Safely Managing Data Variety in Big Data Software Development

Thomas Cerqueus, Eduardo Cunha de Almeida, Stefanie Scherzinger

Safely Managing Data Variety in Big Data Software Development

Thomas Cerqueus[†], Eduardo Cunha de Almeida[‡] and Stefanie Scherzinger[*]
[†] Université de Lyon, CNRS, INSA-Lyon, LIRIS, UMR5205, thomas.cerqueus@insa-lyon.fr
[‡] Federal University of Paraná, eduardo@inf.ufpr.br
[*] OTH Regensburg, stefanie.scherzinger@oth-regensburg.de

***Abstract*—We consider the task of building Big Data software systems, offered as software-as-a-service. These applications are commonly backed by NoSQL data stores that address the proverbial *V*s of Big Data processing: NoSQL data stores can handle large *volumes* of data and many systems do not enforce a global schema, to account for structural *variety* in data. Thus, software engineers can design the data model on the go, a flexibility that is particularly crucial in agile software development. However, NoSQL data stores commonly do not yet account for the *veracity* of changes when it comes to changes in the structure of persisted data. Yet this is an inevitable consequence of agile software development. In most NoSQL-based application stacks, schema evolution is completely handled within the application code, usually involving object mapper libraries. Yet simple code refactorings, such as renaming a class attribute at the source code level, can cause data loss or runtime errors once the application has been deployed to production. We address this pain point by contributing type checking rules that we have implemented within an IDE plugin. Our plugin ControVol statically type checks the object mapper class declarations against the code release history. ControVol is thus capable of detecting common yet risky cases of mismatched data and schema, and can even suggest automatic fixes.**

I. Introduction

Big Data software requires database systems that can scale to large volumes of data, such as the NoSQL data stores Google Cloud Datastore [1] or MongoDB [2]. Apart from being highly scalable, these systems are schema-flexible. As such, they do not require the specification of a fixed, global data schema up front, to account for the structural *variety* of Big Data. This is particularly convenient in agile development, since the team can design the data model as it goes along.

Yet keeping the schema in sync with the application code is a well-known data management challenge that arises with relational data stores, object-oriented data stores, XML, and NoSQL data stores alike (e.g., [3]–[10]). The problem presents itself with a new acuteness and with different challenges in the context of NoSQL application development:

First, there is a new pace to schema changes, with the growing popularity of agile software development and therefore shorter release cycles. Popular Big Data products such as Youtube are released weekly, if not daily (quoting Marissa Meyer in [11]). With frequent schema changes, agile developers do not want to have to wait for the next scheduled database maintenance window in order to release. Also, with software being available 24/7, there simply may not be a time window for migrating legacy data.

Second, the problem arises at a new scale in terms of the number of people affected, as schema-flexible data stores are becoming more and more popular in Big Data software development. Unfortunately, solutions for controlling schema evolution in relational data stores do not immediately carry over to schema-flexible data stores, since relational data stores assume that all data adheres to a global and fixed schema. At the same time, the developer community is currently promoting its own tools and strategies, e.g., dedicated object mappers capable of lazy migration [12], [13], and frameworks such as Kiji [14] (c.f. our related works section). This may be seen as a strong signal that developers are experiencing a pain that they do not yet see addressed by existing tools.

Third, the problem is hitting a new group of people. Traditionally, the team's designated database administrator would be responsible for migrating legacy data, in preparation for a new release. Since schema-flexible NoSQL data stores do not provide much functionality in terms of schema management, the problem is now addressed by the software developers themselves. This target group is used to working with powerful IDEs, yet it has little experience with expert-level data management tools such as ETF toolchains. To effectively assist them in their daily work, developers require tools that fit seamlessly into their development environment.

Let us sketch a typical setup of a Big Data web development project. Figure 1 shows the development environment on the left. The software engineers have just finalized version v1 of the application using an Integrated Development Environment (IDE). The code repository manages the source code versions, and currently stores the upcoming release v1, as well as the earlier release v0. On the right, version v0 of the application has been serving in production, hosted on a platform-as-a-service infrastructure (PaaS). To the bottom is a NoSQL data store, offered as database-as-a-service (DaaS).

