

# Freeing Data From the Silos

## A Relationistic Approach to Information Processing

Ralf Westphal, [www.ralfw.de](http://www.ralfw.de)

### **Abstract**

Current data processing is limited by a container-reference dichotomy. Data once stored and connected is hard to rearrange and connect in new ways required by needs that have changed over time.

This paper explains an approach to remove this fundamental limitation. It argues data should no longer be recorded and stored, but assimilated and represented/described. Instead of copying data into data structure for further processing, data should be described by a “system of pure relations” in which the data itself is nowhere to be found anymore, but can be re-generated as needed. The benefits of such a “system of pure relations” are infinite connectability of data at any level of abstraction.

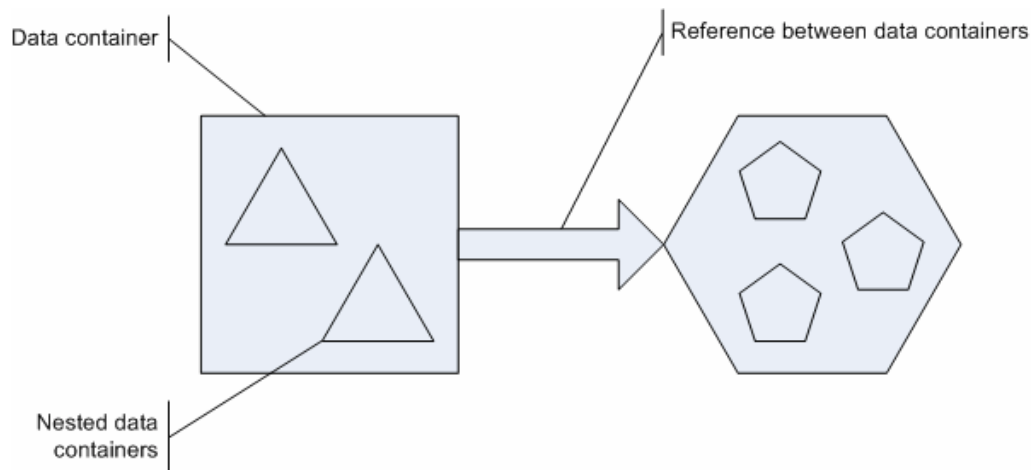
### **Motivation**

Current data processing is suffering from the bane of the many data silos. Data is locked up in a hierarchy of containers and can hardly be connected in new ways. Once data has been grouped into relational database tables or object oriented classes, it's difficult to regroup it or set up new relations. Regrouping would either break existing code or mean duplication of data. New relations would entail schema changes and be limited to connections between containers on only a few levels of abstraction. Products like Microsoft's WinFS, dynamic languages and database refactoring tools are trying to overcome this lamentable state of data processing while having their own perspective on it.

This, of course, does not mean there is no value in current concepts and technologies any more. Relational databases and object oriented languages are very useful and will continue to be so for a long time. However, the general need to go beyond them to solve the data silo problem should be obvious.

### **The Container-Reference Dichotomy**

At the heart of the data silo problem are the basic and mostly unquestioned concepts of *data container* and *container reference* (Fig 1).



*Fig 1: Data containers and references between containers are the current means to relate data*

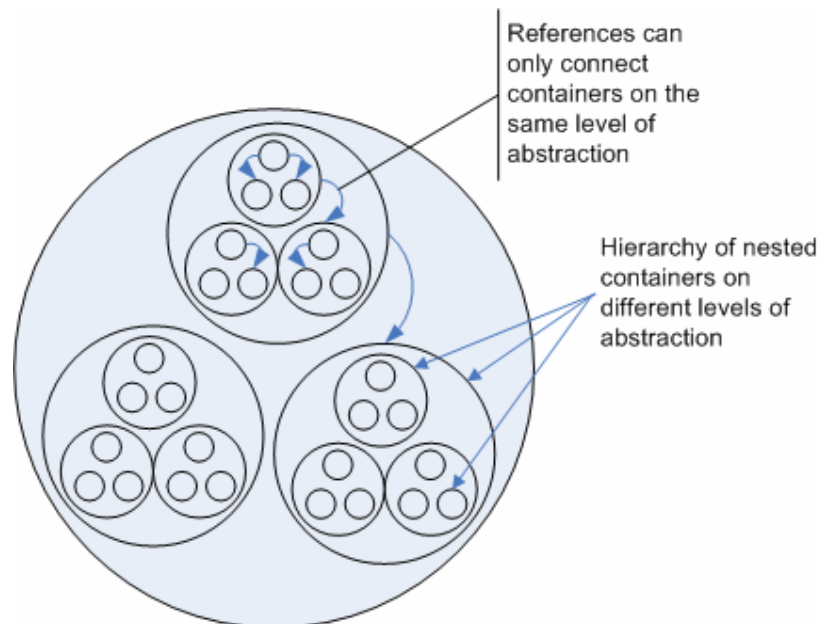
Data is stored in containers of various sizes and on various levels of abstraction. Bits are the smallest and thus atomic data containers. They are aggregated into bytes. Usually, though, the relevant units of data are aggregates of bytes called columns or fields. A column in a relational database table or a field in an object oriented class thus can be viewed as the smallest relevant physical data containers.

Several columns then are combined into rows, rows into tables, tables into databases. Fields are combined into structures and classes.

Apart from separating one data item from another these containers more than anything else serve the purpose to make data access and modification efficient. That also is the reason why it is comparatively difficult to change a data schema. Flexibility in changing a schema would oppose access efficiency.

A data container relates data only in one way by putting data items close to each other. But of course, data items should be relatable in more than just one way. So there needs to be another, more flexible means to connect data. Enter container references: Containers of a certain kind can reference each other, e.g. one row in a relational database table can reference another using a foreign key, or one object can hold an object pointer to another.

Such references have been proven highly successful and efficient – but they are suffering from a drawback: they can only connect containers on the same level of abstraction and of the same basic kind (e.g. rows, objects). References cannot reach into containers (Fig 2). Hence they can only connect what has been foreseen to be connected during schema design.



*Fig 2: References can relate data only on the same level of abstraction*

This is progressively limiting, though, because what once seemed a reasonable schema design might not be reasonable anymore sometime in the future. And what once seemed to be unrelated islands of data now needs to be merged into a unified repository.

In addition, the two kinds of relations between data – vicinity and reference – are completely different concepts which need two different programming models. References could even be viewed as the second class citizens of data processing, since they are purely logical where containers have a physical reality.

The bane of the many data silos thus has its origin in the fundamental dichotomy between data containers and container references. As long as there are fixed physical data containers honed for efficiency with limited ways to connect them across the different levels of abstraction there will be data silos.

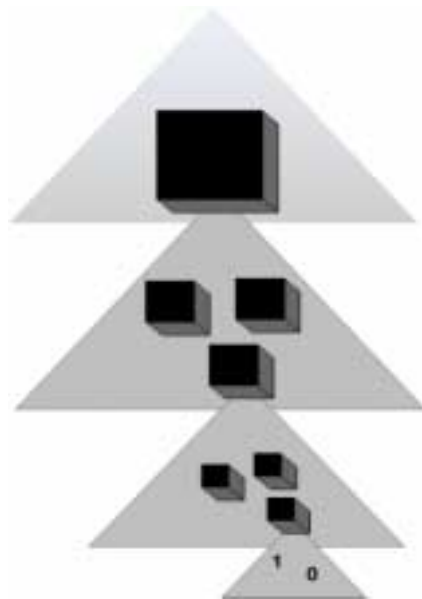
### ***Overcoming the Container-Reference Dichotomy***

The container-reference dichotomy makes it hard to build new connections between data. As long as physical data containers exist, crossing container boundaries to connect data formerly not connected requires explicit effort – which often cannot be afforded although it would be beneficial.

So the question is: How can it be made easier to flexibly relate and connect data? The answer lies in getting rid of data altogether.

### **Step 1: Getting Rid of Data**

As Fig 3 shows, what data is is not really clear. Data on one level of data containers which could be viewed as a black box, is just a set of constituent smaller black boxes when zooming in on it and opening the black box. Then, when in turn zooming in on them, they again fall apart into even smaller black box data items and so on until the bit level is reached (Fig 3).

