CoDEL – A Relationally Complete Language for Database Evolution

Kai Herrmann[#], Hannes Voigt[#], Andreas Behrend[*], and Wolfgang Lehner[#]

[#]Database Technology Group, Technische Universität Dresden, Germany,
firstname.lastname@tu-dresden.de
[*]Computer Science III, University of Bonn, Germany,
behrend@cs.uni-bonn.de

Abstract. Software developers adapt to the fast-moving nature of software systems with agile development techniques. However, database developers lack the tools and concepts to keep pace. Data, already existing in a running product, needs to be evolved accordingly, usually by manually written SQL scripts. A promising approach in database research is to use a declarative database evolution language, which couples both schema and data evolution into intuitive operations. Existing database evolution languages focus on usability but did not aim for completeness. However, this is an inevitable prerequisite for reasonable database evolution to avoid complex and error-prone workarounds. We argue that relational completeness is the feasible expressiveness for a database evolution language. Building upon an existing language, we introduce CoDEL. We define its semantic using relational algebra, propose a syntax, and show its relational completeness.

Keywords: Descriptive Database Evolution, Evolution Language, Relational Completeness

1 Introduction

Changes in modern software systems are no longer an exception but have become daily business. Following the mantra "Evolution instead of Revolution", agile software development centers the creativity and excellence of people to handle the unpredictably dynamic world of software development [3]. Agile methods are characterized by short development cycles, each with the goal of a shippable product. This provides constant feedback, which helps to establish a customer-oriented development process resulting in products that fit customer's true needs and yield high customer acceptance. It is in the very nature of agile development, that requirement specifications are in perpetual flux. Adjusting the software's design to updated requirements is as daily business as developing new features.

However, a major obstacle in this process are the database systems [2]. Whereas software development tools support developers in the process of designing changes with a comprehensive set of automatized refactoring features, the evolution of databases is usually realized by manually writing scripts of SQL-DDL and -DML operations. This manual database evolution is expensive and

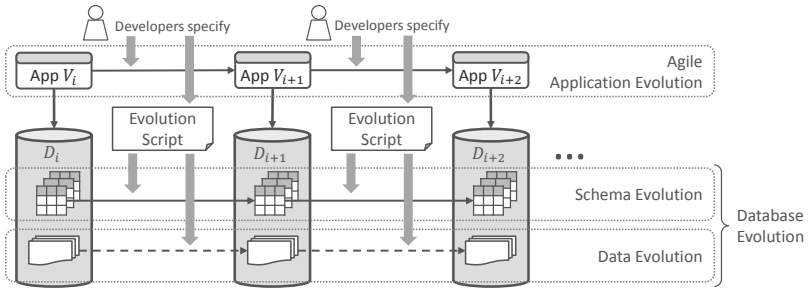

Fig. 1. Database evolution.

error-prone. Furthermore, many software projects show poor integration of the database developers. According to a survey [1], two third of the pooled software developers perform database-related changes without consulting the responsible database developers, which certainly increases the software developer's productivity but is not necessarily helping the quality of the resulting database.

To keep pace with agile software development, the database systems have to supply software-refactoring-like features. Such database evolution features need to evolve the database schema (schema evolution) and payload data (data evolution) in a single consistent step [15]. Such a database evolution processes as illustrated in Figure 1. While evolving an application, the application developer specifies the corresponding database evolution with the help of *schema modification operations* (SMOs). In contrast to SQL-DDL and -DML statements, SMOs specify the evolution of the schema and the data in a descriptive, integrated way and ensure that the data is consistently evolved with the schema. SMOs are typically more compact than a script of DDL and DML operations resulting in the same evolution. On the user side, SMOs increase the developer's productivity while dealing with database evolution and reducing the chances of faulty evolution scripts and unintended data loss. On the database system side, SMOs open the opportunity to optimize and reduce the actual data movement involved in an evolution step or even invert evolution steps for database versioning. These benefits are enabled by the use of SMOs instead of DDL/DML.

A set of SMOs forms a *database evolution language* (*DEL*). Naturally, the design of a particular *DEL* determines its expressiveness. A powerful *DEL* lets the user easily specify all necessary evolution steps. In contrast, a weak *DEL* forces the user into more complicated evolution scripts or even to fall back on DDL/DML statements, which renders the *DEL* useless. In principal, a *DEL* should at least cover the power of DDL and DML of an ordinary database system. We argue, that a *DEL* for relational databases should at least be *relationally complete*: For any relational DDL/DML script, there exists a semantically equivalent sequence of SMOs. Relational DDL/DML scripts create, alter, and drop database objects, while conditions and the actual data are specified using expressions from a given DQL. The latter motivates the relational algebra [5] as the natural reference for determining the power of relational DDL and DML.

Given a relational database $D = \{R_1, \ldots, R_n\}$ with tables R_i, a *DEL* is relationally complete if it can transform D into any other relational database $D' = \{R'_1, \ldots, R'_m\}$ with each R'_i being computable from D with operators from the relational algebra. A minimal language providing relational completeness is $\mathcal{L}_{\text{min}} = \{\textsc{Add}(\cdot,\cdot), \textsc{Del}(\cdot)\}$ with

$$\textsc{Add}(R', \epsilon) \rightarrow D \cup \{R' = \epsilon(R_1, \ldots, R_n)\}$$
$$\textsc{Del}(R) \rightarrow D \setminus \{R\}$$

The add operation adds a new table R' to the database D based on the given relational algebra expression ϵ. The delete operation removes the specified table R from D. Let $inst(\mathcal{L}_{\text{min}})$ be the set of all operation instances of $\mathcal{L}_{\text{min}}$ with valid parameters. Then obviously, a database D can be transformed into any other database D' with a sequence $s \in inst(\mathcal{L}_{\text{min}})^+$. Hence, $\mathcal{L}_{\text{min}}$ is relationally complete. From a practical standpoint however, $\mathcal{L}_{\text{min}}$ is not very appealing, because it is rather unintuitive and not oriented on actual evolution steps. However, any other *DEL* which is as expressive as $\mathcal{L}_{\text{min}}$ is relationally complete as well.