Object mapper libraries translate between objects in the application space and persisted entities. To clearly distinguish between the data structures handled within the application code and the data structures persisted in the data store, we call the former *objects* and the latter *entities*. Let us assume that we are building an online role playing game, where players are the central objects. So far, version v0 has persisted player entities according to the Java class declaration in Figure 2(a). Player Frodo's entity is shown in JSON format:

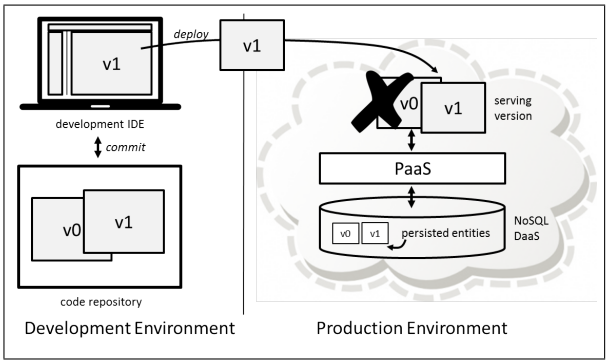
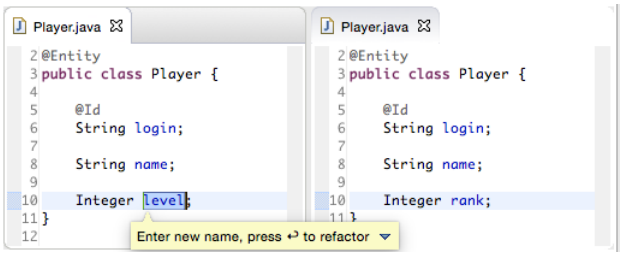


Fig. 1: Releasing new version v1 of the application from the development environment into production. While version v0 no longer serves users, its *legacy* entities remain in the NoSQL data store.



(a) Before renaming. (b) After renaming.

Fig. 2: Refactoring the source code: Attribute *level* is renamed to *rank*. The Eclipse IDE consistently changes all references in the source code, but this does not affect persisted legacy entities in production. So when legacy players are loaded into the application space, attribute *level* cannot be matched and its value is not loaded.

```
{ "kind":  "Player",
  "login": "ringbearer",
  "name":  "Frodo Baggins",
  "level": 1 }
```

Version v1 is now ready for release and the engineers deploy it to production. This step is visualized in Figure 1. We assume that the new application code persists players according to the new Java class declaration shown in Figure 2(b).

The player entities persisted by version v0 have now become *legacy* entities. The new application code must robustly handle these legacy entities. For instance, if the new class declaration loads the legacy entity for player Frodo, it expects an attribute *rank*, but does not expect the attribute *level*. In this scenario, the Objectify object mapper [12] will not raise a runtime error. Instead, the legacy entity is loaded, but the unmatched *rank* attribute is set to *null*. The value of *level* in the persisted entity cannot be loaded. Worse yet, the next time that the player object is persisted, the value of *level* is effectively dropped:

```
{ "kind":  "Player",
  "login": "ringbearer",
  "name":  "Frodo Baggins",
  "rank":  null }
```

Today, developers largely rely on their foresight and exhaustive testing to catch such errors early on. What is missing is a framework that type checks object mapper class declarations against their schema evolution history and gives feedback already during the development process. Type checking in database programming has a long history [15]. In this tradition, we present a set of custom-tailored type checking rules.

Contributions: The main contributions of this paper are:

- We introduce a set of type checking rules that allow for static type checking of object mapper class declarations against the release history, as recorded in the code repository. Our rules detect possible cases of runtime errors or data loss already at development time.
- We have implemented our type checking rules within the ControVol Eclipse plugin. This plugin has been demoed in the past [16], [17], yet the theory on ControVol, in particular the type checking rules, are presented in this paper for the first time. In contrast, the demo papers focus on the workflow from the viewpoint of the developer interacting with ControVol.
- The average developer will be agnostic of our type checking rules. However, power users may customize or extend the rules. Accordingly, we propose pragmatic relaxations and extensions to our type checking rules, e.g., to allow for safe type promotions in loading entities.
- By checking the complete chain of class declarations over time, we can even catch more subtle schema evolution problems that appear when class member attributes that have been removed from the source code in some past release are being reintroduced.

Organization: Section II reviews NoSQL object mappers with life-cycle annotations. We point out common pitfalls, before we introduce our type checking rules in Section III. Section IV discusses relaxations and extensions. Section V gives a brief overview of the implementation as the ControVol Eclipse plugin. We conclude with related work and a summary.

II. Preliminaries

We briefly introduce NoSQL-specific object mapper libraries and their capabilities, and then point out common schema evolution pitfalls.

A. Lazy Schema Evolution with NoSQL Object Mappers

In building applications against relational data stores, object-relational mappers have become state-of-the-art for accessing persisted data. Let us return to our example of an online role playing game. Figure 2(a) shows the Java class Player with typical object mapper annotations. The annotation `@Entity` denotes that instances of this class can be persisted. The annotation `@Id` marks the unique identifier. Based on these annotations, the object mapper takes care of marshalling between objects in the application space and tuples in a relational database. Object mappers allow developers to build more robust and portable applications, since they introduce a desirable level of abstraction.

