Implementing Marked Nulls in PostgreSQL

Peter Storeng

Master of Science

Computer Science

School of Informatics

University of Edinburgh

2016

Abstract

SQL's use of nulls has been widely criticised. This is for good reason, for simple queries databases with nulls are giving wrong answers. One solution to this problem is to avoid polluting databases with nulls altogether. Where possible nulls should of course be avoided, however incomplete data is very much a reality. The amount of data being captured and stored logically in databases is increasing rapidly and as a consequence so is the amount of incomplete data - it cannot be ignored.

The theoretical concept of *certain answers* provides correctness guarantees, however calculating them is not efficient or even useful. Recently query translations which approximate certain answers accurately on First Order queries, without producing wrong answers and, with good complexity bounds have been suggested. Further, they have been tested with encouraging results [14]. One issue with these translations is that they rely on a notion of Codd Nulls, where every null is distinct.

SQL attempts to model Codd Nulls, but fails to do so exactly. A more general null concept is one of a *marked null*, where nulls are given identifiers and two nulls with the same marker/identifier are considered equal. Indeed SQL in its current modelling of nulls cannot represent the fact two values may be the same, but that their actual value is unknown.

This project implements marked nulls in the Postgres flavour of SQL. The aims are twofold: firstly to allow us to represent more general results and secondly to enforce correctness in the translations presented in [14].

Acknowledgements

Many thanks to Paolo Guagliardo and Leonid Libkin who were always willing to give their precious time to discuss ideas and results both in person and via email. This project would not have gone very far without their guidance.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Peter Storeng*)

Table of Contents

Chapter 1

Introduction

SQL's evaluation procedure uses three valued logic where the result of logical operations can either be true or false but also *unknown*. Unknown is often thought of as somewhere in between true and false and is caused by the presence of null values.

Null values themselves can be interpreted to mean different things. For instance a null value could represent the fact that a value exists but is missing (missing-but-applicable), conversely it could be interpreted that no value exists/a value does not make sense (missing-and-inapplicable). An example of the missing-and-inapplicable situation could be a PERSON relation with a salary attribute. The salary of someone who is unemployed is not applicable. Indeed Codd himself argued for two distinct null types to represent this, this however would have required a four valued logic query evaluation procedure [5], complicating things further. We note that the missing-and-inapplicable nulls are often due to poor database design so this paper focuses on the case where a null is interpreted as a missing value.

Ultimately SQL must either return a tuple as part of a query or not return such a tuple. This binary result system is at odds with three valued logic, thus at the evaluation point unknown is collapsed to false. This causes some paradoxes, for example consider the query against a PERSON relation with name and age attributes[1]:

```
SELECT * FROM PERSON WHERE age>=18 OR age<18
```

Clearly the `WHERE` constraint in the query is a tautolgy and should return all tuples in the PERSON relation. However tuples with nulls where an age should be present will

[1] Designing tables with age columns is poor design, date of birth columns are used instead. Here we use age to illustrate examples

not be selected. The selection predicates `Null>=18` and `Null<18` both return unknown as does their disjunction which collapses to false under SQL's evaluation procedure.

This "collapsing" in subqueries combined with negation is where SQL starts to produce blatantly wrong answers. Consider the query:

```
SELECT name FROM PERSON WHERE
NOT name IN
(SELECT name FROM PERSON WHERE age>=18 OR age<18)
```

The above query asks for the names of people who do not have an age greater than or equal to 18 OR less than 18, which should of course be nobody. However, SQL returns the names of all the people in the relation who have a null value for age. This is a simplistic query but it serves the point that nulls are dangerous and even with a seemingly harmless straightforward query it is possible to get blatantly wrong answers. In fact it has been shown in [14] that queries from the TPC-H benchmark [1] produce *false positive* results in the presence of nulls and the rate of false positives increases with the null rate in the database.

From the above example it is easy to understand why some database practitioners advocate the avoidance of nulls altogether and a return to boolean logic [7]. Theoreticians have come up with the idea of *certain answers*, answers which will be in the query result regardless of how a null value is interpreted [16, 3]. This is a step forward, however there is an inherent problem with this in that the computation of certain answers is intractable, and at least coNP-complete [4]. Perhaps there is a middle ground where we can approximate certain answers, avoiding wrong answers, but with good complexity bounds.

The conclusion in [18, 14] is that such an approximation is feasible at least for first-order SQL queries where nulls are interpreted as missing values. [14] provides a translation of first order SQL queries to new queries which have correctness guarantees. However these correctness guarantees are based on an assumption of Codd Nulls which SQL does not quite accurately model.

In the Codd Null model every null value is distinct, in SQL this is also the case but in SQL, somewhat surprisingly, a null is not equivalent to itself. Consider for example the table `R(A,B)` with single tuple `(1,NULL)` and the query:

```
SELECT R1.A FROM R R1, R R2
```

```
WHERE R1.A=R2.A AND R1.B=R2.B
```

ie. the query $\pi_A(R \cap R)$, which in SQL returns the empty set and not the intuitive $\{1\}$ [18].

The proposed translations' correctness is also enforced in the marked null model. Marked nulls allow for more general results, thus an implementation of this in SQL would allow for more expressiveness than SQL is currently capable of.

Table 1.1: Codd Null Model

Lecturer	Subject
Jones	Databases
x	Operating Systems
Smith	y
z	Algorithms and Data Structures

Above is an example of Codd nulls being represented by x, y, z. However if we wanted to represent the fact that the Operating System and Algorithms and Data Structures courses were being taught by the same (unknown) lecturer it would be not be possible, since nulls are distinct and do not repeat. In the marked null model, shown below, we are able to do this simply by placing a null with the same ID for both Lecturer values.

Table 1.2: Marked Null Model

Lecturer	Subject
Jones	Databases
x	Operating Systems
Smith	y
x	Algorithms and Data Structures

Marked nulls have already been implemented in association with data integration and data exchange tools [19, 15]. Indeed it is easy to imagine how transforming an attribute which contains nulls from one schema where it appears once to a new one where it appears multiple times can generate marked nulls.

Thus with the motivation of 1. Enforcing correctness on the translation from [14] and 2. The representation of more general results this projects aims to implement marked nulls in SQL.

To do this we will use the open source PostgreSQL. We aim to implement new versions of particular types (integer and varchar for this project) to allow for marked nulls. We will also aim to make these new types as efficient as possible, so indexing and hashing methods for the new types will be implemented. We will further test the new types are behaving as expected and finally do performance testing against databases containing SQL nulls in place of marked nulls.

Chapter 2

Roadmap

In Chapter 3 we will present formally the key concepts and definitions. We will define the ideas of certain answers, correctness guarantees and potential answers. We will show how using these theoretical concepts we can translate queries which may give false positives to ones which do not. One of the key motivating factors behind this project is that these translations are based on an assumption of Codd Nulls or Marked Nulls neither of which SQL's version of Nulls models.

In chapter 4 we state our implementation strategy and in chapters 5 and 6 we will show the steps we took to implement marked nulls for integer and varchar types.

Chapter 7 will show experimental results testing for both correctness and performance.

Finally we conclude in chapter 8 and discuss potential future work.

Chapter 3

Definitions and Preliminaries

A *vocabulary* (or *relational schema*) is a set of relation names with associated arities. A *database* is an instance of such a relational schema. Each relational symbol S of arity k in a database D is a finite subset of $(\mathbf{Adom} = (\mathbf{Const} \cup \mathbf{Null})^k)$ where **Const** and **Null** are countably infinite sets of all the non-null and null values appearing in databases respectively. $\mathbf{Const}(D)$ and $\mathbf{Null}(D)$ denote the sets of constants and nulls appearing in database D. A *tuple* in relational symbol S over database D is denoted $\bar{u} \in S$. We represent nulls with the symbol $\perp$ and we use the marked null model so nulls are allowed to repeat, thus we distinguish nulls by applying subscripts to $\perp$.

A database D is *complete* if $\mathbf{Null}(D) = \emptyset$ and conversely a database is said to be *incomplete* if $\mathbf{Null}(D) \neq \emptyset$. The *active domain* of D is its full domain ie.

$$\mathbf{Adom}(D) = \mathbf{Const}(D) \cup \mathbf{Null}(D)$$

3.1 Homomorphisms, Valuations and Semantics

For relational database instances D and D' over a relational schema a *homomorphism* $h : D \to D'$ is a mapping from $\mathbf{Adom}(D)$ to $\mathbf{Adom}(D')$ where

1. $\forall S \in D$ and tuple $\bar{u} \in S$ then tuple $h(\bar{u}) \in S$ in D'
2. $h(c) = c \; \forall c \in \text{Const}(D)$

Further we say h is a *valuation* if D' is complete. I.e. a valuation replaces any nulls with constant values, leaving constants as they are. A valuation can also be thought of as a mapping from an incomplete database to a complete one. We define $[\![D]\!]$ the semantics of incomplete databases as:

$$[\![D]\!] = \{h(D) \mid h \text{ is a valuation}\} \tag{3.1}$$

where $h(D)$ is the image of $h : D \to D'$. So $h(D) \subseteq D'$. So $[\![D]\!]$ is the set of all possible complete databases an incomplete database could represent.