To the best of our knowledge, the most advanced *DEL* design is PRISM++ [8, 6]. PRISM++ provides SMOs to create, rename, and drop both tables and columns, to divide and combine tables both horizontally and vertically, and to copy tables. The PRISM++ authors claim practical completeness for their powerful *DEL*, by validating it against evolution histories of several open source projects. Although this evaluation suggests that PRISM++ is sufficient also for other software projects, it does not provide any reliable completeness guarantee. For instance, we do not see an intuitive way to remove all rows from a table A, which also occur in a table B using the PRISM++ *DEL*, since it does not offer any direct or indirect outer join functionality. Thus, we consider PRISM++ not to be relationally complete. Nevertheless, PRISM++ has an intuitive and field-proven design.

In this paper, we present a relationally *Complete DEL* (CoDEL), building on the set of PRISM++ SMOs to inherit its practical feasibility. However, CoDEL is relationally complete and equally expressive as $\mathcal{L}_{\text{min}}$. Our contributions are:

1. We provide a formal definition of the semantics of all CoDEL operations and propose an SQL-like syntax. With that, CoDEL can serve as a reference language for the formal evaluation of other *DELs*.
2. We show the relational completeness of CoDEL. We show that all operations of the relational algebra – as presented in [5] plus selected extensions – can be expressed in CoDEL and whereby any $\mathcal{L}_{\text{min}}$ expression, as well.
3. We lay the foundation for further research. CoDEL is a *DEL*, whose SMOs are compact with precisely defined semantics. Hence researchers can tackle their challenges on a per-SMO-level ("Divide and Conquer"). For instance, database versioning requires full invertibility of a database evolution. CoDEL allows to define invertibility locally for each operation, which greatly simplifies such research.

We define CoDEL in Section 2, prove its relational completeness in Section 3, discuss related work in Section 4, and conclude the paper in Section 5.

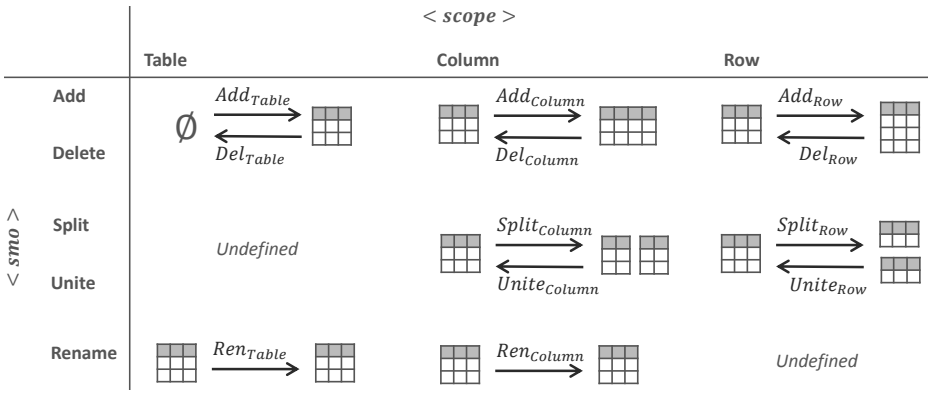

Fig. 2. Structuring of CoDEL.

2 CoDEL

Database evolution changes the schema of a database and/or the already existing data. A *DEL* contains operations to descriptively specify such changes as units, which clearly distinguishes it from SQL-DDL and -DML. PRISM++ limits itself to operations that modify individual tables – no PRISM++ operation accepts more than two tables. This keeps the PRISM++ *DEL* intuitive and easy to learn. CoDEL adopts this principle. However, CoDEL operations systematically cover all possible changes that can be applied to tables. Tables are the fundamental structuring element and the container for primary data in a relational database. Secondary database objects such as views, constraints, functions, stored procedures, indexes, etc. should be considered in database evolution as well. However, in this paper we focus on the evolution of primary data.

CoDEL defines SMOs of the pattern $\langle smo \rangle_{\langle scope \rangle}(\Theta)$, where $\langle smo \rangle$ is the type of operation, $\langle scope \rangle$ is the general database object the operation works on, and Θ is the set of parameters the SMO requires. Figure 2 gives a systematic overview of all SMOs in CoDEL. A relational database table is a two-dimensional structure consisting of columns and rows, hence, SMOs can operate on the level of columns, of rows, or of whole tables. On all three levels there are five basic operations: ADD, DEL, SPLIT, UNITE, and REN. We will now introduce the meaningful operations, as shown in Figure 2. First, CoDEL has two basic operations to create (ADD_{table}) and drop (DEL_{table}) tables as a whole, similar to their counterparts in a standard DDL. Second, CoDEL has a set of operations to modify a table. Hence, CoDEL offers eight table modification SMOs $\langle smo \rangle_{\langle scope \rangle}$ with $\langle scope \rangle \in \{column, row\}$ and $\langle smo \rangle \in \{\text{ADD}, \text{DEL}, \text{SPLIT}, \text{UNITE}\}$. For instance, DEL_{column} removes a column from a given table and SPLIT_{row} partitions a table horizontally, while SPLIT_{column} partitions it vertically. CoDEL defines no SPLIT or UNITE of whole tables, since these operations are restricted to either column or row scope. Third, CoDEL includes two SMOs to rename a table (REN_{table}) and a column (REN_{column}). The renaming of rows is undefined.