The growing popularity of NoSQL data stores has triggered a new generation of object mappers [18]. Interestingly, some of these mappers address the schema evolution problem for

Fig. 3: Lazily migrating legacy entities: A developer has renamed attribute *level* to *rank*. Annotation `@AlsoLoad` ensures that a legacy attribute *level* can still be loaded and will be renamed to *rank*.

NoSQL data stores by providing so-called *life-cycle annotations*, e.g., Objectify [12] (for Google Cloud Datastore) and Morphia [13] (for MongoDB).

Life-cycle annotations *lazily* migrate legacy entities when they are loaded. Figure 3 shows an Objectify class declaration with annotation `@AlsoLoad`. This mapper will successfully load player entities with an attribute *rank*, as well as legacy players with an attribute *level*. In the latter case, the *level* attribute will be renamed to *rank* when loading a persisted entity as a Java object. The change is manifested the next time that the object is persisted.

For illustration of the life-cycle capabilities, we briefly list the Objectify life-cycle annotations to class member attributes.

- `@AlsoLoad` renames an attribute lazily at loading time.
- Annotation `@Ignore` denotes that an attribute will neither be loaded nor saved. This is typically used for class member attributes with derived values. Then the attribute exists in the Java object, but not in the persisted entity.
- `@IgnoreSave` denotes that an attribute will be loaded from the data store, but not saved. This allows developers to load legacy attributes, usually as input to more complex migration code, and to remove them from the persisted entity the next time that the object is persisted.
- `@IgnoreLoad` denotes that an attribute will not be loaded from the data store, but it will be saved to the datastore. This allows developers to overwrite an existing attribute in a persisted entity.

Developers may further annotate class methods, in particular,

- `@OnLoad` calls a method immediately after all attributes have been loaded, and
- `@OnSave` calls a method just before persisting an object.

In both cases, the methods execute migration code when they are invoked, usually performing more complex migrations.

Life-cycle-annotations are a powerful means to manage changes to the schema. Since they are executed on production data, they require great care: As we discuss in the following, some errors may be subtle and hard to detect by testing alone.

B. Schema Evolution Pitfalls with Life-Cycle Annotations

We now discuss common pitfalls in evolving the software without regard to the entities already persisted in production.

1) Retyping Attributes: Changing the type of a class member attribute can be problematic when legacy entities are loaded. When types are incompatible, runtime exceptions will be raised. Such errors can be difficult to trace and to debug, since they occur only when certain legacy entities are loaded.

For instance, let us assume that the NoSQL data store contains the following legacy player entity, where the level is a String value.

```
{ "kind":  "Player",
  "login": "ringbearer",
  "name":  "Frodo Baggins",
  "level": "beginner" }
```

Loading this entity according to the object mapper class declaration from Figure 2(a) triggers a runtime exception, since the String value cannot be converted into an Integer.

However, there are also cases where object mapper libraries can successfully convert types at runtime, e.g., a legacy value of type Boolean to a String. Yet even if the type conversion occurs without immediate runtime errors, the resulting string value can still cause problems if the application code does not expect the values “true” or “false”.

2) Renaming Attributes: Naively renaming class member attributes (without adding the appropriate life-cycle-annotations) can cause data loss, as already described in the scenario from the introduction: IDEs such as Eclipse provide convenient refactoring support, and consistently change all references to this attribute in the code. Yet when the new application loads a persisted legacy player with the new class declaration, its *level* data will not be loaded, since there is no matching Java class member. Instead, the *rank* attribute will be initialized to null, which may cause runtime errors if the remaining code relies on all attributes being initialized with meaningful values. Moreover, once the object is persisted in its new form, the value of *level* is irretrievably lost.

The safer way to rename an attribute is shown in Figure 3. The attribute is renamed lazily using the `@AlsoLoad` annotation. Yet as a developer, it is easy to loose track of the code history, and to simply forget to add the life-cycle annotation.

3) Ambiguous Mappings: With the class declaration from Figure 3, the object mapper Objectify raises a runtime exception when loading the legacy entity shown below, since the migration specification is ambiguous.

```
{ "kind":  "Player",
  "login": "ringbearer",
  "name":  "Frodo Baggins",
  "level": 1
  "rank":  6 }
```

Again, it is easy to overlook that the data store may contain legacy players that have both a *level* and a *rank* attribute.

