NULL 'Value' Algebras and Logics

Bernhard THALHEIM [a,1] and Klaus-Dieter SCHEWE [b,2]
[a] *Christian-Albrechts-University Kiel, Computer Science Institute, 24098 Kiel, Germany*
[b] *Software Competence Center Hagenberg, 4232 Hagenberg, Austria*

Abstract. NULL is a special marker used in SQL to indicate that a value for an attribute of an object does not exist in the database. Its aim is a representation of "missing information and inapplicable information". Although NULL is called null 'value' is not at all a value. It is a marker. It is only an annotation of incomplete data. Since it is typically interpreted as a value, NULL has led to controversies and and debates because of its treatment by 3-valued logics, of its special requirements for its use in SQL joins, and the special handling required by aggregate functions and SQL grouping operators.

The three-valued logics does not properly reflect the nature of this special marker. Markers should be based on their specific data type. This data type is then different from any other data types used in relational database technology. Due to this orthogonality we can combine any type with the special type. To support this we introduce a non-standard generalisation of para-consistent logics. This logics reflects the nature of these markers. This paper aims in developing a general approach to NULL 'values' and shows how they can be used without changing database technology.

Keywords. NULL, NULL 'value', constraints, NULL logics, database engineering,

1. Introduction

In the early 1970s E. F. Codd introduced the relational database model including the relational calculus, relational algebra, and relational database normalization. Later he also introduced an approach to handle queries in the presence of NULL 'values' in a relational database and proposed a 3-valued logic with truth values 'True', 'False', and 'Unknown'. When a NULL value appears in a table, its evaluation in a condition produces the 'Unknown' truth value. He gave truth values to complex conditions by giving truth-tables for the connectives '`AND`', '`OR`', and '`NOT`'. For example, 'True `OR` Unknown' has the truth value 'True' because a disjunction is true if one of its disjuncts is true. Codd evaluated 'Unknown `OR` Unknown' to 'Unknown'. This specific Łukasiewicz logic $\mathfrak{L}_3$ was the starting point for a rather difficult treatment of missing or incomplete values in the database literature and technology.

[8] states that "nulls are ipso facto nonsense". "I apologize for the wording "contains a null"; as I've written elsewhere, to talk about anything "containing a null" actually

[1] thalheim@is.informatik.uni-kiel.de http://www.is.informatik.uni-kiel.de/∼ thalheim
[2] kd.schewe@scch.at http://www.scch.at/

makes no logical sense. Indeed one of the problems with nulls is precisely that you can't talk about them sensibly! ... the entire topic is a perfect illustration of The *Principle of Incoherence* ... ". The third manifesto [7,9] is an attempt to clarify and bring up to date E.F. Codd's 1970 model. It requires every tuple of a relation to contain exactly one value for each attribute of that relation, the value being "drawn from the domain" (Codd) or, synonymously, "of the declared type" (Date and Darwen) of that attribute.

1.1. NULL 'Values' in the Relational Database Model

NULL 'values' can be used for attributes in the relational model. They are often considered to be a programmer's nightmare. Allowing NULL 'values' into attribute values introduces a whole new degree of uncertainty into your database. Qualified guesses must be made by the SQL programmer to counter for erroneous results of NULL values in a database. We need to distinguish at least between the following kinds of NULLs:

- NULL 'values' represent currently *unknown values* that may be replaced later with values when we know something. These NULL values can be represented by specific default values. For example, *Gender* can be coded by the following scheme: 0 (unknown), 1 (male), 2 (female), 9 (inapplicable).
- Domain-specific NULL 'values' are used to denote ordinal or cardinal numbers. Ordinal numbers measure position. Cardinal numbers measure quantity or magnitude. There is a difference between the quantity 0 and an unknown quantity. 0 is the common default value for all numeric domain types. The blank can be used as a default type for character types. Date and time is specified by relative values and required by the schema to be absolute. In this case, NULL 'values' are not the appropriate solution. We split the corresponding attributes.
- NULL 'values' are also used to represent *inapplicability* of a characterization for a given object. In this case, hierarchies can be used for separation of aspects.

NULL 'values' can be derived. For instance, if two values are incomparable then the comparison evaluates to 'unknown' or 'null'. For example, the color of a car and the color of hair can be incomparable in the application. From the other side, NULL 'values' used for characterization of properties of different objects can be equal. In this case, marked NULL 'values' or variables should be used.

The treatment of NULL 'values' is different in DBMS. Some of them treat NULL 'values' as missing or unknown values. Evaluation of expressions with NULL 'values' is different in DBMS. For this reason, it is a good idea to restructure all relations to relations without NULL 'values' whenever possible. Since this approach is an implementation approach we are not using it during conceptual modelling.

[30] proposed a specific treatment of missing information. NULL 'values' can be used based on the interpretations 'value does not exist', 'no information neither on existence nor on values' or 'value exists but is currently unknown'[3,18]. This concept can be introduced in HERM in a shortened form. Since NULL 'values' can also be used in keys [28], the identification property is not lost in general. We can use NOTEXIST for null-valued components in the first approach. We can use UNKNOWN for 'unknown' and NOINFORMATION for 'no information'. The 'value exists but is unknown' approach can be represented by a formula $\exists x P_R(..., x, ...)$. The 'no-information' approach is based on the projection of R to the defined components $compon(R) \setminus \{A\}$,

i.e. $P_{\pi(compon(R)\setminus\{A\})}(...)$. The non-existence of values on A can be represented by $\neg\exists x P_R(..., x, ...) \wedge P_{\pi(compon(R)\setminus\{A\})}(...)$. We are going to extend this approach in this paper.

