A Journey Into The Pile Universe

The following chapters are taken as is from Ralf Westphal's WebLog at http://weblogs.asp.net/ralfw/category/10111.aspx. Ralf is a freelance software architect, consultant, and trainer on .NET Framework software development, author of more than 200 technical publications and frequent speaker at developer conferences (for more information see his website www.ralfw.de). His series of blog postings on Pile describes Ralf's continuing journey into the world of Pile which started on November 23, 2005. The blog postings are a unique combination of learning and teaching at the same time and might be helpful to others coming from mainstream computer science.

1. Storing Relations Instead of Data - Just a Cool Idea or a Revolutionary New Data Storage Paradigm?

After my recent talk on the Software Cells model for designing software at the iX developer conference in Cologne, Germany, I was approached by Peter Krieg of Pile Systems who saw some similarity in our thinking. He asked me, if I was ready for a new idea for storing data: How about not storing data anymore, but just relations? He said this would be similar to how computer games like "Doom" work with images: Such games don't come with a huge database of images, which are shown as we move through their game landscapes. Instead the images we see are generated from models - which essentially consist only of relations. That means the data (the images) is generated from those relations as needed. The data is not stored anywhere 1:1.

Well, this idea of storing relations instead of data sounded very interesting - even though I did not fully understand it. And I guess, I still do not fully understand it - but Peter Krieg got me hooked. I'm always open for unusual cool new ideas, and this seems to be one. Later that day he did a keynote titled "When PowerPoint meets Doom" where he tried to explain the idea to a larger audience. Reactions where mixed, but I immediately ran out to get his book "Die paranoide Maschine - Computer zwischen Wahn und Sinn". (Sorry, this book is only available in German; but nevertheless I can recommend it for his quite refreshing view on what computers are and how they need to change, if we want them to become more "intelligent".)

To substantiate his claim, that storing relations instead of data makes sense, Peter Krieg later showed me a demo app based on a so called *Pile Engine*. Pile is the theory behind the idea and has been developed by Erez Elul of Pile Systems over the past years. You can read about the whole theory/philosophy of seeing the (data) world as a universe of relations at PileWorks. But beware: Unless you have a certain inclination towards studying philosophical texts or reading about formal logic, well, it won't be easy for you follow Erez´ thoughts. However, even though I too am still

somewhat lost in the Pile world, I have the feeling, those guys are on to somewhat very interesting. I'm looking forward to meeting them next weekend in Berlin - and they can be sure, I'll have a lot of questions for them.

Ok, so Peter showed me this Pile Engine demo program which did a full text search on a large corpus of text. What can I say? It was impressing. Without using any index of words it was blazingly fast. But what was even more fascinating: You could search for any text string without performance loss. Usual full text search engines index all word in a text (except for stop words). That's why they are fast when searching for those words. But if you're looking for non-word strings, there index is of no help. If you'd be searching for "ch en" (as in "sear*ch en*gine"), this string would not have been index, so the search engine would need to scan the raw text itself. Very slow! But for the Pile Engine this does not make a difference. It does not need an index, so it does not care if you're searching for "engine" or "ch en". (If you're asking where such kind of query on texts might make sense, look at bio informatics: Those guys work with huge texts representing DNA molecule sequences, and they want to search for arbitrary combinations of letters in those texts.)

Now comes the interesting part: How does the Pile Engine accomplish its feat? Or more generally: What does Pile Systems mean, when they talk about storing relations instead of data? Where does the data go? Let me assure you, you're not alone asking those questions. And it took me some time to get used to "Pile thinking" - and I'm still an apprentice in it.

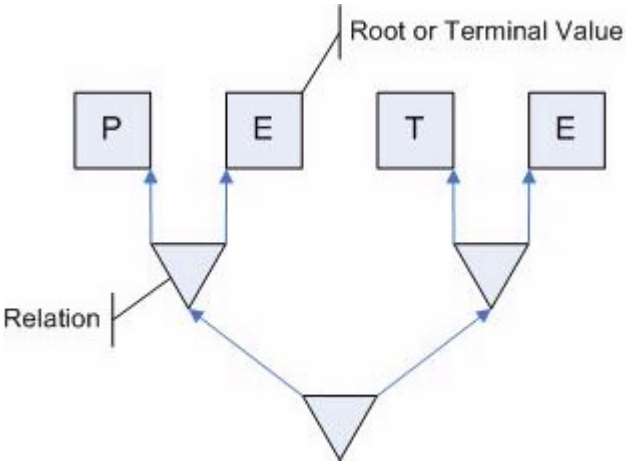
Let's start with simple text to get an idea about those relations.

