On MatBase's algorithm for preventing cycles in binary Cartesian function products

Christian Mancas *

DATASIS ProSoft srl, Bucharest, Romania.

World Journal of Advanced Engineering Technology and Sciences, 2022, 07(01), 023–037

Publication history: Received on 01 August 2022; revised on 06 September 2022; accepted on 09 September 2022

Article DOI: https://doi.org/10.30574/wjaets.2022.7.1.0092

Abstract

This paper introduces the algorithm that MatBase (an intelligent knowledge and database management system prototype) uses for enforcing acyclicities of binary Cartesian function products, characterizes it -proving that it is complete, sound, optimal, and linear- and, besides its pseudocode embedding SQL, also provides an example implementation in standard ANSI-99 SQL and MS VBA.

Keywords: Database constraint enforcement; Binary Cartesian function product acyclicity; MatBase; The (Elementary) Mathematical Data Model

1. Introduction

Database (db) constraint enforcement is crucial for guaranteeing db instances plausibility [1]. While this is fairly simple for the Relational Data Model (RDM) [1, 2, 3], for which all RDM-based Management Systems (RDBMS) provide six types of constraints (namely, domain/range, not-null, reference integrity, key/uniqueness, tuple/check, and default values), things are not at all simple for most of the non-relational ones, e.g., those provided by the (Elementary) Mathematical Data Model ((E)MDM) [4, 5, 6].

For example, let us consider the RDM table scheme and instance from Figure 1 (where the primary key is $x$, an integer autonumber generated by the underlying RDBMS, Name, a mandatory text made of at most 64 ASCII characters, is a unique key as well, whereas Mother, Father, and Spouse are foreign keys referencing $x$; please recall that for any function $f: D \rightarrow C$, its image, denoted $\text{Im}(f)$, is the set of all the values taken by $f$, i.e. $\text{Im}(f) = \{y \in C \mid \exists x \in D, f(x) = y\} \subseteq C$).

Mathematically, the five columns of this relational table are implementing the following five functions, respectively (where $\leftrightarrow$ denotes a one-to-one function, NATURALS(8) is the subset of naturals having at most 8 digits, and NULLS is a distinguished countable set of null values):

$$x: \text{PERSONS} \leftrightarrow \text{autonumber} \subseteq \text{NATURALS(8)}$$

$$\text{Name} : \text{PERSONS} \leftrightarrow \text{ASCII}(64)$$

$$\text{Mother} : \text{PERSONS} \rightarrow \text{PERSONS} \cup \text{NULLS}$$

$$\text{Father} : \text{PERSONS} \rightarrow \text{PERSONS} \cup \text{NULLS}$$

$$\text{Spouse} : \text{PERSONS} \rightarrow \text{PERSONS} \cup \text{NULLS}$$
Please recall that:

- Either they are not known, not interesting for the time being or inapplicable (e.g. Mihai, the grand grandson of King Mihai I of Romania is a baby, who is not married).

All relational constraints of this table are satisfied, namely:

- \( x \) has integer unique values for all 16 elements;
- \( Name \) has unique ASCII strings values of at most 64 characters for all 16 elements;
- \( Mother, Father, \) and \( Spouse \) have either null values, or integer values between 1 and 16, i.e. of the image of \( x \).

However, even this so simple db scheme has non-relational constraints as well, namely:

1. \( Mother \) acyclic (i.e. nobody may be his/her mother, grandmother, grand grandmother, etc.);
2. \( Father \) acyclic (i.e. nobody may be his/her father, grandfather, grand grandfather, etc.);
3. \( Spouse \) irreflexive (i.e. nobody may be his/her spouse);
4. \( Spouse \) symmetric (i.e. if \( x \) is the spouse of \( y \), then \( y \) is the spouse of \( x \), for any \( x, y \) in \( PERSONS \));
5. \( Mother \) \& \( Father \) acyclic (i.e. nobody may be his/her ancestor or descendant);
6. \( Mother \) \& \( Spouse \) acyclic (i.e. nobody may be both the spouse and the mother/grandmother, etc. of somebody);
7. \( Father \) \& \( Spouse \) acyclic (i.e. nobody may be both the spouse and the father/grandfather, etc. of somebody).

Please recall that:

- both \( Mother \) and \( Father \) model genealogical trees and trees are acyclic (i.e. they may not contain cycles, i.e., generally, for any self-map \( f: D \to D \), there is no sequence \( x_1, ..., x_n, x_{n+1} \) of elements of \( D \) such that \( f(x_1) = y_1, ..., f(x_n) = y_n, f(x_{n+1}) = y_1 \), i.e. \( f^n(x) \neq f(x) \), for any \( x \) in \( D \) and any natural \( n > 1 \);)
- acyclicity implies anti-symmetry, which implies irreflexivity (e.g. \([5, 6] \));
- anti-symmetry (or asymmetry) is the dual of symmetry (e.g. if \( x \) is the mother/father of \( y \), then \( y \) may not be the mother/father of \( x \) for any \( x, y \) in \( PERSONS \));
- irreflexivity of a self-map \( f: D \to D \) means that \( f(x) \neq x \), for any \( x \) in \( D \);
- for any functions \( f: D \to C_1 \) and \( g: D \to C_2 \), their Cartesian product is the function \( f \times g: D \to C_1 \times C_2 \).

