

On Computing Transitive Closures in *MatBase* (Does HRH Prince Charles of Wales Also Have Cuman Blood?)

Christian Mancas
 DATASIS ProSoft srl
 Bucharest, Romania
christian.mancas@gmail.com

Abstract — This paper presents how transitive closures and their instantiations for all possible types of db interesting dyadic relations, self-functions, and function products are computed by the *MatBase Datalog* subsystem. Moreover, it is proved that the corresponding algorithms are linear, solid, complete, and optimal.

Keywords — dyadic relation, self-function, function product, transitive closure, Datalog, fixpoint semantics, Relational Algebra, (Elementary) Mathematical Data Model, *MatBase*

Abbreviations — card – cardinal, db – database, DKNF – Domain Key Normal Form, e.g. – for example, EMDM – (Elementary) Mathematical Data Model, E-RDM – Entity-Relationship Data Model, HRH – His Royal Highness, i.e. – that is, kb – knowledge base, q.e.d. – what was to be proved, RA – relational algebra, RDBMS – Relational DataBase Management System, RDM – Relational Data Model

I. INTRODUCTION

In everyday life we often talk about parent-child relationships: binary relations on a set of people, dead or alive. For example, in a *REIGNS* database, we might have a *RULERS* table having (among other columns as well) a surrogate autonumber primary key x , a key (unique) text $Name$ one, and two foreign keys (both referencing x) $Father$ and $Mother$ (for storing corresponding parents) as in Fig. 1 (also presenting a partial instance from Romania's history between the XIIIth and XVth).

We are often also interested in ancestor-descendant relations. Although the latter relation can be obtained from the former (hence, it is redundant in that sense), ancestor-descendant relations give us necessary information much more directly.

For example, some years ago, HRH Prince Charles of Wales declared that he is a descendant of Vlad Țepeș (“the Impaler”) “Dracula” (apparently, according to [24], through his stepbrother Vlad Călugărul (“the Monk”), who was one of the ancestors of Queen Elizabeth II, HRH's mother); does this mean, for example, that he is also a descendant of the founders of the two Romanian medieval states, Basarab I for Walachia and Bogdan I for Moldavia?

RULERS (\underline{x} , $Name$) $Father \subset x$, $Mother \subset x$

x	$Name$	$Father$	$Mother$
36	Vlad Țepeș (“Dracula”)	35	493
33	Vlad Călugărul	35	485
23	Radu cel Frumos	35	15
35	Vlad Dracul	19	255
15	Vasilisa Mușat	42	
19	Mircea cel Bătrân	26	248
26	Radu I	22	58
22	Nicolae Alexandru	5	243
5	Basarab I	241	240
241	Thocomerius		
240	Ana	239	
239	Bărbat		
42	Alexandru cel Bun	71	44
71	Roman I	75	57
75	Ștefan I Mușat	218	73
218	Ștefan	46	
46	Bogdan I		

Fig. 1. *Walachia and Moldavia first rulers and some of their parents*

This ancestor-descendant relation relates two people if there is a sequence of parent-child relations from one to the other; obviously, it includes the parent-child relation as a subset. The ancestor-descendant relation is an example of the closure of a relation, in particular the *transitive closure* of the parent-child one.

A. Transitive closures

Recall that a relation R' is the *transitive closure* of a relation R if and only if

- (1) R' is transitive,
- (2) $R \subseteq R'$, and
- (3) for any relation R'' , if $R \subseteq R''$ and R'' is transitive, then $R' \subseteq R''$, that is, R' is the smallest relation that satisfies (1) and (2).

Note that the *digraph of the transitive closure* of a relation is obtained from the digraph of the relation by adding for each directed path the edge that shunts the path, if one is not already there.

Obviously, both above self-mappings $Father : RULERS \rightarrow RULERS$ and $Mother : RULERS \rightarrow RULERS$ are parent-child relations.

Among others, there is a well-known Roy-Floyd-Warshall algorithm [23] for computing transitive closures; based on the relation's adjacency matrix, this elegant algorithm is, however, practical only for small amounts of data, not for db size table instances (as their closures generally cannot fit into memory,

even if the base corresponding relations can: e.g., the actual instance of the *RULERS* table only for Dacia, all 3 former Romania kingdoms, and Romania has 650 lines; its transitive closure for both *Father* and *Mother* relationships has 6377 lines and only few daughter data has been added yet to it).

As known since 1979 [21], neither relational algebra (RA), nor relational calculus cannot compute transitive closures; although, lately, most of the commercially available RDBMSs extended their SQLs to compute it, however, the appeal to *Datalog*→ (and other similar paradigms) for solving this problem is still strong, as they are more powerful and elegant.

Let us denote by $G_f = \{ \langle x, f(x) \rangle \mid x \in R \}$ the graph of any self-function $f : R \rightarrow R$; we denote (by notational abuse) its transitive closure G_f^* with f^* . For example, to compute *Father*^{*}, the transitive closure of *Father*, the following *Datalog* program [16] can be used:

$$\begin{aligned} \text{Father}^*(x,y) &\leftarrow \text{RULERS}(x,y) \\ \text{Father}^*(x,y) &\leftarrow \text{Father}^*(x,z), \text{RULERS}(z,y) \end{aligned}$$

Fig. 2. *Datalog* program for computing *Father*'s transitive closure

Similarly, to compute *Mother*^{*}, the transitive closure of *Mother*, the following *Datalog* program can be used:

$$\begin{aligned} \text{Mother}^*(x,v) &\leftarrow \text{RULERS}(x,v) \\ \text{Mother}^*(x,v) &\leftarrow \text{Mother}^*(x,z), \text{RULERS}(z,v) \end{aligned}$$

Fig. 3. *Datalog* program for computing *Mother*'s transitive closure

Note that, in both of them, the first rule is the equivalent of (2) above (i.e. the transitive closure of a relation includes that relation), whereas the second is the equivalent of (3) above (i.e. if somebody has x as an ancestor, then x 's father/mother is also an ancestor of that somebody).

Computing the transitive closure of only one element (e.g. "Dracula", i.e. 36 in Fig. 1 above) can be done through *instantiating* the corresponding program for it: Fig. 4 shows an instantiation of the program in Fig. 2, whereas Fig. 5 shows the same one for the *Datalog* program in Fig. 3.

Unfortunately, neither of these two programs, nor their instantiations can answer, for example, the question on whether HRH Prince Charles of Wales' ancestors also Cuman blood have (for the fact that the founder of the Wallachian Kingdom was a Cuman see, for example, [10]), as Romanian dynasties too have from time to time survived only by females and, of course, they interrelated to each other through marriages. For example, in Fig. 1, "Dracula"'s stepmother, Vasilisa Mușat, was one of the links between some of his brothers and sisters Walachian (from their father) and Moldavian descentance.

$$\begin{aligned} \text{Father}^*(36,y) &\leftarrow \text{RULERS}(36,y) \\ \text{Father}^*(36,y) &\leftarrow \text{Father}^*(36,z), \text{RULERS}(z,y) \end{aligned}$$

Fig. 4. *Datalog* program instantiation for computing *Father*'s transitive closure of ruler 36 from Fig. 1

$$\begin{aligned} \text{Mother}^*(36,v) &\leftarrow \text{RULERS}(36,v) \\ \text{Mother}^*(36,v) &\leftarrow \text{Mother}^*(36,z), \text{RULERS}(z,v) \end{aligned}$$

Fig. 5. *Datalog* program instantiation for computing *Mother*'s transitive closure of ruler 36 from Fig. 1

Obviously, what should be computed for correctly answering such queries is the transitive closure of the function product $(\text{Father} \bullet \text{Mother})^*$, or, at least, the transitive closure, according to this product, of the corresponding person (in this case, "Dracula"), i.e. the corresponding instantiation of the transitive closure of the function product $(\text{Father} \bullet \text{Mother})^*$.

