Adding HL7 version 3 data types to PostgreSQL

Yeb Havinga
MGRID
Oostenburgervoorstr. 106-114
1018MR Amsterdam
y.t.havinga@mgrid.net

Willem Dijkstra
MGRID
Oostenburgervoorstr. 106-114
1018MR Amsterdam
w.p.dijkstra@mgrid.net

Ander de Keijzer
University of Twente
Institute of Technical Medicine
7500AE Enschede
a.dekeijzer@utwente.nl

ABSTRACT

The HL7 standard is widely used to exchange medical information electronically. As a part of the standard, HL7 defines scalar communication data types like physical quantity, point in time and concept descriptor but also complex types such as interval types, collection types and probabilistic types. Typical HL7 applications will store their communications in a database, resulting in a translation from HL7 concepts and types into database types. Since the data types were not designed to be implemented in a relational database server, this transition is cumbersome and fraught with programmer error. The purpose of this paper is two fold. First we analyze the HL7 version 3 data type definitions and define a number of conditions that must be met, for the data type to be suitable for implementation in a relational database. As a result of this analysis we describe a number of possible improvements in the HL7 specification. Second we describe an implementation in the PostgreSQL database server and show that the database server can effectively execute scientific calculations with units of measure, supports a large number of operations on time points and intervals, and can perform operations that are akin to a medical terminology server. Experiments on synthetic data show that the user defined types perform better than an implementation that uses only standard data types from the database server.

1. INTRODUCTION

The HL7 version 3 standard specifies how to exchange medical information electronically. The original intent of the HL7 standard is to provide a framework for designing medical messaging applications, though there is an increasing number of initiatives that apply HL7 methodology and structure to their entire medical information system. This has been noticed by the HL7 organization and in 2008 the RIM[1] Based Application Architecture (RIMBAA) working group has been founded[8].

Part of the HL7 specification is the data type specification. Since differences in data type implementations between programming environments are common, HL7 makes no assumptions about available data types, but defines it's own instead[6]. The definitions of the individual types are structured in an object oriented fashion; generic types are extended by more specific types, with all types inheriting from a top abstract type. The top type has a notion of a flavor of null; this allows HL7 developers to provide information about a value when it is null. HL7 defined data types include scalar types like integer, string, point in time, but also complex types and templated types such as interval types, collection types and types expressing probabilities. The HL7 data types in its second revision called R2 has become an ISO standard in 2008[1].

[1]RIM is the HL7 version 3 *Reference Information Model*.

Example data type

Data type `Boolean` specializes top abstract type `ANY`; it inherits `ANY`'s ability to express a nullflavor and extends that with the boolean value domain. This is defined in the following fashion:

```
type Boolean alias BL specializes ANY
    values(true, false) {
        BL and(BL x);
        BL not;
        literal ST.SIMPLE;
        BL or(BL x);
        BL xor(BL x);
        BL implies(BL x);
};
```

Assertions are then made on how the particulars of this data type should be interpreted, such as the definition $x.not$ as negation of x:

```
invariant(BL x) {
    true.not.equal(false);
    false.not.equal(true);
    x.isNull.equal(x.not.isNull);
};
```

The mapping of the HL7 data types can be done in a number of ways. PostgreSQL pioneered the following approach: the definition of User-Defined Types that uniquely encapsulate the HL7 data types, allowing full access to the types and methods on those types from SQL[10] [9] [2].

HL7 data types were not designed to be implemented in a relational database, and it is no surprise that some facets of the data types are a challenge to match to relational database theory:

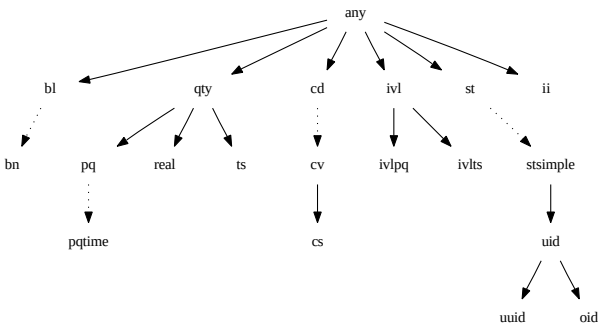

Figure 1: *Part of the HL7 data type hierarchy*

- The hierarchy of types; this implies that table columns of a general type must be able to hold instances of its subtypes. See figure 1 for a subset of the hierarchy.
- Not all data types have a string literal format; making it hard to work out what the domain of those types are.
- The top type in the hierarchy has the ability to express that it is null in a particular flavor. For instance: a data value can be null with flavor `asked but unknown`.

We investigate to what extent the HL7 data type definitions can be matched with relational database theory, define a set of conditions for a data type that must be satisfied before it can be implemented and implement a subset of the data types in PostgreSQL.

2. PRELIMINARIES

In this section we review relevant portions of relational database theory, as well as the parts of the specification of HL7 data types that are relevant to this paper.

2.1 HL7 Data type definitions

HL7 Data types are defined in a hierarchy, with the data type `any` as most general data type. Instances of data types are called *data values*. The hierarchy of types that are discussed in this paper is shown in figure 1.

For every data type *properties* are defined. A property is referred to by its name. The *domain* of a property is the set of values a property can have. Properties may have arguments. For instance, the `plus` property of one integer argument requires another integer as argument.

Any. The data type `any` is the most generic data type. All other data types extend `any` and inherit functions defined on `any`. `any` is abstract, and is only used to define a function interface and common properties.

The HL7 notion of NULLs, called *nullflavors*, are defined on `any`. Nullflavors are specified to extend the domain of the data type, allowing nullflavor symbols in all HL7 data type literals. The nullflavors are defined in a hierarchy, which coincides in meaning with the ordering on degree of knowledge described in [11]. The most general nullflavor is no information, or `ni`, all other nullflavors add meaning to that. Table 1 shows all 15 and their hierarchy. The level code in the first column encodes the position in the hierarchy; the nullflavor in each row is subsumed by the closest nullflavor above it with a lower number. For instance, trace / `trc` is subsumed by unknown / `unk` but not by invalid / `inv`.

The `any` data type has several properties that are true if the data value is of a certain nullflavor, like `nonnull`, `isnull` and `unknown`. Also the properties `equal` and `identical` are defined to take another data value as argument and have `bl` as domain.