Often the use of NULL 'values' is forbidden for the primary keys or for all keys. This restriction is an implementation restriction which is required by most commercial DBMSs. *Default values* or *initial values* can be used for specific values. The *entity integrity rule* forbids nulls in primary key columns. Primary keys cannot contain NULL (missing) data. The reason for this rule should be obvious for keys with a singleton attribute. An object cannot uniquely identified or referenced in a table if the primary key of that table can be NULL and there is more than one object with this property. It's important to note that this rule must not be applied to composite keys. The entity integrity rule requires however for composite keys that none of the individual columns can be null.

1.2. The SQL Standard and NULL 'Values'

The SQL:1999 standard defines NULL values in a similar way as the SQL2 standard. In detail, we find the following statements: "A null value (NULL) is a special value or mark that is used to indicate the absence of any data value. ... Values are either null values or non-null values. A null value is an implementation-dependent special value that is distinct from all non-null values of the associated data type. There is effectively only one null value and that value is a member of every SQL data type. There is no <literal> for a null value although the keyword NULL is used in some places to indicate that a null value is desired. ... Two sets are equal if they contain the same elements and those elements are all non-null values. The equality of two sets is unknown if the sets contain null values and the replacement of those null value with appropriate non-null values would make the two sets equal. Otherwise, two sets are unequal. ...

Every domain, column in a base table, SQL variable, and SQL-supplied parameter, has a null class. If no null class is specified, it is the general null class (which contains only the general null value); otherwise it is the defined null class that is specified.

A defined null class, is created by a <null class definition> and is a named set of possible null values known as null states, together with the general null value. A null state is a named, implementation-dependent null value that is distinct from both the general null value and all other null states of the same null class.

The null values of a null class are ordered on their position number, the general null value having position number one in every null class. This ordering is used to determine the result of an operation when more than one of its operands are null values.

Except in the few cases where the null substitution principle yields a different result, if one or more operands of an expression is a null value then the result is a null value.

The result of an attempted transfer of any null value between objects having different null classes is the general null value.

The null class of the result of an operation is determined as follows.

Case:

- If no operand is null, then the result is as determined by the application of other General Rules.
- If the operator is AND and either operand is false, then the result is false.
- If the operator is OR and either operand is true, then the result is true.
- If the result has a defined null class, and one or more operands are null, then:

  Case:

  * If the operator is OR then the result is the null value having the maximum position number;
  * Otherwise, the result is the null value having the minimum position number.

  Note: The general null value effectively has the position number one.

- Otherwise, the result is the general null value.

... Every column an stored attribute has a nullability characteristic that indicates whether any attempt to store a null value into that column or stored attribute will inevitably cause an exception to be raised, and whether any attempt to retrieve a value from that column or stored attribute can ever result in a null value. The possible values of the nullability characteristic are known not nullable and possibly nullable. ... ”

SQL follows the semantics of predicate logic also in the case of missing values and thus cause a number of problems: The interpretation of nulls ‘values’ is not easy to adapt to a particular intuition; the meaning of relational operations when applied to both null values and certain data values is not well-defined; and the treatment of integrity constraints in databases containing null values is not completely defined.

1.3. 14 (or 20) Kinds of NULL ‘Values’ in Databases

The ANSI/Sparc report [1] has been introducing 14 different kinds of incomplete data that could appear as a result of queries or as attribute values. This collection is not complete. We may distinguish the following kinds of NULL ‘values’:

1. The property is not applicable for this object but belongs to this class of objects.
   - 1.1. Independently from the point of time t. *“not applicable”*
   - 1.2. At the current point of time t. *“currently not applicable”*
2. The property does not belong to the object.
   - 2.1. The property is not representable in the schema.
     - 2.1.1. Due to changes of value type (temporarily, fuzzy, ...). *“many-typed”*
   - 2.2. The property is representable in the schema.
     - 2.2.1. But there is no value for the object. *“unknown”*
       - 2.2.1.1. Because it has not been transferred from another database.
       - 2.2.1.2. Because is has not yet inserted into the database. *“existential null”*
     - 2.2.2. The value for the property exists but is *“under change”*.
       - 2.2.2.1. However the value is trackable.
         - 2.2.2.1.1. But is at the moment forbidden.
         - 2.2.2.1.2. At the moment permitted.
           - 2.2.2.1.2.1. But not defined for the database.
             - 2.2.2.1.2.1.1. Because it is currently under change.
           - 2.2.2.1.2.2. The value is defined for the system.
             - 2.2.2.1.2.2.1. But is currently incorrect.
             - 2.2.2.1.2.2. But is currently doubtful.
       - 2.2.2.2. The value is not trackable.
         - 2.2.2.2.1. Because of changes.
         - 2.2.2.2.2. Because of reachability. *“place-holder null”*
     - 2.2.3. There are several values for the property of this object. *“partial null”*

