema evolution. In relational database systems, NULL values represent unknown information, and are processed with a three-valued logic in query evaluation. Yet in NoSQL data stores, there is no common notion of NULLs across systems:

- Some systems follow the same semantics of NULL values as relational databases, e.g. [10].
- Some systems allow for NULL values to be stored, but do not allow NULLs in query predicates, e.g. [1, 15].
- Some systems do not allow NULL values at all, e.g. [13], arguing that NULL values only waste storage.

While there is no common strategy on handling unknown values yet, the discussion is ongoing and lively. Obviously, there is a semantic difference between a property value that is not known (such as the first name for a particular user), and a property value that does not exist for a variant of an entity (since home addresses and business addresses are structured differently). Consequently, some NoSQL data stores which formerly did not support NULL values have introduced them in later releases [11, 30, Chapter 6].

In Section 4, we present a generic NoSQL data store programming language. As the approaches to handling NULL values are so manifold, we choose to disregard NULLs as values and in queries, until a consensus has been established among NoSQL data stores.

2.2 NoSQL Data Stores in Scope for this Paper

In this paper, we investigate schema evolution for feature-rich, interactive web applications that are backed by NoSQL data stores. This makes document stores and schema-less extensible record stores our primary platforms of interest. Since key-value stores do not know any schema apart from distinguishing keys and values, we believe they are not the technology of choice for our purposes; after all, one cannot even run the most basic predicate queries, e.g. to find all blogs posted within the last ten hours.

We assume a high-level, abstract view on document stores and extensible record stores and introduce our terminology. Our terminology takes after Google Datastore [15]. We also state our assumptions on the data and query model.

Data model. Objects stored in the NoSQL data store are called *entities*. Each entity belongs to a *kind*, which is a name given to groups of semantically similar objects. Queries can then be specified over all entities of the same kind. Each entity has a unique *key*, which consists of the entity kind and an *id*. Entities have several *properties* (corresponding to attributes in the relational world). Each entity property consists of a *name* and a *value*. Properties may be scalar, they may be multi-valued, or consist of nested entities.

Query model. Entities can be inserted and deleted based on their key. We can formulate queries against all entities of a kind. At the very least, we assume that a NoSQL data store supports conjunctive queries with equality comparisons. This functionality is commonly provided by document stores and extensible record stores alike.

Freedom of schema. We assume that the global structure of entities cannot be fixed in advance. The structure of a single entity can be changed any time, according to the developers' needs.

```
evolutionop ::= add | delete | rename | move | copy;

add ::= "add" property "=" value [selection];
delete ::= "delete" property [selection];
rename ::= "rename" property "to" pname [selection];
move ::= "move" property "to" kname [complexcond];
copy ::= "copy" property "to" kname [complexcond];

selection ::= "where" conds;
complexcond ::= "where"  (joincond | conds
                          | (joincond "and" conds));
joincond ::= property "=" property;
conds ::= cond {"and" cond};
cond ::=  property "=" value;

property ::= kname "." pname;
kname ::= identifier;
pname ::= identifier;
```

Figure 2. EBNF of the NoSQL schema evolution language.

Example 1. The blogging application example from the Introduction is coherent with this terminology and these assumptions. □

3. A NoSQL Schema Evolution Language

In schema-less NoSQL data stores, there is no explicit, global schema. Yet when we are building feature-rich, interactive web applications on top of NoSQL data stores, entities actually do display an implicit structure (or schema); this structure manifests in the entity kind and entity property names. This especially holds when object mappers take over the mundane task of marshalling objects from the application space into persisted entities, and back. These object mappers commonly map class names to entity kinds, and class members to entity properties. (We discuss object mappers further in the context of related work in Section 7.)

Thus, there is a large class of applications that use NoSQL data stores, where the data is *somewhat* consistently structured, but has no fixed schema in the relational sense. Moreover, in an agile setting, these are applications that evolve rapidly, both in their features and their data. Under these assumptions, we now define a compact set of declarative schema migration operations, that have been inspired by schema evolution in relational databases, and update operations for semi-structured data [31]. While we can only argue empirically, having read through discussions in various developer forums, we are confident that these operations cover a large share of the common schema migration tasks.

Figure 2 shows the syntax of our *NoSQL schema evolution language* in Extended Backus-Naur Form (EBNF). An evolution operation adds, deletes, or renames properties. Properties can also be moved or copied. Operations may contain conditionals, even joins. The property kinds (derived from `kname`) and the property names (`pname`) are the terminals in this grammar. We will formally specify the semantics for our operations in Section 5. For now, we discuss some examples to develop an intuition for this language.

We introduce a special-purpose numeric property "version" for all entities. The version is incremented each time an entity is processed by an evolution operator. This allows us to manage heterogeneous entities of the same kind. This is an established development practice in entity evolution.

We begin with operations that affect all entities of one kind:

- The add operation adds a property to all entities of a given kind. A default value may be specified (see Example 2).
- The delete operation removes a property from all entities of a given kind (see Example 3).

- The rename operation changes the name of a property for all entities of a given kind (see Example 4).