Boolean (BL). The data type `bl` has as domain the truth values *true* and *false* with literal form `true` and `false`. It inherits from `any`, which means that (1) its literal form is extended with the nullflavors. The nullflavors `ninf`, `pinf`, `unc`, `der`, `qs` and `trc` are not allowed on the boolean data type. (2) `bl` inherits all properties of `any` such as the boolean null properties like `isnull` from `any`. (3) It must also provide an implementation for the `equal` property. The equality of two operands is true when both operands are non-null and have the same value. It adds the following properties `not`, `and`, `or`, `xor`, `equal` and `implies`, as expected with their usual meaning. When one or two of the operands have a nullflavor, the operators behave like Codds three valued semantics[4]. Since the number of nullflavors allowed on `bl` is 9, there are 11^2 possible combinations. According to the HL7 data type specification, when operands are two different nullflavors, the result is the nullflavor 'that is the first common ancestor of the two nullflavors'. The ancestor relation is the usual, except that in this context it is also reflexive; if both operands are the same nullflavor, the result is the same nullflavor. This results in a 11 valued logic as is shown in tables 2 and 3. (4) Finally, since `bl` is a proper subtype of `any`, table attributes of type `any` can hold instances of type `bl`.

Boolean NonNull (BN). The data type `bn` is a constrained version of the `bl` data type so that it is not null. Constrained versions of types are called *flavors* in the HL7 data type specification. In figure 1 flavors can be distinguished from normal specialized types by the dotted arrows.

Quantity (QTY). The data type `qty` specializes `any` and like `any`, it is an abstract type. No data values can be just of type `qty` without belonging to a proper subtype of `qty`. `qty` subsumes ordered data types and hence introduces the properties `lessorequal`, `lessthan`, `greaterthan`, `greaterorequal`. Also it adds the addition and subtraction operators `plus` and `minus` on operands of the same data type.

Real Number (REAL). The real number is a scalar magnitude, which is used whenever quantities are measured, estimated, or computed from other real numbers. The literal form is decimal, where the number of significant decimal digits is known as the precision. Conforming implementations are not required to be able to represent the full range of real numbers. The standard declares the representations of the real value space as floating point, rational, scaled integer, or digit string, and their various limitations to be out of the scope of the specification.

Physical Quantity (PQ). One of the most practical HL7 types is `pq`, which is a complex data type consisting of a unit and a value property, where the unit is a string constant that conforms to the *Unified Code for Units of Measure* or UCUM specification [7], and the value is the quantity. Example

Table 1: *HL7 NullFlavors*

Level	Symbol	Meaning	Description
1	**ni**	no information	Default and most general exceptional value
2	**inv**	invalid	Value not permitted in constrained domain
3	**oth**	other	Actual value not permitted in constrained domain
4	**ninf**	negative infinity	Negative infinity on numbers
4	**pinf**	positive infinity	Positive infinity on numbers
3	**unc**	unencoded	Information not encoded (yet)
3	**der**	derived	Actual value must be derived
2	**unk**	unknown	Proper value is applicable but not known
3	**asku**	asked but unknown	Information sought but not found
4	**nav**	temporarily unavailable	Expected to be available later
3	**qs**	sufficient quantity	Quantity not known but non-zero
3	**nask**	not asked	Information not sought
3	**trc**	trace	Greater than zero but too small for quantification
2	**msk**	masked	Information available but not provided
2	**na**	not applicable	No value applicable in context

Table 2: *AND truth table*

AND	asku	false	inv	msk	na	nask	nav	ni	oth	true	unk
asku	*asku*	false	ni	ni	ni	unk	asku	ni	ni	asku	unk
false	false	*false*	false	false	false	false	false	false	false	false	false
inv	ni	false	*inv*	ni	ni	ni	ni	ni	inv	inv	ni
msk	ni	false	ni	*msk*	ni	ni	ni	ni	ni	msk	ni
na	ni	false	ni	ni	*na*	ni	ni	ni	ni	na	ni
nask	unk	false	ni	ni	ni	*nask*	unk	ni	ni	nask	unk
nav	asku	false	ni	ni	ni	unk	*nav*	ni	ni	nav	unk
ni	ni	false	ni	ni	ni	ni	ni	*ni*	ni	ni	ni
oth	ni	false	inv	ni	ni	ni	ni	ni	*oth*	oth	ni
true	asku	false	inv	msk	na	nask	nav	ni	oth	*true*	unk
unk	unk	false	ni	ni	ni	unk	unk	ni	ni	unk	*unk*

Table 3: *OR truth table*

OR	asku	false	inv	msk	na	nask	nav	ni	oth	true	unk
asku	*asku*	asku	ni	ni	ni	unk	asku	ni	ni	true	unk
false	asku	*false*	inv	msk	na	nask	nav	ni	oth	true	unk
inv	ni	inv	*inv*	ni	ni	ni	ni	ni	inv	true	ni
msk	ni	msk	ni	*msk*	ni	ni	ni	ni	ni	true	ni
na	ni	na	ni	ni	*na*	ni	ni	ni	ni	true	ni
nask	unk	nask	ni	ni	ni	*nask*	unk	ni	ni	true	unk
nav	asku	nav	ni	ni	ni	unk	*nav*	ni	ni	true	unk
ni	ni	ni	ni	ni	ni	ni	ni	*ni*	ni	true	ni
oth	ni	oth	inv	ni	ni	ni	ni	ni	*oth*	true	ni
true	true	true	true	true	true	true	true	true	true	*true*	true
unk	unk	unk	ni	ni	ni	unk	unk	ni	ni	true	*unk*

Table 4: *Interval forms for TS*

Name	Example
Interval form	[20080101131251;20080131155629]
Comparator form	<20080101
Centerwidth form	20010115135108 [10s]
Width form	[10d]
Center form	20010101
Any form	?200101?
Hull form	20010101..20010131

units are m for meter, J for Joule, mm[Hg] for pressure. The UCUM specification consists of a few hundreds units, and defines 7 base units which are m for meter, g for gram, s for second, rad for radian, K for Kelvin, C for Coulomb and cd for candela. Every other unit can be expressed in terms of these base units together with metric prefixes like k for kilo and m for milli. For instance, mm[Hg] can be expressed as $m^{-1}.g.s^{-2}$. Example pqs are 10 ml and 0.5 kg/m2.

The *canonical value* property of a pq x is a pq y such that $x = y$ and the unit of y consists only of base units. Two pqs *compare* if their canonical values have the same base units. For instance, ml and dm3 compare, but mm (millimeter) and m3 (a cubic meter) do not.

Two units are *equal* if the values of their canonical values are equal. Two units are *identical* if both the units and values are the same. For instance, 1 m and 100 cm are equal, but are not identical.

Point in time (TS). ts is a specialization of qty that defines a point on the axis of natural time. Since there is no absolute zero-point on the time-axis a ts is measured in the amount of time that has elapsed since some arbitrary zero-point called the *epoch*. The current standard supports only the Gregorian calendar, with ts measurements in quantities comparable to any pq with base unit s. Every ts has a precision that specifies the number of significant digits in the calendar expression. The calendar expression can hold year, month, day, hour, minute, second, fractional seconds and timezone.

Interval (IVL<T>). The interval definition is a template that turns any quantity type into an interval type of that quantity. Two specific intervals are discussed below.