     (2.2.3.1., 2.2.3.2.1, 2.2.3.2.2. similarly to 2.2.2.) *“nondeterministic”*
     - 2.2.4. There is no value for the property of this object. *“not exists”*
     - 2.2.5. There is never a value for the property of this object. *“never exists”*
3. The property is may-be applicable for this object but it unknown whether it is true for the object in this case. *“may-be null”*
   - 3.1. It is not known whether the property is applicable to the given object. If it is applicable then its value for this property is taken from certain domain. *“partial may-be null”*

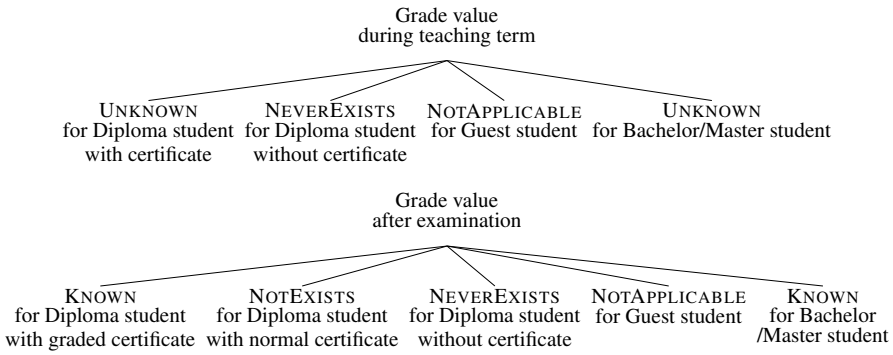
The nondeterministic and the many-typed cases are caused by wrong modelling of the database. If we abstract from temporarily not available values then we need to distinguish at least the NULL 'values' into UNKNOWN, NOT APPLICABLE, NOT EXISTS, and NEVER EXISTS.

Finally, we should separate NULLs from other real values. Therefore we call these values either NULL 'values' or better NULL **markers**.

1.4. An Application Example

It has often been claimed that specific markers are not necessary for relational databases. We need however to consider also *schema parsimony* as a schema quality criterion. Large schemata are a burden for users and programmers. They increase the possibility of inherent schema errors. Therefore, practicians tend to denormalise schemata for performance reasons and to combine types of common behaviour and shared structures into singleton types. As a surprise for theoreticians, the schema becomes easier to maintain and the theory of integrity constraints is not much more complex. We call this phenomenon the *efficiency unapprehensiveness*.

Let us consider a simple example of a relational type *Enrolls* associating the *Student* and *Course* within a *Program*. A student might obtain a grading if this is applicable after the examination. The grading is however different for each type of student. A natural separation of concern would be the following one.



1.5. Research on NULL Values

Database research that tackles the null value problem is fairly rich, e.g. [2,5,10,11,12, 19,21,23,25,26,17,27,31,33]. Research on null values is not yet complete. Most papers cover two or three kinds of null markers. The most comprehensive survey on some kinds of null values (existential, may-be, place-holder, partial, partial may-be) is given by [6]. The situation is different for efficient query computation with null-valued object. [20] discusses in detail whether sure and potential answers can be generated from databases with different kinds of nulls (unknown, no information). Comprehensive data-complexity analysis for querying databases with null values has led to a deep understanding of the reasons for complexity increase. There is no extensive research on database modelling techniques for null values. [14,15] provides an insight into treatment of different nulls through variables and equational logics. [28] shows that the entity integrity rule can also be weakened in the case of primary keys.

[7,9] claim that NULLs are a disaster and explains how we can use SQL without any utilisation of NULL values. The proposal is based on horizontal and vertical decomposition. Each relational database structure R is decomposed for each nullable attribute A into a relational structure that contains all objects with no-NULL values and into another one $R.WithoutA$ without this attribute for objects that have a NULL value. This approach directly leads to a combinatorial explosion of database schemata since relationship types are also decomposed as a consequence.

It is surprising to see that in almost all approaches to solve the null value problem by many-valued logics the truth values form a lattice with **F** being minimal with respect to a partial order on truth values – an exception is the recent paper [13] . In this paper we propose a logic that combines a para-consistent logic with a null value smaller than **F** and fold it into a four-valued logic. Thus, besides **U** (unknown) we consider values **NE** (does not exist) and **NA** (not applicable) as completely legal truth values. In the logic all six binary operators of Peirce's triadic logic are present.

Using **NE** states that it does not make sense to consider the conjunction with other truth values, which implies $\mathbf{NE} \wedge \mathbf{F} = \mathbf{NE}$ and $\mathbf{NE} \vee \mathbf{F} = \mathbf{F}$. Therefore, it makes sense to apply Kleene's conjunction alteration [16] to this null value. For **NA** it makes sense to adopt Turquette's definitions for conjunction and disjunction, which means that it is ignored in conjunctions and disjunctions.

1.6. Brief Survey on this Paper

We are going to develop another logical and algebraic foundation for different kinds of null markers and show that these markers can be implemented in SQL:1999. This paper discovers that classical 3-valued logics does not provide a good foundation. Instead we introduce a specific para-consistent logics and fold it into a many-valued logics. Since we do not target on a full-fledged treatment of all twenty different kinds of null markers we restrict the consideration to four main types.