3.2 Certain Answers and Correctness Guarantees

We can now formally define the certain answers to a query Q over database D as:

$$\text{certain}(Q,D) = \cap\{Q(D') \mid D' \in [\![D]\!]\} \tag{3.2}$$

Essentially a certain answer is one which is in the result of a query no matter how the null values present are interpreted.

There is however an issue with this definition in that any tuples which have null values will not be returned. Consider again the query

```
SELECT * FROM PERSON WHERE age>=18 OR age<18
```

against a simple PERSON instance shown below:

Name	Age
Sarah	18
Alison	$\perp_1$

The certain answers in this case are $\{$(Sarah,18)$\}$ and not the expected $\{$(Sarah,18), (Alison, $\perp_1$)$\}$. This is because $\perp_1$ is interpreted differently in different valuations and thus not in the intersection.

This deficiency motivates the concept of *certain answers with nulls* [18]:

Definition 3.2.1. For an incomplete database D and a query Q of arity k defined over complete databases, then we define *certain answers with nulls*, $\text{certain}_\perp(Q,D)$, as

$$\{\bar{u} \in \text{adom}(D)^k \mid h(\bar{u}) \in Q(h(D)) \text{ for all valuations } h : D \to \textbf{Const}\}$$

We note that for the previous example the certain answers with nulls are $\{$(Sarah,18), (Alison, $\perp_1$)$\}$ since regardless of how $\perp_1$ is interpreted either $h(\perp_1) >= 18$ or $h(\perp_1) < 18$ is true.

Definition 3.2.2. We say a query evaluation algorithm Q has *correctness guarantees* if for any database D its result is a subset of the certain answers with nulls i.e. if $Q(D) \subseteq \text{certain}_{\perp}(Q,D)$.

So a query evaluation method with correctness guarantees will never return wrong answers (false positives). The aim in [18, 14] was to find a translation for a query Q which does not have correctness guarantees to a new one Q' say which does. i.e. Q' such that $Q'(D) \subset \text{certain}_{\perp}(Q,D)$.

3.3 Three Valued Logic

We alluded to three valued logic previously. This is the evaluation procedure which SQL uses in query evaluation. Table 3.1 below shows the complete truth table for such a logic. We note than an unknown truth value is a result of a comparison with a missing value null. The test: $\perp$ op C; always results in unknown. In particular if op is any $=, <>, >, <, >=, <=$. In SQL query σ_θ where θ is the selection condition will select only tuples which evaluate to true and not the ones which evaluate to false or unknown. We will refer to this evaluation procedure as Eval_{SQL}. As shown by way of counter examples in the previous section there is no relationship between this evaluation procedure and certain answers, though it has been shown [18] that if we restrict SQL queries to *unions of conjunctive queries* (UCQs) (queries of the form $\varphi_1 \wedge ... \wedge \varphi_n$) where each φ_i is a conjunctive query) then in that case Eval_{SQL} has certainty guarantees.

Table 3.1: Truth table for Three Valued logic

x	y	x AND y	x OR y	not x
True	True	True	True	False
True	Unknown	Unknown	True	False
True	False	False	True	False
Unknown	True	Unknown	True	Unknown
Unknown	Unknown	Unknown	Unknown	Unknown
Unknown	False	False	Unknown	Unknown
False	True	False	True	True
False	Unknown	False	Unknown	True
False	False	False	False	True

3.4 Translations with Correctness Guarantees

Here we examine translations of queries to ones with certainty guarantees. Importantly these should be good approximations. Indeed if we only wanted certainty guarantees we could simply return nothing for each translated query.

3.4.1 A First Attempt

Instead of attempting to alter SQL's evaluation procedure (Eval_{SQL}) [18] makes a first attempt at providing translations of queries to ones with correctness guarantees for the part of SQL equivalent to relational algebra. It shows that we can make simple changes to the selection conditions of a query so that a query has certainty guarantees. For this translation new operators **const**(attribute) and **null**(attribute) need to be introduced returning true if the value of an attribute is a constant or null respectively. Further new queries formed by closing queries under the operations $\cap, \cup, -, \times, \sigma_\theta, \pi_\alpha$ should also provide certainty guarantees.

This provides some issues particularly due to the difference operator. We use a superscript t to denote a translated query to one with correctness guarantees and a superscript f to denote the translation of a query Q to one with certainty guarantees for the complement to the query, (i.e. $\bar{Q}(D) = \text{Adom}(D)^k - Q(D)$, for k-ary query Q).

The difference of two queries $Q_1 - Q_2$ can be translated to one which has certainty guarantees i.e. $(Q_1 - Q_2)^t$ by: $Q_1^t \cap Q_2^f$. It is clear that this is correct (and proof is given in [18]) since Q_1 except Q_2 is a superset of the certain answers of Q_1 intersected with the certainly not answers to Q_2 (ie certain answers to $\bar{Q}_2$).

The issue here is that calculating Q^f, though it has good theoretical bounds, is expensive in practice as it requires calculating the active domain and cartesian products of this. Further to that the Q^f queries can be infeasibly complex.

3.4.2 A Better Solution

A better solution introduced in [14] emerged from a better understanding of how to approximate certain answers to the difference query. To do this we use a concept of potential answers which is a more compact improvement of *maybe answers* [22] where a maybe answer to $Q(D)$ is one in $Q(v(D))$ for any valuation v. To explain this we need some definitions.

Definition 3.4.1. For a query Q with arity k on incomplete database D, we say $A \subseteq \mathbf{Adom}^k$

represents *potential answers* to Q on D if

$$Q(v(D)) \subseteq v(A),\ \forall \text{ valuations } v$$

A query Q' returns potential answers to Q if $Q'(D)$ represents potential answers to Q on database $D, \forall D$.

If we have a method for calculating potential answers we could see how we could get certainty guarantees for calculating the difference $Q_1 - Q_2$. Conceptually the certain answers would be the answers which are certain answers to Q_1 but *not* possible answers to Q_2. We use a modified definition of the anti-semijoin to formally define the not.

Definition 3.4.2. We say two tuples $\bar{r}$ and $\bar{s}$ of same length, k, over **Const** $\cup$ **Null** *unify* and write $\bar{r} \Uparrow \bar{s}$ if $\exists$ a valuation v such that $v(\bar{r}) = v(\bar{s})$.

Definition 3.4.3. For relations R and S with matching attributes over **Const**$\cup$**Null** we define the *left unification semijoin* as

$$R \ltimes_{\Uparrow} S = \{\bar{r} \in R \mid \exists \bar{s} \in S : \bar{r} \Uparrow \bar{s}\}$$

and the *left unification anti-semijoin* as

$$R \bar{\ltimes}_{\Uparrow} S = \{\bar{r} \in R \mid \nexists \bar{s} \in S : \bar{r} \Uparrow \bar{s}\}$$

which is equivalent to $R - (R \ltimes_{\Uparrow} S)$

Now we present the full translations for $Q \to Q^+$ and $Q \to Q^?$ described in [14].

$$R^+ = R \quad (1.1) \qquad R^? = R \quad (2.1)$$

$$(Q_1 \cup Q_2)^+ = Q_1^+ \cup Q_2^+ \quad (1.2) \qquad (Q_1 \cup Q_2)^? = Q_1^? \cup Q_2^? \quad (2.2)$$

$$(Q_1 \cap Q_2)^+ = Q_1^+ \cap Q_2^+ \quad (1.3) \qquad (Q_1 \cap Q_2)^? = Q_1^? \ltimes_{\Uparrow} Q_2^? \quad (2.3)$$

$$(Q_1 - Q_2)^+ = Q_1^+ \bar{\ltimes}_{\Uparrow} Q_2^? \quad (1.4) \qquad (Q_1 - Q_2)^? = Q_1^? - Q_2^+ \quad (2.4)$$

$$(\sigma_\theta(Q))^+ = \sigma_{\theta^*}(Q^+) \quad (1.5) \qquad (\sigma_\theta(Q))^? = \sigma_{\theta^{**}}(Q^?) \quad (2.5)$$

$$(Q_1 \times Q_2)^+ = Q_1^+ \times Q_2^+ \quad (1.6) \qquad (Q_1 \times Q_2)^? = Q_1^? \times Q_2^? \quad (2.6)$$

$$(\pi_\alpha(Q))^+ = \pi_\alpha(Q^+) \quad (1.7) \qquad (\pi_\alpha(Q))^? = \pi_\alpha(Q^?) \quad (2.7)$$

Note that σ^* and σ^{**} refer to the translation of selectivity conditions which we give below. Firstly for θ^*

$$
\begin{aligned}
(A = B)^* &= (A = B) \\
(A = c)^* &= (A = c) \text{ if } c \text{ a constant} \\
(A \neq B)^* &= (A \neq B) \wedge \mathbf{const}(A) \wedge \mathbf{const}(B) \\
(A \neq c)^* &= (A \neq c) \wedge \mathbf{const}(A) \\
(\theta_1 \vee \theta_2)^* &= \theta_1^* \vee \theta_2^* \\
(\theta_1 \wedge \theta_2)^* &= \theta_1^* \wedge \theta_2^*
\end{aligned}
$$

And next for for θ^{**}:

$$
\begin{aligned}
(A = B)^{**} &= (A = B) \vee \mathbf{null}(A) \vee \mathbf{null}(B) \\
(A = c)^{**} &= (A = c) \vee \mathbf{null}(A) \\
(A \neq B)^{**} &= (A \neq B) \\
(A \neq c)^{**} &= (A \neq c) \text{ if } c \text{ is a constant} \\
(\theta_1 \vee \theta_2)^{**} &= \theta_1^{**} \vee \theta_2^{**} \\
(\theta_1 \wedge \theta_2)^{**} &= \theta_1^{**} \wedge \theta_2^{**}
\end{aligned}
$$

[14] proves that for the translation $Q \to (Q^+, Q^?)$, Q^+ provides correctness guarantees for Q and $Q^?$ represents potential answers to Q. Further to this it shows the queries have the data complexity AC^0, as does SQL, and shows empirically that they work well in practice.