Interval of point in time (IVL<TS>). The ivl<ts> interval form has seven different literal forms, listed in table 4. While few forms define a start and end of an interval, other define only a single point in time or only a period of time without specific starting point. Brackets in the interval, centerwidth and width forms show if the interval is inclusive or exclusive of the boundary. An example: the interval between the first of January, 13:12:51 and the 31st of January, time 15:56:29 in 2008 is [20080101131251;20080131155629]. An entire calendar month must be denoted using an open high boundary, like so: [20010101;20010201[.

The promotion of ts to ivl<ts> uses the precision of the source ts to define the width of the new ivl<ts>. An example: the promotion of ts 20010131 yields an ivl<ts> of [20010131000000;20010201000000[.

Interval of physical quantities (IVL<PQ>). ivl<pq> also has 7 different literal forms. It lacks a hull form, but it gains a dash form like so: 3ml - 5ml which is equivalent to [3ml;5ml].

Concept Descriptor (CD). Controlled vocabularies play an important role in medical informatics. In the HL7 data type specification, the concept descriptor data type provides terminological functionality.

A cd is a reference to a concept defined in a code system, such as LOINC, ICD or SNOMED-CT. Properties of cd include a code, an original text that served as the basis of the coding, and zero or more translations of the concept into multiple code systems. The code can be an atomic code, an elementary concept directly defined by the reference code system, for instance C41.9 for the concept 'Bone and articular cartilage, unspecified' in ICD-10. A code can also be an expression in some syntax defined by the referenced code system to facilitate post-coordination. Post-coordination is the usage of a set of codes to explain a single concept that does not have a code in the vocabulary yet. SNOMED-CT is an example of a code system that supports post-coordination. Many code systems have an explicit notion of concept specialization and generalization. The implies property on cd takes another cd as argument and returns true iff the argument is a generalization of the data value.

Coded Value (CV). A cv is a flavor of cd , such that there are no translations and only a single concept is allowed. cv is used when only a single code value is required.

Coded Simple Value (CS). A cs is a specialization of cv. It is coded data in its simplest form, where everything but the code itself is predetermined by the context in which the cs data value occurs. Hence the literal form of a cs consists of only the code itself.

Instance Identifier (II). An instance identifier uniquely identifies a thing or object. Properties of ii include the root and the extension. The root is character string that is either an DCE Universal Unique Identifier (UUID) or an ISO/IEC 8824:1990 Object Identifier and is a unique identifier that guarantees the global uniqueness of the ii data value. The extension is a character string that is unique in the namespace of the root. The extension may be null. The literal form of ii is defined to be the root, followed by a colon, followed by the extension.

2.2 Database definitions

The database definitions we need are the usual, with two exceptions. First, we need the distinction between the set of constants and the object domain. Though this distinction is rarely made in relational database literature, we need it in the discussion of semantics of nullflavors, as well as in section 3.1 where a number of conditions are described that data types must satisfy, in order to be suitable for implementation in a relational database. Also, since this paper is concerned with data types only, we do not formally define relation schemes, instances and attributes, but only a set of types.

dom is a countably infinite set of individual objects. **con** is a countably infinite set of constant symbols. Let **typ** denote the set of all data types. *Dom* is a mapping from **typ** to **dom**. *Con* is a mapping from **typ** to **con**. If A is a

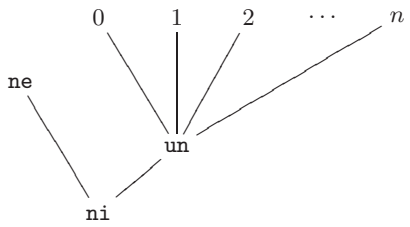

Figure 2: *Degree of knowledge ordering*

data type, $Dom(A)$ is called the *domain of* A and $Con(A)$ is called the *literal form of* A. Constants describe objects. The meaning $[\![a]\!] \subseteq$ **dom** of a constant a is a mapping from **con** to **dom**, and is the set of all objects described by a. a is a better description than a', if $[\![a]\!] \subset [\![a']\!]$. The *unique names assumption* means that no two different constant symbols denote the same object.

2.3 Partial information

The concept of a null value in relational databases has been widely discussed in database literature because of its problematic nature. One of the problems is the interpretation of a null value; common meanings found in database literature are *unknown*, a value is applicable and exists but is at present not known, and *nonexistent*, a value does not exist. Zaniolo[12] proposes a third interpretation called *no information* and shows that a sound relational algebra can be constructed with this interpretation, though meaning is lost due to the more general nature of *no information*. In [4] a three valued semantics for boolean operations with unknown values was given. [11] describes problems that occur in many valued semantics and introduces ordering partial information with increasing degree of knowledge. This idea is expanded by others, for a complete reference we refer to [5].

What is the meaning of a null value with respect to domains? [12] extends each domain to include the distinguished symbol `ni`. [3] proposes an ordering on values of domains based on the number of real-world objects a described by a symbol, such as a null value. For example, for the domain of natural numbers and the null values `ni`, `un` and `ne`, meaning no information, unknown and nonexistent, respectively, the ordering is shown in figure 2. This picture shows a number of things. First of all it captures the idea that the *unknown* null value means that any of the natural numbers could be the actual value. In other words, one of the naturals is equal to *unknown* on this data type. This does not hold for `ne`, which does not match any natural number. `ni` is the most general null value. The actual value is not known and it is even not known whether an actual value exists. `ne` as well as every natural number represent *perfect knowledge*, in the sense that the information can not be improved. These correspond to the the leafs of the tree.

3. ANALYSIS

In this section we review the HL7 data type definitions from the standpoint of relational database theory. Our goal is to match HL7 data type definitions to relational database concepts. We discuss relevant portions of some of the implemented constructs and data types.

The concept of data type *properties* was described in section 2.1. Though the HL7 standard does not specialize properties, we make the distinction between two different kinds:

1. *compositional properties* are the intrinsic components of a data type. For instance, `value` is a compositional property of the data type `pq`.

2. *relational properties* define relations on data values of certain types. For instance, the `equal` relational property of the `pq` data type takes as argument another `pq` and thereby defines a relation on the powerdomain of `pq` data values.

A data type property is either a compositional property or a relational property, but never both. This distinction is needed in the next section.

3.1 Conditions

Let A be a data type. The following conditions on A must be satisfied to qualify for implementation in a relational database:

1. $Con(A)$ must be defined. This means that the literal form of the data type must be provided or be trivial. For instance, the unit constants from `pq` have a BNF style grammar definition, which qualifies as an intensional definition of $Con($`pq`$)$.

2. $Dom(A)$ should be well defined, which means that is should be clear what its elements are. There should be a one-to-one correspondence between an object and its compositional properties. In other words, an object is completely defined by its compositional properties.

3. The *compositional properties* of A must be binary representable in the C language. This seemly arbitrary condition makes explicit that a translation from the recursive definitions of types, that are convenient and ubiquitous in the HL7 data types specification, must be translated to an implementable form.

