How to Modify SQL Queries in Order to Guarantee Sure Answers

H.-J. Klein
Department of Computer Science
University of Kiel, 24118 Kiel, Germany
hjk@informatik.uni-kiel.d400.de

Abstract

Some problems connected with the handling of null values in SQL are discussed. A definition of sure answers to SQL queries is proposed which takes care of the "no information" meaning of null values in SQL. An algorithm is presented for modifying SQL queries such that answers are not changed for databases without null values but sure answers are obtained for arbitrary databases with standard SQL semantics.

1 Introduction

The handling of null values in SQL ([SQL89], [SQL92]) may result in answers to queries which do not represent the type of information expected by users ([Da]). A serious problem is that query formulations which are equivalent for databases without null values may have different results in the presence of null values ([NPS]). Thus, if standard rules for the transformation of logical expressions into equivalent expressions are applied to SQL queries, answers may change.

These problems originate in the mixture of set based and logic based techniques used for the specification of SQL semantics: whereas three-valued logic is used to define the result of search conditions in WHERE clauses, a subquery in a predicate is evaluated to an SQL relation (table) before the definition of the predicate is applied. Since a tuple (row) qualifies for the result of a subquery only if the condition in the corresponding WHERE clause takes the value *true*, there is a switch from three-valued logic to Boolean logic when the value of a predicate including a subquery is determined. Together with negation (NOT) or quantification (ALL) this may result in a critical loss of information.

A null value (NULL) in SQL is by definition "a special value, or mark, that is used to indicate the absence of any data value" ([SQL92]). There is no further information connected with any occurrence of this special value. In applications, however, some knowledge about occurrences of NULL values in attributes often follows from schema semantics. As an example, consider a relation scheme named PERSON with BIRTHDAY among its attributes. If the data in a PERSON relation refer to natural persons, then each occurrence of NULL in the attribute BIRTHDAY can be interpreted as "value existent but unknown" (there are other possible reasons for the absence of a value in this attribute such as "value not allowed to be seen"; but this is another level of information which becomes clear if we extend the given characterization to "value existent in reality but unknown to the database").

Let the following relation be given:

PERSON

LAST_NAME	FIRST_NAME	BIRTHDAY
Smith	Richard	NULL
Smith	William	1/19

(LAST_NAME, FIRST_NAME) shall be a key of PERSON. Consider the query Q1:

"*Get the data of all persons such that there is no other person having the same last name and the same birthday*".

A straightforward formulation of this query in SQL is:

```
SELECT *
FROM  PERSON x
WHERE NOT EXISTS
    (SELECT *
     FROM PERSON y
     WHERE  x.LAST_NAME = y.LAST_NAME
       AND  x.FIRST_NAME <> y.FIRST_NAME
       AND  x.BIRTHDAY  = y.BIRTHDAY)
```

According to SQL semantics the answer to this formulation of Q1 consists of all tuples of the given PERSON relation. Under the interpretation "value existent but unknown" this answer does not represent a *sure* information in the sense that it states something which cannot be concluded (for sure) from the data in the database. In case the (existent but unknown) birthday of Richard Smith is the same as that of William Smith the answer is not appropriate. Therefore it should be considered an *uncertain* or *maybe* information to query Q1. It turns into a sure information if Q1 is modified a little bit: "*Get the data of all persons such that there is no other person having, for sure, the same last name and the same birthday*".

