Dieses Dokument ist eine Zweitveröffentlichung (Postprint Version) /

This is a self-archiving document (accepted version):

Kai Herrmann, Hannes Voigt, Thorsten Seyschab, Wolfgang Lehner

InVerDa - co-existing schema versions made foolproof

Erstveröffentlichung in / First published in:

International Conference on Data Engineering. Helsinki, 16. – 20.05.2016. IEEE Xplore, S. 1362 – 1365. ISBN 978-1-5090-2020-1.

DOI: https://doi.org/10.1109/ICDE.2016.7498345

Diese Version ist verfügbar / This version is available on:

https://nbn-resolving.org/urn:nbn:de:bsz:14-qucosa2-752855

SLUB Wir führen Wissen.

TECHNISCHE UNIVERSITÄT DRESDEN

QUCOSA Quality Content of Saxony

Final edited form was published in "International Conference on Data Engineering. Helsinki 2016", S. 1362 - 1365. ISBN: 978-1-5090-2020-1
https://doi.org/10.1109/ICDE.2016.7498345

InVerDa – Co-existing Schema Versions Made Foolproof

Kai Herrmann, Hannes Voigt, Thorsten Seyschab and Wolfgang Lehner
Database Technology Group
Technische Universität Dresden, Germany
Email: <firstname>.<lastname>@tu-dresden.de

***Abstract*—In modern software landscapes multiple applications usually share one database as their single point of truth. All these applications will evolve over time by their very nature. Often former versions need to stay available, so database developers find themselves maintaining co-existing schema version of multiple applications in multiple versions. This is highly error-prone and accounts for significant costs in software projects, as developers realize the translation of data accesses between schema versions with hand-written delta code. In this demo, we showcase InVerDa, a tool for integrated, robust, and easy to use database versioning. We rethink the way of specifying the evolution to new schema versions. Using the richer semantics of a descriptive database evolution language, we generate all required artifacts automatically and make database versioning foolproof.**

I. Introduction

Relational database management systems (DBMS) lack proper support for co-existing schema versions within the same database. With today's realities in information system development and deployment – namely agile development methods, short release cycles, customization, stepwise deployment and migration, large numbers of independent stakeholders, varying update adoption time, etc. – such support becomes increasingly desirable. Code versioning in GIT or SVN and build systems such as Maven make it fairly easy to maintain multiple versions of the same application and deploy and run various application versions concurrently. The same is hard for a database, though. Current DBMSs force developers to migrate a database completely in one haul to a new schema version. Keeping other schema versions alive before and after such a migration typically requires manually written and maintained delta code either in the database (views and triggers) or in the application. This makes the handling of co-existing schema versions very costly and error-prone. Valuable tools, such as Liquibase[1], help to manage schema versions outside the DBMS. The remarkable and inspiring research work Prima[1] adds delayed propagation of data from an old to a newer version, which enables forward propagation of updates and backward propagation of reads. However, to our best knowledge, there is no full-fledged support for co-existing schema versions, including the propagation of data from new schema versions back to older ones for both data accesses and migrations. In consequence developers at the database end see themselves forced into acting against their realities – forced into less agile development, longer release cycle, more risky big-bang migration, etc. – to organize co-existing schema versions and reduce the overhead and bug potential attached to them.

[1]http://www.liquibase.org/

Database versioning provides a solution out of that dilemma. With database versioning, the DBMS allows a single database to have multiple co-existing schema versions. Developers can evolve an existing database schema to add a new schema version to the database. The DBMS makes the database available through all co-existing schema versions. Data can be read and written through all schema versions; writes in one version are reflected in all other versions. On the administration end, the database administrator can easily configure in which schema version the data is primarily materialized. The DBMS transparently decouples the logical availability of a schema version from the physical migration of the data without the need of any handwritten delta code.

In this demo, we present InVerDa. InVerDa adds database versioning functionality to a DBMS. To specify the evolution from one schema version to another, InVerDa provides a declarative Database Evolution Language (DEL) called InDEL. InDEL is very similar to established DELs [2], [3], with the distinction that its Schema Modification Operators (SMOs) are specifically designed to be invertible. Based on an InDEL-specified schema evolution, InVerDa is able to generate all required delta code in the DBMS – particularly views and triggers – to make the database instantly available through the new schema version. InVerDa also offers a foolproof migration statement that allows changing the actual materialization of the data to another schema version without a single line of SQL. Upon such a migration statement, InVerDa migrates the data and regenerates all necessary delta code to reflect the new materialization and keep all existing schema versions fully available. InVerDa decouples the logical schema versions from the physical materializing schema, and hence allows to freely adapt the materialization to the workload. The contributions of InVerDa are

- **Invertible database evolution language** InDEL is an evolution language with invertible SMOs. InDEL SMOs allow establishing bi-directional translations between schema versions, which is the basis for database versioning.
- **Co-existing schema versions** InVerDa automatically generates delta code to continuously support read and write operations on all co-existing schema versions. All schema versions provide an individual view on the same shared dataset.
- **Foolproof migration** InVerDa makes manual migrations obsolete by generating migration scripts from InDEL scripts and executing them on the database. The generated migration scripts include the physical data movement as well as the adaptation of all involved delta code.