4. The *relational properties* of A must be implementable as a User-Defined Function in the relational database.

5. Meaning of data values may not be lost or altered when the context in which it is used is changed. Specifically, its semantics may not change after the application of any number of relational operators.

3.2 Nullflavors

Since the nullflavors are extensions to each data type, the conditions of section 3.1 also apply here.

The Nullflavors extend the literal from of the underlying data type, i.e. for an arbitrary data type A, $Con(A)$ is extended with the nullflavor constant symbols, which are defined by extension.

Regarding conditions on $Dom(A)$, it is clear that Nullflavors serve the same purpose of NULLs known from database literature, and it is likely that HL7s nullflavor definitions originate from database literature. For instance, the NULL in SQL:2003 has as meaning *value at present unknown*, which coincides with the nullflavor `unk`. Also, the most general nullflavor `ni` (no information) has the same name and meaning as the most general null described by [3]. We believe the methodology developed in [3] can be applied to the the nullflavors as well, for each data type. The nullflavors extend the literal form of the underlying data type, so the nullflavor symbols are added to the set

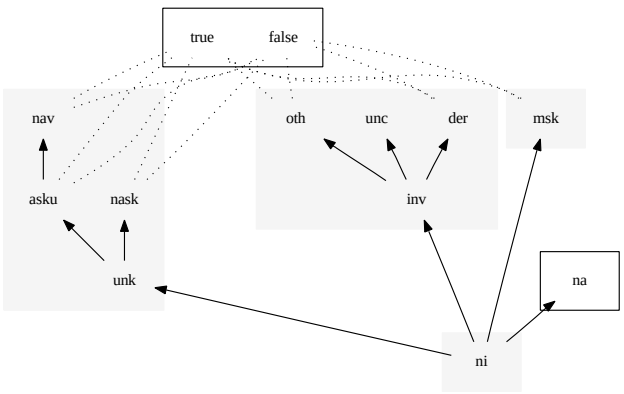

Figure 3: *Degree of knowledge ordering with nullflavors in boolean domain*

of constants **con**. The most general nullflavor `ni` can be any object, no information is known. In the case of all 9 nullflavors allowed on `bl`, except `ni` and `na`, the knowledge can be improved to the actual value. For instance, the knowledge of `nav` can be improved to be either true or false. Hence $[\![\texttt{nav}]\!] = \{\text{true, false}\}$. The nullflavor `na` can never be improved, it is already perfect knowledge. Like [12] we extend the underlying domain $Dom(\texttt{bl})$ with the object described by $[\![\texttt{na}]\!]$. I.e. $\text{Dom}(\texttt{bl}) = \{\text{true}, \text{false}, \texttt{na}\}$. The nullflavor `ni` gives no information at all, hence $[\![\texttt{ni}]\!] = Dom(\texttt{bl})$. The degree of knowledge expressed by nullflavors defines an ordering on the descriptions of the elements of $Con(\texttt{bl})$. The description $[\![\texttt{unk}]\!] = \{\text{true}, \text{false}\}$ is a better description than $[\![\texttt{ni}]\!]$, since $[\![\texttt{unk}]\!] \subset [\![\texttt{ni}]\!]$. True, false and `na` are perfect descriptions, since the knowledge they describe cannot be improved. This is shown in figure 3.

A result of this analysis is that `na` can never be improved to be either *true* or *false*. This has consequences for the 11-valued BL logic in cases where `na` is involved. For instance, according to HL7, the conjunction of *true* and `na` is `na`. But it might as well be *false*, since `na` cannot be improved to *true*. The same holds for disjunction. As an example, consider the following example with possible situations for the propositions 'Mary is pregnant' (m) and 'Bob is pregnant' (b). Tables 5 and 6 show the behaviour of boolean operators involving `na` as defined by HL7 and an alternative behaviour, with the differences typeset italic. The tables also show differences for logical combinations of `na` and other nullflavors. The rationale behind the differences for conjunction is that since a conjunction is true iff both conjuncts are true, the conjunction is always false, since true is not an element of $[\![\texttt{na}]\!]$. For the disjunction of a nullflavor and `na`, the question is what answer gives the most information we know about the result. Since the result can only 'be made true' by the disjunct that is not `na`, if it is true, it is because of the not-`na` disjunct. Hence it makes sense to propagate the information this disjunct carries in its nullflavor to the result of the disjunction. For instance, suppose that the value of the proposition 'Mary is pregnant' is masked, nullflavor `msk`. Under the alternative interpretation of $\vee$, the disjunction 'Mary is pregnant' or 'Bob is pregnant' is `msk`.

Regarding the binary representation of the nullflavors *compositional properties*, it is easy to enumerate the 15 kinds into e.g. an integer or nibble. The question however is in which structure in the database server to store this integer. The

Table 5: *HL7* `na` *behaviour*

m	b	$m \wedge b$	$m \vee b$
true	na	na	true
false	na	false	na
asku	na	ni	ni
inv	na	ni	ni
msk	na	ni	ni
na	na	na	na
nask	na	ni	ni
nav	na	ni	ni
ni	na	ni	ni
oth	na	ni	ni
unk	na	ni	ni

Table 6: *Altered* `na` *behaviour*

m	b	$m \wedge b$	$m \vee b$
true	na	*false*	true
false	na	false	*false*
asku	na	*false*	*asku*
inv	na	*false*	*inv*
msk	na	*false*	*msk*
na	na	*false*	*false*
nask	na	*false*	*nask*
nav	na	*false*	*nav*
ni	na	*false*	*ni*
oth	na	*false*	*oth*
unk	na	*false*	*unk*

extensible type system of the PostgreSQL relational database was not meant to support more kinds of NULLs. Since it is open source, we considered a number of alternatives, for instance a complete replacement of the boolean `isnull` in source code and the isnull bitmap in heaptuple structure in memory and on disk by a set of nibbles. We did not choose this option mainly because it would break compliance with SQL:2003 on standard NULL handling. That meant that any implementation would be a hybrid solution of database NULLs and HL7 NullFlavors. The question was to what extent there should be an interaction between database NULLs and HL7 NullFlavors. For instance, should `NOT NULL` constraints on table columns mean that the data values may not contain a nullflavor? And if a record contains a database NULL value, should the HL7 `isnull` relational property return true? We chose to implement no interaction between database NULLs and NullFlavors at all. As a consequence, any kind of nonnull constraints should be provided by the HL7 data type implementation itself. In the case of the `bn` data type, HL7 shows it is able to provide such constraints. In the case of the `ii` data type however, we believe that a nonnull flavor would be appropriate for use in a relational database. See section 3.8 for a discussion. The nibble representing the nullflavor or 0 for nonnull, is added to the binary representation of every implemented data type.

The *relational properties* like `isnull`, `nonnull` are trivial to implement with user defined functions.