Let us look at another way to write Q1 in SQL:

```
SELECT *
FROM PERSON x
WHERE x.LAST NAME  <> ALL
   (SELECT  y.LAST_NAME
    FROM PERSON y
    WHERE x.BIRTHDAY  = y.BIRTHDAY
      AND x.FIRST_NAME  <> y.FIRST_NAME)
```

This formulation of Q1 produces the same answer as the first formulation. Things change if a transformation is applied to this formulation which preserves equivalence in standard as well as in three-valued logic resulting in the following formulation:

```
SELECT *
FROM PERSON x
WHERE x.BIRTHDAY <> ALL
   (SELECT y.BIRTHDAY
    FROM PERSON y
    WHERE x.LAST_NAME = y.LAST_NAME
      AND x.FIRST_NAME <> y.FIRST_NAME)
```

Because of the treatment of NULL in quantified comparison predicates, the SQL answer to this formulation of Q1 is an empty table. Here Q1 is interpreted as "*Get the data of all persons such that there is, for sure, no other person having the same last name and the same birthday*". Independent of the interpretation of null values it is unsatisfactory that different results can be obtained for these straightforward formulations of Q1.

Let us consider another interpretation of missing values: "value not existent (in reality)". Look at the following relation with scheme ORDER (ORDNO, DATE, VOLUME, WORKS_ON) where ORDNO shall be a key:

ORDER

ORDNO	DATE	VOLUME	WORKS_ON
1	11/3	10000	NULL
3	11/3	5500	E2
4	11/4	3000	NULL

A tuple in this relation gives information on a single order comprising its date of entry, its volume, and in case an employee is working on the order, the identification number of this employee. An occurrence of NULL in the attribute WORKS_ON shall represent the fact that no employee is working on the respective order. We have a situation different from the former example insofar as the information in each ORDER relation is assumed to be complete with respect to the domain modelled by it.

The following query Q2 is an analogue to the "birthday" query:

"*Get the data of each order such that there is at most one employee working on orders with the same date of entry as this order*".

Consider a formulation of Q2 corresponding to the first formulation of Q1:

```
SELECT *
FROM ORDER x
WHERE NOT EXISTS
       (SELECT *
        FROM ORDER y
        WHERE  x.DATE = y.DATE
          AND  x.ORDNO <> y.ORDNO
          AND  x.WORKS_ON = y.WORKS_ON)
```

In this case the SQL answer, the tuples of the given relation, represents a sure information with respect to the meaning assumed for the absence of a data value in the attribute WORKS_ON.

The examples show that the type of information an answer represents may depend upon the interpretation of null values in the database. Thus, SQL allows to interpret "no information data" in a specific way by chosing appropriate query formulations.

In general, sure answers can be characterized as answers representing information which is valid independent of the meaning null values may have due to their underlying interpretation. Since in SQL an occurrence of NULL means that a single value is absent and nothing more, sure answers must be valid for this "no information" interpretation, i.e. in particular for the "value unknown" as well as for the "value not existent" interpretation. For answers representing *maybe* or *uncertain* information, on the other hand, it is sufficient to require that they are valid for some possible meaning of absent data values. In this paper we concentrate on answers representing sure information because this type of answer seems to be what SQL semantics was intended to produce.

The "no information" interpretation of missing values has been investigated formally in [Za]. A definition of extended relational algebra operations is given there in a framework of the relational data model generalized for this type of interpretation. The evaluation of algebraic expressions corresponds to the evaluation of queries by SQL insofar as results of subexpressions are always sure answers in our sense. As the above examples demonstrate, this functionality principle causes problems in connection with the logical equivalence of query formulations. Therefore we do not base our definition of sure answers on this approach.

In order to always get the same answer for queries which are equivalent for databases with no values missing in tuples, SQL semantics has to be changed or a set of admissible SQL query formulations has to be specified appropriately. We consider the second possibility and give an algorithm for modifying SQL queries in such a way that the same answer is obtained for queries which are equivalent if there are no occurrences of NULL in the database. Answers to modified queries are guaranteed to represent sure information in the sense discussed above.

In the next section we first discuss some characteristics of the null value concept of SQL and then give a definition of sure answers to queries which is appropriate to this concept. The definition is non-descriptive since it is based upon the evaluation of queries by SQL. It does not take care of occurrences of null values in argument sets of function expressions and in operands of set operations. The reason for this restriction is that these language constructs are not directly involved in the logical equivalence problem and should therefore be discussed seperately. Based on sure and so-called potential answers to queries and subqueries we develop an algorithm for modifying SQL queries using the IS NULL predicate form. The application of this algorithm produces SQL queries with sure answers for arbitrary databases.

