Curating Variational Data in Application Development

Uta Störl, Daniel Müller
Darmstadt University of Applied Sciences, Germany
uta.stoerl@h-da.de

Alexander Tekleab, Stephane Tolale, Julian Stenzel
Darmstadt University of Applied Sciences, Germany
bdcc.fbi@h-da.de

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

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

***Abstract*—Building applications for processing data lakes is a software engineering challenge. We present *Darwin*, a middleware for applications that operate on variational data. This concerns data with heterogeneous structure, usually stored within a schema-flexible NoSQL database. *Darwin* assists application developers in essential data and schema curation tasks: Upon request, *Darwin* extracts a schema description, discovers the history of schema versions, and proposes mappings between these versions. Users of *Darwin* may interactively choose which mappings are most realistic. *Darwin* is further capable of rewriting queries at runtime, to ensure that queries comply with legacy data. Alternatively, *Darwin* can migrate legacy data to reduce the structural heterogeneity. Using *Darwin*, developers may thus evolve their data in sync with their code. In our hands-on demo, we curate synthetic as well as real-life datasets.**

***Index Terms*—NoSQL databases; schema evolution; schema management; data migration; query rewriting; variational data**

I. Introduction

With data accumulating in data lakes, there is a rejuvenated interest in data governance for semi-structured and heterogeneous data. Recent research efforts on so-called *variational data* [1] include the extraction of a schema description [2], [3] or integrity constraints [4], or even handling multiple sets of schemas when several applications require access [5].

In this paper, we focus on a specific scenario within this context that agile developers are facing in their daily work, namely an application evolving along several versions. This brings about evolutionary changes in the structure of persisted data, often as simple as added fields, but also involving more complex changes.

For applications that need to be available 24/7, NoSQL databases can be an appealing architectural choice: The new version of the application can be deployed against the same database that has also been serving the predecessor versions, without application downtime due to database migration.

While the database now stores records that adhere to the structure expected by the latest version of the application, it also stores legacy records, created by some earlier version of the application. Even though the database itself may not manage any schema (as the schema is implicit in the application code), in its essence, this can be considered an instance of the problem of schema evolution.

In this paper, we introduce *Darwin*, a middleware for Java applications running against variational data, as stored in a NoSQL database. Our production installation of *Darwin* is running on 40 physical nodes with installations of Cassandra, Couchbase, and MongoDB. *Darwin* assists software developers in a range of vital data curation tasks:

1) Extracting a comprehensive schema description.
2) Provided that persisted records are timestamped, restoring a timeline of legacy schema versions. This reveals how variation was introduced in the data.
3) Proposing mappings between legacy schema versions, such as adding, removing, or renaming properties, as well as copying or moving properties. In interaction with the developers, *Darwin* restores the *schema evolution history*.
4) Given the restored schema evolution history, rewriting queries at application runtime, ensuring that queries also take into account legacy versions of the data.
5) Further, migrating legacy records to obtain a homogeneous data instance.

As a tool, *Darwin* enables application developers to focus on writing new application code, rather than throw-away code for curating variational data.[1]

II. Darwin System Overview

Figure 1 shows the system architecture of *Darwin*. *Darwin* is a middleware between a Java application and a database storing variational data:

- To the top of the application stack is the Java application. It stores its data in a NoSQL database, interacting with the system-independent *Darwin Persistence API*.
- Via the *Darwin WebApp*, application developers may trigger data curation tasks directly.
- The *Darwin Core REST API* interfaces with the modules for schema extraction, handling schema evolution, data migration, and query rewriting.

We next describe the interplay of the components of *Darwin* along the workflow sketched in Figure 2.

[1]While we have demoed an earlier version of *Darwin* [6], features (2) through (4) above are new material. In particular, features (2) and (3) are now partly automated, as opposed to being manually specified by the users.

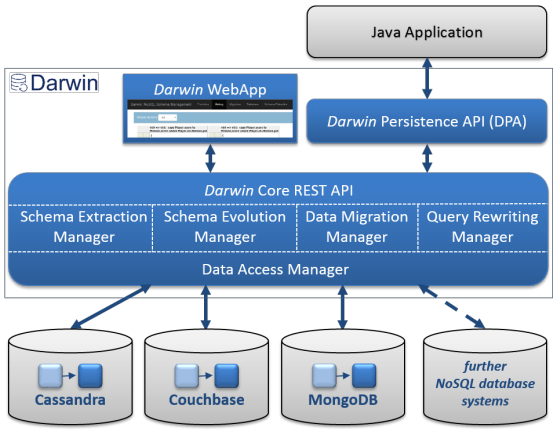

Fig. 1. The *Darwin* system architecture.

