World Journal of Advanced Engineering Technology and Sciences
eISSN: 2582-8266
Cross Ref DOI: 10.30574/wjaets
Journal homepage: https://wjaets.com/

(RESEARCH ARTICLE)

On variable geometry database keys and their subkeys

Christian Mancas *

DATASIS ProSoft srl, Bucharest, Romania.

World Journal of Advanced Engineering Technology and Sciences, 2022, 06(02), 071–080

Publication history: Received on 25 June 2022; revised on 28 July 2022; accepted on 30 July 2022

Article DOI: https://doi.org/10.30574/wjaets.2022.6.2.0086

Abstract

This paper introduces variable geometry database (db) keys and their subkeys, both mathematically and in db terminology, provides an algorithm for assisting their discovery, characterizes it, and presents methods for enforcing variable geometry keys and their subkeys in db applications using SQL embedding event-driven programming languages, with an example in MS VBA.

Keywords: Database constraint design; Database constraint enforcement; Database variable geometry key; Database subkey; *MatBase*; The (Elementary) Mathematical Data Model

1. Introduction

Key is the most important type of database (db) constraint. By (*unique*) *key*, db theory and management systems (DBMS) understand both single ones, i.e. table columns that do not allow for duplicates, and concatenated (composite) ones, i.e. sets of table columns that do not allow for duplicates.

Mathematically, single keys are one-to-one functions, while concatenated ones should be minimally one-to-one function products [1]. Any non-minimal one-to-one product is called a *superkey* in dbs. Unfortunately, most DBMS versions allow for declaring both keys and superkeys (which is taking unneeded storage space and processing time).

1.1. Keys and foreign keys

Even more unfortunately, especially beginners are confused by some DBMS versions that are grouping together (unique) keys (that we will refer from now on as, simply, *keys*) and the so-called *foreign keys* (i.e. either table columns or table column sets that are referencing other ones of same or compatible types, generally from another table, but sometimes from the same one).

As we have already shown [1, 2], these are in fact orthogonal concepts (e.g. in a *COUNTRIES* table, column *CountryName* is a key, but not a foreign one, *Continent* is a foreign key, but not a key one, *Population* is neither a key, nor a foreign key, and *Capital* is both a key and a foreign key). In this paper we are only interested in keys and not in foreign keys.

1.2. Keys and null values: variable geometry db keys and their subkeys

Mathematically, functions are not only functional, but also totally defined (i.e. for each element in their domain there is one and only one associated element from their codomain). In dbs, as rather often it is the case that we do not know, at least temporarily, or do not care for some column values, or they are not possible in some contexts, columns are functional, but not total: totality is considered a constraint type, generally denoted NOT NULL.

* Corresponding author: Christian Mancas; Email: christian.mancas@gmail.com
DATASIS Pro Soft srl, Bucharest, Romania.

Mathematically, this implies the existence of a countable distinguished set NULLS that is merged with the base table column codomains. For example, again in a table *COUNTRIES*, a column *MaxAltitude* would not be declared NOT NULL, so the corresponding function would be $MaxAltitude : COUNTRIES \rightarrow [0, 8850] \cup NULLS$.

Normally, being infinitely many (and distinct between them), null values (nulls) should not interact with db single keys. Fortunately, for example, in MS Access you can still declare *Capital* a key, even if it is not NOT NULL, and store in it as many null values as you wish (including for all the rows of the *COUNTRIES* table). Unfortunately, for example, MS SQL Server assumes that there is a single null value, so you can still declare *Capital* as a key, but it cannot store more than one null value at any moment. The only exception that is uniformly treated by all DBMS versions are the primary keys, which do not accept nulls by definition and which may be declared in any table, but there may be only one such key per table.

However, even theoretically, null values sometimes heavily interact with db concatenated keys. For example, in any operating system in which files reside in folders, should have a name, and might have an optional extension, and in which folders are files as well, in any folder there may not be two files having either same name and extension, or same name without extensions.

Mathematically, this means that not only $Folder \bullet FName \bullet FExt : FILES \rightarrow FILES \times ASCII(255) \times ASCII(255) \cup NULLS$ is minimally one-to-one, but also $Folder \bullet FName : FILES \rightarrow FILES \times ASCII(255)$ is minimally one-to-one for any $x \in FILES$ for which $FExt(x) \in NULLS$. Please note, however, that $Folder \bullet FName \bullet FExt$ is not a superkey of $Folder \bullet FName$, as there may be in any folder two files having same name, but distinct not-null extensions or one extension not-null and the other one null, so $Folder \bullet FName \bullet FExt$ is minimally one-to-one.