2 Answers in the Presence of Null Values

Queries against complete relational databases, i.e. databases without null values, are usually evaluated under the *closed world assumption* ([Re]): It is assumed that in a database only positive knowledge is represented and that this knowledge is complete, i.e. all facts not implied by the database are taken to be false.

By the syntactical structure of queries, commercial query languages such as QUEL or SQL ensure that query evaluation can be done in finite time and that answers to queries are finite for finite databases. Answers consist of sets or multisets (bags) of tuples with values from specified domains. The interpretation of answers to queries is intuitively evident with the closed world assumption in mind.

The implicit representation of negative facts (tuples) under the closed world assumption supposes all tuples in a database to be totally defined. For such complete databases there is a one-to-one correspondence between facts of the domain which is modelled and tuples in the database. A tuple with values not defined for some attributes, however, represents a set of possible facts under the "no information" interpretation. Therefore, incomplete databases, i.e. databases with partially defined tuples, are not consistent with the closed world assumption under this interpretation. The nonunique representation of facts makes it necessary to distinguish two types of knowledge: knowledge which can be concluded for *sure* from the data in the database and knowledge which is *uncertain* insofar as it neither can be concluded nor ruled out for sure. Independent of how query evaluation is performed for incomplete databases, two questions arise in connection with answers:

- In which form should the result of query evaluation be represented?

- What kind of knowledge does an answer represent?

Since we are interested in the null value concept of SQL, we focus our attention on the representation of answers by sets or multisets of tuples. SQL allows null values to occur in tuples of answers. This is a necessary feature since otherwise it would not be possible to represent the content of a base relation as answer. The handling of missing values in SQL suggests that each occurrence of a null value represents the fact that a single value (and not a set of values) is missing. Furthermore, no distinction is made between a null value occurring in a base relation and a null value occurring in the result of an SQL query or subquery. In connection with the way duplicates are defined in SQL, this concept involves problems with the interpretation of answers as the following example demonstrates.

Example 1: Consider the relation R = {(a,2,NULL), (b,2,NULL)} with attributes A, B, and C (in this order). For the query "SELECT DISTINCT x.B, x.C FROM R x" we get {(2,NULL)} as result because in this context SQL takes two different occurrences of NULL as duplicates. Obviously, the interpretation of NULL in (2,NULL) should not be the same as in the tuples of R. It has to be taken into account that (2,NULL) represents an unknown number of tuples (2,NULL) where each occurrence of NULL may have a different meaning.

Another more general problem with the null value concept of SQL has to do with the expressive power of answers: for some queries it is not possible to represent the information which can be concluded for sure from the query and a given database.

Example 2: Let two relations R = {(a,2), (a,3), (a,4)} and S = {(a,NULL)} with attributes A and B be given. Consider the expression $R \setminus S$ of relational algebra. Since by assumption the occurrence of a null value in S represents the absence of a single value, we may conclude that the result of $R \setminus S$ is either {(a,2), (a,3)} or {(a,2), (a,4)} or {(a,3), (a,4)} or {(a,2), (a,3), (a,4)}. This knowledge cannot be represented by an SQL answer.

Remark: SQL defines the result of R EXCEPT S and R EXCEPT ALL S as {(a,2), (a,3), (a,4)}, an answer which cannot be concluded for sure because the tuples of R are among those represented by (a,NULL). Therefore R EXCEPT S has to be read as: all tuples of R such that there is no tuple in S having, for sure, the same components.