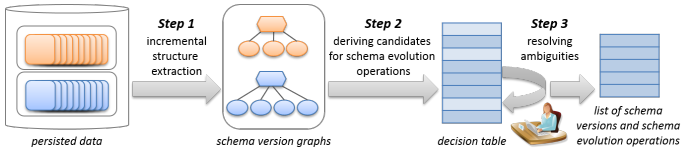

Fig. 2. The interactive workflow for restoring the schema evolution history.

Step 1: The *Schema Extraction Manager* extracts *schema version graphs* as comprehensive schema descriptions from the data instance. It considers the complete data instance when incrementally constructing an internal representation. We merely sketch the basic idea using the toy data from Figure 3, and refer to [7] for the details.

In extracting a schema, we assume that we are able to identify which entities are instances of the same class of the object-oriented application code. Depending on the NoSQL data store and the overall software stack (e.g., the usage of an object-NoSQL mapper), these *entity types* may be encoded in different ways. For instance, a designated property can specify the class identifier, or entities in the same collection or table can belong to the same class. Figure 3 shows entities of type `Player` and `Mission` in JSON format. Each entity carries a unique id as well as a timestamp. The `Mission` property `pid` references the `Player` currently pursuing this mission. Among `Missions`, only the most recently persisted `Mission` has a property `score`.

Figure 4 shows the derived schema version graphs. Each root node is labelled with the entity type and lists the timestamps of all entities of this entity type. The graph further contains a node for each property, recording data type and hierarchy information, as well as the timestamps of all entities carrying this property. For instance, the node for `Mission` property `title` has the timestamp list `[10, 14]`, whereas the node for `score` has the timestamp list `[14]`.

Step 2: The *Schema Evolution Manager* analyzes the timeline inherent in the schema version graphs. It recognizes schema versions and proposes schema evolution operations as mappings between succeeding schema versions. We refer to our companion paper [7] for details.

Darwin is able to derive single-type operations (such as *add*, *remove*, and *rename*) affecting the properties of entities of the *same* entity type. Comparing entities of *different* entity types, *Darwin* can detect multi-type operations for copying and moving properties between entity types. We have introduced these operations in earlier work [8]. *Darwin* pre-formulates a join condition for *copy* or *move* operations, and leaves it to the user to specify the join predicate, as seen in Figure 5.

There may be alternative schema evolution operations mapping between schema versions. In our example, we detect a new property `score` in entity type `Mission` at timestamp 14. This change may be described by an *add* operation, or a *copy* operation from entity type `Player`, joining on the player id. *Darwin* therefore compiles a *decision table*, as shown in Figure 5, listing alternatives for the software developers to choose from, as discussed next. After all, they have the necessary domain knowledge to resolve ambiguities, e.g., remembering how the software has evolved in the past.

Step 3: The software developers now resolve ambiguities in the decision table. This yields a sequence of schema evolution operations, describing the schema evolution history and thus the lineage of the variational data.

Darwin visualizes this history in Figure 6: The JSON schemas for `Mission` are shown in their two latest versions. The changes w.r.t. the previous schema version are marked up by a shaded background, e.g., `Mission` has a newly added property `score` in the version shown to the right.

Taming variational data in a data lake is a big challenge with dynamic requirements and heterogeneous applications. The comprehensive view on the schema evolution history may gain valuables insights for the developers already at this stage.

Notes on Performance and Scalability: Extracting and analyzing the entire data instance is a one-time effort. After the initial schema extraction, newly added entities can be analyzed incrementally and on-the-fly.

In extracting schema version graphs and deriving candidate mappings between schema versions, *Darwin* also proceeds incrementally. Since *Darwin* does not load the entire data instance into main memory, but only incremental batches [7], *Darwin* may safely handle large volumes of data. Our production installation of *Darwin* is running on the *Big Data Cluster* at *Darmstadt University of Applied Sciences*, comprising 40 physical nodes with installations of the NoSQL databases Cassandra, Couchbase, and MongoDB.