Thus a workable solution which provides certainty guarantees for SQL queries equivalent to relational algebra seems realistic. Of course the proof that these translations are correct is based on the Codd Null model. Therefore the next logical step is to implement Codd Nulls and check the translated queries still perform well. Here we choose to implement marked nulls as the translations still hold in this case and we have the added benefit of improving the expressive power of SQL. The upcoming chapters detail the steps taken towards such a goal.

Chapter 4

Marked Nulls in postgresSQL

To implement marked nulls we will use PostgreSQL due to its open source nature, because it is relatively well documented [11] and it is simple to extend due to its catalog driven nature. PostgreSQL itself is mostly coded in C and there is a well defined framework for incorporating extensions through dynamic loading. The loading of shared libraries updates the internal system catalogs allowing modification of PostgreSQL's inner workings 'on the fly'.

4.1 Strategy

Postgres treats Null values independent of the actual type and tends to handle them separately. The logic for the handling of nulls is hardcoded into the postgreSQL internals and so such behaviour for marked nulls is impossible for us to implement without making drastic changes to the core code. Thus to implement marked nulls we shall build them in to the type itself. Therefore, our strategy in implementing the marked nulls will be to create a new version of each type which will be composed of the *base* type and a boolean flag representing whether or not it is a constant.

Defining the new type is relatively simple, however making it useful and efficient is where the complexity lies. To make it useful we must define new operators and functions corresponding to existing ones in the base type. A major issue we face here is that we must handle marked null values explicitly in these operators and therefore must implement three-valued logic. Since SQL nulls are handled separately in Postgres the base type operators do not need to do this.

For the new type to be efficient we must also define how indexing and hashing should work on the new type.

The upcoming sections describe in detail the steps that where taken to implement marked nulls for types equivalent to integer and varchar in PostgreSQL.

Chapter 5

Null Integer Type

As mentioned in the previous section we simply define the new type as a composite type. In C we can do this by using a struct (see definition below). In this case if the boolean cnst component is set to true then the value component is treated as a normal integer. Conversely if the boolean cnst component is set to false then the value component is treated as if it where an identifier (ID) of a marked null.

```
typedef struct Null_integer
{
        int     value;
        bool    cnst;
}       Null_integer;
```

5.1 Input Output Functionality

The above definition simply defines a C struct but we have not yet been able to relate that to a type in Postgres. We can do this using the `CREATE TYPE` construct in Postgres. The minimum arguments that are required are input and output functions so that Postgres knows how to store the type and how it should present itself when printed to screen.

```
CREATE TYPE null_integer (
   INPUT = null_integer_in,
   OUTPUT = null_integer_out
);
```

Where we have already created the SQL input/output functions as:

```
CREATE FUNCTION null_integer_in(cstring)
```

```
   RETURNS null_integer
   AS 'null_integer', 'null_integer_in'
   LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION null_integer_out(null_integer)
   RETURNS cstring
   AS 'null_integer', 'null_integer_out'
   LANGUAGE C IMMUTABLE STRICT;
```

Above is the SQL code which creates the SQL functions `null_integer_in` and `null_integer_out` specified in the type. The actual implementations of these is in the corresponding C functions.

The `IMMUTABLE` tag provides extra information to the query optimiser. It states that the function cannot modify the database and always returns the same result under the same input arguments, thus certain optimisations can be made.

The `STATIC` tag indicates that the functions will return unknown on (SQL) null input. Therefore on null input the function is not executed, a null is returned immediately to allow for further optimisation.

Almost all operators in the base type are declared as `STRICT` allowing them to not have to deal with SQL nulls explicitly. We shall also declare our new type's operators as `STRICT`. However we note that a mixing of SQL nulls with marked nulls isn't really intended and recommend that marked null type attributes are declared `NOT NULL` so marked nulls are allowed but not SQL nulls.

By following the steps described in [13] we can import the basic version of our extension and insert and display the data.

```
DemoDir=# CREATE EXTENSION null_integer;
DemoDir=# CREATE TABLE PERSON(Name varchar(100), Age
   null_integer NOT NULL);
CREATE TABLE
DemoDir=# INSERT INTO PERSON VALUES('Sarah','18'),('
   Alison','NULL:1');
INSERT 0 2
DemoDir=# SELECT * FROM PERSON;
  name   |  age
---------+--------
```

```
 Sarah   | 18
 Alison | NULL:1
```

One annoyance of the above is that we must read input as strings, in particular we are required to write '18' instead of 18. A good way to fix this issue is to define a CAST function from integers to null_integers.

To input marked nulls we must use the format 'NULL:ID'. The input function can then detect that the constant flag should be set to false. Note the 'NULL:' characters are not restricted to being upper case and are not stored in the data structure itself, but are stripped off in the input function and prepended back on in the output function. A 'nice' extra feature which we have not implemented would be to make this prepended body customisable. Also, inside the input function we have handling for integer overflow and ensuring the value only contains digits:

```
DemoDir=# INSERT INTO PERSON VALUES('Hannah','fifty-five'
   );
ERROR:  22P02: invalid input syntax for null_integer: "
   fifty-five"

DemoDir=# INSERT INTO PERSON VALUES('Hannah','NULL:fifty-
   five');
ERROR:  22P02: invalid input syntax for null_integer: "
   NULL:fifty-five"

DemoDir=# INSERT INTO PERSON VALUES('Hannah','
   188496353453453');
ERROR:  22003: value "188496353453453" is out of range
   for type integer
```

5.2 Operators

Operators are implemented as monadic/dyadic functions with left or/and right arguments. As an example we will discuss the implementation of the = operator shown below. Note that the `CREATE OPERATOR` construct is a Postgres extension and not a part of the SQL standard [9].

```
CREATE OPERATOR = (
```

```
    leftarg = null_integer,
    rightarg = null_integer,
    procedure = null_integer_eq,
    commutator = = ,
    negator = <> ,
    restrict = mk_eqsel,
    join = mk_eqjoinsel,
    HASHES,
    MERGES
);
```

The leftarg and rightarg specifications allow us to use the more intuitive infix notation, with the procedure clause specifying the underlying function. ie:

```
DemoDir=# SELECT * FROM PERSON WHERE age = 18;
 name  | age
-------+-----
 Sarah | 18
```

As opposed to calling the underlying procedure directly:

```
DemoDir=# SELECT * FROM PERSON WHERE null_integer_eq(age
   ,18);
 name  | age
-------+-----
 Sarah | 18
```

The procedure given relates to the SQL function defined as:

```
CREATE FUNCTION null_integer_eq(null_integer,
   null_integer)
   RETURNS bool
   AS 'null_integer', 'null_integer_eq'
   LANGUAGE C IMMUTABLE STRICT PARALLEL SAFE;
```

Which in turn is defined in C with a function of the same name. The `PARALLEL SAFE` refers to a new feature in PostgreSQL version 9.6 that the function can be executed safely in parallel mode. The C function itself is where the implementation of all the logic is including the three valued logic which we now demonstrate. Firstly we enter some new data:

```
DemoDir=# INSERT INTO PERSON VALUES('Hannah','NULL:2'),('
   Chris',25),('Alistair','NULL:1');
INSERT 0 3
DemoDir=# SELECT * FROM PERSON;
   name    |  age
-----------+--------
 Sarah     | 18
 Alison    | NULL:1
 Hannah    | NULL:2
 Chris     | 25
 Alistair  | NULL:1
```

We have entered data for Hannah whose age is missing and Chris who is 25. We have also entered data for Alistair who is Alison's twin. We don't know Alistair's age but we do know that it must be the same as Alison's. This fact was not previously expressible in SQL, thus we have achieved one part of the aim of this project. We can now check the logic behind = is working properly:

```
DemoDir=# \pset null 'u'
Null display is "u".
DemoDir=# SELECT name,age='NULL:1' FROM PERSON;
   name    | ?column?
-----------+----------
 Sarah     | u
 Alison    | t
 Hannah    | u
 Chris     | u
 Alistair  | t

DemoDir=# SELECT name,age=18 FROM PERSON;
   name    | ?column?
-----------+----------
 Sarah     | t
 Alison    | u
 Hannah    | u
 Chris     | f
```

```
 Alistair | u
```

We set nulls to be shown as `'u'` standing for unknown and first test if any ages are equal to 'NULL:1'. This is true for Alison and Alistair but since we don't what missing value 'NULL:1' represents the remaining results are unknown. The second query is shown for completeness to demonstrate that $A = B$ where A and B are constants will return a definite answer.