B. Related work

Lot of work was published on computing transitive closures (e.g. [2, 3, 4, 9, 11, 12, 17, 18, 19, 20, 22]), but none, to our knowledge, for function products.

Datalog engines are behind specialized db systems such as LDL [7], *MatBase* [14, 16], and Intellidimension's database for the semantic web [25]. Moreover, some widely used database systems include ideas and algorithms developed for *Datalog*. For example, the SQL:1999 standard includes recursive queries, and the *Magic Sets* algorithm (initially developed for the faster evaluation of *Datalog* queries) is implemented in IBM's DB2.

C. MatBase

MatBase [14, 16] is a prototype intelligent db and kb management system based on both the (Elementary) Mathematical Data Model (EMDM) [13, 16], Entity-Relationship Data Model (E-RDM) [6, 15], Relational Data Model (RDM) [1, 8, 15], and *Datalog*→ [1, 16], developed by the author in two versions: one in MS Access (mainly for use in university labs) and (a commercial) one in MS .net C# and SQL Server.

D. Paper outline

Next Section presents first how *MatBase* computes transitive closures of dyadic relations, including the syntax-directed algorithms for translating *Datalog* programs to Relational Algebra (RA) equation systems and for computing the least fixpoint of recursive RA equations. Then, it continues with the algorithms for computing transitive closures for function products (first, of arity 2, then, of any arity). Section III deals with *MatBase*'s algorithms for computing transitive closures of function products' instance elements. The paper ends with conclusion and references.

II. COMPUTING FUNCTION PRODUCT CLOSURES

Through its (E)MDM interface, *MatBase* users can define, among many others, any number of function products; obviously, all members of such a product should have a same domain. If, moreover, all their codomains are the same as (or included into) their common domain, then, like for any other dyadic relation, users can also ask for computation of their closures (be them reflexive, symmetric, transitive, or any combinations between them).

A. Computing transitive closures of dyadic relations

Although there are other faster methods for computing transitive closures, *MatBase* is using the semi-naïve implementation of the least fixpoint semantics for RA equations [1, 16], which is fast enough (for instance, computing the 6377 lines of $(Father \bullet Mother)^*$ above only takes under 1 second on a current "standard" notebook).

This approach is first translating each inference rule of a *Datalog* program into a *RA disequation* (i.e. inclusion relationship) by using a syntax-directed algorithm; then, by applying the closed world assumption (i.e. we are only interested in those ground facts that are a consequence of the corresponding *Datalog* program), all such disequations having same head (i.e. left-hand side intentional predicate) are collapsed into a single *RA equation*, having same head and as body (i.e. right-hand side expression) the union of all involved disequations' bodies, thus obtaining a corresponding *RA equation system*; such systems are then solved first by using substitution (just like for numbers algebra equation systems) and then, as equations may be *recursive* (i.e. containing, for example, the head intensional predicate also in the body), each such equation is solved by using the *least fixpoint computational semantics* (i.e. computing the smallest relation instance that satisfies the equation).

A.1. Syntax-directed translation algorithm from *Datalog* to RA

Without entering into details, for example, if a query p is defined on attribute A and a relation r on attributes B and C , using the natural correspondence between positional and non-positional notations, a *Datalog* inference rule of the form $p(x) \leftarrow p(y), r(x, y)$ is translated by this algorithm into the disequation

$p \supseteq \rho_{A \leftarrow B}(\pi_B(\rho_{A=C} r))$, where ρ , π , and \bowtie are the relational algebra (RA) *renaming*, *projection*, and *join* operators, respectively.

Generally, given a rule $p(x_1, \dots, x_n) \leftarrow q_1(y_1, \dots, y_{k1}), \dots, q_m(y_1, \dots, y_{km})$, this algorithm translates it into the disequation $P \supseteq E(Q_1, \dots, Q_m)$, where P , Q_1, \dots, Q_m are the query and fundamental relations that correspond to predicates p, q_1, \dots, q_m , respectively. By collapsing all such disequations having P as left-

hand side, a RA equation of the type $P = E_1(Q_1, \dots, Q_{m1}) \cup \dots \cup E_l(Q_1, \dots, Q_{ml})$ is obtained.

For instance, the two rules of the program in Fig. 2 above are translated into the following two RA disequations:

$$\begin{aligned} & Father^* \supseteq \rho_{Descendant, Ancestor \leftarrow x, Father}(\pi_x, \\ & Father(RULERS)) \\ & \bowtie_{Ancestor = x \pi_x, Father(RULERS)} \\ & Father^* \supseteq \rho_{Ancestor \leftarrow Father}(\pi_{Descendant, Father}(Father^* \\ & \bowtie_{Ancestor = x \pi_x, Father(RULERS)})) \end{aligned}$$

Fig. 6. RA disequations corresponding to the *Datalog* program in Fig. 2

These disequations are then collapsed into the following RA recursive equation:

$$\begin{aligned} & Father^* = \rho_{Descendant, Ancestor \leftarrow x, Father}(\pi_x, \\ & Father(RULERS)) \cup \\ & \bowtie_{Ancestor = x \pi_x, Father(RULERS)} \\ & \rho_{Ancestor \leftarrow Father}(\pi_{Descendant, Father}(Father^* \\ & \bowtie_{Ancestor = x \pi_x, Father(RULERS)})) \end{aligned}$$

Fig. 7. Recursive RA equation corresponding to the disequations in Fig. 6

A.2. Computing the least fixpoint of RA recursive equations

It can be shown that RA equations obtained as in II.A.1 from *Datalog* programs always have a fixpoint (trivially, as db instances are finite). This fixpoint is obtained as follows: from every RA recursive equation, a family of recurrent ones is obtained, where, by definition, P_0 is the empty set, for any query P , for every natural $j \geq 0$; by definition, the least fixpoint of P is the first P_j in the sequence P_0, P_1, \dots such that $P_j = P_{j+1}$.

For instance, the RA equation in Fig. 7 is transformed into the family of recurrent ones shown in Fig. 8:

$$\begin{aligned} & Father^*_{j+1} = \rho_{Descendant, Ancestor \leftarrow x, Father}(\pi_x, \\ & Father(RULERS)) \cup \\ & \rho_{Ancestor \leftarrow Father}(\pi_{Descendant, Father}(Father^*_j \bowtie \\ & \bowtie_{Ancestor = x \pi_x, Father(RULERS)})) \end{aligned}$$

Fig. 8. Recurrent RA equations corresponding to the equation in Fig. 7

Obviously, $Father^*_1 = \rho_{Descendant, Ancestor \leftarrow x, Father}(\pi_x, Father(RULERS))$, as joining anything with the empty set always yields the empty set; so, if *RULERS* instance were the one in Fig. 1, $Father^*_1$ instance would also have 17 lines, the ones in Fig. 9.

Then, $Father^*_2 = \rho_{Descendant, Ancestor \leftarrow x, Father}(\pi_x, Father(RULERS)) \cup \rho_{Ancestor \leftarrow Father}(\pi_{Descendant, Father}(Father^*_1 \bowtie_{Ancestor = x \pi_x, Father(RULERS)}))$, so that $Father^*_2$ contains 14 more lines (the bottom ones in Fig. 10).