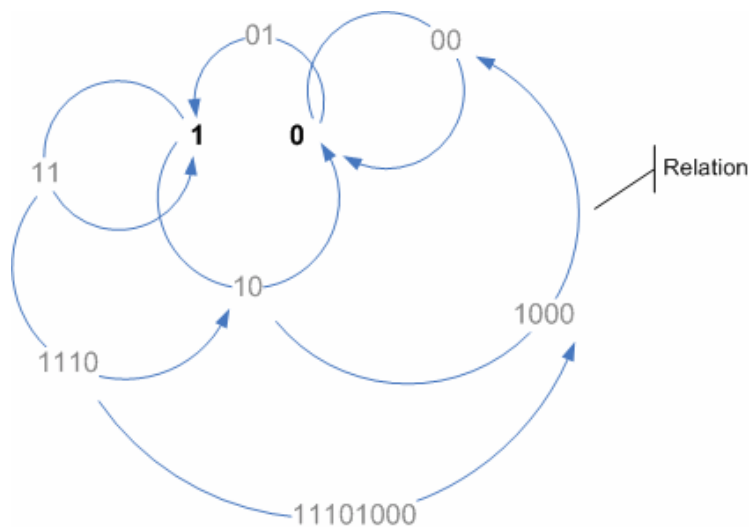


*Fig 3: Decomposing data containers into ever more fine grained data items until reaching the bit level*

This means: In the end there are only two universal and absolute “real” data items: 1 and 0.

All other data items on higher levels can be expressed as just connections between 1 and 0 and connections between such connections (Fig 4): 1 and 0 can be related in 4 ways: 1 connected to 1 (or: 1 combined with 1), 1 connected to 0 etc. (11, 10, 01, 00); the resulting relations can again be related: (11 connected to 10) connected to (10 connected to 01) etc. (e.g. 1110, 1001, 0100). The resulting relations can again be related reaching the byte level, e.g. 11101001 etc. And so on ad infinitum.

Thus any bit pattern can be interpreted as a relation between relations between relations ... between relations between 1 and 0. And since data always is only bit patterns it follows that any data can be expressed by a hierarchy of relations which ultimately is based on relations between 1 and 0.

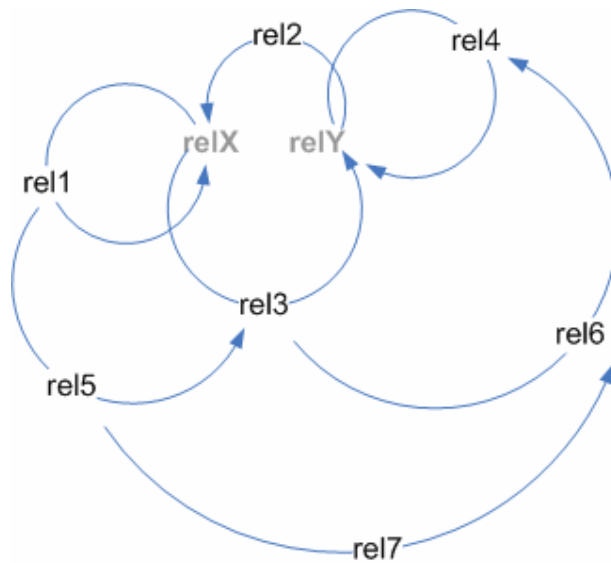


*Fig 4: Any bit pattern can be described by relations of relations of relations ... of relations of 1 and 0*

How such relations look like or are “named” is irrelevant. What is important is the pure ability to express any data through them. Data thus is reduced to the only “real” data items 1 and 0 plus a tree of relations, where the boundary between different data items is vanishing, since there is no data anymore on higher levels but just relations.

So, a *system of pure relations* (SOPR, Fig 5) is sufficient to express any bit sequence or data. Relations are enough to build a tree of relations whose root represents a particular sequence of bits and therefore can represent any data.

However, since the only “real” data items are 1 and 0, which means they are well known and fixed, they can be neglected. As long as it is clear what’s related by a relation like *rel2* or what *relX* stands for, no data at all is necessary anymore in a “description of data”.



*Fig 5: A system of pure relations*

To fully accept this, the only somewhat new notion is directed “relateable relations”, i.e. relations as “something tangible” which can in turn be related. Relations or connections are no longer just an invisible, intangible tie between containers like in Fig 1. Rather they themselves are “objects” of composition. In a SOPR without data, relations are no longer second class citizens. Instead they have become first class citizens, because they are the only citizens of such a system (Fig 6).

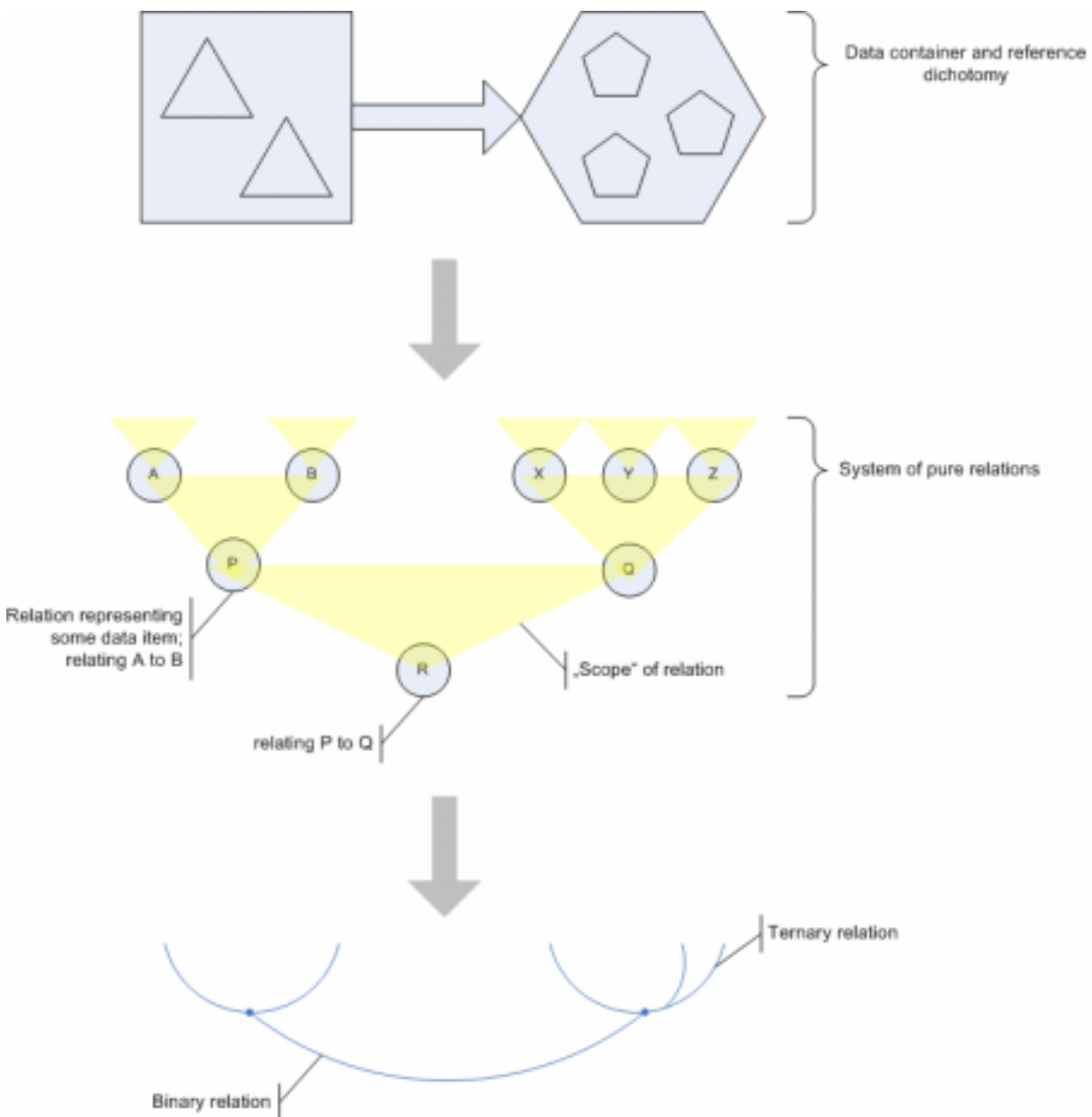


Fig 6: In a system of pure relations there exists only one kind of citizen: relations

## Step 2: Re-Generating Data

A system of pure relations gets rid of data, because it no longer contains copies of data, but instead describes data. Classical data processing stores data by copying it into some kind of data structure (data container). A SOPR on the other hand builds a mesh of relations just losslessly representing the data. That means: The data is nowhere within this mesh, but can be re-generated from it at any time.