III. Leveraging the Schema Evolution History

Having restored the schema evolution history, the software developers may pursue two different strategies in curating variational data with *Darwin*:

1) Leave the variational data as it is (e.g., due to compliance reasons), and have *Darwin* rewrite queries at runtime to account for the structural heterogeneity of the data.
2) Have *Darwin* migrate the variational data to a single, homogeneous structure.

Both strategies allow the application developers to code against the latest schema, without having to worry about variational

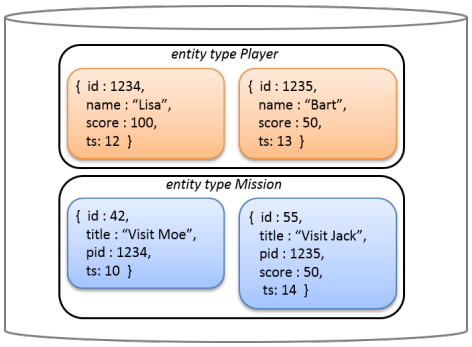

Fig. 3. Example: Variational data in a gaming application.

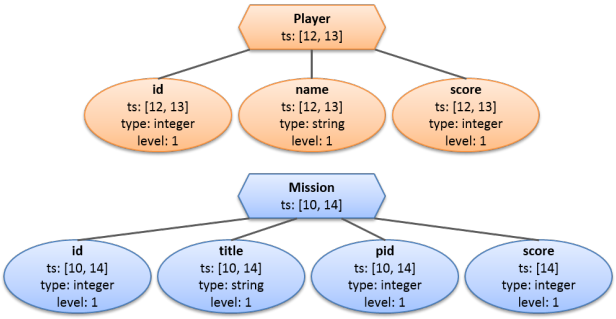

Fig. 4. Example: Schema version graphs for `Player` and `Mission`.

data. We briefly sketch these approaches.

Query Rewriting: With the first approach, the database stores variational data, yet the application developers write code as if all records adhere to a global (virtual) schema. Provided that queries are simple enough, queries can be automatically rewritten by *Darwin* such that they account for the structural variations in legacy data. To this end, we have adapted the approach from [9], based on the Chase algorithm, for selection and projection queries.

Let us illustrate this with a simple example. The application issues a query that assumes the latest schema, where all `Mission` entities carry a `score` property:

```
select title from Mission
where score > 10
```

Darwin has detected that starting with timestamp 14, score properties are copies from the `Player` owning the mission. The query is now automatically rewritten:[2]

```
(select title from Mission
 where score > 10 and ts >= 14)
union
(select title from Mission M, Player P
 where P.score > 10 and P.id = M.pid and M.ts < 14)
```

Data Migration: Alternatively, developers may migrate the data to adhere to a single schema: Since *Darwin* is aware of the historical schema versions, as well as the sequence of mappings, *Darwin* can reliably carry out the required steps.

IV. Demonstration Scenario

Our demo follows the workflow outlined in Figure 2 and walks through the steps that software developers may take from there, as sketched in Section III. We have prepared several datasets containing versioned data to illustrate different challenges. Besides the synthetic *gaming data* used in this paper, we also present biological data with observations of species in the Baltic sea [7], and real world configuration data from the Wendelstein 7-X plasma experiments [2], [10].

Restoring the Schema Evolution History: First, we select and register the data source in the *Darwin WebApp*. We let *Darwin* extract the schema versions and propose candidate evolution operations. The developers then interactively resolve any ambiguities. Figure 5 shows a screenshot of *Darwin* with the alternative schema evolution operations *add* and *copy* from our running example. The user of the *Darwin WebApp* is editing the join condition for the *copy* operation.

[2]We show a simplified query: Rather than selecting on timestamps, *Darwin* filters on a *Darwin*-internal version property. This property is annotated by *Darwin* during reconstruction of the schema evolution history.

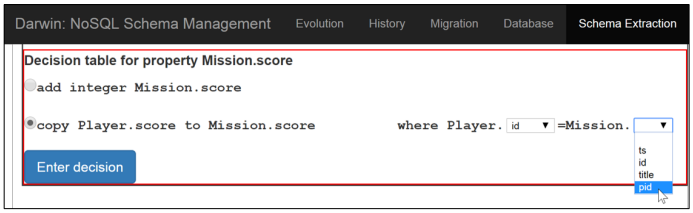

Fig. 5. *Darwin* screenshot: Software developers interactively resolve ambiguities via the decision table and edit the join condition for a *copy* operation.