We call keys like $Folder \bullet FName \bullet FExt$ *variable geometry keys*, i.e. concatenated keys that have at least one non-totally defined member, without which they are still keys whenever that member has null values, and constraints like $Folder \bullet FName$ minimally one-to-one for any $x \in FILES$ for which $FExt(x) \in NULLS$ *subkeys*.

Denotationally, we write $Folder \bullet FName \bullet FExt$ *key* (which is a shortcut for $(\forall x,y \in FILES, x \neq y)(Folder(x) \neq Folder(y) \vee FName(x) \neq FName(y) \vee FExt(x) \neq FExt(y))$) and $Folder \bullet FName$ *subkey whenever FExt IS NULL* (which is a shortcut for $(\forall x,y \in FILES, x \neq y)(FExt(x) \in NULLS \wedge FExt(y) \in NULLS \Rightarrow Folder(x) \neq Folder(y) \vee FName(x) \neq FName(y))$), respectively.

Generally, given a key $f_1 \bullet \ldots \bullet f_n$, $n > 1$, a *subkey* of it is a constraint of the form $(\forall x,y \in dom(f_1 \bullet \ldots \bullet f_n), x \neq y)(f_1(x) \in NULLS \wedge f_1(y) \in NULLS \wedge \ldots \wedge f_m(x) \in NULLS \wedge f_m(y) \in NULLS \Rightarrow f_{m+1}(x) \neq f_{m+1}(y) \vee \ldots \vee f_n(x) \neq f_n(y))$, where $1 \leq m < n$, m and n naturals; then, formally, a *variable geometry key* is a concatenated key that has at least one associated subkey.

Obviously, if we denote by s the number of subkeys of a key $f = f_1 \bullet \ldots \bullet f_n$ having at least one non-total member, according to Newton's binomial, $0 \leq s \leq 2n - 2$, as there may be none and at most $C(n, 1) + \ldots + C(n, n - 1)$ subkeys of it; if f has i non-total members, $1 \leq i \leq n$, then, if f is a variable geometry key, $1 \leq s \leq 2i - 1 = C(i, 0) + \ldots + C(i, i - 1)$.

All business rules (constraints) that are governing the sub-universes modelled by dbs should be discovered and enforced in the corresponding dbs' schemas and db managing applications: otherwise, their instances might be implausible. Besides the key type ones, the subkeys of variable geometry keys also play a cornerstone role, both practically and theoretically.

Unfortunately, not only minimal uniqueness is both relative, highly dependent on the context, semantic, hence only discoverable by humans, and not that easy to fully detect, as the complexity of this process is exponential in the number of involved table prime columns, but, as shown in the second section of this paper, subkeys are too.

Consequently, it is our firm belief that any purely syntactic approach to inferring keys and/or subkeys may not be successful. Fortunately, math and computer science may assist db designers in both validating existing enforced keys and discovering all those that are possibly missing, including subkeys of the variable geometry ones, by algorithmically guiding them not to miss any possible key and subkey, not to waste time with either non-prime attributes or superkeys, or NOT NULL ones, as well as when to safely stop looking for keys, as none others might be discovered afterwards.

1.3. Related work

Keys, foreign keys, and null values have been extensively studied for the last half of century, e.g. [1, 2, 3]. Interactions between keys and null values have also been investigated (e.g [4, 5, 6, 7, 8, 9, 10]), but, to our knowledge, variable geometry keys and their subkeys were not.

The subkey constraint type presented here was introduced in the framework of the (Elementary) Mathematical Data Model [11] and is completely different from the homonym concept used in the design of normal forms in the framework of the Relational Data Model [3].

1.4. Paper outline

The second section of this paper investigates variable geometry keys and their subkeys discovery, while the third one provides hints on their enforcement. The paper ends with conclusion and references.

2. Variable geometry keys and their subkeys discovery

Currently, there are two main approaches for keys discovery [2]: data mining (syntactical) and db design techniques (semantical). As advocated in [2], we strongly favor the semantical approach, mainly as only humans can decide whether a function should be one-to-one or a function product should be minimally one-to-one in a given sub-universe of discourse, but also to guarantee plausibility of db instances even before they are created, as well as to avoid both false negative and false positive keys (obtained through data mining). Consequently, we proposed a series of algorithms for key discovery assistance [1, 2].