Example 2. Below, we show an entity from our blogpost example before and after applying operation **add blogpost.likes = 0**. This adds a likes-counter to all blogposts, initialized to zero. We chose a compact tabular representation of entities and their properties.

key	**(blogpost, 331175)**
title	NoSQL Data..
content	NoSQL databases ..
version	*1*

key	**(blogpost, 331175)**
title	NoSQL Data..
content	NoSQL databases..
likes	0
version	*2*

□

Example 3. The operation **delete blogpost.url** deletes the property "url" from all blogposts.

key	**(blogpost, 331175)**
title	NoSQL Data..
content	NoSQL databases ..
url	www.mypage.com
version	*1*

key	**(blogpost, 331175)**
title	NoSQL Data..
content	NoSQL databases..
version	*2*

We can also specify a selection predicate. For instance, **delete blogpost.url where blogpost.version = 1** deletes the url-property only from those blogposts that are at schema version 1. □

Example 4. rename blogpost.text to content renames the property "text" to "content" for all blogpost entities.

key	**(blogpost, 331175)**
title	NoSQL Data..
text	NoSQL databases..
version	*1*

key	**(blogpost, 331175)**
title	NoSQL Data..
content	NoSQL databases..
version	*2*

□

We define further operations that affect two kinds of entities. Such migration operations are not available in schema definition languages for relational databases. Yet since NoSQL data stores typically do not support joins, denormalization is a technique heavily relied upon. When building interactive web applications, responsiveness is key, which usually forbids programmatic joins in the application. Instead, one would reorganize that data such that it renders joins unnecessary. Thus, duplication and denormalization are first-class citizens when building applications on top of NoSQL data stores. Accordingly, we introduce dedicated operations for supporting these schema refactorings.

- The move operation moves a property from one entity-kind to another entity-kind (see Example 5).
- The copy operation copies a property from one entity-kind to another entity-kind (see Example 6).

Of course, in moving and copying we also compute joins. Yet this is done in offline batch processing, and not during time-critical interactions with users.

Example 5. To move the property "url" from users to all their blogposts, we specify the operation **move user.url to blogpost where user.name = blogpost.author**. Figure 3 shows its application to a blog by user Gerhard. □

Example 6. The next example shows the copy operation: The property "email" is copied from users to all their blogposts: **copy user.email to blogpost where user.name = blogpost.author**. Figure 4 shows its application to a blog by user Gerhard. The copy operation does not change the user entities. □

Section 5 formalizes the semantics and investigates the effort of our migration operations. As a prerequisite, we next introduce a generic NoSQL database programming language.

key	**(user, 1234)**
name	Gerhard
email	gerhard@acm.org
status	professional
url	http://bigdata.org
version	*1*

key	**(user, 1234)**
name	Gerhard
email	gerhard@acm.org
status	professional
version	*2*

key	**(blogpost, 331175)**
title	NoSQL Data ..
content	NoSQL databases..
author	Gerhard
version	*1*

key	**(blogpost, 331175)**
title	NoSQL Data..
content	NoSQL databases..
author	Gerhard
url	http://bigdata.org
version	*2*

Figure 3. Moving property "url" (c.f. Example 5).

key	**(user, 1234)**
name	Gerhard
email	gerhard@acm.org
status	professional
version	*1*

key	**(user, 1234)**
name	Gerhard
email	gerhard@acm.org
status	professional
version	*1*

key	**(blogpost, 331175)**
title	NoSQL Data ..
content	NoSQL databases ..
author	Gerhard
version	*1*

key	**(blogpost, 331175)**
title	NoSQL Data ..
content	NoSQL databases ..
author	Gerhard
email	gerhard@acm.org
version	*2*

Figure 4. Copying property "email" (c.f. Example 6).

4. A NoSQL Database Programming Language

Relational databases come with a query language capable of joins, as well as dedicated data definition and data manipulation language. Yet in programming against NoSQL data stores, the application logic needs to take over some of these responsibilities. We now define the typical operations on entities in NoSQL data stores, building a purposeful NoSQL database programming language. Our language is particularly modeled after the interfaces to Google Datastore [15], and is applicable to document stores (e.g. [7]) as well as schema-less extensible record stores (e.g. [13]).

We consider system architectures such as shown in Figure 1. Each user interacts with an instance of the application, e.g. a servlet. Typically, the application fetches entities from the data store into the application space, modifies them, and writes them back to the data store. We introduce a common abstraction from the current state of the data store and the objects available in the application space. We refer to this abstraction as the *memory state*.

The memory state. We model a memory state as a set of mappings from entity keys to entity values. Let us assume that an entity has key κ and value ϑ. Then the memory contains the mapping from this key to this value: $\kappa \mapsto \vartheta$. Keys in a mapping are unique, so a memory state does not contain any mappings $\kappa \mapsto \vartheta_1$ and $\kappa \mapsto \vartheta_2$ with $\vartheta_1 \neq \vartheta_2$.

The entity value itself is structured as a mapping from property names to property values. A property value may be from an atomic domain *Dom*, either single-valued (*Dom*) or multi-valued (Dom^+), or it may consist of the properties of a nested entity.

Example 7. We model a memory state with a single entity managing user data. The key is a tuple of kind *user* and the id 42. The entity value contains the user's login "hhiker" and password "galaxy": $\{(\textit{"user"}, 42) \mapsto \{\textit{login} \mapsto \textit{"hhiker"},\ \textit{pwd} \mapsto \textit{"galaxy"}\}\}$. □

Substitutions. We describe manipulations of a memory state by substitutions. A substitution σ is a mapping from a set K (e.g. the entity keys) to a set V (e.g. the entity values) and the special symbol $\perp$. To access ϑ_i in a substitution $\{\kappa_1 \mapsto \vartheta_1, \ldots, \kappa_n \mapsto \vartheta_n\}$, we write $\sigma(\kappa_i)$. If $\sigma(\kappa_i) = \perp$, then this explicitly means that this mapping is not defined.