2. NULL 'Value' Logics

2.1. NULL Types

A base type is an algebraic structure $B = (Dom(B), Op(B), Pred(B))$ with a name, a set of values in a domain, a set of operations and a set of predicates. A class B^C on the base type is a collection of elements from $Dom(B)$. Usually, B^C is required to be set. It can be also a list, multi-set, tree etc. Classes may be changed by applying operations. Elements of a class may be classified by the predicates.

The value set can be discrete or continuous, finite or infinite. We typically assume discrete value sets. Typical predicates are the comparison predicates $<, >, \leq, \neq, \geq, =$. Typical functions are arithmetic functions such as $+, -$, and $\times$.

The NULL type is a nominal type, i.e. a specific kind of intension based data types beyond the extension based types such as absolute types or ratio types [30]. It can also be an ordinal type.We use the following truth values:

U = unknown: This results, when a value for a property exists, but is not known.

NA = not applicable: This results, when a property is not applicable.

NE = not exists: This results, when a value for a property does not exist.

The NULL type is an enumeration type with $Dom(NULL) = \{ \mathbf{U}, \mathbf{NA}, \mathbf{NE} \}$[3], with two predicates $Pred(NULL)\{=_{NULL}, \neq_{NULL}\}$ and an empty set of operations. We require that $Dom(NULL)$ is disjoint with any other type we are using for modelling.

2.2. NULL Logics

In the following we will present an integrated propositional logic combining the three-valued logics of Łukasiewicz and Turquette with para-consistent logic based on Kleene [16]. The syntax is standard with conjunction $\wedge$, disjunction $\vee$ and negation $\neg$, but we will also add a weak negation [32]. For the semantics will use truth tables. On these grounds we can then define different types of implication. We first show that negation $\neg$ and Łukasiewicz implication are sufficient to express all constructs, then we extend Wajsbergs axiomatisation for $\mathcal{L}_3$ to our logic.

We base interpretations of propositional formulae on five-valued interpretations using truth values **NE**, **F**, **NA**, **U**, **T**, i.e. we assume a total order on the truth values.

The most commonly known approaches to handle null values concentrate on three-valued Łukasiewicz logic $\mathcal{L}_3$, in which $\wedge$ corresponds to building the minimum, while $\vee$ requires taking the maximum. In particular, this applies to **U**, in which case we obtain the following Łukasiewicz truth tables with the classical Peirce negation $\neg$. In addition, we define weak negation $\sim$. Note that the truth table definitions for conjunction and disjunction correspond to the operators Z and Θ in Peirce's triadic logic.

$\wedge$	**T**	**U**	**F**
T	**T**	**U**	**F**
U	**U**	**U**	**F**
F	**F**	**F**	**F**

$\vee$	**T**	**U**	**F**
T	**T**	**T**	**T**
U	**T**	**U**	**U**
F	**T**	**U**	**F**

$\neg$	
T	**F**
U	**U**
F	**T**

$\sim$	
T	**F**
U	**T**
F	**T**

For **NA** it makes more sense to adopt Turquette's conjunction and disjunction, which correspond to the operators Ψ and Φ in Peirce's triadic logic:

$\wedge$	**T**	**NA**	**F**
T	**T**	**T**	**F**
NA	**T**	**NA**	**F**
F	**F**	**F**	**F**

$\vee$	**T**	**NA**	**F**
T	**T**	**T**	**T**
NA	**T**	**NA**	**F**
F	**T**	**F**	**F**

$\neg$	
T	**F**
NA	**NA**
F	**T**

$\sim$	
T	**F**
NA	**T**
F	**T**

The main difference is that both in conjunction and disjunction the result is **NA** only, if both factors are **NA**. This results from the simple consideration that if a property is not applicable, then it can be omitted in a conjunction or disjunction. This argument can be used to combine this two three-valued logics, i.e. $\mathbf{U} \wedge \mathbf{NA} = \mathbf{U}$, and $\mathbf{U} \vee \mathbf{NA} = \mathbf{U}$ hold.

For the truth value **NE**, however, it makes sense to adopt Kleene's weak conjunction and alteration [16] – these correspond to the operators Ω and Y in Peirce's triadic logic – which define the following para-consistent logic:

[3] We restrict the treatment of null markers to these four values. The value NEVEREXISTS and other NULL markers can be treated in a similar way.

$\wedge$	**T**	**NE**	**F**
T	**T**	**NE**	**F**
NE	**NE**	**NE**	**NE**
F	**F**	**NE**	**F**

$\vee$	**T**	**NE**	**F**
T	**T**	**NE**	**T**
NE	**NE**	**NE**	**NE**
F	**T**	**NE**	**F**

$\neg$	
T	**F**
NE	**NE**
F	**T**

$\sim$	
T	**F**
NE	**NE**
F	**T**

If **NE** is involved, this reflects the consideration that if a value for a property does not exist, a conjunction or disjunction does not make sense. We can integrate this paraconsistent logic with the other two three-valued logics. Using the arguments above for disjunction we only have to add $\mathbf{NE} \vee \mathbf{L} = \mathbf{NE}$ for all **L**. Thus, we obtain the following truth tables, which complement the truth tables above:

$\wedge$	**NE**	**NA**	**U**
NE	**NE**	**NE**	**NE**
NA	**NE**	**NA**	**U**
U	**NE**	**U**	**U**

$\vee$	**NE**	**NA**	**U**
NE	**NE**	**NE**	**NE**
NA	**NE**	**NA**	**U**
U	**NE**	**U**	**U**

Let $\mathcal{L}_5$ denote the propositional logic with junctors $\wedge, \vee, \neg, \sim$, and the five-valued semantics defined by the truth tables above. Using these tables it is easy to verify the following result.