Let us now consider answers representing sure knowledge in more detail. The "no information" interpretation of null values suggests that a *sure* or *certain* answer to a query against some database contains only tuples qualifying for every meaning that occurrences of null values in the database may have. For our "birthday" example it follows that no tuple qualifies for the sure answer to any of the three formulations of query Q1 because in each case the search condition of the outer WHERE clause is evaluated to *false* if NULL in the first tuple of the relation is replaced by the possible meaning "1/19". Under the "no information" interpretation the sure answer to the formulation of query Q2 is {(4,11/4,3000,NULL)} : if we assume not to know anything about the WORKS_ON value in the first tuple of the ORDER relation, a possible meaning of this missing value is "E2". Thus, the first two tuples cannot be contained in a sure answer. The third tuple qualifies independent of what the absence of a value in the attribute WORKS_ON means.

Under the "unknown" interpretation each comparison predicate in an SQL query takes a Boolean value for every possible meaning of null values such that tautologies involving null values have to be taken into account ([Co], [Gr]) with consequences for the efficiency of query evaluation ([AKG]). Under the "no information" interpretation some of these tautologies disappear (e.g. $A \geq 0$ OR $A < 0$) because of the possible "not existent" meaning of null valued components if the following point of view is taken: it makes no sense to compare "nothing" with a data value ([Vas]). Some tautologies, however, remain tautologies as the following SQL query demonstrates: SELECT x FROM S x, S y WHERE x.A = y.A AND x.B = y.B. For relation S from Example 2 the SQL answer to this query is an empty table. The evaluation of the query does not take care of the fact that attribute values of a tuple are compared to themselves (notice that this query is not equivalent in SQL to S INTERSECT S).

Since we are interested in answers to SQL queries which can be produced by SQL and because of the well-known efficiency problems connected with the detection of tautologies, the following demand for sure answers under the "no information" interpretation seems to be appropriate: sure answers must not contain tuples whose qualification depends on the value of comparison predicates with null values among the arguments. A formal definition of this type of sure answers to SQL queries can be based upon the evaluation of queries specified by SQL.

Definition: A tuple t qualifies for the <u>sure answer</u> to an SQL query q if the qualification part of q takes the value *true* for every modified evaluation of t obtained from the SQL evaluation of t as follows: each occurrence of *unknown* determined as result of a predicate because NULL is among its arguments is replaced by one of the Boolean values *true* or *false*.

Remark: An empty sure answer may not be interpreted as "it is sure that no tuple qualifies". Its meaning rather is "no tuple surely qualifies".

If we adopt the definition of sure answers to a first order query language such as the relational calculus, three-valued logic can be used to determine sure answers. This follows from the monotonicity property of three-valued logic: no formula can change its value from *true* to *false* or vice-versa if some occurrence of *unknown* in the formula is replaced by *true* or *false*. Thus, there is an efficient method to compute sure answers for first order query languages. Because of the switch from three-valued logic to Boolean logic in SQL mentioned above, the monotonicity property is lost in general such that answers are not always sure answers in SQL. Therefore, other methods have to be developed if sure answers are to be computed efficiently.

The method presented in the next section is based on a simple idea for changing the evaluation of formulas under three-valued logic if only *true* results are of interest. Because of the monotonicity property of three-valued logic, it is sufficient in this case to guarantee that replacing *unknown* in a formula by one of the Boolean values does not change the value of the formula from *unknown* to *true*. Since it is possible to replace all occurrences of *unknown* in a formula in this way, Boolean logic can be used for the evaluation of the modified formula. Which value to take for replacement is determined uniquely by the syntactical structure of the formula. In Boolean expressions without negation each occurrence of *unknown* has to be replaced by *false*. Negation causes a switch for replacement from *false* to *true* (and vice-versa if there is a nesting of negations). In quantified formulas this switch is also caused by universal quantification.

The replacement of *unknown* by *true* means that what we call a potential answer is needed instead of a sure answer for the corresponding subexpression. For SQL this type of answer can be defined similar in form to sure answers as follows:

Definition: A tuple t qualifies for the <u>potential answer</u> to an SQL query q if the qualification part of q takes the value *true* under some modified evaluation of t obtained from the SQL evaluation of t as follows: each occurrence of *unknown* determined as result of a predicate because NULL is among its arguments is replaced by one of the Boolean values *true* or *false*.