Usually, to store the letter “A” its ASCII bit sequence 01000001 is recorded at a certain place. But a SOPR does not record data, but *registers* it by describing it with a tree of, for example, binary relations (Fig 7).

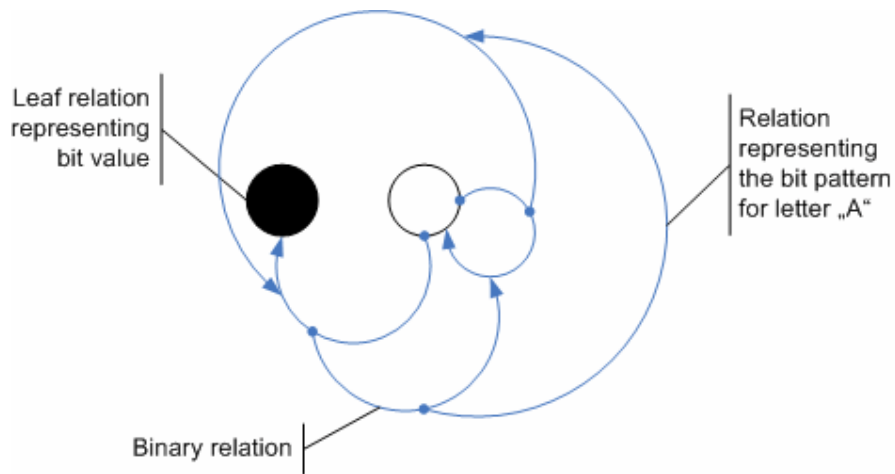


Fig 7: System of pure relations describing the bit sequence for the ASCII letter “A”

So, the data is effectively gone. But the process of registering it as a mesh of relations can be reversed to re-generate it:

1. Start from the “placeholder relation” (at the right of Fig 8) for the data (step 0).
2. Recursively walk up the binary tree representing the bit sequence until reaching leaf relations (black and white circles in Fig 8, steps 1 to 14).

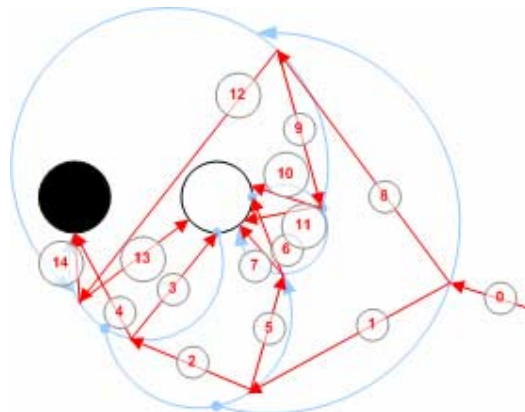


Fig 8: Recursive traversal of binary tree representing the bit sequence 01000001

The result of such a recursive tree traversal is a sequence of leaf nodes visited, e.g. white, black, white, white, etc.

Then the final step to re-generate the data is to map the leaf nodes encountered to 1 and 0. If the black relation is translated to 1 and the white one to 0, then 01000001 is generated. The original data has been reconstituted.

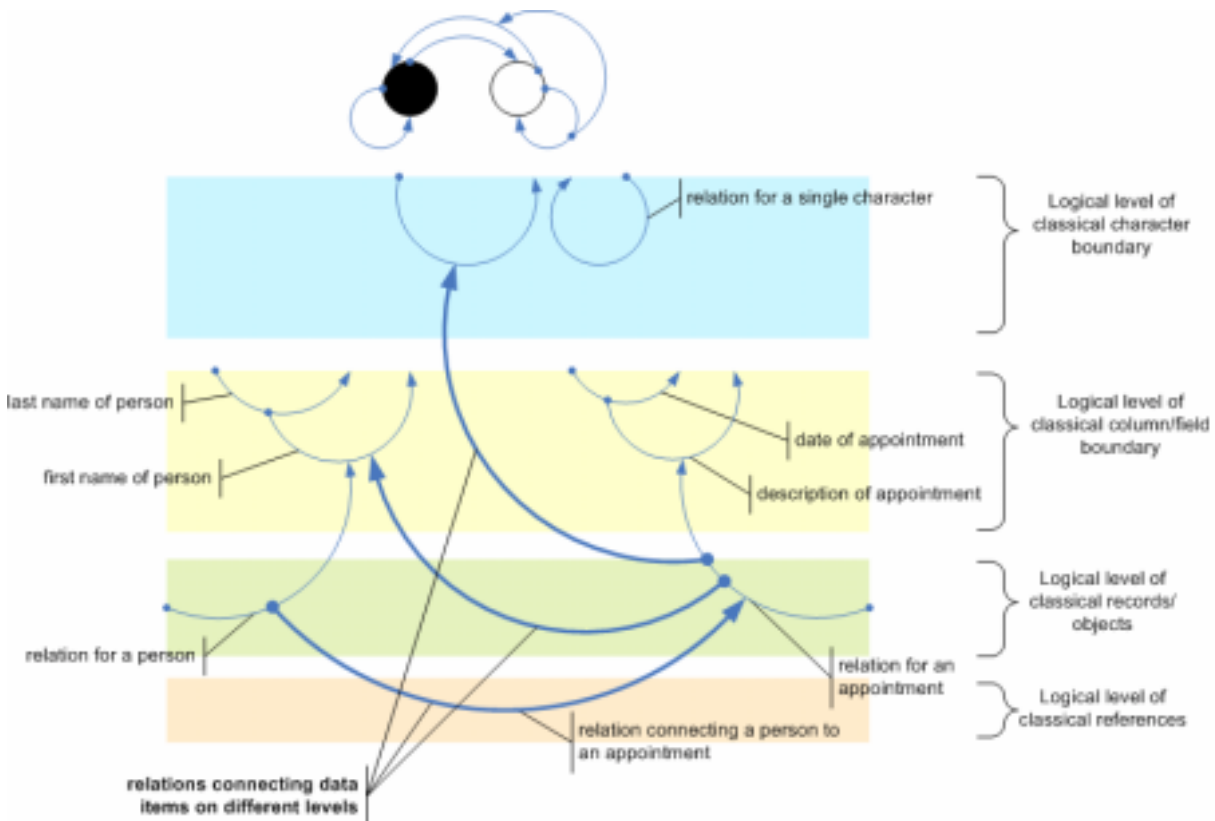
Mapping between bit data and relations is done by an *observer*, which also determines how to describe bit sequences using relations. The relations themselves, though, don’t know anything of bit values or which data they describe. This lies completely “in the eye of the beholder”, i.e. it is the responsibility and knowledge of the observer, who registers and re-generates data.

### Step 3: Unlimited Relations

How does a SOPR break up boundaries and allow for more flexible connections between data?

- Firstly, in a SOPR, the dichotomy of data container and container references is gone. That means there are no hard, physical boundaries around data items anymore. And there are no explicit references anymore. The smallest “data item” and the only one is a relation.
- Secondly, relations are not related physically, but only logically. Bytes, when recording data, are related within a container by putting them one after another. But since relations are free of such limitation, they can be related in arbitrary ways.

Both properties taken together allow the connection of data at any level across any logical boundaries: a person can be connected to an appointment, a name within a person can be connected to an appointment, and even a single letter within a person’s name can be connected to an appointment (Fig 9).



