A Declarative Approach to Analyzing Schema Objects and Functional Dependencies

Christiane Engels[1], Andreas Behrend[1(✉)], and Stefan Brass[2]

[1] Institut für Informatik III, Rheinische Friedrich-Wilhelms-Universität Bonn, Bonn, Germany
{engelsc,behrend}@cs.uni-bonn.de

[2] Institut für Informatik, Martin-Luther-Universität Halle-Wittenberg, Halle, Germany
brass@informatik.uni-halle.de

Abstract. Database schema elements such as tables, views, triggers and functions are typically defined with many interrelationships. In order to support database users in understanding a given schema, a rule-based approach for analyzing the respective dependencies is proposed using Datalog expressions. We show that many interesting properties of schema elements can be systematically determined this way. The expressiveness of the proposed analysis is exemplarily shown with the problem of computing induced functional dependencies for derived relations.

Keywords: Schema analysis · Functional dependencies · Datalog

1 Introduction

The analysis of database schema elements such as tables, views, triggers, user-defined functions and constraints provide valuable information for database users for understanding, maintaining and managing a database application and its evolution. In the literature, schema analysis has been investigated for improving the quality of SQL/program code or detecting program errors [3] and for determining the consequences of schema changes [10], versioning [8], or matching [11]. In addition, the analysis of schema objects plays an important role for tuning resp. refactoring database applications [2]. All these approaches rely on exploring dependencies between schema objects and an in-depth analysis of their components and interactions. A comprehensive and flexible analysis of schema elements, however, is not provided as these approaches are typically restricted to some subparts of a given schema.

The same is true for analysis features provided by commercial systems where approaches such as integrity checking, executing referential actions or query change notification (as provided by Oracle) already use schema object dependencies but in an implicit and nontransparent way, only. That is, no access to the underlying meta-data is provided to the user nor can be freely analyzed by means of user-defined queries. Even the meta-data about tables and SQL views

M. Kirikova et al. (Eds.): ADBIS 2017, CCIS 767, pp. 73–81, 2017.
DOI: 10.1007/978-3-319-67162-8_9

which are sometimes provided by system tables cover only certain information of the respective schema elements. In this paper, we propose a uniform approach for analyzing schema elements in a comprehensive way. To this end, the schema objects are compiled and their meta-data is stored into a Datalog program which employs queries for deriving interesting properties of the schema. This way, indirect dependencies between tables, views and user-defined functions (UDFs) can be determined which is important for understanding follow-up changes. In order to show the expressiveness of the proposed analysis, our rule-based approach is applied to the problem of deducing functional dependencies (FDs) for derived relations, i.e., views, based on FDs defined for base relations. This so-called *FD propagation* or *FD-FD implication* problem has been studied since the 80s [7,9,12] and has applications in data exchange [6], data integration [4], data cleaning [7], data transformations [5], and semantic query optimization. We show that our rule-based approach to schema analysis is well-suited for realizing all known techniques for FD propagation indicating the expressiveness of the proposed analysis. In particular, our contributions are as follows:

- We propose an approach for analyzing the properties of views, tables, trigger and functions in a uniform way.
- Our declarative approach can be easily extended for refining the analysis by user-defined queries.
- The employed Datalog solution can be simply transfered into SQL systems.
- In order to show the expressiveness of our approach, the implication problem for functional dependencies is investigated using our approach.

2 Rule-Based Schema Analysis

A database schema describes the structure of the data stored in a database system but also contains views, triggers, integrity constraints and user defined functions for data analysis. Functions and these different rule types, namely deductive, active and normative rules, are typically defined with various inter-dependencies. For example, views are defined with respect to base relations and/or some other views inducing a hierarchy of derived queries. In particular, the expression `CREATE VIEW q AS SELECT ... FROM` $\mathrm{p}_1, \mathrm{p}_2, ..., \mathrm{p}_n$ leads to the set $\{\mathrm{p}_1 \rightarrow \mathrm{q}, \ldots, \mathrm{p}_n \rightarrow \mathrm{q}\}$ of direct dependencies between the derived relation q and derived or a base relations p_i. which are typically represented by means of a predicate dependency graph for analyzing indirect dependencies via the transitive closure, too. This allows for understanding the consequences of changes made to the instances of the given database schema (referred to as update propagation in the literature) or to its structure. This is important when a database user wants to know all view definitions potentially affected by these changes. Various dependencies can occur in a database schema such as table-to-table dependencies induced by triggers or view-to-table dependencies which can be induced by functions. The analysis of such dependencies can be further refined by structural details (e.g., negative vs. positive dependencies as needed for update propagation) as well as by considering the syntactical components of