$Father^*_1$

Descendant	Ancestor
36	35
33	35
23	35
35	19
15	42
19	26
26	22
22	5
5	241
241	
240	239
239	
42	71
71	75
75	218
218	46
46	

Fig. 9. First approximation of $Father^*$'s fixpoint for $RULERS$ instance in Fig. 1

$Father^*_2$

Descendant	Ancestor
36	35
33	35
23	35
35	19
15	42
19	26
26	22
22	5
5	241
241	
240	239
239	
42	71
71	75
75	218
218	46
46	
36	19
33	19
23	19
35	26
15	71
19	22
26	5
22	241
5	
240	
42	75
71	218
75	46
218	

Fig. 10. Second approximation of $Father^*$'s fixpoint for $RULERS$ instance in Fig. 1

Note that the first operand of this union asks for duplication of all existing lines, which, obviously, must be rejected.

Also note that, in fact, if user does not ask explicitly the contrary, *MatBase* eliminates null ancestors (which, for the dynasties founders may be interesting, but for all others are not: as such, in the above instance, tuples <5,>, <240,>, and <218,> are not generated, so only 11 new lines are added in this step), generating in fact as second rule (in *Datalog*–) of the program from Fig. 2:

$$Father^*(x,y) \leftarrow Father^*(x,z), RULERS(z,y), \neg IsNull(y)$$

instead, which yields the following disequation (see Fig. 6, where σ is the selection RA operator):

$$Father^* \supseteq \rho_{Ancestor \leftarrow Father(\pi_{Descendant, Father}(Father^*_{Ancestor = x} \pi_x, Father(\sigma_{NOT ISNULL(Father)}(RULERS))))}$$

Fig. 11 shows $Father^*_3$'s instance, obtained this time (as always from now on) according to this enhanced second rule.

It should be noted that the 9 new rows that were added in this step (at the bottom of the table) were generated only from the 11 ones obtained in the previous one; the other lines (i.e. those originally coming from *RULERS* in the first step) re-generated these existing 11 ones and have to be rejected as duplicates. Moreover, just as in the previous step, attempts to re-duplicate the first 17 lines are again asked by the first operand of the union operator, but they must be rejected once more.

In fact, based on these facts, *MatBase* is not even generating duplicates ever, as, in any step, it joins *RULERS* with only those lines of $Father^*$ that were added in the previous step (by adding a column *Level* to $Father^*$ for also storing for each pair <descendant, ancestor> its depth level, which is also an interesting information *per se*).

The third iteration adds 7 new lines, the fourth one another 5, the fifth – other 3, and, finally, the sixth none: the process stops as $Father^*_5$ has just been identified as being the least fixpoint (hence, the solution) of $Father^*$ (see its instance in Fig. 12). From these 42 lines it is immediately computable that Vlad Călugărul (“the Monk”, id 33) is a descendent of Basarab I (id 5), the founder of Wallachia, but that he is not descending on his father side from Bogdan I (id 46), the founder of Moldavia (as no pair <33, 46> exists in this transitive closure).

Moreover, as the latest allegations of historians (see, for instance, [10]) that Basarab I's father, Thocomerius (id 241), was a Cuman are most probably true, and as Vlad Călugărul (“the Monk”) is also a descendent of Thocomerius (see the tuple <33, 241> from $Father^*$ in Fig. 12), then in HRH Charles of Wales blood there are also Cuman reminiscences.

$Father^*_3$

Descendan t	Ancesto r
36	35
33	35
23	35
35	19
15	42
19	26
26	22
22	5
5	241
241	
240	239
239	
42	71
71	75
75	218
218	46
46	
36	19
33	19
23	19
35	26
15	71
19	22
26	5
22	241
42	75
71	218
75	46
36	26
33	26
23	26
35	22
15	75
19	5
26	241
42	218
71	46

Fig. 11. Third approximation of $Father^*$'s fixpoint for RULERS instance in Fig. 1, excluding nulls (except for dynasties' founders)

Generally, *MatBase* algorithm (presented here in a pseudo-code embedding SQL [15, 16] –invoked through function *execute*–, where // introduces comments, & is the string concatenation operator and *domain*, *codomain*, *error*, *existsTable*, and *iif* are librarian functions performing obvious tasks: for instance, the result of *iif(cond, T, F)* is *T* when *cond* is *true* and *F* otherwise) for computing the transitive closure of any relation is the one presented in Fig. 13.

Please note that in *MatBase* all tables are stored in a very restrictive variant of the DKNF [1, 15], with every table having an integer surrogate primary key – which, obviously, stands for the *x* of all other columns of the table when regarded as functions defined on their table instance– and any foreign key referencing

only the corresponding primary key; this is why both a_1 and a_2 are always integers.

$$Father^* = Father^*_5 (= Father^*_6 = Father^*_7 = \dots)$$

Descendant	Ancestor
36	35
33	35
23	35
35	19
15	42
19	26
26	22
22	5
5	241
241	
240	239
239	
42	71
71	75
75	218
218	46
46	
36	19
33	35
23	19
35	26
15	71
19	22
26	5
22	241
42	75
71	218
75	46
36	26
33	26
23	26
35	22
15	75
19	5
26	241
42	218
71	46
36	22
33	22
23	22
35	5
15	218
19	241
42	46
36	5
33	5
23	5
35	241
15	46
36	241
33	241
23	241

Fig. 12. $Father^*$ fixpoint for RULERS instance in Fig. 1

For “deciphering” the computed transitive closure instance, two inner joins of this result with two instances of the corresponding “deciphering” table on these columns with the corresponding surrogate key are all that’s needed; for instance:

SELECT RULERS.Name AS Descendant,

FATHERS.Name AS Ancestor
 FROM (RULERS INNER JOIN Father*
 ON RULERS.[x]=Father*.Descendant)
 INNER JOIN RULERS AS FATHERS
 ON Father*.Ancestor=FATHERS.[x]);

Algorithm computeTransClosure

Input: c_1, c_2 – the two columns of a table R storing the desired dyadic relation’s graph instance;
transClosure, a_1, a_2 – the names of the desired table (distinct in the db) and its two columns for storing the result;
nulls? – 0, if no null values are desired in c_2 , or
 1, if no null values are desired in c_2 except for those in R (which is the default value), or
 2, if all null values are desired in c_2 ;

Output: table *transClosure* instance, storing the corresponding transitive closure;

Strategy: $R = \text{domain}(c_1)$;

if $R \neq \text{domain}(c_2)$ or $\text{codomain}(c_1) \neq \text{INT}$ or $\text{codomain}(c_2) \neq \text{INT}$ *then*
 return error(“impossible to compute transitive closure: c_1 and c_2 are either not columns of a same table or have incompatible data types!”);

if exists Table(*transClosure*) *then execute* (“DELETE FROM “ & *transClosure*)

else execute(“CREATE TABLE “ & *transClosure* & “([Level] INT, “ & a_1 & “ INT, “ & a_2 & “ INT);”);

oldcard = 0; // *transClosure* is empty

execute(“INSERT INTO “ & *transClosure* & “ SELECT 1 AS [Level], “ & c_1 & “ , “ & c_2 & “ FROM “ & R &
 iif(*nulls?*=0, “ WHERE “ & c_2 & “ NOT IS NULL ”,)); // initialize result with <“son”, “father”>
 pairs

level = 2; // next step will add second level “ancestors”

card = *execute*(“SELECT Count(*) FROM “ & *transClosure*);

while *card* \neq *oldcard*

oldcard = *card*; // prevent infinite looping

execute(“INSERT INTO “ & *transClosure* & “ SELECT “ & *level* & “ AS [Level], “ & *transClosure* & “.” & a_1 &
 “ , “ & R & “.” & c_2 & “ FROM “ & R & “ INNER JOIN “ & *transClosure* & “ ON “ & *transClosure* & “.” & a_2 &
 “=“ & R & “.” & c_1 & “ WHERE [Level]=“ & *level* – 1 & iif(*nulls?*=1, “ AND “ & R & “.” & c_2 &
 “ NOT IS NULL ”,));

card = *execute*(“SELECT Count(*) FROM “ & *transClosure*);

level = *level* + 1; // prepare next level “ancestors”

end while;

end algorithm computeTransClosure;