*Fig 9: In a system of pure relations any relation, i.e. data item representation, can be connected to any other on any logical level*

In a system of pure relation there is no physical difference between a single character, a name, a person, or even all persons registered. Data on all levels is accessible by just a single relation.

All data registered in a SOPR can be connected at any time in arbitrary ways by setting up new relations between existing ones. There is no notion of a physical container anymore, which needed to be treated as a black box with regard to connections. All relations are always visible and accessible and connectable at all times. Boundaries exist only as a logical concept of the observer who registers the data.

This not only removes the limitations of current data handling with its container/reference dichotomy and provides ultimate flexibility. It also moves data processing to a new level: Currently data processing generally and naturally and literally means processing the data itself. This can lead

to considerable data to be moved around. With a SOPR, though, data processing can become symbol processing, because the data is gone and has been replaced by symbols in the SOPR.

It could be argued, the step from data to symbol resembles the step from analog to digital and is a logical evolution: As long as data (e.g. pictures, music) was stored on analog media (e.g. tape), it was hard to manipulate. But once analog data is stripped off of its physical body by digitizing it and thus made available to software, it can be modelled with much more flexibility. So, what will become possible, if now even digital data inside a computer is stripped of its “physical body” (i.e. data containers)?

## **Summary**

To really open up the data silos of the world, the fundamental dichotomy of data container and container reference needs to be overcome. Only when there are no fixed boundaries around data anymore, only when data on any level is equally well connectable to any other data, then the flexibility is attained to cope with ever changing data integration and processing requirements.

However, this flexibility comes at a price. It has to be weighed against efficiency. But only with a system of pure relations there exists a second choice at all. Only now a decision becomes necessary, how to store data: in some classical manner, or as a SOPR, or in a hybrid way?

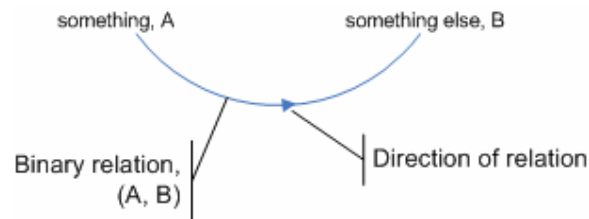
## **Pile: A System of Pure Relations**

To overcome the container-reference dichotomy current data models have to be abandoned, since they are all built on the very notions which are limiting data connectivity. A system of pure relations (SOPR) out of necessity needs to be completely different.

### **Terminology**

*Pile* – invented by Erez Elul – is such a SOPR.<sup>1</sup> It is built solely on the notion of relations.

**Relations are binary and directed:** Pile relations always relate just two “things” (Fig 10).



*Fig 10: A basic Pile Relation*

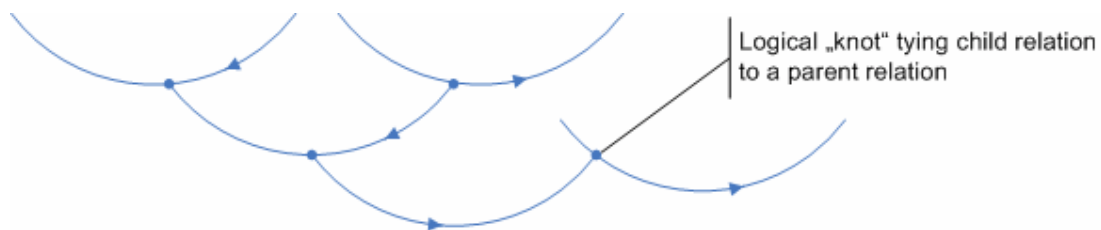
Example:

$(A, B)^2$

relates A to B. However,  $(A, B)$  is different from  $(B, A)$ .

**Relations relate relations:** The only “thing” a Pile relation can relate are relations, since there are just relations in Pile. Pile does not know of any data items and thus relations have no “meaning” or “business value” by themselves. A relation is a relation is a relation.

A relation is not data, and a relation is not a pointer. A relation is more like a string tying two “things” together, and incidentally those “things” tied together are again just strings (Fig 11).



*Fig 11: Pile relationist structure*

**Family relations:** A relation relating two other relations (relatees) is called a child relation. The relatees are its parent relations. Since Pile relations are directed, the relationship between the relatees is not symmetric, they cannot be exchanged. Pile thus gives each parent a specific role

<sup>1</sup> This paper deviates from Erez Elul’s definition of Pile, though. Some concepts have been omitted because they seem implementation dependent, some concepts have been added. However, this explanation of Pile is convinced to be true to the essence of Erez Elul’s intentions behind Pile.

<sup>2</sup> The notation used here is the Pile Definition Language (PDL). See Appendix A for a formal definition.

name: the left parent is called the *Normative parent* (Np), the right parent is called *Associative parent* (Ap).<sup>3</sup> Np is the origin of a relation, Ap the target (Fig 12).

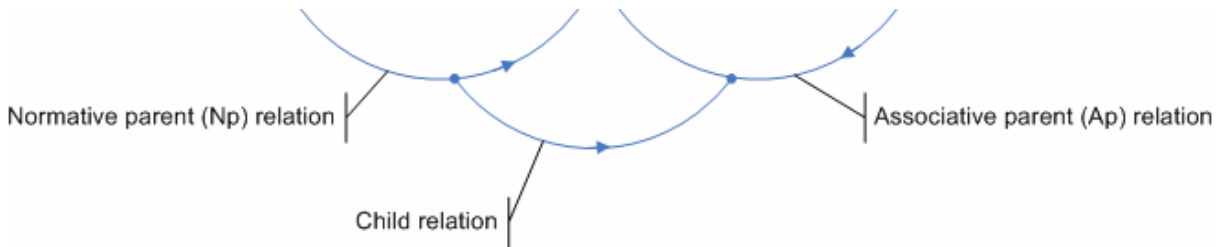


Fig 12: Family ties between Pile relations

Example:

$R=(X,Y)$

R is the child relation relating Np X to Ap Y.

**Relations are unique:** There is only one relation for every sequence of two relations to be related.

Example:

If

$R=(A,B)$

and

$S=(A,B)$

then

$R=S$ .

Pile relations have an identity through their uniqueness.

Also Pile relations are immutable: They cannot be changed after creation, because changing one of the parents would lead to a different child relation altogether. The result would not be the same relation with some property changed, but a different relation. Thus relations either exist with their identity, or they don't. Relations cannot be changed, because they are no containers; there's nothing to change. They are, so to speak, just pure beings.

**Representing data:** Although a SOPR does not contain any data, it would hardly make sense, if it was not somehow related to data. But of course this data cannot be inside the system! Fig 7 shows how relations can be used to represent data based on atomic data values not being part of the SOPR: the atomic data values 1/0 are represented inside the SOPR by "placeholder relations" (black/white).

---

<sup>3</sup> Why Pile has chosen those role names for the parents is not important for the purpose of this document. Take them as idiomatic for now.

That means, when using a SOPR one last physical boundary has to be drawn: the inside of the SOPR needs to be separated from the outside (Fig 13). In the outside world there still are data containers and references, but within the SOPR there are only relations.