4) Reintroducing Attributes: Reintroducing attributes that are still present in some legacy entities may yield unexpected values when loading these entities, since the legacy value may have been written with different semantics. For instance, let us assume the following legacy entity in the data store:

```
{ "kind":  "Player",
  "login": "sammy",
  "name":  "Sam Gamgee",
  "rank":  3 }
```

Here, the attribute *rank* denotes the ranking in the overall game, i.e., Sam is the third-best player. A developer is unaware of the existence of this entity. Starting from the class declaration in Figure 2(a), the developer renames *level* to *rank*, as shown in Figure 3. In the latest class declaration, the *rank* corresponds to the *level* the player has reached so far.

Now when our legacy entity is loaded with the new class declaration, the *rank* attribute can indeed be matched. However, the loaded value has different semantics.

With frequent releases, it becomes easy to lose track of the schema evolution history. Relying on the developers' discipline alone is risky. Thus, a framework that can automatically detect these pitfalls is of great value.

III. Static Type Checking

In checking an object mapper class declaration against the code history, we ask: Can all legacy entities be successfully loaded by the latest class declaration? We consider loading successful if it proceeds (1) without risking data loss due to unmapped attributes and (2) without runtime exceptions.

Our approach will flag some cases as problematic that might actually be fine. We therefore discuss pragmatic relaxations in the next chapter.

A. Basic Notation

We introduce *judgments* as expressions of the form

$$C \vdash \texttt{@annot}\ \textit{type}\ \textit{att};$$

where C is an object mapper class declaration annotated by `@Entity`. The judgment says that C declares attribute *att* with type *type* and annotation `@annot`.

Example 1. *The following judgments are derived from Figure 2. We denote the class declarations from subfigures (a) and (b) by $Player_a$ and $Player_b$ respectively:*

$$\begin{aligned} \textit{Player}_a \quad &\vdash \quad \texttt{@Id String login;} \\ \textit{Player}_b \quad &\vdash \quad \texttt{@Id String login;} \end{aligned}$$

□

In type checking, we compare pairs of object mapper class declarations C_W and C_R. We assume that both are different versions of a class C, and that C_W writes the entities that are then read (or loaded) by C_R. We therefore also consider judgments of the form

$$C_R \circ C_W \vdash \texttt{@annot}\ \textit{type}\ \textit{att};\quad \textit{ok}$$

which say that the class declaration C_R declaring class member attribute *att* of type *type* with annotation `@annot` is able to successfully read attribute *att* from the entity that was written according to the declaration C_W. (We use the function composition operator to express that the entity is first written according to C_W, and then read according to C_R.)

Example 2. *The class declaration $Player_b$ can successfully read the* login *attribute from an entity written according to the declaration of $Player_a$:*

$$\textit{Player}_b \circ \textit{Player}_a \vdash \texttt{@Id String login};\quad \textit{ok}$$

□

Judgments are then used in typing rules of the form

$$\frac{C_W \vdash \texttt{@annot}_1\ \textit{type}_1\ \textit{att}_1;}{C_R \circ C_W \vdash \texttt{@annot}_2\ \textit{type}_2\ \textit{att}_2;\quad \textit{ok}}\quad (\textit{cond})$$

This rule concerns the declaration of a class member attribute att_1 with type $type_1$ and annotation $\texttt{@annot}_1$ in the earlier class declaration C_W, and further the declaration of att_2 with type $type_2$ and annotation $\texttt{@annot}_2$ in the later declaration C_R. The rule states that when an entity is persisted according to C_W, attribute att_2 can be read successfully according to the declaration in C_R.

A rule can further carry a condition *cond*, which must be satisfied in order for the rule to apply.

Example 3. *By rule (1) in Figure 4, if an entity is written according to $Player_a$, then the declaration in $Player_b$ can successfully read attribute* login*:*

$$\frac{\text{Player}_a \vdash \texttt{@Id String login;}}{\text{Player}_b \circ \text{Player}_a \vdash \texttt{@Id String login;}\quad \textit{ok}}$$

□

Let C_W, C_R be two versions of an object mapper class declaration where C_W was released before C_R. We assume that entities written according to C_W are to be read according to C_R. We say C_R *type checks as valid w.r.t.* C_W if the following conditions hold:

- For each class member attribute declared in C_W, we can apply a type checking rule with the declaration from C_W in the premise (the judgement above the line), and
- for each class member attribute declared in C_R, we can apply a type checking rule with the declaration from C_R in the conclusion (the judgement below the line).

We say a new software release type checks if each object mapper class declaration type checks as valid w.r.t. each of its previously released versions.

B. Type Checking Object Mapper Class Declarations

We next present a core set of typing rules. We do not provide an exhaustive enumeration, but merely illustrate the most prominent cases since the remaining rules are similar.

1) Keeping, adding, and removing attributes: Figure 4 shows rules for basic changes to class member attributes.