schema objects such as column names (attributes) or operator types (sum, avg, insert, delete, etc.). To this end, the definitions of schema objects need to be parsed and the obtained tokens stored as queryable facts. This kind of analysis is well-known from meta-programming in Prolog which led to the famous vanilla interpreter. For readability reasons we use Datalog with facts such as `base(R,A)` (base relation R with arity A), `derived(V,A)` (view V with arity A), `dep(To,From)` (dependency between relations), `call(V,I,O,F)` (input I and output O of function F in view V), `attr(R,P,N)` (position P of attribute named N in relation R) for representing meta-information about a given view or user-defined function. Based on these facts, schema analysis can be realized by queries like

```
attr_dups(N) ← attr(R1,_,N),attr(R2,_,N),R1<>R2
idb_func_pred(V) ← derived(V,_),call(V,_,_,_)
base_changes(B) ← path(B,f1),base(B,_),func(f1,_)
tbl_dep(A,B) ← base(A,_),base(B,_),path(A,F),path(F,B),func(F,_)
```

for determining reused attribute names, views calling a function, base tables possibly changed by function f_1, and cyclic dependencies between two base tables through a function. This way, many interesting properties of schema elements can be systematically determined which supports users in understanding the interrelationships of schema elements. Most database systems already allow for storing and querying meta-data about schema elements in a simple way but a comprehensive (and in particular user-driven) analysis like this is still missing.

3 Functional Dependency Propagation

In order to show the expressiveness of our approach, we investigate the possibility to compute induced FDs for derived relations using the deductive rules introduced above. FDs form special constraints which are assumed to hold for any possible valid database instance. The FD propagation problem is undecidable in the general setting for arbitrary relational expressions [9]. Even restricted to SC views, i.e., relational expressions allowing selection and cross product only, the propagation problem turns out to be coNP-complete (for an in-depth discussion on complexity see [7]). In favor of addressing the general setting, we drop the ambition of achieving completeness by considering a special case, only. Instead, we allow for arbitrary expressions over all relational operators, multiple propagation steps and possibly finite domains[1] in order to cover the majority of practical cases.

3.1 Preliminaries

A functional dependency $\alpha = \{A_1, \ldots, A_n\} \rightarrow B$ states that the attribute values of α determine those of B. The restriction to univariate right sides can be done

[1] Finite domains may introduce new FDs because of limited value combinations.

without loss of generality as well as the representation of FDs satisfying $B \notin \alpha$, only.[2] We allow $\alpha = \emptyset$ which means that the attribute values of B are constant. For our FD propagation rules, we employ a Datalog variant with special data types for finite, one-leveled sets and finite, possibly nested lists. In our approach we use the extended transitivity axiom

$$\alpha \rightarrow B,\ \gamma \rightarrow D,\ B \in \gamma,\ D \notin \alpha\ \Rightarrow\ \alpha \cup (\gamma - B) \rightarrow D \tag{1}$$

to derive transitive FDs. Note that if $B \notin \alpha$ and $D \notin \gamma$, then the derived FD also satisfies $D \notin \alpha \cup (\gamma - B)$.

Rule Normalization. For our systematic FD propagation approach, we assume the Datalog rules defining views to be in a normal form, where each rule corresponds to exactly one of the relational operators π, π', σ, $\times$, $\cup$, $\cap$, $-$, or $\bowtie$.[3] Any set of Datalog rules can be transformed into an equivalent set of normalized rules while preserving important properties like being stratifiable [1].