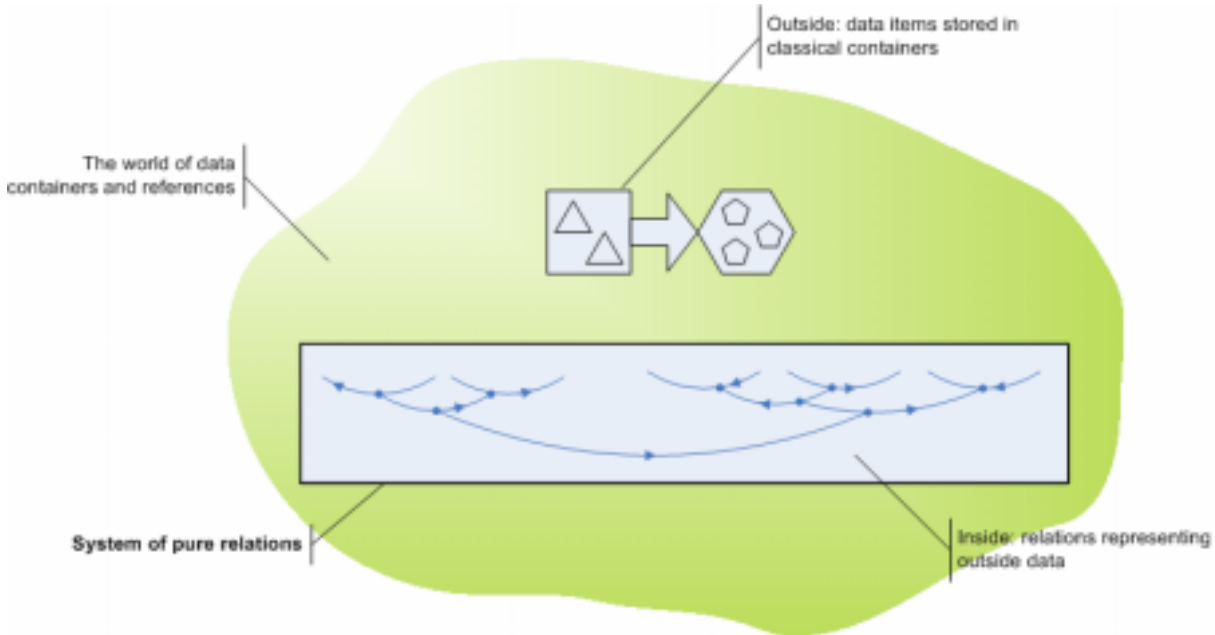


Fig 13: The outside-inside distinction of a SOPR

To connect the inside to the outside, to relate the world of relations to the world of data, outside data items need to be uniquely represented within an SOPR. Such outside data items are called *Terminal Values* (TV) in Pile and their “placeholders” within a Pile are called *Top relations* or *Tops* for short.

Tops are special relations without parent/parent relations. They don't relate other relations but rather relate the outside to the inside. Each Top represents one TV so there is a 1:1 correspondence between TVs and Tops. Example:

$R=()$

R is a Top relation, no parents are given in its definition. All Tops are distinct. Example:

If

$R=()$

and

$S=()$

then  $R!=S$ .

Fig 14 shows how bit patterns like in Fig 4 can be represented using Pile: Bit values 1 and 0 are chosen as TVs and for each a Top relation is created. Then those two Tops are related by child relations using them as parents.

Please note: The labels on the relation lines are just used to denote, what is related. They are not supposed to suggest any values for the relations. Relations don't have any particular values, they are just relations and those pure identities.

Example:

Top1=()  
 Top2=()  
 Top3=()  
 R=(Top1, Top2)  
 S=(R, Top3)

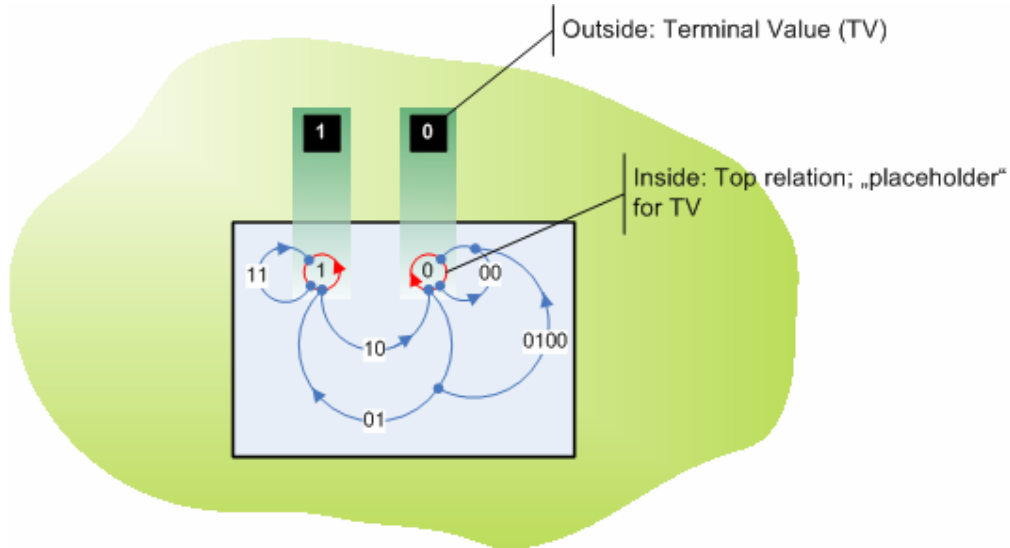


Fig 14: Bit values as Terminal Values

Since Pile does not know anything about data, arbitrary data items can be chosen as TVs. Fig 14 uses bit values, but also single characters can be used as TVs (Fig 15), or strings of characters (Fig 16), or 64-bit integer values, or even bitmaps (Fig 16). Data of any size and form or even concepts can be chosen as TVs and represented inside a Pile by a dedicated Top. From the point of view of a Pile, all TVs are equal in that they are atomic, because Tops don't have parents.