Fig. 13. MatBase algorithm for computing dyadic relations’ transitive closures

Obviously, the result showed in Fig. 12 is obtainable by the following call to this method: *computeTransClosure*($[x]$, *Father*, *Father**, *Descendant*, *Ancestor*,). Fig. 14 presents the result of a call *computeTransClosure*($[x]$, *Mother*, *Mother**, *Descendant*, *Ancestor*, 0), which, obviously, is *Mother** (without any null ancestors). It is obvious that $Father^* \cup Mother^*$ does not contain the pair <23, 46> either, although, by his mother, Radu cel Frumos (id 23), one of “Dracula”’s stepbrothers, also

descends from Bogdan I (id 46), the Moldavia’s founder.

Naturally, this method can be used for computing the transitive closure of any dyadic relation; for example, in a *FOOTBALL_CHAMPIONSHIP* db, we might want to compute the transitive closure of a relation *MATCHES* for its two columns *Host* and *Visitor* (both referencing the surrogate primary key $[x]$ of a table *FOOTBALL_CLUBS*); trivially (note that

both being in fact canonical projections of *MATCHES*, neither *Host* nor *Visitor* should accept nulls), this can be computed by the call: *computeTransClosure(Host, Visitor, MATCHES*, Host, Visitor, 0)*.

$$Mother^* = Mother^*_1 (= Mother^*_2 = Mother^*_3 = \dots)$$

Descendant	Ancestor
36	493
33	485
23	15
35	255
19	248
26	58
22	243
5	240
42	44
71	57
75	73

Fig. 14. *Mother**'s fixpoint for *RULERS* instance in Fig. 1 (without nulls)

Moreover, obviously, this method can compute the transitive closure of any function product $f \bullet g : R \rightarrow INT \times INT$; for example, in a *RAILROADS* db, the call *computeTransClosure(DepartureStation, DestinationStation, CONNECTIONS*, Departure, Destination, 0)* would compute the transitive closure of the product of the columns *DepartureStation* and *DestinationStation* (both of them referencing the surrogate primary key $[x]$ of a table *STATIONS* and not accepting nulls) of a table *LINES*.

Theorem 1: Algorithm *computeTransClosure* from Fig. 13 has the following four properties:

- (i) it is linear in the longest path in the digraph of the input relation
- (ii) it is sound (i.e. it is not generating anything else but elements of the transitive closure of the input relation)
- (iii) it is complete (i.e. it generates all elements of the transitive closure of the input relation)
- (iv) it is optimal (i.e. it computes the transitive closure of the input relation in the least number of steps possible)

Proof:

(i) Trivial, as it has only one finite loop (hence, it never loops infinitely) depending on the longest path in the digraph of the input relation (trivially, as db instances are finite, any such length is finite).

(ii) Obviously, the initial step only adds the input relation; then, in each iteration of the loop, for any pair $\langle x, y \rangle$ in the current result approximation and $\langle y, z \rangle$ in the input relation, it is only added the pair $\langle x, z \rangle$, according to the transitivity rule. Moreover, trivially, attempts to call the corresponding method with wrong input parameters (i.e. columns not of the

same table or not referencing both a same table) are rejected.

(iii) Obviously, the loop is executed up until no further transitively computable pairs may be added to the result: the first statement of the loop makes sure that variable *oldcard* is storing the current result cardinal, the last but one one is updating variable *card* value to the cardinal of the result after adding current iteration elements, and the *while* statement condition ensures that the process repeats only as long as these two values are not equal (i.e. as long as the previous iteration was adding at least one new element to the result).

(iv) Obviously, as soon as the previous loop iteration did not add any new elements to the result, the process stops (i.e. the algorithm only computes the first two fixpoints, which is the minimum possible in order to discover the least fixpoint). Moreover, the algorithm never generates duplicates on a same level (as it joins to the input relation only the current result elements that were added in the previous iteration), which is minimizing disk accesses for both reading and writing operations. *q.e.d.*

B. Computing transitive closures for function products

Let $f : R \rightarrow R$ be any self-function defined on and taking values into some set R . By definition, for any (generally other, but not necessarily distinct) $g : R \rightarrow R$, we define $(f \bullet g)^* = (G_f \cup G_g)^*$.

Proposition 1: $f^* \cup g^* \subseteq (f \bullet g)^*$

Proof: let us assume that there is a pair $\langle a, b \rangle \in f^* \cup g^*$, which does not belong to $(f \bullet g)^*$; then, it either belongs to f^* or/and to g^* ; if it belongs to f^* (i.e. G_f^*), then it should belong to $(f \bullet g)^*$ too, even if G_g were the empty set; if it belongs to g^* (i.e. G_g^*), then it should belong to $(f \bullet g)^*$ too, even if G_f were the empty set; consequently, the assumption that it does not belong to $(f \bullet g)^*$ too was absurd *q.e.d.*

As we should expect, generally, $(f \bullet g)^* \not\subseteq f^* \cup g^*$, as we will see, for instance, with the $\langle 23, 46 \rangle$ element, which belongs to $(Father \bullet Mother)^*$ (Fig. 18 below), although it does not belong to $Father^* \cup Mother^*$ (see Fig. 12 and 14).

Proposition 2: $(f \bullet g)^* \not\subseteq f^* \cup g^*$

Proof: see the $\langle 23, 46 \rangle$ counterexample in Fig. 18 (as compared to Fig. 12 and 14); *alternative proof:* (see Fig. 18) $card((Father \bullet Mother)^*) = 72 > 53 = 42 + 11 = card(Father^*)$ (see Fig. 12) + $card(Mother^*)$ (see Fig. 14). *q.e.d.*

B.1. Computing transitive closures for self-function products of arity 2

MatBase is computing self-function products' transitive closures with this definition, starting with the generation of the following *Datalog* program type

(with slight variations on nulls, depending on the user's request and preserving notations used in Fig. 13 and the ones above):

$$\begin{aligned} \text{transClosure}(a_1, a_2) &\leftarrow R(c_1, f) [, \neg \text{IsNull}(f)] \\ \text{transClosure}(a_1, a_2) &\leftarrow R(c_1, g) [, \neg \text{IsNull}(g)] \\ \text{transClosure}(a_1, a_2) &\leftarrow \text{transClosure}(a_1, x), R(x, f) \\ &[, \neg \text{IsNull}(f)] \\ \text{transClosure}(a_1, a_2) &\leftarrow \text{transClosure}(a_1, x), R(x, g) \\ &[, \neg \text{IsNull}(g)] \end{aligned}$$

Fig. 15. MatBase Datalog \rightarrow generic program for computing $(f \bullet g)^*$

Algorithm computeBinarySelfProductTransClosure

Input. x, f, g – columns of a table R storing the desired self-function product's graph instance;