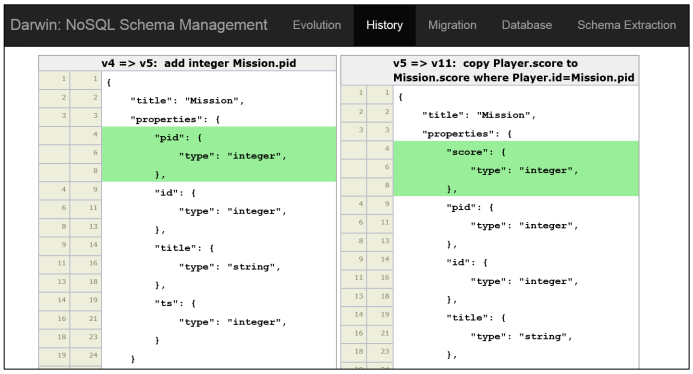

Fig. 6. *Darwin* screenshot: Showing two versions of `Mission` side by side, in JSON schema notation. The schema evolution operations are stated above. Changes w.r.t. the previous schema version are highlighted.

Visualizing the Schema Evolution History: Now we can browse the schema evolution history, as seen in Figure 6.

Data Inspection: We may also sample data from the database, to examine its structural heterogeneity. Figure 7 shows `Mission` entities in versions `5` and `11`.

Query Rewriting: In the *Darwin WebApp*, we issue ad-hoc queries that assume that the entire data instance adheres to the latest schema. For instance, we assume that `Missions` carry a property `score`. We inspect the rewritten queries as well as the results of evaluating them on the variational data.

Data Migration: Darwin can carry out data migration to reconcile all entities with the schema expected by the latest

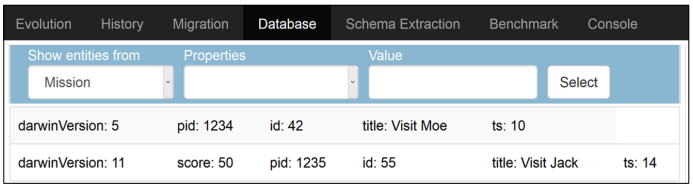

Fig. 7. *Darwin* screenshot: Sampled entities of type `Mission` with heterogeneous structure.

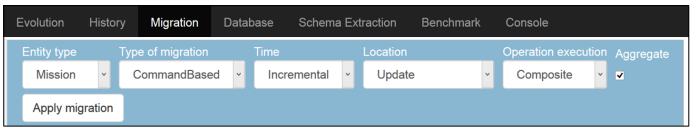

Fig. 8. *Darwin* screenshot: Configuring data migration upon a few clicks, rather than having developers write low-level data curation scripts.

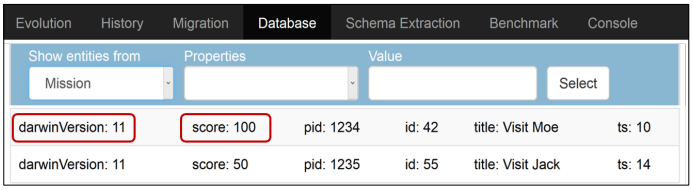

Fig. 9. *Darwin* screenshot: After data migration, the sampled entities of type `Mission` no longer display structural variations.

application release. Figure 8 shows the interactive GUI for configuring a data migration task. As we can see in Figure 9, as a result of migration, all `Mission` entities shown are in the latest version `11`. We have highlighted the changed properties. In particular, the `Mission` entity with `id = 42` now has a `score` property, copied from its `Player`.

V. Recent Related Work

There is a large body of related work, and given the page limitations, we cherry pick from the most recent contributions.

Extracting a schema and constraints from unstructured data (in particular, XML) has been studied intensively. For handling *big data*, the need for highly scalable solutions has inspired new contributions e.g. [2]–[4], [11], [12].

Handling versions of a schema, even for several applications requesting access, has been recently proclaimed a highly relevant research area [5]. Similarly, the authors in [1] stress the need for dedicated databases to handle variational data. With *Darwin*, we do not pitch a new kind of database system, but instead a powerful middleware for application developers to use. This leaves it up to the developers to choose among the *Darwin*-supported off-the-shelf NoSQL database products (currently Cassandra, Couchbase, and MongoDB).

