A Rule-Based Approach to Analyzing Database Schema Objects with Datalog

Christiane Engels\textsuperscript{1}, Andreas Behrend\textsuperscript{1} and Stefan Brass\textsuperscript{2}

\textsuperscript{1} Rheinische Friedrich-Wilhelms-Universität Bonn, Institut für Informatik III, Römerstraße 164, D-53117 Bonn, Germany
\{engelsc, behrend\} @ cs.uni-bonn.de

\textsuperscript{2} Martin-Luther-Universität Halle-Wittenberg, Institut für Informatik, Von-Seckendorff-Platz 1, D-06099 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. The propagation of functional dependencies plays an important role in data integration and query optimization but represents an undecidable problem in general. And yet, our rule-based analysis covers all relational operators as well as linear recursive expressions in a systematic way showing the depth of analysis possible by our proposal. The analysis of functional dependencies is well-integrated in a uniform approach to analyzing dependencies between schema elements in general.

Keywords: Schema Analysis, Functional Dependencies, Dependency Propagation, Datalog

1 Introduction

The analysis of database schema elements such as tables, views, triggers, user-defined functions and constraints provides 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 \cite{5}, for detecting the consequences of schema changes \cite{18}, for versioning \cite{12}, and matching \cite{19}. In addition, the analysis of schema objects plays an important role for tuning resp. refactoring database applications \cite{4}. 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 which are sometimes provided by system tables cover only certain information of the respective schema elements. This makes it difficult for database users to understand a given schema, explain specific derivations or oversee the consequences of intended schema modifications.

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 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 [6,11,14,15,21,23] and has applications in data exchange [10], data integration [7], data cleaning [11], data transformations [8], and semantic query optimization [20].

Functional dependencies describe relationships between attributes of a database relation and are the most widely used uni-relational dependencies [9]. They arise naturally in many ways, for instance when modeling key constraints, one-to-one or one-to-many relationships. The problem of FD propagation is undecidable in the general setting and coNP-complete for many special cases [11]. Consequently, the task of finding induced FDs is rather complex and needs to be flexible in order to allow for further refinements. We show that our rule-based approach to schema analysis is well-suited for realizing techniques for FD propagation in a declarative way 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, triggers 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 transferred into SQL systems.
- In order to show the expressiveness of our approach, the implication problem for functional dependencies is investigated using our approach.

The paper is organized as follows: First, we introduce the rule-based framework for analyzing schema objects in Section 2. Afterwards, the problem of FD propagation is investigated serving as a use case in Section 3. In this section, a systematic way for deriving FD propagation rules is developed (Subsection 3.3) before the most difficult operations ‘union’ and ‘recursion’ are investigated in more detail in Subsection 3.4. Finally, we draw a conclusion in Section 4.
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 interdependencies. 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

\[
\text{CREATE VIEW } q \text{ AS SELECT } \ldots \text{ FROM } p_1, p_2, \ldots, p_n
\]

leads to the set \(\{p_1 \rightarrow q, \ldots, p_n \rightarrow q\}\) of direct dependencies where \(q\) is a derived relation and \(p_i\) denote either a derived or a base relation. These direct dependencies are typically represented by means of a predicate dependency graph which allows for analyzing indirect dependencies, too. Those indirect dependencies allow 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. Understanding the consequences of structural changes of a base table, for example, is important if a database user wants to know all view definitions potentially affected by these changes.

Various dependencies are provided by the rules and functions in a database schema such as table-to-table dependencies induced by triggers or views-to-table dependencies which can be induced by functions. A sample dependency graph is given in Figure 1 depicting dependencies between the base relations \(\{b_1, b_2, b_3\}\), the derived relations \(\{v_1, v_2, v_3\}\), the triggers \(\{t_1, t_2\}\), and the functions \(\{f_1, f_2\}\). For example, trigger \(t_2\) fires upon changes in \(b_2\) and refers in its action part to \(b_3\) whereas function \(f_2\) is called from \(v_3\) and executes operations affecting \(b_3\). The transitive closure allows for detecting indirect dependencies such as the one between \(b_2\) and \(b_3\) due to the path \(b_2 \rightarrow v_2 \rightarrow v_3 \rightarrow f_2 \rightarrow b_3\).