Regarding relational completeness, REN_{column}, REN_{table}, DEL_{column}, and DEL_{row} are not necessary. However, they are very common [9] and included in CoDEL for usability's sake. To summarize, CoDEL is the *DEL* $\mathcal{L}_C$ with:

$$\mathcal{L}_C = \left\{ \begin{array}{llll} \text{ADD}_{table}, & \text{DEL}_{table}, & & \\ \text{ADD}_{column}, & \text{DEL}_{column}, & \text{SPLIT}_{column}, & \text{UNITE}_{column}, \\ \text{ADD}_{row}, & \text{DEL}_{row}, & \text{SPLIT}_{row}, & \text{UNITE}_{row}, \\ \text{REN}_{table}, & \text{REN}_{column} & & \end{array} \right\}$$

All CoDEL SMOs require a set Θ of parameters. Let $inst\,(o, D)$ be the set of instances of the SMO o with a valid parameterization regarding the database D. For instance, the only parameter to remove a table with $\text{DEL}_{table}(\Theta)$ is the name of an existing table, so that $inst\,(\text{DEL}_{table}\,(\Theta)\,, D) = \{\text{DEL}_{table}(R) | R \in D\}$. Further, let $inst\,(\mathcal{L}, D) = \bigcup_{o \in \mathcal{L}} inst\,(o, D)$ be the set of all validly parameterized SMO instances of the *DEL* $\mathcal{L}$. Then, a CoDEL evolution script s for a database D is a sequence of instantiated SMOs with $s \in inst\,(\mathcal{L}_C, D_i)^+$, where D_i is the database after the application of the i-th SMO.

In the following, we specify the semantics of all CoDEL SMOs. Table 1 summarizes the definition of the semantics based on $\mathcal{L}_{\text{min}}$. The table also shows the SQL-like syntax we propose for the implementation of CoDEL. In the remainder, $R.C = \{c_1, \ldots, c_n\}$ denotes the set of columns of table R and R_i specifies the version i of the table R. Whenever an SMO does not change the table's name but its columns or rows, we increment this version counter i. CoDEL SMOs take tables as input and return tables. According to the SQL standard, tables are multisets. Our semantics definition with $\mathcal{L}_{\text{min}}$ is based on the relational algebra, though, where tables are sets. However, relational database systems internally manage row identifiers, which are at least unique per table. At the level of SMO implementation, we consider the row identifiers as part of the tables and hence, tables as sets. The corresponding multiset semantics of the SMOs can be achieved, by adding a multiset projection of the resulting tables that removes the row identifiers without eliminating duplicates.

ADD_{table} and DEL_{table} The SMOs ADD_{table} and DEL_{table} are the simplified version of their $\mathcal{L}_{\text{min}}$ counterparts. $\text{ADD}_{table}(R, \{c_1, \ldots, c_n\})$ requires two parameters, a table name R and a set of column definitions c_i. It creates an empty table with the specified name and schema. $\text{DEL}_{table}(R)$ takes only a single parameter, the name of the table to be dropped.

ADD_{column} and DEL_{column} ADD_{column} adds a new column to an existing table. As parameter $\text{ADD}_{column}(R_i, c, f(c_1, \ldots, c_n))$ takes the name R_i of the table, the column definition c of the new column, and a function f. The resulting table is R_{i+1}. ADD_{column} applies the function f to each row in R_i to calculate the row's value for the new column c. The function f receives all other column values of the row as parameters. Figure 3 shows an example: $\text{ADD}_{column}(Person_0, zip, getZip(name, age, address))$ adds a column *zip* to $Person_0$ by determining the zip code based on the currently available information.

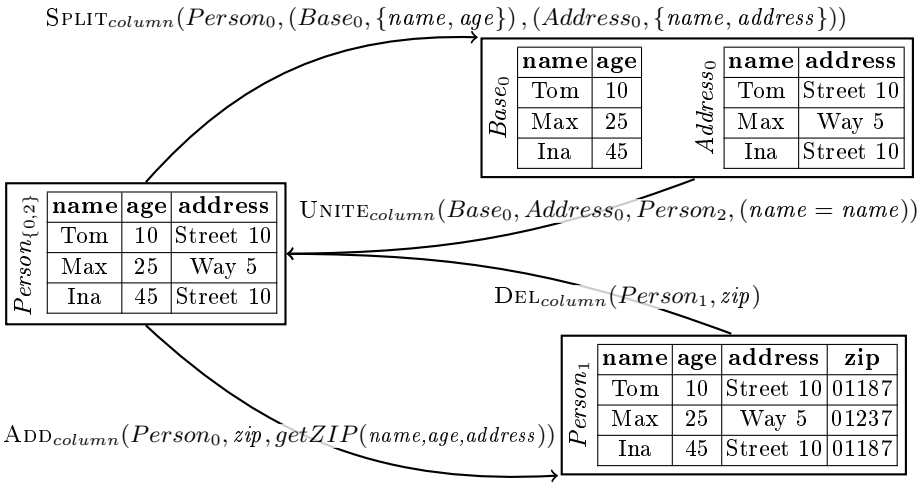

Fig. 3. Example for the operations on columns.

DEL$_{column}$ removes a column from a table. Specifically, DEL$_{column}(R_i, c)$ takes the name R_i of an existing table and the name $c \in R_i.C$ of the column that should be removed from R_i. The resulting table is R_{i+1}. Figure 3 shows an example, where we remove the column *zip* from table $Person_1$.