Figure 1 A relational table and a plausible instance of it

As usual, the three self-maps \( Mother, Father, \) and \( Spouse \) may have null values for some of the elements of the \( PERSONS \) set because either they are not known, not interesting for the time being in this context, or inapplicable (e.g. Mihai, the grand grandson of King Mihai I of Romania is a baby, who is not married).
• acyclicity of a homogeneous binary function product \( f \circ g : D \rightarrow C \times C \) means that there is no sequence \( x_1, \ldots, x_n \), \( x_{n+1} \) of elements of \( D \) such that \( (f \circ g)(x_1) = < y_1, z_1 >, \ldots, (f \circ g)(x_n) = < y_n, z_n >, (f \circ g)(x_{n+1}) = < y_{n+1}, y_{n+1} > \) or such that \( (f \circ g)(x_{n+1}) = < z_1, y_{n+1}, y_{n+1} >, n \) natural.

We carefully entered data in the table PERSONS from Figure 1, so that these non-relational constraints are satisfied as well. Generally, however, especially when working with huge data, end-users should never be left unassisted in face of non-relational constraints as, for example, in this case, they are generally not historians and even historians are humans, so prone to errors.

For example, if someone were saving for Queen Victoria of the UK the values

- 1, 4, 5, or 14 in the Mother cell, the constraint Mother acyclic would be violated;
- 1 in the Spouse cell, the constraint Spouse irreflexive would be violated;
- 2, 3, 6, 7, 8, 10, 12, 15, or 16 in the Spouse cell, the constraint Spouse symmetric would be violated;
- 7, 12, 15, or 16 in the Father cell, the constraint Mother • Father acyclic would be violated.

For example, if someone were saving for King Mihai I of Romania the values

- 6, 7, 12, 15, or 16 in the Father cell, the constraint Father acyclic would be violated;
- 1, 4, 5, 9, or 11 in the Spouse cell, the constraint Mother • Spouse acyclic would be violated;
- 14 in the Spouse cell, the constraint Father • Spouse acyclic would be violated.

1.1. MatBase

MatBase [5 - 13] is an intelligent knowledge and database management system prototype developed by us and based on both the (E)MDM, RDM, Datalog ~ [3, 6, 13], and the Entity-Relationship (E-R) Data Model (E-RDM) [1, 14, 15]. MatBase accepts (E)MDM schemes and automatically translates them into RDM ones and may extract corresponding E-R Diagrams (E-RD); dually, it accepts both E-RDs and RDM schemes and automatically translates them into (E)MDM ones. Finally, it accepts Datalog– programs as well and, whenever correct, it computes their results against the corresponding RDM schemes and instances.

Moreover, MatBase automatically generates software applications for managing the data it stores, including code for enforcing both relational and non-relational constraints.

Currently, MatBase has two versions: one developed in MS Access VBA, for small dbs, and a MS C# and SQL Server, for professional management of big and huge dbs.

1.2. Enforcing the non-relational constraints

There are only two possible ways to enforce non-relational constraints: either use an intelligent DBMS as MatBase (that are not commercially available) or through event-driven software db applications (APP) managing data stored by RDBMSes.

Whenever possible, it is best to enforce them preventively, i.e. to filter out un plausible data that would violate the constraints and provide end-users with only plausible data to choose from.

For example, constraint Spouse irreflexive may be easily enforced as follows: Spouse should be implemented in the PERSONS graphical user interface (GUI) form as a combo-box (storing the x values, but presenting end-users the corresponding Name values, alphabetically ordered); whenever end-users (or the APP) successfully moved the cursor on another data row of the underlying PERSONS table, the APP recomputes the Spouse combo-box, eliminating the current person from it (of course that on new lines, nothing has to be eliminated as the current person data is not yet saved in the db).

For example, to do it in MS VBA, only the following six lines of code must be added to the Form_Current method of the PERSONS class (which is automatically invoked by the system each time the data cursor arrives on another line):

```vba
If NewRecord Then
    Spouse.RowSource = "SELECT x, Name FROM PERSONS ORDER BY Name"
Else
```

```vba
    App compute for Spouse the current person, eliminating the previous one
```
However, most of the non-relational constraints may not be enforced preventively, but only curatively, i.e. after end-users chose data the APP should check whether it is plausible and reject unplausible values.