Let *ms* be the memory state, and let σ be a substitution. In *updating the memory state ms by substitution* σ, we follow a create-or-replace philosophy for each mapping in the substitution. We denote the updated memory by $ms[\sigma]$:

$$ms[\sigma] = \bigcup_{m \in ms} (m[\sigma]).$$

Let $a, b \in K$, and let $v, w \in V \cup \{\perp\}$. Then

$$\{a \mapsto v\}[\sigma] = \bigcup_{\{b \mapsto w\} \in \sigma} (\{a \mapsto v\}[\{b \mapsto w\}])$$

$$\{a \mapsto v\}[\{b \mapsto w\}] = \begin{cases} \{b \mapsto w\} & a = b \\ \{a \mapsto v, b \mapsto w\} & \text{otherwise} \end{cases}$$

We further use the shortform $ms[\kappa \mapsto \vartheta]$ to abbreviate the substitution with a single mapping $ms[\{\kappa \mapsto \vartheta\}]$.

Example 8. We continue with Example 7 and abbreviate the key $(\textit{“user”}, 42)$ by k. To delete the user's account, we update the memory state as

$$\{k \mapsto \{\textit{login} \mapsto \textit{“hhiker”}, \textit{pwd} \mapsto \textit{“galaxy”}\}\}[k \mapsto \perp] = \{k \mapsto \perp\}.$$

To mark the account as expired (state “x”), we write

$$\{k \mapsto (\{\textit{login} \mapsto \textit{“hhiker”},\ \textit{pwd} \mapsto \textit{“galaxy”}\}[\textit{state} \mapsto \textit{“x”}])\} = \{k \mapsto \{\textit{login} \mapsto \textit{“hhiker”},\ \textit{pwd} \mapsto \textit{“galaxy”}, \textit{state} \mapsto \textit{“x”}\}\}.$$

To change the user's password to “g2g”, we write

$$\{k \mapsto (\{\textit{login} \mapsto \textit{“hhiker”},\ \textit{pwd} \mapsto \textit{“galaxy”}\}[\textit{pwd} \mapsto \textit{“g2g”}])\} = \{k \mapsto \{\textit{login} \mapsto \textit{“hhiker”},\ \textit{pwd} \mapsto \textit{“g2g”}\}\}.$$

□

Evaluating operations. Operations may change the state of the data store and the application space. We call the former the *data store state*, and call the latter the *application state*. We denote the impact of operations by rules of the form

$$[\![op]\!](ds, as) = (ds', as')$$

where *op* denotes the operation to be executed on the data store state *ds* and the application state *as*. By evaluating the operation, the data store state changes to ds', and the application state to as'. Operations may be executed in sequence, which we define as

$$[\![op_1;\ op_2]\!](ds, as) = [\![op_2]\!]\big([\![op_1]\!](ds, as)\big).$$

4.1 Manipulating Entities

We next formalize operations common to most NoSQL data stores, namely creating and persisting entities, as well as retrieving and deleting single entities. Figure 5 defines our operations. Let *Kind* be the set of entity kinds. Let *Id* be a set denoting identifiers. The set of entity keys is defined as $\textit{Keys} = \textit{Kind} \times \textit{Id}$, i.e. an entity key is a tuple of the kind and an identifier. Entity properties are named. Let *Names* be the set of property names. A property value can be either an atomic value from domain *Dom*, multi-valued (i.e. from $\textit{Dom}^+$), or a nested entity.

Creating entities. We start with the rules to create entities and their properties. They affect the application state only. (For a change to have lasting effect, the entity must be persisted.) Rule 1 creates a new entity with key κ. Initially, an entity does not have any properties. To also set initial properties, we can use rule 2. Rule 3 adds a new property with name n and value v to the entity with key κ. Adding a nested entity as a property is specified in Rule 4. Rule 5 removes the property with name n from the entity with key κ: By setting the property value to $\perp$, the property by that name is no longer defined.

Persisting entities. Rule 6 persists the entity with key κ, replicating this entity to the data store state. The put-operation replaces any entity by the same key, should one exist. Rule 7 deletes the entity with key κ from the data store state. With rule 8, we retrieve a particular entity by key from the data store state.

Example 9. The following sequence of operations creates the entity from Example 7 and persists it in the data store.

① new$\big((\textit{“user”}, 42)\big)$;
② setProperty$\big((\textit{“user”}, 42), \textit{login}, \textit{“hhiker”}\big)$;
③ setProperty$\big((\textit{“user”}, 42), \textit{pwd}, \textit{“galaxy”}\big)$;
④ put$\big((\textit{“user”}, 42)\big)$

We evaluate the operations one by one on the initially empty data store and application state:

$$(\emptyset, \emptyset) \xrightarrow{①} (\emptyset, \{(\textit{“user”}, 42) \mapsto \emptyset\})$$
$$\xrightarrow{②} (\emptyset, \{(\textit{“user”}, 42) \mapsto \{\textit{login} \mapsto \textit{“hhiker”}\}\})$$
$$\xrightarrow{③} (\emptyset, \{(\textit{“user”}, 42) \mapsto \{\textit{login} \mapsto \textit{“hhiker”}, \textit{pwd} \mapsto \textit{“galaxy”}\}\})$$
$$\xrightarrow{④} (\{(\textit{“user”}, 42) \mapsto \{\textit{login} \mapsto \textit{“hhiker”}, \textit{pwd} \mapsto \textit{“galaxy”}\}\}, \{(\textit{“user”}, 42) \mapsto \{\textit{login} \mapsto \textit{“hhiker”}, \textit{pwd} \mapsto \textit{“galaxy”}\}\})$$