The last condition that meaning of data values do not change under the application of relational operators, poses no problems for most nullflavors, but there might be a problem with the `oth` nullflavor. If a data value has the nullflavor

oth this means that the actual value is not a member of the set of permitted data values in the constrained value domain of a variable. This may occur when the value exceeds some constraints that are defined in too restrictive a manner. For example, if the value for age is a 100 years, but the constraining model specifies that the age must be less than 100 years, the age may still be given, provided that the model does not make the attribute mandatory.

One problem is the literal form of this value is not defined, though we could choose, for instance, to write `100 yr NullFlavor.OTH` [2]. Second, what is the status of this object in Dom(pq)? By the definition of the *equal* property, it is equal to `100 yr`. But this is in conflict with the unique names assumption. Also, depending on the way the constraint is specified, the meaning of the nullflavor is altered under projection. If the constraint the data value does not satisfy is defined on a relation attribute with a check constraint like `CHECK (age < '100 yr'::pq)`, it is known that oth means that the value violates this constraint. But projection of the relation attribute 'loses' the check constraint, and hence the meaning of the data value is altered. We solved these issues by having data values with the other nullflavor not compare with any other data value, so they do not violate any check constraint except being nonnull or not the other nullflavor.

3.3 Type hierarchy

The internal representation of each subtype is binary compatible with its direct super type on all components defined by the super type. Let A, B be data types. If A and B are different data types, a *cast* is a function of type $A \to B$. For each two data types A, B where A is a specialization of B, we implemented two casts $A \to B$ and $B \to A$ that allow the data values to be converted.

This allows the creation of a table with an attribute of type any , that can then hold data values of proper specializations of any such as pq .

```
CREATE TABLE testany (a hl7.any);
INSERT INTO testany VALUES ('10 ml'::pq);
SELECT * FROM testany;
   a
-------
 10 ml
(1 row)
```

3.4 Booleans

Both boolean data types bl and bn satisfy all conditions of section 3.1 and the implementation was trivial aside from implementing the nullflavors. One important aspect is that besides the casts from bl and bn to any and vice versa for the type hierarchy, we also implemented casts from and to PostgreSQL's native data type boolean. The bulk of the relational properties on HL7 data types have either bl or bn as domain. The casts to boolean enable that user defined functions implementing the relational properties, may be used in filter expressions.

3.5 Real Number

A BNF-style grammar is defined by the standard for the literal form of reals, so the first condition on $Con(real)$ is met. $Dom(real)$ is also well defined, since it is $\mathbb{R}$.

[2]The literal yr was chosen as example. Actual UCUM units for year are a_t, a_j and a_g.

Current CPUs provide IEEE-754 floating point arithmetic, making for an obvious binary representation. The double precision numbers have a precision of more than 15 fractional digits, which seems adequate for the use cases with the HL7 real number data type.

However, as the following example calculation shows, IEEE-754 is not adequate for the HL7's real. Comparing pqs that have comparable but different units means computing and comparing canonical values which may result in rounding errors[3] .

```
/* canonical values of 1l and dm3 differ */
SELECT 0.1^3::float8 - 0.001::float8 AS error,
       0.001::float8 = 0.1^3::float8 AS equal;
        error         | equal
----------------------+-------
 2.16840434497101e-19 | f
```

The rounding errors described above can be avoided with an arbitrary precision real implementation.

3.6 Physical Quantity

The literal form Con(pq) is well defined; it is the string representation of the value followed by optional white space, followed by a unit string that conforms to UCUM[7], which defines a BNF-style grammar for the unit constants.

The domain Dom(pq) poses no problems either, as it consists the union of all data values of each unit, where the set of data values of a single unit is isomorphic to the real numbers.

The binary representation of compositional properties are trivial, except the value property, which is specified to be a real number. Section 3.5 discusses binary representations of real numbers.

A philosophical question arises when considering the following: is `1 m` equal to `100 cm`? To answer this, it is necessary to think of what object is described by the term `1 m`. Is this the same object as `100 cm` and are both terms just different names for the same object? Or do both terms describe two distinct objects? It is a choice that can be made, and there is no wrong or right choice. Both the HL7 specification and conventions in database theory make the choice that the terms describe different objects. From database theory it follows from the unique names assumption. The HL7 specification does so by making the distinction between the equal and identical relational properties on pq. Two pq objects are identical iff all compositional properties equal. The equal relation is not the usual equality on objects, but an equivalence relation identical relation on the powerdomain of pq, which is defined as 'has the same canonical value'.

```
SELECT equal('1m'::pq,'100cm'::pq);
 equal
-------
 true

SELECT identical('1m'::pq,'100cm'::pq);
 identical
-----------
 false
```

[3]The rounding error ε is inherent in the IEEE-754 specification. The actual value of ε differs per implementation. ε on Intel x86 hardware is $\approx 1.08e-19$.

The distinction between equality and identity has consequences for the index access methods on the pq data type. According to the HL7 specification, the identical property has no use other than in the definition of discrete sets. In relational database terms, discrete sets can be constructed using unique indexes. With index operator classes defined as follows, the user may choose whether or not the unique names assumption holds.

```
CREATE OPERATOR CLASS pq_ops_equal
  DEFAULT FOR TYPE pq USING btree AS
    OPERATOR  3  = ,
    FUNCTION  1  btpqcmp_equal(pq, pq);

CREATE OPERATOR CLASS pq_ops_identical
  FOR TYPE pq USING btree AS
    OPERATOR  3  == ,
    FUNCTION  1  btpqcmp_identical(pq, pq);

/* default class uses equality */
CREATE INDEX idx ON rel
  USING btree(a);

/* choose identical operator */
CREATE UNIQUE INDEX idx ON rel
  USING btree(a pq_ops_identical);
```

The following examples of the pq data type show that it is easy to perform calculations on physical quantities. Combined with the expression syntax of SQL this results in concise source code that is easy to write and understand.

```
CREATE TABLE obs
  (ptnt int, effectivetime ts, dosage pq);
INSERT INTO obs VALUES
  (1, '200910011214', '10 ml'),
  (1, '200910041307', '100 ml'),
  (2, '200910080856', '1000 ml'),
  (1, '200910010915', '10 ml'),
  (3, '200910022312', '50 ml'),
...

SELECT ptnt,
       effectivetime::date,
       convert(SUM(dosage),'l') AS "sum",
       convert(AVG(dosage),'l') AS "average"
FROM obs
GROUP BY ptnt,effectivetime::date
HAVING contains('[100ml;500ml[', sum(dosage))
ORDER BY ptnt,effectivetime;

 ptnt | effectivetime |  sum   | average
------+---------------+--------+---------
    1 | 2009-10-01    | 0.12 l | 0.01 l
    1 | 2009-10-02    | 0.15 l | 0.05 l
    2 | 2009-10-02    | 0.3 l  | 0.05 l
    3 | 2009-10-01    | 0.15 l | 0.01 l
    3 | 2009-10-02    | 0.45 l | 0.05 l
    4 | 2009-10-04    | 0.3 l  | 0.1 l
(6 rows)
```

3.7 Flavors of physical quantities

HL7s *flavors* are data types that are defined by constraining another data type. For instance, a pq_time is a physical quantity where the unit compares to seconds. The CREATE DOMAIN command of PostgreSQL provides exactly this feature.

```
CREATE DOMAIN pq_time AS pq
        CONSTRAINT pq_time_compares_to_s
        CHECK (compares(value,'s'));

SELECT '10 ml'::pq_time;
ERROR:  value for domain pq_time violates
        check constraint "pq_time_compares_to_s"
```

3.8 Instance Identifier

Like in the boolean case, it is easy to see that the instance identifier data types satisfies the conditions from section 3.1.