For example, enforcing the constraint Spouse symmetric should be done as follows:

- after end-users changed data in the Spouse combo-box for the current person and successfully saved it in the db, the APP should analyze the context and:
  - if Spouse was nullified, then the Spouse value for the former spouse must be automatically nullified as well;
  - if Spouse was null and it has become not-null, then the Spouse value of the newly saved spouse must be automatically set to the current person’s x value;
  - if Spouse was not null and was updated to another not-null value, then, automatically, the Spouse value of the former spouse must be nullified and the one of the current spouse must be set to the current person’s x value;
  - recompute the Spouse combo-box for all persons.

For example, in MS VBA this needs some 18 code lines to be added to the Form.AfterUpdate method of the PERSONS class (which is automatically invoked by the system each time the data of the current line was saved in the db).

Enforcing acyclicity of self-maps is even costlier; for example, enforcing the constraint Mother acyclic may be done as follows:

- after end-users changed data in the Mother combo-box for the current person and ask for saving it in the virtual memory, the APP should analyze the context and:
  - if Mother was nullified, then nothing has to be done (as null values may not violate acyclicity);
  - if Mother was set to a not-null value (distinct of the previous one, if any), then a corresponding call to a library Boolean function (called IsPath in MatBase) must be made, passing the corresponding x value, the self-map name and its domain name (Mother and PERSONS, in this case), as well as its value for the current data row (i.e. $y = \text{Mother}(x)$, in this case); this method returns true if there is a path in the self-map’s graph between y and x (i.e. a cycle would close in this graph were y accepted) or false otherwise; if it returns false, then the new value is saved, otherwise it is rejected (with a corresponding error message).

For example, in MS VBA this needs 3 code lines to be added to the Mother.BeforeUpdate method of the PERSONS class (which is automatically invoked by the system each time the data in the Mother combo-box was changed and end-users would like to save it in the virtual memory) and 28 code lines for the IsPath method of the CONSTRAINTS library.

Please note that, for any person, the corresponding graph of Mother/Father is a binary tree growing upwards for storing his/her ancestors adjacent with a n-ary tree growing downwards for storing his/her descendants (n natural). However, for the product Mother • Father the corresponding graph is not tree-like, but lattice-like, i.e., as a undirected graph it may have cycles, as there are persons descending more than one time from another one, both on same and on different length paths (see, e.g., [6, 10, 13]).

Consequently, enforcing acyclicity for binary homogeneous function products is not at all a trivial task, especially when both functions may also take null values.

1.3. Related work

Detecting cycles in both undirected and directed graphs were intensively studied (e.g., [5, 6, 7, 9, 10, 11, 16, 17, 18, 19, 20, 21, 22]). Cartesian function products were studied as well, but not intensively (e.g., [23, 24]).

However, to our knowledge, no research was done on the acyclicity of homogeneous binary Cartesian function products.

1.4. Paper outline

The second section of this paper presents and characterizes the pseudocode MatBase's algorithm for enforcing acyclicity of homogeneous binary Cartesian function products, while the third one provides its MS VBA embedding SQL implementation. The paper ends with conclusion and references.
2. *MatBase*'s algorithm for enforcing acyclicities of homogeneous binary Cartesian function products

*M*atBase's algorithm $xFunctProductAcyclic$ for enforcing acyclicity of binary Cartesian function products is based on the $computeFunctProductInstantiationTransClosure$ algorithm presented and characterized in section 3 of [13], for $n = 2$.

The main difference between them is that, after each insertion performed within the *while* loop, the instance of the result table TransClosure is searched for the values of $f(x)$ and $g(x)$, provided they are not null; moreover, whenever $f$ and $g$ are self-maps, the value of $x$ is also searched for (as, for example, nobody may be an ancestor of his/her mother/father); if such a value is found, i.e., a cycle would be created in the graph of $f \circ g$, the values of $f(x)$ and $g(x)$ saved in the db, then the computation stops and the $xFunctProductAcyclic$ Boolean function displays a corresponding error message and returns true; otherwise, computation of the transitive closure goes on and when it ends without any cycle detected the $xFunctProductAcyclic$ Boolean returns false.

Figure 2 presents the corresponding pseudocode embedding SQL [1, 6] algorithm (where SQL statements are invoked through function *execute*, // introduces comments, & is the string concatenation operator, and *codomain*, isNull, existsTable, Abs, and *if* are librarian functions performing obvious tasks: for instance, the result of the inline *if* function iif($cond$, $T$, $F$) is $T$ when $cond$ is true and $F$ otherwise).

$xFunctProductAcyclic$ should be automatically called whenever on the current line $x$ either $f(x)$ or and $g(x)$ have been updated, at least one of them is not null, and saving of data in the db for the current line has been requested (either implicitly or explicitly). For example, in MS Access VBA the corresponding event-driven method is *Form* $\_BeforeUpdate*, while in MS$ .NET$ the corresponding event is called Validating. Obviously, whenever $xFunctProductAcyclic$ returns false saving of the new data should be allowed, while otherwise it must be rejected.