Example 4. *Let us consider the class declarations in Figure 2. We assume that an entity is written according to $Player_a$ and then read according to $Player_b$. Example 3 has already shown that class member attribute* login *can be loaded successfully. By rule (2),* name *can be loaded successfully as well:*

$$\frac{\text{Player}_a \vdash \texttt{String name;}}{\text{Player}_b \circ \text{Player}_a \vdash \texttt{String name;}\quad \textit{ok}}$$

Rule (3) allows new class member attributes to be introduced. Applying this rule yields

$$\frac{}{\text{Player}_b \circ \text{Player}_a \vdash \texttt{String rank;}\quad \textit{ok}}\quad \begin{pmatrix}\textit{rank not declared}\\ \textit{in Player}_a\end{pmatrix}$$

As there is no rule that matches the implicit removal of level*, $Player_b$ does not type check as valid w.r.t. $Player_a$.* □

$$\frac{C_W \vdash \texttt{@Id}\ \textit{type}\ \textit{att};}{C_R \circ C_W \vdash \texttt{@Id}\ \textit{type}\ \textit{att};\quad \textit{ok}} \quad (1)$$

$$\frac{C_W \vdash \textit{type}\ \textit{att};}{C_R \circ C_W \vdash \textit{type}\ \textit{att};\quad \textit{ok}} \quad (2)$$

$$\frac{}{C_R \circ C_W \vdash \textit{type}\ \textit{att};\quad \textit{ok}} \quad \begin{pmatrix}\textit{att}\text{ not declared}\\ \text{in } C_W\end{pmatrix} \quad (3)$$

$$\frac{C_W \vdash \texttt{@Ignore}\ \textit{type}_1\ \textit{att};}{C_R \circ C_W \vdash \textit{type}_2\ \textit{att};\quad \textit{ok}} \quad (4)$$

$$\frac{C_W \vdash \textit{type}\ \textit{att};}{C_R \circ C_W \vdash \texttt{@IgnoreLoad}\ \textit{type}\ \textit{att};\quad \textit{ok}} \quad (5)$$

$$\frac{C_W \vdash \textit{type}\ \textit{att};}{C_R \circ C_W \vdash \texttt{@IgnoreSave}\ \textit{type}\ \textit{att};\quad \textit{ok}} \quad (6)$$

$$\frac{C_W \vdash \textit{type}\ \textit{att};}{C_R \circ C_W \vdash \texttt{@Ignore}\ \textit{type}\ \textit{att};\quad \textit{ok}} \quad (7)$$

Fig. 4: Selected rules for (1, 2) keeping an attribute, lazily (3, 4) adding, (5) overwriting, and (6, 7) explicitly removing an attribute.

2) Renaming Attributes: Figure 5 lists rules concerning the annotation `@AlsoLoad`. They detect ambiguous mappings in renamings (c.f., Section II-B3).

3) Ignoring Attributes: Rule 13 in Figure 5 concerns the annotations for ignoring attributes in writing or reading. Ignoring annotations can be combined freely without risk.

C. Nontranstitivity of Type Checking Rules

Our type checking rules are not transitive in the following sense: Given a sequence of versions X, Y, and Z of an object mapper class declaration, if Y type checks as valid w.r.t. X, and Z type checks as valid w.r.t. Y, it does not necessarily follow that Z type checks as valid w.r.t. X. It turns out that non-transitivity is helpful in detecting actual problems, as the next example shows.

Example 5. *We assume a sequence of versions X, Y, and Z of an object mapper class declaration, with the following judgments for attribute* level*:*

$$X \vdash \texttt{String level};$$
$$Y \vdash \texttt{@Ignore String level};$$
$$Z \vdash \texttt{Integer level};$$

Then according to rules (7) and (4), it holds that

$$\frac{X \vdash \texttt{String level};}{Y \circ X \vdash \texttt{@Ignore String level};\quad \textit{ok}}$$

$$\frac{Y \vdash \texttt{@Ignore String level};}{Z \circ Y \vdash \texttt{Integer level};\quad \textit{ok}}$$

yet Z does not type check as valid w.r.t. X. Thus, the release of Z does not type check, due to issues with retyping. □

In the example above, attribute *level* is re-introduced with a different type. If the type had not been changed, then our type checking rules would not have detected an issue. To be able to capture this case as well, we extend our approach to checking chains of object mapper class declarations (rather than pairs) in Section IV-B.

IV. Relaxations and Extensions

We next discuss pragmatic relaxations and extensions.

A. Allowable Type Promotions

Our examples so far have only considered basic types for class member attributes, such as String and Integer. We can easily generalize to complex types.