An important aspect of relational databases is entity integrity, which states that no primary key value of a base relation, which are those relations defined independently of other relations, is allowed to be null or to have a null component. Thus, if ii is used to identify entities, it may not have a null value. Unlike bl, ii has no flavor that constrains it to be nonnull. It is neither possible to enforce this constraint on HL7 nullflavors with the database NOT NULL constraint, since we implemented no interaction between the database NULL and HL7s nullflavor, described in section 3.2. In other words, it is impossible to force ii data values to not have a nullflavor. A solution would be to introduce the data type or flavor in, analogue to the bn flavor of bl .

3.9 Interval PQ

Although the literal form is specified, parsing intervals of physical quantities is complicated by two HL7 definitions; the dash form is allowed in ivl<pq> and UCUM units can also contain brackets, as is shown by the following two examples respectively:

```
SELECT '-8m--2m'::ivl_pq;
  ivl_pq
-----------
 [-8 m;-2 m]

SELECT '[100mm[Hg];120mm[Hg]]'::ivl_pq;
  ivl_pq
-----------
 [100 mm[Hg];120 mm[Hg]]
```

These forms do make parsing of literal forms more complicated; HL7 specification writers can make the implementers job easier if they separate different items in a literal form by characters not allowed in those items.

What are the elements of Dom(ivl<pq>)? Is ⟦<1m⟧ the same object as ⟦]NullFlavor.NINF m;1m[⟧? The answer to this question depends on the meaning of the *same* predicate. The equal property is implemented as user defined function that is true if the low and high values of both intervals are the same, and identical returns true iff they are equal and have the same literal form.

```
SELECT equal('30m [20m]'::ivl_pq,
             '[20m; 40m]'::ivl_pq);
 equal
-------
 true
(1 row)
```

3.10 TS and interval of TS

Like ivl<pq>, intervals of timestamps inherit attributes from the ivl<t> template and thus have a clear literal definition.

The binary representation of time can be implemented as an offset to a common epoch. Intervals can be seen as a two of these offsets when the bounds are known, and a width if not. All these pose no problem when implementing the compositional properties as a user defined type.

The HL7 specification defines a number of relational properties on `ts` and `ivl<ts>`. Promotion and demotions are relations between quantity data types and their interval data types. Since `ts` have a certain precision, it can be viewed as an interval of time. This interval is called the promotion of the `ts` . Take for instance the promotion of the `ts` 2008 to the entire year 2008:

```
SELECT '2008'::ts, promotion('2008'::ts),
       demotion(promotion('2008'::ts));
  ts  |  promotion  | demotion
------+-------------+----------
 2008 | [2008;2009[ | 2008
(1 row)
```

Other useful functions on `ivl<ts>` are the containment and overlap functions, that are often used in filter expressions. Here is an example with the `@>` containment operator, that queries all records with a time interval that contains the time interval `2001..2002`.

```
SELECT v FROM rel WHERE v @> '2001..2002';
      v
-------------
 [2000;2003[
 [2000;2004[
(2 rows)
```

3.11 Concept Descriptor

The most used data type in the HL7 specification is the concept descriptor. HL7 defines three forms of concept descriptors: `cd`, `cv` and `cs`. The standard defines the literal form only for `cs`. We adopt a common used literal form like `EVN:2.16.840.1.113883.5.1001`, for the event code in the `ActMood` codesystem, and added valuesetoid, versions of codesystem and valueset, and the originaltext.

Since user input of concepts using the standard literal form is error prone because of the OID numbers involved, we also allow input of concepts using an alternative format. To this end, we implemented a type modifier on the `cv` data type, that, if used, denotes the *conceptdomainname*. For instance:

```
WITH cvvaluetable AS
(SELECT 'active|Ongoing treatment'::cv('ActStatus')
 AS c)
SELECT code(c), codesystem(c), codesystemname(c),
       codesystemversion(c), valueset(c),
       valuesetname(c), valuesetversion(c),
       originaltext(c)
FROM   cvvaluetable;
-[ RECORD 1 ]-----+----------------------------
code              | active
codesystem        | 2.16.840.1.113883.5.14
codesystemname    | ActStatus
codesystemversion | 2009-08-30
valueset          | 2.16.840.1.113883.1.11.15933
valuesetname      | ActStatus
valuesetversion   | 2009-08-30
originaltext      | Ongoing treatment
```

The type modifier format of the `cv` data type may also be used to create table attributes, which then acts as a constraint on the attribute:

```
CREATE TABLE rel (code cv('ActStatus'));
INSERT  INTO rel VALUES ('x');
ERROR:  invalid code 'x' for codeSystem ActStatus
INSERT INTO rel VALUES ('completed');
SELECT * FROM rel;
                                          code     ..
-----------------------------------------------..
 completed:2.16.840.1.113883.5.14@2009-08-30:2.16..
```

Regarding the domain, there are similar issues as with the domain definitions of `pq` and `ivl<ts>`. What are the distinct objects in the concept descriptor domain? It makes sense to define the elements to be the objects that are isomorphic to `cv`'s compositional properties, but under this interpretation there is a problem with the translations of the concept into other codesystems. We believe translations are not among the basic intrinsic compositional properties of the concept descriptor. Translations are added to concept descriptions in communications as a service from a sending communication service to the receiver, but adding a translation can never mean that the actual described concept, the underlying object, is a different one.

An important property of the concept descriptor is the `implies` property with operator `<<`, which is defined as follows: a `<<` b iff a implies b, which means that a is a specific kind of b. With the `cv` data type it is possible to query for all specializations of e.g. a SNOMED-CT clinical finding code `404684003`:

```
SELECT displayname(conceptid) from rel
WHERE conceptid <<
   '404684003|Clinical finding'::cv('SNOMED-CT');
               displayname
-----------------------------------------
 Rupture of papillary muscle
 Unspecified visual field defect
 Oesophageal body web
 Benign tumour of choroid plexus
 Sulphaguanidine adverse reaction
..
```

The last condition of section 3.1, which states that the meaning of data values must be closed under application of relational operators, is not satisfied by the `cs` data type as it is specified. The conceptdomain is 'fixed' by the context in which the data value occurs. It has a literal form that consists only of the code; other properties are omitted. In a relational database setting, contexts of data values change under the application of relational operators, and so will a possible fixed codesystem, if it exists.