For example, let us call the function $xFunctProductAcyclic$ from Figure 2 with the following input from Figure 1: $R =$ "PERSONS"; $xName =$ "x"; $fName =$ "Mother"; $gName =$ "Father"; $TransClosure =$ "TransClosure"; $a =$ "RelatedPersons"; $x = 1 ; f = 5 ; g \in$ NULLS (which corresponds to an attempt of saving the fact the Queen Mary of Romania were the mother of her grandmother, Queen Victoria of UK); here is what would happen (supposing table TransClosure does not exist; corresponding TransClosure instance is shown in Figure 3):

- $xFunctProductAcyclic = false$;
- $S = "(1, 5)"$;
- *execute("CREATE TABLE TransClosure(x COUNTER, [Level] INT, [RelatedPersons] INT)")*;
- *execute("CREATE VIEW TransClosureDuplicates AS SELECT Count([Level]) AS DupsNo, Max(x) AS y FROM TransClosure GROUP BY [RelatedPersons] HAVING Count([RelatedPersons])>1")*;
- $oldcard = 0$;
- *execute("INSERT INTO TransClosure([Level], [RelatedPersons]) SELECT 1 AS [Level], [x] FROM PERSONS WHERE [Mother] = 1 OR [Father] = 1"; // 1st line of TransClosure (Alfred)
- $level = 2$;
- $card = 1$;
- $oldcard = 1$;
- *execute("INSERT INTO TransClosure([Level], [RelatedPersons]) SELECT 2 AS [Level], PERSONS.[x] FROM PERSONS INNER JOIN TransClosure ON TransClosure.[RelatedPersons] = PERSONS.[Mother] WHERE [Level] = 1"; // no line (Alfred was not a mother)
- *execute("INSERT INTO TransClosure([Level], [RelatedPersons]) SELECT 2 AS [Level], PERSONS.[x] FROM PERSONS INNER JOIN TransClosure ON TransClosure.[RelatedPersons] = PERSONS.[Father] WHERE [Level] = 1"; // 2nd line of TransClosure (Mary)
- $cycleLen = execute("SELECT [Level] FROM TransClosure WHERE [RelatedPersons] IN (1, 5)") = 2;
- $xFunctProdAcyclic = true$;
- $oldCard = -1$;
- *execute("INSERT INTO TransClosure([Level], [RelatedPersons]) VALUES (-1 , 5)"; // 3rd line (Victoria)
- $level = -2$;
- $card = 3$;
- display " Request rejected: these values for Mother and/or Father would create a cycle of length 3 in the graph of the Cartesian function product Mother $\bullet$ Father: please change either Mother or/and Father!";

For example, suppose that end-users would like to save 7 (Carol II) in *Father* for line 15 (Nicholas, his grand grandson), which would trigger another call to $xFunctProductAcyclic$ from Figure 2 with the following input from Figure 1:
**Boolean Function** xFunctProductAcyclic(R, xName, fName, gName, TransClosure, a, x, f, g)

**Input:**
xName, fName, gName – column names of a table R storing the \( f \times g \) Cartesian function product’s graph;

\( TransClosure, a \) – the names of the desired table (distinct in the db) and its column for storing the closure;

\( x \) – The value of the surrogate key of \( R \) for the current element;

\( f = f(x) \) and \( g = g(x) \) – corresponding values of the Cartesian function product (at least one of them not null);

**Output:**
false, if \( f(x) \) and \( g(x) \) would not close a cycle in \( f \times g \)’s graph; true otherwise;

**Strategy:**
// uses the least fixpoint semantics for computing transitive closures

\[ x \text{FunctProductAcyclic} = \text{false}; \]  // no cycle detected yet

// prepare list of values that could close cycles
\[ S = \text{iif}(\text{codomain}(f) = R, x, ""); \]  // are \( f \) and \( g \) self-maps?
\[ S = \text{iif}(\text{isNull}(f(x)), S, \text{iif}(\text{isNull}(S), f(x), S \& "," \& f(x))); \]  // is \( f(x) \) not null?
\[ S = \text{iif}(\text{isNull}(g(x)), S, \text{iif}(\text{isNull}(S), g(x), S \& "," \& g(x))); \]  // is \( g(x) \) not null?

if not isNull(S) then
   // nothing to do if \( S \) is null: no cycle possible!
\[ S = (" & S & ")\];  // the set of values that should not be encountered in the transitive closure