Proposition 1 *The following equivalences hold in the five-valued propositional logic* $\mathcal{L}_5$*:*

$$\neg\neg\alpha = \alpha \qquad \neg(\alpha \wedge \beta) = \neg\alpha \vee \neg\beta \qquad \neg(\alpha \vee \beta) = \neg\alpha \wedge \neg\beta$$

In particular, conjunction can be expressed by disjunction and strong negation, which implies the following result.

Proposition 2 *The system* $\{\vee, \neg, \sim\}$ *is complete for* $\mathcal{L}_5$.

2.3. Folding NULL Types with Other Types

Any ordinary domain type $B = (Dom(B), Op(B), Pred(B))$ can be used to extend to a new type
$B_{WithNULL} = (Dom(B) \cup Dom(NULL), Op(B), Pred(B) \Cup Pred(NULL))$
with partial operations that are defined for $Dom(B)$ and undefined for arguments from $Dom(NULL)$ and with combined predicates $=, \neq$ that inherit these predicates from the component types if they exist there and stating that values from $Dom(B)$ and $Dom(NULL)$ are different in the specific form explained above.

3. NULL 'Values' and Integrity Constraint Logics

3.1. Dependencies with Unknown NULLs

Dependencies can be generalized to relations containing null 'values'. Two tuples t and t' are strongly equivalent with respect to X (denoted by $t \approx_X t'$) if both are defined on X and are equal on X. They are weakly equivalent on X (denoted by $t \sim_X t'$) if they are both equal on A whenever both are defined on A for any $A \in X$, i.e. if they are

both equal on A whenever both are defined on A or both are undefined for any $A \in X$. Now we can define different kinds of validity for the functional dependency $X \rightarrow Y$ in a relation R^C with null 'values'. Some of them are as follows:

- The relation R^C 1-satisfies the functional dependency $X \rightarrow Y$ if all pairs of strongly X-equivalent tuples are strongly Y equivalent.
- The relation R^C 2-satisfies the functional dependency $X \rightarrow Y$ if all pairs of weakly X-equivalent tuples are weakly Y equivalent.
- The relation R^C 3-satisfies the functional dependency $X \rightarrow Y$ if all pairs of strongly X-equivalent tuples are weakly Y equivalent.
- The relation R^C 4-satisfies the functional dependency $X \rightarrow Y$ if all pairs of weakly X-equivalent tuples are strongly Y equivalent.

We conclude with [30]:

(1) 2-satisfiability implies 3-satisfiability.
(2) 1-satisfiability implies 3-satisfiability.
(3) 4-satisfiability implies 1-satisfiability and 2-satisfiability.

The classical Armstrong axiomatisation [30] for functional dependencies can be directly applied to the axiomatisation of 1- and 2-satisfiability. The augmentation axiom $X \cup Y \rightarrow Y$ is not valid for 4-satisfiability. The transitivity rule does not apply to 3-satisfiability, i.e., the 3-satisfiability of $X \rightarrow Y$ and $Y \rightarrow Z$ in a relation R^C does not imply the 3-satisfiability of $X \rightarrow Z$.

A key K is called a *sure key* of R^C if R^C 4-satisfies $K \rightarrow attr(R)$. The key is called a *possible key* of R^C if R^C 3-satisfies $K \rightarrow attr(R)$.

In the same manner multivalued, join and other dependencies can be generalized for relations with null 'values'.

3.2. Dependencies with Inapplicable NULLs

There are several kinds of null values which should be distinguished in this case, depending on whether a property is applicable to an object, whether a property is under change (incomplete, not committed), whether a value is available, whether a value is stored, whether a value is derivable from inconsistent or incomplete data and whether a value is secured. Context-dependent null values [28,29] are semantically defined null values.

3.3. Possible Worlds Semantics for Unknown and Inapplicable NULLs

Another approach to null values is based on possible world semantics [2,25,24]. A tuple t without null values is a completion of a tuple t' which uses null values if the tuples t, t' are weakly equivalent. A relation $R^{C'}$ is a completion of a relation R^C with null values if it is obtained by substitution of null values by non-null values from the corresponding domains. A functional dependency is *weakly satisfied* in R^C if it is satisfied in one of the completions of R^C. We observe that a functional dependency can be weakly satisfied in R^C but is not i-satisfied for $i \in \{1, 2, 3, 4\}$. Weak satisfaction leads to the additivity problem [24], i.e. R^C weakly satisfies a functional dependency α and weakly satisfies a functional dependency β but does not weakly satisfy $\{\alpha, \beta\}$.

3.4. Towards a Logic for Integrity Constraints

Integrity constraints are typically specified using the B(erri)V(ardi) frame $\mathcal{Q}(\alpha \rightarrow \beta)$ where α and β are quantor-free formulas with a set of variables X and $\mathcal{Q}$ is a sequence of quantifiers on X. We therefore need to introduce implication if we wish to use integrity constraints.