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "International Conference on Data Engineering. Helsinki 2016", S. 1362 - 1365. ISBN: 978-1-5090-2020-1
https://doi.org/10.1109/ICDE.2016.7498345

The aim of this demo is to showcase the benefits and convenience of database versioning with InVerDa, i.e., of the proper support for co-existing schema versions within the same database. During the demo, participants can practically experience InDel by writing own evolution declarations and creating new schema versions with the comfortable InVerDa Console. The InVerDa Explorer allows the participants to easily inspect the different schema versions existing for a database. Finally, participants can trigger migration statements to directly experience the simplicity of physical data migration with InVerDa as well as how migration is transparently decoupled from the availability of schema versions.

In the remainder of the demo proposal we introduce InVerDa using a comprehensible example (Section II), elaborate on chosen technical aspects (Section III), and outline demo details (Section IV).

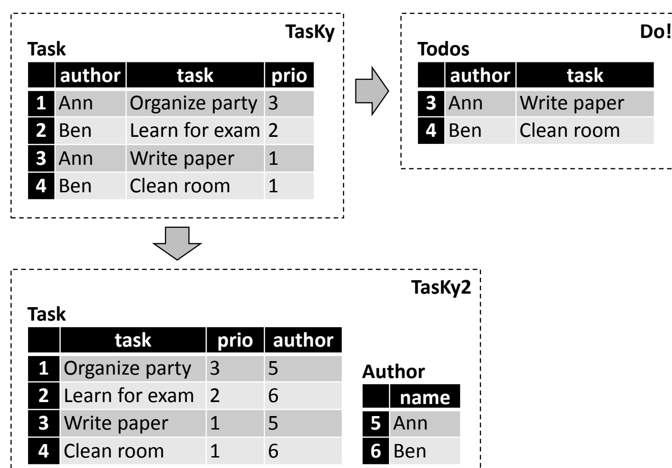

Fig. 1. Exemplary database evolution.

II. InVerDa – User Perspective

In the following, we take the perspective of a database developer. Let us assume we are the developer of a simple task management system called *TasKy*, which users can install on their desktops and which is backed by a central database. *TasKy* allows users to create new tasks, list, update, and delete them. Each task has an author and a priority ranging from 1 to 3 with 1 being most urgent. The first release *TasKy* stores all its data in a single table `Task(author,task,prio)`. *TasKy* has productive go-live and users begin to feed the database with their tasks as shown in Figure 1.

A. Creating New Schema Versions

TasKy gets widely accepted and after some weeks we collect some user feedback to discover that users like to have their most urgent tasks listed on their phone. To quickly respond to this demand we incorporate a third party phone app called *Do!*. However *Do!* expects a different database schema than *TasKy* is using. The *Do!* schema consists of a table `Todos(author,task)` containing only tasks of priority 1. Obviously, the initial schema version needs to stay alive for *TasKy*, which is broadly installed. Traditionally, we would use a view to create an external schema fitting *Do!*. Since views are not necessarily updatable this likely also includes writing triggers for the propagation of writes in *Do!* back to what *TasKy* sees in the database. InVerDa greatly simplifies this process and handles all the necessary delta code for us. We merely have to execute the following InDel evolution script with InVerDa to create the new schema version called *Do!*:

```
EVOLUTION START FROM 'TasKy';
PARTITION TABLE Task INTO Todos WITH prio=1;
DROP COLUMN prio FROM Todos DEFAULT 1;
EVOLUTION COMMIT AS 'Do!';
```

The script instructs InVerDa to derive schema *Do!* from schema *TasKy* by creating a horizontal partition of `Task` with `prio=1` and dropping the priority column. Executing the script, immediately creates a new schema including the view `Todos` as well as triggers for write propagation. When a user inserts a new todo to the `Todos` view, the triggers will automatically insert an corresponding task with priority 1 to `Task` in *TasKy* instead. Equally, updates and deletes are propagated back to the *TasKy* schema. Hence, the *TasKy* data is immediately available to be read and written through the newly incorporated *Do!* app by simply executing the above InDel evolution script. At this point InVerDa has already simplified our job significantly.