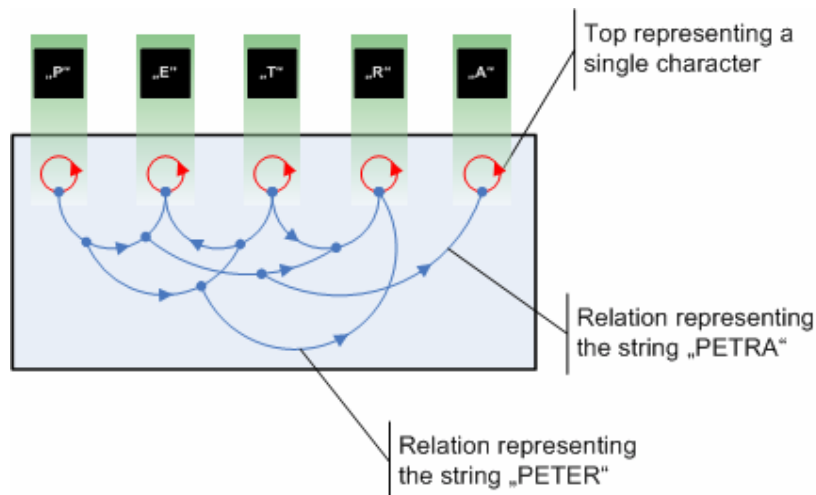


Fig 15: Single Characters as Terminal Values

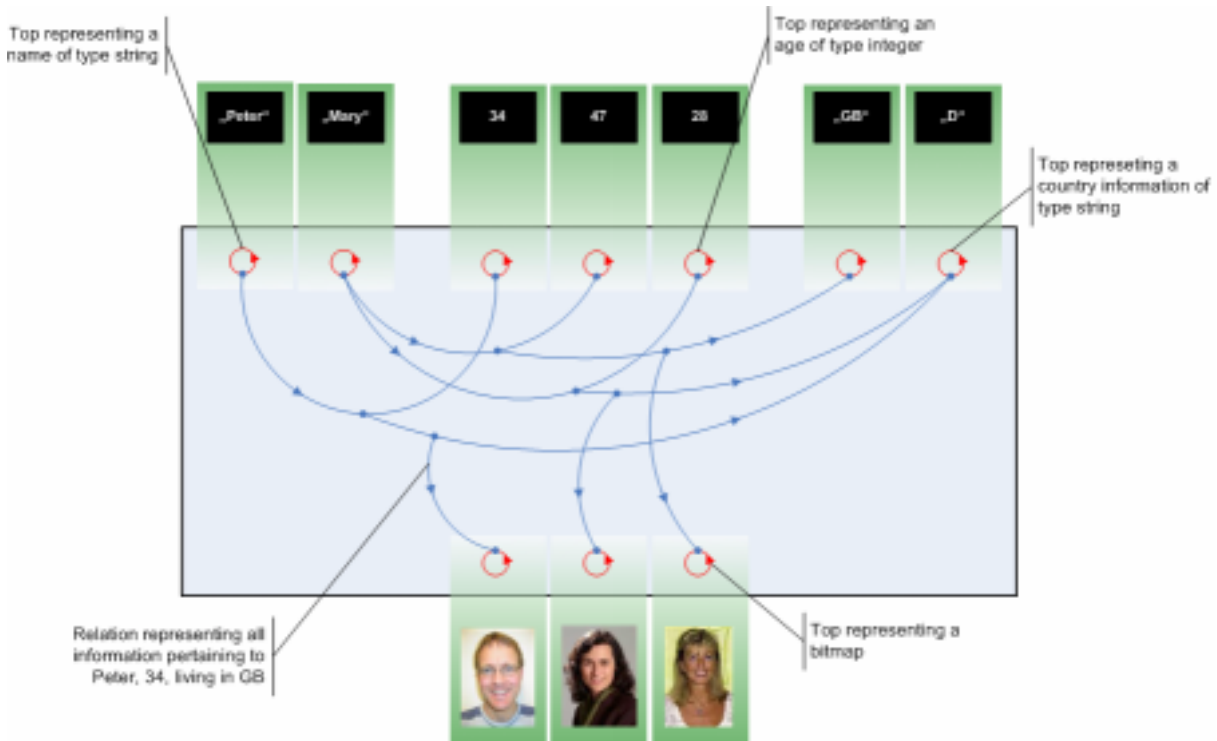


Fig 16: Arbitrary Terminal Values

What follows from the uniqueness of the TV-Top relationship is the “reuse of data”. Since a relation can be used as a parent to any number of child relations, a single Top can represent its TV in any number of contexts. See for example the name “Mary” in Fig 15: Two persons are named Mary, thus its Top is used in two contexts describing those persons, but the TV, i.e. the string “Mary”, actually needs to be present outside the Pile only once.<sup>4</sup>

**The eye of the beholder:** A Pile does not contain any data. But a Pile is related to or associated with data “at its edge” through Tops and thus can represent or describe any data by relating Tops directly or indirectly in arbitrary ways. Tops thus somewhat sit on the rim of a Pile black box (Fig 17) connecting the outside world of data containers and references with the world of pure relations inside the black box.

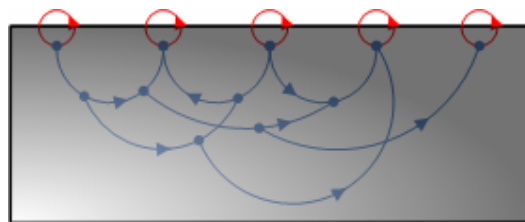


Fig 17: A Pile as a black box

Now, what is chosen as a TV and what not, and how relations are build upon Tops and on each other, is the sole responsibility of an observer. The observer receives data from the outside world, shreds it into TV-pieces, and creates relations for the structure encountered in a manner it sees fit.

<sup>4</sup> The ability to reuse any relation representing any number of TVs through its parent and grandparent relations etc. promises a logarithmic growth of a Pile compared to a database storing the same information.

The observer maps TVs to Tops and the other way around. It thus also is responsible for re-generating any data from the relational mesh it created in a Pile.

Only the observer knows how to interpret a Pile. Only the observer can make sense of the Tops. To help himself with this mapping, an observer can attach a unique URN<sup>5</sup> to any Top upon creation (Fig 18).

Example:

TopP=("urn:A")  
 TopE=("urn:E")  
 PE=(TopP, TopE)

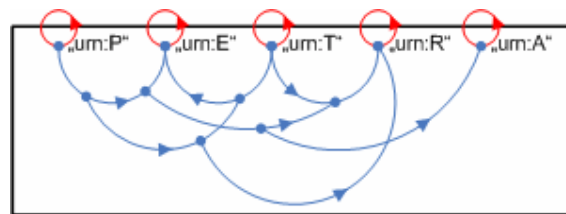


Fig 18: Pile URNs

This URN is of no interest to a Pile itself. Its sole purpose is to include in a Pile minimal identity information about the part (TV) of the outside world a Top is referring to.

**Classifying relations:** All relations are created equal. Except for Tops having no parents there is no difference between the relations in a Pile. Relations are the only concept in a Pile and all of them are unique. This makes for a very homogeneous structure.

However, in the outside world, there is not only different data, but also different kinds of data. Data usually is of some type, e.g. a basic data type like *integer* or *string*, or an aggregate type like a class or table row. Types categorize data items, they provide a logical dimension to physical data.

Although a SOPR does not know of any data, it can benefit from retaining the type concept. If relations can have a type, it is easier to interpret them. Hence Pile allows assignment of a type to every relation upon creation; this type is called a relation's *Qualifier* (or Q for short) and is itself a relation (Fig 19).<sup>6</sup>

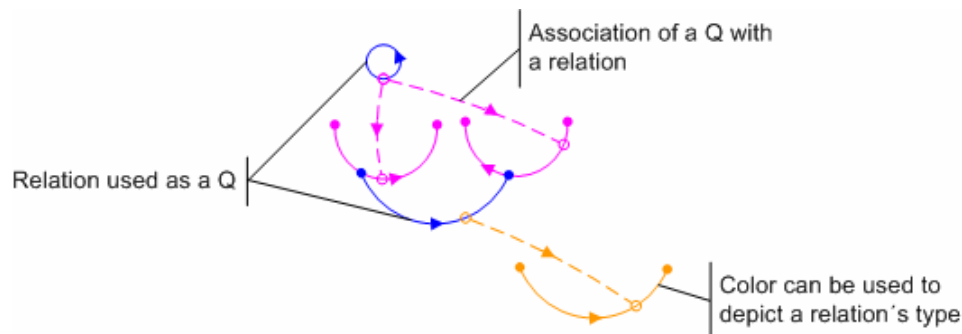


Fig 19: Pile Qualifier

<sup>5</sup> [http://en.wikipedia.org/wiki/Uniform\\_Resource\\_Name](http://en.wikipedia.org/wiki/Uniform_Resource_Name)

<sup>6</sup> The original suggestion for Qualifiers from Erez Elul was to limit them to a certain range of integer values (e.g. 0 to 255) depending on the number of bits (e.g. 8 bits) reserved for them in an integer representation (e.g. 32 bits) of Pile relations. This, though, seems to put an unnecessary constraint on Qualifiers and unnecessarily introduces Qualifiers as a completely distinct concept from relations.

Example:

Q1=()  
Q2=(X, Y)  
...  
R=(A [Q1] B)  
S=([Q2] "urn:myTV")  
T=(Q1 [R] Q2)

However, even though a Q itself is a relation, it is not another parent relation to the relation it is assigned to. Relations are still binary.

By using relations to classify other relations Pile is unique in using the same concept to model information as well as meta-information. Where object orientation distinguishes between class/type and object/instance and relational databases distinguish between table definitions and rows, Pile remains just a system of pure relations.

Fig 20 shows an example of how Qualifiers could put to use to structure a Pile representing a business entity like a flight information record. A flight's properties like number or arrival data have been chosen to be TVs, relations connect their Tops to set up a relation between all flight information, and additional Tops were chosen to represent type information for the relations representing atomic data (e.g. "Arrival date") as well as relations connecting them (e.g. "arriving at").<sup>7</sup>

---

<sup>7</sup> Please note, this is but one way to represent business entities and types. Many other schemes to encode meta-information and data could be devised.