Example 6. *We consider the following judgments from two different versions of the Player class declaration:*

$$\textit{Player}_c \vdash \texttt{@Embedded}\ A\ \texttt{description};$$
$$\textit{Player}_d \vdash \texttt{@Embedded}\ B\ \texttt{description};$$

The Objectify annotation `@Embedded` *stores structured data within a single entity in a way so that it remains queryable. We assume that A and B are different Java class names.* □

By our type checking approach, Player_d does not check as valid w.r.t Player_c, since the types for attribute *description* differ. Yet there are cases where this rule is overly restrictive.

Rule (14) below relaxes rule (2). We use the notation $t \triangleright t'$ to express that type t is *compatible* with t'. This means that any instance of t must be "loadable" as an instance of t' without data loss. The trivial case is that $t = t'$, yet we can also allow for Short $\triangleright$ Integer, since all instances of Short can be converted to Integers.

$$\frac{C_W \vdash \textit{type}\ \textit{att};}{C_R \circ C_W \vdash\ \textit{type}'\ \textit{att};\quad \textit{ok}} \quad (\textit{type} \triangleright \textit{type}') \qquad (14)$$

We look at type compatibility in more detail.

1) Conversion of Primitive Types: For the remainder of this section, we focus on primitive types[1] (i.e., Void, Boolean, Byte, Short, Integer, Long, Float, Double, Character, String and Object), and we discuss the risks when primitive type attributes are loaded from a datastore.

a) Successful Conversion: A number of primitive types can be converted safely into other primitive types, for instance, Short$\triangleright$Integer, Float$\triangleright$Double, and Integer$\triangleright$String. Typically, a primitive type t can be converted to a type t' if the memory allocated to store objects of type t' is larger (or equal) than the one allocated to objects of type t.

b) Incorrect Conversion: When a type t cannot be converted to a type t', an exception may be raised. For instance, in general it is not possible to convert a literal into a number, so the conversion of a String into a Short will raise an exception.

[1]We somewhat abusively use the terms *primitive types* to refer to classes of the `java.lang` package that wrap Java primitive types. Void, Character and Object are not considered, as Objectify does not support them in entities.

$$\frac{C_W \vdash \mathit{type}\ \mathit{att}_1;}{C_R \circ C_W \vdash \texttt{@AlsoLoad("}\mathit{att}_2\texttt{")}\ \mathit{type}\ \mathit{att}_1; \quad \mathit{ok}} \quad (\mathit{att}_2 \text{ not declared in } C_W) \tag{8}$$

$$\frac{C_W \vdash \mathit{type}\ \mathit{att}_2;}{C_R \circ C_W \vdash \texttt{@AlsoLoad("}\mathit{att}_2\texttt{")}\ \mathit{type}\ \mathit{att}_1; \quad \mathit{ok}} \quad (\mathit{att}_1 \text{ not declared in } C_W) \tag{9}$$

$$\frac{C_W \vdash \texttt{@annot}\ \mathit{type}\ \mathit{att}_1 \qquad C_W \vdash \mathit{type}\ \mathit{att}_2;}{C_R \circ C_W \vdash \texttt{@AlsoLoad("}\mathit{att}_2\texttt{")}\ \mathit{type}\ \mathit{att}_1; \quad \mathit{ok}} \quad (\texttt{@annot is @Ignore or @IgnoreSave}) \tag{10}$$

$$\frac{C_W \vdash \mathit{type}\ \mathit{att}_1 \qquad C_W \vdash \texttt{@annot}\ \mathit{type}\ \mathit{att}_2;}{C_R \circ C_W \vdash \texttt{@AlsoLoad("}\mathit{att}_2\texttt{")}\ \mathit{type}\ \mathit{att}_1; \quad \mathit{ok}} \quad (\texttt{@annot is @Ignore or @IgnoreSave}) \tag{11}$$

$$\frac{}{C_R \circ C_W \vdash \texttt{@AlsoLoad("}\mathit{att}_2\texttt{")}\ \mathit{type}\ \mathit{att}_1; \quad \mathit{ok}} \quad (\mathit{att}_1 \text{ and } \mathit{att}_2 \text{ not declared in } C_W) \tag{12}$$

$$\frac{C_W \vdash \texttt{@annot}_1\ \mathit{type}\ \mathit{att}_1;}{C_R \circ C_W \vdash \texttt{@annot}_2\ \mathit{type}\ \mathit{att}_1; \quad \mathit{ok}} \quad \left(\begin{array}{c}\texttt{@annot}_1 \text{ and } \texttt{@annot}_2 \text{ are any of} \\ \texttt{@Ignore}, \texttt{@IgnoreSave}, \texttt{@IgnoreLoad}\end{array}\right) \tag{13}$$

Fig. 5: Selected rules for the `@AlsoLoad` annotation (8 - 12) and for ignoring attributes (13).

c) Conversion Returning a Corrupted Value: Surprisingly, there are cases where Objectify allows a conversion, even when types cannot be casted in Java. For instance, the conversion of a positive Integer to a Short may return a negative number. This situation is particularly dangerous from a development and testing point-of-view as, since it does not raise an exception, developers might not expect to be able to convert an Integer to a Short. So when a corrupted value is retrieved, they might not suspect a conversion problem, and the problem does not appear systematically (it only appears for certain values), which complicates testing.

