Supporting Schema Evolution in Schema-Less NoSQL Data Stores

Loup Meurice and Anthony Cleve
PReCISE Research Center, University of Namur, Belgium
{loup.meurice,anthony.cleve}@unamur.be

***Abstract*—NoSQL data stores are becoming popular due to their schema-less nature. They offer a high level of flexibility, since they do not require to declare a global schema. Thus, the data model is maintained within the application source code. However, due to this flexibility, developers have to struggle with a growing data structure entropy and to manage legacy data. Moreover, support to schema evolution is lacking, which may lead to runtime errors or irretrievable data loss, if not properly handled. This paper presents an approach to support the evolution of a schema-less NoSQL data store by analyzing the application source code and its history. We motivate this approach on a subject system and explain how useful it is to understand the present database structure and facilitate future developments.**

I. Introduction

NoSQL data stores are becoming increasingly popular in the context of big data software development. These data stores were designed to manipulate big volumes of data that are not organized according to the relational model. NoSQL technologies were introduced to address some relational database limitations: simplicity of design, faster query execution and flexibility. Indeed, most NoSQL data stores are schema-less and can thus manage data with ever-changing structures. In a continuously changing environment, database schema evolution becomes an unavoidable activity and therefore, proposing such a flexibility is a precious asset. Schema-less NoSQL data stores do not require developers to specify a global schema, which makes data evolution simpler. For instance, adding new fields to a data structure can be done at any time and instantly.

However, this flexibility may lead to an increasing data structure entropy within the system. When the schema evolves, the outdated entities must be migrated to fit with the new structures. Nevertheless, migrating data may be time-consuming and expensive; especially when a huge amount of data has to be migrated or when the system is contractually linked to a database-as-a-service provider for all data store reads and writes. As a consequence, data migration may never be achieved and thus, data entities of different schema versions may co-exist in the data store. Figure 1 illustrates an example of co-existence of legacy and up-to-date data entities within the same NoSQL database, after several changes of the data structure. Such an entropy may prove error-prone. For instance, conflictual entities can cause runtime errors and data loss, or can even corrupt the database, if not handled properly. For instance, changing the type of a particular field requires to deal with both the legacy and the up-to-date entities when manipulating data in the program. In other words, in NoSQL data stores, the past belongs to the present, and clearly affects the future. Therefore, understanding schema evolution in schema-less NoSQL databases is essential for future developments.

[0]The first author is supported by the F.R.S.-FNRS via the DISSE project.

In *schema-less* data stores, no explicit database schema is declared by developers. Thus, the main source of information concerning the data structures is the source code itself. In particular, the database writes and reads located in the source code give concrete clues about data structures. Among others NoSQL technologies, MongoDB is a schema-less document-oriented database. It stores JSON-like documents in *collections*. Collections are similar to tables in relational databases, and are composed of *fields* [1].
Figure 2 depicts an example of Java code manipulating entities from a MongoDB database. Useful information can be extracted from that code sample; the existence of several collections as well as the type of some of their fields can be inferred. Moreover, analyzing how the source code (especially the database-related code) evolved over time can significantly help developers to understand how the schema evolved and thus to prevent potentially severe errors.

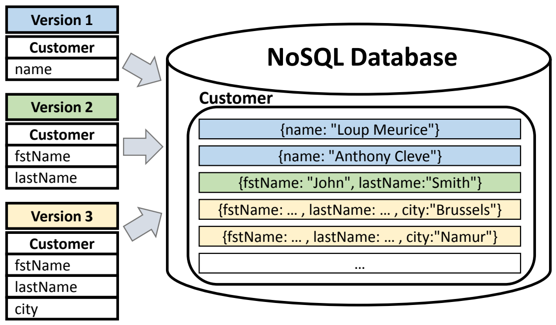

Fig. 1. Example of the co-existence of legacy and up-to-date entities within the same NoSQL database.

This paper presents an automatic approach that aims at supporting schema evolution in NoSQL data stores. The paper makes two main contributions: (1) an automatic approach that infers the database schema of a schema-less NoSQL data store by analyzing the application source code and (2) the application of this approach to the whole system history in order to understand schema evolution and to prevent errors and data losses. The remainder of this paper is structured as follows. Section II presents our approach. Section III illustrates the benefits of our approach on a subject system. A related work discussion is provided in Section IV. Concluding remarks are given in Section V.