Also note that the intersection query involving nulls from chapter 1 now returns the intuitive result:

```
DemoDir=# CREATE TABLE R(A null_integer,B null_integer);
CREATE TABLE
DemoDir=# INSERT INTO R VALUES(1,'NULL:999');
INSERT 0 1
DemoDir=# SELECT R1.A FROM R R1, R R2 WHERE R1.A=R2.A AND
    R1.B=R2.B;
 a
---
 1
```

The real power in creating operators is down to supplying extra information to the query optimiser. This extra information can lead to significant performance benefit. The remaining clauses in the `CREATE OPERATOR` construct above exist for this reason, we now summarise these:

5.2.1 COMMUTATOR

This clause simply states that for operator $\texttt{OP}_1$ which has a commutator $\texttt{OP}_2$ then:

$$A \,\texttt{OP}_1\, B \iff B \,\texttt{OP}_2\, A$$

I.e. we can switch the positions of A and B and use $\texttt{OP}_2$ instead of $\texttt{OP}_1$. The reason that this clause is important to Postgres is that if B is a column with an index on it Postgres cannot make use of this fact without the commutator clause because it always expects the index column to be the left argument. Clearly equality is commutative thus the commutator above is itself.

5.2.2 NEGATOR

$\texttt{OP}_2$ is a negator to $\texttt{OP}_1$ if:

$$A\ \texttt{OP}_1\ B \iff \texttt{NOT}(A\ \texttt{OP}_2\ B)$$

The benefit of this is it allows expressions to strip the `NOT` operator and replace $\texttt{OP}_2$ with $\texttt{OP}_1$. The negator to equality is clearly not equal which we will implement as $<>$.

5.2.3 RESTRICT

The restrict clause allows us to name a *selectivity estimation function*. These are functions which provide estimates on the number of rows in a table that will satisfy a `WHERE` condition involving this operator. The optimiser then has a good idea of how many rows will be selected.

5.2.4 JOIN

Similar to the restrict clause the join clause specifies a *join selectivity estimation function*. I.e. it estimates the percentage of rows in a table will satisfy the join condition in a `WHERE` condition of the form: `table1.column1 OP table2.column2`.

5.2.5 HASHES & MERGES

Hashes means it is permissible to use hashing for join/selection conditions involving this operator. For this to work a hash function must be defined. The merges clause indicates it is possible to use this operator in a merge join. For this to work there must be a total order defined on the data type. These two clauses are desirable as they tend to be much faster than the default nested loops join method. However they cause some issues on implementation for the new type which we shall discuss in detail subsequently.

5.3 Indexing, Ordering and Merging

PostgreSQL provides a simple structure for implementation of different indexing strategies. To implement a B-tree index we just need to be able define a total ordering for a type and Postgres will handle the rest. The PostgreSQL code for doing so is shown below:

```
CREATE OPERATOR CLASS null_integer_ops
    DEFAULT FOR TYPE null_integer USING btree AS
    OPERATOR    1       <& ,
    OPERATOR    2       <=& ,
    OPERATOR    3       = ,
    OPERATOR    4       >=& ,
    OPERATOR    5       >& ,
    FUNCTION    1       null_integer_btree_cmp(
       null_integer, null_integer);
```

Postgres uses the notion of an operator class. This specifies that certain operators will fulfil different roles. We are free to specify the operators for these roles when creating an operator class. For example for the B-tree indexing method operator 1 should be a 'less than' operator, operator 2 'less than or equal', operator 3 'equal', operator 4 'greater than or equal' and operator 5 'greater than'.

Notice here we have appended the ampersand character to the expected operators (except for equality). This is because we must be able to define a total order for a B-tree index to make sense. The normal inequality operators will return unknown on a marked null input, this cannot be allowed here. Thus we create the 'dual' to the normal inequality operators which define a total order with the marked nulls 'less' than constants. We reserve $<, <=, >=, >$ for the correct three value operators. To demonstrate consider '<&'.

We say A <& B is true if:

1. If A and B are both constants and $A < B$.
2. If A is a marked null and B is a constant.
3. If A is a marked null with ID ID_A and B is a marked null with ID ID_B such that $\mathrm{ID}_A < \mathrm{ID}_B$.

and false otherwise (never unknown).

For B-tree indexing Postgres also requires us to define a comparison function which returns a negative number, 0 or a positive number depending on whether the first argument is less than equal to, equal to or greater than the second argument. This is the role of `null_integer_btree_cmp` function above.
We are now able to create indexes and are able to do ordering on the null_integer column with respect to the operators defined for indexing:

```
DemoDir=# SELECT * FROM PERSON ORDER BY age;
   name    |  age
-----------+--------
 Alison    | NULL:1
 Alistair  | NULL:1
 Hannah    | NULL:2
 Sarah     | 18
 Chris     | 25

DemoDir=# CREATE INDEX ageIndex ON PERSON(age);
CREATE INDEX

DemoDir=# \d PERSON
            Table "public.person"
 Column |          Type          | Modifiers
--------+------------------------+-----------
 name   | character varying(100) |
 age    | null_integer           | not null
Indexes:
    "ageindex" btree (age)
```

Since the datatype is now capable of being fully ordered the `MERGES` clause in the definition of the = operator now makes sense as a merge join works on the principle that two tables can be sorted by their join column(s).

5.4 Hashing

Similar to the `MERGES` clause requiring a total order to be defined for the type, the `HASHES` clause requires a hash function to be defined for the type. We define the hashing operator class as such:

```
CREATE OPERATOR CLASS null_integer_ops
    FOR TYPE null_integer USING hash AS
    OPERATOR    1       = ,
    FUNCTION    1       hash_null_integer(null_integer);
```

where the `hash_null_integer` specifies a hashing function ultimately implemented in C. Here instead of writing our own hash function we indirectly call one defined in existing Postgres libraries. This is an adaptation of 'My Hash' [17].

5.5 Extra Functions

The translations described in 3.4 require operators **const()** and **null()** which return true/false on constant input and false/true on (marked/codd) null input respectively. Thus we create the functions `is_null()` and `not_null()` for this purpose:

```
CREATE FUNCTION is_null(null_integer)
    RETURNS bool
    AS 'null_integer', 'is_null'
    LANGUAGE C IMMUTABLE PARALLEL SAFE;

CREATE FUNCTION not_null(null_integer)
    RETURNS bool
    AS 'null_integer', 'not_null'
    LANGUAGE C IMMUTABLE PARALLEL SAFE;

CREATE OPERATOR @@ (
    leftarg = null_integer,
    procedure = is_null,
    restrict = mknull_sel,
    negator = @!
);

CREATE OPERATOR @! (
    leftarg = null_integer,
    procedure = not_null,
    restrict = mk_not_null_sel,
    negator = @@
);
```

Here we also create operators corresponding to the functions. This is so that we can take advantage of providing the restrict and negator clauses for optimisation.

5.6 Issues

Issues with this implementation of marked nulls all stemmed from attempting to implement three-valued logic into operators that Postgres expects should use boolean logic. Although Postgres implements three-valued logic it does so by labelling operators strict. This allows nulls to be handled outside the operator. Since these operators never receive a null input they never return null/unknown output.

Postgres uses the terminology that a function is *complete* if it never returns null/unknown on non-null input. In the eyes of the Postgres internals our null_integer operators are not complete since an operation involving a marked null, which is built into the type and considered by Postgres a normal non-null value, will return unknown.

By default this is allowed however once we try to make certain optimisations Postgres starts to complain. In particular the inbuilt restrict and join clauses for equality like operators 'eqsel' and 'eqjoinsel' require the operator to be complete or they will return an error. This was a major issue as without this optimisations a major performance hit is incurred.

To remedy this we went through the onerous task of creating dual versions of `eqsel` and `eqjoinsel`, `mk_eqsel` and `mk_eqjoinsel` respectively. This essentially involved copying the underlying C functions and any sub-functions and editing the code sections which caused error on unknown output. We instead returned false on unknown output. We packaged this class of functions into another extension thus allowing us to keep the optimisations required.

Chapter 6

Null Varchar Type

The implementation of the Null Varchar datatype turned out to be significantly more challenging than implementation of a Null Integer datatype. The reason for this of course seeds from its varaiable length nature.