□

Accessing entity values. To access a particular value of an entity, we introduce a dedicated operation. We consider all variables as in Figure 5, with v being a property value. If such a value exists, v is either in $\textit{Dom}^+$ or a set of properties (from a nested entity):

$$[\![\text{getProperty}(\kappa, n)]\!](ds, as \cup \{\kappa \mapsto (\{n \mapsto v\} \cup \pi)\}) = v.$$

If property n is not defined for the entity with key κ, calling getProperty(κ, n) yields $\perp$.

Example 10. We illustrate nesting and unnesting of entities in close accordance with existing APIs (c.f. [15]). Let κ be the key of an entity with a nested entity as property n. To add a further property m with value w to the nested entity, we unnest property n into a temporary entity. Let *tmp* be a new entity key. After modification and re-nesting, we can persist the changes.

```
get(κ);
new(tmp, getProperty(κ, n));    // unnesting
setProperty(tmp, m, w);
setProperty(κ, n, tmp);         // nesting
put(κ)
```

□

4.2 Queries

Given an entity key κ, we define the function $\textit{kind}(\kappa)$ such that it returns the kind of this entity. Then rule 9 retrieves all entities from the store that are of the specified kind c.

In addition to querying for a particular kind, we can also query with a predicate θ, as described by rule 10. We consider conjunctive queries, with equality as the only comparison operator. This type of queries is typically supported by all of today's NoSQL data stores. Various systems may even have more expressive query languages (e.g. with additional comparison operators and support for disjunctive queries, yet typically no join).

More precisely, θ is a conjunctive query over atoms of the form $n = v$ where n is a property name and v is a property value from *Dom*. The predicate θ is evaluated on one entity at-a-time.

Let *ds* and *as* be a data store state and an application state. Let κ, κ' be entity keys. Let n, n' be property names, and let v be a property value. The symbol $\perp$ denotes the undefined value. Let π, π' be properties, i.e. a set of mappings from property names to property values. *kind* : *Keys* $\mapsto$ *Kind* is a function that extracts the entity kind from a key. θ is a conjunctive query, and c is a string constant.

$$[\![\text{new}(\kappa)]\!](ds, as) = (ds, as[\kappa \mapsto \emptyset]) \quad (1)$$

$$[\![\text{new}(\kappa, \pi)]\!](ds, as) = (ds, as[\kappa \mapsto \pi]) \quad (2)$$

$$[\![\text{setProperty}(\kappa, n, v)]\!](ds, as \cup \{\kappa \mapsto \pi\}) = (ds, as \cup \{\kappa \mapsto (\pi[n \mapsto v])\}) \quad (3)$$

$$[\![\text{setProperty}(\kappa, n, \kappa')]\!](ds, as \cup \{\kappa \mapsto \pi\} \cup \{\kappa' \mapsto \pi'\}) = (ds, as \cup \{\kappa \mapsto (\pi[n \mapsto \pi'])\} \cup \{\kappa' \mapsto \pi'\}) \quad (4)$$

$$[\![\text{removeProperty}(\kappa, n)]\!](ds, as \cup \{\kappa \mapsto \pi\}) = (ds, as \cup \{\kappa \mapsto (\pi[n \mapsto \perp])\}) \quad (5)$$

$$[\![\text{put}(\kappa)]\!](ds, as \cup \{\kappa \mapsto \pi\}) = (ds[\kappa \mapsto \pi], as \cup \{\kappa \mapsto \pi\}) \quad (6)$$

$$[\![\text{delete}(\kappa)]\!](ds, as) = (ds[\kappa \mapsto \perp], as) \quad (7)$$

$$[\![\text{get}(\kappa)]\!](ds \cup \{\kappa \mapsto \pi\}, as) = (ds \cup \{\kappa \mapsto \pi\}, as[\kappa \mapsto \pi]) \quad (8)$$

$$[\![\text{get}(\mathit{kind} = c)]\!](ds, as) = (ds, as[\{\kappa \mapsto \pi \mid \kappa \mapsto \pi \in ds \wedge \mathit{kind}(\kappa) = c\}]) \quad (9)$$

$$[\![\text{get}(\mathit{kind} = c \wedge \theta)]\!](ds, as) = (ds, as[\{\kappa \mapsto \pi \mid \kappa \mapsto \pi \in ds \wedge \mathit{kind}(\kappa) = c \wedge [\![\theta]\!](\kappa \mapsto \pi)\}]) \quad (10)$$

Figure 5. Operations for creating, manipulating, and persisting entities, as well as queries over entities.