// initialize transitive closure computation
if existsTable(TransClosure) then execute("DELETE FROM " & TransClosure);
else execute("CREATE TABLE " & TransClosure & "(x COUNTER, [Level] INT, [" & a & "] INT)");
execute("CREATE VIEW " & TransClosure & "Duplicates AS SELECT Count([" & a & "]) AS DupsNo, Max(x) AS y FROM " & TransClosure & " GROUP BY [" & a & "] HAVING Count([" & a & "])>1");
end if;

oldcard = 0;  // TransClosure is empty
// a. add “descendants” of \( x \)
execute("INSERT INTO " & TransClosure & "([Level], [" & a & "]) SELECT 1 AS [Level], [" & xName & "] FROM " & R & " WHERE [" & fName & "] =" & x & " OR [" & gName & "] =" & x);  // initialize result with \( x \)’s “children”

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

card = execute("SELECT Count (*) FROM " & TransClosure);  // “children”’s cardinal
while card ≠ oldCard And Not xFunctProdAcyclic  // loop until no “descendant” is added or cycle is found
   oldcard = card;
end while;

Figure 2 MatBase pseudocode algorithm for preventing cycles in binary homogeneous Cartesian function products
execute("INSERT INTO " & TransClosure & "([Level], [" & a & "] SELECT " & level & " AS [Level], " & R & ".[" & fName & "] FROM " & R & "+ INNER JOIN " & TransClosure & " ON " & TransClosure & ".[" & a & "] = " & R & ".[" & fName & "] WHERE [Level] =" & level & "- 1); // add current level of fName "descendants"

execute("INSERT INTO " & TransClosure & "([Level], [" & a & "]) SELECT " & level & " AS [Level], " & R & ".[" & fName & "] FROM " & R & "+ INNER JOIN " & TransClosure & " ON " & TransClosure & ".[" & a & "] = " & R & ".[" & gName & "] WHERE [Level] =" & level & "- 1); // add current level of gName "descendants"

cycleLen = execute("SELECT [Level] FROM " & TransClosure & " WHERE [" & a & "] IN " & S);

if Not IsNull(cycleLen) then xFunctProdAcyclic = true; // cycle discovered!
else level = level + 1; // prepare next level of "descendants"

card = execute("SELECT Count (*) FROM " & TransClosure); // compute new (current) result cardinal

end if;
end while;

// b. add "ancestors" of x

oldCard = -1; // make sure that next while is entered at least once if no cycle yet detected

// initialize result with x's "parents"

if not IsNull(f) then execute("INSERT INTO " & TransClosure & "([Level], [" & a & "]) VALUES (-1, " & f & ");
if not IsNull(g) then execute("INSERT INTO " & TransClosure & "([Level], [" & a & "]) VALUES (-1, " & g & ");

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

card = execute("SELECT Count (*) FROM " & TransClosure); // current TransClosure's cardinal

while card # oldCard And Not xFunctProdAcyclic // loop until no "ancestor" is added or cycle is found

oldcard = card; // prevent infinite looping

// add current level of fName "ancestors"

execute("INSERT INTO " & TransClosure & "([Level], [" & a & "]) SELECT " & level & " AS [Level], " & R & ".[" & fName & "] FROM " & R & "+ INNER JOIN " & TransClosure & " ON " & TransClosure & ".[" & a & "] = " & R & ".[" & fName & "] WHERE [Level] =" & level & "+ 1 & " AND NOT " & R & ".[" & fName & "] IS NULL");

// add current level of gName "ancestors"

execute("INSERT INTO " & TransClosure & "([Level], [" & a & "]) SELECT " & level & " AS [Level], " & R & ".[" & gName & "] FROM " & R & "+ INNER JOIN " & TransClosure & " ON " & TransClosure & ".[" & a & "] = " & R & ".[" & fName & "] WHERE [Level] =" & level & "+ 1 & " AND NOT " & R & ".[" & gName & "] IS NULL");

// legitimate occurrences of "father" and "mother" are not closing cycles, so they must be deleted now

if level = -2 then prevcard = execute("SELECT Count (*) FROM " & TransClosure);
execute("DELETE FROM " & TransClosure & " WHERE [Level] = -1");

Figure 2 (Continued)
oldcard = oldcard – (prevcard – execute("SELECT Count (*) FROM " & TransClosure));

end if;

cycleLen = execute("SELECT [Level] FROM " & TransitiveClosure & " WHERE [" & a & "] IN " & S);

if Not IsNull(cycleLen) then xFunctProdAcyclic = true; // cycle discovered!
else // eliminate duplicates that might appear when z is a "ancestor" of a same y from both fName and gName
execute("DELETE From" & TransClosure "& " WHERE x IN (SELECT y FROM " & TransClosure & "Duplicates")");

end if;

end while;

if xFunctProdAcyclic then // display corresponding error message when a cycle is detected

display " Request rejected: these values for " & fName & " and/or " & gName & " would create a cycle of length " & Abs(cycleLen) + 1 & " in the graph of the Cartesian function product " & fName & " • " & gName & "; please change either " & fName & " or/and " & gName & ";!