In Postgres all variable length data types share the 'varlena' header defined in C as:

```
struct varlena
{
        char    vl_len_[4];
        char    vl_dat[];
};
```

In particular we see the C VarChar type (the underlying datatype for the corresponding PosgreSQL Varchar) is defined as so:

```
typedef struct varlena VarChar;
```

The varlena struct is a struct containing a *flexible array member* where the last element is an incomplete array. The first element of the struct, vl_len_, is used to describe the variable length data portion vl_dat which for varchar types will store the string. Note this is different to the normal way of storing C strings in memory as a pointer to a null terminated string.

Postgres chooses to use this method because firstly storing pointers to disk is meaningless and so requires extra handling and secondly this allows for the storing of data contiguously on disk allowing for faster disk reads.

The obvious way therefore to define our Null Varchar type would be as follows:

```
typedef struct Null_varchar
{
    bool        cnst;
```

```
    VarChar     value;
}    Null_varchar;
```

Again using the cnst flag to indicate whether value is a marked null, and so contains its ID, or if it a normal constant value.

The C standard states that a struct containing a flexible array member ”shall not be a member of a structure or an element of an array” [6]. This effectively ruled this method out so we instead defined it as such:

```
typedef struct Null_varchar
{
    char        nv_len[4];
    bool        cnst;
    char        nv_dat[];
}    Null_varchar;
```

Note that after proceeding with this definition further research showed that the C standard does in fact allow for a nested struct with a flexible array member in the case it is the last element of that structure [20]. In any case the two definitions amount to the same thing and in fact the existing C macros for setting and getting both the size and data values would be required to be altered for both methods. We discuss this in the next subsection.

6.1 Extra Complications

Using the flexible array member method to store variable length data has some added complications. The input function must be careful to allocate the correct amount of memory on the heap. The input function is provided with the input as a null terminated string as usual. From this we can extract the length and add the extra offset (5 bytes) required for the nv_len and cnst members to allocate memory for the Null_varchar struct.

6.1.1 Type Modifiers and Length Coercion

Before we define the type input function we must first define *type modifier input/output functions*. These are required for types that support extra constraints attached to a type declaration known as *modifiers* [10]. The Varchar datatype supports an integer modifier which specifies the maximum number of characters that it can hold. Similarly this is mandatory for the Null Varchar type. The type modifier should be made available to

the input function so that Varchar entries with too many characters can be rejected before being inserted to the table. Once these functions are defined we can create the type:

```
CREATE TYPE null_varchar (
   input = null_varchar_in,
   output = null_varchar_out,
   LIKE = pg_catalog.varchar,
   typmod_in =  null_varchartypmodin,
   typmod_out = null_varchartypmodout
);
```

In fact although the type modifier is passed as an argument to the input function, oddly it is not the correct value. Instead to ensure the input is of correct length we must define a *length coercion cast function* which is called after the input function (if defined) to ensure the length is correct:

```
CREATE FUNCTION null_varchar(null_varchar, integer,
   boolean)
    RETURNS null_varchar
    AS 'null_varchar', 'null_varchar'
    LANGUAGE C IMMUTABLE STRICT PARALLEL SAFE;

CREATE CAST (null_varchar AS null_varchar)
    WITH FUNCTION null_varchar(null_varchar, integer,
       boolean)
    AS IMPLICIT;
```

This behaviour was entirely undocumented and required much trial and error to be uncovered. We note the length coercion function has the same name as the type as this matches the way other length coercion functions are defined. Also, the `CAST` function 'casts' from null_varchar to null_varchar. Essentially the underlying function checks the null_varchar does not have more characters than it should. If it does but these are only whitespace characters these are stripped out, otherwise an error is returned:

```
DemoDir=# CREATE TABLE PERSON(Name null_varchar(10) NOT
   NULL, Age null_integer NOT NULL);
CREATE TABLE
DemoDir=#
```

```
DemoDir=# INSERT INTO PERSON VALUES('Sarahhhhhhhh','18');
ERROR:  22001: value too long for type character varying
   (10)
LOCATION:  null_varchar, null_varchar.c:247
DemoDir=#
DemoDir=# INSERT INTO PERSON VALUES('Sarah       ','18');
INSERT 0 1
```

6.1.2 Macros and Bit layouts

Recall the deinition of the varlena struct included a 4 character array header and the flexible array member. By inspecting the source code and comments in [2] we see that Postgres uses the first two bits of the first byte in v_len_ to store information about how the data is stored (compressed/uncompressed and aligned/unaligned). The remaining bits are for the length of the data.

Using the 'first two' bits has a different meaning depending on the endianess of the machine the code is running on. Thus there are two versions of the get/set macros depending on the architecture. Big endian machines require us to mask the first two bits to extract the length whereas little endian machines require us to shift two bits. See the comments in [2] for further details.

We note these details because the new type has the extra cnst member causing the existing access macros to be invalid and new versions of them to be written adding significantly to the complexity, see the header file in Appendix A for the definition of these macros (specifically for little endian machines). Another method to define the Null_varchar struct may have been to typedef it to be the same as the varlena struct and use the first byte of the vl_dat array to encode the constant/null information. However this resolves to the same problem as macros will still be required to access the data/constant value.

6.2 Operators

We leave most of details of the implementation of the operators for the null_varchar type out as they are generally similar to the null_integer type. We just remark that to define an ordering we use the normal alphabetic ordering with marked nulls appearing before constants (using alphabetic ordering on IDs within marked nulls). This

can then be used for merge joins and indexing. As before we use two versions of the inequality operators, the original $<, <=, >, >=$ ones for three valued logic and $<\&, <=\&, >\&, >=\&$ to define a total order.

Additionally we implement a `like` function for null_varchar. This function behaves as expected when the comparison involves constants, will always return unknown if one of the two comparisons is a marked null the only exception to this being when the two things being compared are the same marked null.

```
DemoDir=# SELECT * FROM PERSON;
   name    |  age
-----------+--------
 Sarah     | 18
 Alison    | NULL:1
 Hannah    | NULL:2
 Chris     | 25
 Alistair  | NULL:1
 NULL:AAA  | NULL:2

DemoDir=# SELECT name LIKE 'NULL:A%',name LIKE 'A%',name
   LIKE 'NULL:AAA' FROM PERSON;
 ?column? | ?column? | ?column?
----------+----------+----------
 u        | f        | u
 u        | t        | u
 u        | f        | u
 u        | f        | u
 u        | t        | u
 u        | u        | t
```

In particular the comparison `NULL:AAA like 'NULL:A%'` is unknown as the `'AAA'` is just an ID and though the IDs seem to match this says nothing about the missing data that they represent.

Chapter 7

Experiments

Having implemented the new types we now show some experimental results. We will firstly test for correctness and then performance.

7.1 Correctness Tests

The testing that the translations in [14] are correct was completed as part of that research. The correctness which we wish to test for is that if we replace the integer and varchar data types with null_integer and null_varchar datatypes and populate incomplete databases with precisely the same data (except replacing SQL nulls with marked nulls) that the marked null queries return the exact same results.

7.1.1 Experimental Set Up

We create two mirror incomplete databases (one for SQL nulls and one for marked nulls) and check that we get the same query results (possibly with marked nulls in place of SQL nulls).

As done in [14] we will use the DBGen tool from the TPC-H benchmark to construct databases of size approximatley 1GB. The TPC-H database and queries are chosen ”to have broad industry-wide relevance” [21]. We will use the four translated queries from [14]. Two of which are modified TPC-H queries and two of which are standard textbook queries [8] with subqueries listed in Appendix C.

To test the queries in the two separate databases are the same we simply save the query results to csv files and use the linux diff command to check they are the same.

Note that we need to slightly modify the queries for the marked null database to use

the `is_null()` and `not_null()` functions instead of `IS NULL` or `IS NOT NULL`. In fact instead we use the equivalent operators `@@` and `@!` (see chapter 5.5) to take advantage of operator optimisation.

```
experimentsSQL=# COPY 'Q1' To 'q1_SQL.csv' With CSV;
experimentsSQL=# COPY 'Q2' To 'q2_SQL.csv' With CSV;
experimentsSQL=# COPY 'Q3' To 'q3_SQL.csv' With CSV;
experimentsSQL=# COPY 'Q4' To 'q4_SQL.csv' With CSV;
```

experimentsSQL is the database with integer and varchar datatypes with SQL nulls. We replace Q1- Q4 with the queries shown in Appendix C.

```
experiments=# COPY 'Q1' To 'q1_marked.csv' With CSV;
experiments=# COPY 'Q2' To 'q2_marked.csv' With CSV;
experiments=# COPY 'Q3' To 'q3_marked.csv' With CSV;
experiments=# COPY 'Q4' To 'q4_marked.csv' With CSV;
```

experiments is the database with null_integer and null_varchar datatypes and marked nulls.

```
$: diff q1_marked.csv q1_SQL.csv
$: diff q2_marked.csv q2_SQL.csv
$: diff q3_marked.csv q3_SQL.csv
$: diff q4_marked.csv q4_SQL.csv
```

We run the diff command comparing the csv output. No differences are shown. We generated 10 different mirror instances and compared the queries as above each time and continued to find no differences. Thus we conclude the marked null type and operators work as expected.

7.2 Performance Testing

We have shown that queries on the marked null data types are correct. However for them to become useful the extra query latency should not be too expensive. We shall complete two performance tests.