In classical logic implication $\alpha \rightarrow \beta$ is only a shortcut for $\neg\alpha \vee \beta$. In our logical treatment of null values, however, we introduced two different kinds of negation: strong Peirce negation $\neg$, and weak negation $\sim$. Accordingly, we have to permit at least two kinds of implication to capture integrity constraints: (strong) logical implication $\alpha \rightarrow \beta = \neg\alpha \vee \beta$, and (weak) material implication $\alpha \supset \beta = \sim \alpha \vee \beta$. Then we also obtain two different kinds of equivalence: $\alpha \leftrightarrow \beta = (\alpha \rightarrow \beta) \wedge (\beta \rightarrow \alpha)$ (logical equivalence), and $\alpha \equiv \beta = (\alpha \supset \beta) \wedge (\beta \subset \alpha)$ (material equivalence). The following truth tables indicate these two forms of implication – they can be easily verified from the definition of $\vee$, $\neg$ and $\sim$ in the previous subsection:

$\supset$	**T**	**U**	**NA**	**F**	**NE**
T	**T**	**U**	**F**	**F**	**NE**
U	**T**	**T**	**T**	**T**	**NE**
NA	**T**	**T**	**T**	**T**	**NE**
F	**T**	**T**	**T**	**T**	**NE**
NE	**NE**	**NE**	**NE**	**NE**	**NE**

$\rightarrow$	**T**	**U**	**NA**	**F**	**NE**
T	**T**	**U**	**F**	**F**	**NE**
U	**T**	**U**	**U**	**U**	**NE**
NA	**T**	**U**	**NA**	**F**	**NE**
F	**T**	**T**	**T**	**T**	**NE**
NE	**NE**	**NE**	**NE**	**NE**	**NE**

For the three-valued Łukasiewicz logic $\mathcal{L}_3$, which we adopted for the truth value **U**, we have a third implication $\alpha \Rightarrow \beta$, which we call the *Łukasiewicz implication*. This is defined by the following truth table:

$\Rightarrow$	**T**	**U**	**F**
T	**T**	**U**	**F**
U	**T**	**T**	**U**
F	**T**	**T**	**T**

Then it is easy to see that disjunction $\alpha \vee \beta$ can be expressed as $(\alpha \Rightarrow \beta) \Rightarrow \beta$, and weak negation $\sim \alpha$ as $\alpha \Rightarrow \neg\alpha$. Thus, both logical and material implication can be expressed by strong negation $\neg$ and Łukasiewicz implication $\Rightarrow$. With the following truth table we extend the definition of the Łukasiewicz implication to the other null values.

$\Rightarrow$	**T**	**U**	**NA**	**F**	**NE**
T	**T**	**U**	**NA**	**F**	**NE**
U	**T**	**T**	**U**	**U**	**NE**
NA	**T**	**T**	**T**	**T**	**NE**
F	**T**	**T**	**F**	**T**	**NE**
NE	**NE**	**NE**	**NE**	**NE**	**NE**

So the logic $\mathcal{L}_5$ will have an additional implication operators $\rightarrow$, $\supset$ and $\Rightarrow$. At the same time we preserve the equivalences for disjunction and weak negation, which again can be easily verified by using the truth tables. This gives the following result.

Proposition 3 *In $\mathcal{L}_5$ the following equivalences hold:*

$$\alpha \vee \beta = (\alpha \Rightarrow \beta) \Rightarrow \beta \qquad \sim \alpha = \alpha \Rightarrow \neg\alpha$$

In particular, all operators of the logic $\mathcal{L}_5$ can be expressed by using only Łukasiewicz implication and Peirce negation.

Proposition 4 *The system $\{\neg, \Rightarrow\}$ is complete for $\mathcal{L}_5$.*

4. SQL Realisation

4.1. User-Defined Domain Types and Functions for NULL Markers

SQL:1999 supports user-defined domain types and user-defined functions. This support can be used for the development of extensions that can easily be integrated into any database application. We do thus not need an extension of SQL. Instead we use templates for generation of supplements to database schemata that can be integrated into any application. We therefore proceed in a way different from classical SQL handling.

The first template we use is the definition of a user-defined domain. For instance,

```
CREATE DOMAIN BOOLExt AS VARSTRING(14)
    [ DEFAULT value ]
    CHECK (VALUE = 'TRUE' OR VALUE = 'FALSE'
        OR VALUE IS 'UnknownNULL'
        OR VALUE IS 'NotExistNULL'
        OR VALUE IS 'NeverExistNULL'
        OR VALUE IS 'NotApplNULL');
GO;
```

This example is only one realisation. We might also use such domain type extensions for other kinds of null values.