$\text{transClosure}, a_1, a_2$ – the distinct names of the desired output table and its two columns for storing the result;

nulls? – a pair of the type $\langle 0 \text{ or } 1 \text{ or } 2, 0 \text{ or } 1 \text{ or } 2 \rangle$, where the first element is describing user request for nulls processing for f , while the second one is for g (using same codes meaning as in Fig. 13);

Output. table transClosure instance, storing the transitive closure of $f \bullet g$;

Strategy. $R = \text{domain}(x); S = \text{codomain}(f)$;

if $R \neq \text{domain}(f)$ or $R \neq \text{domain}(g)$ or $S \neq \text{codomain}(g)$ then return error ("impossible to compute transitive closure: either x, f , and g are not columns of a same table or f and g are not referencing a same table!");

if existsTable(transClosure) then execute ("DELETE FROM " & transClosure)

else execute ("CREATE TABLE " & transClosure & "([Level] INT, " & a_1 & " INT, " & a_2 & " INT);");

oldcard = 0; // transClosure is empty

execute ("INSERT INTO " & transClosure & " SELECT 1 AS [Level], " & x & ", " & f & " FROM " & R &

 iif($\text{nulls?}[1]=0$, " WHERE " & f & " NOT IS NULL,"); // initialize result with <"son", "father"> pairs

execute ("INSERT INTO " & transClosure & " SELECT 1 AS [Level], " & x & ", " & g & " FROM " & R &

 iif($\text{nulls?}[2]=0$, " WHERE " & g & " NOT IS NULL,"); // initialize result with <"son", "mother"> pairs

level = 2; // 2nd level "ancestors" // card = execute ("SELECT Count(*) FROM " & transClosure);

while card \neq oldcard

 oldcard = card; // prevent infinite looping

 execute ("INSERT INTO " & transClosure & " SELECT " & level & " AS [Level], " & transClosure & "." & a_1 & ", " & R & "." & f & " FROM " & R & " INNER JOIN " & transClosure & " ON " & transClosure & "." & a_2 & " = " & R & "." & x & " WHERE [Level]= " & level - 1 & iif($\text{nulls?}[1]=1$, " AND " & R & "." & f & " NOT IS NULL,");

 execute ("INSERT INTO " & transClosure & " SELECT " & level & " AS [Level], " & transClosure & "." & a_1 & ", " & R & "." & g & " FROM " & R & " INNER JOIN " & transClosure & " ON " & transClosure & "." & a_2 & " = " & R & "." & x & " WHERE [Level]= " & level - 1 & iif($\text{nulls?}[2]=1$, " AND " & R & "." & g & " NOT IS NULL,");

 execute ("DELETE FROM " & transClosure & " WHERE x IN (SELECT y FROM (SELECT Min(Descendent), Min(Ancessor), Count(Descendent) AS NumberOfDups, Max(x) AS y FROM " & transClosure & " GROUP BY Descendent, Ancessor HAVING Count(Descendent)>1);")

 card = execute ("SELECT Count(*) FROM " & transClosure);

 level = level + 1; // prepare next level "ancestors"

end while;

end algorithm computeBinarySelfProductTransClosure;

Fig. 17. MatBase algorithm for computing binary self-function products' transitive closures

which yields (when all null values are desired) the following RA equation:

$$\begin{aligned} \text{transClosure} &= \rho_{a_1, a_2 \leftarrow c_1, f}(\pi_{c_1, f}(R) \cup \rho_{a_1, a_2 \leftarrow c_1, g}(\pi_{c_1, f}(R))) \\ &\cup \rho_{a_2 \leftarrow f}(\pi_{a_1, f}(\text{transClosure} \triangleleft_{a_2=c_1} \pi_{c_1, f}(R))) \\ &\cup \rho_{a_2 \leftarrow g}(\pi_{a_1, g}(\text{transClosure} \triangleleft_{a_2=c_1} \pi_{c_1, g}(R))) \end{aligned}$$

Fig. 16. RA equation corresponding to the Datalog \rightarrow generic program in Fig. 15 (all nulls)

Its evaluation can be done by a method *computeBinarySelfProductTransClosure*, whose algorithm is presented in Fig. 17 (which is, except for the duplicate deletion step, an obvious extension of *computeTransClosure*):

For example, the following call of this method computes the transitive closure $(Father \bullet Mother)^*$ into table *RulersTransClosure*, with nulls only for those initially existing in *Father*: `computeBinarySelfProductTransClosure([x], Father, Mother, RulersTransClosure, Descendant, Ascendant, , 0)`; the corresponding computed instance is showed in Fig. 18.

$$RulersTransClosure = (Father \bullet Mother)^* = (Father \bullet Mother)_6^* (= (Father \bullet Mother)_7^* = \dots)$$

Level	Descendant	Ancestor
1	36	35
1	33	35
1	23	35
1	35	19
1	15	42
1	19	26
1	26	22
1	22	5
1	5	241
1	241	
1	240	239
1	239	
1	42	71
1	71	75
1	75	218
1	218	46
1	46	
1	36	493
1	33	485
1	23	15
1	35	255
1	19	248
1	26	58
1	22	243
1	5	240
1	42	44
1	71	57
1	75	73
2	36	19
2	33	19
2	23	19
2	35	26
2	15	71
2	19	22
2	26	5
2	22	241
2	42	75
2	71	218
2	75	46
2	23	42
2	5	239
2	42	218
2	71	46
3	36	26
3	33	26
3	23	26
3	35	22
3	15	75

3	19	5
3	26	241
3	23	71
3	23	44
3	42	46
4	36	5
4	33	5
4	23	5
4	35	241
4	15	46
4	23	75
4	33	57
5	36	241
5	33	241
5	23	241
5	23	218
5	23	73
5	36	240
5	33	240
5	23	240
6	36	239
6	33	239
6	23	239
6	23	46

Fig. 18. $(Father \bullet Mother)^*$'s fixpoint for RULERS instance in Fig. 1

It is obvious that $(Father \bullet Mother)^*$ contains the pair $\langle 23, 46 \rangle$ (see its last instance line), so Radu cel Frumos (id 23) has also been discovered as descending too from Bogdan I (id 46), the Moldavia's founder, but does not contain a pair $\langle 36, 46 \rangle$, i.e. Vlad Ţepeş (the Impaler) "Dracula" (id 36) was not descending from Bogdan I; this, finally, is proving not only that the answer to the question whether HRH Prince Charles of Wales also descends from the founders of both Wallachia and Moldavia is negative (i.e. partially true -for Wallachia- but partially false -for Moldavia), but, much more important, that, indeed, $(f \bullet g)^* \not\subseteq f^* \cup g^*$ (i.e. $(f \bullet g)^*$ is richer than $f^* \cup g^*$).

Please note first that, in fact, *MatBase* actual algorithm for computing transitive closures is more powerful and complicated: as its metacatalogue also stores (fundamental) functions, (computed) function products, and their members, only the ids of the desired function product is needed instead of the first three parameters of the `computeBinarySelfProductTransClosure` method from Fig. 17; but the major advantage of this approach is the fact that *MatBase* can compute transitive closures for relations and function products of any arity (i.e. not only for binary ones).

B.2. The need for deleting duplicates in each step of the computation

Formalizing genealogical trees, both *Father* and *Mother* are acyclic. However, the graph of their product may contain cycles (i.e. generally, the union of tree-type graphs is no more tree-like, but a lattice-type one).

Especially in royal houses, it is frequently the case that somebody is a descendant of a same person several times, on several branches of his/her family. For example, to keep it simple, let us recall that the famous pharaoh Akhenaten (founder of the monotheism), son of Amenhotep III and Tiye, married one of his sisters (which was common not only in ancient Egypt). Consequently, their famous son Tutankhamun was twice descending from both his grandparents.