For example, variable geometry keys and their subkeys may be discovered with only an addition at the end of Algorithm *A*3 from [2] shown in Figure 1 (i.e. pseudo-code statements between 16 and 28). The main ideas behind this addition to *A*3 are the following two ones:

- Variable geometry keys can be discovered only after discovering a concatenated key that has at least one non-total member.
- There may be several non-total members in a function product, so there may be several subkeys in a variable geometry key, all of which should be discovered, which means that for any concatenated key that has *i* not NOT NULL members, all its possible $2^i - 1$ combinations should be investigated.

ALGORITHM *A*4 Keys Discovery Assistance, Including Variable Geometry Ones

Input: a set of *n* prime not key columns $S = \{c_1, c_n\}$ of a same table *T* and a set of associated keys *K*, $card(K) = k$,

k and *n* naturals, $n > 1$.

Output: *K'*, the set of all the keys, and *SK*, the set of all the subkeys of *T*.

01. $K' = K$;

02. if $n > 0$ and $c_1 \bullet \ldots \bullet c_n$ is unique then // for all mappings, if any, look for keys

03. if $c_1 \bullet \ldots \bullet c_n$ is minimally unique (in the given context) then

04. $K' = K' \cup \{c_1 \bullet \ldots \bullet c_n\}$; // add newly found key

else

05. $kmax = C(n, [n/2])$; // maximum possible numbers of keys

06. $i = 1$; // starting column products arity

07. *allSuperkeys* = *false*; // initially, no superkeys possible for $i = 1$

08. while $i < n$ and $l < kmax$ and *not allSuperkeys* do

09. *allSuperkeys* = *true*; // all $C(n, i)$ combinations might be superkeys

10. repeat for all $C(n, i)$ mapping products *p* made out of *i* elements

11. if *p* is not a superkey then // at least one no superkey discovered on

12. *allSuperkeys* = *false*; // the current level *i*
13. if *p* is minimally unique (in the given context) then
14. $K' = K' \cup \{ p \}$; // add newly found key
end if;
end if; // (of 11.: *if p* is not a superkey)
end repeat;
15. $i = i + 1$; // increment level (mapping products arity)
end while;
end if; // (of 03.: *if* $c_1 \bullet \ldots \bullet c_n$ is minimally unique)
end if; // (of 02.: *if* $n > 0$ and $c_1 \bullet \ldots \bullet c_n$ unique)
16. $SK = \emptyset$; // initialize *SK*
17. for $j = 1$ to $card(K')$ // discover variable geometry keys
18. $a = arity(K'(j))$; // $K'(j) = c_1 \bullet \ldots \bullet c_a$
19. if $a > 1$ then // is current key a concatenated one?
20. $i = 0$; // number of not NOT NULL columns
21. for $m = 1$ to a
22. if c_m is not NOT NULL then // does current key member allow nulls?
23. $i = i + 1$;
end if;
end for; // (of 21: for $m = 1$ to a)
end if; // (of 19: if $a > 1$)
24. if $i > 0$ then // $K'(j)$ is a possible variable geometry key
25. for $s = 0$ to $2^i - 2$ // investigate all possible subkeys of $K'(j)$
26. generate $sk(s)$, the s-th possible combination of $K'(j)$'s columns;
27. if $sk(s)$ is a subkey (in the given context) then
28. $SK = SK \cup \{sk(s)\}$; // another subkey discovered
end if;
end for; // (of 25: for $s = 0$ to $2^i - 2$)
end if; // (of 24: if $i > 0$)

end for; // (of 17: for j = 1 to *card* (K'))

End ALGORITHM $A4$

Figure 1 Algorithm $A4$ (Best practical keys discovery assistance algorithm, including variable geometry keys and their subkeys)

For example, let us apply Algorithm $A4$ from Figure 1 with the above input: T = *FILES*; S = {*Folder*, *FName*, *FExt*}; n = *card*(S) = 3; $K = \varnothing$; k = *card*(K) = 0:

01. $K' = \varnothing$;

02. if 3 > 0 and *Folder* • *FName* • *FExt* is unique then // yes, it is

03. if *Folder* • *FName* • *FExt* is minimally unique then // yes, it is

04. K' = {*Folder* • *FName* • *FExt*};

end if; // (of 03. if *Folder* • *FName* • *FExt* is minimally unique)

end if; // (of 02. if 3 > 0 and *Folder* • *FName* • *FExt* unique)

16. $SK = \varnothing$;