II. Approach

This section presents our automatic approach allowing developers to understand and analyze schema evolution in schema-less NoSQL data stores. Our approach, summarized in Figure 3, is made up of three phases, namely *schema extraction*, *historical schema extraction* and *exploitation*.

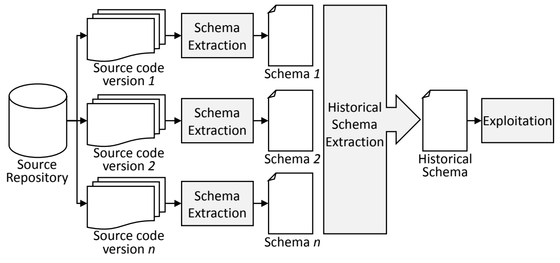

Fig. 3. Overview of our approach.

A. Schema Extraction

As previously explained, useful information about the NoSQL database schema can be extracted by analyzing the source code, especially those code locations accessing the database. The first step of our automatic approach aims to infer the database schema by *statically* analyzing the database accesses from the source code. Our technique is currently implemented for systems using the MongoDB Java driver to communicate with the database. The choice for Java is because it is the most popular programming language today according to different sources such as the TIOBE Programming Community index [4]. Moreover, we focus on MongoDB which is currently ranked among the top five database systems and at first position among the NoSQL database systems [2].

For describing our technique, we will consider the Java code example depicted in Figure 2. If one further observes this code, one can detect the presence of three database accesses, at lines 3, 5 and 23, respectively. (1) Line 3 reads the database to find a particular *author* based on a given identifier; (2) line 5 reads the database to find a particular *show* based on a given identifier; (3) line 23 updates a particular *author*.
Through this example and by reading the Java MongoDB API documentation[1], we can make two important observations:

1) A database access may use one or several selection criteria to create the query (e.g., `authorQuery` at line 3, `showQuery` at line 5 and, `authorQuery` and `author` at line 23). Thus, analyzing the operations performed on those objects *before* the database access execution, brings added information about the collection fields concerned by the selection criteria.

2) A database access may return a set of objects resulting from the operation (e.g., `author` and `show` variables respectively at line 3 and 5, storing the query results). Analyzing the read operations performed on those objects *after* the access execution, gives indications about the fields they contain.

[1]https://api.mongodb.com/java/current/

In other words, information about the database structures may be inferred by analyzing (1) the usage flow of the query inputs **before** query execution and (2) the usage flow of the query outputs **after** query execution. A first algorithm can be proposed:

```
1 foreach access $\in$ getAccesses() do
2     getCollectionNames(access);
3     foreach input $\in$ access.inputs do
4         analyzeUsageFlowBefore(input);
5     end
6     foreach output $\in$ access.outputs do
7         analyzeUsageFlowAfter(output);
8     end
9 end
```

Line 1 detects the code locations accessing the database. We achieve this first step by exhaustively listing the set of Java methods provided by the MongoDB API that allows developers to query the database. Once this list established, we use a *visitor* class which parses the Java code and detects any MongoDB database accesses. For each detected access, we have to determine which collection(s) is queried (line 2). This information can be obtained by analyzing instructions in the form of `DB.getCollection(String collectionName)`. The answer is in the value affected to `collectionName`. However this value depends on the call graph of the application and the intra-procedural control-flow of the methods. Indeed, building a string value may necessitate to pass through different statements and boolean conditions (e.g., for, while, if-then-else statements). Thus, our static analysis has to consider all the possible program paths. Similarly, the string value construction may be done by successive concatenations of string fragments or by using some input parameters of the local method. Therefore, we need to consider the call graph of the application to get the actual values of the string input parameters. To achieve this task of string reconstruction, we use the tool support we developed in our previous work dedicated to database access recovery in Java source code [6]. We developed an extended version of this static analysis approach to automatically reconstruct a string value by exploring the call graph of the application and the intra-procedural control-flow of the methods. Let us come back to our example in Figure 2: our static analyzer automatically detects that the database accesses at line 3, 5 and 23 actually query, respectively, the *author*, *show* and *author* collections.

