Enabling Efficient Agile Software Development of NoSQL-backed Applications

Uta Störl[1] Daniel Müller[2] Meike Klettke[3] Stefanie Scherzinger[4]

Abstract: NoSQL databases are popular in agile software development, where a frequently changing database schema imposes challenges for the production database. In this demo, we present *Darwin*, a middleware for systematic, tool-based support specifically designed for NoSQL database systems. *Darwin* carries out schema evolution and data migration tasks. To the best of our knowledge, *Darwin* is the first tool of its kind that supports both eager and lazy NoSQL data migration.

Keywords: NoSQL Databases, Schema Management

1 Introduction

In application development, software releases that also change the database schema are a common scenario [**?**, **?**]. Due to their schema-flexibility, NoSQL database systems are very popular in such setups, especially with agile software development. However, an outstanding challenge are situations where the structure of data already stored in the production database no longer matches what the latest application code expects. Such *legacy* data must be migrated. Today, many teams rely on custom-coded migration scripts, which is expensive in terms of person hours, as well as error-prone. What is missing is a tool-based support for an optional schema management in NoSQL database systems.

In [**?**], we discussed the need for a schema-management component capable of managing the schema, schema evolution, as well as data migration within NoSQL database systems. A first prototype called *Darwin*, designed according to these requirements, was sketched in [**?**]. Now, we present a live demo of a significantly enhanced and extended version of *Darwin*. The key contributions of this *Darwin* demo are:

- *Darwin* supports the complete schema management life cycle: Declaring an initial schema, repeatedly applying schema changes, and migrating legacy data.
- *Darwin* features different options for carrying out data migration tasks. Developers can choose a suitable approach for a given application development scenario.
- *Darwin* is implemented for different types of NoSQL database systems. A public interface makes it easy to integrate further NoSQL database products.

[1] Darmstadt University of Applied Sciences, uta.stoerl@h-da.de
[2] Darmstadt University of Applied Sciences, daniel.n.mueller@stud.h-da.de
[3] University of Rostock, meike.klettke@uni-rostock.de
[4] OTH Regensburg, stefanie.scherzinger@oth-regensburg.de

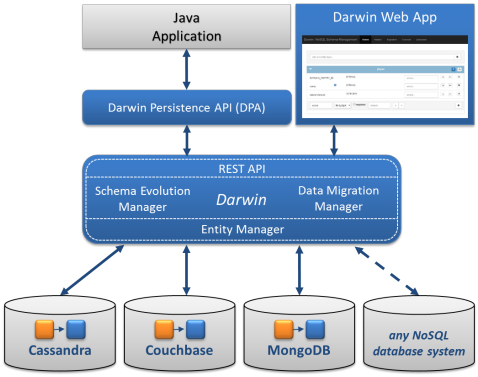

Fig. 1: The *Darwin* system architecture.

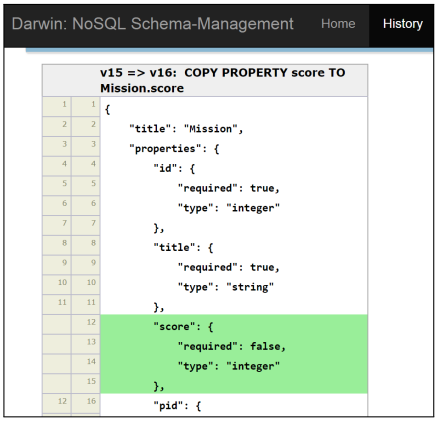

Fig. 2: *Darwin* screenshot: Schema history.

2 Supporting Schema Management with *Darwin*

Darwin operates as middleware between the applications and the NoSQL database systems (see Figure 1). Conceptually, Darwin can support any type of NoSQL database system. Currently, the document stores MongoDB and Couchbase, as well as the column family store Cassandra are supported. Darwin has a system-independent API (c.f. the *Entity Manager* in Figure 1) that makes it easy to integrate other NoSQL systems. We next lay out the schema management process and point out how *Darwin* supports each of its subtasks.

Initial Schema. We assume that an initial schema is available. As illustrated in Figure 3, there are different ways to obtain this schema:

- The schema can be *explicitly* declared by the developers.[5] The *Darwin Web App* (see Figure 1) offers two options, either using the *schema evolution language* introduced in [**?**], or a *graphical interface*.
- The schema may be *implicitly* derived from Object-NoSQL mappers such as Hibernate or Kundera, or from class declarations within the application code.
- The schema may be *extracted* from data persisted in the NoSQL database in a reverse engineering step. *Darwin* implements schema extraction as proposed in [**?**].