17. for j = 1 to 1 // *card* (K') = 1

18. a = 3; // $K'(1)$ = *Folder* • *FName* • *FExt*

19. if 3 > 1 then

20. i = 0;

21. for m = 1 to 3

22. if *Folder* is not NOT NULL end if; // *Folder* is NOT NULL

22. if *FName* is not NOT NULL end if; // *FName* is NOT NULL

22. if *FExt* is not NOT NULL then // *FExt* is not NOT NULL

23. i = 1;

end if;

end for; // (of 21: for m = 1 to 3)

end if; // (of 19: if 3 > 1)

24. if 1 > 0 then // $K'(1)$ is a possible variable geometry key

25. for s = 0 to 0

26. *sk* (0) = *Folder* • *FName whenever FExt IS NULL*;

27. if *Folder* • *FName* is a *subkey whenever FExt IS NULL* then // yes, it is

28. *SK* = {*Folder* • *FName subkey whenever FExt IS NULL*};

```
        end if;
      end for;                    // (of 25: for s = 0 to 0)
    end if;                       // (of 24: if 1 > 0)
  end for;                        // (of 17: for j = 1 to 1)
```

Let now us investigate the characterization of *A*4, starting with the one of *A*3 [1, 2] (which is made up of statements 01 to 15 from Figure 1):

Theorem *TA*3 (*Keys Discovery Algorithm Characterization*)

The algorithm *A*3 has the following properties:

- its complexity is $O(2n)$;
- it is complete (i.e. it is generating all possible keys);
- it is sound (i.e. it is generating only possible keys);
- it is relatively optimal (i.e. it generates the minimum possible number of questions to users relative to its strategy, i.e. only one per possible key, and it is doing its job with minimum number of statement executions).

First, let us note that the added sub-algorithm made of statements between 16 to 28 always stops in finite time:

- 17 runs exactly $l = card(K')$ times, $0 \le l \le C(n, [n/2])$, naturals [1, 2];
- 21 runs exactly $a = arity(K'(j))$ times, $2 \le a \le n$, $1 \le j \le$ l, naturals;
- 25 runs exactly $2^i - 1$ times, $0 \le i \le a$, naturals.
- Consequently, the complexity of this sub-algorithm is $O(C(n, [n/2]) * (n + 2^n))$.
- 25 to 28 are generating all possible subkeys of a key; 17 runs 25 to 28 for all table keys; consequently, *A*4 is complete for the subkeys as well.
- 21 to 24 eliminate all single keys and all concatenated ones that do not have at least one not NOT NULL column; consequently, *A*4 is sound for subkeys as well.
- 16 to 28 inspects any key only once, generates any possibly subkey and asks users if it should be one only once, so *A*4 is relatively optimal for subkeys as well.

This proves that A4 is characterized by Theorem *TA*4:

Theorem *TA*4 (*Keys and Subkeys Discovery Algorithm Characterization*)

The algorithm *A*4 has the following properties:

- its complexity is $O(C(n, [n/2]) * (n + 2^n))$;
- it is complete (i.e. it is generating all possible keys and their subkeys);
- it is sound (i.e. it is generating only possible keys and their subkeys);
- it is relatively optimal (i.e. it generates the minimum possible number of questions to users relative to its strategy, i.e. only one per possible key and possible subkey, and it is doing its job with minimum number of statement executions, both for keys and their subkeys).

The exponential complexity of algorithm *A*4 should not scare anybody: *n*, the number of prime columns of any fundamental db table is generally small, of one digit only, out of which generally more than half are NOT NULL; hence, the corresponding maximum number of keys is $C(9, 4) = 126$, while the maxim number of subkeys per variable geometry key is $2^4 - 1 = 15$.

Practically, in our over 40 years of real life db modelling and db software application architecture, design, and development, even in large dbs having over 1000 fundamental tables, in average we discovered some 3 keys per table, not more than one variable geometry key per 10 tables, with no more than 2 subkeys per such keys.

Consequently, it is our firm belief that any DBMS should implement at least *A*4 to assist its users in discovering all keys and subkeys in any sub-universe modelled by a db. Algorithm *A*4 is already implemented in *MatBase* [12], a prototype

intelligent Knowledge and DBMS. Consequently, its users do not need to bother with either missing combinations, detecting superkeys, skipping variable geometry ones or their subkeys, or in vain looking for subkeys of not variable geometry keys. Moreover, *MatBase* automatically generates needed SQL and C# / VBA code for enforcing both keys and their subkeys.