We evaluate an atom $\kappa.n = v$ on a single entity:

$$[\![n = v]\!](\kappa \mapsto \pi) = \begin{cases} \text{true} & (n \mapsto v) \in \pi \\ \text{true} & (n \mapsto \vartheta) \in \pi, \vartheta \in \mathit{Dom}^+, v \in \vartheta \\ \text{false} & \text{otherwise} \end{cases}$$

An atom involving $\perp$ as an operand is always evaluated to false. Queries over nested entities are not supported, e.g. as in [15]. The evaluation of conjunctions follows naturally:

$$[\![\theta_1 \wedge \theta_2]\!](\kappa \mapsto \pi) = [\![\theta_1]\!](\kappa \mapsto \pi) \wedge [\![\theta_2]\!](\kappa \mapsto \pi)$$

4.3 Iteration Statements

For batch updates on entities, we define a for-loop. Let x be a variable denoting an entity key, and let *op* denote an operation from our NoSQL database programming language. θ is a conjunctive query with atomic equality conditions. The operands in atoms are of the form x, $x.n$, or v, where x is a variable denoting a key, n is a property name, and v is a value from *Dom*.

We consider the execution of for-loops on a data store state *ds* and an application state *as*:

$$[\![\textbf{foreach } x \textbf{ in } \text{get}(\theta) \textbf{ do } \mathit{op} \textbf{ od }]\!](ds, as)$$

Let as_θ be the result of evaluating query θ, i.e.

$$[\![\text{get}(\theta)]\!](ds, \emptyset) = (ds, as_\theta)$$

and let $K = \{\kappa \mid (\kappa \mapsto \pi) \in as_\theta\}$ be the keys of all entities in the query result. We can then evaluate the for-loop as follows.

while $(K \neq \emptyset)$ **do**
  there exists some key κ in K;
  $K := K \setminus \{\kappa\}$;
  evaluate operation *op* for the binding of x to key κ:
  $(ds, as) := [\![\ \mathit{op}[x/\kappa]\ ]\!](ds, as)$;
od

Above, $\mathit{op}[x/\kappa]$ is obtained from operation *op* by first substituting each occurrence of x in *op* by κ, and next replacing all operands $\kappa.n$ in query predicates by the value of "getProperty(κ, n)".

Example 11. We add a new property "email" to all user entities in the data store, and initialize it with the empty string ϵ.

foreach x **in** get(*kind* = *"user"*) **do**
  setProperty(x, *email*, ϵ);
  put(x)
od

Since denormalization is vital for performance in NoSQL data stores, we show how to copy the property "url" from each user entity to all blogposts written by that user.

foreach u **in** get(*kind* = *"user"*) **do**
  foreach b **in** get(*kind* = *"blogpost"* $\wedge$ *author* = u.*login*) **do**
    setProperty(b, *url*, getProperty(u, *url*));
    put(b)
  od
od □

5. Safe and Eager Migration

Now that we have a generic NoSQL database programming language, we can implement the declarative schema evolution operations from Section 3. We believe the declarative operations cover common schema evolution tasks. For more complex migration scenarios, we can always resort to a programmatic solution. This matches the situation with relational databases, where an "ALTER TABLE" statement covers the typical schema alterations, but where more complex transformations require an ETL-process to be set up, or a custom migration script to be coded.

Figure 6 shows the implementation for the operations add, delete, and rename. A for-loop fetches all matching entities from the data store, modifies them, and updates their version property (as introduced in Section 3). The updated entities are then persisted.

Figure 7 shows the implementation for copy and move. Again, entities are fetched from the NoSQL data store one by one, updated, and then persisted. This requires joins between entities. Since joins are not supported in most NoSQL data stores, they need to be encoded in the application logic.

This batch update corresponds to the recommendation of NoSQL data store providers on how to handle schema evolution (e.g. [23]).

Note that the create-or-replace semantics inherent in our NoSQL database programming language make for a *well-defined* behavior of operations. For instance, renaming the property "text" in blogposts to "content" (c.f. Example 4) effectively overwrites any existing property named content.

Moreover, the version property added to all entities makes the migration *robust* in case of interruptions. NoSQL data stores commonly offer very limited transaction support. For instance, Google Datastore only allows transactions to span up to five entities in so-called *cross-group transactions* (or alternatively, provides the concept of entity groups not supported in our NoSQL database pro-

Legend: Let c be a kind, let n be a property name, and let v be a property value from *Dom*. θ is a conjunctive query over properties.

add $c.n = v$ **where** θ
  foreach e **in** get(*kind* $= c \wedge \theta$) **do**
    setProperty(e, n, v);
    setProperty(e, *version*, getProperty(e, *version*) $+1$);
    put(e)
  od

delete $c.n$ **where** θ

  foreach e **in** get(*kind* $= c \wedge \theta$) **do**
    removeProperty(e, n);
    setProperty(e, *version*, getProperty(e, *version*) $+1$);
    put(e)
  od

rename $c.n$ **to** m **where** θ

  foreach e **in** get(*kind* $= c \wedge \theta$) **do**
    setProperty(e, m, getProperty(e, n));
    removeProperty(e, n);
    setProperty(e, *version*, getProperty(e, *version*) $+1$);
    put(e)
  od

Figure 6. Implementing add, delete, and rename.

gramming language) [15]. So a large-scale migration cannot be performed as an atomic action. By restricting migrations to all entities of a particular version (using the where-clause), we may correctly recover from interrupts, even for move and copy operations.

Interestingly, not all migrations that can be specified are desirable. For instance, assuming a 1:N relationship between users and the blogposts they have written, the result of the migration

copy user.url **to** blogpost **where** user.login = blogpost.author

does not depend on the order in which blogpost entities are updated. However, if there is an N:M relationship between users and blogposts, e.g. since we specify the copy operation as cross product between all users and all blogposts,

copy user.url **to** blogpost

then the execution order influences the migration result. Naturally, we want to be able to know whether a migration is safe before we execute it. Concretely, we say a migration is *safe* if it does not produce more than one entity with the same key.