Firstly, for the four queries, we will compare the original untranslated query on SQL null database (with varchar and integer datatypes), shown in Appendix B, against the best translated query from [14] on the marked null database (with null_varchar and null_integer datatypes), shown in Appendix C. This gives us the full price of correctness taking into account the translations and marked nulls.

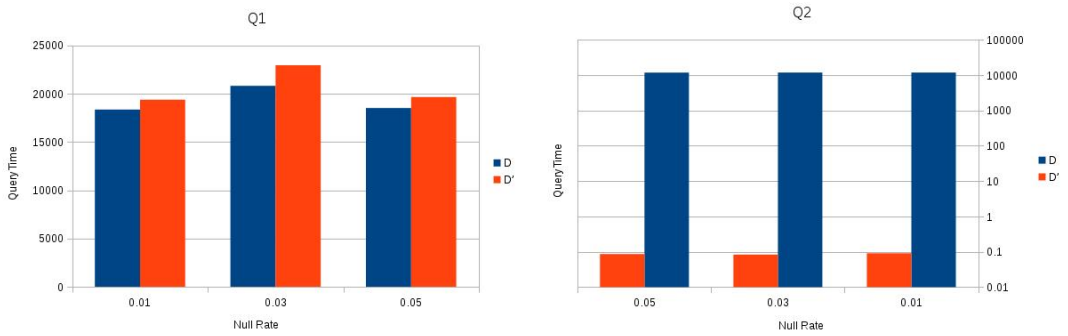

Figure 7.1: Q1 and Q2 statistics

Secondly we will compare the original queries on SQL null database against the original queries on the marked null database. This will give the pure cost of the marked nulls.

For each of these tests we will randomly generate null values at three different null rates $(0.01, 0.03$ and $0.05)$. For each of these rates we generate 5 pairs of databases (D, D'). Where D is populated with SQL nulls and uses varchar and integer datatypes and D' uses the null_varchar and null_integer datatypes and has marked nulls where D has SQL nulls for columns of these types. Then we create 5 instantiations of each query (different arguments each time) which we run three times on each database pair. We time these and inspect the results.

7.3 Full Price of Correctness

We see the comparisons for Q1 and Q2 in figure 7.1 and for Q3 and Q4 in figure 7.2. We see that the full price of correctness for queries Q1 and Q3 is cheap. For these two queries the relative cost is less than 10% for Q1 and less than 2% for Q3 for all null rates. For query Q2 there was a big speedup, around 10^5 times faster. This was expected and a similar speedup was shown in [14].

The most interesting result is for Q4. Here we see around a 3.5 times slowdown. [14] reported a factor 2 slowdown without the marked nulls. Is this extra slowdown due the marked nulls? we return to this question after studying the pure cost of marked nulls in the upcoming section.

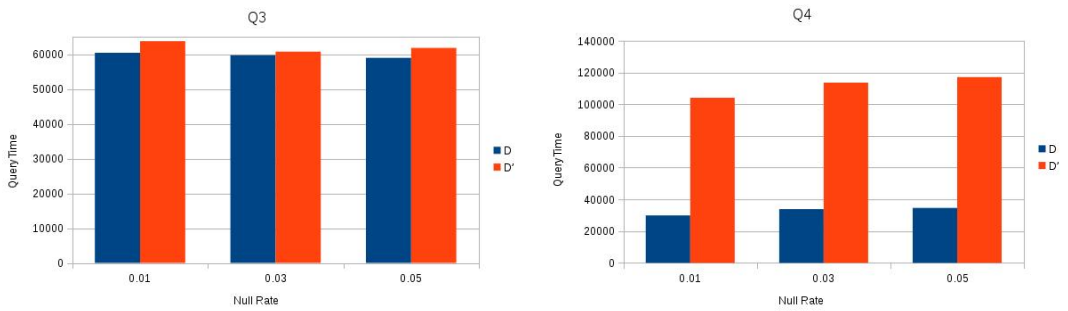

Figure 7.2: Q3 and Q4 statistics

7.4 Pure Cost of Marked Nulls

Next we attempted to assess the pure cost of marked nulls by comparing the original query on the SQL database against the original query on the marked null database (queries in Appendix B). However in running these tests we ran into a major issue. The query Q4 running on the marked null database failed with error:

```
ERROR:  XX000: function 608457 returned NULL
LOCATION:  FunctionCall2Coll, fmgr.c:1327

experiments=# SELECT proname FROM pg_proc WHERE oid
   =608457;
     proname
-----------------
 null_integer_eq
```

And by checking Postgres internal catalog tables we see that the underlying function `null_integer_eq` associated with the `'='` operator is the cause of the issue. This was the same error which was returned when we observed (and fixed) issues with the restrict and join clauses in chapter 5.6. In particular the failure was inside the C function `FunctionCall2Coll` and due to the operator returning null on non-(SQL) null input.

We were able to track this down to the index scan. Indeed if we turned index scanning off in Postgres by running `set enable_indexscan = false;` and rerunning the query it succeeded. Fixing this issue in the way we did previously, by copying code and removing the hard coded errors on null output for this is infeasible due to the large swathes of code involved, if even possible. In fact on further research it seems that this

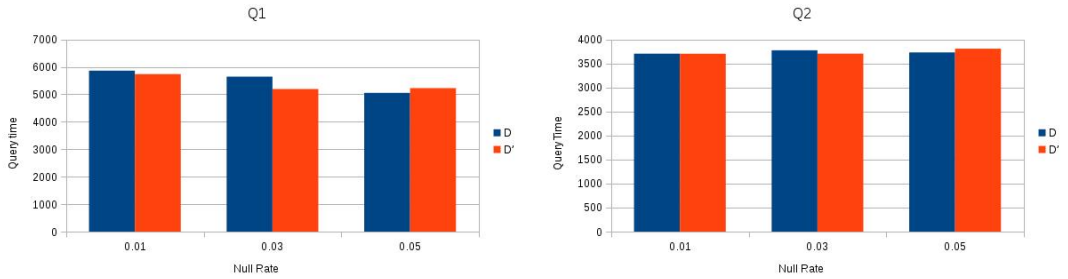

Figure 7.3: Q1 and Q2 pure cost of marked null statistics

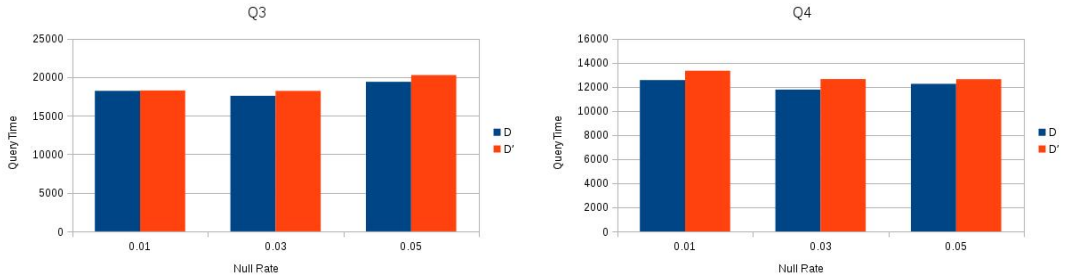

Figure 7.4: Q3 and Q4 pure cost of marked null statistics

three valued behaviour is not allowed for hash joins either. With respect to hashing we read in [12] that the underlying function should ”be complete: that is, it should return true or false, never null, for any two nonnull inputs.... it might yield an error complaining that it wasn’t prepared for a null result”. Thus despite not observing any issues with hashing it seems it is not even a good idea for our three valued `'='` to be involved in hash joins.

With no immediate fix to this problem we decided to implement a two valued equality operator `'=&'` which returned false in all operations involving marked nulls (except for returning true on comparing the same marked null). Using the two valued equals which could do hashing and indexing without any issues we generated ’pure marked null cost’ results.

In figure 7.3 and figure 7.4 we see the pure cost of marked nulls. Indeed for all queries and for all null rates the queries against the marked null database are within a couple of percentage points of the SQL queries. This is of course using the two valued equality operator rather than the 3 valued.

One outstanding question is why there is an extra slowdown in the translated Q4 in the marked null database compared to the translated query in SQL null database. It was shown to be around a factor 2 slower in [14] but here we observe a factor 3.5 slowdown.

The pure cost of marked nulls seems to be almost free from our results in figures 7.3 and 7.4. However these tests used the original untranslated query. In particular the translations on the marked null database uses the `is_null` and `not_null` functions via their operators '@@' and '@!'. For these we did create our own restrict optimisation functions. However these were very crude and returned hardcoded estimations. We conjecture that Posgres by treating null values separately from their types can more easily gather statistics on these which can be used for optimisation which may be part of the extra slowdown. If this is the case it is hard to see how we can optimise `is_null` and `not_null` further since the marked nulls are built in to the type. Further research is required here to check whether this is the cause of the extra slowdown, and if so research into Postgres' optimisation techniques should be undertaken to see if it can be improved.