This analysis can be further refined by structural details (e.g., negative vs. positive dependencies as needed in approaches for update propagation) as well as by considering the syntactical components of schema objects such as col-
CREATE VIEW v3 AS
(SELECT name, AVG(sum), f1() FROM v2
GROUP BY name);

CREATE FUNCTION f1()
RETURN datetime IS
PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
time := NOW();
INSERT INTO b1 (time_log) VALUES (time);
RETURN time;
END;

Fig. 2. Translating an SQL view and function into a Datalog Meta-Program

unnamed 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 metaprogramming in Prolog which led to the famous vanilla interpreter [13]. For readability reasons we use Datalog instead of Prolog or SQL. A sample translation is given in Figure 2 where Datalog facts of the form

\[\text{dep}(\text{To}, \text{From}) \leftarrow \text{base}(\text{R}, A), \text{derived}(V, A)\]

are used (amongst others) for representing meta-information about the given view and user-defined function. Based on these facts, the analysis of schema elements can be simply realized by means of Datalog queries like

\[\text{attr.dups}(R1, R2, N) \leftarrow \text{attr}(R1, _, N), \text{attr}(R2, _, N), R1 \neq R2.\]

\[\text{idb.func.pred}(V) \leftarrow \text{derived}(V, _), \text{call}(V, _, _).\]

\[\text{base.changes}(B) \leftarrow \text{path}(B, f1), \text{base}(B, _), \text{func}(f1, _).\]

\[\text{tbl_dep}(A, B) \leftarrow \text{base}(A, _), \text{base}(B, _), \text{path}(A, F), \text{path}(F, B), \text{func}(F, _).\]

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. In doing so, 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 is missing. In contrast, the proposed Datalog program can be easily extended by user-defined rules and the respective program can be directly transferred into a given database using SQL. This way, the schema analysis becomes a natural part of a database application.

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 analyzes how FDs are propagated through the dependency graph of a database. The problem is undecidable in the general setting for arbitrary relational expressions [14] and even restricted to SC views, i.e., relational expressions allowing selection and cross product only, it turns out to be coNP-complete. 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 in order to cover the majority of practical cases. Due to the inherent complexity of the necessary reasoning process, the FD propagation problem is well-suited for showing the expressiveness of using deductive rules for schema analysis.

3.1 Preliminaries

A functional dependency \( \alpha \rightarrow B \) states that the attribute values of \( \alpha \) determine those of \( B \). The restriction to univariate right sides can be done without loss of generality as multivariate right sides can be composed using Armstrong’s composition axiom [2]. We allow \( \alpha = \emptyset \) which means that the attribute values of \( B \) are constant and restrict the represented FDs to those satisfying \( B \notin \alpha \) (omitted FDs can be retrieved via Armstrong’s augmentation axiom).

For our FD propagation rules, we employ a Datalog variant with special data types for finite, one-leveled sets (with the corresponding set operations union \( \cup \), intersection \( \cap \), set minus \( - \), as well as the check for empty sets \( \alpha \neq \emptyset \)) and finite, possibly nested lists (with comparison and manipulation functionality). We use lower case letters for predicate names and constants, capital letters for variable names and Greek letters for sets. For example, the Datalog expression

\[
p(L[1,2], \epsilon) \leftarrow b(L[1,2], \alpha, Y), c(Y, \gamma, Z), \epsilon = \alpha \cup (\gamma - \{Z\})
\]

defines a join between \( b \) and \( c \) where the first attribute of \( p \) is a list comprising two elements 1 and 2, and the second attribute of \( p \) is the union of the sets \( \alpha \) and \( \gamma \) without the value in \( Z \).

An in-depth discussion on the complexity can be found in [11].