3.2 Representation of FDs and Normalized Rules

We assume that functional dependencies for EDB predicates are given in a relation `edb_fd`$(\mathtt{p}, \alpha, \mathtt{B}, \mathtt{ID})$. Here α and B are (sets of) column numbers of the relation `p`. The fact represents the functional dependency $\alpha \rightarrow B$ for the relation `p`. The `ID` is of type list and used to identify the dependency in later steps, e.g., in case of union. The derived functional dependencies will be represented in the same way in an IDB predicate `fd`$(\mathtt{p}, \alpha, \mathtt{B}, \mathtt{ID}')$. Here `ID'` is related to the dependency's ID where the FD is derived from for propagated FDs or to a newly created ID for FDs that arise during the propagation process.

As in normal form every rule corresponds to exactly one operator, we can refine the above defined dependency relation `dep/2` to `rel/3` by adding the respective operator. A fact `rel(p,q,op)` indicates that a relation `p` depends (positively) on `q` via an operator `op` which is one of `'projection'`, `'extension'`, `'selection'` `'product'`, `'join'`, `'negation'`, `'intersection'`, and `'union'`. We further introduce an EDB predicate `pos(head,body,pos_head,pos_body)` for storing information on how the positions of non position preserving operators (cf. Table 1) transform from rule body to head (as FDs are represented via column numbers). Remembering that each relation is defined via one operator only and that we exclude self joins for simplicity (cf. Sect. 3.1), the above defined relation `pos/4` is non-ambiguous. Finally, we have two additional EDB predicates `eq(pred,pos1,pos2)` and `const(pred,pos,val)` for information on equality conditions (e.g., X = Y or X = const) in extension and selection rules.

[2] Multivariate right sides and omitted FDs are retrievable via Armstrong's axioms.

[3] In order to simplify the FD propagation process we limit w.l.o.g. a union rule to two relations and do not allow self joins or cross products.

Table 1. Properties of FD propagation categorized by operator

Properties	π	π'	σ	$\times$	$\cup$	$\cap$	$-$	$\bowtie$
FDs are preserved	$\times$[a]	$\times$	$\times$	$\times$	–	$\times$	$\times$[b]	$\times$
Positions are preserved	–	$\times$	$\times$	–[c]	$\times$	$\times$	$\times$	–
Transitive FDs can appear	–	$\times$	$\times$	–	–	–	–	$\times$
Additional FDs from equality conditions (variables and constants)	–	$\times$	$\times$	–	–	–	–	–
Additional FDs caused by instance reduction may appear	–	–	$\times$	–	–	$\times$	$\times$	$\times$

$\times \hat{=}$ yes, $- \hat{=}$ no
[a]Those where all contained variables are maintained
[b]Those of the minuend
[c]Positions of the first factor are preserved, positions of the second factor get an offset

3.3 Propagation Rules

In this section, we present three different types of propagation rules for (a) propagating FDs to the next step, (b) introducing additional FDs arising from equality constraints, and (c) calculating transitive FDs.

Example 1. Consider the following rule set given in normal form together with two FDs $\texttt{fd}(\texttt{s}, \{1\}, 2, \texttt{ID}_1)$ and $\texttt{fd}(\texttt{t}, \{1, 2\}, 3, \texttt{ID}_2)$ for the base relations `s` and `t`:

```
p(W,Z)       ← q(W,X,Y,Z)
q(W,X,Y,Z)   ← r(W,X,Y,Z), Y=2
r(W,X,Y,Z)   ← s(W,X), t(X,Y,Z)
```

Omitting IDs, we obtain the following propagation process: First, both FDs are propagated to `r` resulting in `fd(r,{1},2,-)` and `fd(r,{2,3},4,-)` (with the appropriate column renaming for the latter FD). By transitivity we have `fd(r,{1,3},4,-)` as a combination of the two. All three FDs are propagated to `q` together with `fd(q,∅,3,-)` resulting from the equality constraint `Y=2`. Applying transitivity results in three more FDs for `q`, but only `fd(q,{1},4,-)` is propagated further to `p` as `fd(p,{1},2,-)`. The complete list of propagated FDs including IDs is given in Example 2.