The following propositions follow from the implementations of schema evolution operators in Figures 6 and 7.

Proposition 1. *An add, delete, or rename operation is safe.*

Proposition 2. *For a move or copy operation, and a data store state ds, the safety of executing the operation on ds can be decided in* $O(|ds|^2)$.

Deciding whether a copy or move operation is safe can be done in a simulation run of the evolution operator. If an entity has already been updated in such a "dry-run" and is to be overwritten with different property values, then the migration is not safe.

In relational data exchange, the existence of solutions for relational mappings under constraints is a highly related problem. There, it can be shown that while the existence of solutions is an undecidable problem per-se, for certain restrictions, the problem is PTIME-decidable (c.f. Corollary 2.15 in [2]). Moreover, the vehicle for checking for solutions is the chase algorithm, which fails when equality-generating dependencies in the target schema are violated. This is essentially the same idea as our dry-run producing

Legend: Let c_1, c_2 be kinds and let n be a property name. Conditions θ_1 and θ_2 are conjunctive queries. θ_1 has atoms of the form $c_1.m = v$, where m is a property name and v is a value from *Dom*. θ_2 has atoms of the form $c_2.m = v$ or $c_1.a = c_2.b$, where a, b, and m are property names. v is a value from *Dom*.

move $c_1.n$ **to** c_2 **where** $\theta_1 \wedge \theta_2$
  foreach e **in** get(*kind* $= c_1 \wedge \theta_1$) **do**
    foreach f **in** get(*kind* $= c_2 \wedge \theta_2$) **do**
      setProperty(f, n, getProperty(e, n));
      setProperty(f, *version*, getProperty(f, *version*) $+1$);
      put(f)
    od;
    setProperty(e, *version*, getProperty(e, *version*) $+1$);
    removeProperty(e, n);
    put(e)
  od

copy $c_1.n$ **to** c_2 **where** $\theta_1 \wedge \theta_1$

  foreach e **in** get(*kind* $= c_1 \wedge \theta_1$) **do**
    foreach f **in** get(*kind* $= c_2 \wedge \theta_2$) **do**
      setProperty(f, n, getProperty(e, n));
      setProperty(f, *version*, getProperty(f, *version*) $+1$);
      put(f)
    od
  od

Figure 7. Implementing copy and move.

entities with the same key, but conflicting values. Since our schema evolution operations copy and move require two nested for-loops, we can check for safety in quadratic time. (Keeping track of which entities have already been updated can be done efficiently, e.g. by maintaining a bit vector in the size of *ds*.)

6. An Outlook on Lazy Migration

Our NoSQL database programming language can also express operations for lazy migration. To illustrate this on an intuitive level, we encode some features of the Objectify object mapper [27].

We will make use of some self-explanatory additional language constructs, such as if-statements and local variables. Additionally, we assume an operation "hasProperty(κ, n)" that tests whether the entity with key κ in the application state has a property by name n.

Example 12. The following example is adapted from the Objectify documentation. It illustrates how properties are renamed when an entity is loaded from the data store and translated into a Java object.

The Java class Person is mapped to an entity. The annotation `@Id` marks the identifier for this entity, the entity kind is derived from the class name. The earlier version of this entity has a property "name", which is now renamed to "fullName". Legacy entities do not yet have the property "fullName". When they are loaded, the object mapper assigns the value of property "name" to the class attribute "fullName". The next time that the entity is persisted, its new version will be stored.

```
public class Person {
  @Id Long id;
  @AlsoLoad("name") String fullName;
}
```

In our NoSQL database programming language, we implement the annotation `@AlsoLoad` as follows.

Key p := (*"Person"*, *id*);
if hasProperty(p, *name*) **do**

setProperty(p, *fullName*, getProperty(p, *name*));
removeProperty(p, *name*)
od □

Example 13. The following example is adapted from [27]. The annotation `@OnLoad` specifies the migration for an entity when it is loaded. If the entity has properties street and city, these properties are moved to a new entity storing the address. These properties are then discarded from the person entity when it is persisted (specified by the annotation `@IgnoreSave`). Saving an entity is done by calling the Objectify function `ofy().save()`.

```
public class Person {
  @Id Long id;
  @IgnoreSave String street;
  @IgnoreSave String city;

  @OnLoad void onLoad() {
    if (this.street != null && this.city != null) {
      Entity a = new Entity("address");
      a.setProperty("person", this.id);
      a.setProperty("street", this.street);
      a.setProperty("city", this.city);
      ofy().save().entity(a);
     }
  }
}
```

We implement the method with annotation `@OnLoad` as follows.

Key p := (*"Person"*, *id*);
if (hasProperty(p, *street*) $\wedge$ hasProperty(p, *city*)) **do**
  Key a = (*"Address"*, *id*);
  new(a);
  setProperty(a, *person*, *id*);
  setProperty(a, *street*, getProperty(p, *street*));
  setProperty(a, *city*, getProperty(p, *city*));
  put(a);
  removeProperty(p, *street*);
  removeProperty(p, *city*);
od □

It remains future work to explore lazy migrations in greater detail, and develop mechanisms to statically check them prior to execution: The perils of using such powerful features in an uncontrolled manner, on production data, are evident. Lazy migration is particularly difficult to test prior to launch, since we cannot foretell which entities will be touched at runtime. After all, users may return after years and re-activate their accounts, upon which the object mapper tries to evolve ancient data.