Chapter 8

Conclusions and Future Work

This research has implemented marked nulls in SQL. We have seen this fixes some unreasonable deficiencies of SQL's current null model. Using this marked null model we can say definitively that a null is equal to itself. Further to this we are also able to represent the fact that two or more missing values are the same, albeit unknown, value.

We have confirmed that queries involving the marked null types return correct results. One of the main motivating factors behind this project was to enforce correctness on the translations with certainty guarantees first proposed in [18] and further improved on in [14]. We saw that the translations combined with the marked null datatypes performed at least not significantly slower in three out of the four test queries. On one query we saw a 3.5 times slowdown. This is not too disappointing however as we are still within realistic reach of the original query performance and certainly are not orders of magnitudes away. Indeed further to that it certainly seems with more testing and research that gap can be closed.

What was more disappointing was some of the errors that we uncovered due to the discord between our three value logic and the Postgres internals. Postgres has a notion of strict operators allowing it to short circuit evaluation where the input is a null value. In fact for certain optimisations to be taken advantage of we have seen that Postgres expects these operators to be *complete*, in particular equality. Since we have built our marked null values into the type there is the possibility to return null on (to Postgres eyes) non-null input.

We were able to create a work around for the restrict and join operator clauses, but for the B-tree index scan issue we were not. We decided to implement two versions of all the equality and inequality operators. We have boolean operators which always give definite answers to be used for ordering, indexing and hashing. However we also

kept the three value 'more correct' operators which we observed failed in some specific instances.

The failure we observed could be fixed by altering one of the C functions in the Postgres internals. However this is not a good solution since for the new types to become readily available we want to bundle them into a packaged extension which can be installed as a simple Postgres add on.

With further work it would almost certainly be possible to implement custom hashing and B-tree indexing methods which allow for operators returning null. This would allow us to keep the standard SQL evaluation procedure Eval_{SQL} while still benefiting from all possible performance optimisations. Thus we view this research as a major step towards a realistic ambition of efficient marked null implementation. Ultimately a goal where we have a syntax such as `SELECT CERTAIN` built into SQL (as suggested by [14]) allowing us to return only certain answers, possibly with a small performance hit, is achievable.

If more general results is always a good thing why stop at the level we have? For example we can represent the fact that two missing values are the same by marking them with same ID, but we cannot represent further relationships between two values. E.g. going back to our Person relation with name and age attributes if we had missing ages for Alison (NULL:1) and Hannah (NULL:2) but we knew that Hannah was Alison's mother and gave birth to her on her 25th birthday we would know there was a relationship between these two values (NULL:1 = NULL:25) so even in the marked null model we can lose information.

Such information could perhaps be represented by adding an extra 'clause' member to the underlying C struct. However we believe in reality the extra storage requirements and extra maintainability complexity would be too expensive to be useful except in very rare instances.

There are further questions to be asked in the marked null model also. We expect nulls to be input with an ID and suggest a mixing of marked nulls with SQL nulls to be at best bad practice and at worst disallowed entirely. This means that new incomplete information should be marked with a relevant ID. If we leave this to the user we leave open the possibility that they mark a null with an ID which already exists in the database and accidentally creating a relationship where one should not exist. We advocate that for such situations, unless an ID is specifically supplied with a null, the database should be responsible for generating a new previously unseen ID. This could easily be done by using integer IDs and maintaining a variable with current highest ID

and incrementing by one to generate new IDs.

Our closing remark is that although this project ultimately encountered some unexpected obstacles preventing a perfectly clean implementation we have demonstrated that not only are marked nulls achievable they generally, with some imperfections, perform well. Further to this when we combine them with correctness translations they continue to perform well. Therefore we expect to see in the near future a solution to SQL's blatantly wrong query results based on a marked null model.

Appendix A

Null Varchar Header file

```
/*
    ****************************************************************

 * Null_varchar type Based on varlen but need the extra boolean
     field
 * to check for marked null/constant
 ****************************************************************
     */

typedef struct Null_varchar
{
    char          nv_len[4];
    bool          cnst;
    char          nv_dat[FLEXIBLE_ARRAY_MEMBER];
}     Null_varchar;

typedef struct
{
    uint32        nva_header;
    bool          nva_const;
    char          nva_data[FLEXIBLE_ARRAY_MEMBER];
} nvarattrib_4b;

typedef struct
{
    uint8         nva_header;
    bool          nva_const;
    char          nva_data[FLEXIBLE_ARRAY_MEMBER];
} nvarattrib_1b;
```

```
#define NVARHDRSZ (VARHDRSZ + sizeof(bool))
#define NVARHDRSZ_SHORT                    offsetof(nvarattrib_1b,
    nva_data)
#define SET_NVARSIZE(NV_PTR, len) (((nvarattrib_4b *) (NV_PTR))->
    nva_header = (((uint32) (len)) << 2))
#define NVARDATA(NV_PTR) (NV_PTR)->nv_dat
#define NVARDATA_4B(NV_PTR)          (((nvarattrib_4b *) (NV_PTR))->
    nva_data)
#define NVARDATA_1B(NV_PTR)          (((nvarattrib_1b *) (NV_PTR))->
    nva_data)
#define NVARSIZE(NV_PTR) ((((nvarattrib_4b *) (NV_PTR))->nva_header
   >> 2) & 0x3FFFFFFF)
#define NVARCNST_1B(NV_PTR)          (((nvarattrib_1b *) (NV_PTR))->
    nva_const)
#define NVARCNST_4B(NV_PTR)          (((nvarattrib_4b *) (NV_PTR))->
    nva_const)


#define NVARATT_IS_1B(PTR) \
        ((((nvarattrib_1b *) (PTR))->nva_header & 0x01) == 0x01)

#define NVARATT_IS_SHORT(NV_PTR)
    NVARATT_IS_1B(NV_PTR)

#define NVARDATA_ANY(PTR) \
         (NVARATT_IS_1B(PTR) ? NVARDATA_1B(PTR) : NVARDATA_4B(PTR))

#define NVARCNST(PTR) \
        (NVARATT_IS_1B(PTR) ? NVARCNST_1B(PTR) : NVARCNST_4B(PTR))

#define NVARATT_IS_1B_E(PTR) \
        ((((nvarattrib_1b *) (PTR))->nva_header) == 0x01)

#define NVARSIZE_1B(PTR) \
        ((((nvarattrib_1b *) (PTR))->nva_header >> 1) & 0x7F)

#define NVARSIZE_4B(PTR) \
        ((((nvarattrib_4b *) (PTR))->nva_header >> 2) & 0x3FFFFFFF)
```

```
#define NVARSIZE_SHORT(PTR)
    NVARSIZE_1B(PTR)

#define NVARSIZE_ANY_EXHDR(PTR) \
          (NVARATT_IS_1B(PTR) ? NVARSIZE_1B(PTR)-NVARHDRSZ_SHORT : \
           NVARSIZE_4B(PTR)-NVARHDRSZ)


/*Toasting Macros */
#define NVARATT_IS_EXTERNAL(PTR)
    NVARATT_IS_1B_E(PTR)

#define NVARTAG_EXTERNAL(PTR)
    NVARTAG_1B_E(PTR)

static int null_varchar_cmp(Null_varchar *arg1, Null_varchar *arg2);
```

Appendix B

Original Untranslated Queries

```
--original untranslated SQL null Queries from:
--Making SQL Queries Correct on Incomplete Databases: A Feasibility
    Study
--Guagliardo, Paolo and Libkin, Leonid and others

--No changes required to these queries to run marked null database

--Q1 (adjusted tpch query 1)

SELECT DISTINCT
        s_suppkey, o_orderkey
    FROM
        supplier,
        lineitem l1,
        orders,
        nation
    WHERE
        s_suppkey = l1.l_suppkey
        AND o_orderkey = l1.l_orderkey
        AND o_orderstatus = 'F'
        AND l1.l_receiptdate > l1.l_commitdate
        AND EXISTS (
            SELECT
                *
            FROM
                lineitem l2
            WHERE
                l2.l_orderkey = l1.l_orderkey
                AND l2.l_suppkey <> l1.l_suppkey
```

```
        )
        AND NOT EXISTS (
            SELECT
                *
            FROM
                lineitem l3
            WHERE
                l3.l_orderkey = l1.l_orderkey
                AND l3.l_suppkey <> l1.l_suppkey
                AND l3.l_receiptdate > l3.l_commitdate
        )
        AND s_nationkey = n_nationkey
        AND n_name = '$nation';

--parameter $nation replaced by a random nation when generating
    queries


--Q2 (adjusted tpch query 2)

SELECT
        c_custkey,
        c_nationkey
    FROM
        customer
    WHERE
        c_nationkey IN ($countries)
        AND c_acctbal > (
            SELECT
                avg(c_acctbal)
            FROM
                customer
            WHERE
                c_acctbal > 0.00
                AND c_nationkey IN ($countries)
        )
        AND NOT EXISTS (
            SELECT *
            FROM   orders
            WHERE  o_custkey = c_custkey
        );
```

```
--parameter $countries replaced by a random nation key when
    generating queries


--Q3 (textbook query 1)

SELECT o_orderkey
    FROM   orders
    WHERE  NOT EXISTS (
           SELECT *
           FROM   lineitem
           WHERE  l_orderkey = o_orderkey
           AND    l_suppkey <> $supp_key );

--parameter $supp_key replaced by a random supplier key when
    generating queries


--Q4 (textbook query 2)

SELECT o_orderkey
    FROM   orders
    WHERE  NOT EXISTS
        (  SELECT *
           FROM   lineitem, part, supplier, nation
           WHERE  l_orderkey = o_orderkey
           AND    l_partkey = p_partkey
           AND    l_suppkey = s_suppkey
           AND    p_name LIKE '%$color%'
           AND    s_nationkey = n_nationkey
           AND    n_name = '$nation' );


--parameter $nation replaced by a random nation when generating
    queries
--parameter $color replaced by a random colour on query generation
```