The authors in [13] also address variational data, yet within the relational model. Their strategy is to migrate data back and forth between schema versions (rather than performing traditional Chase-based query rewriting), allowing both reads and even writes to data in legacy schemas.

One of the prominent systems tackling schema evolution in *relational* databases by query rewriting has been made to work by [9]. Our approach leans on these ideas designed for relational databases, and extends them to handling hierarchical data. We plan to publish the details in a future paper.

While we find several connections to related work (especially given the long-standing history of schema evolution research [14]), the workflow presented in this demo is novel: *Darwin* is the first system to implement this end-to-end support for curating variational data in agile application development.

VI. Conclusion

We presented *Darwin*, a middleware that assists application developers in curating variational data. Using *Darwin*, developers can not only extract a schema, but also reconstruct a plausible schema evolution history. This already can contribute new insights. The developers may further choose to have *Darwin* rewrite queries at run-time, to account for variational data, or they may homogenize their data instance.

This kind of tool support frees up time, so that the developers may focus on implementing new application features, rather than writing low-level data curation scripts. Tools like *Darwin* can thus substantially simplify the long-term maintenance of variational data gathering in data lakes.

Acknowledgements

This project was funded by the *Deutsche Forschungsgemeinschaft* (DFG, German Research Foundation), grant #385808805.

We thank O. Haller, T. Landmann, T. Lehwalder, K. Möchel, H. Nkwinchu, M. Richter, and M. Shenavai from *Darmstadt University of Applied Sciences* for contributing to the *Darwin* code base.

References

[1] P. Ataei, A. Termehchy, and E. Walkingshaw, "Variational Databases," in *Proc. DBPL'17*, 2017.

[2] M. Klettke, U. Störl, and S. Scherzinger, "Schema Extraction and Structural Outlier Detection for JSON-based NoSQL Data Stores," in *Proc. BTW'15*, 2015.

[3] M.-A. Baazizi, G. G. Dario Colazzo, and C. Sartiani, "Counting Types for Massive JSON Datasets," in *Proc. DBPL'17*, 2017.

[4] M. Farid, A. Roatis, I. F. Ilyas, H.-F. Hoffmann, and X. Chu, "CLAMS: Bringing Quality to Data Lakes," in *Proc. SIGMOD '16*, 2016.

[5] B. Chandramouli, J. Gehrke, J. Goldstein, D. Kossmann, J. J. Levandoski, R. Marroquin, and W. Xie, "READY: Completeness is in the Eye of the Beholder," in *Proc. CIDR'17*, 2017.

[6] U. Störl, D. Müller, M. Klettke, and S. Scherzinger, "Enabling Efficient Agile Software Development of NoSQL-backed Applications," in *Proc. BTW'17*, 2017.

[7] M. Klettke, H. Awolin, U. Störl, D. Müller, and S. Scherzinger, "Uncovering the Evolution History of Data Lakes," in *Proc. SCDM'17*, 2017.

[8] S. Scherzinger, U. Störl, and M. Klettke, "Managing Schema Evolution in NoSQL Data Stores," in *Proc. DBPL'13*, 2013.

[9] H. J. Moon, C. A. Curino, and C. Zaniolo, "Scalable Architecture and Query Optimization for Transaction-time DBs with Evolving Schemas," in *Proc. SIGMOD'10*, 2010.

[10] A. Spring, M. Lewerentz, T. Bluhm, P. Heimann, C. Hennig *et al.*, "A W7-X experiment program editor - A usage driven development," in *Proc. 8th IAEA Technical Meeting on Control, Data Acquisition, and Remote Participation for Fusion Research*, 2012.

[11] S. Cebiric, F. Goasdoué, and I. Manolescu, "Query-oriented summarization of RDF graphs," *PVLDB*, vol. 8, no. 12, 2015.

[12] L. Wang, S. Zhang, J. Shi, L. Jiao *et al.*, "Schema Management for Document Stores," *Proc. VLDB Endow.*, vol. 8, no. 9, May 2015.

[13] K. Herrmann, H. Voigt, J. Rausch, A. Behrend, and W. Lehner, "Living in Parallel Realities – Co-Existing Schema Versions with a Bidirectional Database Evolution Language," in *Proc. SIGMOD'17*, 5 2017.

[14] J. F. Roddick, "A survey of schema versioning issues for database systems," *Information & Software Technology*, vol. 37, no. 7, 1995.