Finite domains cause troubles as FDs can occur due to the fact of limited possible value combinations.
In our approach we use the extended transitivity axiom

\[ \alpha \rightarrow B, \gamma \rightarrow D, \ B \in \gamma, \ D \notin \alpha \Rightarrow \alpha \cup (\gamma - B) \rightarrow D \]  

(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** In order to provide a systematic approach for FD propagation, we need the input rules to be in a so-called normal form, where each rule corresponds to exactly one relational operator. Any set of Datalog rules can be transformed into an equivalent set of normalized rules while properties of the original rule set like being stratifiable are preserved [3]:

*Example 1.* The rule \( p(W,Z) \leftarrow s(W,X), t(X,Y,Z), Y=2. \) can be normalized as

\[
\begin{align*}
p(W,Z) & \leftarrow q(W,X,Y,Z). \\
q(W,X,Y,Z) & \leftarrow r(W,X,Y,Z), Y=2. \\
r(W,X,Y,Z) & \leftarrow s(W,X), t(X,Y,Z).
\end{align*}
\]

where the first rule corresponds to the projection operator \( \pi \), the second rule is a selection \( \sigma \) with constraint \( Y=2 \), and the third rule represents the join \( \bowtie \) of \( s \) and \( t \).

In the following, we assume that the Datalog rules defining views are transformed into normal form for further analysis. That is, each Datalog rule corresponds to exactly one of the following relational algebra operators (\( \{X_h\}, \{Y_i\}, \{Z_j\} \) denote pairwise disjoint variables):

**Projection** \( \pi \): \( p(X_1, \ldots, X_k) \leftarrow q(Y_1, \ldots, Y_n). \),

for \( \{X_1, \ldots, X_k\} \subseteq \{Y_1, \ldots, Y_n\} \)

**Extension** \( \pi' \): \( p(X_1, \ldots, X_k, Y_1, \ldots, Y_n) \leftarrow q(X_1, \ldots, X_k). \) with equality conditions \( Y_i = X_j \) or \( Y_i = 'a_i' \) for \( 1 \leq i \leq n \) and constants \( a_i \)

**Selection** \( \sigma \): \( p(X_1, \ldots, X_k) \leftarrow q(X_1, \ldots, X_k), \ <\text{Condition}(X_1, \ldots, X_k)> \).

**Cross product** \( \times \): \( p(X_1, \ldots, X_k) \leftarrow q(Y_1, \ldots, Y_n), \ r(Z_1, \ldots, Z_m). \)

for \( \{X_1, \ldots, X_k\} = \{Y_1, \ldots, Y_n\} \cup \{Z_1, \ldots, Z_m\} \)

**Union** \( \cup \): \( p(X_1, \ldots, X_k) \leftarrow p_i(X_1, \ldots, X_k). \) for \( 1 \leq i \leq 2 \)

**Intersection** \( \cap \): \( p(X_1, \ldots, X_k) \leftarrow q(X_1, \ldots, X_k), \ r(X_1, \ldots, X_k). \)

**Negation** \( \neg \): \( p(X_1, \ldots, X_k) \leftarrow q(X_1, \ldots, X_k), \ not \ r(X_1, \ldots, X_k). \)

**Join** \( \bowtie \): \( p(X_1, \ldots, X_k) \leftarrow q(Y_1, \ldots, Y_n), \ r(Z_1, \ldots, Z_m). \)

for \( \{X_1, \ldots, X_k\} = \{Y_1, \ldots, Y_n\} \cup \{Z_1, \ldots, Z_m\} \)

In order to simplify the FD propagation, we will not allow for self joins or cross products (i.e., \( q = r \)) which can always be achieved by applying renaming of one of the respective relations first.
3.2 Representation of FDs and Normalized Rules

We assume that functional dependencies for EDB predicates are given in a relation $edb_fd(p, \alpha, B, ID)$. Here $\alpha$ and $B$ are (sets of) column numbers of 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 $fd(p, \alpha, B, ID')$. 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 a relation $q$ via a relational operator $op$ which is one of ‘projection’, ‘extension’, ‘selection’, ‘product’, ‘join’, ‘negation’, ‘interaction’, 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 (since FDs are represented via column numbers). For the rules of Example 1 these are:

- $p(W,Z) \leftarrow q(W,X,Y,Z)$  \rightarrow  \text{pos}(p,q,1,1). \text{pos}(p,q,2,4).
- $r(W,X,Y,Z) \leftarrow s(W,X), t(X,Y,Z)$  \rightarrow  \text{pos}(r,s,1,1). \text{pos}(r,s,2,2).
  \text{pos}(r,t,2,1). \text{pos}(r,t,3,2).
  \text{pos}(r,t,4,3).