It is true that, however, discovering of all keys and subkeys of a large db, even when benefitting from working with an intelligent DBMS as *MatBase*, is requiring an important amount of db modelling and architectural effort, but it is a one-time one and it then guarantees the data plausibility forever, so we consider it as extremely worthwhile.

3. Variable geometry keys and their subkeys enforcement

The best solution for enforcing subkeys is developing corresponding methods to be automatically invoked by the event-driven methods of type *Validating* (in MS .NET), *BeforeUpdate* (in MS VBA), etc. from the classes attached to the GUI forms managing the data from the corresponding tables.

For example, the subkey *Folder* • *FName* could be enforced in MS VBA (the simplest event-driven programming language) by the following function *subkeyFolder_FName*, called from the *Form_BeforeUpdate* event-driven method of the class *Form_FILES* managing data from table *FILES* (assuming that the primary key of it is denoted *x*):

```
Sub Form_BeforeUpdate (Cancel As Integer)

Cancel = subKeyFolder_FName ()

End Sub

Function subKeyFolder_FName() As Boolean

Dim v as Variant

subKeyFolder_FName = False

If IsNull(FExt) Then

  v = DLookup("x", "FILES ", "x <> " & x & " AND FExt Is Null and Folder = " & Folder & " AND FName = '" & FName & "'")

  If Not IsNull(v) Then

    subKeyFolder_FName = True

    Beep

    MsgBox "There is already a file in this folder having this name and no extension!", vbCritical, _

          "Please change either folder or file name, or add a file extension…"

  End If

End If

End Function
```

Each time users would try to create a new file or change the name and/or the extension of and/or copy or move an existing one in another folder, the system is automatically invoking the above *Form_BeforeUpdate* method; if, when finishes, *Cancel* is *False*, the requested operation is performed; otherwise, it is not, as the subkey *Folder* • *FName* would be violated.

The function *subKeyFolder_FName* returns *False* whenever corresponding *FExt* is not null (i.e. users also specified an extension for the current file) or it is null, but in the *Folder* folder there is no other file having same file name and no extension; otherwise, it displays the corresponding error message and returns *True*.

The syntax of its *DLookup* function is equivalent to the following SQL query:

SELECT x FROM FILES WHERE x <> cr.x AND FExt IS NULL AND Folder = cr.Folder AND FName = "cr.FName";

Where *cr* is the current row from the table *FILES*. Whenever the result of such queries is the empty set, *DLookup* returns null.

The corresponding pseudocode algorithm is the following:

Boolean Function *subKeyFolder_FName* (table *FILES*, int *x*, int *Folder*, text *FName*, text *FExt*)

// returns *true* if *FExt* is null and, in *FILES*, there is in *Folder* another file having same *FName* as file *x*;

// otherwise, returns *false*

subKeyFolder_FName = *false*;

if *FExt* ∈ NULLS then

if ∃*y*∈*FILES* such that *y* ≠ *x* and *FExt*(*y*) ∈ NULLS and *Folder*(*y*) == *Folder* and *FName*(*y*) == *FName* then

subKeyFolder_FName = *true*;

display “There is already a file in this folder having same name and no extension: please change

either folder or file name, or add a file extension!”;

endif;

endif;

The variable geometry key *Folder* • *FName* • *FExt*, can be simply enforced in any DBMS that correctly assumes NULLS being infinite (e.g. MS Access) just like any other key, with the following SQL statement:

ALTER TABLE FILES ADD CONSTRAINT keyFolderFNameFExt UNIQUE (Folder, FName, FExt);

In all other DBMS (which either assume that there is only one null value, e.g. MS SQL Server, Oracle, etc. or does not allow not NOT NULL columns in their keys) they may be enforced together with their subkeys. For example, the following VBA method *varGeomKeyFolder_FName_FExt* is enforcing both the above variable geometry key and its subkey:

```
Function varGeomKeyFolder_FName_FExt() As Boolean

Dim v As Variant

varGeomKeyFolder_FName_FExt = False

If IsNull(FExt) Then

 v = DLookup("x", "FILES ", "x <> " & x & " AND FExt Is Null AND Folder = " & Folder & " AND FName = '" & FName & "'")

 If Not IsNull(v) Then

  varGeomKeyFolder_FName_FExt = True
```

```
   Beep
   MsgBox "There is already a file in this folder having this name and no extension!", vbCritical, _
          "Please change either folder or file name, or add a file extension…"
  End If
Else
  v = DLookup("x", "FILES", "x <> " & x & " AND FExt = '" & FExt & "' AND Folder = " & Folder & " AND FName = '" & _
                                                                                              FName & "'")
  If Not IsNull(v) Then
   varGeomKeyFolder_FName_FExt = True
   Beep
   MsgBox "There is already a file in this folder having these name and extension!", vbCritical, _
          "Please change either folder, or file name, or extension…"
  End If
End If
End Function
```