SPLIT$_{column}$ and UNITE$_{column}$ SPLIT$_{column}$ partitions a table vertically and removes the original table. SPLIT$_{column}$ has a generalized semantics, where the resulting partitioning is allowed to be incomplete and overlapping. SPLIT$_{column}(R, (S, \{s_1, \ldots, s_n\}), (T, \{t_1, \ldots, t_m\}))$ takes the name R of the original table, a pair of table name S and a set of column names s_i as specification of the first partition and optionally a second pair $(T, \{t_1, \ldots, t_m\})$ as specification of the second partition. The two sets of column definitions are independent. In case $S.C \cap T.C \neq \emptyset$, the columns $S.C \cap T.C$ are copied. In case $S.C \cup T.C \subset R.C$, the partitioning is incomplete. If the second partition is not specified, T is not created. Note that CoDEL prohibits empty sets of column definitions for S and T, since tables must have at least one column. Figure 3 shows an example with the SPLIT$_{column}$ SMO. Table $Person_0$ is vertically partitioned to general information ($Base_0$) and address information ($Address_0$). The partitions overlap on the column *name* to maintain the connection between addresses and person.

UNITE$_{column}$ is the inverse operation of SPLIT$_{column}$. It joins two tables based on a given condition and removes the original tables. As parameters, UNITE$_{column}(R, S, T, cond, o)$ takes the names R and S of the original tables, the name T of the resulting table, a join condition *cond* using SQL predicates without further nesting, and the optional request o for an outer join. In case $o = \top$, UNITE$_{column}$ performs an outer join, so that no rows from the original tables are lost. In case $o = \bot$ (or not specified) UNITE$_{column}$ performs an inner join. With the inner join, UNITE$_{column}$ loses all rows from R and S that do not find a join partner, since R and S are dropped after the join. Note that restrict-

SMO:	$\textsc{Add}_{table}(R, \{c_1, \ldots, c_n\})$	$\textsc{Del}_{table}(R)$
Semantic:	$\textsc{Add}(R, \pi_{c_1,\ldots,c_n}(\emptyset))$;	$\textsc{Del}(R)$;
Syntax:	`CREATE TABLE` R `(`c_1`,...,`c_n`)`	`DROP TABLE` R
SMO:	$\textsc{Add}_{column}(R_i, c, f(c_1, \ldots, c_n))$	
Semantic:	$\textsc{Add}\big(R_{i+1}, \pi_{R_i.C \cup \{c \leftarrow f(c_1,\ldots,c_n)\}}(R_i)\big)$; $\textsc{Del}(R_i)$;	
Syntax:	`ADD COLUMN` c `AS` f`(`c_1`,...,`c_n`) INTO` R_i	
SMO:	$\textsc{Del}_{column}(R_i, c)$	
Semantic:	$\textsc{Add}\big(R_{i+1}, \pi_{R_i.C \setminus \{c\}}(R_i)\big)$; $\textsc{Del}(R_i)$;	
Syntax:	`DROP COLUMN` c `FROM` R_i	
SMO:	$\textsc{Split}_{column}(R, (S, \{s_1, \ldots, s_n\}), (T, \{t_1, \ldots, t_m\}))$	
Semantic:	$\textsc{Add}(S, \pi_{s_1,\ldots,s_n}(R))$; $[\textsc{Add}(T, \pi_{t_1,\ldots,t_m}(R))]$; $\textsc{Del}(R)$;	
Syntax:	`DECOMPOSE TABLE` R `INTO` S `(`s_1`,...,`s_n`) [,` T `(`t_1`,...,`t_m`)]`	
SMO:	$\textsc{Unite}_{column}(R, S, T, cond, o)$	
Semantic:	$o = \bot$: $\textsc{Add}(T, R \bowtie_{cond} S)$;	$o = \top$: $\textsc{Add}(T, R \rtimes_{cond} S)$; $\textsc{Del}(R)$; $\textsc{Del}(S)$;
Syntax:	`[OUTER] JOIN TABLE` R`,` S `INTO` T `WHERE` $cond$	
SMO:	$\textsc{Add}_{row}(R_i, G, \{(a_1, f_1(G,V)), \ldots, (a_m, f_m(G,V))\}, S)$	
Semantic:	S given: $\textsc{Add}\Big(S, \gamma_{G, \{f_j(G,V) \to a_j \mid 1 \le j \le m\}}(R_i)\Big)$	
	S not given: $\textsc{Add}\Big(R_{i+1}, R_i \cup \gamma_{G, \{f_j(G,V) \to a_j \mid 1 \le j \le m\}}(R_i)\Big)$; $\textsc{Del}(R_i)$;	
Syntax:	`AGGREGATE TABLE` R_i `(`g_1`,...,`g_n`) WITH` $a_1 = f_1(G,V)$`,...` `[INTO` S`]`	
SMO:	$\textsc{Del}_{row}(R_i, cond)$	
Semantic:	$\textsc{Add}(R_{i+1}, \sigma_{\neg cond}(R_i))$; $\textsc{Del}(R_i)$;	
Syntax:	`REMOVE FROM TABLE` R_i `WHERE` $cond$	
SMO:	$\textsc{Split}_{row}(R, (S, cond_S), (T, cond_T))$	
Semantic:	$\textsc{Add}(S, \sigma_{cond_S}(R))$; $[\textsc{Add}(T, \sigma_{cond_T}(R))]$; $\textsc{Del}(R)$;	
Syntax:	`PARTITION TABLE` R `INTO` S `WITH` $cond_S$ `[,` T `WITH` $cond_T$`]`	
SMO:	$\textsc{Unite}_{row}(R, S, T)$	
Semantic:	$\textsc{Add}\big(T, \pi_{R.C \cup \{\omega \to a_i \mid a_i \in S.C \setminus R.C\}}(R) \cup \pi_{S.C \cup \{\omega \to a_i \mid a_i \in R.C \setminus S.C\}}(S)\big)$; $\textsc{Del}(R)$; $\textsc{Del}(S)$;	
Syntax:	`MERGE TABLE` R`,` S `INTO` T	
SMO:	$\textsc{Ren}_{table}(R, R')$	$\textsc{Ren}_{column}(R_i, c, c')$
Semantic:	$\textsc{Add}(R', R)$; $\textsc{Del}(R)$;	$\textsc{Add}\big(R_{i+1}, \rho_{c'/c}(R_i)\big)$; $\textsc{Del}(R_i)$;
Syntax:	`RENAME TABLE` R `INTO` R'	`RENAME COLUMN` c `IN` R_i `TO` c'

Table 1. Syntax and Semantic of CoDEL operations.

ing the join to foreign key relations as other *DELs* do, does not prevent this information loss. A foreign key does not guarantee that every row in the referenced table is actually referenced by at least one row in the referencing table. Figure 3 also shows an example of $\textsc{Unite}_{column}$. The tables $Base_0$ and $Address_0$ are inner joined to the table $Person_2$ based on equal names. Since all persons have an address in this example, no rows are lost.