Remembering that each relation is defined via one operator only and that we exclude self joins for simplicity (cf. Section 3.1), the above defined relation $pos/4$ is non-ambiguous. Finally, we have two EDB predicates $eq(pred,pos1,pos2)$ and $const(pred,pos,val)$ for information on equality conditions (e.g., $X = Y$ or $X = const$ resp.) in extension and selection rules.

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 2.** Consider again the rule set introduced in Example 1. If we assume two FDs $fd(s,\{1\},2,ID_1)$ and $fd(t,\{1,2\},3,ID_2)$ for the base relations $s$ and $t$ we obtain the following propagation process (omitting IDs). First, both FDs are propagated to $r$ resulting in $fd(r,\{1,2\},4,-)$ 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 $fd(q,\emptyset,3,-)$ 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 3.
Properties | π | π′ | σ | × | ∪ | ∩ | − | ⋊ ⋉
--- | --- | --- | --- | --- | --- | --- | --- | ---
FDs are preserved | ×*1 | × | × | × | – | × | × | ×
Positions are preserved | – | × | × | –*3 | × | × | × | –
Transitive FDs can appear | – | × | × | – | – | – | – | ×
Additional FDs from equality conditions (variables and constants) | – | × | × | – | – | – | – | –
Additional FDs caused by instance reduction may appear | – | – | × | – | – | × | × | ×

*1: those where all contained variables are maintained
*2: those of the minuend
*3: positions of the first factor are preserved, positions of the second factor get an offset

Table 1. Properties of FD propagation categorized by operator

Table 1 summarizes the properties of how FDs are propagated via the different relational operators which form the basis for the propagation rules. (Note that in the first two rows attention has to be paid for no/–, whereas in the last three rows special attention has to be paid for yes/×.) In most cases, the FDs are propagated as they are (with adjustments on the positions for π, ×, and ⋊ ⋉). If there is a single rule defining a derived relation, the source FDs transform to FDs for the derived relation (restricted to the attributes in use). Union forms an exception where even common FDs are only propagated in special cases and is therefore treated separately (cf. Section 3.4). For extensions and selections where additional FDs can occur due to equality conditions as well as for joins transitive FDs may appear so that taking the transitive closure becomes necessary. In cases where the number of tuples is reduced (i.e., σ, ∩, ⋊ ⋉, and −) it is possible that new FDs appear as there are less tuples for which the FD constraint must be satisfied. But as we are working on schema and not on instance level, this FD had to be present in a specific part of the parent relations. The only case, where a FD is propagated to just a part of the derived relation is union.

The different propagation rules for all relational operators except union, additional FDs due to equality conditions, and transitive FDs are specified in the following. The definition and usage of IDs and how they are propagated is deferred to Section 3.4.

(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 as they are (2), whereas in the latter adjustments on the column numbers are necessary (3). The two
Fig. 3. Position preserving (left) and non position preserving (right) operators.

EDB predicates \texttt{pos\_pres} and \texttt{non\_pos\_pres} comprise the respective operators as listed in Figure 3.

\[
\text{fd}(P, \alpha, B, -) \leftarrow \text{fd}(Q, \alpha, B, -), \text{rel}(P, Q, \text{op}), \text{pos\_pres}(\text{op}). \quad (2)
\]

\[
\text{fd}(P, \{X_1, \ldots, X_n\}, Y, -) \leftarrow \text{fd}(Q, \{A_1, \ldots, A_n\}, B, -), \text{pos}(P, Q, X_1, A_1), \ldots, \text{pos}(P, Q, X_n, A_n), \text{pos}(P, Q, Y, B), \text{rel}(P, Q, \text{op}), \text{non\_pos\_pres}(\text{op}). \quad (3)
\]

**b) Additional FDs** For any equality constraint \(X = Y\) we can deduce the dependencies \(X \rightarrow Y\) and \(Y \rightarrow X\) as after the application of the constraint the values of \(X\) and \(Y\) coincide. Similar, a constant constraint \(X = \text{const}\) induces the dependency \(\emptyset \rightarrow X\). Translated to our approach that is for any fact \(\text{eq}(R, \text{pos1}, \text{pos2})\) and \(\text{const}(R, \text{pos}, \text{val})\) respectively we derive the following FDs:

\[
\text{fd}(R, \{\text{pos1}\}, \text{pos2}, -) \leftarrow \text{eq}(R, \text{pos1}, \text{pos2}). \quad (4)
\]

\[
\text{fd}(R, \{\text{pos2}\}, \text{pos1}, -) \leftarrow \text{eq}(R, \text{pos1}, \text{pos2}). \quad (5)
\]

\[
\text{fd}(R, \emptyset, \text{pos}, -) \leftarrow \text{const}(R, \text{pos}, \text{val}). \quad (6)
\]