The next step consists in analyzing the input objects which serve as selection criteria for the query creation (`analyzeUsageFlowBefore` procedure). Thanks to the MongoDB API, we pointed out that the creation criteria are mainly expressed by means of the `DBObject`, `BasicDBObject` and `BasicDBList` classes. An instance of those classes is a key-value map that can be stored in the database, where the key represents a field name and the value is the field value. Accordingly, analyzing the usage flow of that map from its creation until the access execution allows us to spot the fields used to define the query. To realize this task, we

```java
1  public String save(ContributionToSave contributionToSave) {
2          BasicDBObject authorQuery = new BasicDBObject("_id", new ObjectId(contributionToSave.getAuthor().getId()));
3          DBObject author = db.getCollection("author").findOne(authorQuery);
4          BasicDBObject showQuery = new BasicDBObject("_id", new ObjectId(contributionToSave.getShow().getId()));
5          DBObject show = db.getCollection("show").findOne(showQuery);
6          addContributionToAuthor(contributionToSave, authorQuery, author, show);
7          return "ok"; }
8
9  private void addContributionToAuthor(ContributionToSave contributionToSave, BasicDBObject authorQuery, DBObject author,
       DBObject show) {
10         BasicDBList contributions = (BasicDBList) author.get("contributions");
11         if (contributions == null) {
12             contributions = new BasicDBList();
13             author.put("contributions", contributions);
14         }
15         BasicDBObject contribution = new BasicDBObject();
16         contribution.put("nick", contributionToSave.getNick());
17         BasicDBObject contributionShow = new BasicDBObject();
18         contributionShow.put("alias", show.get("alias"));
19         contributionShow.put("name", (String) show.get("name"));
20         contributionShow.put("ref", new DBRef(db, "show", show.get("_id")));
21         contribution.put("show", contributionShow);
22         contributions.add(contribution);
23         db.getCollection("author").update(authorQuery, author); }
```

Fig. 2. Java code example using the MongoDB API to access the database.

overloaded our static analyzer so that it can control the usage flow of the inputs. For instance, the database access at line 3 uses a unique selection criterion to create its query, i.e., the `authorQuery` object. By analyzing the usage flow of this given object (and by reusing our string value extractor), our analyzer automatically spots line 2 which actually represents a value assignment to the `author._id` field. It is worth noticing that the `analyzeUsageFlowBefore` procedure is actually recursive. Indeed, the value assigned to a particular key in the map may be, itself, an instance of the `DBObject`, `BasicDBObject` and `BasicDBList` classes and thus, a recursive call is needed to analyze that instance.

The final step consists in analyzing the usage flow of the output objects resulting of the database access (`analyzeUsageFlowAfter` procedure). Indeed, analyzing the operations performed on those output objects may reveal new fields of the target collection. This step is similar to the previous one, the only difference being that, instead of analyzing the object usage flow before the database access, it focuses on the object usage flow after the database access. In Figure 2, variable `show` contains the result of the database access at line 5. However, analyzing the usage flow of an object after a given event (i.e., a database access) requires the analysis of the application call graph, since the object may be part of the input parameters of a method call. In the example, our static analyzer determines that `show` is used as input in the `addContributionToAuthor` method call (line 6), and it visits this method to observe how the object is manipulated. At line 18, 19 and 20, the analyzer detects the read of, respectively, the `alias`, `name` and `_id` fields.

As output, the analyzer returns the database schema fragment which is concerned by each detected database access. Finally, our analyzer merges all the extracted schema fragments in order to obtain a unique condensed schema. Figure 4 depicts the schema (according to the Entity-Relationship model) automatically inferred by our approach when applied to Figure 2. Our analyzer is also able to deal with the referential constraints, i.e., *foreign keys*, declared in the source code. At line 20 in Figure 2, a referential constraint is declared between `author.contributions.show.ref` and `show._id`. Indeed, `DBRef` allows documents located in multiple collections to be more easily linked to documents from a single collection.

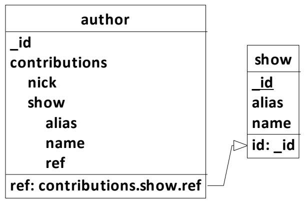

Fig. 4. Schema automatically inferred by our approach applied to Figure 2.