The next extension is the explicit support for connectives. Since the theory presented above uses a conservative extension of conjunction, disjunction and negation, we may define functions that can be used instead of Boolean expressions in `WHERE`-clauses. These functions can directly be generated while parsing a query. Therefore, there is no need for SQL programmers to change their style of writing Boolean expressions.

```
CREATE FUNCTION [dbo.]OrExtend (@FirstBool BOOLExt , @SecondBool BOOLExt)
     RETURNS BOOLExt WITH ENCRYPTION AS
     BEGIN
      DECLARE @ResultBool BOOLExt

      ....
      RESULT @ResultBool
     END;
GO;
```

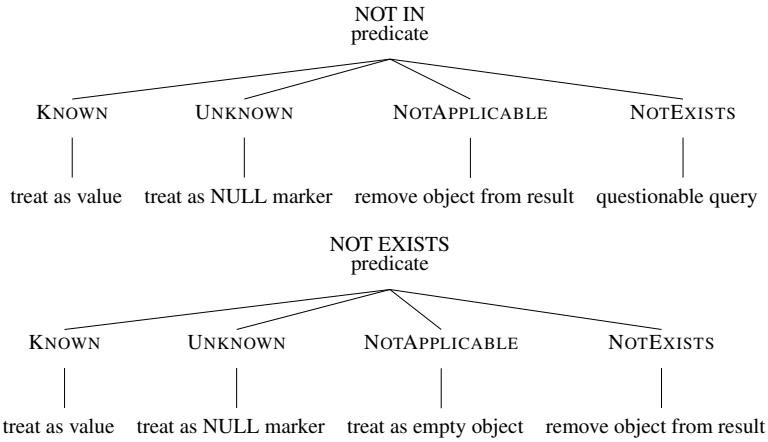
The function directly encodes the truth table for the connective $\vee$. In a similar way we may implement the truth table for negation and conjunction.

4.2. Treatment in Predicates

There are two special operands used to test for the presence of the NULL. `ISNULL` returns TRUE only when the supplied operand has a NULL. Conversely, `IS NOT NULL`

returns TRUE when the supplied operand does not have a NULL. These are quite important functions. One of the most common database mistakes is to test an operand for a NULL by comparing it to the empty string or zero. Instead we reprogram this function as well and use a number of user defined functions.

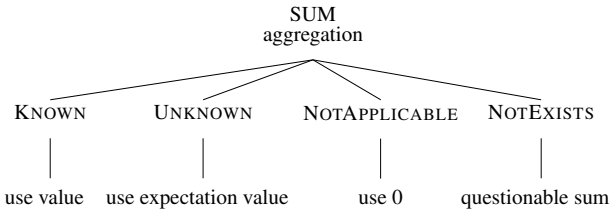
SQL allows also set predicates. These set predicates can consider in better detail the meaning of elements in a set if our proposal for handling NULL markers is used. In this case also observe that `NOT IN` and `NOT EXISTS` are becoming different and cannot be transferred to each other by simply applying double negation and collection. Compare the following two variants for these predicates.



4.3. Support for Aggregation

Most aggregate functions eliminate NULL in calculations; one exception is the `COUNT` function. When using the `COUNT` function against a column containing NULLs, the NULL will be eliminated from the calculation. However, if the `COUNT` function uses an asterisk, it will calculate all rows regardless of NULLs being present.

A formal and coherent framework to aggregation handling has been developed in [22]. Aggregation functions may be either distributive (`MAX`, `MIN`, `COUNT`, `SUM`) or algebraic (`AVG`) or holistic (`MEDIAN`, `RANK`). For instance, `SUM` is reprogrammed by using the following options:



Distributive aggregation functions are defined by structural recursion and can thus be directly derived from the logic discussed above. Distributive functions can also be corrected for correct computation. for instance, we might use the `ISNULL` function for replacing the NULL marked attributes with a valid value (`COUNT(ISNULL(Attribute,0))`).

Algebraic aggregation functions need a corrective. For instance, the `AVG` function must then separate objects for which a value is known from objects for which a value is

marked by a NULL marker. In this case we derive a split of the answer that corresponds to this separation. Instead we might also use completion strategies for algebraic functions. For instance, unknown but applicable and existing values may be completed to the expectation value if the distribution function is known. Holistic aggregation functions are far more difficult.

5. Concluding

Null 'values' are often considered to be the nightmare for programmers and database processing. We claim that they are an opportunity for better treatment of real world things. It is often natural in an application domain that null markers are attached to an object. This situation is similar to the reluctance of exceptions in programming. Since the application world is never perfect we need an explicit exception handling [4] that is implementable. Null markers are a means to handle exceptions for the occurrence of values in objects.

Null markers are a very powerful weapon for database modelling. NULL is not the number zero and NULL is not the empty string ('') value. They allow to combine object collections with a general common behaviour into a class. We do not require to separate objects of the same natural kinds into different classes depending on a missing value. If we consider, for instance, the *Enroll* type for *Student* then we might have use other null-marked attributes beside the attribute *Grade*. Examples in real applications are *Laboratory* (Students may participate in practical courses and thus improve their grading.), *Project* (Students may submit a project and thus be graded in a different way.), *Bonus* (Students may use a bonus system for final examinations and thus improve their chance for good grades in final examinations.), and *History* (Students may have already enrolled a similar course and thus might be given a credit for their history.). If we would model all these different kinds by subtypes then we might easily have to use 5×5 different types instead of the singleton type *Enroll* together with complex integrity constraints. The nightmare is then complete if *Enroll* is used in other relationship types.

It is surprising that logic users consider the logical value 0 as the minimal value. It is a value that has an infological value and a meaning, namely the truth value 'False' that is known to us. We consider the value **NE** as completely legal value since it states that it never makes sense to consider the truth value. It is thus valuable information for query processing.

References

[1] ANSI/X3/SPARC. Study group on data base management systems. Interim Report. *ACM SIGMOD Records*, 7(2), 1975.