**c) Transitive FDs** Since transitive FDs can only arise for certain operators it is sufficient to deduce transitive FDs in those cases (cf. Table 1). For the computation we use the following two rules:

\[
\text{fd}(P, \epsilon, D, -) \leftarrow \text{fd}(P, \alpha, B, -), \text{fd}(X, \gamma, D, -), \text{B} \in \gamma, \text{D} \notin \alpha, \epsilon = \alpha \cup (\gamma - \{B\}), \text{trans}(P). \quad (7)
\]

\[
\text{fd}(P, \{X\}, Y, -) \leftarrow \text{fd}(P, \alpha, X, \text{ID}), \text{fd}(P, \alpha, Y, \text{ID}), \text{trans}(P). \quad (8)
\]

The first rule implements the extended transitivity axiom (1) and the second equates the right sides of two identical FDs (identified by matching IDs) with the same left side. The IDB predicate \texttt{trans/1} comprises those relations where transitive FDs may occur. Base relations are included to start with a complete set of representatives.

\[
\text{trans}(R) \leftarrow \text{base}(R, \_).
\]

\[
\text{trans}(R) \leftarrow \text{rel}(R, \_, 'join').
\]

\[
\text{trans}(R) \leftarrow \text{eq}(R, \_, \_).
\]

\[
\text{trans}(R) \leftarrow \text{const}(R, \_, \_).
\]
3.4 Union and Recursion

In case of union \( p = p_1 \cup p_2 \) even common FDs of \( p_1 \) and \( p_2 \) are only propagated in special cases. Consider the following example of student IDs. For each university, the student ID uniquely identify the student associated with it. But the same student ID can be used by different universities for different students. So although we have the FD \( \text{student ID} \rightarrow \text{student name} \) in the relations \text{Bonn students} and \text{Cologne students}, 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 with the \text{path} relation of Section 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.). So we employ a system of identifiers to track those changes made to a certain FD. For the IDs we use a list structure that adopts the tree structure of [14] 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 transitive composed FDs, only.

At the beginning, each 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 that FD. For this purpose we attach an ordered tuple, a (possibly nested) list to the ID, i.e., \( ID_i[A_1, \ldots, A_n] \) for the above base FD with \( \alpha = \{A_1, \ldots, A_n\} \). 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 that 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 \( \text{fd}(X, \gamma, B, -) \) by the ID of \( \text{fd}(x, \alpha, B, -) \). Note that this corresponds to a column number replacement for equality constraints.

Our propagation rules including IDs are the following:

\[
\text{fd}(P,\alpha,B,ID) \leftarrow \text{fd}(Q,\alpha,B,ID), \text{rel}(P,Q,\text{op}), \text{pos_pres}(\text{op}). \tag{2'}
\]

\[
\text{fd}(P,A,B,ID) \leftarrow \text{fd}(Q,A,B,ID), \text{rel}(P,Q,\text{op}), \text{pos_pres}(\text{op}).
\]

\[
\text{fd}(P,\{X_1,\ldots,X_n\},Y,ID<X_i>) \leftarrow \text{fd}(Q,\{A_1,\ldots,A_n\},B,ID<A_i>), \tag{3'}
\]

pos(P,Q,X_1,A_1),\ldots,\text{pos}(P,Q,X_n,A_n), \text{pos}(P,Q,Y,B), \text{rel}(P,Q,\text{op}), \text{non_pos_pres}(\text{op}).
\]

\[
\text{fd}(R,\{\text{pos}\},\text{pos2},\text{pos1}) \leftarrow \text{eq}(R,\text{pos1},\text{pos2}). \tag{4'}
\]

\[
\text{fd}(R,\{\text{pos2}\},\text{pos1},\text{pos2}) \leftarrow \text{eq}(R,\text{pos1},\text{pos2}). \tag{5'}
\]

\[
\text{fd}(R,\emptyset,\text{pos},\text{val}) \leftarrow \text{const}(R,\text{pos},\text{val}). \tag{6'}
\]
Analyzing Database Schema Objects with Datalog

fd(P, ε, D, ID_ε) ← fd(P, α, B, ID_α), fd(X, γ, D, ID_γ),
B ∈ γ, D /∈ α, ε = α ∪ (γ − {B}), trans(P),
ID_ε = replace(ID_α, B, ID_α).  