The field extraction process also tries to gain information about the types of the fields. For instance, after detecting an access to the `name` field at line 19, our analyzer considers the extracted field as a `String` object.

B. Historical Schema Extraction

The second step of our approach aims to apply the schema extraction process to the whole system history by exploiting the versioning system. This step, inspired by our previous work [5], consists in extracting and comparing the successive versions of the database schema, in order to produce the so-called *historical database schema*. The latter is a representation of the database schema evolution over time. It contains all database schema objects (i.e., collections, fields and referential constraints) that have existed in the history of the system. Those schema objects are annotated with meta-information about their lifetime such as (1) the list of schema versions where the object is present and (2) for each version of this list, the code locations accessing the object. In addition, each field owns information about its data type and its evolution; one can know the data type of any field at any version of the system. In this way, one has an accurate overview of the field evolution over time which could allow detecting error-prone data type changes. The historical database schema thus constitutes an integrated representation of the system past and present. Exploiting this historical schema can help to detect potential runtime errors or data corruptions and to facilitate future developments.

Finally, we apply an automatic procedure that colourizes each historical schema object, depending on its age and its liveness. All schema objects depicted in green are still present in the latest schema version. All red schema objects have disappeared. The colour shade corresponds to the object age. A dark red schema object is an object that has disappeared a long time ago. A light red object is an object that has recently disappeared from the schema. An object depicted in green corresponds to an object that is still present in the latest schema version. The darker the green, the older the object is, and vice versa. Figure 5 shows an example of schema evolution and the corresponding colourized historical schema.

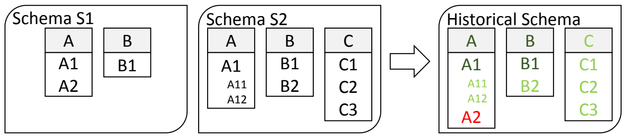

Fig. 5. Example of schema evolution and the corresponding historical schema.

C. Exploitation

Exploiting the historical schema can facilitate the understanding of the schema evolution, and it can allow developers to spot potentially severe runtime errors or irretrievable data losses, together with the related code locations.

Colourization benefits. The color assigned to an object gives indications about its liveness. The red color is assigned to objects which have disappeared in the latest schema version. However, a red object does not necessarily represent a deleted object; it might only represent a field/collection that is no longer accessed in the latest source code version but that still exists in the database schema. Moreover, even if a red object is actually deleted from the schema, it does not ensure that there is no legacy data linked to this object. A red object only represents a soft warning to make developers aware of potentially outdated entities which should be either migrated or kept in mind for future developments.

Type mismatch detection. As previously explained, a data type change may cause error/crash if not properly managed. Detecting a data type change occurred over time is made possible by the historical schema and its meta-information.

Renaming detection. When a field or a collection is renamed or moved, developers need to keep it in mind for future developments. Indeed, similarly to a data type change, the legacy data have to be managed. Therefore, our automatic approach supports the identification of implicit (field/collection) renamings. The detection algorithm is based on different comparison criteria (e.g., name similarity, the field type similarity, etc.).

Data corruption/loss detection. Since a schema-less data store does not require an explicit schema, no verification before a data write is done, which might cause a data corruption/loss (e.g., accidentally removing/erasing stored data). Analyzing the historical schema of the system can help detecting such scenarios.

III. Early Evaluation

We applied our approach to a particular subject system containing the backend services of the *Tilos Radio*[2]. The Tilos Radio is a community, non-profit radio station in Budapest, Hungary. The versioning system of this project has a two-year history [3]. Since the introduction of MongoDB in the project, 303 versions of the system were committed on a period of one year. We applied our schema extraction approach to each version and computed the corresponding historical schema. In this period, the number of fields has doubled (from 39 to 79).

The historical schema, shown in Figure 6, is visualized by our visualization tool. The latter provides developers with reports about what happened in the system past. Icons warn developers of particular past events; clicking on those icons allows one to display automatic reports about those events. *Error* icons report on past events that might cause program crashes or data corruption. *Warning* icons aim to make developers aware of past events that should be considered for future developments (e.g., renamed fields/collections).