Schema Evolution. There are different strategies for declaring schema changes (illustrated in Figure 3). *Darwin* uses the schema evolution language first proposed in [**?**]. Upon request, *Darwin* can also visualize the schema history, as shown in Figure 2.

- The schema changes may be declared *explicitly*. Again, *Darwin* offers two options, using the schema evolution language or a graphical interface. Figure 4 shows an example of a *copy* operation, declared in the graphical interface. The property `score`

[5] An explicitly declared schema is no longer uncommon even for schema-flexible NoSQL databases. For instance, with recent versions of MongoDB, an optional schema may be registered [**?**].

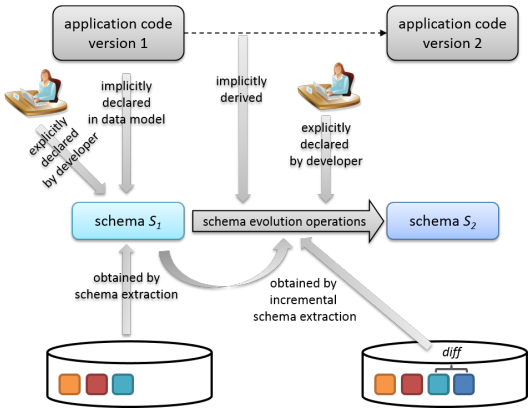

Fig. 3: The schema management process end-to-end.

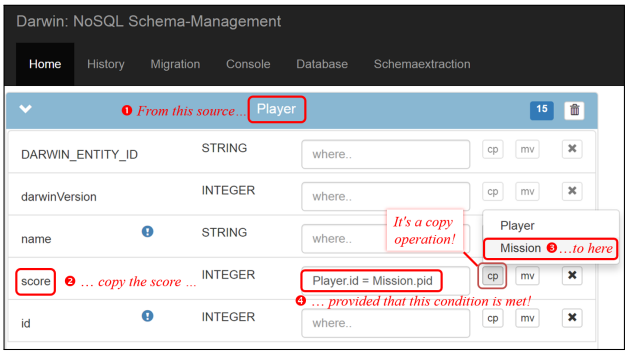

Fig. 4: Declaring a schema evolution operation using the graphical interface of *Darwin*.

is copied from each `Player` to the player's `Mission` entities. *Darwin* then generates the schema evolution operation accordingly:

```
copy Player.score to Mission where Player.id = Mission.pid.
```

- Another way is to *implicitly derive* schema evolution operations by analyzing changes to the application code. Addressing this task is scheduled for future work.
- The third possibility is to *incrementally* maintain a schema: An initial schema is extracted, and then maintained along with all updates to entities in the database. This approach is also implemented within *Darwin*, based on ideas proposed in [**?**].

Data Migration. *Darwin* further supports two data migration strategies [**?**]:

- *Darwin* implements *eager migration*, where the entire legacy data is migrated as a consequence of schema evolution. Eager migration can be explicitly initiated by users of *Darwin*, or it can be declared as the default behaviour of *Darwin*.
- *Darwin* also implements *lazy migration*, where single entities are only migrated on demand, when they are loaded by the application. This mechanism is triggered by calls from the application code to the *Darwin Persistence API*.

3 Demo Outline

Our demo shows how *Darwin* supports the schema management process end-to-end:

1. We start with schema extraction from synthetic gaming data persisted in the database.
2. Visitors of our demo may generate schema evolution operations using the graphical interface (Figure 4) or declare changes in our schema evolution language. We cover adding, removing, and renaming the properties of entities, as well as copying and moving properties between different kinds of entities.
3. Afterwards, we inspect the schema history, as visualized in Figure 2.
4. We can interactively assess the impact of *eager* or *lazy* migration on the data instance.

4 Outlook

Our next step is to implement additional migration strategies besides *eager* and *lazy* migration, to support a broad range of application needs [**?**]. We further work on detailed cost models for data migration, to help developers choose the most suitable approach.

Acknowledgements: We thank O. Haller, T. Landmann, T. Lehwalder, K. Möchel, H. Nkwinchu, and M. Richter from the Darmstadt University of Applied Sciences for implementation work on *Darwin*.