ADD$_{row}$ and DEL$_{row}$ ADD$_{row}$ adds new rows to an existing table by aggregating the data in the current rows. As parameter ADD$_{row}(R_i, G, \{(a_j, f_j(G,V)) \,|1 \leq j \leq m\}, S)$ requires the name R_i of the original table, the set of grouping columns $G = \{g_1, \ldots, g_n\} \subseteq R_i.C$, a set of pairs of column name a_j and aggregations function f_j, and optionally a new table name S. ADD$_{row}$ produces new rows by grouping table R_i by all columns $g_k \in G$ and calculating the values for the columns a_j with the functions f_j. The functions f_j may contain constants, the values of the grouping columns G, and aggregate functions upon the remaining columns $V = R_i.C \setminus G$. If the new table name S is specified, ADD$_{row}$ creates S with the newly produced rows and R_i remains available, which is particularly necessary, when the newly created rows have a different set of columns than $R_i.C$. Otherwise, ADD$_{row}$ appends the new rows to R_i to form its new version R_{i+1}. In this case, we require the column definitions of the new rows to match the original table R_i, hence $\{g_1, \ldots, g_n\} \cup \{a_1, \ldots, a_m\} = R.C$. In general, the set of grouping columns is also allowed to be empty resulting in one group and hence, one new row.

DEL$_{row}$ removes rows from a given table. DEL$_{row}(R_i, cond)$ takes the name of an existing table R_i and a condition *cond*. It removes all rows, which satisfy the condition and evolves the table to R_{i+1}.

SPLIT$_{row}$ and UNITE$_{row}$ SPLIT$_{row}$ partitions a table horizontally. However, its semantics is more general than standard horizontal partitioning [4]. The SMO creates at most two partitions out of a given table – with the partitioning allowed to be incomplete and overlapping – and removes the original table. More precisely, SPLIT$_{row}(R, (S, cond_S), (T, cond_T))$ takes the name of the original table, a pair of table name S and condition $cond_S$ as specification of the first partition and optionally a second pair $(T, cond_T)$ as specification of the second partition. Both conditions $cond_S$ and $cond_T$ are independent. If the original tables contain rows that fulfill neither of the conditions, the resulting partitioning is incomplete. Rows that fulfill both conditions are copied resulting in overlapping partitions. In case both conditions hold for all rows, i.e., $cond_S = \top$ and $cond_T = \top$, T is a complete copy of S. Hence, SPLIT$_{row}$ subsumes the functionality of a copy operations that can be found in other *DELs*. If $cond_T$ is not specified, SPLIT$_{row}$ does not create table T.

UNITE$_{row}$ is the inverse operation of SPLIT$_{row}$; it merges two given tables along the row dimension and removes the original tables. As parameters UNITE$_{row}(R, S, T)$ requires, the names R and S of the original tables and the name T of the resulting table. The schema of R and S are not required to by equivalent. In case both schemas differ, T contains null values (ω) in the corresponding cells. UNITE$_{row}$ eliminates duplicates in T. In case R and S contain equivalent rows, these rows will show up only once in T.

REN$_{table}$ and REN$_{column}$ The last two SMOs rename schema elements. REN$_{table}(R, R')$ renames the table with the name R into R'. REN$_{column}(R_i, c, c')$ renames the column c in table R_i into c', which results in table R_{i+1}.

We use the semantics definition, as summarized in Table 1, to show the relational completeness of CoDEL in the following section.

3 Relational Completeness

To show the relational completeness of CoDEL, we argue that it is at least as powerful as $\mathcal{L}_{\min}$ (Section 1), which is relationally complete by definition. There is always a semantically equivalent expression in CoDEL for any expression in $\mathcal{L}_{\min}$. The $\textsc{Del}(R)$ operation from $\mathcal{L}_{\min}$ is trivial, since it is equivalent to CoDEL's $\textsc{Del}_{table}(R)$. On the contrary, $\textsc{Add}(R, \epsilon)$ from $\mathcal{L}_{\min}$ is more complex, as ϵ covers the power of the relational algebra. Since both the relational algebra and CoDEL are closed languages, it is reasonable to address each operation of the relational algebra separately. We show that, for each operation from the relational algebra, there is a semantically equivalent sequence of SMOs in CoDEL.

We assume the basic relational algebra [5] and add common extensions like the extended projection, aggregation, and outer joins. However, we intentionally exclude other extensions like the transitive closure and sorting. CoDEL does not cover these extensions, since CoDEL is non-recursive and set-based. We maintain these characteristics, since they proved to be a reasonable trade-off between expressiveness and usability, however, they are open for further research. With respect to implementations based on current database management systems, the distinction between different types of null values [19] is not considered. For instance $\textsc{Unite}_{row}$ adds null values in columns, which existed in only one input table, losing the information, whether a value was null before or did not exist at all. The following sections will consider all constructs from the relational algebra including the chosen extensions and show that CoDEL is capable to obtain the semantically equivalent results.

Relation: $\boldsymbol{R}$ The basic elements of the relational algebra are relations. They contain the data and are directly accessible by CoDEL as tables. Whenever one table is required multiple times within a relational algebra expression, CoDEL allows to copy them using $\textsc{Split}_{row}(R, (S, \top), (T, \top))$.