We continue to improve and extend *TasKy* to make it a more useful tool for its users. While adding functionality, we also refactor the code every now and then. For the next release *TasKy2*, we decide to normalize the table `Task` into `Task` and `Author`. Since we plan a stepwise roll-out of *TasKy2*, the old schema of *TasKy* has to remain alive until all clients have been updated. Again, we use InVerDa to simplify the task. With the InDel evolution script

```
EVOLUTION START FROM 'TasKy';
DECOMPOSE TABLE Task INTO Task(task,prio)
                      AND Author(author)
                      ON FK author;
RENAME COLUMN author IN Author TO name;
EVOLUTION COMMIT AS 'TasKy2';
```

InVerDa creates the schema version *TasKy2* and decomposes the table `Task` to separate the tasks from their authors, as shown in Figure 1. InVerDa supports three kinds of decomposition. It can eliminate duplicates in both resulting tables, only in the second table, or in neither of them, which fits a many-to-many, one-to-many, and one-to-one relationship, respectively. In case of *TasKy2*, we decide for the second option, which is denoted in InDel by **ON FK** *fk*, where *fk* names the foreign key column in the first table. Additionally, we state in the evolution script that column `author`, which contains the author's name after the decomposition should be renamed to `name`. Based on the script, InVerDa generates delta code (views and triggers) for `Task` and `Author` to make the *TasKy2* schema immediately available. The delta code ensures that all write operations to any of the three schema versions are propagated to all other schema versions. Assume user Ann has already upgraded to *TasKy2* and changes the priority of *Organize party* to 1, then this task will immediately occur in the *Do!* app on her phone. After the party, Ann deletes this todo using *Do!*, which also removes this task from the other schema versions.

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "International Conference on Data Engineering. Helsinki 2016", S. 1362 - 1365. ISBN: 978-1-5090-2020-1
https://doi.org/10.1109/ICDE.2016.7498345

- **CREATE TABLE** R(c₁, ..., cₙ)
- **DROP TABLE** R
- **RENAME TABLE** R **INTO** R'
- **RENAME COLUMN** c **IN** R_i **TO** c'
- **ADD COLUMN** c **AS** $f(c_1, \ldots, c_n)$ **INTO** R_i
- **DROP COLUMN** c **FROM** R_i **DEFAULT** $f(c_1, \ldots, c_n)$
- **DECOMPOSE TABLE** R
  INTO $S(s_1, \ldots, s_n)$
  [, $T(t_1, \ldots, t_m)$ **ON** (**PK**|**FK** fk|$cond$)]
- [**OUTER**] **JOIN TABLE** R, S **INTO** T $cond$
- **PARTITION TABLE** R **INTO** S **WITH** $cond_S$
  [, T **WITH** $cond_T$]
- **MERGE TABLE** R ($cond_R$), S ($cond_S$) **INTO** T

Fig. 2. INDEL Syntax.

B. Migrating Data

The initial materializing schema of the stored data is determined by the initial schema. All other schema versions are implemented with the help of delta code. However, the initial schema is likely not the optimal internal structure. The delta code introduces an overhead on read and write accesses to new schema versions. The more SMOs are between schema versions, the more delta code is involved, the higher is the overhead.

In case of our task management system, the schema versions *TasKy2* and *Do!* have delta code towards the initial schema version *TasKy*. Assume, some weeks after releasing *TasKy2* the majority of the users have upgraded to the new version. *TasKy2* comes with its own phone app, so that the schema *TasKy* and *Do!* are still accessed but merely by a minority of users. Hence, it seems appropriate to migrate data physically to the *TasKy2* schema, now.

Traditionally, we would have to write a migration script, which moves data, and implements new delta code. All that would accumulate to some hundred lines of code, which we would have to test intensively to prevent from messing up our data. With INVERDA, we achieve the same blindfold with the one-liner:

```
MATERIALIZATION 'TasKy2';
```

Upon this statement, INVERDA transparently runs the physical data migration to schema *TasKy2*, maintaining the well-established transaction guarantees, and updates the involved delta code of all schema versions. We do not need to perform any further manual adjustments. All schema versions stay available; read and write operations are merely propagated to a different materializing schema, now.

III. INVERDA – SYSTEM PERSPECTIVE

With INVERDA developers specify the evolution from one schema version to another with a DEL. Existing DELs describe the forward evolution from schema version n to schema version $n+1$. Database versioning with co-existing schemas also requires backward evolution to propagate data manipulations from the schema version $n+1$ back to schema version n.