If not deleted in the Algorithm above, such duplicates (on a same level) are only polluting the final answer with more and more duplicates (on each lower levels).

There may be, however, such duplicates on different levels as well; for example, let x be a father of y and z , w a descendant of y , and v a child of w and z : then, v is a descendant of x twice, once as his grandfather (through z) and once as his grand-

grandfather (through y and w). This means that, on a level l , the above algorithm adds to the result a pair $\langle v, x \rangle$ and then, on level $l + 1$, would add it once more.

Consequently, the above Algorithm would never stop if such duplicates were not deleted.

B.3. Computing transitive closures for self-function products of any arity

In fact, *MatBase* is able to compute transitive closures for any $f_1 \bullet \dots \bullet f_n: R \rightarrow R^n$ product (n being a strictly positive natural), as $(f_1 \bullet \dots \bullet f_n)^* = (G_{f_1} \cup \dots \cup G_{f_n})^*$. Fig. 19 presents this generalized algorithm, also based on the associativity of the union operator. Note that, as *MatBase* does not allow definition of function products having different domains, there is no need for checking that anymore.

Corresponding extended *Datalog* \rightarrow and RA counterparts are trivially obtainable for any $n > 2$.

Algorithm computeProductTransClosure

Input: id F of an integer function product $f_1 \bullet \dots \bullet f_n$, $n > 0$, whose graph is stored by a table R , having an integer column as its surrogate primary key;

transClosure, a_1 , a_2 –the (distinct) names of the desired table and its two columns for storing the result;

nulls? – a tuple of the type $\langle 0 \text{ or } 1 \text{ or } 2, \dots, 0 \text{ or } 1 \text{ or } 2 \rangle$, where each element is describing user request for nulls processing for the corresponding member functions of F (using same codes meaning as in Fig. 13);

Output: *transClosure* instance, storing the transitive closure of $F = f_1 \bullet \dots \bullet f_n$;

Strategy:

if F does not correspond to a function then return error("wrong parameter: there is no function having this id!");

$R = \text{domain}(F)$; $n = \text{arity}(F)$;

loop for $i = 1, n, 1$

if $\text{codomain}(F[i]) \not\subset \text{INT}$ then return error("impossible to compute transitive closure: i -th F 's member data type is not an integer one!");

end loop;

if existsTable(*transClosure*) then execute ("DELETE FROM " & *transClosure*)

else execute("CREATE TABLE " & *transClosure* & "([Level] INT, " & a_1 & " INT, " & a_2 & " INT);");

oldcard = 0; // *transClosure* is empty

if $\text{codomain}(f_1) = R$ or $n = 1$ then $x = \text{primaryKeyName}(R)$;

else begin $n = n - 1$; $x = F[1]$;

loop for $i = 1, n, 1$

$F[i] = F[i + 1]$;

end loop;

end;

loop for $i = 1, n, 1$

execute("INSERT INTO " & *transClosure* & " SELECT 1 AS [Level], " & x & ", " & $F[i]$ & " FROM " & R & " iff(*nulls?*[i]=0, " WHERE " & $F[i]$ & " NOT IS NULL",); // initialize result with $\langle x, f_i(x) \rangle$ pairs

```

end loop;
level = 2; // next step will add second level "ancestors"
card = execute("SELECT Count(*) FROM " & transClosure);
while card ≠ oldcard
    oldcard = card; // prevent infinite looping
    loop for i = 1, n, 1
        execute("INSERT INTO " & transClosure & " SELECT " & level & " AS [Level], " & transClosure & "." & a1 &
            ", " & R & "." & F[j] & " FROM " & R & " INNER JOIN " & transClosure & " ON " & transClosure & "."
            & a2 & " = " & R & "." & x & " WHERE [Level] = " & level - 1 & " AND " & R & "." & F[j] &
            " NOT IS NULL",);
    end loop;
    execute("DELETE FROM " & transClosure & " WHERE x IN (SELECT y FROM (SELECT Min(Descendent),
        Min(Ancessor), Count(Descendent) AS NumberOfDups, Max(x) AS y FROM " & transClosure &
        " GROUP BY Descendent, Ancessor HAVING Count(Descendent) > 1);")
    card = execute("SELECT Count(*) FROM " & transClosure);
    level = level + 1; // prepare next level "ancestors"
end while;
end algorithm computeProductTransClosure;

```

Fig. 19: MatBase algorithm for computing function products (of any arity) transitive closures

Theorem 2: Algorithm *computeProductTransClosure* from Fig. 19 has the following four properties:

(i) its complexity is $O(n * level)$, where n is the arity of the input function product (i.e. the number of its member functions) and $level$ is the maximum of all lengths of the corresponding n digraphs

(ii) it is sound (i.e. it is not generating anything else but elements of the transitive closure of the input function product)

(iii) it is complete (i.e. it generates all elements of the transitive closure of the input function product)

(iv) it is optimal (i.e. it computes the transitive closure of the input function product in the least number of steps possible)

Proof: (similar to the one of *Theorem 1* above)

(i) Trivial, as it has only three finite loops (hence, as it is also deleting any duplicates in each iteration, it never loops infinitely): the first two of them depending on the finite input function product arity n , and the last one depending on the maximum longest path ($level - 2$ at the end of the loop execution, as the final iteration does not add any new elements to the result, thus corresponding to the second fixpoint, and as one more execution of the last statement of the loop takes place before discovering that the *while* condition has become *true*) in the digraphs of the input function members (trivially, as db instances are finite, any such length is finite), and also on the inner fourth loop, which is executed in each iteration of the third one for n times (and even if some such

executions would not add any new elements to the result, so no disk writes are necessary, only read ones are).

(ii) Obviously, the second loop only adds the n input function members digraphs; then, in each iteration of the third loop, for any pair $\langle x, y \rangle$ in the current result approximation and $\langle y, z \rangle$ in the current corresponding i -th function member digraph, it is only added the pair $\langle x, z \rangle$, according to the transitivity rule.

(iii) Obviously, the second and the fourth (i.e. the inner to the third one) loops are executed for each member function of the input function product, while the third one is executed up until no further transitively computable pairs may be added to the result: the first statement of this loop makes sure that variable *oldcard* is storing the current result cardinal, the last but one one is updating variable *card* value to the cardinal of the result after adding current iteration elements, and the *while* statement condition ensures that the process repeats only as long as these two values are not equal (i.e. as long as the previous iteration was adding at least one new element to the result).

(iv) Obviously, as soon as the previous third loop iteration did not add any new elements to the result, the process stops (i.e. the algorithm only computes the first two fixpoints, which is the minimum possible in order to discover the least fixpoint). Moreover, the algorithm never generates duplicates on a same level (as it joins to the input relation only the current result elements that were added in the previous iteration of

the third loop), which is minimizing disk accesses for both reading and writing operations. *q.e.d.*

Obviously, corresponding extensions of Propositions 1 and 2 above for n functions ($n > 2$) are trivial and their proofs obvious.

MatBase users may call this method also for computing transitive closures of dyadic relations (i.e. over a same set), by considering its canonical Cartesian projections as the members of a function product (defining its scheme) of the type $F = f_1 \bullet f_2 : R \rightarrow S^2$ ($S \neq R$).