Table 1 summarizes the properties of how FDs are propagated via the different relational operators which form the basis for the propagation rules. In most cases, the FDs are propagated as they are (with adjustments on the positions for π, $\times$, and $\bowtie$). If there is a single rule defining a derived relation, the source FDs transform to FDs for the new relation (restricted to the attributes in use). Union forms an exception where even common FDs are only propagated in special cases (cf. Sect. 3.4). For extensions π' and selections σ where additional FDs can occur due to equality conditions as well as for joins $\bowtie$ transitive FDs may appear so that taking the transitive closure becomes necessary. In cases where the number of tuples is reduced (i.e., σ, $\cap$, $\bowtie$, and $-$) it is possible that new FDs appear as

pos_pres
'selection'
'extension'
'negation'
'intersection'

non_pos_pres
'projection'
'product'
'join'

```
trans(R) ← base(R,_).
trans(R) ← rel(R,_,'join').
trans(R) ← eq(R,_,_).
trans(R) ← const(R,_,_).
```

Fig. 1. Position preserving (left) and non position preserving (middle) operators, and relations where transitive FDs may occur (right).

there are less tuples for which the FD constraint must be satisfied. The different propagation rules for all relational operators except union are specified in the following.

(a) Induced FDs. For direct propagation of FDs from one level to the next, we distinguish between *position preserving* and *non position preserving* operators. In the first case FDs can be directly propagated (2), whereas in the latter adjustments on the column numbers are necessary (3). The EDB predicates `pos_pres` and `non_pos_pres` comprise the respective operators as listed in Fig. 1.

$$\texttt{fd}(\texttt{P}, \alpha, \texttt{B}, \texttt{-}) \leftarrow \texttt{fd}(\texttt{Q}, \alpha, \texttt{B}, \texttt{-}), \texttt{rel}(\texttt{P}, \texttt{Q}, \texttt{op}), \texttt{pos\_pres}(\texttt{op}). \tag{2}$$

$$\begin{aligned}\texttt{fd}(\texttt{P}, \{\texttt{X}_1, \ldots, \texttt{X}_\texttt{n}\}, \texttt{Y}, \texttt{-}) \leftarrow\ & \texttt{fd}(\texttt{Q}, \{\texttt{A}_1, \ldots, \texttt{A}_\texttt{n}\}, \texttt{B}, \texttt{-}), \\ & \texttt{pos}(\texttt{P}, \texttt{Q}, \texttt{X}_1, \texttt{A}_1), \ldots, \texttt{pos}(\texttt{P}, \texttt{Q}, \texttt{X}_\texttt{n}, \texttt{A}_\texttt{n}), \texttt{pos}(\texttt{P}, \texttt{Q}, \texttt{Y}, \texttt{B}), \\ & \texttt{rel}(\texttt{P}, \texttt{Q}, \texttt{op}), \texttt{non\_pos\_pres}(\texttt{op}).\end{aligned} \tag{3}$$

(b) Additional FDs. For any equality constraint $X = Y$ we can deduce the dependencies $X \rightarrow Y$ and $Y \rightarrow X$. Similar, a constant constraint $X = c$ induces the dependency $\emptyset \rightarrow X$. That is for any fact `eq(R,pos1,pos2)` and `const(R,pos,val)` respectively we derive the following FDs:

$$\texttt{fd}(\texttt{R}, \texttt{pos1}, \texttt{pos2}, \texttt{-}) \leftarrow \texttt{eq}(\texttt{R}, \texttt{pos1}, \texttt{pos2}). \tag{4}$$

$$\texttt{fd}(\texttt{R}, \texttt{pos2}, \texttt{pos1}, \texttt{-}) \leftarrow \texttt{eq}(\texttt{R}, \texttt{pos1}, \texttt{pos2}). \tag{5}$$

$$\texttt{fd}(\texttt{R}, \emptyset, \texttt{pos}, \texttt{-}) \leftarrow \texttt{const}(\texttt{R}, \texttt{pos}, \texttt{val}). \tag{6}$$

(c) Transitive FDs. Since transitive FDs can only occur for certain operators it is sufficient to deduce them for those cases, only (cf. Table 1):

$$\begin{aligned}\texttt{fd}(\texttt{P}, \varepsilon, \texttt{D}, \texttt{-}) \leftarrow\ & \texttt{fd}(\texttt{P}, \alpha, \texttt{B}, \texttt{-}), \texttt{fd}(\texttt{X}, \gamma, \texttt{D}, \texttt{-}), \\ & \texttt{B} \in \gamma, \texttt{D} \notin \alpha, \varepsilon = \alpha \cup (\gamma - \{\texttt{B}\}), \texttt{trans}(\texttt{P}).\end{aligned} \tag{7}$$

$$\texttt{fd}(\texttt{P}, \texttt{X}, \texttt{Y}, \texttt{-}) \leftarrow \texttt{fd}(\texttt{P}, \alpha, \texttt{X}, \texttt{ID}), \texttt{fd}(\texttt{P}, \alpha, \texttt{Y}, \texttt{ID}), \texttt{trans}(\texttt{P}). \tag{8}$$

The first rule implements the extended transitivity axiom (1) and the second equates the right sides of two identical FDs (identified by matching IDs). The IDB predicate `trans/1` comprises all relations where transitive FDs may occur.

3.4 Union

In case of union $p = p_1 \cup p_2$ even common FDs are only propagated in special cases. Consider the following example of post codes. In each country, the post code uniquely identify the city associated with it. But the same post code can be used in different countries for different cities. So although we have the FD post code → city in the relations `german_post_codes` and `us_post_codes`, it is not a valid FD in the union of both. A common FD of p_1 and p_2 is only propagated to p if the domains of the FD are disjoint, or if they match on common instances. The first case can only be handled safely on schema level if constants are involved. The latter is the case if the FDs have the same origin and are propagated in a similar way. Whether two FDs have the same origin can be easily checked (e.g. using `path` from Sect. 2). This criteria is not yet enough as the FDs might have been manipulated during the propagation process (e.g., changes in the ordering, equality constraints, etc.). We employ identifiers to track changes made to certain FDs using a list structure that adopts the tree structure of [9] who represents FDs as trees with source domains as leaves and the target domain as the tree's root. As the target is already handled in the FD itself, we keep track of the source domains and transitively composed FDs, only.

At the beginning, each base FD $\alpha \rightarrow B$ gets a unique identifier ID_i. The idea is to propagate this ID together with the FD and to keep track of the modifications made to the FD. For this purpose we attach an ordered tuple, a (possibly nested) list, to the ID, i.e., $\mathrm{ID}_i[A_1, \ldots A_n]$ for $\alpha = \{A_1, \ldots A_n\}$. For the position preserving operators (that in particular do not change the FD's structure) the ID is identically propagated in (2). For the non position preserving operators the positions are updated (using a UDF) similarly to the position adjustments of the FD itself in (3). The difference is that the ID maintains an ordering and the cardinality stays invariant. For constant constraints, we set the constant value as ID in (6), equality constraints in (4), (5) and (8) get the (column number of the) left side as ID. In (7) we replace the occurrences of the column number `B` in the ID of $\mathtt{fd}(\mathtt{X}, \gamma, \mathtt{D}, \mathtt{-})$ by the ID of $\mathtt{fd}(\mathtt{X}, \alpha, \mathtt{B}, \mathtt{-})$.

Example 2. For the FD propagation in Example 1 we have the following IDs:

```
fd(s,  {1},  2,ID1[1]).
fd(t, {1,2}, 3,ID2[1,2]).

fd(r,  {1},  2,ID1[1]).
fd(r, {2,3}, 4,ID2[2,3]).
fd(r, {1,3}, 4,ID2[ID1[1],3]).
```

```
fd(q,  {1},  2,ID1[1]).
fd(q, {2,3}, 4,ID2[2,3]).
fd(q, {1,3}, 4,ID2[ID1[1],3]).
fd(q,   ∅,   3,'3').
fd(q,  {2},  4,ID2[2,'3']).
fd(q,  {1},  4,ID2[ID1[1],'3']).
fd(p,  {1},  2,ID2[ID1[1],'3']).
```

A common ID implies that the same modifications have been made to a common base FD. This means that the FD is preserved in the case of union:

$$\mathtt{fd}(\mathtt{P}, \alpha, \mathtt{B}, \mathtt{ID}) \leftarrow \mathtt{fd}(\mathtt{P}_1, \alpha, \mathtt{B}, \mathtt{ID}), \mathtt{fd}(\mathtt{P}_2, \alpha, \mathtt{B}, \mathtt{ID}), \quad (9)$$
$$\mathtt{rel}(\mathtt{P}, \mathtt{P}_1, \text{'union'}), \mathtt{rel}(\mathtt{P}, \mathtt{P}_2, \text{'union'}), \mathtt{path}(\mathtt{P}_1, \mathtt{X}), \mathtt{path}(\mathtt{P}_2, \mathtt{X}).$$

4 Conclusion

In Sect. 3.3 we introduced our propagation rules for propagating functional dependencies. To compute the set of propagated FDs these rules are simultaneously applied to the input Datalog program in normal form. The rules are based on the observations in Table 1 which can be easily verified. The propagated functional dependencies of our approach are not complete as the problem is undecidable in general. Also limited to a less expressive subset of the relational operators (e.g., restricted operator order SPC views) one has to assume the absence of finite domains to achieve completeness. Nevertheless, we are able to deal with many cases appearing in real world applications. Our FD propagation approach can be flexible extended to allow for user-defined functions in the extension operator and even recursion can be covered with some modifications.

In [9] the FD implication problem was addressed first. We provided a full declarative approach covering most cases stated in this work. In addition, we are able to cover linear recursion in a similar way as proposed by [12]. Other related approaches like the work of [7] for conditional FDs can be incorporated into our approach, too. Besides these rule-based approaches, a detailed comparison with *the chase*, an established algorithm for FD implication, is object for future research.

References

1. Behrend, A., Manthey, M.: A transformation-based approach to view updating in stratifiable deductive databases. In: FOIKS 2008, pp. 253–271 (2008)
2. Boehm, A.M., Seipel, D., Sickmann, A., Wetzka, M.: Squash: a tool for analyzing, tuning and refactoring relational database applications. In: Seipel, D., Hanus, M., Wolf, A. (eds.) INAP/WLP -2007. LNCS (LNAI), vol. 5437, pp. 82–98. Springer, Heidelberg (2009). doi:10.1007/978-3-642-00675-3_6
3. Brass, S., Goldberg, C.: Proving the safety of SQL queries. In: QSIC 2005, pp. 197–204 (2005)
4. Calì, A., Calvanese, D., De Giacomo, G., Lenzerini, M.: Data integration under integrity constraints. Inf. Syst. **29**(2), 147–163 (2004)
5. Davidson, S.B., Fan, W., Hara, C.S., Qin, J.: Propagating XML constraints to relations. In: ICDE 2003, pp. 543–554 (2003)
6. Fagin, R., Kolaitis, P.G., Popa, L., Tan, W.C.: Reverse data exchange: Coping with nulls. In: PODS 2009, pp. 23–32 (2009)
7. Fan, W., Ma, S., Hu, Y., Liu, J., Wu, Y.: Propagating functional dependencies with conditions. PVLDB **1**(1), 391–407 (2008)
8. Herrmann, K., Voigt, H., Behrend, A., Rausch, J., Lehner, W.: Living in parallel realities co-existing schema versions with a bidirectional database evolution language. In: SIGMOD 2017 (2017)
9. Klug, A.C.: Calculating constraints on relational expressions. TODS **5**(3), 260–290 (1980)
10. Maule, A., Emmerich, W., Rosenblum, D.S.: Impact analysis of database schema changes. In: ICSE 2008, pp. 451–460 (2008)

11. Milo, T., Zohar, S.: Using schema matching to simplify heterogeneous data translation. In: VLDB 1998, p. 122 (1998)
12. Paramá, J.R., Brisaboa, N.R., Penabad, M.R., Places, Á.S.: Implication of functional dependencies for recursive queries. In: Ershov Memorial Conference 2003, pp. 509–519 (2003)