Selection: $\boldsymbol{\sigma_{cond}(R)}$ The selection returns the subset of rows from R, which satisfy the condition *cond*. CoDEL's $\textsc{Split}_{row}(R, (S, cond))$ is semantically equivalent, which directly follows from the semantics definition in Table 1.

Rename: $\boldsymbol{\rho_{c'/c}(R_i)}$ Renaming a column is subsumed by the extended projections, however, we include it here for completeness. CoDEL's obvious semantic equivalent according to Table 1 is $\textsc{Ren}_{column}(R_i, c, c')$.

Extended Projection: $\boldsymbol{\pi_P(R)}$ We will immediately consider the extended projection, as it subsumes the traditional projection. The extended projection defines a new set of columns, whose values are computed by functions depending on the existing columns. Assume the projection $P = \{f_k(R.C) \to a_k | 1 \leq k \leq m\}$ with $n = |R.C|$. The CoDEL sequence below, realizes such an extended projection. Without loss of generality, we use for-loops to iterate over the attribute sets. Since this is only schema depending and data independent, it does not extend the expressiveness of the DEL but is simply a short notation.

1: **for** $k = [1..m]$ **do**
2: $\quad$ ADD$_{column}(R_{i+k-1}, a'_k, f_k(r_1, \ldots, r_n))$;
3: **for** $r_j \in R.C$ **do**
4: $\quad$ DEL$_{column}(R_{i+m+j-1}, r_j)$;
5: **for** $k = [1..m]$ **do**
6: $\quad$ REN$_{column}(R_{i+m+n+k-1}, a'_k, a_k)$;
7: **for** $k = [i..(i+2m+n-1)]$ **do**
8: $\quad$ DEL$_{table}(R_k)$;

$$R_{i+1} \overset{2}{=} \pi_{r_1,\ldots,r_n,f_1(r_1,\ldots,r_n)\to a'_1}(R_i) \tag{1}$$

$$R_{i+m} \overset{1,2}{=} \pi_{r_1,\ldots,r_n,f_1(r_1,\ldots,r_n)\to a'_1,\ldots,f_m(r_1,\ldots,r_n)\to a'_m}(R_i) \tag{2}$$

$$R_{i+m+1} \overset{4}{=} \pi_{r_2,\ldots,r_n,a'_1,\ldots,a'_m}(R_{i+m}) \tag{3}$$

$$R_{i+m+n} \overset{3,4}{=} \pi_{a'_1,\ldots,a'_m}(R_{i+m}) = \pi_{f_1(r_1,\ldots,r_n)\to a'_1,\ldots,f_m(r_1,\ldots,r_n)\to a'_m}(R_i) \tag{4}$$

$$R_{i+m+n+1} \overset{6}{=} \pi_{a'_1\to a_1,a'_2,\ldots,a'_m}(R_{i+m+n}) \tag{5}$$

$$\begin{aligned} R_{i+m+n+m} &\overset{5,6}{=} \pi_{a'_1\to a_1,\ldots,a'_m\to a_m}(R_{i+m+n}) \\ &= \underline{\underline{\pi_{f_1(R_i.C)\to a_1,\ldots,f_m(R_i.C)\to a_m}(R_i)}} \end{aligned} \tag{6}$$

The first SMO adds a new column, with a masked name, for each column of the output table. This allows to compute the new values based on all existing ones. Afterwards, we drop the old columns, rename the new columns to their unmasked name, and remove all intermediate tables. Applying the semantics definitions of the CoDEL SMOs results in the desired extended projection, as shown above. The concrete line of the CoDEL sequence, which is applied in the semantics computation, is indicated by the numbers above the equal signs.

Outer Join: $\boldsymbol{R \bowtie\!\!\!\!\!\subset_p S}$ The outer join is another common extension to the traditional relational algebra. Beyond the rows according to an inner join, it also includes those rows in the result, which did not find a join partner. The missing values for columns of the other table are filled with null values ω respectively. Obviously, CoDEL's UNITE$_{column}(R, S, T, p, \top)$ is semantically equivalent, since we explicitly introduced the option to perform outer joins.

Cross Product: $\boldsymbol{R \times S}$ The cross product produces a row in the output table for each pair of rows from the input tables. The following sequence of CoDEL SMOs is semantically equivalent as shown below.

1: $\textsc{Add}_{column}(R_i, j, 1)$;
2: $\textsc{Add}_{column}(S_k, j, 1)$;
3: $\textsc{Unite}_{column}(R_{i+1}, S_{k+1}, T_0, R_{i+1}.j = S_{k+1}.j, \perp)$;
4: $\textsc{Del}_{column}(T_0, j)$;

$$R_{i+1} \overset{1}{=} \pi_{r_1,\ldots,r_n,1 \to j}(R_i) = \{(r_1, \ldots, r_n, 1) \mid (r_1, \ldots, r_n) \in R_i\} \tag{7}$$

$$S_{k+1} \overset{2}{=} \{(s_1, \ldots, s_m, 1) \mid (s_1, \ldots, s_m) \in S_k\} \tag{8}$$

$$\begin{aligned} T_0 &\overset{3}{=} R_{i+1} \bowtie_{R_{i+1}.j = S_{k+1}.j} S_{k+1} \\ &= \{(r_1, \ldots, r_n, s_1, \ldots, s_m, 1) \mid (r_1, \ldots, r_n) \in R_i, (s_1, \ldots, s_m) \in S_k\} \end{aligned} \tag{9}$$

$$\begin{aligned} T_1 &\overset{4}{=} \{(r_1, \ldots, r_n, s_1, \ldots, s_m) \mid (r_1, \ldots, r_n) \in R_i, (s_1, \ldots, s_m) \in S_k\} \\ &= \underline{\underline{R \times S}} \end{aligned} \tag{10}$$

We add a new column j to both tables with $j \notin R_i.C$ and $j \notin S_k.C$ and the default value 1 to perform an inner join on j. Since its value is always 1, there will be one row in the output table for each pair of rows from the two input tables. We remove the additional column j and finally show the semantic equivalence between the relational cross product and the presented sequence of CoDEL SMOs.