[2] P. Atzeni and N. M. Morfuni. Functional dependencies in relations with null values. *Information Processing Letters*, 18(4):233–238, 1984.

[3] P. Atzeni and R. Torlone. A metamodel approach for the management of multiple models and the translation of schemes. *Information Systems*, 18(6):349–362, 1993.

[4] A. Berztiss and B. Thalheim. Exceptions in information systems. In *Digital Libaries: Advanced Methods and Technologies, RCDL 2007*, pages 284–295, 2007.

[5] J. Biskup and H. H. Brüggemann. Designing acyclic database schemes. In H. Gallaire, J. Minker, and J.-M. Nicolas, editors, *Advances in Database Theory, Vol. II*, pages 3–26. Plenum Press, New York, 1983.

[6] K. S. Candan, J. Grant, and V. S. Subrahmanian. A unified treatment of null values using constraints. *Inf. Sci.*, 98(1-4):99–156, 1997.

[7] H. Darwen and C. J. Date. The third manifesto. *SIGMOD Record*, 24(1):39–49, 1995.

[8] C. J. Date. *Logic and Databases - The roots of relational theory*. Trafford Publishing, 2007.

[9] C. J. Date and H. Darwen. *Database Explorations. Essays on The Third Manifesto and related topics*. Trafford Publishing, 2010.

[10] B. S. Goldstein. Formal properties of constraints on null values in relational databases. Technical Report 80-013, SUNY at Stony Brook, Dept. of Computer Science, 1981.

[11] J. Grant. Null values in a relational data base. *Inf. Process. Lett.*, 6(5):156–157, 1977.

[12] J. Grant. Null values in SQL. *SIGMOD Record*, 37(3):23–25, 2008.

[13] S. Hartmann and S. Link. When data dependencies over sql tables meet the logics of paradox and s-3. In *PODS*, pages 317–326. ACM, 2010.

[14] N. Cat Ho and H. Rasiowa. Subalgebras and homomorphisms of semi-Post algebras. *Studia Logica*, 46(2):161–175, 1987.

[15] N. Cat Ho and B. Thalheim. On semantic and syntactic issues of null values in the relational model of databases. Manuscript, Dresden, 1987.

[16] S. W. Jablonski, G. P. Gawrilow, and W. B. Kudrjavcev. *Boolesche Funktionen und Postsche Klassen*. Akademie-Verlag, Berlin, 1970.

[17] W. Lipski Jr. On semantic issues connected with incomplete information databases. *ACM Trans. Database Syst.*, 4(3):262–296, 1979.

[18] P. Kandzia and H.-J. Klein. *Theoretische Grundlagen relationaler Datenbanksysteme*. Bibliographisches Institut, Darmstadt, 1993.

[19] A. M. Keller. Set-theoretic problems of null completion in relational databases. *Information Processing Letters*, 22(5):261–265, 1986.

[20] H.-J Klein. *Gesicherte und mögliche Antworten auf Anfragen an relationale Datenbanken mit partiellen Relationen*. Habilitationsschrift, CAU Kiel, 1997.

[21] A. L. Kulenovic and A. Kulenovic. Treatment of null values in the nfset data model. In *BNCOD*, pages 226–244, 1991.

[22] H.-J. Lenz and B. Thalheim. A formal framework of aggregation for the OLAP-OLTP model. *Journal of Universal Computer Science*, 15(1):273 – 303, 2009.

[23] M. Levene and G. Loizou. Inferring null join dependencies in relational databases. *BIT*, 32:413–429, 1992.

[24] M. Levene and G. Loizou. The additivity problem for data dependencies in incomplete relational databases. In L. Libkin and B. Thalheim, editors, *Proc. Semantics in Databases*, LNCS 1358, pages 136–169. Springer, Berlin, 1998.

[25] Y. E. Lien. Multivalued dependencies with null values in relational databases. In A. L. Furtado and H. L. Morgan, editors, *Proc. 5th Int. Conf. on Very Large Data Bases - VLDB'79*, pages 61–66, Rio de Janeiro, 1979, 1979. IEEE-CS.

[26] S. Link. On the implication of multivalued dependencies in partial database relations. *Int. J. Found. Comput. Sci.*, 19(3):691–715, 2008.

[27] M. A. Roth, H. F. Korth, and A. Silberschatz. Null values in nested relational databases. *Acta Inf.*, 26(7):615–642, 1989.

[28] B. Thalheim. On semantic issues connected with keys in relational databases permitting null values. *Journal of Information Processing and Cybernetics*, EIK, 25(1/2):11–20, 1989.

[29] B. Thalheim. *Dependencies in relational databases*. Teubner, Leipzig, 1991.

[30] B. Thalheim. *Entity-relationship modeling – Foundations of database technology*. Springer, Berlin, 2000.

[31] Y. Vassiliou. Null values in data base management: A denotational semantics approach. In *SIGMOD Conference*, pages 162–169. ACM, 1979.

[32] G. Wagner. A database needs two kinds of negation. In *MFDBS*, volume 495 of *Lecture Notes in Computer Science*, pages 357–371. Springer, 1991.

[33] C. Zaniolo. Database relations with null values. *J. Comput. Syst. Sci.*, 28(1):142–166, 1984.