end if;

End Function xFunctProductAcyclic;

Figure 2 (Continued)

<table>
<thead>
<tr>
<th>x</th>
<th>Level</th>
<th>RelatedPersons</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>-1</td>
<td>5</td>
</tr>
</tbody>
</table>

Figure 3 The instance of TransClosure for the first above example

R = "PERSONS"; xName = "x"; fName = "Mother"; gName = "Father"; TransClosure = "TransClosure"; a = "RelatedPersons"; x = 15; f = 14; g = 7; here is what would happen (corresponding TransClosure instance is shown in Figure 4):

- xFunctProductAcyclic = false;
- S = "(15, 14, 7)"
- execute("DELETE FROM TransClosure"); // TransClosure is emptied
- oldcard = 0;
- execute("INSERT INTO TransClosure([Level], [RelatedPersons]) SELECT 1 AS [Level], [x] FROM PERSONS WHERE [Mother] = 15 OR [Father] = 15"); // 1st line of TransClosure (Mihai)
- level = 2;
- card = 1;
- oldcard = 1;
- execute("INSERT INTO TransClosure([Level], [RelatedPersons]) SELECT 2 AS [Level], PERSONS.[x] FROM PERSONS INNER JOIN TransClosure ON TransClosure.[RelatedPersons] = PERSONS.[Mother] WHERE [Level] = 1"); // no line (Mihai is not a mother)
- execute("INSERT INTO TransClosure([Level], [RelatedPersons]) SELECT 2 AS [Level], PERSONS.[x] FROM PERSONS INNER JOIN TransClosure ON TransClosure.[RelatedPersons] = PERSONS.[Father] WHERE [Level] = 1"); // no line (Mihai is not a father)
Theorem: Let us pollute the transitive closure. Please note that Figure 4

Figure 4 The instance of TransClosure for the second above example (from which 2\textsuperscript{nd} and 3\textsuperscript{rd} lines were deleted)

Please note that xFunctProductAcyclic needs a TransClosureDuplicates SQL view to delete any duplicates that might pollute the transitive closure.

Let us now investigate the characterization of xFunctProductAcyclic:

**Theorem:** Algorithm xFunctProductAcyclic from Figure 2 has the following four properties:

(i) it is linear in the cardinal of the input relation
(ii) it is sound (i.e., it is not returning $true$, except when a $f \bullet g$ acyclicity violation would occur)
(iii) it is complete (i.e., it is returning $true$ whenever a $f \bullet g$ acyclicity violation would occur)
Proof:

(i) Obviously, as it has only two finite loops (so, as it also deletes any duplicates that might occur in the transitive closure, it never loops infinitely) depending on the sum of the heights in the graphs of \( f \) and \( g \), which are at most \( n = |R| \) (trivially, as db instances are finite, any such height is finite and at most equal to \( n - 1 \)); consequently, in the worst case (i.e. no null values for both \( f \) and \( g \) and no cycle found), the algorithm performs \( 2 \times (n - 1) \) steps, so its complexity is \( O(n) \).

(ii) Obviously, the only times when \( xFunctProductAcyclic \) returns true is when either in the subset of "descendants" or in the one of "ancestors" the transitive closure computation added either \( f(x) \) or/and \( g(x) \), on any other level than -1 (those of the "parents" of \( x \)), or/and, only when \( f \) and \( g \) are self-maps, the current element \( x \) from \( R \).

(iii) Obviously, as variable \( S \) stores any not null value of \( f(x) \) and \( g(x) \), plus, for self-maps, the one of \( x \) as well, any time when a cycle might be closed \( xFunctProductAcyclic \) returns true.

(iv) Obviously, for both while loops, 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 neither on a same level (as it joins to the input relation only the current result elements that were added in the previous iteration), nor on different ones (as, using the TransClosureDuplicates SQL view, duplicates are deleted as soon as they occur). Finally, \( xFunctProductAcyclic \) stops as soon as a would-be cycle is detected. q.e.d.

3. A MS Access VBA embedding SQL implementation of the \( xFunctProductAcyclic \) Boolean function

Figure 5 presents, as an example, the MatBase's MS Access VBA embedding SQL implementation of the \( xFunctProductAcyclic \) Boolean function (of the Constraints module) whose pseudocode algorithm is shown in Figure 2.

```
Public Function xFunctProdAcyclic(ByVal tblName As String, ByVal xName As String, _
ByVal fName As String, ByVal gName As String, ByVal codom As String, ByVal _
TransitiveClosure As String, ByVal RP as String, ByVal x As Long, _
f As Variant, g As Variant) As Boolean
'**********************************************************************************************
’Computes the transitive closure of the product fName x gName, having codomain codom ’for element x from table tblName, having primary key xName, into the temporary table ’TransitiveClosure, by using the least fixpoint semantics for recursive relational ’algebra equations, and looking for cycles in it (i.e. for the values of x, when fName ’and gName are self-maps, or for those of f and g, the current values of fName and ’gName, on any level, except the -1 one, i.e., the one of "parents"). ’Strategy: ’Both "children", "parent", "descendant", and "ancestor" are used here metaphorically, ’as most common example is when tblName = PEOPLE, fName = Mother, and gName = Father ’figure 5 MatBase MS Access VBA Algorithm for preventing cycles in binary homogeneous Cartesian function products
```