Potential answers have to be distinguished from maybe (uncertain) answers mentioned above. In maybe answers tautologies involving null values have to be taken into account in order to hit what maybe (or uncertain) means. For potential answers such tautologies (as well as contradictions) play no role. We use this type of answer to simplify the description of our algorithm presented in the next section. Apart from this application, potential answers to queries may be interesting for themselves as alternative to maybe answers because they can be computed efficiently.

3 Modification of Queries

In the following we refer to SQL2 ([SQL92]) without function expressions and set operations.

The result of the application of a WHERE clause to a table T derived by the application of a proceeding FROM clause consists of those rows of T for which the search condition of the WHERE clause evaluates to *true*. Since three-valued logic is applied for the evaluation of search conditions, the resulting table always represents sure information in the sense discussed above provided that all information used for the evaluation is sure. Under the latter assumption the comparison or test specified by a predicate including a subquery is performed on a table which represents a sure answer to the subquery. If such a predicate is negated by NOT, the type of answer which is obtained on the next nesting level of the query may change from *sure* to *potential*.

Consider the first formulation of query Q1 above. If NOT in front of the EXIST predicate is removed, we get a query which has always sure answers. Since for all tuples of a given PERSON relation not contained in the sure answer the value of the EXIST predicate is *false*, the negation of the EXIST predicate results in a potential answer if there are tuples such that the condition of the subquery evaluates to *unknown*. In order to guarantee that all tuples in the answer qualify for sure, tuples evaluating to *unknown* for the search condition of the subquery have to be taken into account. This means that a potential answer to the subquery is needed instead of a sure answer (cf. the extension of algebra operations in [Bi] and the algebra for C-tables in [IL]).

Which type of answer to a subquery is needed in order to get a sure answer to a query with SQL semantics depends on the predicate in which the subquery occurs and on the nesting structure of the whole query, especially its "negation structure". The form of answer to a subquery (sure or potential) determines the variants of all predicates in its WHERE clause. Negations cause a switch from sure to potential and vice versa.

For each predicate form of SQL we have to specify how sure and potential variants of predicates can be obtained without change of SQL semantics. As we shall see this can be achieved by use of the predicate form IS NULL which allows to check whether an attribute value is a null value or not. We discuss the method for the predicate form <quantified comparison predicate>. From the discussion follows how the other predicate forms have to be treated.

The general form of <quantified comparison predicate> is

<row value constructor> <compop> <quantifier> <table subquery>.

For simplicity we assume that <row value constructor> is a simple attribute, say A. Two forms have to be discussed: A θ SOME(S) and A θ ALL(S) where S stands for a suitable subquery and $\theta \in \{<, <=, =, <>, >=, >\}$.

1) A θ SOME(S).

a) Let the sure variant be required.

Then we need a sure result for S. Two critical cases have to be considered:

i) A $\equiv$ NULL.

NULL θ SOME(S) evaluates to *false* if the result of the evaluation of S is an empty table and to *unknown* otherwise. Instead of *unknown* the value *false* is needed for the computation of a sure result in order to take possible negations into account. This can be achieved by the addition of AND A IS NOT NULL to the predicate.

ii) The answer to S contains NULL.

Example: 5 >= SOME {7,8,NULL} . According to SQL semantics this predicate evaluates to *unknown*. In order to always get either *true* or *false* it is sufficient to exclude NULL from the answer to the subquery. This can be achieved by transforming S into a subquery S' as follows: let S be SELECT A' FROM .... WHERE Φ; S' results from S by adding AND A' IS NOT NULL to Φ.

b) The potential variant is needed.

In this case the potential answer has to be determined for S. We have to consider the same critical cases as above.

i) A $\equiv$ NULL.