By analyzing that historical schema and the automatic reports processed by our tool, we made interesting observations concerning the schema evolution of this subject system. Figure 7 shows an example of data type change detected by our tool. The type of the `comment.identifier` field changed at version 68 (from Integer to String). Furthermore, the tool provides the user with direct links to the source code showing the code locations where the change was performed[3]. Developers are now warned that this past data type change might require to be handled for future developments (e.g., migrating legacy data to new type, managing the reads of legacy data in the source code, ...).

Another interesting observation is the automatic detection of a particular renaming (see Figure 8) occurred at version 198: the `bookmark` collection was moved (renamed) and became a *compound* field of the `episode` collection[4]. Thus, the remaining legacy entities of the outdated `bookmark` collection should be managed accordingly by developers.

Figure 9 shows an example of potential data loss occurred in the system past. Our approach automatically spotted a potential data loss due to a misuse of the `user.passwordChangeTokenCreated` field. Indeed, developers assigned a wrong value to this field, which overwrites the correct value [5]. This mistake stayed unfixed during 20 system versions (from version 0 to version 19), what could represent an important data loss since the correct values to store in the database are definitely lost. Further analysis revealed that the wrong value should have been assigned to another field of the `user` collection.

IV. Related Work

Recent approaches and studies have focused on the evolution of NoSQL databases. Scherzinger et al. [8] present a

[2]https://tilos.hu/page/english
[3]Before change:http://bit.ly/2dk4HEQ. After change:http://bit.ly/2cW5NnC
[4]Before move:http://bit.ly/2d35A1b. After move:http://bit.ly/2dlzLQw
[5]Before fix:http://bit.ly/2cxI1A3. After fix:http://bit.ly/2cPbQII

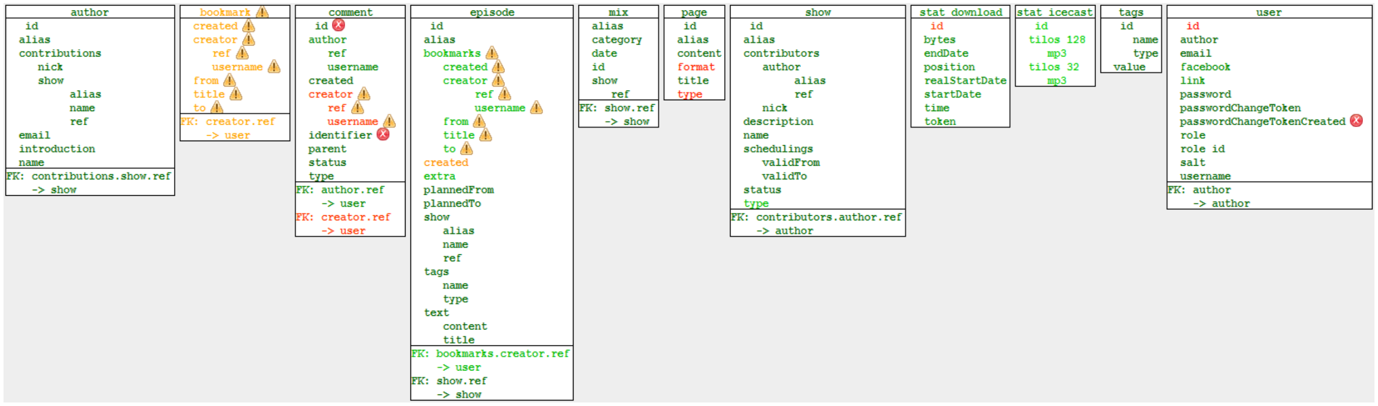

Fig. 6. Historical schema of Radio Tilos displayed by our visualization tool. Error and warning icons report on particular past events.

(a)
```
public List<CommentData> list(CommentType type, int id) {
    BasicDBObject query = new BasicDBObject();
    query.put("identifier", id);
    DBCursor comments = db.getCollection("comment").find(query);
```
(b)
```
public List<CommentData> list(CommentType type, String id) {
    BasicDBObject query = new BasicDBObject();
    query.put("identifier", id);
    DBCursor comments = db.getCollection("comment").find(query);
```

Fig. 7. Before and after changing the type of `identifier` at version 68.