Appendix C

Translated Queries

```
--best translated queries from:
--Making SQL Queries Correct on Incomplete Databases: A Feasibility
    Study
--Guagliardo, Paolo and Libkin, Leonid and others

--Note that here we have replaced IS NULL with @@
--and IS NOT NULL with @! on marked null columns

--Q1 (marked null version of translated tpch query 1)

SELECT DISTINCT
        s_suppkey, o_orderkey
    FROM
        supplier,
        lineitem l1,
        orders,
        nation
    WHERE
        s_suppkey = l1.l_suppkey
        AND o_orderkey = l1.l_orderkey
        AND o_orderstatus = 'F'
        AND l1.l_receiptdate > l1.l_commitdate
        AND l1.l_receiptdate IS NOT NULL
        AND l1.l_commitdate IS NOT NULL
        AND EXISTS (
            SELECT
                *
            FROM
                lineitem l2
```

```
            WHERE
                l2.l_orderkey = l1.l_orderkey
                AND l2.l_suppkey <> l1.l_suppkey
                AND l2.l_suppkey @!
                AND l1.l_suppkey @!
        )
        AND NOT EXISTS (
            SELECT
                *
            FROM
                lineitem l3
            WHERE
                l3.l_orderkey = l1.l_orderkey
                AND ( l3.l_suppkey <> l1.l_suppkey
                      OR l3.l_suppkey @@
                      OR l1.l_suppkey @@ )
                AND ( l3.l_receiptdate > l3.l_commitdate
                      OR l3.l_receiptdate IS NULL
                      OR l3.l_commitdate IS NULL )
        )
        AND s_nationkey = n_nationkey
        AND n_name = '$nation';

--parameter $nation replaced by a random nation when generating
    queries


--Q2 (marked null version of translated tpch query 2)

SELECT
        c_custkey,
        c_nationkey
    FROM
        customer
    WHERE
        c_nationkey IN ($countries)
        AND c_acctbal > (
            SELECT
                avg(c_acctbal)
            FROM
                customer
            WHERE
```

```
                c_acctbal > 0.00
                AND c_nationkey IN ($countries)
        )
        AND NOT EXISTS (
            SELECT *
            FROM   orders
            WHERE  o_custkey = c_custkey
        )
        AND NOT EXISTS (
            SELECT *
            FROM   orders
            WHERE  o_custkey @@
        )

--parameter $countries replaced by a random nation key when
    generating queries


--Q3 (marked null version of translated textbook query 1)

SELECT o_orderkey
    FROM   orders
    WHERE  NOT EXISTS (
            SELECT *
            FROM   lineitem
            WHERE  l_orderkey = o_orderkey
            AND    ( l_suppkey <> $supp_key OR l_suppkey @@) )

--parameter $supp_key replaced by a random supplier key when
    generating queries


--Q4 (marked null version of translated textbook query 2)

WITH
    part_view AS (
        SELECT p_partkey
        FROM   part
        WHERE  p_name @@
        UNION
        SELECT p_partkey
        FROM   part
```

```
        WHERE  p_name LIKE '%$color%' ),
    supp_view AS (
        SELECT s_suppkey
        FROM   supplier
        WHERE  s_nationkey @@
        UNION
        SELECT s_suppkey
        FROM   supplier, nation
        WHERE  s_nationkey = n_nationkey
        AND    n_name = '$nation' )
    SELECT o_orderkey
    FROM   orders
    WHERE  NOT EXISTS (
           SELECT *
           FROM   lineitem, part_view, supp_view
           WHERE  l_orderkey = o_orderkey
           AND    l_partkey = p_partkey
           AND    l_suppkey = s_suppkey )
    AND    NOT EXISTS (
           SELECT *
           FROM   lineitem, supp_view
           WHERE  l_orderkey = o_orderkey
           AND    l_partkey @@
           AND    l_suppkey = s_suppkey
           AND    EXISTS ( SELECT * FROM part_view ) )
    AND    NOT EXISTS (
           SELECT *
           FROM   lineitem, part_view
           WHERE  l_orderkey = o_orderkey
           AND    l_partkey = p_partkey
           AND    l_suppkey @@
           AND    EXISTS ( SELECT * FROM supp_view ) )
    AND    NOT EXISTS (
           SELECT *
           FROM   lineitem
           WHERE  l_orderkey = o_orderkey
           AND    l_partkey @@
           AND    l_suppkey @@
           AND    EXISTS ( SELECT * FROM part_view )
           AND    EXISTS ( SELECT * FROM supp_view ) )

--parameter $nation replaced by a random nation when generating
```

```
    queries
--parameter $color replaced by a random colour on query generation
```

Appendix D

Source Code

A digital copy of the full source code has been submitted along with this dissertation. However as an extra reference it is stored as a git version control repository here: `https://bitbucket.org/ps288/marked_null/`.

The final version of this code used for this dissertation was committed on August 18th 2016.

Bibliography

[1] (2014). Tpc benchmark h, standard specification.

[2] (2016). Postgres source code. http://doxygen.postgresql.org/postgres_8h_source.html. Accessed: 2016-08-12.

[3] Abiteboul, S., Hull, R., and Vianu, V. (1995). *Foundations of databases: the logical level*. Addison-Wesley Longman Publishing Co., Inc.

[4] Abiteboul, S., Kanellakis, P., and Grahne, G. (1987). *On the representation and querying of sets of possible worlds*, volume 16. ACM.

[5] Codd, E. F. (1990). *The relational model for database management: version 2*. Addison-Wesley Longman Publishing Co., Inc.

[6] Committee, I. C. (2011). 6.7.2.1 structure and union specifiers. http://c0x.coding-guidelines.com/6.7.2.1.html. Accessed: August 9, 2016.

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

[8] Daye, C. J. (2003). *An Introduction to Database Systems*. Pearson.

[9] Developers, P. (2016 (accessed August 1, 2016)a). *PostgreSQL 9.5.3 CREATE OPERATOR*. https://www.postgresql.org/docs/9.6/static/sql-createoperator.html.

[10] Developers, P. (2016 (accessed August 1, 2016)b). *PostgreSQL 9.5.3 CREATE TYPE*. https://www.postgresql.org/docs/9.5/static/sql-createtype.html.

[11] Developers, P. (2016 (accessed August 1, 2016)c). *PostgreSQL 9.5.3 Documentation*. http://www.postgresql.org/docs/9.5/static/index.html.

[12] Developers, P. (2016 (accessed August 1, 2016)d). *PostgreSQL 9.5.3 Operator Optimization Information*. `http://www.postgresql.org/docs/9.5/static/xoper-optimization.html`.

[13] Developers, P. (2016 (accessed August 1, 2016)e). *PostgreSQL 9.5.3 Packaging Related Objects into an Extension*. `https://www.postgresql.org/docs/9.4/static/extend-extensions.html`.

[14] Guagliardo, P., Libkin, L., et al. (2016). Making sql queries correct on incomplete databases: A feasibility study.

[15] Haas, L. M., Hernández, M. A., Ho, H., Popa, L., and Roth, M. (2005). Clio grows up: from research prototype to industrial tool. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 805–810. ACM.

[16] Imieliński, T. and Lipski Jr, W. (1984). Incomplete information in relational databases. *Journal of the ACM (JACM)*, 31(4):761–791.

[17] Jenkins, B. (1997 (accessed August 1, 2016)). Burtleburtle hashing. `http://burtleburtle.net/bob/hash/doobs.html`.

[18] Libkin, L. (2016). Sqls three-valued logic and certain answers. *ACM Transactions on Database Systems (TODS)*, 41(1):1.

[19] Marnette, B., Mecca, G., Papotti, P., Raunich, S., Santoro, D., et al. (2011). ++ spicy: an open-source tool for second-generation schema mapping and data exchange. *Clio*, 19:21.

[20] Seacord, R. C. (2014). *The CERT C Coding Standard: 98 Rules for Developing Safe, Reliable, and Secure Systems*. Pearson Education.

[21] TPC (2016 (accessed August 15, 2016)). Tpc=h decision support benchmark. `http://www.tpc.org/tpch/`.

[22] van der Meyden, R. (1998). Logical approaches to incomplete information: A survey. In *Logics for databases and information systems*, pages 307–356. Springer.