The equivalent pseudocode method is the following:

Boolean Function *varGeometryKeyFolder_FName*(table *FILES*, int *x*, int *Folder*, text *FName*, Text *FExt*)

// returns *true* if, in *FILES*, there is in *Folder* another file having same *FName* and *FExt* as file *x* or if both of them have

// no extension; otherwise, returns *false*

varGeometryKeyFolder_FName = *false*;

if *FExt* ∈ NULLS then

 if ∃*y*∈*FILES* such that *y* ≠ *x* and *FExt*(*y*) ∈ NULLS and *Folder*(*y*) == *Folder* and *FName*(*y*) == *FName* then

  subKeyFolder_FName = *true*;

  display “There is already a file in this folder having same name and no extension: please change

       either folder or file name, or add a file extension!”;

 endif;

else

 if ∃*y*∈*FILES* such that *y* ≠ *x* and *FExt*(*y*) == *FExt* and *Folder*(*y*) == *Folder* and *FName*(*y*) == *FName* then

  varGeometryKeyFolder_FName = *true*;

```
    display “There is already a file in this folder having same name and extension: please change
        either folder, or file name, or extension!”;
  endif;
endif;
```

4. Conclusion

This paper provides db and mathematical definitions for both variable geometry keys and their subkeys, an extension to, in our opinion, the best algorithm assisting keys discovery for also discovering the variable geometry ones and their subkeys, a Theorem that characterizes this algorithm, as well as programmatical solutions for enforcing the variable geometry keys and their subkeys for both DBMS types – i.e. assuming either an infinite number of nulls or only one.

Declaring and enforcing subkeys as well further contribute to guaranteeing db instances quality. Consequently, subkeys should 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

[1] Mancas C. (2015). Conceptual Data Modeling and Database Design: A Completely Algorithmic Approach. Volume I: The Shortest Advisable Path. Apple Academic Press / CRC Press / Francis & Taylor, Waretown, NJ.

[2] Mancas C. (2016). Algorithms for Database Keys Discovery Assistance. In: Řepa V, Bruckner T (eds) Perspectives in Business Informatics Research. BIR 2016. LNBIP 261: 322–338.

[3] Date CJ. (2013). Relational Theory for Computer Professionals: What Relational Databases Are Really All About? O’Reilly Media, Sebastopol, CA.

[4] Thalheim B. (1989). On semantic issues connected with keys in relational databases permitting null values. J. Inform. Process. Cybernet. 25: 11–20.

[5] Philip GC. (2002). Normalization of Relations with Nulls in Candidate Keys. JDN 13(3): 35–45.

[6] Hartmann S, Leck U, Link S. (2010). On Codd Families of Keys over Incomplete Relations. The Computer Journal 54(7): 1166–1180.

[7] Thalheim B, Schewe KD. (2010). NULL 'Value' algebras and logics. Frontiers in Artificial Intelligence and Applications 225: 354–367.

[8] Kohler H, Link S, Zhou X. (2015). Possible and Certain SQL Keys. In: Proc. of the VLDB Endowment 8: 1118–1129.

[9] Alattar M, Sali A. (2019). Keys in Relational Databases with Nulls and Bounded Domains. In: Weltzer T, Eder J, Podgorelec V, Kamišalić Latifić A (eds), Advances in Databases and Information Systems, ADBIS 11695: 33–50.

[10] Al-Atar MH. (2021). Key and Functional Dependency Constraints for Incomplete Databases with Limited Domains. Ph.D. Dissertation Booklet, Budapest Univ. of Techn. & Econ., Dept. Comp. Sci. & IT.

[11] Mancas C. (2018). *MatBase* Constraint Sets Coherence and Minimality Enforcement Algorithms. In: Benczur A, Thalheim B, Horvath T (eds.), Proc. 22nd ADBIS Conf. on Advances in DB and Inf. Syst., LNCS 11019, 263–277.

[12] Mancas C. (2019). *MatBase* – a Tool for Transparent Programming while Modelling Data at Conceptual Levels. In: Meghanathan N et al (eds.), Proc. CSITEC 2019, 15–27.