(a)
```
BasicDBObject bookmark = new BasicDBObject();
bookmark.put("created", new Date());
...
db.getCollection("bookmark").insert(bookmark);
```
(b)
```
DBObject episode = db.getCollection("episode").findOne(q);
BasicDBObject bookmark = new BasicDBObject();
bookmark.put("created", new Date());
...
if (episode.get("bookmarks") == null)
    episode.put("bookmarks", new BasicDBList());
((BasicDBList) episode.get("bookmarks")).add(bookmark);
```

Fig. 8. Before and after moving `bookmark` in `episode` at version 198.

```
DBObject user = db.getCollection("user").findOne(
    new BasicDBObject("email", passwordReset.getEmail()));
String token = authUtil.generateSalt();
user.put("passwordChangeTokenCreated", new Date());
user.put("passwordChangeTokenCreated", token);
db.getCollection("user").update(
    new BasicDBObject("username", user.get("username")), user);
```

Fig. 9. Loss of the value to store in `passwordChangeTokenCreated`.

model checking approach to reveal scalability bottlenecks in NoSQL schemas. In [9], the authors describe a framework controlling schema evolution in NoSQL applications. Ringlstetter et al. [7] analyzed how developers evolve NoSQL document stores by means of evolution annotations for object-NoSQL mappers. Object-NoSQL mappers allow defining mappings between classes/attributes and NoSQL entities. They rely on annotations within class declarations in the application source code. Those mappers make the communication between the database and the source code more abstract and thus, can facilitate the comprehension of the schema, since the latter is *implicitly* declared within the source code. In contrast to those works, we focus on NoSQL databases where the schema is not declared. In a previous work, we presented an approach [6] allowing developers to automatically locate and extract all the relational database accesses that use JDBC, Hibernate and JPA.

In summary, this paper presents two main novel contributions: (1) a static analysis approach which extracts the NoSQL database schema from the application source code (as unique information source) by exploring the call graph and the intra-procedural control-flow of the application; (2) a historical analysis which helps developers to understand the schema evolution and allows the automatic detection of potential errors and data losses.

V. Conclusion

In this paper, we presented an automatic approach to infer the schema of a schema-less NoSQL database, and to analyze its evolution over time. We applied this approach to the whole history of a subject system and we computed the so-called historical schema. We finally showed how analyzing the past of a system, by using this historical schema, can be useful to understand the present version and to ease future developments. In particular, our approach automatically detects and warns developers about potential risks, such as past data structure changes, data type mismatches and data losses. In the future, we intend to lead empirical studies on a large set of systems to analyze how developers evolve NoSQL databases in practice and to further study the entropy introduced by this evolution.

References

[1] https://docs.mongodb.com/manual/. Accessed: 2016-09-21.
[2] http://db-engines.com/en/ranking. Accessed: 2016-09-21.
[3] https://github.com/tilosradio/web2-backend/. Accessed: 2016-09-21.
[4] Tiobe programming community index. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html. Accessed: 2016-09-21.
[5] A. Cleve, L. Meurice, M. Gobert, J. Maes, and J. Weber. Understanding database schema evolution: A case study. *Science of Computer Programming*, 97:113–121, 2015.
[6] L. Meurice, C. Nagy, and A. Cleve. Static analysis of dynamic database usage in java systems. In *Proc. of CAiSE 2016*, LNCS, pages 491–506. Springer, 2016.
[7] A. Ringlstetter, S. Scherzinger, and T. F. Bissyandé. Data model evolution using object-nosql mappers: Folklore or state-of-the-art? In *Proc. of BIGDSE 2016*, pages 33–36. ACM, 2016.
[8] S. Scherzinger, E. C. De Almeida, F. Ickert, and M. D. Del Fabro. On the necessity of model checking nosql database schemas when building saas applications. In *Proc. of TTC 2013*, pages 1–6. ACM, 2013.
[9] S. Scherzinger, T. Cerqueus, and E. C. d. Almeida. Controvol: A framework for controlled schema evolution in nosql application development. In *Proc. of ICDE 2015*, pages 1464–1467, 2015.