Aggregate: $\boldsymbol{\gamma_{G,F}(R)}$ The aggregation is another typical extension to the relational algebra. The rows are grouped by one set of columns $G = \{g_1, \ldots, g_n\} \subseteq R.C$. Additional columns $A = \{a_i | 1 \leq i \leq p\}$ are computed by functions $F = \{f_i(G, V) \to a_i | a_i \in A\}$ with $V = \{v_1, \ldots, v_m\} = R.C \setminus G$. These functions may contain values from grouping columns G, aggregate functions on the remaining columns in V, constants, and arithmetic functions. CoDEL contains a dedicated operation $\textsc{Add}_{row}(R, G, F, S)$. It writes the result of the aggregation to the new table S. According to the semantics definition in Table 1, the semantics of $\textsc{Add}_{row}$ equals the discussed aggregation semantics from the relational algebra.

Union: $\boldsymbol{R \cup S}$ The relational union, merges the rows from both input tables to the one output table including an elimination of duplicates. Using the SMO $\textsc{Unite}_{row}$, CoDEL provides a semantic equivalent to the relational union operation.

1: $\textsc{Unite}_{row}(R, S, T)$;

$$T \overset{1}{=} \pi_{R.C}(R) \cup \pi_{S.C}(S) = \underline{\underline{R \cup S}} \tag{11}$$

Please note, that the union in the relational algebra requires R and S to have identical sets of attributes ($R.C = S.C$), which justifies the simplification step.

Difference: $R \setminus S$ The relational difference returns all rows, which occur in the first, but not in the second table. Analogous to the union, it requires R and S to have identical sets of columns ($R.C = S.C$). The following CoDEL sequence is semantically equivalent to the relational difference.

1: $\textsc{Add}_{column}(S_k, j, 1)$;
2: $\textsc{Unite}_{column}(R_i, S_{k+1}, T_0, (R_i.c_1 = S_{k+1}.c_1 \wedge \ldots \wedge R_i.c_n = S_{k+1}.c_n), \top)$;
3: $\textsc{Del}_{row}(T_0, j \neq \omega)$;
4: $\textsc{Del}_{column}(T_1, j)$;

$$S_{k+1} \overset{1}{=} \pi_{s_1,\ldots,s_m,1\to j}(S_k) \tag{12}$$

$$\begin{aligned} T_0 &\overset{2}{=} R_i \rtimes S_{k+1} \\ &= \{(r_1,\ldots,r_n,1) \mid (r_1,\ldots,r_n) \in R_i, (r_1,\ldots,r_n,1) \in S_{k+1}\} \\ &\quad \cup \{(r_1,\ldots,r_n,1) \mid (r_1,\ldots,r_n) \notin R_i, (r_1,\ldots,r_n,1) \in S_{k+1}\} \\ &\quad \cup \{(r_1,\ldots,r_n,\omega) \mid (r_1,\ldots,r_n) \in R_i, (r_1,\ldots,r_n,1) \notin S_{k+1}\} \end{aligned} \tag{13}$$

$$\begin{aligned} T_1 &\overset{3}{=} \sigma_{\neg(j\neq\omega)}(T_0) = \sigma_{(j=\omega)}(T_0) \\ &= \{(r_1,\ldots,r_n,\omega) \mid (r_1,\ldots,r_n) \in R_i, (r_1,\ldots,r_n,1) \notin S_{k+1}\} \end{aligned} \tag{14}$$

$$\begin{aligned} T_2 &\overset{4}{=} \pi_{R.C}(T_1) \\ &= \{(r_1,\ldots,r_n) \mid (r_1,\ldots,r_n) \in R_i, (r_1,\ldots,r_n) \notin S_k\} = \underline{\underline{R_i \setminus S_k}} \end{aligned} \tag{15}$$

We add a new column j to S_k with $j \notin S_k.C$ and the default value 1. The outer join on all columns $c_i \in R_i.C = S_k.C$ is applicable, since the initial column sets are equal. Due to the nature of the outer join, the resulting table contains all rows which were in at least one of the two input tables. However, all rows, which occurred in S_k have the value 1 in the column j and are removed by the third SMO. All rows which occurred exclusively in R have a null value ω in the column j and remain as result. Applying the semantics definition of the SMOs finally leads to the relational difference operation. Please note, that $(r_1,\ldots,r_n) \notin S_k$ is equal to $(r_1,\ldots,r_n,1) \notin S_{k+1}$ due to the first step.

Finally, we successfully showed that CoDEL provides a semantic equivalent for each relational algebra expression, which makes it equally expressive as $\mathcal{L}_{\min}$. Hence, it is relationally complete and a sound foundation for further research.

4 Related Work

Database evolution is a well recognized topic in the database research community [13, 18]. There are a number of approaches to increase comfort and efficiency in database evolution, for instance by defining a schema evolution aware query language [14]. Another approach is to define database evolution languages graph-based [12]. This allows modeling dependencies between different artifacts in the information system and applying changes globally. Furthermore, MeDEA [10] provides a general framework to describe database evolution in the context of

evolving applications. MoDEF [17] basically introduces an IDE extension to automate the co-evolution of the evolving client schemas and the store.