B. Checking Chains of Class Declarations

We consider the unintentional reintroduction of attributes a schema evolution pitfall (c.f. Section II-B4). These cases are difficult to anticipate for developers, as the attribute that existed in previous versions may have been removed from the source code several releases back. Yet as discussed in Section III-C, we may not always capture this problem by type checking pairs of class declaration versions.

In order to reliably detect these problems at development time, we check the whole sequence of class declarations released into production to detect when attributes by the same name have been removed and are now being reintroduced. The necessary information is usually accessible from within the IDE, since the source code repository is commonly integrated with the IDE.

V. The ControVol Framework

As a proof-of-concept, we have implemented our type checking rules as the ControVol Eclipse plugin. We briefly highlight the core features of ControVol, and refer to [16] and [17] for details on the workflow as experienced by developers. Our ControVol prototype currently supports Java development against Google Cloud Datastore [1], using the Objectify object mapper library. It is straightforward to extend ControVol to other IDEs, NoSQL data stores, and object mapper libraries.

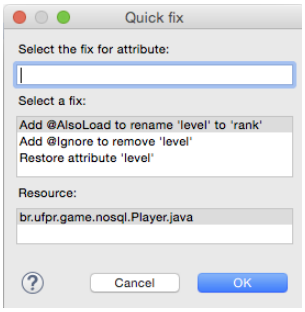


Fig. 6: ControVol suggests quick fixes to resolve warnings.

ControVol captures changes to object mapper class declarations during the IDE-integrated build process. ControVol then compares object mapper class declarations to the release history, as managed by the code repository. The plugin is able to detect common schema evolution pitfalls involving adding, renaming, and removing attributes. ControVol then issues warnings accordingly.

ControVol can even suggest IDE-supported quick fixes to help resolve problems. For instance, in the case of the renaming problem triggered by the refactoring in Figure 2, the ControVol dialog in Figure 6 proposes several fixes:

- Adding the Objectify annotation `@AlsoLoad` lazily renames *level* to *rank*. This ensures that no values are lost.
- Adding annotation `@Ignore` makes clear that attribute *level* is intentionally discarded.
- Restoring attribute *level* prevents that its value is lost. In this case, attributes *level* and *rank* co-exist.

ControVol currently type checks Java class declarations w.r.t. all Objectify life-cycle annotations for class member attributes, as listed in Section II. In the future, we plan to also type check life-cycle annotations for methods. Yet since the methods may contain arbitrary Java code, these are

undecidable problems when addressed in full generality. Thus, we will need to focus on a pragmatic subset of transformations.

VI. Related Work

Managing schema evolution has a long history in database research (c.f. [3]), although it has not yet been thoroughly addressed in the domain of NoSQL data stores.

Among the related and contemporary systems known to us is Kiji [14], a NoSQL-backed middleware for data analytics. Schema evolution is a major concern in data analytics, due to continuously changing requirements in organizations. In Kiji, the schema evolution is recorded in data dictionaries and a registration process validates whether the new upcoming schema is ready for production. The validation process compares the old and new schema versions. Changes that can lead to data loss flag the new version as incompatible, and it is not deployed to production. Type demotions that lead to loss of precision are therefore not allowed. Contrasting Kiji with our work, both projects share the basic notion of distinguishing read and write schemas. However, while we interpret class declarations as schema specifications, Kiji maintains its own data dictionary. Also, ControVol not only detects schema incompatibility, but also proposes fixes.

In much earlier work [19], we have found a version control mechanism for schemas in object-oriented databases. The goal is to associate objects with the schema versions to which they belong. When an object belongs to a set of versions, the mechanism creates an "object life-cycle" divided into several periods. For each period, an object has its own context in terms of data and integrity rules. One downside in adapting this earlier work to the context of NoSQL data stores is that this mechanism requires access to data dictionaries, which is not necessarily a given with NoSQL data stores.

PRISM [6], [20] has been designed for relational data stores and also relies on data dictionaries. It allows tracking any schema modification, in contrast to [19]. For this, it relies on a custom language to support DBAs carrying out safe schema evolution. With F1 [7], Google has designed a highly scalable data store with a relational schema where daily schema changes are rolled out safely and asynchronously across a cluster of nodes.