Though `cs` as it is defined does not comply with closure under relational operators, the way it is used in the HL7 specification much resembles our implementation of the `cv` data type in its type modifier form, such as `cv('ActMood')`. In the literal form input, only the code may be listed, and when used in table attributes, the type modifier 'fixes' the conceptdomain. But in the `cv` case, if e.g. the data value is projected into another tuple, no context information is lost, since that is carried by the modified `cv` type.

This concludes the analysis of the data types.

4. PERFORMANCE

This section shows the results of a number of tests that were performed on synthetic data, were two implementations of the physical quantity data type are compared:

1. *MGRID user defined type PQ* uses the user defined `pq` data type as described in this paper.
2. *PQ properties as columns* uses the standard types of the database server, where the compositional properties are distinct columns in the test relation.

The graphs show the average of 20 test runs for each different number of tuples. During each run, new random data was generated with the following characteristics:

- 20 different units chosen at random and always the unit `m`. We believe 20 different units is a reasonable ballpark figure for common uses of relations with `pq` values. The unit `m` was chosen to match queries in the performance test.
- for each unit, values were randomly distributed in a Gaussian distribution with μ 0 and σ 10000. The size of the distribution was chosen to match the range queries to match a small fraction of the total number of tuples. The Gaussian distribution was chosen because it matches many natural occurring phenomena.

Figure 4 shows the amount of time it took to create the test relations. In both cases the same UCUM parser validated correctness of unit strings and comparable calculations were performed to insert proper values into the database. Figure 5 shows a sequential scan that queries for values between `1.0km` and `1.2km`. The user defined type `pq` is between 30% to 50% faster in these operations, which is probably due to the SQL function call and context switch overhead that is expected in the implementation with properties as columns, as well as the larger number of columns involved in the latter case.

Figures 6, 7, 8 and 9 compare index performance in creation time, size and equality and range scanning. The equality scan selected the value `1.2km` and returned at least a single row. The range scan selected all values between `1.0 km` and `1.2km` and returned about 0.1% of all tuples. The graphs show that the user defined type outperforms the implementation based on a set of distinct table columns in both size of the index as well as index scans. The equality scan does not exhibit a clear difference between some of the data sizes. We believe this is due to the fact that these timings differ tens of microseconds, and are below the point where our timing method can produce accurate results. It is expected that if a more accurate measurement for short timing could be used, these values would be in tune with the timings for the larger relation sizes. The timings for the index range scan may seem large relative to the equality scan. It must be kept in mind however that the scan must skip values that are not of the same unit and that there is nullflavor handling taking place. For instance, a value of `Nullflavor.TRC ml` must be returned in a query that requests values `> 0 ml`. These properties make the index structure more complex.

On the whole these figures show that the user defined type `pq` outperforms an implementation based on a set of distinct columns that match `pq`'s compositional properties. With regard to index size, the user defined `pq` type is only 40% of the size of the compositional `pq` implementation. The user defined type is about 40% faster at index creation than