Figure 5 MatBase MS Access VBA algorithm for preventing cycles in binary homogeneous Cartesian function products
build text \((x, f, g)\) or \((f, g)\) or \((x, f)\) or \((f, g)\) or \((x, g)\) or \((g)\) as \(S\), eliminating 'null values for \(f\) and \(g\), and adding \(x\) only when \(f\) and \(g\) are self-maps

\[
S = IIf(codom = tblName, x, '')
\]

'are \(f\) and \(g\) self-maps?

\[
S = IIf(IsNull(f), S, IIf(IsNull(S), f, S & '', '' & f))
\]

'is \(f(x)\) not null?

\[
S = IIf(IsNull(g), S, IIf(IsNull(S), g, S & '', '' & g))
\]

'is \(g(x)\) not null?

If Not IsNull(S) Then 'nothing to do if \(S\) is null: no cycle possible!

\[
S = '' & S & ''
\]

'the set of values that should not be encountered in the 'transitive closure

'initialize transitive closure computation

If Len(CurrentDb.Tabledefs(TransClosure).Name & '') > 0 Then

DoCmd.RunSQL "DELETE FROM " & TransClosure

Else

DoCmd.RunSQL "CREATE TABLE " & TransClosure & ";(x COUNTER, [Level] INT, ['" & RPerson & "] INT)"

If Not addQuery(TransClosure & "Duplicates", "SELECT Count(['" & RPerson & "]) AS DupsNo, Max(x) AS y FROM " & TransClosure & " GROUP BY ['" & RPerson & "]") HAVING Count(['" & RPerson & "])>1") Then GoTo err_point

End If

DoCmd.RunSQL "DELETE FROM & TransitiveClosure 'initial instance is the empty set

oldCard = 0

'a. "add descendants" of \(x\)

DoCmd.RunSQL "INSERT INTO " & TransitiveClosure & ";(['" & RPerson & "], [Level]) = (" & x & ") OR ['" & fName & "] = " & x & ' compute "children"s cardinal

card = DCount("x", TransitiveClosure)

level = 2 'initialize genealogy tree level with the next one ("grandchildren")

'loop until no descendant are added (i.e. compute least fixpoint) or cycle found

While card <> oldCard And Not xFunctProdAcyclic

'prepare checking whether or not new descendants will be added

oldCard = card

'add current level of \(f\)Name descendants

DoCmd.RunSQL "INSERT INTO " & TransitiveClosure & ";(['" & RPerson & "], [Level]) = (" & x & ") OR ['" & fName & "] = " & x & ' compute "children"s cardinal

card = DCount("x", TransitiveClosure)

level = level + 1 'increase "descendants" level

End If

Wend

'b. add "ancestors" of \(x\)