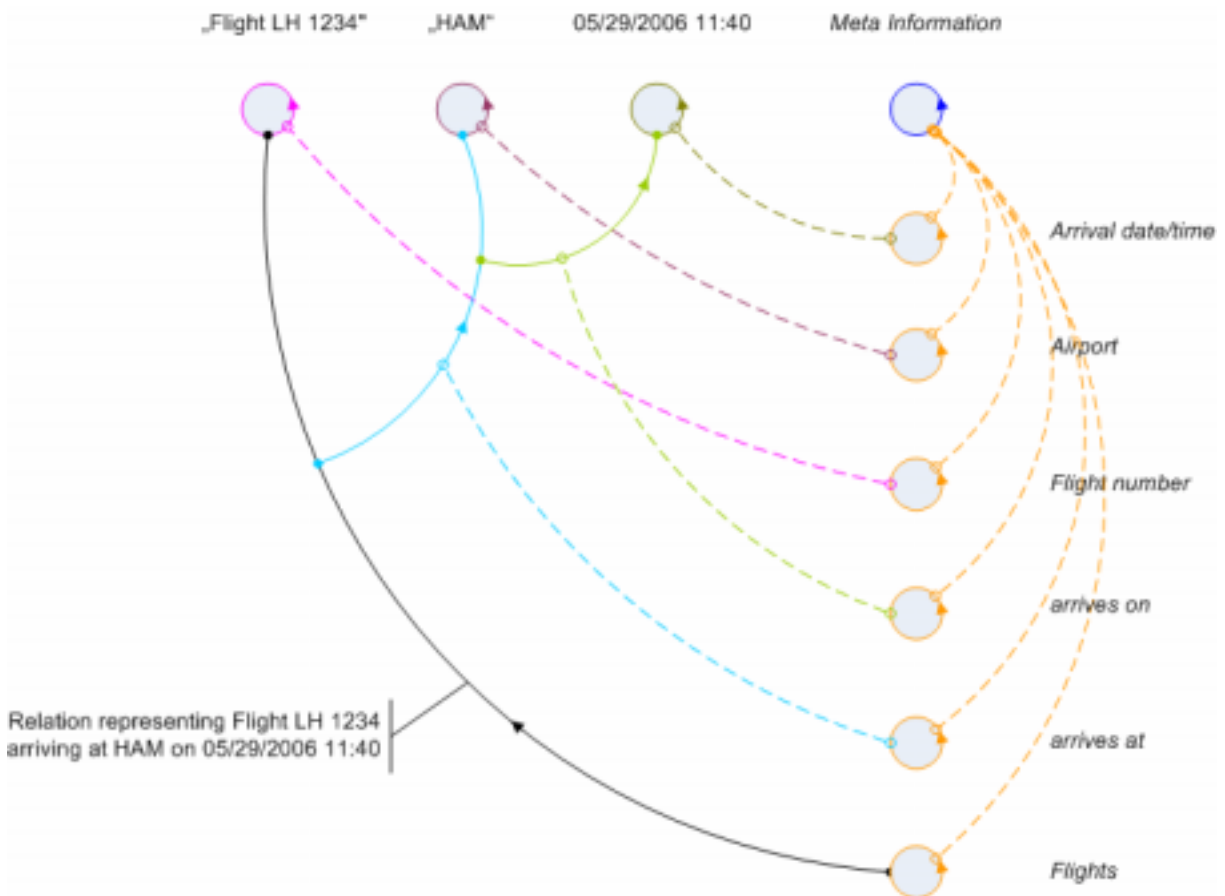


Fig 20: Possible qualifier use in business applications (here: Flight Information System)

## Operations

Relations are unique, binary, directed, and bidirectionally navigable. When drawing them on paper and using a finger to traverse them, these definitions and rules are easy to follow. But before trying to implement a Pile as an Abstract Data Type (ADT) in a programming language, the set of permitted operations on a Pile should be formalized a bit more.

The following operations are a suggestion for a minimal and simple set of functions an ADT should provide to allow building and navigating a Pile according to the above described concepts. However, even though specific suggestions are made for functions (or even object oriented classes with methods), no suggestions are made as to how a Pile's internal data structures should be implemented. What exactly a relation is, whether a simple number, a GUID or an object, purposely is left for an implementer to decide. The only requirement is, they need to exhibit the described conceptual properties and operations.<sup>8</sup>

## Creation and existence

Relations can be created “out of thin air” to represent a TV or by combining two parent relations. If two parents are related and they have been related before, no new relation is created.

<sup>8</sup> This is especially important to note since at other places Pile is described in a way which does not strictly separates theory from implementation. For example it is an implementation details to say, relations are 32-bit integer values. Maybe it's necessary to use integer values for relations in order to attain scalability for a Pile implementation – but maybe not. In any case there are no conceptual properties or claims which involve integer relations that are not satisfied by the concepts presented in this paper.

## Create a Top relation

*PDL*

Top=()

Top=(*[Q]*)

Top=("urn:myTop")

Top=(*[Q]* "urn:myTop")

*Programming language binding*

TopRelation CreateTop()<sup>9</sup>

TopRelation CreateTop(Relation qualifier)

TopRelation CreateTop(string urn)

TopRelation CreateTop(Relation qualifier, string urn)

## Create a child relation

*PDL*

R=(Np, Ap)

R=(Np *[Q]* Ap)

*Programming language binding*

ChildRelation CreateChild(Relation normativeParent, Relation associativeParent)<sup>10</sup>

ChildRelation CreateChild(Relation normativeParent, Relation associativeParent,  
Relation qualifier)

## Check for existence of a Child relation

*Programming language binding*

bool Exists(Relation normativeParent, Relation associativeParent)

## Check, if relation is Top or Child

*Programming language binding*

bool IsTop(Relation relation)

*Object oriented alternative:*

bool Relation.IsTop

## Parent axis

To use of a Pile its relational mesh needs to be traversed. One direction to traverse it is from child relation to parent relations up to any Tops.

*Programming language binding*

Relation GetNormativeParent(Relation relation)<sup>11</sup>

Relation GetAssociativeParent(Relation relation)

*Object oriented alternative:*

Relation ChildRelation.NormativeParent

Relation ChildRelation.AssociativeParent

## Child axis

Where the parent axis points "upwards" within a Pile's relational mesh, the child axis points downwards, and thus makes a Pile bidirectionally traversable.

---

<sup>9</sup> Always returns a new Top relation

<sup>10</sup> Either returns a new Child relation or returns an already existing Child relation for the two parents

<sup>11</sup> Returns *null* if relation is a Top

Since relations can play two different roles when used as parents, children should also be classified according to whether they are children where a relation is the Np or the Ap. Children thus can be divided into Normative children (Nc) and Associative children (Ac) (Fig 21).

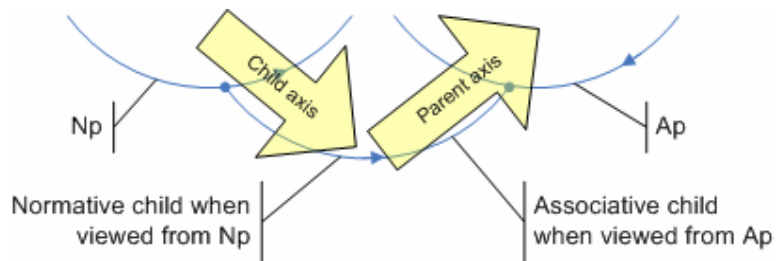


Fig 21: A child relation's role is either normative or associative depending on the parent starting a traversal from

#### Programming language binding

```
Relation[] GetChildren(Relation relation)
Relation[] GetNormativeChildren(Relation relation)
Relation[] GetAssociativeChildren(Relation relation)
```

#### Object oriented alternative:

```
Relation[] Relation.Children
Relation[] Relation.NormativeChildren
Relation[] Relation.AssociativeChildren
```

### Qualifier axis

Although Qualifiers are no parents to relations they connect relations and thus provide a path through a Pile. In order to traverse a Pile on a meta-level, operations are needed to travel along the Qualifier axis.

#### Programming language binding

```
Relation GetQualifier(Relation relation)
Relation[] GetInstances(Relation qualifier)
```

#### Object oriented alternative:

```
Relation Relation.Qualifier
Relation[] Relation.Instances
```

### What does it all mean?

Pile defines a system of pure relations which are unique, binary, directed, and bidirectionally navigable.

A Pile knows no data, no data containers, no container references. A Pile does not impose any physical boundaries on representations of data or concepts.

A Pile is never closed to new relations. At any "point" in a Pile, to any relation new relations can be attached at any time.<sup>12</sup>

Each relation can be the parent on either side of an arbitrary number of relations.

<sup>12</sup> Until now Pile does not even define a deletion operation. Instead of deleting a relation it seems to be "more natural", to "change its meaning" by changing its context by adding new relations. However, at least deleting relations having no child relations yet, should be easy to implement.

Each relation can be the type (Qualifier) of an arbitrary number of relations. It can even represent data and meta-data at the same time.