There are several other research contributions to support schema evolution [8]–[10], but they cannot be directly applied to our context. They were not designed with NoSQL data stores in mind, but for relational databases, or they do not target the widely adapted web programming languages. Our contribution differs from them in the sense that we provide direct feedback to developers (not yet aware of the evolution pitfalls) instead of DBAs, and we infer the schema evolution from the code source itself instead of data dictionaries.

Our static type checking scheme was originally inspired by the earlier work on the Machiavelly type system [21] for typing heterogeneous objects in object-oriented data stores.

VII. Conclusion

Schema evolution has been intensively studied for relational, object-oriented, and XML databases. The problem presents itself with new acuteness in building Big Data software systems on top of schemaless NoSQL data stores. When developers use object mapper libraries capable of lazy schema migration, there is a need to ensure that migrations are robust and safe.

In this paper, we present a static type checking scheme that detects serious schema migration pitfalls already during the development process. When object mapper class declarations do not type-check against their earlier versions recorded in the software repository, our ControVol plugin can warn developers already from within IDE. Moreover, in many cases we can also offer useful quick fixes, so that developers may automatically resolve the problems detected.

Acknowledgment

This project was partially funded by Serpro Brazil and CNPq grant 441944/2014-0.

References

[1] Google Developers, "Google Cloud Datastore," 2015, https://developers.google.com/datastore/.
[2] "MongoDB," 2015, http://www.mongodb.org/.
[3] M. Hartung, J. F. Terwilliger, and E. Rahm, "Recent Advances in Schema and Ontology Evolution," in *Schema Matching and Mapping*, 2011, pp. 149–190.
[4] S. Scherzinger, M. Klettke, and U. Störl, "Managing Schema Evolution in NoSQL Data Stores," in *Proc. DBPL*, 2013.
[5] S. Ambler, *Agile Database Techniques: Effective Strategies for the Agile Software Developer*. New York, NY, USA: John Wiley & Sons, Inc., 2003.
[6] C. Curino, H. J. Moon, A. Deutsch, and C. Zaniolo, "Automating the Database Schema Evolution Process," *The VLDB Journal*, vol. 22, no. 1, pp. 73–98, 2013.
[7] I. Rae, E. Rollins, J. Shute, S. Sodhi, and R. Vingralek, "Online, Asynchronous Schema Change in F1." *PVLDB*, vol. 6, no. 11, pp. 1045–1056, 2013.
[8] M. Piccioni, M. Oriol, and B. Meyer, "Class Schema Evolution for Persistent Object-Oriented Software: Model, Empirical Study, and Automated Support," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 184–196, 2013.
[9] B. S. Lerner, "A model for compound type changes encountered in schema evolution," *ACM Trans. Database Syst.*, vol. 25, no. 1, pp. 83–127, 2000.
[10] T. Milo and S. Zohar, "Using schema matching to simplify heterogeneous data translation," in *Proc. VLDB'89*, 1998, pp. 122–133.
[11] S. Lightstone, *Making it Big in Software*. Prentice Hall, 2010.
[12] "Objectify," 2015, https://code.google.com/p/objectify-appengine/.
[13] "Morphia. A type-safe Java library for MongoDB," 2015, https://github.com/mongodb/morphia/.
[14] "Kiji Project," 2015, http://www.kiji.org/.
[15] M. P. Atkinson and O. P. Buneman, "Types and Persistence in Database Programming Languages," *ACM Computing Surveys*, vol. 19, no. 2, pp. 105–170, 1987.
[16] S. Scherzinger, E. C. de Almeida, and T. Cerqueus, "ControVol: A Framework for Controlled Schema Evolution in NoSQL Application Development," in *Proc. ICDE'15, demo paper*, 2015.
[17] T. Cerqueus, E. C. de Almeida, and S. Scherzinger, "ControVol: Let yesterday's data catch up with today's application code," in *Proc. WWW'15, poster*, 2015.
[18] U. Störl, T. Hauf, M. Klettke, and S. Scherzinger, "Schemaless NoSQL Data Stores – Object-NoSQL Mappers to the Rescue?" in *Proc. BTW'15*, 2015.
[19] J. Andany, M. Léonard, and C. Palisser, "Management of schema evolution in databases," in *Proc. VLDB'91*, 1991, pp. 161–170.
[20] C. Curino, H. J. Moon, L. Tanca, and C. Zaniolo, "Schema Evolution in Wikipedia - Toward a Web Information System Benchmark," in *Proc. ICEIS (1)*, 2008, pp. 323–332.
[21] P. Buneman and A. Ohori, "Polymorphism and Type Inference in Database Programming," *ACM Transactions on Database Systems*, vol. 21, no. 1, pp. 30–76, 1996.