oldCard = -1 'make sure that next while is entered at least once

'if no cycle yet detected

'initialize result with \(x\)'s parents

Figure 5 (Continued)
If Not IsNull(f) Then DoCmd.RunSQL "INSERT INTO " & TransClosure & "([Level], [" & RPerson & "]) VALUES (-1, " & f & ")"
If Not IsNull(g) Then DoCmd.RunSQL "INSERT INTO " & Trans Closure & "([Level], [" & RPerson & "]) VALUES (-1, " & g & ")"
level = -2 'next step will add second level "ancestors"
card = DCount("*", TransClosure)
'loop while there exists ancestors of x and no cycle detected
While card <> oldCard And Not xFunctProdAcyclic
    oldCard = card 'prepare checking whether or not new ancestors will be added
    'add current level of fName ancestors
    DoCmd.RunSQL "INSERT INTO " & TransClosure & "(" & fName & ")", [Level])" _ & 
    "SELECT " & tblName & ",[" & fName & "], " & level & " AS [Level] FROM " _ & 
    TransClosure & " INNER JOIN " & tblName & " ON " & TransClosure & _ & ".[" & RPerson & "] = " & tblName & ",[" & xName & "] WHERE [Level]=" & _ & 
    level + 1 & " AND NOT " & tblName & ".[" & fName & "] IS NULL"
    'add current level of fName ancestors
    DoCmd.RunSQL "INSERT INTO " & TransClosure & "(" & fName & ")", [Level])" _ & 
    "SELECT " & tblName & ",[" & gName & "], " & level & " AS [Level] FROM " _ & 
    TransClosure & " INNER JOIN " & tblName & " ON " & TransClosure & _ & ".[" & RPerson & "] = " & tblName & ",[" & xName & "] WHERE [Level]=" & _ & 
    level + 1 & " AND NOT " & tblName & ".[" & gName & "] IS NULL"
    'legitimate occurrences of "father" and "mother" are not closing cycles,
    'so they must be deleted now
    If level = -2 Then
        prevCard = DCount("*", TransClosure)
        DoCmd.RunSQL("DELETE FROM " & TransClosure & " WHERE [Level] = -1")
        oldCard = oldCard - (prevCard - DCount("*", TransClosure))
    End If
    cycleLen = DLookup("[Level]", TransClosure, RPerson & " IN " & S)
    If Not IsNull(cycleLen) Then
        xFunctProdAcyclic = True 'cycle discovered!
    Else
        'eliminate duplicates (that might appear when a "person" is an "ancestor" of
        'a same "person" from both his/her "mother" and "father"
        DoCmd.RunSQL "DELETE FROM " & TransClosure & " WHERE x IN (SELECT y " & _ 
        "FROM " & TransClosure & "Duplicates)"
        card = DCount("*", TransClosure) 'compute new (current) result cardinal
        level = level - 1 'decrease ancestor level
    End If
    Wend
    'display corresponding error message when a cycle is detected
    If xFunctProdAcyclic Then MsgBox "These values for " & fName & " and/or " & gName _ & " would create a cycle of length " & Abs(cycleLen) + 1 & " in the " & _ 
    "genealogical tree of this person: please change either " & fName & " or/and " _ & 
    "& gName & ":!", vbCritical, "Request rejected..."
    End If
End If
exit_point: Exit Function
err_point: MsgBox Err.Source & "->" & Err.Description, vbCritical, "Error in " & _ 
"method xFunctProdAcyclic of module Constraints...
xFunctProdAcyclic = True
End Function

Figure 5 (Continued)

For example, the call to this function for the second example above (see Figure 4) is performed from the PERSONs's GUI class Form_BeforeUpdate method as shown in Figure 6.
Private Sub Form_BeforeUpdate(Cancel As Integer)
'enforces constraint Mother x Father acyclic
If Not Cancel And ((Not IsNull(Mother) And (IsNull(Mother.OldValue) Or Mother <> Mother.OldValue)) Or (Not IsNull(Father) And (IsNull(Father.OldValue) Or Father <> Father.OldValue))) Then
    Cancel = xFunctProdAcyclic("PERSONS", "x", "Mother", "Father", "PERSONS", "TransitiveClosure", "RelatedPersons", 15, 14, -7)
End Sub

Figure 6 An example of a MS Access VBA call to the Boolean function xFunctProductAcyclic from Figure 5

A library should contain the declaration of the Boolean function addQuery (called by xFunctProductAcyclic) shown in Figure 7. For it to run, the corresponding MS VBA project must include optional libraries MS ActiveX Data Objects (ADODB) and ADO Ext. for DDL and Security (ADOX), both, preferably, having versions at least 6.0.

Public Function addQuery(ByVal PQueryName As String, ByVal PQuerySQLTxt As String) As Boolean
'adds to the current db a query named PQueryName having the body PQuerySQLTxt
'returns True if creation succeeds or False otherwise
Dim objCatalog As ADOX.Catalog
Dim objCommand As ADODB.Command
Dim PConnection As ADODB.Connection
On Error GoTo err_point
Set objCatalog = New ADOX.Catalog
objCatalog.ActiveConnection = CurrentProject.Connection
Set objCommand = New ADODB.Command
objCommand.CommandType = adCmdText
objCommand.CommandText = PQuerySQLTxt
objCatalog.Views.Append PQueryName, objCommand
Set objCommand = Nothing
Set objCatalog = Nothing
addQuery = True
Exit Function
err_point: MsgBox Error()
    addQuery = False
End Function

Figure 7 MatBase MS Access VBA public function for dynamically creating queries (views)

4. Conclusion

This paper provides an algorithm (both in pseudocode and in MS Access VBA embedding SQL) for enforcing acyclicites of binary homogeneous Cartesian function products, examples of using it for a genealogical db, as well as a formal proof that this algorithm is linear, sound, complete, and optimal. This algorithm is embedded in MatBase, an intelligent knowledge and db management system prototype designed and developed by us.

Declaring and enforcing acyclicites of binary homogeneous Cartesian function products further contribute to guaranteeing db instances quality. Consequently, it is our firm belief that this type of non-relational constraints should also be added to all DBMSes, so that developers need not enforce them through their code.
Compliance with ethical standards

Acknowledgments
The author declares that no funds, grants, or other support were received during the preparation of this manuscript.

Disclosure of conflict of interest
The author has no relevant financial or non-financial interests to disclose.

References