fd(P, {X}, Y, X) ← fd(P, α, X, ID), fd(P, α, Y, ID), trans(P).  

(7')

(8')

Example 3. For the FD propagation in Example 2 we have the following IDs:

fd(s, {1}, 2, ID_1[1]).
fd(t, {1,2}, 3, ID_2[1,2]).
fd(r, {1}, 2, ID_1[1]).
fd(r, {2,3}, 4, ID_2[2,3]).
fd(r, {1,3}, 4, ID_2[ID_1[1],3]).
fd(p, {1}, 2, ID_2[ID_1[1], '3']).
fd(q, {1}, 2, ID_1[1]).
fd(q, {2,3}, 4, ID_2[2,3]).
fd(q, {1,3}, 4, ID_2[ID_1[1],3]).
fd(q, {2}, 4, ID_2[2, '3']).
fd(q, {1}, 4, ID_2[ID_1[1], '3']).

A common ID implies that the same modifications (not counting column number
shifts due to joins, projections and equality constraints) have been made to a
common base FD. This means that the FD is preserved in the case of union. So
the final propagation rule for union reduces to a check for similar IDs:

fd(P, α, B, ID) ← fd(P_1, α, B, ID), fd(P_2, α, B, ID),
rel(P, P_1, 'union'), rel(P, P_2, 'union'), path(P_1, X), path(P_2, X).

(9)

Common FDs resulting from equality and constant constraints are propagated
in any case. Due to the choice of IDs this case is captured in the above ID com-
parison rule. Note that there are FDs with different IDs for with the common
FD is maintained in the union. Examples of such cases which are not covered by
our approach can be found in [14].

Recursion Recursion forms another special case. Consider the following recur-
sive example modeling the ancestor relationship of a tree. We have the following
two rules recursively defining the relation p:

p(Child, Parent, Parent) ← q(Child, Parent).
p(Child, Parent, Ancestor) ← p(Child, Parent, X), q(X, Ancestor).

Or equivalently transformed into the normal form:

p(C, P, A)   ← p_1(C, P, A).
p(C, P, A)   ← p_2(C, P, A).
p_1(C, P, P) ← q(C, P).
p_2(C, P, A) ← p_3(C, P, X, A).
p_3(C, P, X, A) ← p(C, P), q(X, A).
Each vertex has a unique parent, so we assume the FD $Child \rightarrow Parent$ for $q$, i.e., $fd(q,\{1\},2,*[1])$. With the rules defined above we cannot compute any FD for $p$ although the FD $Child \rightarrow Parent$ is propagated from $q$. This is because the FD is propagated to the recursion’s base case and survives the recursion step. The rules above cannot detect this FDs as it is only propagated to $p_1$ but not derivable for $p_2$ or $p_3$.

Our solution is to take the potential FDs propagated to the recursion’s base case and feed them into the recursion step(s). If they survive the recursion step, i.e., if they are propagated with the above defined rules (2)-(9) with identical ID, then they are propagated as FD for the whole relation. Similar to the case of union it is important that the FD’s ID is maintained, otherwise the recursion’s union might destroy the FD property.