In fact, trivially, *MatBase* is able to compute transitive closures for any integer function product $f_1 \bullet \dots \bullet f_n : R \rightarrow \text{INT}^n$ (n being a strictly positive natural), as $(f_1 \bullet \dots \bullet f_n)^* = (G_{f_1} \cup \dots \cup G_{f_n})^*$.

III. COMPUTING DYADIC RELATION AND FUNCTION PRODUCT CLOSURES FOR THEIR DOMAIN ELEMENTS

As seen in Fig. 3 and 4 above, a simpler (and faster) way to compute somebody's ascendance is by using *Datalog* programs instantiations.

For example, given any relation $S \subseteq R \times R$ and any given $x \in R$, we define x 's transitive closure $x^* =$

Algorithm computeDyadicRelInstantiationTransClosure

Input. c_1, c_2 – the two columns of a table R storing the desired relation's graph instance;

transClosure, a_1, a_2 – the (distinct) names of the desired table and its two columns for storing the result;

nulls? – 0, if no null values are desired in c_2 , or

1, if no null values are desired in c_2 except for those in R (which is the default value), or

2, if all null values are desired in c_2 ;

x – the value of the surrogate key of R for which the closure is computed;

Output. table *transClosure* instance, storing the corresponding transitive closure;

Strategy.

$R = \text{domain}(c_1)$;

if $R \neq \text{domain}(c_2)$ or $\text{codomain}(c_1) \neq \text{INT}$ or $\text{codomain}(c_2) \neq \text{INT}$ then

return error("impossible to compute transitive closure: c_1 and c_2 are either not columns of a same table or have incompatible data types!");

if existsTable(*transClosure*) then execute ("DELETE FROM " & *transClosure*)

else execute("CREATE TABLE " & *transClosure* & "([Level] INT, " & a_1 & " INT, " & a_2 & " INT);");

oldcard = 0; // *transClosure* is empty

execute("INSERT INTO " & *transClosure* & " SELECT 1 AS [Level], " & c_1 & " , " & c_2 & " FROM " & R &

" WHERE $x =$ " & x & " and " & *nulls?* = 0, " AND " & c_2 & " NOT IS NULL."); // initialize with <"son", "father"> pairs

level = 2; // next step will add second level "ancestors"

card = execute("SELECT Count(*) FROM " & *transClosure*);

while *card* \neq *oldcard*

oldcard = *card*; // prevent infinite looping

execute("INSERT INTO " & *transClosure* & " SELECT " & *level* & " AS [Level], " & *transClosure* & "." & a_1 &

$S^*|_x = \{ y \in R \mid \langle x, y \rangle \in S^* \}$; in particular, given any self-function $f : R \rightarrow R$ (having graph $G_f = \{ \langle x, f(x) \rangle \mid x \in R \}$) and any given $x \in R$, $x^* = G_f^*|_x = \{ y \in R \mid \langle x, y \rangle \in G_f^* \}$. Trivially, given another (not necessarily distinct) $g : R \rightarrow R$ (having graph $G_g = \{ \langle x, g(x) \rangle \mid x \in R \}$), corresponding x 's transitive closure for the function product $(f \bullet g)^* = (G_f \cup G_g)^*$ is $x^* = (G_f \cup G_g)^*|_x = \{ y \in R \mid \langle x, y \rangle \in (G_f \cup G_g)^* \}$.

Obviously, by translating, for example, the program instantiation from Fig. 3 above into RA, the following equation is obtained:

$$36^* = \rho_{\text{Ancestor} \leftarrow \text{Father}}(\pi_{\text{Father}}(\sigma_{x=36}(\text{RULERS}))) \cup \rho_{\text{Ancestor} \leftarrow \text{Father}}(\pi_{\text{Father}}(36^* \bowtie_{\text{Ancestor} = x} \pi_x, \text{Father}(\sigma_{x=36}(\text{RULERS}))))$$

Fig. 20. Recursive RA equation corresponding to the instantiation in Fig. 3

Fig. 21 presents *MatBase*'s algorithm for computing such instantiation closures.

Similarly, Fig. 22 presents *MatBase*'s algorithm for computing instantiation closures for function products of any arity.

```

    “, “ & R & “.” & c2 & “ FROM “ & R & “ INNER JOIN “ & transClosure & “ ON “ & transClosure & “.” & a2 & “=“
    & R & “.” & c1 & “ WHERE [Level]=“ & level – 1 & “ AND x = “ & x & iif(nulls?=1, “ AND “ & R & “.” & c2 &
    “ NOT IS NULL”),);
    card = execute(“SELECT Count(*) FROM “ & transClosure);
    level = level + 1; // prepare next level “ancestors”
end while;
end algorithm computeDyadicRelInstantiationTransClosure;

```

Fig. 21. MatBase algorithm for computing transitive closures of dyadic relations' elements

Algorithm computeFuncProductInstantiationTransClosure

Input: id F of an integer function product $f_1 \bullet \dots \bullet f_n$, $n > 0$, whose graph is stored by a table R , having an integer column as its surrogate primary key;

$transClosure$, a_1 , a_2 – the (distinct) names of the desired table and its two columns for storing the result;

$nulls?$ – a tuple of the type $\langle 0 \text{ or } 1 \text{ or } 2, \dots, 0 \text{ or } 1 \text{ or } 2 \rangle$, where each element is describing user request for nulls processing for the corresponding member functions of F (using same codes meaning as in Fig. 13);

x – the value of the surrogate key of R for which the closure is computed;

Output: $transClosure$ instance, storing the transitive closure of $F = f_1 \bullet \dots \bullet f_n$ for the element x ;

Strategy:

if F does not correspond to a function then return error(“wrong parameter: there is no function having this id!”);

$R = \text{domain}(F)$; $n = \text{arity}(F)$;

loop for $i = 1, n, 1$

if $\text{codomain}(F[i]) \not\subset \text{INT}$ then return error(“impossible to compute transitive closure: i -th F 's member data type is not an integer one!”);

end loop;

if existsTable(transClosure) then execute (“DELETE FROM “ & transClosure)

else execute(“CREATE TABLE “ & transClosure & “([Level] INT, “ & a₁ & “ INT, “ & a₂ & “ INT);”;

$oldcard = 0$; // $transClosure$ is empty

if $\text{codomain}(f_1) = R$ or $n = 1$ then $x = \text{primaryKeyName}(R)$;

else begin $n = n - 1$; $x = F[1]$;

loop for $i = 1, n, 1$

$F[i] = F[i + 1]$;

end loop;

end;

loop for $i = 1, n, 1$

execute(“INSERT INTO “ & transClosure & “ SELECT 1 AS [Level], “ & x & “, “ & F[i] & “ FROM “ & R &

iif(nulls?[i]=0, “ WHERE x = “ & x & “ AND “ & F[i] & “ NOT IS NULL”),); // initialize with $\langle x, f_i(x) \rangle$ pairs

end loop;

$level = 2$; // next step will add second level “ancestors”

$card = \text{execute(“SELECT Count(*) FROM “ & transClosure);}$

while $card \neq oldcard$

```

oldcard = card;    // prevent infinite looping
loop for i = 1, n, 1
    execute("INSERT INTO " & transClosure & " SELECT " & level & " AS [Level], " & transClosure & "." & a1 &
        ", " & R & "." & F[i] & " FROM " & R & " INNER JOIN " & transClosure & " ON " & transClosure & "." & a2
        & "=" & R & "." & x & " WHERE [Level]=" & level - 1 & " AND x =" & x & " AND " & R & "."
        & F[i] & " NOT IS NULL",);
end loop;
execute("DELETE FROM " & transClosure & " WHERE x IN (SELECT y FROM (SELECT Min(Descendent),
    Min(Ancessor), Count(Descendent) AS NumberOfDups, Max(x) AS y FROM " & transClosure & " GROUP
    BY Descendent, Ancessor HAVING Count(Descendent)>1);")
card = execute("SELECT Count(*) FROM " & transClosure);
level = level + 1;    // prepare next level "ancestors"
end while;
end algorithm computeFunctProductInstantiationTransClosure;
    
```

Fig. 22. MatBase algorithm for computing transitive closures of function products' elements

These two latter algorithms also enjoy the same four properties as those from Fig. 13 and 19, respectively (i.e their complexities are $O(\text{longest digraph path for element } x)$ and $O(n * \text{level}_x)$, respectively, and they are sound, complete, and optimal). The corresponding proofs are left to the reader, as they are only slight simplifications of those of the two *Theorems* above.