Currently, PRISM [7] appears to provide the most advanced database evolution tool including an SMO-based *DEL*. PRISM was first introduced in 2008 and focused on the plain database evolution [8]. Later, the authors extended it to PRISM++, which includes the modification of constraints and update rewriting [6]. To benchmark database evolution languages and tools, researchers also analyzed the evolution histories of Wikimedia and other open source projects [9, 16]. Finally, database versioning extends the ideas of database evolution to allow both forward and backward compatibility between the different versions of evolving schemas [15]. Another extension of PRISM takes a first step into this direction by answering queries on former schema versions according to the current data [11]. The presented *DEL* CoDEL inherits the principle style of SMOs from PRISM. However, PRISM is not relationally complete, while CoDEL is. This additional characteristic provided by CoDEL is highly valuable with respect to further research, particularly in the field of automated database versioning based on SMOs, where falling back on common DDL and DML evolution scripts is not an option.

5 Conclusion

Agile software development methods embrace the change. While software developers find support in refactoring methods to evolve their software, database developers still have to fiddle with DDL/DML scripts to evolve schema and data of a productive database consistently. Adding evolution support to a DBMS involves the design of a database evolution language (*DEL*). In this paper we considered the relational completeness of *DELs* for relational databases. Relational completeness is an important property of *DELs*. *DELs* that are incomplete in this respect, can force the user back to the manual evolution process based on DDL and DML limiting the utility of the evolution functionality. We presented the relationally complete *DEL* CoDEL. We detailed its formal definition and showed its relational completeness. CoDEL is to our best knowledge the first well-defined, relationally complete *DEL*. CoDEL can serve as a reference language for productive implementations of database evolution in DBMSs.

The solid formal base of CoDEL is also important for research and development beyond database evolution. For instance in database versioning, multiple clients access the same data in different schema versions. Database versioning requires invertible SMOs, so that the database system can translate data back and forth between schema versions. For the investigation of the invertibility of SMOs a solid formal definition of the SMOs is a prerequisite. Hence, CoDEL offers a good starting point towards database versioning. For the near future, however, we hope CoDEL helps to jump start more implementations of proper database evolution features in the DBMSs on the market, so that agile development methods final arrive at the database layer.

6 Acknowledgments

This work is funded by the German Research Foundation (DFG) within the Research Training Group RoSI (GRK 1907). The final publication is available at Springer via http://dx.doi.org/10.1007/978-3-319-23135-8_5.

References

1. Scott W. Ambler. Whence Data Management? Dr. Dobb's Journal, 2006, no. 390, p. 79.
2. Scott W. Ambler and Pramod J. Sadalage. Refactoring Databases: Evolutionary Database Design. Addison-Wesley Signature, 2006, isbn 978-0321774514.
3. Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for Agile Software Development, 2001.
4. Stefano Ceri, Mauro Negri, and Giuseppe Pelagatti. Horizontal Data Partitioning in Database Design. SIGMOD Conference, 1982, pp. 128-136.
5. E. F. Codd. A Relational Model of Data for Large Shared Data Banks. Communications of the ACM, 1970, vol. 15, no. 3, pp. 162-166.
6. Carlo A. Curino, Hyun J. Moon, Alin Deutsch, and Carlo Zaniolo. Update Rewriting and Integrity Constraint Maintenance in a Schema Evolution Support System: PRISM++. VLDB Endowment, 2010, vol. 4, no. 2, pp. 117-128.
7. Carlo A. Curino, Hyun J. Moon, Alin Deutsch, and Carlo Zaniolo. Automating the Database Schema Evolution Process. VLDB Journal, 2012, vol. 22, no. 1, pp. 73-98.
8. Carlo A. Curino, Hyun J. Moon, and Carlo Zaniolo. Graceful Database Schema Evolution: the PRISM Workbench. VLDB Endowment, 2008, vol. 1, no. 1, pp. 761-772.
9. Carlo A. Curino, Letizia Tanca, Hyun J. Moon, and Carlo Zaniolo. Schema Evolution in Wikipedia: Toward a Web Information System Benchmark. ICEIS, 2008, pp. 323-332.
10. Eladio Domínguez, Jorge Lloret, Ángel L. Rubio, and María A. Zapata. MeDEA: A Database Evolution Architecture with Traceability. Data & Knowledge Engineering, 2008, vol. 65, no. 3, pp. 419-441.
11. Hyun J. Moon, Carlo A. Curino, Myungwon Ham, and Carlo Zaniolo. PRIMA – Archiving and Querying Historical Data with Evolving Schemas. SIGMOD Conference, 2009, pp. 1019-1022.
12. George Papastefanatos, Panos Vassiliadis, Alkis Simitsis, Konstantinos Aggistalis, Fotini Pechlivani, and Yannis Vassiliou. Language Extensions for the Automation of Database Schema Evolution. ICEIS, 2008, pp. 74-81.
13. Erhard Rahm and Philip A. Bernstein. An Online Bibliography on Schema Evolution. SIGMOD Record, 2006, vol. 35, no. 4, pp. 30-31.
14. John F. Roddick. SQL/SE – A Query Language Extension for Databases Supporting Schema Evolution. SIGMOD Record, 1992, vol. 21, no. 3, pp. 10-16.
15. John F. Roddick. A Survey of Schema Versioning Issues for Database Systems. Information and Software Technology, 1995, vol. 37, no. 7, pp. 383-393.
16. Ioannis Skoulis, Panos Vassiliadis, and Apostolos Zarras. Open-Source Databases: Within, Outside, or Beyond Lehman's Laws of Software Evolution? LNCS, 2014, vol. 8484, pp. 379-393.
17. James F. Terwilliger, Philip A. Bernstein, and Adi Unnithan. Worry-Free Database Upgrades. SIGMOD Conference, 2010, p. 1191.
18. James F. Terwilliger, Anthony Cleve, and Carlo A. Curino. How Clean is Your Sandbox? LNCS, 2012, vol. 7307, 2012, pp. 1-23.
19. Carlo Zaniolo. Database Relations with Null Values. Journal of Computer and System Sciences, 1984, vol. 28, no. 1, pp. 142-166.