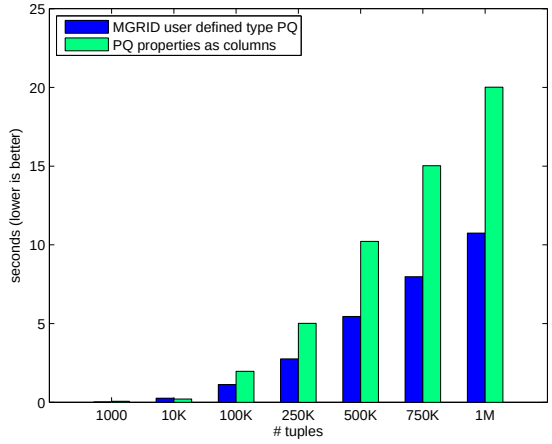

Figure 4: *Insert time*

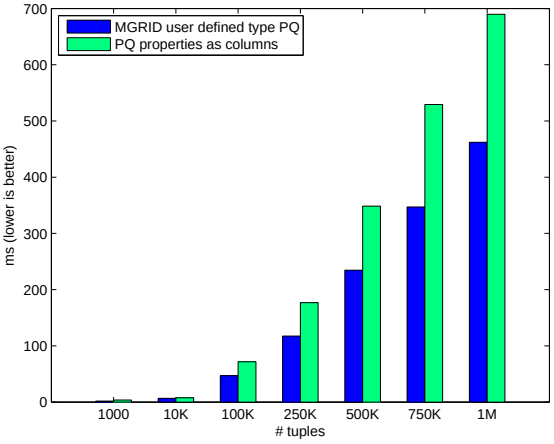

Figure 5: *Sequential scan*

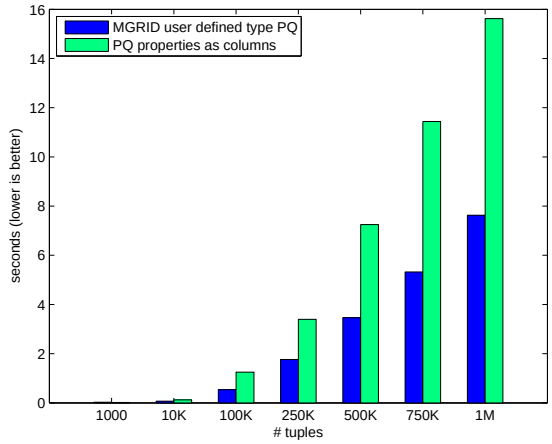

Figure 6: *Index creation time*

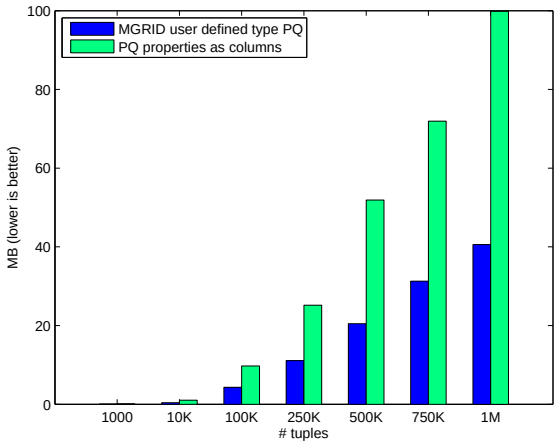

Figure 7: *Index size*

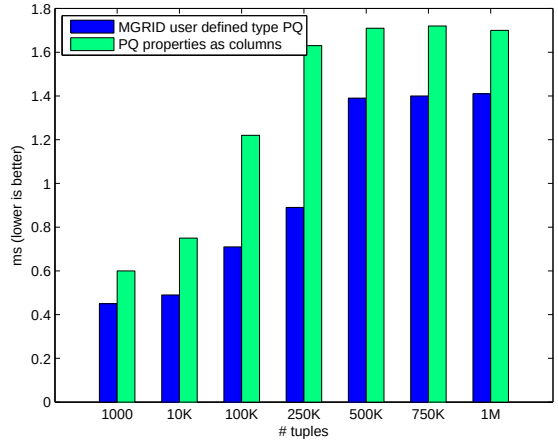

Figure 8: *Index equality scan*

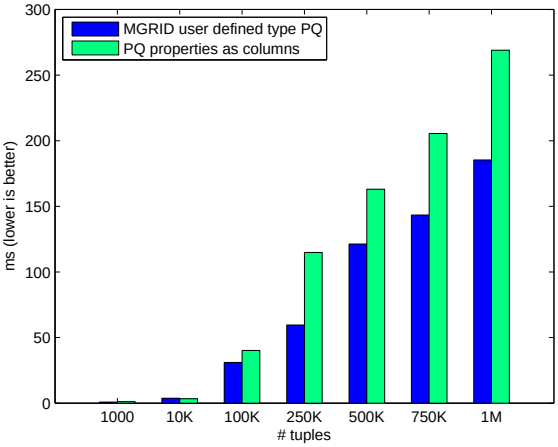

Figure 9: *Index range scan*

the compositional implementation. The consequences for insertion of tuples under the presence of indexes are similar. With index scanning the performance gain of using the user defined `pq` is about 25% for range queries and comparable for equality queries.

5. CONCLUSIONS

Though the HL7 data types are not designed to be implemented in a relational database, it is possible to implement several of the most used data types such as `pq`, `bl`, `ii` and `cv`.

The type hierarchy posed no problem to implement. Other aspects of the data type specification had a less obvious solution, such as nullflavor integration, defining missing literal forms, and choice in application of the unique names assumption. Some data type constructs could be implemented in a single database data type, but are more appropriately mapped to multiple data types or even relations instead of single data values. Good examples here are the translations of concepts in the concept descriptor data type, and discrete sets. The *other* nullflavor and the `cs` data type are problematic in the sense that their meaning depends on the context in which a data value is used. None of these problems were show stoppers.

The relational properties are easy to implement with user defined functions. Expressions involving HL7 predicates may be used in filter expressions, after adding casts that allow interaction between the database boolean and HL7's `bl` and `bn`. It is possible to provide indexing for the complete range of relational operators on data types.

Though the HL7 specification defines names for relational properties, it lacks operator symbols such as `+` for addition. As a result, differences in operator symbol designations can be expected between implementations of the HL7 data types.

Our analysis results in the following suggestions for improvement in the HL7 data type specification:

- Add a literal form for all data types that do not yet have one defined.
- Translations of a concept descriptor should not be part of the concept descriptor data type.
- Remove constructs where the meaning of an artifact depends on the context in which it is used.
- An alternative interpretation of binary operations involving the *not applicable* nullflavor is proposed in table 6.
- Add a note that IEEE-754 floating point representations found in hardware co-processors is inadequate for calculations involving physical quantities.
- Add a nonnull flavor of the `ii` data type.
- Add operator symbols for relational properties.

It may seem old fashioned to implement HL7 data types as user defined types; current practice is to implement complex types in the persistence layer, where the database is used to store the individual components of these types, using object relational mapping (ORM). We believe that augmenting the database server by adding these types as native type makes sense for several reasons. First, it allows the database server to store and retrieve information more effectively, thereby reducing the amount of data exchanged with

the persistence layer and providing better scaling. Besides a smaller performance footprint of the persistence layer, the user defined types also provide powerful primitive operations on HL7 data types to the persistence layer, thereby reducing its size and complexity and indirectly the prevalence of software engineering errors. Third, consistent reasoning about HL7 data in a heterogeneous application environment requires a consistent common understanding between the applications, that the database server with user defined types is in a unique position to supply.

Our approach does not rule out ORMs, but does simplify them by making use of the database HL7 data types. We have shown that operations on the user defined types are significantly faster and occupy less space than an alternative implementation that resembles current practice of implementing complex types.

6. REFERENCES

[1] *Health Informatics - Harmonized datatypes for information interchange*. International Organization for Standardization, 2008.

[2] PostgreSQL 8.3.6 documentation, user-defined types. http://www.postgresql.org/docs/8.3/static/xtypes.html, accessed 2009/03/07, 2008.

[3] P. Buneman, A. Jung, and A. Ohori. Using powerdomains to generalize relational databases. *Theor. Comput. Sci.*, 91(1):23–55, 1991.

[4] E. F. Codd. Extending the database relational model to capture more meaning. *ACM Transactions on Database Systems*, 4:397–434, 1979.

[5] L. Libkin. *Aspects of partial information in databases*. PhD thesis, Philadelphia, PA, USA, 1995.

[6] G. Schadow. Hl7 version 3 standard: Data types - abstract specification, release 2. http://www.hl7.org/v3ballot/html/infrastructure/datatypes_r2/datatypes_r2.htm, accessed 2009/03/07, 2008.

[7] G. Schadow, C. J. McDonald, J. G. Suico, U. Fohring, and T. Tolxdorff. Units of measure in clinical information systems. *J Am Med Inform Assoc*, 6(2):151–62, 1999.

[8] R. Spronk. Hl7 creates a rim based application architecture (rimbaa) group. http://www.ringholm.de/column/hl7_RIM_based_application_architecture.htm, accessed 2009/03/07, 2008.

[9] M. Stonebraker. Inclusion of new types in relational data base systems. pages 262–269, 1986.

[10] M. Stonebraker and L. A. Rowe. The design of postgres. *SIGMOD Rec.*, 15(2):340–355, 1986.

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

[12] C. Zaniolo. Database relations with null values. In *PODS '82: Proceedings of the 1st ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 27–33, New York, NY, USA, 1982. ACM.

APPENDIX

A. MGRID

The mission statement of MGRID, a Dutch IT company, is to develop the ultimate medical database. Part of the database solution of MGRID is a structured HL7 RIM database. This database makes use of the HL7 data types that are the subject of this paper. Other aspects of MGRID's database solution encompass solutions for high availability, linear scalability, performance, encryption of private medical data and authorization for users based on their role as known in the RIM database. MGRID also provides fast terminological services and tooling for model driven architectures with Detailed Clinical Models (DCM).

The MGRID RIM database can be used in conjunction with object relational mapping (ORM) tools like hibernate. In this combination MGRID guarantees performance and delivers powerful primitive operations on medical data in the HL7 and SNOMED-CT languages. The mapping with hibernate ensures that the application software engineer can make optimal use of MGRID's primitives in his or her own language.