CONCLUSION

The main contribution of this paper is not only presenting how should be theoretically computed transitive closures for both dyadic relations, self-functions, function products of the type $f_1 \bullet \dots \bullet f_n: R \rightarrow S^n$ (by computing the transitive closure of the union of their members' graphs), and for their domain elements, but also introducing the elegant way in which *MatBase*, a prototype intelligent db and kb management system developed by the author, is actually computing them, both in *Datalog*-, RA, and a pseudo-code embedding SQL.

Moreover, it is proved that *MatBase* algorithms for computing transitive closures (both for n -ary homogeneous relations and function products, as well as for their instance elements) are linear, solid, complete, and optimal. In the sequel, it is also proved that the union of the transitive closures of the members of any function product is always included in the transitive closure of their product, but the reverse is not true: generally, the transitive closure of a function product is richer than the union of its members' transitive closures.

Note that no commercially available, nor prototype system is offering to its users the possibility to compute transitive closures for function products and that this facility is crucial in order to be able to correctly answer such questions as the one in the

subtitle of this paper, without first designing and running costly preliminary queries.

Anecdotically, examples also show that HRH Prince Charles of Wales, who has among his maternal ancestors Vlad Călugărul, a stepbrother of Vlad „Dracula” „the Impaler”, is descending from Wallachia's founder, Basarab I (of Cuman origin by his father), but not from Moldova's one, Bogdan I, although most of „Dracula”'s stepbrothers and sisters are also, by their mother, descending from Bogdan I.

REFERENCES

- [1] S. Abiteboul, R. Hull, and V. Vianu. Foundations of Databases. Addison-Wesley, 1995.
- [2] R. Agrawal and H. V. Jagadish, "Direct algorithms for computing the transitive closure of database relations," in Proc. 13th Int. Conf. Very Large Data Bases, Brighton, England, Sept. 1987, pp. 255-266.
- [3] A. Beletska, D. Barthou, W. Bielecki and A. Cohen. "Computing the Transitive Closure of a Union of Affine Integer Tuple Relations", Lecture Notes in Computer Science, 2009, Volume 5573/2009, pp. 98-109, DOI: 10.1007/978-3-642-02026-1_9.
- [4] M. Bozga, R. Iosif, and F. Konecný. "Transitive Closures of Ultimately Periodic Relations", VERIMAG Technical Report TR-2011-14, 2011.
- [5] S. Ceri, G. Gottlob, L. Tanca, "What You Always Wanted to Know About Datalog (And Never Dared to Ask)," *IEEE Transactions on Knowledge and Data Engineering*, vol. 1, no. 1, pp. 146-166, Mar. 1989.
- [6] P.P. Chen. "The Entity-Relationship Model: Toward a Unified View of Data". ACM Trans. on Database Syst. 1(1): 9-36, 1976.

- [7] D. Chimenti, R. Gamboa, R. Krishnamurti, S. Naqvi, S. Tsur, and C. Zaniolo. "The LDL System Prototype". *IEEE Trans. on Knowledge and Data Eng.* 2(1): 76-90, 1990.
- [8] E. F. Codd. "A Relational Model for Large Shared Data Banks". *Comm. Of the ACM* 13(6): 377-387, 1970.
- [9] C. Demetrescu and G.F. Italiano. "Fully dynamic transitive closure: Breaking through the $O(n^2)$ barrier". In *Proc. of the 41st IEEE Annual Symposium on Foundations of Computer Science (FOCS'00)*, pp. 381–389, 2000.
- [10] N. Djuvara. "Thocomerius – Negru Vodă a Ruling Prince of Cuman Origins at the beginnings of Wallachia" (in Romanian). Humanitas Publishing House, Bucharest, Romania, 2nd ed., 2011.
- [11] W. Kelly, W. Pugh, E. Rosser, and T. Shpeisman, "Transitive closure of infinite graphs and its applications", University of Maryland at College Park, College Park, MD, 1994, CS-TR-3457.
- [12] V. King. "Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs". In *Proc. 40th IEEE Symposium on Foundations of Computer Science (FOCS'99)*, pp. 81–99, 1999.
- [13] C. Mancas. "On Knowledge Representation using an Elementary Mathematical Data Model". *Proc. of IASTED Intl. Conf. on Information and Knowledge Sharing*, St. Thomas, U.S. Virgin Islands, 206-211, 2002.
- [14] C. Mancas, S. Dragomir: "Matbase Datalog-subsystem meta-catalog conceptual design". *IASTED Conf. on Software Engineering and Applications 2004*, pp. 34-41.
- [15] C. Mancas, *Conceptual Data Modeling and Database Design: A Completely Algorithmic Approach. Volume I: The Shortest Advisable Path.* Apple Academic Press / CRC Press (Taylor & Francis Group), Waretown, NJ, 2015.
- [16] C. Mancas, *Conceptual Data Modeling and Database Design: A Completely Algorithmic Approach. Volume II: Refinements for an Expert Path.* Apple Academic Press / CRC Press (Taylor & Francis Group), Waretown, NJ (2022, in press).
- [17] H. De Meyer, H. Naessens, B. De Baets. "Algorithms for computing the min-transitive closure and associated partition tree of a symmetric fuzzy relation", *European Journal of Operational Research* Volume 155, Issue 1, 16 May 2004, pp. 226-238.
- [18] C. Pang, G. Dong, and K. Ramamohanarao, "Incremental Maintenance of Shortest Distance and Transitive Closure in First-Order Logic and SQL," *ACM Transactions on Database Systems*, 30(3), pp. 698-721, 2005.
- [19] T. Przymusiński, "Every logic program has a natural stratification and an iterated least fixed point model", in *8th ACM Symp. Principles Database Syst. (PODS)*, Mar. 1989, pp. 11-21.
- [20] C. Sternagel and R. Thiemann. "Executable Transitive Closures of Finite Relations", *The Archive of Formal Proofs*, 2011.
- [21] J. D. Ullman. "Implementation of Logical Query Languages for Databases". *ACM Trans. On Database Systems*, 10(3): 289-321, 1985.
- [22] S. Verdoolaege, A. Cohen, and A. Beletskaya. "Transitive Closures of Affine Integer Tuple Relations and their Overapproximations", *INRIA N° RR-7560*, 2011.
- [23] S. Warshall. "A theorem on Boolean matrices". *Journal of the ACM* 9 (1): 11–12, 1962.
- [24] Vlad the Impaler: How is Prince Charles, Queen Elizabeth related to him? - CBS News, 2004. <http://www.redicecreations.com/news/2004/princevlad-the-impaler.html>
- [25] Intellidimension - Semantic Web Tools and Technology for Windows, RDF for .NET Framework (C#), SPARQL on SQL Server, 2021.