Since we limited the union rule in the definition of the normal form to two operands (cf. Section 3.1) the rule defining a linear recursive relation $P$ has only two components, which are w.l.o.g. a base case $Q$ and a recursive component $R$. We maintain this information in an EDB predicate $rec(P,Q,R)$. As potential FDs we introduce those that are propagated from the base case $Q$ to the base part of the recursive relation $P$.

$$pfd(P,\alpha,B,ID) \leftarrow fd(Q,\alpha,B,ID), rec(P,Q,R).$$

Finally after propagating them through the dependency graph, we can deduce those FDs present in the recursive relation by ID comparison:

$$fd(P,\alpha,B,ID) \leftarrow fd(Q,\alpha,B,ID), pfd(R,\alpha,B,ID), rec(P,Q,R).$$

Applied to the ancestor example this results in the following facts:

$$fd(q,1,2,*[1]). \quad pfd(p,1,2,*[1]). \quad fd(p_1,3,4,*[3]). \quad pfd(p_3,1,2,*[1]).$$
$$fd(p_1,2,3,2). \quad pfd(p,1,3,*[1]). \quad pfd(p_3,2,3). \quad pfd(p_3,3,2).$$
$$fd(p_3,1,3,*[1]). \quad fd(p_1,2,*[1]). \quad fd(p_3,2,4,*[2]). \quad fd(p_3,1,4,*[1]).$$

The FD $1 \rightarrow 2$, i.e., $Child \rightarrow Parent$, is correctly derived for relation $p$.

### 3.5 Discussion

In Section 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 and the extracted meta-data tokens. (Note that the relation $fd$ comprising the propagated functional dependencies is recursively defined.) The rules are based on the observations in Table 1 which can be easily verified. For example an equality constraint $X=Y$ introduces two FDs between the attributes $X$ and $Y$ as the attribute
values of one determine the identical attribute values of the other. Due to a new FD a transitive deduction of FDs inside the view may become possible, e.g., \( X = Y, \ Y \rightarrow Z \Rightarrow X \rightarrow Z \). If there is a functional dependency \( \alpha \rightarrow B \) for which not all attributes of \( \alpha \cup \{ B \} \) are present in the projected relation \( p \), then either one of the source domains or the target domain is missing, so the FD cannot be propagated to \( p \). If \( A \) is missing in \( p \) and \( \alpha - \{ A \} \rightarrow B \) is a valid FD in \( p \), then the FD has a matching FD in the parent relation and is propagated to \( p \) via this FD.

The propagated functional dependencies of our approach are not complete as the problem is undecidable in the general setting. 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.

The FD propagation approach can be further generalized to allow user-defined functions \( Y_i = f(\ldots, X_i, \ldots) \) in the extension operator \( \pi' \). As the EDB predicate \( \text{call}(V, I, O, F) \) already maintains the information which function \( F \) is called in a view \( V \) with input \( I \) and output \( O \), we just can use the propagation rule

\[
\text{fd}(V, I, O, F[I]) \leftarrow \text{call}(V, I, O, F).
\]

to include this case as well. After a function call, the output of the UDF is functionally determined by the input, i.e., \( I \rightarrow O \).

### 3.6 Related Work

The work of [14] was the first addressing the FD propagation problem. In his paper, Klug considers relational algebra expressions and defines a set of rules which iteratively compute the set of induced FDs. The rules are sound and – limited to restricted operator order which is as powerful as relational algebra without set difference – complete, assuming the absence of finite domains. The propagation with this approach is not complete anymore when considering more than one propagation step as in this case the restricted operator order might be violated which can cause the loss of FDs that reappear in later steps. In any rule based approach, special attention has to be paid to the union operator. In [14] this difficulty is addressed with an algorithm systematically testing possible value combinations using formal values.

The authors of [11] use the concept of conditional FDs to handle union. The idea is that the dependencies are not propagated to the whole view but are still maintained in its subsets. In the student ID example above, the IDs are still unique restricted to one university. So on the condition that the university is University of Bonn (Cologne), the FD student ID → student name can be propagated.

Besides the mentioned rule based approaches, the chase is an established algorithm for FD implication. Originally developed as lossless join test [1], the chase has been used to infer dependencies inside one relation [16] and for FD-FD implication [6]. Related to the approach of [14] the idea is to equate formal
values following and applying functional dependencies. In [21] the chase is used to deal with FD propagation in linear recursive Datalog programs.