INVERDA's INDEL combines standard SMOs for forward evolution with strategies to fill missing information and resolve ambiguity occurring in backward evolution. INDEL builds on CODEL [3], a relationally complete database evolution language with precisely defined syntax and semantics. Because of its completeness property, CODEL basically has for every SMO another SMO that achieves the opposite kind of evolution. To account for backward evolution, INDEL essentially adds to each SMO the arguments of its opposite SMO. This provides INVERDA the required information to propagate data forward as well as backward.

As a simple example, consider dropping a column c from source schema version S to derive the target schema version T. In case the user inserts data in the new schema version T, the delta code for backward propagation to S has to provide a value for missing column c. The opposite add-column SMO provides a value for the new column during forward propagation based on a value generator function given by the user. Accordingly, INDEL's drop-column SMO also features a user-given value generator function to enable backward propagation. Figure 2 summarizes the INDEL SMOs. As can be seen, INDEL is of similar complexity and hence similar usability as established DELs.

However, guaranteeing correct forward and backward propagation from one version to another is not trivial. Since many SMOs are not information-preserving, INVERDA manages additional auxiliary tables. For instance, when adding a column, INVERDA stores the data of this column in an auxiliary table next to the original table until this new schema version is materialized.

INVERDA defines each SMO in two templates of Datalog rules for the forward propagation $S \rightarrow T$ (read: get_T / write: put_S) and for the backward propagation $S \leftarrow T$ (read: get_S / write: put_T), respectively. As example, say we evolve schema S to T by dropping column c from table R with the primary key p and the remaining columns A. Let R' be the resulting table in the target schema T. The auxiliary table C stores the data of the dropped column c and relates it to the remaining data with the primary key p. The corresponding Datalog templates are:

$$get_T/put_S: \quad R'(p,A) \leftarrow R(p,A,_) \quad C(p,c) \leftarrow R(p,_,c)$$

$$get_S/put_T: \quad R(p,A,c) \leftarrow R'(p,A), C(p,c) \quad R(p,A,c) \leftarrow R'(p,A), \neg C(p,_), c = f(A)$$

An INDEL SMO is correct if sequential evaluation of the Datalog rules keeps any initial payload data unchanged. In other words, after a propagation round trip from schema version S to T and back to S (and also from T to S and back to T), the initial payload data is still correct and complete. Formally, for every INDEL SMO the two following conditions hold:

$$Data = get_S(put_S(Data))$$
$$Data = get_T(put_T(Data))$$

To execute an SMO, INVERDA instantiates the corresponding Datalog templates based on the arguments provided by the user. From the instantiated rules INVERDA generates the necessary delta code – views for reads (get_X) and triggers for writes (put_X). Migration scripts are generated in similar manner.

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "International Conference on Data Engineering. Helsinki 2016", S. 1362 - 1365. ISBN: 978-1-5090-2020-1
https://doi.org/10.1109/ICDE.2016.7498345

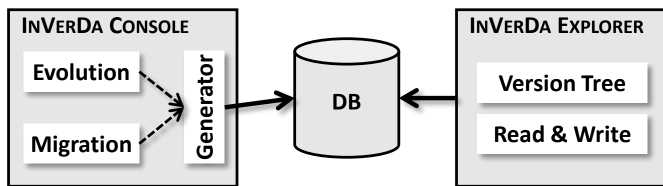

Fig. 3. InVerDa tooling.

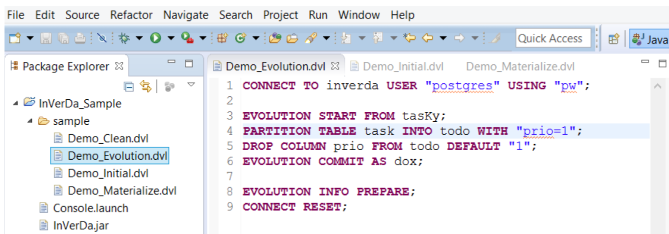

Fig. 4. InVerDa Console with InDel editor evolving *TasKy* to Do!.