It is easy to imagine scenarios where lazy migration fails, due to artifacts in the entity structure that developers are no longer aware of. In particular, we would like to be able to determine whether an annotation for lazy migration is safe. At the very least, we would like to check whether a lazy migration is *idempotent*, so that when transactions involving evolutions fail, there is no harm done in reapplying the migration.

7. Related Work

We define a NoSQL database programming language as an abstract interface for programming against NoSQL data stores. In recent work, [5] present a calculus for NoSQL systems together with its formal semantics. They introduce a Turing-complete language and its type system, while we present a much more restricted language with a focus on updates and schema evolution.

For relational databases, the importance of designing database programming languages for strong programmability, concerning both performance and usability, has been emphasized in [19]. The language presented there can express database operators, query plans, and also capture operations in the application logic. However, the work there is targeted at query execution in relational databases, while we cover aspects of data definition and data manipulation in NoSQL data stores. Moreover, we treat the data store itself as a black box, assuming that developers use a cloud-based database-as-a-service offering that they cannot manipulate.

All successful applications age with time [29], and eventually require maintenance or evolution. Typically, there are two alternatives to handling this problem on the level of schema: Schema versioning and schema evolution. Relational databases have an established language for schema evolution ("ALTER TABLE"). This schema definition language is part of the SQL standard, and is implemented by all available relational databases systems.

For evolving XML-based applications, research prototypes have been built that concentrate on the co-evolution of XML schemas and the associated XML documents [18]. The authors of [25] have developed a model driven approach for XML schema design, and support co-evolution between different abstraction levels. A dedicated language for XML evolution is introduced in [26] that formalizes XML schema change operations and describes the corresponding updates of associated XML documents.

JSONiq is a quite new query language for JSON documents, the first version was published in April 2013 [32]. Future versions of JSONiq will contain an update facility and will offer operations to add, delete, insert, rename, and replace properties and values. Our schema evolution language can be translated into corresponding update expressions. If JSONiq establishes itself as a standard for querying and updating NoSQL datastores, we can also base our schema evolution method on this language.

The question whether an evolution is safe corresponds to the existence of (universal) solutions in data exchange. In particular, established practices from XML data exchange, using regular tree grammars to specify the source and the target schema [2], are highly relevant to our work. The use of object mappers translating objects from the application space into persisted entities can be seen as a form of schema specification. This raises an interesting question: Provided that all entities conform to the class hierarchy specified by an object mapper, if we evolve entities, will they still work with our object mapper? This boils down to checking for absolute consistency in XML data exchange [2], and is a current topic in database theory (e.g. [6]). It is therefore part of our plans to see how we can leverage the latest research on XML data exchange for evolving data in schema-less data stores.

There are various object-relational mapping (ORM) frameworks fulfilling well established standards such as the Java Persistence API (JPA), and supporting almost all relational database systems. Some ORM mappers are even supported by NoSQL data stores, of course not implementing all features, since joins or foreign-keys are not supported by the backend (e.g. see the JPA and JDO implementations for Google Datastore [16, 17]).

So far, there are only few dedicated mappers for persisting objects in NoSQL data stores (sometimes called object-data-store mappers (ODM)). Most of today's ODMs are proprietary, supporting a particular NoSQL data store (e.g. Morphia [24] for MongoDB, or Objectify[28] for Google Datastore). Few systems support more than one NoSQL data store (e.g. Hibernate OGM [20]).

Today, these objects-to-NoSQL mapping tools have at best rudimentary support for schema evolution. To the best of our knowledge, Objectify and Morphia go the furthest by allowing developers to specify lazy migration in form of object annotations. However, we could not yet find any solutions for systematically managing and expressing schema changes. At this point, the ecosystem of tools for maintaining NoSQL databases is still within its infancy.

8. Summary and Future Work

This work investigates the maintainability of feature-rich, interactive web applications, from the view-point of schema evolution. In particular, we target applications that are backed by schema-less *document stores* or *extensible record stores*. This is an increasingly popular software stack, now that database-as-a-service offerings are readily available: The programming APIs are easy to use, there is near to no setup time required, and pricing is reasonable. Another sweet spot of these systems is that the data's schema does not have to be specified in advance. Developers may freely adapt the data's structure as the application evolves. Despite utter freedom, the data nevertheless displays an *implicit* structure: The application class hierarchy is typically reflected in the persisted data, since object mappers perform the mundane task of marshalling data between the application and the data store.

As an application evolves, so does its schema. Yet schema-free NoSQL data stores do not yet come with convenient schema management tools. As of today, virtually all data migration tasks require custom programming (with the exception of very basic data inspection tools for manipulating *single* entities). It is up to the developers to code the migration of their production data "on foot", getting the data ready for the next software release. Worse yet, with weekly releases, the schema evolves just as frequently.

In this paper, we lay the foundation for systematically managing schema evolution in this setting. We define a declarative *NoSQL schema evolution language*, to be used in a NoSQL data store administration console. Using our evolution language, developers can specify common operations, such as adding, deleting, or renaming properties in batch. Moreover, properties can be moved or copied, since data duplication and denormalization are fundamental in NoSQL data stores. We emphasize that we do not mean to enforce a relational schema onto NoSQL data stores. Rather, we want to ease the pain of schema evolution for application developers.

We regard it as one of our key contributions that our operations can be implemented for a large class of NoSQL data stores. We show this by an implementation in a generic *NoSQL database programming language*. We also discuss which operations can be applied safely, since non-deterministic migrations are unacceptable.

Future work. Our NoSQL schema evolution language specifies operations that are executed *eagerly*, on all qualifying entities. An alternative approach is to migrate entities *lazily*, the next time they are fetched into the application space. Some object mappers already provide such functionality. We believe that lazy evolution is still little understood, and at the same time poses great risks when applied erroneously. We will investigate how our NoSQL schema evolution language may be implemented both safely *and* lazily. Ideally, a dedicated schema evolution management tool would allow developers to migrate data eagerly for leaps in schema evolution, and to patch things up lazily for minor changes.

References

[1] Apache Cassandra, 2013. `http://cassandra.apache.org/`.

[2] M. Arenas, P. Barceló, L. Libkin, and F. Murlak. *Relational and XML Data Exchange*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2010.

[3] J. Baker, C. Bond, J. C. Corbett, J. Furman, et al. "Megastore: Providing Scalable, Highly Available Storage for Interactive Services". In *Proc. CIDR*, pages 223–234, 2011.

[4] Basho Technologies. riak/docs, 2013. `http://docs.basho.com/riak/latest/`.

[5] V. Benzaken, G. Castagna, K. Nguyen, and J. Siméon. "Static and dynamic semantics of NoSQL languages". In *Proc. POPL*, pages 101–114, 2013.

[6] M. Bojańczyk, L. A. Kolodziejczyk, and F. Murlak. "Solutions in XML data exchange". In *Proc. ICDT*, pages 102–113, 2011.

[7] M. Brown. *Developing with Couchbase Server*. O'Reilly, 2013.

[8] R. Cattell. "Scalable SQL and NoSQL data stores". *SIGMOD Record*, 39(4):12–27, 2010.

[9] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, et al. "Bigtable: A Distributed Storage System for Structured Data". In *Proc. OSDI*, pages 205–218, 2006.

[10] K. Chodorow. *MongoDB: The Definitive Guide*. O'Reilly, 2013.

[11] Couch Potato, 2010. `https://github.com/langalex/couch_potato/issues/14`.

[12] J. Dean and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In *Proc. OSDI*, pages 137–150, 2004.

[13] L. George. *HBase: The Definitive Guide*. O'Reilly, 2011.

[14] Google Inc. Google Datastore Admin, 2013. `https://developers.google.com/appengine/docs/adminconsole/datastoreconsole`.

[15] Google Inc. Google Datastore, 2013. `https://developers.google.com/appengine/docs/java/datastore/`.

[16] Google Inc. Using JDO with App Engine, 2013. `https://developers.google.com/appengine/docs/java/datastore/jdo/`.

[17] Google Inc. Using JPA with App Engine, 2013. `https://developers.google.com/appengine/docs/java/datastore/jpa/overview`.

[18] G. Guerrini, M. Mesiti, and M. A. Sorrenti. "XML Schema Evolution: Incremental Validation and Efficient Document Adaptation". In *Proc. XSym*, pages 92–106, 2007.

[19] D. Habich, M. Boehm, M. Thiele, B. Schlegel, U. Fischer, H. Voigt, and W. Lehner. "Next Generation Database Programming and Execution Environment". In *Proc. DBPL*, 2011.

[20] JBoss Community. Hibernate OGM (Object/Grid Mapper), 2013. `http://www.hibernate.org/subprojects/ogm.html`.

[21] JSON.org. Introducing JSON, 2013. `http://www.json.org/`.

[22] S. Lightstone. *Making it Big in Software*. Prentice Hall, 2010.

[23] J. McWilliams and M. Ivey. Google Developers: Updating your model's schema, 2012. `https://developers.google.com/appengine/articles/update_schema`.

[24] Morphia. A type-safe java library for MongoDB, 2013. `http://code.google.com/p/morphia/`.

[25] M. Necaský, J. Klímek, J. Malý, and I. Mlýnková. "Evolution and change management of XML-based systems". *Journal of Systems and Software*, 85(3):683–707, 2012.

[26] T. Nösinger, M. Klettke, and A. Heuer. "XML Schema Transformations - The ELaX Approach". In *Proc. DEXA*, 2013.

[27] Objectify AppEngine. Migrating Schemas, 2012. `https://code.google.com/p/objectify-appengine/wiki/SchemaMigration`.

[28] Objectify AppEngine. The simplest convenient interface to the Google App Engine datastore, 2013. `https://code.google.com/p/objectify-appengine/`.

[29] D. L. Parnas. "Software Aging". In *Proc. ICSE*, pages 279–287, 1994.

[30] E. Redmond and J. R. Wilson. *Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement*. Pragmatic Bookshelf, 2012.

[31] J. Robie, D. Chamberlin, M. Dyck, D. Florescu, J. Melton, and J. Simeon. XQuery Update Facility 1.0, 2011. `http://www.w3.org/TR/xquery-update-10/`.

[32] J. Robie, G. Fourny, M. Brantner, D. Florescu, T. Westmann, and M. Zaharioudakis. JSONiq Grammar - Extensions, 2013. `http://www.jsoniq.org/grammars/extension.xhtml`.

[33] Stack Overflow, 2012. `http://stackoverflow.com/questions/8920610/add-a-new-attribute-to-entity-in-datastore`.

[34] S. Tiwari. *Professional NoSQL*. John Wiley & Sons, 2011.