4 Conclusion

In this paper, we have discussed a uniform framework for analyzing schema objects using deductive rules. In doing so, many interesting schema object properties as well as relationships can be systematically deduced. These form valuable information for database users. Our approach can be easily extended for refining the analysis by user-defined queries and is suitable for SQL systems due to the choice of Datalog. The expressiveness of the underlying reasoning process has been demonstrated by the analysis of induced functional dependencies. The respective FD propagation problem is known to be undecidable and intricate in detail. And yet, the proposed deductive rules allow for covering known approaches to solving the FD propagation problem proposed so far. This includes multiple propagation steps, union and linear recursion showing the depth of analysis which can be achieved for schema analysis in this way. Identifying how other FD propagation approaches can be integrated into our rule based approach as well as an experimental evaluation of our proposal is part of further research.

References

1. A. V. Aho, C. Beeri, and J. D. Ullman. The Theory of Joins in Relational Databases. TODS 4(3): 297–314, 1979.
2. W. W. Armstrong. Dependency Structures of Data Base Relationships. IFIP Congress 1974, pp. 580–583.
3. A. Behrend and M. Manthey. A Transformation-Based Approach to View Updating in Stratifiable Deductive Databases. FOIKS 2008, pp. 253–271.
4. A. M. Boehm, D. Seipel, A. Sickmann, and M. Wetzka. Squash: A Tool for Analyzing, Tuning and Refactoring Relational Database Applications. INAP/WLP 2007, pp. 82–98.
5. S. Brass and C. Goldberg. Proving the Safety of SQL Queries. QSIC 2005, pp. 197–204.
6. N. R. Brisaboa, A. González, H. J. Hernández, and J. R. Paramá. The Chase of Datalog Programs. BNCOD 1998, pp. 165–166.
7. A. Calì, D. Calvanese, G. De Giacomo, and M. Lenzerini. Data Integration Under Integrity Constraints. Inf. Syst. 29(2): 147–163, 2004.
8. S. B. Davidson, W. Fan, C. S. Hara, and J. Qin. Propagating XML Constraints to Relations. ICDE 2003, pp. 543–554.
9. C. Delobel and M. Adiba. Relational Database Systems. Elsevier North-Holland, 1985.
10. R. Fagin, P. G. Kolaitis, L. Popa, and W. C. Tan. Reverse Data Exchange: Coping with Nulls. PODS 2009, pp. 23–32.
11. W. Fan, S. Ma, Y. Hu, J. Liu, and Y. Wu. Propagating Functional Dependencies with Conditions. PVLDB 1(1): 391–407, 2008.
12. K. Herrmann, H. Voigt, A. Behrend, J. Rausch and W. Lehner. Living in Parallel Realities - Co-Existung Schema Versions with a Bidirectional Database Evolution Language. SIGMOD 2017, pp. 1101–1116.
13. P. M. Hill and J. W. Lloyd. Analysis of meta-programs. Meta-Programming in Logic Programming, pp. 23–51, 1989.
14. A. C. Klug. Calculating Constraints on Relational Expressions. TODS 5(3): 260–290, 1980.
15. A. C. Klug and R. Price. Determining View Dependencies Using Tableaux. TODS 7(3): 361–380, 1982.
16. D. Maier, A. O. Mendelzon, and Y. Sagiv. Testing Implications of Data Dependencies. TODS 4(4):455–469, 1979.
17. H. Mannila and K. Räihä. On the Complexity of Inferring Functional Dependencies. Discrete Applied Mathematics 40(2): 237–243, 1992.
18. A. Maule, W. Emmerich, and D. S. Rosenblum. Impact Analysis of Database Schema Changes. ICSE 2008, pp. 451–460.
19. T. Milo and S. Zohar. Using Schema Matching to Simplify Heterogeneous Data Translation. VLDB 1998, pp. 122–133.
20. J. R. Paramá, N. R. Brisaboa, M. R. Penabad, and A. S. Places. A Semantic Query Optimization Approach to Optimize Linear Datalog Programs. ADBIS 2002, pp. 277–290.
21. J. R. Paramá, N. R. Brisaboa, M. R. Penabad, and A. S. Places. Implication of Functional Dependencies for Recursive Queries. Ershov Memorial Conference 2003, pp. 509–519.
22. J. Paredaens and D. Janssens. Decompositions of relations: A comprehensive approach. Advances in Database Theory 1979, pp. 73–100.
23. K. Wang and L. Yuan. Preservation of Integrity Constraints in Definite DATALOG Programs. Information Processing Letters 44(4): 185–193, 1992.