Instead of the value *unknown* the value *true* has to be obtained if the result of S is not empty, since all tuples have to be considered which may qualify for appropriate replacements of NULL. This can be achieved by adding OR A IS NULL AND EXISTS(S) to the predicate. For S in EXISTS(S) the potential answer has to be determined as well.

ii) The answer to S contains NULL.

Consider the example 5 >= SOME {7,8,NULL} again. The comparison 5 >= NULL evaluates to *true* if NULL is replaced by an appropriate data value. Such a value always exists with the exception of comparisons such as minimum of domain > NULL which must be treated specially. Aside from such exceptions, *true* is always a possible value of the predicate if NULL occurs in the result of S. In order to take this into account, the predicate is extended by OR EXISTS(S') where S' is obtained from S, S as above, by adding AND A' IS NULL to Φ. For S' the potential answer has to determined.

2) A θ ALL(S).

a) Let the sure variant be needed.

In this case the potential variant has to be determined for S. Again, two critical cases have to be considered.

i) A ≡ NULL.

According to SQL semantics, NULL θ ALL(S) takes the value *true* if the result of S is an empty relation and the value *unknown*, otherwise. Since the sure variant is needed, the value *false* must be obtained instead of *unknown*. This can be achieved by the addition of AND (A IS NOT NULL OR NOT EXISTS(S)). For S in NOT EXISTS(S) the potential answer has to be computed.

ii) The answer to S contains NULL.

Example: 5 <= ALL {7,8,NULL}. In this case the result must be *false* instead of *unknown*. This outcome can be achieved by the addition of AND NOT EXISTS(S') to the predicate where S' is obtained from S by adding AND A' IS NULL to Φ. For S' the potential answers has to be determined.

b) The potential variant is needed.

Then, the sure answer has to be determined for S. The same critical cases have to be considered as above.

i) A ≡ NULL.

Instead of *unknown* the value *true* is needed. This can be achieved by the addition of OR A IS NULL. AND EXISTS(S) as in case 1)b)i) is not necessary since v θ ALL(S) takes the value *true* for every value v if the answer to S is an empty table.

ii) The answer to S contains NULL. In this case the same proceeding as in case 1)a)ii) is possible.

The application of the given rules to a quantified comparison predicate guarantees that the value *unknown* is no more obtained as result of the evaluation of this predicate in any database. For all other predicate forms of SQL2 similar rules can be specified.

In order to obtain an SQL query with a sure answer for each database, the predicates of a given query have to be modified by application of the corresponding rules. Whether the sure or the potential variant of a predicate has to be produced is determined as follows:

For each predicate of the WHERE clause of a subquery with a sure (potential) answer needed, the sure (potential) variant has to be produced if the predicate is not contained in a negated subexpression. If so, each negation causes a switch from sure to potential and vice versa for all predicates in the subexpression. The type of answer needed for a subquery of a predicate is determined by the variant of the predicate itself and the respective rules. For all predicates of the outer WHERE clause of a query not being contained in a negated subexpression the sure variant has to be produced.

For the second formulation of query Q1 we get the following result:

```
SELECT *
FROM PERSON x
WHERE x.LAST_NAME <> ALL
   (SELECT  y.LAST_NAME
    FROM PERSON y
    WHERE (x.BIRTHDAY = y.BIRTHDAY
        OR x.BIRTHDAY IS NULL
        OR y.BIRTHDAY IS NULL)
      AND x.FIRST_NAME <> y.FIRST_NAME)
```

There is an obvious way to improve the algorithm resulting in much simpler formulations for most queries: each predicate can remain unchanged if its sure variant is needed and if it is not directly contained in a negated subexpression of a WHERE clause. In such cases *unknown* can be replaced by *false* and vice versa without changing the respective sure answer.

Potential answers can be obtained if the potential variant is produced for all predicates of the outer WHERE clause of a query. It shall be stated again that this type of answer is different from what is called a maybe answer in the literature.