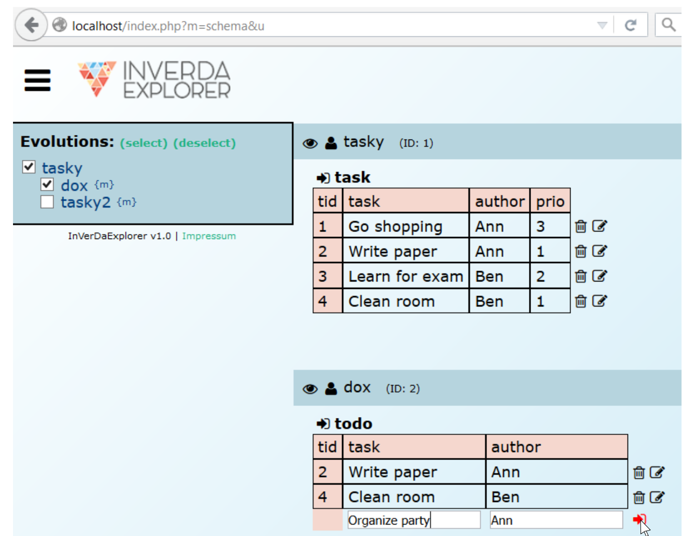

Fig. 5. InVerDa Explorer with *TasKy* and Do!.

IV. Demonstration

For the sake of demonstration InVerDa consist of two parts as illustrated in Figure 3. The first part, the InVerDa Console allows to execute InDel evolution scripts and migration statements; it is where the InVerDa functionality is implemented. The screenshot in Figure 4 shows the InVerDa Console with the InDel script for the evolution from *TasKy* to *Do!*. When executing such a script, InVerDa creates the respective schema within a given database. Applications can now access the schema by standard database connectivity without InVerDa in the loop.

The second part, the InVerDa Explorer conveniently allows browsing all schema versions existing in a database. To demonstrate the effect of the InVerDa-generated delta code, the InVerDa Explorer offers the possibility to manipulate data in any of the schema versions and observe the effect on the data in the other schema versions. Figure 5 shows the InVerDa Explorer with the schema versions *TasKy* and *Do!* and the insertion of new data in Do!.

During the demo, we will have a prepared example database with schema evolutions. For example, we will execute InDel evolution scripts with the InVerDa Console and demonstrate the genuinely co-existing schema versions by manipulating data with the InVerDa Explorer. However, the demo is by no means limited to the prepared example. InVerDa is a working system and we encourage participants to try it interactively. Participants can experience both the developer perspective by writing own InDel evolution scripts and issuing migration statements, as well as the user perspective by probing the delta code generated by InVerDa with own data manipulations. The InVerDa Console and InVerDa Explorer have simple GUIs to be mastered in a minute by every participant.

The aim of the demo is twofold. First, we want to show the benefits of proper support for co-existing schema versions as provided by InVerDa. Specifically, participants can experience the simplicity and power of InDel and InVerDa. First and foremost, however, participants can feel the freedom that transparent decoupling of the logical availability of a schema version from the physical migration of the data brings to data management.

Second, we want to stimulate discussions. Particularly, we like to discuss with practitioners about use cases of schema versioning, practical applicability, missing aspects, etc.. With researchers we are happy to exchange views on technical details of the solution as well as future research questions. On a more general level, we hope to promote our idea of a more comprehensive DBMS support for schema evolution and co-existing schema versions.

InVerDa is a first step towards such a comprehensive support for co-existing schema versions. With InVerDa multiple applications in multiple versions can share one database as a single point of truth, each having an individual view on the data. Traditional management of multiple schema versions requires the costly implementation and maintenance of delta code and migration scripts. We show with InDel that it is possible to use the semantics of a data evolution language to (1) generate delta code and migration scripts and (2) transparently decouple the logical availability of a schema version from the physical migration of the data.

Acknowledgment

This work is funded by the German Research Foundation (DFG) within the Research Training Group RoSI (GRK 1907).

References

[1] H. J. Moon, C. Curino, M. Ham, and C. Zaniolo, "PRIMA - archiving and querying historical data with evolving schemas," in *SIGMOD Conference, Providence, USA*. ACM Press, 2009, pp. 1019–1022. [Online]. Available: http://dl.acm.org/citation.cfm?id=1559845.1559970

[2] C. Curino, H. J. Moon, A. Deutsch, and C. Zaniolo, "Automating the database schema evolution process," *VLDB Journal*, vol. 22, no. 1, pp. 73–98, 2013. [Online]. Available: http://link.springer.com/10.1007/s00778-012-0302-x

[3] K. Herrmann, H. Voigt, A. Behrend, and W. Lehner, "CoDEL - A Relationally Complete Language for Database Evolution," in *ADBIS 2015, Poitiers, France*, ser. Lecture Notes in Computer Science, vol. 9282. Springer, 2015, pp. 63–76. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-23135-8_5

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden