Could packing be specified in terms of closed-open interval semantics without having to commit to granularity ?

Erwin Smout

Introduction

“Temporal Data and the Relational Model”[1], by Hugh Darwen, Chris Date & Nikos Lorentzos, presents a solution for dealing with the problem of “temporal data” in relational databases. Some of the concepts involved in the proposed solution are interval types (data types whose values are the possible -contiguous- ranges of values of other, underlying, ordered types), operators to operate on such interval values (some of them, e.g. OVERLAPS(), MERGES(), … akin to what is commonly known as Allen's interval operators, some others, e.g. UNION(), INTERSECT(), … akin to operators that are commonly known from set theory), and two canonical forms of relations labeled “unpacked form” and “packed form”.

One criticism that regularly pops up is the reliance of their solution on the underlying, ordered, types being not only ordered, but ordinal as well, meaning that the solution requires four operators to be defined for those underlying types, namely FIRST(), LAST(), PRIOR() and NEXT()[2]. This means that if a type FLOAT is ordered (a 'greater than' operator has been defined for it), but not ordinal (e.g. there is no NEXT() operator that allows to compute the “next-in-line” for a given FLOAT value), then the approach by these authors cannot be made to apply to this FLOAT type, and the user is, under those authors' approach, deprived of the possibility to work with “ranges of FLOAT values”. Besides FLOAT, other types that would probably be in this situation are type CHAR (if there is no type-defined maximum length, then getting the NEXT() after “a” would have to mean appending (e.g.) an “a” to get “aa”, then the next would be “aaa”, and so ad infinitum and we'd never get from “a” to “b” - and in particular it would also be impossible to define which CHAR value is the LAST()), the nominator/denominator version of RATIONAL (try figuring out what the NEXT() rational is after 355/113) and if a user wanted to define a type PRIME, he'd also be in a bit of trouble implementing a NEXT() operator for that. Not all these types are as “pathological” as the latter, and so there is indeed a bit of a case to be made for not wanting to depend on types being ordinal.

The issue is that as good as the whole subject and/or purpose of the book, is to arrive at a definition for “packed canonical form”, that the definition that is arrived at, is expressed in terms of “unpacked canonical form”, and the “unpacked canonical form” depends on the notion of granules.

Hence the title question, to which the present document seeks to offer an answer.

Notational stuff

ITA are attribute names for Interval-Typed Attributes in a relation. Since we will be dealing with an arbitrary number of range attributes/dimensions over which is being packed, the involved attributes will be named ITA_1, ITA_2, …, ITA_n, with n the number of range dimensions involved (possibly zero).

---

1 In the remainder of the document, this book will be referred to as “TDATRM”.

2 A more formal definition of the term “ordinal type”, suggested on the discussion list ttm@thethirdmanifesto.com, would be that it is a type that is order isomorphic to the set of the first N natural numbers, where N is the cardinality of the type.

In contexts where there is no relevance to the 'ordering' of the attributes within the packing attributes list, we will denote the collection of packing attributes as a set : $\{ITA_1, ITA_2, \ldots, ITA_n\}$. In contexts where there is relevance to the 'ordering' of the attributes within the packing attributes list, we will denote the collection of packing attributes as a list : $(ITA_1, ITA_2, \ldots, ITA_n)$.

All interval values mentioned in the examples in the present document must be understood to be in "closed-open" notation, barring explicit statements to the contrary.

The definition we're aiming to achieve

A relation R is in $(ITA_1, ITA_2, \ldots, ITA_n)$ packed normal form[3] iff
FORALL t_1,t_2, $t_1 \in R$, $t_2 \in R$, $t_1<>t_2$: $PNF(t_1,t_2)$

The definition essentially aims to state that a relation is in packed normal form (for some specific ordered list of packing attributes) if and only if all pairs of distinct tuples taken from the relation, are in a state of "satisfying (that particular) PNF", or, put alternatively, if and only if no two distinct tuples from the relation are in a "state of violation" of (that particular) PNF. Note how this implies that empty relations and singleton relations are always in packed normal form (for any list of packing attributes), by definition.

That "reduces" the problem to a space in which only two distinct tuples are involved, and leaves us having to define the exact nature of $PNF(t_1,t_2)$. That definition of $PNF(t_1,t_2)$ must satisfy the following properties :

- As we are looking for a definition of packed normal form that "expands" to types that are not granulated, the definition of $PNF(t_1,t_2)$ must be exclusively in terms of operators that are also definable for "contiguous", "ungranulated" types. One consequence is that no algebraic operators can be involved that rely themselves on NEXT() or PRIOR() being available/defined.
- As we are looking for a definition of packed normal form that really is a "proper expansion" of the definition from TDATRM, this definition, when applied to a type that is indeed "granulated", must yield the same results as the definition from TDATRM that relies on granularity.

One consequence of the latter bullet, is that we can be certain that the ordering of the attributes in a packing list, matters. Under the TDATRM approach, packing an unpacked relation on X, then on Y does not always yield the exact same result as packing that same unpacked relation on Y, then on X. Hence when speaking of a "packed normal form", it will be necessary to provide the particular ordering of the packing attributes that we are dealing with. Hence, we will be speaking of "$(ITA_1, ITA_2, \ldots, ITA_n)$ packed normal form", "$(ITA_1, ITA_2, \ldots, ITA_n)$ PNF" for short.

3 We use the term "normal form" here, but note that this is unrelated to the usage of that same term in the context of database design. In the context of database design, "normal form" is used in connection with, and as applying to, a logical database structure, in this document the term is used in connection with, and as applying to, relation values.

Preliminary definitions

Interval types & values

Interval types are types that are generated by an "interval type generator". Apart from having all the obvious properties (such as, e.g., a type constraint that prohibits badly ordered begin and end boundary values), these generated interval types & their values have :

- A possible representation consisting of the components FROM and TO (where FROM is the (closed/included) start point of the range value, and TO is the (open/excluded) end value of the range). Given an interval value iv, these components will be denoted using dotted notation, i.e. iv.FROM, iv.TO. Note that in Tutorial D, these expressions might appear as invocations of THE_FROM() and THE_TO() on these interval values.
- and where it must be understood that, as far as the treatment in this document goes, it is perfectly possible for such expressions to denote some "conceptual value of infinity" that compares to "actual" values of the underlying ordered type as appropriate. The fact that "infinity is not an actual value" (of the underlying ordered type), is a mere implementation problem, and is orthogonal to the treatment here.

OVERLAPS

The expression 'i_1 OVERLAPS i_2', an invocation of the operator OVERLAPS(it,it), where i_1 and i_2 are expressions denoting interval values of the same interval type it, is defined to be equivalent to the expression

$$i_1.FROM < i_2.TO \text{ AND } i_2.FROM < i_1.TO$$

MEETS

The expression 'i_1 MEETS i_2', an invocation of the operator MEETS(it,it), where i_1 and i_2 are expressions denoting interval values of the same interval type it, is defined to be equivalent to the expression

$$i_1.FROM = i_2.TO \text{ OR } i_2.FROM = i_1.TO$$

{ITA_1, ITA_2, …, ITA_n} equivalence classes in a relation

Let r be a relation including at least the attributes {ITA_1, ITA_2, …, ITA_n}. A condition that between two tuples t_1 and t_2 (not necessarily distinct) in r, all corresponding attribute values other than {ITA_1, ITA_2, …, ITA_n} must be equal, is a condition that constitutes an equivalence relation within r. We call this equivalence relation an "{ITA_1, ITA_2, …, ITA_n} equivalence relation", and the equivalence classes defined by it "{ITA_1, ITA_2, …, ITA_n} equivalence classes".

Hence if two tuples t_1 and t_2 do not belong to the same {ITA_1, ITA_2, …, ITA_n} equivalence class, then at least one attribute value other than {ITA_1, ITA_2, …, ITA_n} differs between t_1 and t_2, and we define this condition to be sufficient for those two tuples to satisfy (ITA_1, ITA_2, …, ITA_n) PNF. In the remainder of the document, when two or more distinct tuples in a relation are considered, it will be implicitly assumed that those tuples all belong to the same {ITA_1, ITA_2, …, ITA_n} equivalence class, barring explicit statements to the contrary.

"Box" and "covered points"

Let t be a tuple with interval-typed attributes ITA_1, ITA_2, …, ITA_n, and interval values iv_1, iv_2, …, iv_n. We say the "box" for this tuple is the solid body in n-dimensional space consisting of all the points (c_1, c_2, …, c_n) such that :

- iv_1.FROM <= c_1 < iv_1.TO
- iv_2.FROM <= c_2 < iv_2.TO
- …
- iv_n.FROM <= c_n < iv_n.TO

Of the points in n-dimensional space that are part of the box, we say that they are "covered" by the tuple t, or conversely, that the tuple "covers" such a point :

- t COVERS p(c_1, c_2, …, c_n)
- p(c_1, c_2, …, c_n) COVBY t

Likewise, we say that a relation r "covers" such a point, and such a point "is covered by" that relation, if the relation contains at least one tuple that covers p :

r COVERS p(c_1, c_2, …, c_n) <===> EXISTS t, $t \in r$: t COVERS p(c_1, c_2, …, c_n)

A similar notion of "coverage" could more usefully be also defined for some {ITA_1, ITA_2, …, ITA_n} equivalence class c in a relation :

c COVERS p(c_1, c_2, …, c_n) <===> EXISTS t, $t \in c$: t COVERS p(c_1, c_2, …, c_n)

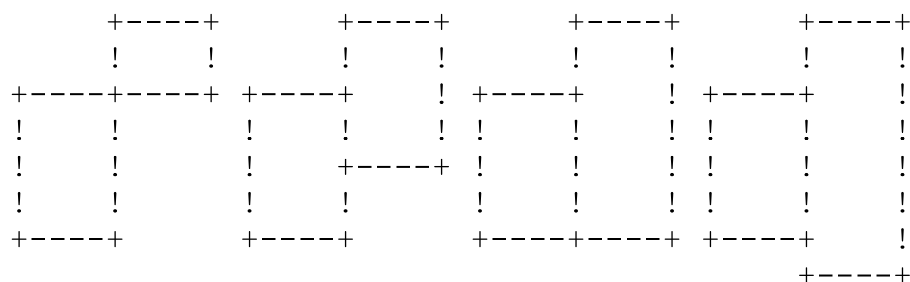
"{ITA_1, ITA_2, …, ITA_n} meeting tuples"

Two tuples t_1 and t_2 are "meeting tuples" (for the dimensions {ITA_1, ITA_2, …, ITA_n}) iff :

EXISTS x,x=1..n : t_1.ITA_x MEETS t_2.ITA_x[4]
FORALL y,y=1..n,y<>x : t_1.ITA_y OVERLAPS t_2.ITA_y

Informally, this property says that two tuples (or the boxes they describe) "meet" if there is exactly one dimension for which the respective interval values from the two tuples MEET, and the values from the tuples for all the remaining dimensions in {ITA_1, ITA_2, …, ITA_n} OVERLAP.

To illustrate in 2D space, according to this definition, the last three of the following are "meeting boxes" (and two tuples with interval values denoting these boxes are then "meeting tuples"), but the first one is not :



FORALL is carefully chosen for the "remaining dimensions" in order to make this condition represent "the opportunity to pick away slices of one box and merge them with [some slice of] the other box to obtain another one".

4 We use the same dotted notation on tuples to "access" their attribute values. In Tutorial D, these constructs would appear in the syntactic form ITA_x FROM t_n.

"(ITA_1, ITA_2, …, ITA_n) rearrangeable tuples"

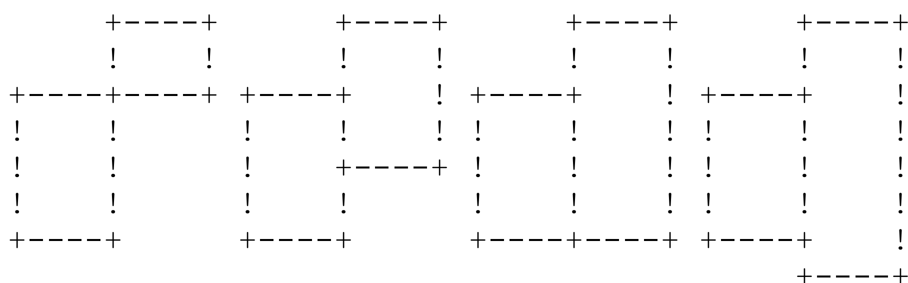
Two tuples t_1 and t_2 are said to be "(ITA_1, ITA_2, …, ITA_n) rearrangeable" iff :

EXISTS y,y=1..n : $t_1.ITA_y$ MEETS $t_2.ITA_y$
FORALL x,x<y : $t_1.ITA_x$ EQUALS $t_2.ITA_x$
FORALL z,z>y,z<=n : $t_1.ITA_z$ OVERLAPS $t_2.ITA_z$

Observe that this definition is strictly stronger than the definition for {ITA_1, ITA_2, …, ITA_n} meeting tuples. Iow, the set of "pairs of rearrangeable tuples" is a subset of the set of "pairs of meeting tuples". Or yet iow, rearrangeable implies meeting, and not meeting implies not rearrangeable.

Informally, this property says that two tuples are "rearrangeable" if an alternative set of tuples exists (that alternative set must not necessarily be of cardinality two) that covers the same set of points in n-dimensional space, but where the "meeting" between the tuples (and the boxes they denote) happens "in a less prevailing dimension".

To illustrate in 2D space (and assuming the usual orientation for the X and Y dimensions), the last three of the following are "(X,Y) rearrangeable", and none of them are "(Y,X) rearrangeable" :



How does this definition behave in the degenerate case of no ranges at all ? The EXISTS clause can clearly not be satisfied, and hence distinct tuples are never () rearrangeable, by definition.

In the case of a single packing range ("(ITA_1) rearrangeable"), both FORALL clauses quantify over the empty set, hence both degenerate to TRUE, and overall it means two such tuples are "(ITA_1) rearrangeable" only if the values for their interval-typed attribute ITA_1 MEET. You will remark that if those values OVERLAP, then the tuples could (should) be considered as being "rearrangeable" too, but cases of overlaps are dealt with in another way, we'll get to that shortly.

"{ITA_1, ITA_2, …, ITA_n} mergeable" tuples

Two tuples t_1 and t_2 are said to be "{ITA_1, ITA_2, …, ITA_n} mergeable" iff :

EXISTS x,x=1..n : $t_1.ITA_x$ MEETS $t_2.ITA_x$
FORALL y,y=1..n,y<>x : $t_1.ITA_y$ EQUALS $t_2.ITA_y$

Informally, if two tuples are "{ITA_1, ITA_2, …, ITA_n} mergeable", this means they can be replaced by a single tuple that covers the same set of points as the two tuples together.

Observe that the definition for "{ITA_1, ITA_2, …, ITA_n} mergeable" is strictly stronger than the one for "(ITA_1, ITA_2, …, ITA_n) rearrangeable", hence, "mergeable" implies "rearrangeable", and "not rearrangeable" implies "not mergeable".

Also observe that the case of tuples/boxes that "overlap in one dimension" (instead of meeting), and are equal in all others, is not taken into account here. Cases of overlaps are dealt with in another way.

“{ITA_1, ITA_2, …, ITA_n} intersectable” tuples

Two tuples t_1 and t_2 (not necessarily distinct, but that's not important) are said to be “{ITA_1, ITA_2, …, ITA_n} intersectable” iff :

FORALL x,x=1..n : $t_1.ITA_x$ OVERLAPS $t_2.ITA_x$

If two tuples are “{ITA_1, ITA_2, …, ITA_n} intersectable”, this means, loosely speaking, that a “range intersection between the two tuples” will produce another tuple that “denotes a nonempty box”.

Observe that “{ITA_1, ITA_2, …, ITA_n} intersectable” precludes “(ITA_1, ITA_2, …, ITA_n) rearrangeable”, and vice versa. “Intersectable” requires all range dimensions to overlap, rearrangeable requires at least one dimension to not overlap.

“(ITA_1, ITA_2, …, ITA_n) PNF” tuples (informal version)

We say that two distinct tuples t_1 and t_2 “satisfy (ITA_1, ITA_2, …, ITA_n) PNF”, iff the two tuples are in different {ITA_1, ITA_2, …, ITA_n} equivalence classes, or they are in the same {ITA_1, ITA_2, …, ITA_n} equivalence class and they are neither {ITA_1, ITA_2, …, ITA_n} intersectable nor (ITA_1, ITA_2, …, ITA_n) rearrangeable.

Iow, one of the following conditions must be satisfied :

- t_1 and t_2 are in different {ITA_1, ITA_2, …, ITA_n} equivalence classes,
- Both of the following conditions must be satisfied :
  1. t_1 and t_2 are not “{ITA_1, ITA_2, …, ITA_n} intersectable”
  2. t_1 and t_2 are not “(ITA_1, ITA_2, …, ITA_n) rearrangeable”

“(ITA_1, ITA_2, …, ITA_n) PNF” tuples (formal version)

Substituting the formal definitions for being “intersectable” and/or “rearrangeable” in the “informal version”, we can say that two distinct tuples t_1 and t_2 “satisfy (ITA_1, ITA_2, …, ITA_n) PNF”, iff one of the following conditions is satisfied :

- At least one attribute value other than {ITA_1, ITA_2, …, ITA_n} differs between t_1 and t_2,
- Or else both of the following are satisfied :
  1. NOT (FORALL x,x=1..n : $t_1.ITA_x$ OVERLAPS $t_2.ITA_x$)
  2. NOT (EXISTS y,y=1..n : $t_1.ITA_y$ MEETS $t_2.ITA_y$
FORALL $x,x<y$: $t_1.ITA_x$ EQUALS $t_2.ITA_x$
FORALL $z,z>y,z<=n$: $t_1.ITA_z$ OVERLAPS $t_2.ITA_z$)

which translates to :

- At least one attribute value other than {ITA_1, ITA_2, …, ITA_n} differs between t_1 and t_2,
- Or else both of the following are satisfied :
  1. EXISTS x,x=1..n : NOT ($t_1.ITA_x$ OVERLAPS $t_2.ITA_x$)
  2. FORALL y,y=1..n : NOT ($t_1.ITA_y$ MEETS $t_2.ITA_y$
AND FORALL $x,x<y$: $t_1.ITA_x$ EQUALS $t_2.ITA_x$
AND FORALL $z,z>y,z<=n$: $t_1.ITA_z$ OVERLAPS $t_2.ITA_z$)

and ultimately to our final definition for PNF(t_1,t_2) :

- At least one attribute value other than {ITA_1, ITA_2, …, ITA_n} differs between t_1 and t_2,
- Or else both of the following are satisfied :
  1. EXISTS x,x=1..n : NOT ($t_1.ITA_x$ OVERLAPS $t_2.ITA_x$)
  2. FORALL y,y=1..n : NOT ($t_1.ITA_y$ MEETS $t_2.ITA_y$)
OR EXISTS $x,x<y$: $t_1.ITA_x <> t_2.ITA_x$
OR EXISTS $z,z>y,z<=n$: NOT ($t_1.ITA_z$ OVERLAPS $t_2.ITA_z$)

Interlude : some case studies with visualizations

In this section, we will explore, starting with a number of cases in 2D space, how these definitions all work out. The visualizations of the boxes (rectangles, thus) will have the X and Y axes in their usual orientation, the attribute names in the relations describing these boxes will be, unsurprisingly, X and Y, respectively.

All cases have in common that they already satisfy the condition of "not being {X,Y} intersectable". It is also assumed that the tuples are all in the same {X,Y} equivalence class. This will not be repeated over and over for each case. Hence, we will exclusively be focusing on the condition that

FORALL y,y=1..n : NOT (t_1.ITA$_y$ MEETS t_2.ITA$_y$)
OR EXISTS $x,x<y$: t_1.ITA$_x$ $<>$ t_2.ITA$_x$
OR EXISTS $z,z>y,z<=n$: NOT (t_1.ITA$_z$ OVERLAPS t_2.ITA$_z$)

Case 1

```
+----+            X     Y
!    !          ----- -----
!    !           1- 5  1-10
!    +----+      5- 9  1- 4
+----+----+
```

Testing for (X,Y) PNF, y=1 is the X dimension, y=2 is the Y dimension.
For the X dimension (y=1) :

- NOT (t_1.X MEETS t_2.X) FALSE
- EXISTS $x,x<1$: t_1.ITA$_x$ $<>$ t_2.ITA$_x$ FALSE
- EXISTS $z,z>1,z<=n$: NOT (t_1.ITA$_z$ OVERLAPS t_2.ITA$_z$) FALSE

Hence this relation is not in (X,Y) PNF.

Testing for (Y,X) PNF, y=1 is the Y dimension, y=2 is the X dimension.
For the Y dimension (y=1) :

- NOT (t_1.Y MEETS t_2.Y) TRUE

For the X dimension (y=2) :

- NOT (t_1.X MEETS t_2.X) FALSE
- EXISTS $x,x<2$: t_1.ITA$_x$ $<>$ t_2.ITA$_x$ TRUE

Hence this relation is in (Y,X) PNF.

Case 2

```
+----+            X     Y
!    !          ----- -----
!    !           1- 9  1- 4
+----+----+      1- 5  4-10
+---------+
```

Testing for (Y,X) PNF, y=1 is the Y dimension, y=2 is the X dimension.
For the Y dimension (y=1) :

- NOT (t_1.Y MEETS t_2.Y) FALSE
- EXISTS $x,x<1$: t_1.ITA$_x$ $<>$ t_2.ITA$_x$ FALSE
- EXISTS $z,z>1,z<=n$: NOT (t_1.ITA$_z$ OVERLAPS t_2.ITA$_z$) FALSE

Hence this relation is not in (Y,X) PNF.

Testing for (X,Y) PNF, y=1 is the X dimension, y=2 is the Y dimension.
For the X dimension (y=1) :

- NOT (t_1.X MEETS t_2.X) TRUE

For the Y dimension (y=2) :

- NOT (t_1.Y MEETS t_2.Y) FALSE
- EXISTS $x,x<2 : t_1.ITA_x <> t_2.ITA_x$ TRUE

Hence this relation is in (X,Y) PNF.

Case 3

```
+----+              X     Y
!    !            ----- -----
! t3 !             1- 5  1- 4    (t1)
!    !             5- 9  1- 4    (t2)
+----+----+        1- 5  4-10    (t3)
! t1 ! t2 !
+----+----+
```

Testing for (Y,X) PNF (and looking at t_1/t_2 exclusively), y=1 is the Y dimension, y=2 is the X dimension.
For the Y dimension (y=1) :

- NOT (t_1.Y MEETS t_2.Y) TRUE

For the X dimension (y=2) :

- NOT (t_1.X MEETS t_2.X) FALSE
- EXISTS $x,x<2 : t_1.ITA_x <> t_2.ITA_x$ FALSE
- EXISTS $z,z>2,z<=n$: NOT ($t_1.ITA_z$ OVERLAPS $t_2.ITA_z$) FALSE

Hence this relation is not in (Y,X) PNF.

Testing for (X,Y) PNF (and again looking at t_1/t_2 exclusively), y=1 is the X dimension, y=2 is the Y dimension.
For the X dimension (y=1) :

- NOT (t_1.X MEETS t_2.X) FALSE
- EXISTS $x,x<1 : t_1.ITA_x <> t_2.ITA_x$ FALSE
- EXISTS $z,z>1,z<=n$: NOT ($t_1.ITA_z$ OVERLAPS $t_2.ITA_z$) FALSE

Hence this relation is not in (X,Y) PNF.

Case 4

```
+----+---------+              X     Y
!    !   t5    !            ----- -----
! t2 !         !             1- 8  1- 4    (t1)
!    +----+----+             1- 4  4-12    (t2)
!    ! t3 !    !             4- 8  4- 7    (t3)
+----+----+ t4 !             8-11  1- 7    (t4)
!   t1    !    !             4-11  7-12    (t5)
+---------+----+
```

This case has a number of tuple pairs that are identical to case 1 (e.g. t_2/t_3) and also a number of tuple pairs that are identical to case 2 (e.g. t_2/t_1). Therefore, the one pair dictates that this relation cannot be in (X,Y) PNF, and the other pair dictates that this relation cannot be in (Y,X) PNF.

Hence, this relation is in no PNF at all (no PNF involving both X and Y, that is).

Case 5

```
+----+---------+
!\   !\        !\
! +--!-+-------!-+
! !  ! !       ! !
! !  +----+----+ !
! !  !\!  !\   !\!
+----+----+-+--!-+
!\!   \!  !\!  ! !
! +----+--!-+  ! !
+---------+----+ !
 \!        \!   \!
  +---------+----+
```

X	Y	Z	
1- 8	1- 4	1- 3	(t_1)
1- 4	4-12	1- 3	(t_2)
4- 8	4- 7	1- 3	(t_3)
8-11	1- 7	1- 3	(t_4)
4-11	7-12	1- 3	(t_5)

Testing for (X,Y,Z) PNF (and again looking at t_2/t_3 exclusively), y=1 is the X dimension, y=2 is the Y dimension, y=3 is the Z dimension.
For the X dimension (y=1) :

- NOT (t_1.X MEETS t_2.X) FALSE
- EXISTS x,x<1 : t_1.ITA$_x$ <> t_2.ITA$_x$ FALSE
- EXISTS z,z>1,z<=n : NOT (t_1.ITA$_z$ OVERLAPS t_2.ITA$_z$) FALSE

Hence this relation is not in (X,Y,Z) PNF. The same tuple pair also determines that this relation is not in (X,Z,Y) PNF.

Testing for (Y,X,Z) PNF (and again looking at t_1/t_3 exclusively), y=1 is the Y dimension, y=2 is the X dimension, y=3 is the Z dimension.
For the Y dimension (y=1) :

- NOT (t_1.Y MEETS t_3.Y) FALSE
- EXISTS x,x<1 : t_1.ITA$_x$ <> t_3.ITA$_x$ FALSE
- EXISTS z,z>1,z<=n : NOT (t_1.ITA$_z$ OVERLAPS t_3.ITA$_z$) FALSE

Hence this relation is not in (X,Y,Z) PNF. The same tuple pair also determines that this relation is not in (Y,Z,X) PNF.

Testing for (Z,X,Y) PNF, y=1 is the Z dimension, y=2 is the X dimension, y=3 is the Y dimension.
For the Z dimension (y=1) (for all of the possible tuple pairs) :

- NOT (t_1.Z MEETS t_3.Z) TRUE

For the X dimension (y=2), and inspecting the tuple pair t_3/t_4 :

- NOT (t_4.X MEETS t_3.X) FALSE
- EXISTS x,x<2 : t_4.ITA$_x$ <> t_3.ITA$_x$ FALSE
- EXISTS z,z>2,z<=n : NOT (t_4.ITA$_z$ OVERLAPS t_3.ITA$_z$) FALSE

Hence this relation is not in (Z,X,Y) PNF.

Testing for (Z,Y,X) PNF, y=1 is the Z dimension, y=2 is the Y dimension, y=3 is the X dimension.
For the Z dimension (y=1) (for all of the possible tuple pairs) :

- NOT (t_1.Z MEETS t_3.Z) TRUE

For the Y dimension (y=2), and inspecting the tuple pair t_4/t_5 :

- NOT (t_4.Y MEETS t_5.Y) FALSE
- EXISTS x,x<2 : t_4.ITA$_x$ <> t_3.ITA$_x$ FALSE
- EXISTS z,z>2,z<=n : NOT (t_4.ITA$_z$ OVERLAPS t_3.ITA$_z$) FALSE

Hence this relation is not in (Z,Y,X) PNF.

Consequentially, this relation is not in any possible PNF involving all of {X,Y,Z} !

Case 6

```
  X     Y     Z
----- ----- -----
 1- 3  2- 4  3- 4  (t1)
 1- 3  2- 3  4- 5  (t2)
 1- 4  3- 4  4- 5  (t3)
 2- 4  3- 5  5- 6  (t4)
 2- 4  4- 5  4- 5  (t5)
```

Given the three range dimensions, once again there are six possible PNF's to consider. The following are violated by, a.o., the tuple pairs indicated in the table below :

Z,Y,X	t_4/t_5
Z,X,Y	t_4/t_5
Y,Z,X	t_2/t_3
Y,X,Z	t_2/t_3
X,Z,Y	t_1/t_2 (do the check for the Z dimension, y=2)

The (X,Y,Z) PNF, on the other hand, is indeed satisfied by this relation (you can check this for yourself if you want to, but the process is a bit tedious).

Proof that (ITA_1, ITA_2, …, ITA_n) PNF is unique for any r

If all points in a box are covered by an equivalence class c in a relation, then c satisfies PNF only if that box is represented in a single tuple

Or, iow, no “subdivision” of an n-dimensional box in multiple tuples/boxes can possibly satisfy (ITA_1, ITA_2, …, ITA_n) PNF, for any permutation of the ITA attributes packing list.

Suppose, contrariwise, a singleton equivalence class ec_1 with a tuple t and n interval-typed attributes ITA_1, ITA_2, …, ITA_n, and a second equivalence class ec_2 with tuples t_{21}, t_{22}, …, t_{2j} (and the same interval-typed attributes) such that

$$\text{FORALL } p(c_1, c_2, \ldots, c_n) : p \text{ COVBY } ec_1 \Longleftrightarrow p \text{ COVBY } ec_2$$

Ec_1 satisfies (ITA_1, ITA_2, …, ITA_n) PNF, by definition, because it is a singleton. Suppose ec_2 satisfies (ITA_1, ITA_2, …, ITA_n) PNF too. Take some arbitrary point $p(c_1, c_2, \ldots, c_n)$ in the box. First, we show that if ec_2 satisfies (ITA_1, ITA_2, …, ITA_n) PNF, then all points $p(c_{1x}, c_2, \ldots, c_n)$ such that $t.ITA_1.FROM <= c_{1x} < t.ITA_1.TO$, must be covered by one and the same tuple in ec_2. For suppose contrariwise that there would be two distinct tuples t_{2j} and t_{2k}, both member of ec_2, each covering one of two distinct points $p(c_{1j}, c_2, \ldots, c_n)$ and $p(c_{1k}, c_2, \ldots, c_n)$ respectively, then we have :

- $t_{2j}.ITA_2.FROM <= c_2 < t_{2j}.ITA_2.TO$
- $t_{2k}.ITA_2.FROM <= c_2 < t_{2k}.ITA_2.TO$
- $t_{2j}.ITA_3.FROM <= c_3 < t_{2j}.ITA_3.TO$
- $t_{2k}.ITA_3.FROM <= c_3 < t_{2k}.ITA_3.TO$
- … (repeat for all subsequent dimensions up to and including n)

from which it follows that :

- $t_{2j}.ITA_2.FROM <= c_2 < t_{2k}.ITA_2.TO$
- $t_{2k}.ITA_2.FROM <= c_2 < t_{2j}.ITA_2.TO$
- $t_{2j}.ITA_3.FROM <= c_3 < t_{2k}.ITA_3.TO$
- $t_{2k}.ITA_3.FROM <= c_3 < t_{2j}.ITA_3.TO$

- … (repeat for all subsequent dimensions up to and including n)

from which it follows that (because all lines taken pairwise constitute the definition of overlapping intervals) :

- $t_{2j}.ITA_2$ OVERLAPS $t_{2k}.ITA_2$
- $t_{2j}.ITA_3$ OVERLAPS $t_{2k}.ITA_3$
- … (repeat for all subsequent dimensions up to and including n)

or iow, all dimensions after ITA_1 OVERLAP.

From this, it follows that if ec_2 satisfies (ITA_1, ITA_2, …, ITA_n) PNF, then t_{2j} and t_{2k} must satisfy (ITA_1, ITA_2, …, ITA_n) PNF, therefore it must not be the case that $t_{2j}.ITA_1$ MEETS $t_{2k}.ITA_1$. If they did MEET, then these tuples would be rearrangeable, and if they OVERLAPped in this dimension, then they would be intersectable.

So therefore we can conclude that for one of the two tuples t_{2j} and t_{2k}, it holds that

$$t_{2j}.ITA_1.TO < t_{2k}.ITA_1.FROM$$

Now consider the point $p(c_{1x}, c_2, \ldots, c_n)$ whose c_{1x} value is equal to $t_{2j}.ITA_1.TO$. This point is not covered by t_{2j}, nor by t_{2k}. But it is covered by t (the single tuple in ec_1 that represents the same box). So there must be some other tuple t_{2m} in ec_2 that does cover this particular point.

But then again, this tuple t_{2m} is subject to the very same reasoning as the tuple t_{2j} that we started out with, so there will have to be yet another tuple t_{2n} in ec_2 that will cover t_{2m}'s $ITA_1.TO$ value, etc. etc., ad infinitum.

Therefore, we can conclude that there is no finite set of tuples such that :

- none of them taken pairwise will violate (ITA_1, ITA_2, …, ITA_n) PNF,
- and all points $p(c_{1x}, c_2, \ldots, c_n)$, for some fixed set of $c_2, \ldots, c_n$ values and for all c_{1x} values in the range $t.ITA_1.FROM <= c_{1x} < t.ITA_1.TO$, will be covered by some tuple in that finite set of tuples.

This implies in turn that the entire set of points $p(c_{1x}, c_2, \ldots, c_n)$, for some fixed set of $c_2, \ldots, c_n$ values and for all c_{1x} values in the range $t.ITA_1.FROM <= c_{1x} < t.ITA_1.TO$, can only be covered by one single tuple in the relation/equivalence class covering that set of points. Hence, the ITA_1 attribute value for all tuples t_{2j} in c_2 must be equal to the ITA_1 interval value from t :

$$\text{FORALL } t_2, t_2 \in ec_2 : t_2.ITA_1 = t.ITA_1$$

Now we can repeat the same reasoning to show that a similar conclusion holds for the "second" dimension, ITA_2, and again and again to show that in fact a similar conclusion holds for all of the involved dimensions. But if that is true, then this simply means that

$$\text{FORALL } t_2, t_2 \in ec_2 : t_2 = t$$

and this simply means that c_2 has to be equal to the singleton c_1.

Hence this constitutes our proof that no "finite subdivision" of a box can possibly satisfy (ITA_1, ITA_2, …, ITA_n) PNF.

If two distinct tuple pairs t_1/t_2 and t_3/t_4, all belonging to the same {ITA_1, ITA_2, …, ITA_n} equivalence class, cover the same set of points, then only one of them can possibly satisfy (ITA_1, ITA_2, …, ITA_n) PNF.

Suppose, contrariwise, that such distinct tuple pairs both satisfy {ITA_1, ITA_2, …, ITA_n} PNF. This means that both t_1/t_2 and t_3/t_4 are neither intersectable nor rearrangeable. Not being intersectable implies that there is at least one dimension for which the interval values do not overlap, both between t_1/t_2 and t_3/t_4.

Two cases can occur. The interval values in t_1/t_2 for at least one such non-overlapping dimension, do not even MEET (i.e. they are AFTER or BEFORE each other in the Allen sense). In that case, the boxes are completely "disjoint", and the fact that the same set of points must also be covered by t_3/t_4, implies that t_3/t_4 must be equal to t_1/t_2.

Or it is the case that for all such non-overlapping dimensions, the interval values from t_1/t_2 do MEET. Once again two cases can be considered. There can be exactly one such non-overlapping dimension (between t_1/t_2) or there can be >1 of them. If there are >1 of them, then once again we do not have a case of "meeting tuples", and once again the fact that the same set of points must also be covered by t_3/t_4 implies that t_3/t_4 must be equal to t_1/t_2.

Hence the last case to consider is when t_1/t_2 have exactly one non-overlapping dimension where their attribute values MEET. Call that dimension ITA_m. T_1/t_2 were assumed to satisfy PNF, hence they are not rearrangeable, hence this means, for that ITA_m dimension, one of the following must hold :

- EXISTS j,j<m : $t_1.ITA_j$ <> $t_2.ITA_j$ (attribute values for some dimension preceding m must be unequal)
- EXISTS z,z>m,z<=n : NOT ($t_1.ITA_z$ OVERLAPS $t_2.ITA_z$) (attribute values for some dimension following m must not overlap)

Since we are considering the case with exactly one non-overlapping dimension, the latter bullet must necessarily be false, and it must be the case that between t_1/t_2, the attribute values for some dimension preceding m must be unequal (also implying that m cannot possibly be the very first dimension ITA_1).

So far so good, attempting to get further from here.

Take a point $p(c_1, c_2, …, c_m, …, c_n)$ on the "meeting place" of t_1/t_2. We have

$$c_m = t_1.ITA_m.TO = t_2.ITA_m.FROM$$

To be completed.

"Alternative" definitions for the USING operators

The foregoing chapters, defining (ITA_1, ITA_2, …, ITA_n) PNF and its properties, have answered the actual title question. But there is more to the problem than just that. TDATRM essentially defines what it calls "USING versions" for all the operators of the relational algebra (and for yet some other operators that are often overlooked in the USING context), by relying on the UNPACK operator, applying UNPACK to the arguments of the operator at hand, applying the operator to the "unpacked" relation(s), and then re-packing the result (if it concerns an operator that returns a relation). Since UNPACK is not available in the approach presented here, alternative definitions are necessary. In the following sections, we will inspect and discuss the various operators concerned, beginning with PACK() itself.

USING (ITA_1, ITA_2, …, ITA_n) PACK

In our "non-granular" approach, USING (ITA_1, ITA_2, …, ITA_n) PACK (r) is defined to return the unique relation that is the (ITA_1, ITA_2, …, ITA_n) PNF equivalent of its sole argument r. That PNF equivalent is the relation r' that :

- is in (ITA_1, ITA_2, …, ITA_n) PNF
- for each (ITA_1, ITA_2, …, ITA_n) equivalence class in it, covers the same set of points as the corresponding equivalence class in r.

Note that the "USING (ITA_1, ITA_2, …, ITA_n) PACK (r)" defined here is completely equivalent to TDATRM's "PACK r ON (ITA_1, ITA_2, …, ITA_n)". (Meaning that if "nongranular PACK" is applied to relations containing intervals over a type that is indeed ordinal[5], the tuples and their interval boundary values in the result will correspond one on one with the results from applying TDATRM's "granular PACK" on the same relation.)

And once again, this claim of equivalence might be accompanied by some kind of formal proof.

USING (ITA_1, ITA_2, …, ITA_n) EQUALS

In TDATRM, the 'USING' version of the relational equality operator tests relation values for "pointwise equality", so to speak. Looking back at cases 1 and 2 (let's call those relations case1 and case2, respectively) of the "Interlude with visualizations" section, the following expressions would yield the indicated truth values :

- case1 = case2 = FALSE
- USING (X,Y) ◄ case1 = case2 ► = TRUE

In our approach, the second expression would have to be written as the following equivalent :

- USING (X,Y) PACK(case1) = USING (X,Y) PACK(case2)

The question remains a bit open to what degree or extent these seeming differences are also "real differences", rather than mere differences in syntactic notation. Or iow, whether there is a real genuine need altogether for a "distinct operator definition" as per TDATRM. If such a need there is, then it is of course perfectly possible for the surface language to expose some syntactic shorthand as "being this operator" and under the covers implementing it as the equivalent expression.

USING (ITA_1, ITA_2, …, ITA_n) UNION

We define both a "tuple version" and a "relation version" of this operator. The "tuple version" takes two tuples as arguments, the "relation version" operates on two relations.

The relation version USING (ITA_1, ITA_2, …, ITA_n) UNION (r_1,r_2) is defined as being equivalent to USING (ITA_1, ITA_2, …, ITA_n) PACK (r_1 UNION r_2). Observe that the outermost operator in this equivalent expression being PACK(), the result of USING (ITA_1, ITA_2, …, ITA_n) UNION (r_1,r_2) is guaranteed to be in (ITA_1, ITA_2, …, ITA_n) PNF. Also observe that the property of associativity of UNION can be applied to define an associative version USING (ITA_1, ITA_2, …, ITA_n) UNION (r_1,r_2,r_3,…), and that such an associative version even validly extends to the case of no arguments at all (returning the empty relation of some heading that must then be specified explicitly in the syntax).

---

5 The "nongranular" approach from this document does not prohibit that !

The tuple version USING (ITA_1, ITA_2, …, ITA_n) UNION (t_1,t_2) is defined as being equivalent to USING (ITA_1, ITA_2, …, ITA_n) UNION (RELATION{t_1}, RELATION{t_2}). Informally, the tuple version thus "wraps the two tuples in two singleton relations" and then computes the "USING (ITA_1, ITA_2, …, ITA_n) UNION" of those two relations. Note that this "tuple version" thus returns a relation (not a single tuple), and that the returned relation must not necessarily be a singleton. Also note that because the "tuple version" is defined in terms of the "relation version", the "tuple version" too returns a result that is guaranteed to be in (ITA_1, ITA_2, …, ITA_n) PNF.

USING (ITA_1, ITA_2, …, ITA_n) INTERSECT

First, we define a "USING (ITA_1, ITA_2, …, ITA_n) INTERSECT" operator on two tuples.

For two tuples t_1 and t_2 that are not in the same {ITA_1, ITA_2, …, ITA_n} equivalence class, or are not {ITA_1, ITA_2, …, ITA_n} intersectable, the operator returns an empty relation.

For two tuples that are in the same {ITA_1, ITA_2, …, ITA_n} equivalence class, and are {ITA_1, ITA_2, …, ITA_n} intersectable, the operator returns a singleton relation whose only tuple has attribute values as follows :

- All attribute values other than (ITA_1, ITA_2, …, ITA_n) are as in t_1 and t_2.
- Attribute values for (ITA_1, ITA_2, …, ITA_n) are interval values with its ITA_x (x=1..n) components as follows :
  FROM MAX(t_1.ITA_x.FROM , t_2.ITA_x.FROM)
  TO MIN(t_1.ITA_x.TO , t_2.ITA_x.TO)

Observe that since this "tuple version" returns either an empty relation or a singleton relation, the result is always, by definition, in (ITA_1, ITA_2, …, ITA_n) PNF.

Then we can define a "USING (ITA_1, ITA_2, …, ITA_n) INTERSECT" operator that operates on two relations. The result of an invocation of this operator on two relations r_1 and r_2, is defined as the "USING (ITA_1, ITA_2, …, ITA_n) UNION" of all the relations obtained by invoking the formerly defined "USING (ITA_1, ITA_2, …, ITA_n) INTERSECT" operator on each possible tuple pair t_1 and t_2, where $t_1 \in r_1$ and $t_2 \in r_2$.

Observe that since this "relation version" returns the result coming from a "USING (ITA_1, ITA_2, …, ITA_n) UNION", that result is guaranteed to be in (ITA_1, ITA_2, …, ITA_n) PNF.

USING (ITA_1, ITA_2, …, ITA_n) MINUS

This operator too can be defined in a "tuple version" (taking two tuples as arguments) and in a "relation version" (taking two relations as arguments).

The "tuple version", with 'minuend' t_1 and 'subtahend' t_2, is defined to return a relation r as follows :

- If t_1 and t_2 are not in the same (ITA_1, ITA_2, …, ITA_n) equivalence class, then r contains just t_1
- If t_1 and t_2 are in the same (ITA_1, ITA_2, …, ITA_n) equivalence class, then r is the relation satisfying (ITA1, ITA2, …, ITAn) PNF such that all its tuples have the same values as t_1/t_2 for all of the attributes other than {ITA_1, ITA_2, …, ITA_n}[6], and furthermore

FORALL p(c_1, c_2, …, c_n) : p COVBY r <===> p COVBY t_1 AND NOT(p COVBY t_2)

Observe that the result is always guaranteed to be in (ITA_1, ITA_2, …, ITA_n) PNF, and this is by definition, whichever of the two cases applies.

6 Guaranteeing thereby that r will consist of only one (ITA_1, ITA_2, …, ITA_n) equivalence class

Likewise, the “relation version”, taking 'minuend' r_1 and 'subtahend' r_2, is defined to return a relation r that satisfies (ITA_1, ITA_2, …, ITA_n) PNF and furthermore :

- For all (ITA_1, ITA_2, …, ITA_n) equivalence classes c_1 in r_1 that don't have a corresponding equivalence class in r_2, r contains all the tuples that are needed to cover all the points covered by the tuples in c_1, that is, there will be an equivalence class c_r in r containing all the tuples that are the result of USING (ITA_1, ITA_2, …, ITA_n) PACK (c_1)[7][8].
- and for all (ITA_1, ITA_2, …, ITA_n) equivalence classes c_1 in r_1 that do have a corresponding equivalence class c_2 in r_2, r will have an (ITA_1, ITA_2, …, ITA_n) equivalence class c_r consisting of the set of tuples such that

FORALL p(c_1, c_2, …, c_n) : p COVBY c_r <===> p COVBY c_1 AND NOT(p COVBY c_2)

Observe that here too, the result is guaranteed to be in (ITA_1, ITA_2, …, ITA_n) PNF, by definition.

USING (ITA_1, ITA_2, …, ITA_n) JOIN

A “USING version” of JOIN must be defined much along the same lines as was done for INTERSECT.

First, we define a tuple version. Let t_a and t_b be two tuples. Let A be the set of attributes that are exclusive to t_a, B be the set of attributes that are exclusive to t_b, and C be the set of attributes that are common between t_a and t_b. If the set {ITA_1, ITA_2, …, ITA_n} is a subset of C[9], then the USING (ITA_1, ITA_2, …, ITA_n) JOIN of t_a and t_b is defined to return a relation that is either empty or a singleton. It is a singleton only if both of the following are satisfied :

- Between t_a and t_b, the values for all attributes in C but not in {ITA_1, ITA_2, …, ITA_n} are equal,
- Between t_a and t_b, all the values for attributes that are in {ITA_1, ITA_2, …, ITA_n} OVERLAP.

For singleton relations, the contained result tuple t_r has attribute values :

- same as in t_a for the attributes in A, and same as in t_b for the attributes in B,
- same as in t_a or t_b for the attributes that are in C but not in {ITA_1, ITA_2, …, ITA_n},
- {ITA_1, ITA_2, …, ITA_n} attributes get the value that is the interval intersection between the corresponding interval values from t_a and t_b, respectively.

The “relation version” for USING (ITA_1, ITA_2, …, ITA_n) JOIN can then simply be defined as it was done for USING (ITA_1, ITA_2, …, ITA_n) INTERSECT : the USING (ITA_1, ITA_2, …, ITA_n) JOIN of two relations r_1 and r_2, is defined as the “USING (ITA_1, ITA_2, …, ITA_n) UNION” of all the relations obtained by invoking the formerly defined “USING (ITA_1, ITA_2, …, ITA_n) JOIN” operator on each possible tuple pair t_1 and t_2, where $t_1 \in r_1$ and $t_2 \in r_2$.

Observe that the result of invoking a USING (ITA_1, ITA_2, …, ITA_n) JOIN is thus, once again, guaranteed to satisfy (ITA_1, ITA_2, …, ITA_n) PNF, both for the “tuple version” and the “relation version”.

7 Note that we cannot simply say “contains all the tuples in c_1, because r_1 is not guaranteed to satisfy (ITA_1, ITA_2, …, ITA_n) PNF, but we do want the result to be. So this invocation of PACK is needed.

8 Also note that the use of an equivalence class as the argument to PACK is not entirely invalid, seeing as such an equivalence class is itself also nothing more than a set of tuples.

9 It is an error if this condition is not satisfied. Alternatively, the definition could be extended to include an additional (outermost) PACK() invocation to handle the non-common interval-typed attributes to be packed on, but this would obfuscate the real point being made here, so the issue is deliberately side-stepped here.

USING (ITA_1, ITA_2, …, ITA_n) GROUP

TDATRM defines a 'USING' version for GROUP. The definition imposes the restriction that none of the packing attributes {ITA_1, ITA_2, …, ITA_n} can be "moved inside an RVA by the grouping", as this would give rise to a compile error on the outermost PACK() invocation of the equivalent expression. If this restriction were lifted, a solution would also have to be found for the unpleasant (and probably unwanted) side-effect that the interval-typed attribute values that are "moved inside an RVA", would all be unit intervals (as a consequence of the implied UNPACK() operation that is carried out first).

We simply define 'R USING (ITA_1, ITA_2, …, ITA_n) GROUP {GA_1,GA_2, …,GA_n} AS RVA_1' to be equivalent to precisely the expression :

- USING (ITA_1, ITA_2, …, ITA_n) PACK (R GROUP {GA_1,GA_2, …,GA_n} AS RVA_1) [10]

That said, it must also be noted that the U_GROUP operator of TDATRM produces a result that seems inachievable without the ability to UNPACK(). With TDATRM's U_GROUP, the individual point values deriving from the UNPACK() operation, are a part of the "grouping key", hence this influences the resulting RVA values in intricate ways : if we have a relvar R with value :

```
DURING  PERSON
[05-17) Hugh Darwen
[03-12) Chris Date
[07-21) Nikos Lorentzos
```

Then the TDATRM expression 'USING (DURING) ◄R GROUP {PERSON} AS PERSONS►' would yield :

```
DURING  PERSONS
[03-05) {Chris Date}
[05-07) {Chris Date, Hugh Darwen}
[07-12) {Chris Date, Hugh Darwen, Nikos Lorentzos}
[12-17) {Hugh Darwen, Nikos Lorentzos}
[17-21) {Nikos Lorentzos}
```

This result is very hard to come by, definitionally, if you don't have UNPACK ! Without UNPACK, a "dedicated" operator definition would have to be spelled out that facilitates obtaining this result.

The least desirable form of this definition, the algorithmical one, might look somewhat like :

- Take a tuple pair t_1,t_2 from the relation
- Construct tuples t_3,t_4,t_5 from t_1,t_2 representing the "intersecting" part and the two "differential" parts from t_1,t_2 (discarding any "empty ranges" that might occur and doing the proper "unioning" for the tuple representing the "intersecting" part)
- replace t_1,t_2 with t_3,t_4,t_5
- repeat recursively until no more t_1,t_2 pairs can be found that have a nonempty intersection.

This aspect and the definitional problem arising in our "nongranular" approach might be regarded as a "weakness", but there is a "weakness" in either approach : if a type is non-granulated, then in the TDATRM approach the corresponding interval type simply cannot and does not exist, and because of this, the TDATRM U_GROUP invocation that would produce such a result, is equally impossible.

10 Accepting the same (inevitable) limitation that {GA_1,GA_2, …,GA_n} and {ITA_1, ITA_2, …, ITA_n} must be disjoint.

USING (ITA_1, ITA_2, …, ITA_n) SUBSET/SUPERSET

In TDATRM, the 'USING' versions of the relational containment operators test relation values for “pointwise containment”, so to speak. Again looking back at cases 1 and 2 of the “Interlude with visualizations” section, the following expressions would yield the indicated truth values :

- case1 SUBSETOF case2 = FALSE
- case1 SUPERSETOF case2 = FALSE
- USING (X,Y) ◄ case1 SUBSETOF case2 ► = TRUE
- USING (X,Y) ◄ case1 SUPERSETOF case2 ► = TRUE

In our approach, the latter two expressions would have to be written as the following equivalents :

- (case1 USING (X,Y) MINUS case2) = RELATION{}[11]
- (case2 USING (X,Y) MINUS case1) = RELATION{}

Or the USING (…) SUBSET/SUPERSET constructs could be defined as syntactic shorthands for such equivalent expressions.

USING (ITA_1, ITA_2, …, ITA_n) UNGROUP

TDATRM defines a 'USING' version for UNGROUP[12]. Its usefulness is fairly limited though, since the packing attribute list cannot include any interval-typed attribute that “originates from the ungrouping operation”. If packing is to be done on an attribute that “originates from the ungrouping operation, then the 'USING' shorthand from TDATRM is not usable, and the overall expression is to be written exactly as such :

- PACK (R UNGROUP RVA_1) ON (ITA_1, ITA_2, …, ITA_n)

In our approach, we simply define 'USING (ITA_1, ITA_2, …, ITA_n) UNGROUP (R , RVA_1)' to be equivalent to precisely that expression :

- USING (ITA_1, ITA_2, …, ITA_n) PACK (UNGROUP (R , RVA_1))

USING (ITA_1, ITA_2, …, ITA_n) TCLOSE/DIVIDEBY

TDATRM does not define USING versions for these relational operators, and at least for TCLOSE the reason should be obvious.

The DIVIDEBY case, might be another matter though. Relational division was invented to cater for any queries that include a universal quantification as a restricting predicate. It is not unreasonable to imagine queries that include a predicate containing some kind of universal quantification over “points in time”, that are represented in the relations using interval-typed values. “Get me all the players that were on the pitch throughout all the time that the match was being led by the substitute referee”, or some such. It might be interesting to investigate such problems in more detail, however, TDATRM hasn't done so, and neither do we.

USING (ITA_1, ITA_2, …, ITA_n) RENAME/PROJECT

In TDATRM, the USING version for these operators amounts to nothing more than just a nested invocation of PACK on an invocation of the operator at hand :

- USING (ITA_1, ITA_2, …, ITA_n) PACK (RENAME/PROJECT (… , …))

---

11 Actually, the literal specification of the empty relation here is invalid because it lacks a specification of a heading. We've omitted it here for convenience and because it does not materially affect the basic idea.

12 But admits at the very start of the section where it defines this operator that the primary reason is “completeness”.

With RENAME, in either approach, attention must be paid to the case where one of the ITA attributes also appears in the RENAME list. Many would yield a syntactically invalid expression, but some intricate cases are possible, e.g.

USING (ITA_1 ITA_2) ◄ R RENAME ITA_1 AS ITA_2, ITA_2 AS ITA_1 ►

With PROJECT, in either approach, it cannot be the case that any one of the ITA attributes gets projected away.

USING (ITA_1, ITA_2, …, ITA_n) RESTRICT/EXTEND

The TDATRM version of these operators, which includes an implicit UNPACK of the restricted/extended expression, has a peculiar side-effect as a consequence of this implicit UNPACK. If the restrict expression (or one of the extend expressions) include expressions that denote some function of one of the ITA intervals, (e.g. LENGTH(ITA_x), the number of points in an interval value that is being unpacked/packed on), then because of the UNPACK, under the TDATRM approach such expressions will only get to see unit intervals, and not the interval values that actually appear in the relation being U_restricted/U_extended. This causes results that might be perceived as counterintuitive, such as

USING IV ◄ RESTRICT RELATION {TUPLE{IV 5-8}} WHERE IV OVERLAPS 7-10 ►
RELATION {TUPLE{IV 7-8}}

USING IV ◄ EXTEND RELATION {TUPLE{IV 5-8}} ADD LENGTH(IV) AS LEN ►
RELATION {TUPLE{IV 5-8 LEN 1}}

USING IV ◄ RELATION {TUPLE{IV 5-8}} WHERE LENGTH(IV) > 1 ►
RELATION {}

In such cases, the USING shorthand as per TDATRM is unusable and the user is forced to explicitly write out the nesting of the expressions himself, e.g. :

(PACK (RELATION {TUPLE{IV 5-8}TUPLE{IV 7-18}}) ON IV) WHERE LENGTH(IV) > 1
RELATION {TUPLE{IV 5-18}}

In our approach without UNPACK, we achieve exactly that by simply defining a "USING … RESTRICT/EXTEND" to be the obvious sequence/nesting of the RESTRICT/EXTEND at hand within an invocation of USING … PACK :

- USING (ITA_1, ITA_2, …, ITA_n) PACK (RESTRICT/EXTEND (… , …))

USING (ITA_1, ITA_2, …, ITA_n) SUMMARIZE

Two remarks need to be made in connection with defining a USING version of SUMMARIZE.

The first is a repetition of the remark that applies also to RESTRICT and EXTEND. With SUMMARIZE, the summary expressions can contain references to one of the ITA attributes. Because of that, and TDATRM's implicit UNPACK(), the summary expressions will "get to see" unit interval values only, and that may or may not influence the result in intricate ways, meaning there might be a serious pitfall to stumble into for the user.

The second is a repetition of the final remark that was made in the discussion of GROUP. Because just as with GROUP, SUMMARIZE … BY … and SUMMARIZE … PER … carry a notion of "tuple grouping" of some sort, it might be the case that therefore certain summarizations are in fact impossible to achieve unless the underlying ordered type is ordinal, or the "non-granular" version of U_GROUP is effectively defined as roughly sketched in the relevant section on USING … GROUP.

U_constraints

TDATRM also pays attention to the support of uniqueness constraints that include interval-typed attributes, and on inclusion dependencies between relvars (“foreign key constraints”) that include such attributes. As was the case with some of the previously discussed relational algebra operators such as equality and JOIN, we need to be able to distinguish between a “pointwise” and an “as-is” treatment of the interval values appearing in such constraints.

U_keys

Let R_1 be a relvar (k_1,k_2,d_1,d_2) where {k_1,k_2} is the key, k_2 is interval-typed but k_1 is not, and {d_1,d_2} non-key attributes.

TDATRM achieves its support for “keys in pointwise modus” using what it calls “U_keys”, denoted by the syntactic construct 'WHEN UNPACKED ON {…} THEN KEY {…}'. While the (usual) syntactic declaration 'KEY {k_1,k_2}' would allow different-but-overlapping k_2 interval values in distinct tuples whose key attribute values are otherwise completely equal, the WHEN UNPACKED ON {k_2} THEN KEY {k_1,k_2} variant wouldn't. The relation value { {"A", [4-8], 1, 7} {"A", [5-9], 1, 8} } would satisfy the “as-is” key, but wouldn't satisfy the “WHEN UNPACKED” key, because there are two distinct tuples both covering the k_2 point values 5,6,7 and 8. Modulo any non-interval attributes that are also part of the key, “WHEN UNPACKED” keys ensure that each combination of point values in the (attributes of the) key, is associated with exactly one combination of non-key attribute values.

Note that the number of distinct possible WHEN UNPACKED ON … specifications is 2^n [13], with n the number of interval-typed attributes in the relvar. This includes the “degenerate” WHEN UNPACKED ON {}, which is equivalent to the “regular” specification for a “plain key”, the “as-is” modus of treatment, that is.

Since we're proposing an approach without UNPACK, how can something equivalent be achieved ? Well, all we have to do is observe the equivalent longhand database constraint declaration for the commonly known shorthand (in our example) 'KEY {k_1, k_2}'. This longhand can be constructed as follows :

- RENAME all the attributes that are not in the key, to a name not appearing in the relvar. Say, d_1 to e_1 and d_2 to e_2.
- JOIN the result of that RENAME to the relvar itself (this is thus a JOIN over the full key).
- If the key values are really unique, then this JOIN will have the property that in all its result tuples, the d_1/e_1, d_2/e_2, … values will be equal, pairwise.
- RESTRICT the result of this JOIN to the tuples for which any of the d_1/e_1, d_2/e_2, … values are not equal, pairwise.
- A relation satsifying the key will thus produce an empty relation after this fourth step, and a non-empty result here can only come from a relation that violates the key. Hence an equivalent database constraint expression for enforcing the very same key is :

```
IS_EMPTY( ( <relvar> JOIN (<relvar> RENAME …) ) WHERE … )
```

The “UNPACK-less” equivalent of TDATRM's 'WHEN UNPACKED ON … THEN …', should now be obvious : just replace the JOIN by the appropriate USING(…) JOIN, as previously defined.

13 But also note that the number of keys is not limited to this number. There can be >1 key for the same WHEN UNPACKED ON … spec !

Of course none of this is to say that this is also the way in which these USING(…) keys should be defined by the user (the surface language is at complete freedom to provide whatever shorthands are useful and appropriate), nor that this is also the way in which the enforcement algorithm must be implemented.

As for the declaration in the language, the "WHEN UNPACKED ON" syntax can be used as such, for example[14]. As for the enforcement algorithm, the one for an "as-is" key is likely to boil down to a simple search for

<relvar> WHERE $k_1 = k_{1n}$ AND $k_2 = k_{2n}$ AND ($d_1 <> d_{1n}$ OR $d_2 <> d_{2n}$)

and for a "pointwise" key it is likely to boil down to a search for

<relvar> WHERE $k_1 = k_{1n}$ AND k_2 OVERLAPS k_{2n} AND ($d_1 <> d_{1n}$ OR $d_2 <> d_{2n}$)[15].

These implementation strategies apply completely unaltered to the non-granular approach as well.

U_foreign keys

As for U_foreign keys, it should be sufficient to note that an equivalent longhand database constraint expression for these is

```
IS_EMPTY ( <referencingrelvar> SEMIMINUS[16]
( ( <referencedrelvar> RENAME … ) {fk1, fk2, …, fkn} ) )
```

The RENAME is possibly needed to ensure matching between referencing/referenced attributes if their names are different between referencing and referenced relvar, the projection is needed if attributes in the referenced relvar match attributes in the referencing relvar but are not part of the referenced attribute set.

"Pointwise" foreign keys in our nongranular approach now amounts to nothing more than replacing the "plain" SEMIMINUS in there by the appropriate USING(…) SEMIMINUS.

As with unique keys, syntactic shorthands could be designed in the language to declare these things conveniently, e.g.

<referencingrelvar> {fk_1, fk_2, ..., fk_n} REFERENCES <referencedrelvar> [{rfk_1, rfk_2, ..., rfk_n}]
[WITH {ifk_1, ifk_2, ..., ifk_n} POINTWISE]

14 Though retaining the use of the keyword 'UNPACKED' would of course admittedly be seriously odd for interval types that have no notion of UNPACK.

15 Both these examples are slightly simplified to disregard the scenario of set-level inserts, when violating tuples are both in the set being inserted.

16 NOT MATCHING in TDATRM

Assessing (ITA_1, ITA_2, …, ITA_n) PNF given two tuples

Determining whether or not two distinct tuples satisfy (ITA_1, ITA_2, …, ITA_n) PNF can be done with the following algorithm :

```
boolean isPNF(t1,t2,dimensions) {
   // parameters : distinct tuples t1 and t2,
   //                 belonging to the same (dimensions) equivalence class
   //               a list of dimensions
   // the pseudocode below "plays loosely", somewhat, with the dimensions.
   //   the dimensions are used in various ways which might not be directly
   //       available in any given real programming language :
   //   (a) as an argument to obtain attribute values from t1/t2,
   //   (b) in magnitude comparisons relating to their relative position in the
   //       dimensions list
   boolean allDimensionsOverlap = true;
   boolean dimensionsRequiringSubsequentNonOverlap = false;
   int firstUnequalDimension = Integer.MAX_VALUE;

   for each dimension in dimensions {
      intervalValue1 = t1.dimension;
      intervalValue2 = t2.dimension;
      if (intervalValue1 <> intervalValue2 && firstUnequalDimension > dimension) {
         firstUnequalDimension = dimension;
      }
      boolean thisDimensionOverlaps = intervalValue1 OVERLAPS intervalValue2;
      allDimensionsOverlap &= thisDimensionOverlaps;
      dimensionsRequiringSubsequentNonOverlap &= thisDimensionOverlaps;

      if (firstUnequalDimension == dimension) {
         // if the firstUnequalDimension OVERLAPS, then a subsequent non-overlap is
         //   still needed in order to make allDimensionsOverlap false.  So we don't
         //   need to bother with that case here
         // if the firstUnequalDimension values are completely disjoint (AFTER or BEFORE)
         //   then alldimensionsOverlap is already made false, and we can in fact already
         //   leave the loop here.  If the number of dimensions gets to be big,
         //   it's probably a good idea to   return true;   immediately here.
         // if the firstUnequalDimension MEETS, then a non-overlap in one of the
         //   subsequent dimensions is still required
         if (intervalValue1 MEETS intervalValue2) {
            dimensionsRequiringSubsequentNonOverlap = true;
         }
      }
   }

   return !allDimensionsOverlap & !dimensionsRequiringSubsequentNonOverlap;
}
```

To be noted : this algorithm does not cover the case of empty packing lists (this case contradicts the precondition of the tuples being both distinct and belonging to the same {} equivalence class).

Variations on the theme can be, e.g., the algorithm not returning a simple boolean, but instead a value of some enumeration enumerating the possible PNF states and causes of violation :

```
PNFEnum assessPNF(t1,t2,dimensions) {
   …
   return computePNFState(allDimensionsOverlap,dimensionsRequiringSubsequentNonOverlap);
}

PNFEnum enum {
   Satisfied, Intersectable, Rearrangeable;
}
```

(and where computePNFState() is the obvious factory method for computing the appropriate PNFEnum value from the given booleans).

Repairing violations of (ITA_1, ITA_2, …, ITA_n) PNF

One obvious question that now comes about, is whether if in a relation we find a tuple pair that does not satisfy some PNF, can we re-arrange or replace those tuples by other ones, which convey the same information (meaning they cover the same set of points in their equivalence class), but which, unlike the original two tuples, are indeed (all of them pairwise) in that PNF ? (The name we used for the state of being "re-arrangeable" suggests as much, of course.)

The definition of being in (ITA_1, ITA_2, …, ITA_n) PNF said that it required the tuples t_1 and t_2[17] to be "neither intersectable nor rearrangeable". Hence violating the condition for being in (ITA_1, ITA_2, …, ITA_n) PNF is equivalent to being "either intersectable or rearrangeable". And it was already observed that "intersectable precludes rearrangeable" (and of course vice versa). Hence there are two separate cases of "violation" that we can address independently.

It will be clear that if we find two tuples to be in violation of (ITA_1, ITA_2, …, ITA_n) PNF, there is nothing we can do about that by manipulating attribute values in them outside of the set of packing attributes {ITA_1, ITA_2, …, ITA_n} ("moving a tuple to another equivalence class", loosely speaking). Not if we want the tuples to keep conveying the same information set. Hence any differences between the "original" tuples and the "replacing" tuples will only be in the {ITA_1, ITA_2, …, ITA_n} attribute values.

T_1 and t_2 are {ITA_1, ITA_2, …, ITA_n} intersectable

Informally, this means that there is some nonemtpy set of points (c_1, c_2, …, c_n) in n-dimensional space that "are part of both the box denoted by t_1 and the box denoted by t_2" (that nonempty set of points is itself a box).

The repair tactic here is fairly obvious :

- Compute the "USING (ITA_1, ITA_2, …, ITA_n) intersection" between t_1 and t_2. The result is a (singleton relation with a) tuple t_3.
- Compute the "USING (ITA_1, ITA_2, …, ITA_n) difference" between t_1 (or t_2)[18] and t_3. The result is a set of tuples rts (replacing tuple set).

The set of tuples that can replace t_1 and t_2 is the union of :

- all the tuples in rts
- t_2 (or t_1, if the alternative choice for t_2 was made in step 2)

This replacing set of tuples satisfies the rule of "not having any pair of tuples that are intersectable". This property follows from the definition of the "USING (ITA_1, ITA_2, …, ITA_n) difference" operator. The set of tuples resulting from this repair can, however, still have violations of (ITA_1, ITA_2, …, ITA_n) PNF on account of some tuple pairs still being rearrangeable.

The computation of rts might be a comparatively expensive operation, since it involves invoking the "USING (ITA_1, ITA_2, …, ITA_n) difference" operator. In the case when only one dimension is involved in the packing attribute list (i.e. t_1 and t_2 are {ITA_1} intersectable, meaning the ITA_1 attribute values OVERLAP), the repair operation can be simplified to computing just the {ITA_1} UNION of t_1 and t_2.

---

17 The assumption still holds that those tuples are in the same equivalence class, which is also obvious from the fact that we are talking of pairs of tuples that are in violation of a PNF, and such violation can only be caused by tuples in the same equivalence class.

18 As the illustrations to follow will show, the choice made here can have a rather drastic effect on the overall time and work needed to complete the repair. However, no effort is made here to explore the possibilities of making the optimal choice at this point of the process.

Worked-out examples for “repairing” intersectable tuples

Two examples are provided, both in 2D space for purposes of visualization. The examples will handle (X,Y) PNF in particular, but the difference with (Y,X) is not that great.

Example 1

```
+--------------+           X     Y
!              !         ----- -----
!      t1      !          1-11  1-12    (t1)
!              !          4- 8  4- 7    (t2)
!    +----+    !
!    ! t2 !    !
!    +----+    !
!              !
+--------------+
```

The “USING (ITA_1, ITA_2, …, ITA_n) intersection” between t_1 and t_2 is t_2 itself. The resulting tuple t_3 is thus equal to t_2.

The next step is to compute the “USING (ITA_1, ITA_2, …, ITA_n) difference” between t_1 and t_3. This results in a four-tuple relation (subtrahend tuple t_2 not displayed) :

```
+--------------+           X     Y
!              !         ----- -----
!      t5      !          1-11  1- 4
!              !          1-11  7-12
+----+----+----+          1- 4  4- 7
! t6 !    ! t7 !          8-11  4- 7
+----+----+----+
!      t4      !
+--------------+
```

Unioning this (straightfoward union, not the using version) with the tuple from the subtrahend relation (t_2), will give a case for “repairing” tuple pairs that are rearrangeable.

Example 2

In our second example, we simply switch minuend and subtrahend of the first example :

```
+--------------+           X     Y
!              !         ----- -----
!      t2      !          4- 8  4- 7    (t1)
!              !          1-11  1-12    (t2)
!    +----+    !
!    ! t1 !    !
!    +----+    !
!              !
+--------------+
```

The “USING (ITA_1, ITA_2, …, ITA_n) intersection” between t_1 and t_2 is t_1 itself. The resulting tuple t_3 is thus equal to t_1.

The next step is to compute the “USING (ITA_1, ITA_2, …, ITA_n) difference” between t_1 and t_1. This results in the empty relation. Unioning this with the subtrahend tuple (t_2), just gives us the subtrahend itself, and that's already the final result of the overall repair.

T_1 and t_2 are (ITA_1, ITA_2, …, ITA_n) rearrangeable

The repair tactic for this case is more complex. The formal properties for being rearrangeable imply that there is some dimension m for which $t_1.ITA_m$ MEETS $t_2.ITA_m$. They also imply that for all dimensions e,e<m, $t_1.ITA_e$ EQUALS $t_2.ITA_e$, and that for all dimensions o,o>m, $t_1.ITA_o$ OVERLAPS $t_2.ITA_o$.

If m=n (no o dimensions – and note that this is by definition the case if n=1, i.e. only a single packing attribute is involved), then t_1 and t_2 can be replaced by the single tuple that is the result of computing the USING (ITA_1, ITA_2, …, ITA_n) UNION of t_1 and t_2. This is a tuple that has all but the ITA_n attribute values taken from t_1 (or t_2, they're equal anyway), and as ITA_n attribute value the range that has the minimum of the two lower bounds, and the maximum of the two upper bounds. Observe that the replacement being a singleton, this replacement is by definition one that satisfies (ITA_1, ITA_2, …, ITA_n) PNF.

If m<n, then :

- For all o dimensions, compute the interval value that is the intersection between the two overlapping interval values $t_1.ITA_o$ and $t_2.ITA_o$. This yields (n-m) interval values iv_{co}.
- Build a tuple value $t1_{co}$ in which all attribute values except the o dimensions are equal to t_1's attribute values, and in which the attribute values for the o dimensions are the iv_{co} values from step 1. Build a tuple $t2_{co}$ in exactly the same way, but using t_2.
- Compute the USING (ITA_1, ITA_2, …, ITA_n) MINUS between t_1 and $t1_{co}$. This is a set of tuples rts_1. Compute the USING (ITA_1, ITA_2, …, ITA_n) MINUS between t_2 and $t2_{co}$. This is a set of tuples rts_2.
- The tuples $t1_{co}$ and $t2_{co}$ are mergeable. Proof: all attribute values other than (ITA_1, ITA_2, …, ITA_n) are equal, because the tuples are in the same (ITA_1, ITA_2, …, ITA_n) equivalence class. All attribute values for the e dimensions are also equal, because that is a required condition for being rearrangeable. All the attribute values for the o dimensions are also equal, because that's how the tuples were built. Hence there is exactly one attribute value between $t1_{co}$ and $t2_{co}$ that differs, and that is the attribute value for the m dimension, and of that one we know that the values MEET.

The replacing set of tuples for t_1 and t_2, then, is the union of :

- all the tuples in rts_1
- all the tuples in rts_2
- the USING (ITA_1, ITA_2, …, ITA_n) UNION of $t1_{co}$ and $t2_{co}$ (call this rt_{co}).

We now show that the result of this procedure indeed satisfies (ITA_1, ITA_2, …, ITA_n) PNF.

- rts1 by itself satisfies the PNF, being the result of a USING (ITA_1, ITA_2, …, ITA_n) MINUS.
- rts2 by itself satisfies the PNF for the same reason.
- Likewise for rt_{co}, observing that it must necessarily be a singleton.
- No tuple pair $t_{1'}/t_{2'}$ with $t_{1'} \in rts_1$ and $t_{2'} \in rts_2$ is (ITA_1, ITA_2, …, ITA_n) intersectable. Proof : $t_{1'}$ and $t_{2'}$ cover only points from the original tuples t_1 and t_2, respectively. These were known to be disjoint (meeting) in the m dimension, hence tuples $t_{1'}$ and $t_{2'}$ can still possibly have MEETing interval values for the m dimension, but never an OVERLAPping one.

- No tuple pair $t_{1'}/t_{2'}$ with $t_{1'} \in rts_1$ and $t_{2'} \in rts_2$ is (ITA_1, ITA_2, …, ITA_n) rearrangeable. Proof : from the previous bullet, we know that between $t_{1'}$ and $t_{2'}$, the ITA_m value can still possibly MEET, but never OVERLAP (meaning those values could also be AFTER or BEFORE one another, in the Allen sense). If they are effectively AFTER or BEFORE one another, then these tuples will not be rearrangeable because of that. Remains to be considered, therefore, the case when the ITA_m values do MEET between $t_{1'}$ and $t_{2'}$. For these two tuples to be rearrangeable, it would take all ITA_e (e<m) attribute values to be equal, and all ITA_o (o>m) values to overlap. But this is impossible, because all the points covered by the resulting range values have been "moved into rt_{co}", so to speak. It is therefore impossible for rts_1 as well as rts_2 to cover any such point. Ergo, we can conclude that $t_{1'}$ and $t_{2'}$ are not (ITA_1, ITA_2, …, ITA_n) rearrangeable.
- Rt_{co} is not (ITA_1, ITA_2, …, ITA_n) intersectable with any tuple in rts_1 or rts_2. This is because rts_1 covers only points not covered by $t1_{co}$ (by definition of MINUS) and also not covered by $t2_{co}$ (because t_1 and t_2 were known to be not interesectable (hence the boxes "disjoint") as the very precondition for this particular repair case), and a similar reasoning applies to rts_2.
- Rt_{co} is not (ITA_1, ITA_2, …, ITA_n) rearrangeable with any tuple in rts_1/rts_2. This can be shown by observing two phenomena :

  1. Looking at the ITA_m value in rt_{co}, we can observe that this value will overlap with, but never be equal to, the ITA_m value for any tuple in rts_1 and rts_2. This is because the m dimension is exactly where $t1_{co}$ and $t2_{co}$ were mergeable, and hence the ITA_m value in rt_{co} will cover the entire range from the lowest of the lower bounds for m in t_1/t_2, to the highest of the upper bounds for m in t_1/t_2. Meaning that the m dimension value of any tuple in rts_1/rts_2 will be properly contained in the value for the m dimension in rt_{co}, and thus this means indeed that those values overlap, but certainly aren't equal.

  2. m<n. From this, it follows that the ITA_n dimension had an overlap between t_1/t_2. (There can be more such dimensions, but the point is there is at least one.) Can it still be the case that for any tuple in rts_1/rts_2, all the original 'o' dimensions overlap with rt_{co} ? No, it can't. For the original 'e' dimensions (ITA_1, ITA_2, …, ITA_{m-1}), the values in t_1, t_2 and rt_{co} are all equal. Hence the range values in any tuple in rts_1/rts_2 can only be sub-ranges (not necessarily proper) of the ones in rt_{co}. The same holds for the original 'm' dimension, because the range value there in rt_{co} is the union of those in t_1/t_2. So all the original 'e' dimension and the 'm' dimension have only sub-ranges appearing as values in rts_1/rts_2. But since we already know that no tuple in rts_1/rts_2 is intersectable with rt_{co}, there must therefore be at least one dimension that does not overlap between a tuple in them and rt_{co}. And since we have established that such a dimension cannot possibly be one of the original 'e' or 'm' dimensions, it must therefore be one of the original 'o' dimensions. Combined with point 1, this establishes that between rt_{co} and any tuple in rts_1/rts_2, there must be some dimension that does not overlap, after one that does (the original 'm' dimension). This makes it impossible for any such tuple pair to still be rearrangeable.

All these observations taken together constitute proof that no pair of distinct tuples taken from any of the three components of the overall replacing tuple set (the result of the three-way UNION) violates the PNF, ergo the UNION satisfies the PNF.

Worked-out examples for “repairing” rearrangeable tuples

The next few sections are intended to illustrate how the foregoing algorithm indeed addresses all possible cases of PNF violations. We'll start with as first example, case 4 from the “case studies with visualizations”. Each subsequent example will be the result that was the outcome of the previous one. The examples taken individually illustrate how the cases of mergeable resp. rearrangeable tuple pairs are handled. The complete chaining together of the examples serves the purpose of illustrating how this same procedure, applied iteratively, can give us an algorithm for doing the PACK() operation on a complete relation. As in the worked-out examples for “repairing” intersectable tuples, we'll be assuming the desired target PNF is (X,Y).

<u>Example 1 (rearrangeable tuples)</u>

```
+----+---------+
!    !         !
!    !   t5    !
! t2 !         !
!    +----+----+
!    ! t3 !    !
+----+----+ t4 !
!   t1    !    !
+---------+----+
```

X	Y	
1- 8	1- 4	(t_1)
1- 4	4-12	(t_2)
4- 8	4- 7	(t_3)
8-11	1- 7	(t_4)
4-11	7-12	(t_5)

As previously indicated in the “case studies with visualizations”, the tuple pair t_2/t_3 is not in (X,Y) PNF[19]. The repair procedure replaces this tuple pair as follows :

The meeting dimension is X, which is not the last, hence the procedure with the four bullet points is applied.

The first bullet point computes the “overlap” in the Y dimension, which is 4-7.

The second bullet point uses this value to create the tuples 1-4,4-7 and 4-8,4-7.

The third bullet “subtracts” these two tuples from the original tuples, respectively, yielding a singleton relation 1-4,7-12 and an empty relation, respectively.

The fourth bullet point computes the union of 1-4,4-7 and 4-8,4-7, yielding 1-8,4-7.

Thus the replacing tuple set for t_2/t_3 contains two tuples : 1-4,7-12 and 1-8,4-7, yielding overall :

<u>Example 2 (mergeable tuples in Y)</u>

```
+----+---------+
!    !         !
!    !   t5    !
! t6 !         !
+----+----+----+
!    t7   !    !
+---------+ t4 !
!   t1    !    !
+---------+----+
```

X	Y	
1- 8	1- 4	(t_1)
1- 4	7-12	(t_6)
1- 8	4- 7	(t_7)
8-11	1- 7	(t_4)
4-11	7-12	(t_5)

The tuple pair t_1/t_7 is not in (X,Y) PNF. The repair procedure replaces this tuple pair as follows :

The meeting dimension is Y, which is indeed the last, hence these tuples are replaced with the USING (X,Y) UNION of the two, which is 1-8,1-7, yielding overall :

19 Note that the choice for t_2/t_3 is essentially arbitrary. Other pairs of rearrangeable tuples are t_4/t_3, t_2/t_5 and t_4/t_1.

Example 3 (mergeable tuples in X)

```
+----+---------+          X     Y
!    !         !        ----- -----
!    !   t5    !         1- 8  1- 7   (t8)
! t6 !         !         1- 4  7-12   (t6)
+----+----+----+         8-11  1- 7   (t4)
!         !    !         4-11  7-12   (t5)
!    t8   ! t4 !
!         !    !
+---------+----+
```

The tuple pair t_4/t_8 is not in (X,Y) PNF. The repair procedure replaces this tuple pair as follows :

The meeting dimension is X, which is not the last, hence the procedure with the four bullet points is applied.

The first bullet point computes the "overlap" in the Y dimension, which is 1-7.

The second bullet point uses this value to create the tuples 1-8,1-7 and 8-11,1-7. Observe that these tuples are both identical to the original tuples.

The third bullet "subtracts" these two tuples from the original tuples, respectively, yielding an empty relation twice.

The fourth bullet point computes the union of 1-8,1-7 and 8-11,1-7, yielding 1-11,1-7.

The overall result after this step looks like :

```
+----+---------+          X     Y
!    !         !        ----- -----
!    !   t5    !         1-11  1- 7   (t9)
! t6 !         !         1- 4  7-12   (t6)
+----+---------+         4-11  7-12   (t5)
!              !
!      t9      !
!              !
+--------------+
```

An algorithm for re-packing relations

An algorithm for computing the unique (ITA_1, ITA_2, …, ITA_n) PNF form of any relation r would then consist of the following steps :

- Partition the relation r in its (ITA_1, ITA_2, …, ITA_n) equivalence classes
- For each equivalence class c, consider all possible pairs of distinct tuples in it and assess whether the tuple pair satisfies (ITA_1, ITA_2, …, ITA_n) PNF or not, and if not, assess the reason why (they must be either {ITA_1, ITA_2, …, ITA_n} intersectable or (ITA_1, ITA_2, …, ITA_n) rearrangeable – with the latter including the case of mergeable).
- If the tuple pair does not satisfy (ITA_1, ITA_2, …, ITA_n) PNF, apply the appropriate repair tactic, remove the two tuples from c and add the result from applying the repair tactic to c.
- Tuples newly added to c in this fashion give rise to a new round of considering tuples from c pairwise. This is repeated iteratively until no more repairs are needed.

A proof is required that this algorithm halts. i.e. that after several rounds of applying repair tactic, we cannot ever end up in a situation where the result of the repair tactic creates new PNF violations with some other tuple in the equivalence class, and repairing that gives new violations with tuples that were the result from the first repair tactic, and so on ad infinitum.

Possibly related to this, another proof is needed that the order in which tuple pairs are considered, is immaterial. I.e. that the "choice points" as illustrated in example 1 (where we had 4 tuple pairs that were all rearrangeable) does not materially affect the final outcome.

If all those proofs are delivered, then the "until no more repairs are needed" will guarantee us that the resulting relation satisfies the concerned PNF (because that's how PNF was initially defined – see "The definition we're aiming to achieve"), that the algorithm is a suitable one to implement PACK(), and the uniqueness proof will then guarantee us additionally that the algorithm is deterministic.

A last issue to be addressed is the following "recursive dependency" between the algorithms suggested here :

- The suggested algorithm for PACK() depends on the repair tactic
- Some of the repair tactics depend on invocations of USING (…) MINUS
- USING (…) MINUS is required to return a relation that satisfies the PNF
- And might therefore itself be dependent on PACK() …

This is only a very basic and crude description of the algorithm, whose time complexity, in the form as stated, is roughly[20] O(n^2) in the number of tuples per equivalence class. Significant refinements and improvements may be possible and prove effective, but these are not investigated in this document.

Possible further generalizations to interval types

In closed-open notation, it is sometimes impossible to denote an interval value that consists of a single point. Interval values such as [5-5] would have to be denoted as [5-6), assuming 6 is the value that comes "NEXT()" after the value 5. But this is impossible precisely in the cases that we wanted to address in this document, i.e. when there is no NEXT().

The next few sections present a possible way to address this problem, by a further generalization to some of our preliminary definitions.

Interval types & values

The representation of interval values so far, consisted of two components, FROM and TO, with the FROM bound being implicitly considered as being included in the range, and the TO bound implicitly considered as excluded. The generalization is to provide 2 additional components that explicitly specify the inclusion or exclusion of each bound, thus giving rise to a possrep with 4 components :

- The components FROM and TO as before,
- A component FINCL, a boolean value that is *true* if the FROM bound is included in the range, and *false* otherwise,
- A component TINCL, a boolean value that is *true* if the TO bound is included in the range, and *false* otherwise.

20 Additional time complexity obviously derives from the number of times repair tactics have to be applied, as this effectively increases the total number of tuple pairs that are to be ultimately considered, overall.

In order to prevent “empty intervals”, a type constraint might have to be imposed preventing interval values whose FROM and TO components are equal, and whose FINCL and TINCL components are not both true.

OVERLAPS

The expression 'i_1 OVERLAPS i_2', an invocation of the operator OVERLAPS(it,it), where i_1 and i_2 are expressions denoting interval values of the same interval type it, is defined to be equivalent to the expression

$$(i_1.\text{FROM} < i_2.\text{TO OR } (i_1.\text{FROM} = i_2.\text{TO AND } i_1.\text{FINCL=true AND } i_2.\text{TINCL=true}))$$
$$\text{AND}$$
$$(i_2.\text{FROM} < i_1.\text{TO OR } (i_2.\text{FROM} = i_1.\text{TO AND } i_2.\text{FINCL=true AND } i_1.\text{TINCL=true}))$$

MEETS

The expression 'i_1 MEETS i_2', an invocation of the operator MEETS(it,it), where i_1 and i_2 are expressions denoting interval values of the same interval type it, is defined to be equivalent to the expression

$$(i_1.\text{FROM} = i_2.\text{TO AND } i_1.\text{FINCL} <> i_2.\text{TINCL})$$
$$\text{OR}$$
$$(i_2.\text{FROM} = i_1.\text{TO AND } i_2.\text{FINCL} <> i_1.\text{TINCL})$$

Covered points

The definitions used for a “box” must be refined to cater for the inclusion of the bounds, which is now variable. We say the “box” for this tuple is the solid body in n-dimensional space consisting of all the points (c_1, c_2, …, c_n) such that :

- iv_1.FROM < c_1 OR (iv_1.FROM = c_1 AND iv_1.FINCL)
- c_1 < iv_1.TO OR (c_1 = iv_1.TO AND iv_1.TINCL)
- iv_2.FROM < c_2 OR (iv_2.FROM = c_2 AND iv_2.FINCL)
- c_2 < iv_2.TO OR (c_2 = iv_2.TO AND iv_2.TINCL)
- …
- iv_n.FROM < c_n OR (iv_n.FROM = c_n AND iv_n.FINCL)
- c_n < iv_n.TO OR (c_n = iv_n.TO AND iv_n.TINCL)

This definition was used in the “no finite subdivision of a box can satisfy PNF” proof. The proof needs to be revisited in light of this refined definition.

Consequences on our definition for PNF

Since the definition for packed normal form depends exclusively on [definitions for] OVERLAPS and MEETS, the two revised definitions just given for OVERLAPS and MEETS, suffice to provide a way for defining packed normal form of relations that have interval values with explicit components indicating the inclusion-or-not of each boundary value of the range.

The definitions of the USING (ITA_1, ITA_2, …, ITA_n) operators

We discuss here the impact on the definitions we gave for the USING (ITA_1, ITA_2, …, ITA_n) operators, of replacing the concept of intervals with 2-component possreps (implicitly assumed to signify closed-open) with that of intervals with 4-component possreps.

PACK

The earlier definition of PACK was defined in terms of PNF. This definition has already been revised such as to cater for our 4-component possrep, no revision is needed to the existing definition of USING (ITA$_1$, ITA$_2$, …, ITA$_n$) PACK().

EQUALS

The earlier definition of EQUALS was defined in terms of PACK. This definition has already been revised such as to cater for our 4-component possrep, no revision is needed to the existing definition of USING (ITA$_1$, ITA$_2$, …, ITA$_n$) EQUALS().

UNION

The earlier definition of UNION was defined in terms of PACK. This definition has already been revised such as to cater for our 4-component possrep, no revision is needed to the existing definition of USING (ITA$_1$, ITA$_2$, …, ITA$_n$) UNION(), both for the "relation version" as well as for the "tuple version".

MINUS

The earlier definition of MINUS was defined in terms of COVBY and PACK. These definitions have both been revised such as to cater for our 4-component possrep, no revision is needed to the existing definition of USING (ITA$_1$, ITA$_2$, …, ITA$_n$) MINUS(), both for the "relation version" as well as for the "tuple version".

INTERSECT

The "USING (ITA$_1$, ITA$_2$, …, ITA$_n$) INTERSECT" operator that operates on two tuples, must be refined (refinements are shown in blue) as follows, in order to cater for the extra 2 components in the 4-component approach for interval values :

For two tuples t_1 and t_2 that are not in the same {ITA$_1$, ITA$_2$, …, ITA$_n$} equivalence class, or are not {ITA$_1$, ITA$_2$, …, ITA$_n$} intersectable, the operator returns an empty relation.

For two tuples that are in the same {ITA$_1$, ITA$_2$, …, ITA$_n$} equivalence class, and are {ITA$_1$, ITA$_2$, …, ITA$_n$} intersectable, the operator returns a singleton relation in which the only tuple has attribute values as follows :

- All attribute values other than (ITA$_1$, ITA$_2$, …, ITA$_n$) are as in t_1 and t_2.
- Attribute values for (ITA$_1$, ITA$_2$, …, ITA$_n$) are interval values with its ITA$_x$ (x=1..n) components as follows :
  FROM MAX(t_1.ITA$_x$.FROM , t_2.ITA$_x$.FROM)
  TO MIN(t_1.ITA$_x$.TO , t_2.ITA$_x$.TO)
  FINCL CASE t_1.ITA$_x$.FROM $<$ t_2.ITA$_x$.FROM : t_2.ITA$_x$.FINCL
  CASE t_1.ITA$_x$.FROM $>$ t_2.ITA$_x$.FROM : t_1.ITA$_x$.FINCL
  CASE t_1.ITA$_x$.FROM = t_2.ITA$_x$.FROM : AND(t_1.ITA$_x$.FINCL,t_2.ITA$_x$.FINCL)
  TINCL CASE t_1.ITA$_x$.TO $<$ t_2.ITA$_x$.TO : t_1.ITA$_x$.TINCL
  CASE t_1.ITA$_x$.TO $>$ t_2.ITA$_x$.TO : t_2.ITA$_x$.TINCL
  CASE t_1.ITA$_x$.TO = t_2.ITA$_x$.TO : AND(t_1.ITA$_x$.TINCL,t_2.ITA$_x$.TINCL)

Note that these refinements amount to nothing more than a refinement to the definition of the "interval intersection operator" for the interval type involved.

JOIN

The earlier definition given for JOIN relies, just like INTERSECT, on a definition for the "interval intersect operator" for some given interval type. Refining the definition of this operator in the way already specified in this section for INTERSECT, will do the job for JOIN.

GROUP/SUMMARIZE

The definition of a "non-granular" USING version of GROUP (and consequently also of SUMMARIZE) that produces results that are compatible with the corresponding definition from TDATRM, was still left a somewhat open question. Repercussions on such a definition of introducing our 4-component possrep, can only be examined if such a definition is first formulated.

All the remaining operators

All the remaining relational operators dealt with earlier, had their USING (ITA_1, ITA_2, …, ITA_n) versions defined as simply being a nested invocation of PACK on an invocation of the relational operator at hand – modulo some specific remarks here and there which we are not reconsidering here. The revision of PACK has already been provided, and the existing definitions for "all the remaining relational operators" can thus be retained.

The question of replacement or co-existence

At the outset, the defined goal was to find an approach to support for temporal data, that could, in some sense, be considered a "proper superset" of the approach outlined in TDATRM. In other words, to find an approach that supports all the use cases supported by the UNPACK-based approach, that supports those in a 100% compatible manner, and that additionally supports a number of use cases that the UNPACK-based approach cannot. The foregoing sections have outlined such an approach, albeit without going the full distance when it comes to formal proofs.

At first sight, this might seem to suggest that the approach described here can replace/supersede the one outlined in TDATRM. But is there perhaps more than meets the eye at first sight ? A slightly more detailed investigation into the matter is warranted.

First of all, there is of course the issue of usability/user-friendliness of the syntax of the language used by the programmer. And then there is the issue of sheer conformance of the language with all of the principles laid out in the "Third Manifesto"[21], the authors of which are also co-authors of the TDATRM book. A survey of thoughts & objections thereto, without much of a firm statement of opinion about any of them.

Programmer-friendliness of the language

In programming languages, programmers are often faced with the need to write literals denoting constant values of the data types they are dealing with. Hence in a language supporting interval types, the need will arise to write interval-value literals, preferably as concisely as possible. An approach that would require programmers to spell out each interval literal using a 4-component value selector such as

```
INTINTERVAL ( FROM (1) TO (5) FINCL (true) TINCL (false) )
```

21 Abbreviated "TTM" for the remainder of the document

might therefore not stand much of a chance of being welcomed with great enthusiasm. Various options at the level of "syntactic sugar" exist, however, to address this. For example, some of the possrep components in the value selector could be made optional, assuming default values for them when indeed they are omitted from a given value selector expression. If the default for FINCL is defined as 'true' and the default for TINCL as 'false' (not coincidentally promoting closed-open usage), then

```
INTINTERVAL ( FROM (1) TO (5) )
```

would denote exactly the same value as the previous expression.

Or, for example, the syntax of the language might simply support notations such as '`[03-05)`' and perhaps even '`( CARTESIAN(2.0 , 3.0) - POLAR(45° , -1.414) )`' and '`["("-"]"]`', though both of these perhaps illustrate some intricate grammar & parsing problems being introduced.

Aspects of "look & feel"

More important seems to be to look at certain aspects of "look & feel" surrounding the manipulation of interval values. TDATRM hasn't really gone into depth on this, so briefly revisiting TDATRM here is warranted.

Intuitive equality

TDATRM exploits the property of ordinality of the point types very neatly to :

- establish the equality between (1-5) , [2-5) , (1-4] and [2-4] (let's call this the principle of Notation-Neutral Equality, NNE)
- sidestep the issue of which notation to use, because, given that it's all the same value anyway, it doesn't really matter which is used.

In a manner of speaking, TDATRM achieves this by exploiting the ordinality of the point types to "associate" 4 different point values with each interval value (well, almost each) :

- ANTE The highest non-included value at the low end
- FROM The lowest included value
- TO The highest included value
- POST The lowest non-included value at the high end

In TTM terms of types and their possreps, and loosely speaking, these are used to make possible 4 distinct possreps for the same interval type :

- POSSREP CLOSEDCLOSED (FROM <pointtype>, TO <pointtype>)
- POSSREP CLOSEDOPEN (FROM <pointtype>, POST <pointtype>)
- POSSREP OPENCLOSED (ANTE <pointtype>, TO <pointtype>)
- POSSREP OPENOPEN (ANTE <pointtype>, POST <pointtype>)

This way of presenting things is not entirely kosjer, however, because :

- TTM as it currently stands requires names of possrep components within a type to be unique for the type across all possreps of the type.
- TTM requires each possrep to be able to denote each value of the type. This is always problematic for at least two of the three "OPEN" possreps whenever a FROM/TO value is the lowest/highest possible value of the point type.

But let's assume for the remainder of the discussion that these are not real issues and "granular" interval types as per TDATRM indeed have these four possreps. Under this assumption, the TDATRM appraoch very neatly conforms with the principle of NNE :

```
OPENOPEN ( ANTE (1) POST (5) ) = CLOSEDCLOSED ( FROM (2) TO (4) )   /* true */
```

THE_ readonly operators as deterministic functions

As a consequence of what it means to be a TTM possrep, the existence of the four possreps automagically leads to the existence of four THE_ read-only operators, named THE_ANTE, THE_FROM, … These read-only operators take an interval value as an argument, and return the "associated" point value that is appropriate for the particular THE_ operator at hand. So the following expressions are possible :

- `THE_ANTE ( CLOSEDCLOSED ( FROM (2) TO (4) ) )     /* 1 */`
- `THE_ANTE ( OPENOPEN ( ANTE (1) POST (5) ) )       /* 1 */`
- `THE_TO ( CLOSEDCLOSED ( FROM (2) TO (4) ) )       /* 4 */`
- `THE_TO ( OPENOPEN ( ANTE (1) POST (5) ) )         /* 4 */`

This illustrates another important and indeed desirable property (one that applies to read-only operators) namely the principle of Read Only Operators Determinacy (ROOD) :

```
FOR ALL v1,v2, FOR ALL f : v1=v2 IMPLIES f(v1) = f(v2)
```

The THE_ readonly operators introduced by the TDATRM interval possreps satisfy this property perfectly and completely.

THE_pseudovariables & the Assignment principle

Another consequence of the existence of a possrep, is the automatic creation of these things called "THE_ pseudovariables". Brief, they allow the syntactic THE_() construct to be used at the LHS of an assignment command :

```
VAR INTINTERVAL II1 := CLOSEDCLOSED ( FROM (2) TO (4) );
THE_ANTE ( II1 ) := 7;
? II1;           /* prints e.g. CLOSEDCLOSED ( FROM (2) TO (6) ) */
? THE_ANTE(II1); /* prints 7 */
```

This illustrates how THE_pseudovariables created by TTM possreps for TDATRM-style intervals satisfy yet another important principle, known as the Assignment Principle (AP) : after assigning some value to some assignment target, evaluating that assignment target yields the value that was assigned.

Now we examine whether & how these properties carry over to a scenario where the 4-component possrep described earlier on is applied to interval types whose point types are indeed ordinal[22].

Intuitive equality with 4-component possrep

Knowing that the concerned point type is indeed ordinal, and on account of "both intervals being concerned with exactly the same set of numbers included", a user might intuitively expect the system to behave as if

```
INTRV( FROM(1) TO(5) FINCL(false) TINCL(false) )
                      =
 INTRV( FROM(2) TO(4) FINCL(true) TINCL(true) )
```

22 A brief analysis can suffice to show that there is no problem with the three stated principles for interval types over a point type that is non-ordinal, but we omit this analysis here.

Some would respond to this with "the equivalence described in there is a useful operator to have available in the system, but it cannot be the interval type's equality operator". Iow, the interval type's equality operator does not expose those semantics, and the operator that does expose those semantics is not the interval type's equality operator. The consequence of this position is that the equivalence semantics are not automatically used in any operations that build internally on the equality operator, such as, e.g. JOIN. This might really not be problematic, since it might be reasonably expected that the vast majority of JOINs over interval-typed attributes are likely to be USING(…) JOINs anyway (which don't suffer from this "semantic difference between equivalence/equality").

But suppose we want the equality operator to expose these desirable equivalence semantics anyway. This may be doable, but we now show why this cannot be done without sacrificing (at least partially) at least one of the other two desirable principles, ROOD and AP.

THE_ readonly operators as deterministic functions

Suppose we wanted to uphold ROOD as well for the THE_ readonly operators, with our 4-component possrep, and still knowing that our point type is ordinal.

Knowing the underlying point type is ordinal, it is no problem to "canonicalize" any given interval value to its "closed-closed" form. And indeed if we want to uphold ROOD, then we're forced into doing that canonicalization, because if we returned 5 for THE_TO() one time and 4 the other time, despite the interval values in both invocations comparing equal, we'd have violated the principle.

Hence by making `THE_TO(INTRV(FROM(1)TO(5)FINCL(false)TINCL(false)))` return the value 4, we've managed to sort of hack our way out of the problem.

A similar situation occurs for the read-only operator THE_TINCL, however … If we want to uphold ROOD for this operator, then we can't afford to return false one time and true another time, depending on which value combinations were used in a value selector (note that that value selector will most of the time not even be visible in the source text !). So presumably we'd want to canonicalize to closed-closed, meaning that this particular operator will always return true !

Hence, things start getting a bit questionable here (why have an operator if it always returns the same value regardless of its operand value ?), but formally speaking, the behaviour is still in line with the ROOD principle, so no problem there.

THE_pseudovariables & the Assignment principle

And then now we turn to THE_TO() and THE_TINCL() as a pseudovariable (i.e. an assignment target), and evaluate the look & feel of this scenario in the light of the assignment principle.

Recall from TTM that assignment to a THE_pseudovariable is shorthand for an assignment to the "argument" of the THE_ pseudovariable, with additional arguments being invocations of THE_ readonly operators on the same variable. In the case of, say

```
VAR INTINTERVAL II1 := INTRV(FROM(1)TO(5)FINCL(false)TINCL(false)) ;
THE_TO(II1) := 9;
```

the pseudovariable assignment gets replaced with

```
II1 :=
INTRV(FROM(THE_FROM(II1))TO(9)FINCL(THE_FINCL(II1))TINCL(THE_TINCL(II1)))
```

Observe that because of the canonicalization in the invocations of the THE_ readonly operators in there, this will end up like

```
II1 := INTRV( FROM(2) TO(9) FINCL(true) TINCL(true) );
```

If after this, we inspect the value of `THE_TO(II1)`, we will indeed obtain the value 9, which is the value used in the assignment.

It will, however, be clear by now that this cannot possibly be maintained as well for assignment to THE_TINCL … If we happen to assign the value true to that component, we will perceive no violation, but if we try to assign false, then we will perceive a problem, by definition of this THE_ readonly operator always returning true !

Also observe that the following case -admittedly silly, but it's only for purposes of illustration- :

```
VAR INTINTERVAL II1 := INTRV(FROM(2)TO(4)FINCL(true)TINCL(true)) ;
THE_TINCL(II1) := false;
```

will _not_ have the effect of making `THE_TINCL(II1)` evaluate to 'false', but instead it will have changed the value of `THE_TO(II1)` (now 3) !!!

This sort of stuff is very clearly crossing the line of what is reasonable in programming language design and behaviour, even if the cases where the damaging effects come to surface could be labeled as "somewhat pathological" …

Also note that all of these considerations apply only to "nongranular" interval types defined over ordinal point types. The "behavioural changes" described here cannot and do not apply to interval types over point types that are not ordinal, but such interval types do still have possreps, associated THE_ readonly operators and associated THE_ pseudovariables. If we were to introduce the "behavioural changes" for our non-granular interval types, in the case when the underlying point type is ordinal, then we'd be creating a setup in which :

- The non-granular interval types over ordinal point types, and the non-granular interval types over non-ordinal point types, both use a syntax (for value selectors, for their possreps, …) that is completely identical (= identical "look").
- But the behaviour of the associated language constructs could be different between the two, (= different "feel").

This in itself is likely to be regarded as poor language design.

So, replacement or co-existence ?

The foregoing considerations seem to suggest that it might be preferable for systems implementing support for interval types, to let the two approaches co-exist, allowing the user to use the TDATRM approach, which is perhaps a better match to his intuition and a bit less quirky in the language department, whenever the point types involved are ordinal, and allowing him to use our "nongranular" approach whenever the circumstances force him to.

Conclusion

This paper has presented a formalism that can underpin systems for manipulation of range data, where support for these range values takes the form of interval types. We regard this formalism as a valid alternative to the one presented in “Temporal Data & The Relational Model”, in that both formalisms give identical results in the use cases supported by both, but the formalism presented here is additionally also able to support intervals over non-ordinal point types.

Proposals have been offered about how the formal notions can be incorporated into the operators of the Relational Algebra. Discussion has also been provided about how the “revised” operators of the Relational Algebra can be put to good use to enforce “temporal keys” and “temporal foreign keys” in relational databases, two important special cases of integrity enforcement.

Two algorithms relating to the core concept of “packed normal form” were presented.

Certain possible problems relating to the closed-open notation that is typically most prevalent in “non-granulated environments” were identified and suggestions to overcome those problems were briefly explored.

All in all, the conclusion seems warranted that the particular criticism of the “granular” approach not supporting the non-granular use cases, can feasibly be addressed, but complete replacement of the “granular” approach may not be entirely desirable.

References

“Temporal Data and the Relational Model”, C.J. Date, Hugh Darwen and Nikos A. Lorentzos, 2003, Morgan Kaufman Publishers, ISBN 1-55860-855-9.

“The Third Manifesto”, C.J. Date and Hugh Darwen,
http://www.dcs.warwick.ac.uk/~hugh/TTM/TTM-2013-02-07.pdf