A Pile does not impose any constraints on what its relations can represent. They might represent single bit values, gigabytes of image data, or physical entities like a car or a person. Pile relations thus can be viewed as symbols, as abstract entities all created equal but representing arbitrary “things” in a uniform manner. Pile relations are never something (in the outside world), but always just represent or describe something qua their identity. Thus the process of observing data items and representing them in a Pile cannot be called “storing” or “recording”, because this would suggest copying data into a Pile. Rather this process should be called “registering” or “assimilating” information, because although the data is not lost during the process, it cannot be found within the system of pure relations.

From a philosophical point of view maybe it can be said, that in Pile data “exists while at the same time does not exist”; it has a kind of invisible presence. This might become a little more understandable when choosing just the bit values 1/0 as the only TVs to represent data and building from them all representations of outside data. Because then truly no data items need to exist outside a Pile<sup>13</sup> and a Pile becomes “self sufficient” and all encompassing. When no data exists but just a Pile and an observer, then a second reality is created by an observer within a Pile, where all of the observed reality, i.e. all data assimilated, is represented/described by an intricate mesh of symbols (relations) – from which, though, all of reality can be re-generated by the observer when needed.

How such a mesh should be woven, an observer would need to decide. It’s solely dependent on the purpose of a Pile: What data and concepts does it represent? What is to be done with the symbolic representation?

Should a Pile help to search patterns in texts? Should a Pile help find patterns in genome nucleotidnucleotide sequences? Should a Pile help find circles in graphs? Should a Pile help managing structured data? Should a Pile help analyse enterprise data?

For each of these and a myriad of other potential fields of application different choices can and should be made as to which TVs to chose and how to associate relations.

Two invariants of the application of Pile seem to exist, though:

- Re-generation of data from Pile representations requires traversal of the relational mesh.
- When using Pile for some kind of pattern finding<sup>14</sup> traversal of the relational mesh should be avoided, because it quickly becomes computationally complex. Instead the uniqueness and identity of Pile relations should be exploited.<sup>15</sup> This means, it’s easier to ask if a relation already exists, than to search for some indirect connection between relations by traversing a Pile along the parent and child axes.

Pile currently is still in its infancy. Although the described system of pure relations seems to be easy while at the same time allowing for very complex applications, it is by no means complete. Its notations<sup>16</sup>, operations, and usage patterns need to be tested in as many scenarios as possible.

---

<sup>13</sup> contrary to for example Fig 20, where whole strings and date were chosen as TVs

<sup>14</sup> e.g. finding a textual pattern in a body of assimilated texts or finding a circle in a graph

<sup>15</sup> Even though this might mean a large number of relations need to be generated from data assimilated. Trading memory usage against access speed seems to be a necessity when working with Pile and might even hint at application scenarios.

<sup>16</sup> of which only two have been used in this paper

Also, it seems to be beneficial to explore what Pile could mean for more theoretical concepts like polycontextural logic or artificial intelligence.

However, that much seems to be sure: Pile in sum is new and unique, although building on established concepts.<sup>17</sup> Pile thus is not another data structure to store data, nor is it another data structure to index data. Also Pile is not a system to describe meta-data, so it's not a data schema.

Rather Pile is a fundamentally different way to represent information of any kind by not keeping copies of it and not separating data from connections between data or meta-data.

Hence, Pile should not be compared to object oriented data structures or relational databases, which are based on the container-reference dichotomy, which in turn is based on bytes being stored one after another, which are a legacy of the basic memory architecture of 'today's computers.

Pile rather is an alternative view on how to represent "the world" (or just information about the world or data). It thus describes an alternative memory architecture, which means it describes an alternative to any low level byte oriented API.<sup>18</sup>

Fig 22 shows the fundamental difference between Pile's relational information representation and representing information as data using bytes.

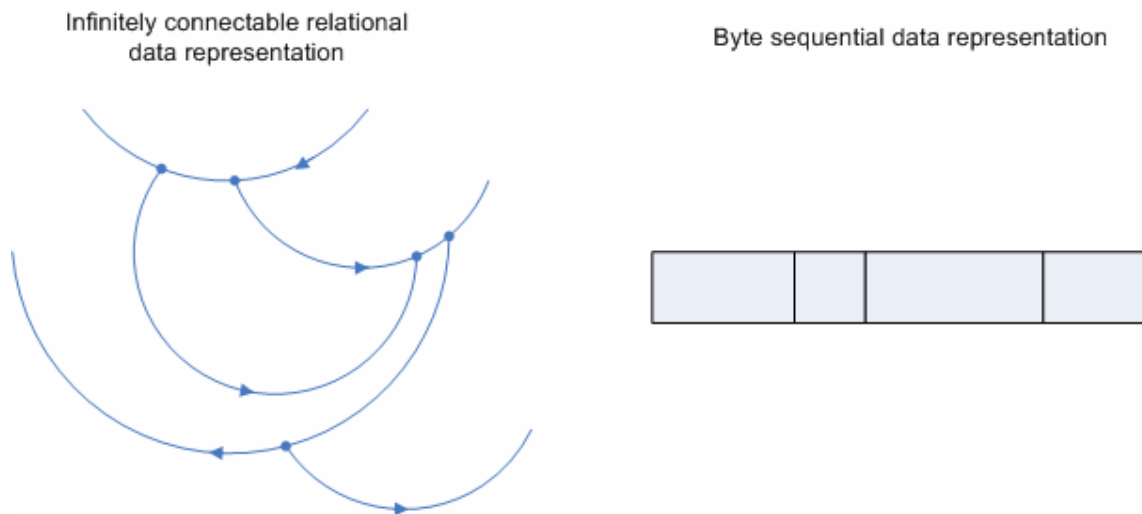


Fig 22: Pile relations vs. data kept in containers

A byte sequential data representation has only one physical dimension to organize information and is closed after choosing a particular layout.

Pile's associative data representation on the other hand is multi-dimensional and always open.

© Ralf Westphal 2006

<sup>17</sup> e.g. the notions of relation, trees, types

<sup>18</sup> One suggestion has been made to implement a Pile on graphic processing hardware whose basic unit are triangles or triples of numeric values, because Pile relations could be viewed as triples consisting of identity values for Np, Ap and Child relation.

## Appendix A – Pile Definition Language (PDL) Grammar

### Syntactical Rules of PDL

Pile ::= RelationAssignment { RelationAssignment } .  
RelationAssignment ::= RelationIdentifier “=” RelationExpression .  
RelationExpression ::= “(“ (TopRelation | MediumRelation) “)” .  
TopRelation ::= [ Qualifier ] [ URNstring ] .  
MediumRelation ::= Relation (“,” | Qualifier) Relation .  
Qualifier ::= “[“ Relation “]” .  
Relation ::= RelationIdentifier | RelationExpression .

### Lexical Rules of PDL

RelationIdentifier ::= (letter | digit | “\_” | “@”) { letter | digit | “\_” | “@” } .  
URNstring ::= *Sequence of printable characters enclosed in “”, but no line break.*

### Examples

#### RelationIdentifier

R, @, \_ABC, 1234, 987\_ABC, @XYZ

#### URNString

“urn:A”, “urn:www.pileworks.org/terminalvalues/9238283”

#### RelationExpression with TopRelation

() , ([Q]) , (“urn:A”) , ([Letter] “urn:A”)

#### RelationExpression with MediumRelation

(X, Y) , (X [Q] Y)

#### RelationDefinition

MyTop=() , A=([Letter] “urn:A”) , R=(X, Y) , Q=((X,Y) , (Z [Q] (A, B)))

#### Pile

Letter=()  
P=([Letter] “urn:P”)  
E=([Letter] “urn:E”)  
Text=()  
PE=(P [Text] E)

If *RelationAssignments* are written on different lines, then no explicit delimiter is needed. Otherwise a “;” or “.” or “;” should be sufficient, e.g.

Text=(); PE=(P [Text] E)

### References

- Homepage of Pile Systems - “We develop, license and distribute the patent pending Pile technology.”: [www.pilesys.com](http://www.pilesys.com)
- Open Source Pile site: <http://sourceforge.net/projects/pileworks/>
- Earlier weblog postings on Pile by Ralf Westphal: <http://weblogs.asp.net/ralfw/archive/tags/Pile/default.aspx>