4 Final Remarks

The definition of sure answers to SQL queries given in this paper takes into account the "no information" interpretation of absent attribute values as specified in the SQL International Standard. It restricts answers to those tuples whose qualification is not dependent upon absent values in tuples of the database the query is issued against. The method proposed for the modification of SQL queries is correct and complete with respect to this type of answer. The logical equivalence problem discussed in Section 1 disappears if all queries are modified before evaluation.

If the absence of attribute values is interpreted as "value existent but at present unknown", then our definition of sure answers has to be changed. A less formal definition could be: a sure answer contains all tuples which qualify independent of the replacement of the occurrences of null values in the database by defined values. Formal definitions of sure answers for the "unknown" interpretation can be found in [Li] and [IL]; some problems with these definitions and several more sophisticated variants of sure answers are discussed in [Kl]. It is a co-NP-complete problem to compute sure answers to first order queries under the "unknown" interpretation ([Var], [AKG]). Our efficient method to get sure answers to SQL queries under the "no information" interpretation is correct but not complete with respect to the "unknown" interpretation. It is not complete since tautologies involving null values are not taken into account.

We have excluded function expressions and set operations from our considerations. For function expressions the SQL way to raise a special warning indicating that

NULL is among the arguments seems to be a good choice. Expressions with set operations cannot be modified along the lines of our method in order to always produce sure answers. This follows from results in [Bi] and [IL] showing that operations of relational algebra as well as relations themselves have to be extended in order to guarantee sure information in answers under the "unknown" interpretation. The extension proposed in [Za], on the other hand, does not always produce sure answers in our sense.

There is a special problem in SQL with respect to comparison predicates where one or both arguments are given by subqueries. Such predicates evaluate to *unknown* if the answer to one or both subqueries is empty. This means that *unknown* can be obtained during query evaluation even if there is no occurrence of NULL in the database. In these cases *unknown* should therefore be read as undefined.

References

[AKG] S. Abiteboul, P. Kanellakis, and G. Grahne: On the representation and querying of sets of possible worlds. Theor. Comp. Science 78, pp. 159-187, 1991

[Bi] J. Biskup: A foundation of Codd's relational maybe-operations. ACM Trans. on Database Systems 8 (4), pp. 608-636, 1983

[Co] E.F. Codd: Extending the database relational model to capture more meaning. ACM Trans. on Database Systems 4 (4), pp. 397-434, 1979

[Da] C. J. Date: Relational Database Writings 1985-1989. Addison-Wesley, 1990

[Gr] J. Grant: Null values in a relational data base. Inform. Process. Lett. 6 (5), pp. 156-157, 1977

[IL] T. Imielinski und W. Lipski: Incomplete information in relational databases. Journal of the ACM 31 (4), pp. 761-791, 1984

[Kl] H.-J. Klein: Sure answers to queries for databases with incomplete information. Technical Report 9310, Inst. f. Informatik u. Prakt. Mathem., Univ. Kiel, 1993

[Li] W. Lipski: On semantic issues connected with incomplete information databases. ACM Trans. on Database Systems 4 (3), pp. 262-296, 1979

[NPS] M. Negri, S. Pelagatti und L. Sbattella: Formal semantics of SQL queries. ACM Trans. on Database Systems 16 (3), pp. 513-534, 1991

[Re] R. Reiter: On closed world databases. In Logic and Databases (H. Gallaire and J. Minker, eds.), pp. 55-76, 1978

[SQL89] International Standard ISO/IEC 9075, second edition. Genève, 1989

[SQL92] International Standard ISO/IEC 9075, third edition. Genève, 1992

[Var] M.Y. Vardi: Querying logical databases. Journal of Comp. and System Sciences 33, pp. 142-160, 1986

[Vas] Y. Vassiliou: Null values in data base management: a denotational approach. Proc. of the ACM SIGMOD Int. Conf. of Data, Boston, pp. 162-169, 1979

[Za] C. Zaniolo: Database relations with null values. J. of Comp. and System Sciences 28, pp. 142-166, 1984