Representing Text as Relations

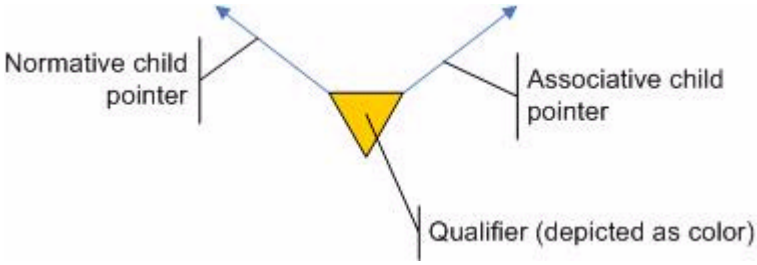
When we store text as data, then we store the text's bytes. That's so straightforward, we don't even think about it anymore. We never question what we're doing. The text "Peter Piper picked a peck of pickled peppers..." gets stored like this:

Well, what's wrong with it, you might ask? It's redundant! In the above excerpt from the famous tongue twister there 4 "P", 4 "E", 2 "R" etc. Each occurrence of a letter leads to storing the letter. That's what "storing data" means. Take that to the level of words: Each occurrence of "Peter" or any other word in the text needs to be stored. So the longer the text, the larger the amount of data you store.

Now, the Pile guys say: Why store the data, when you can regenerate it any time from relations? To store a text, they'd "relationize" it and end up with a data structure roughly similar the following:



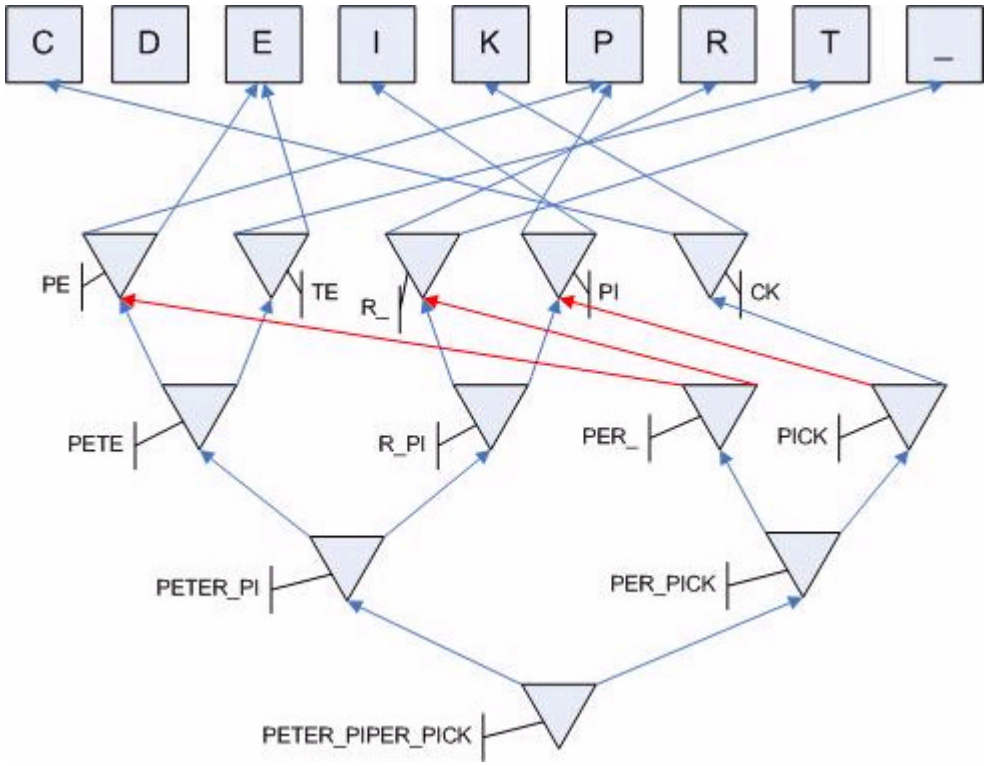
The letters are not stored one after another anymore, but instead are tied together by relations. A relation being a triple consisting of two pointers to child nodes and a qualifier.



Each relation is uniquely defined by its children (as far as I understand Pile). There can be no two relations pointing to the same children, i.e. all relations are distinct. (This is like with rows in Codd´s relational model.) (Don't ask me, why Pile calls the left pointer "normative" and the right one "associative" :-) Also, please don't mind, that Pile is calling nodes higher up children. You'll get used to this upside down view of trees.)

When you traverse such a tree from the bottom to the top in depth-first manner, you get "PETE" as the resulting text. So you effectively generate the original text from the relations, because the text itself as a sequence of letters is nowhere present in the system anymore.

But this is only a first rough picture to make the whole concept more palatable to you. Because when you look closer, the above representation still contains redundancy. What the Pile Engine really does is the following:

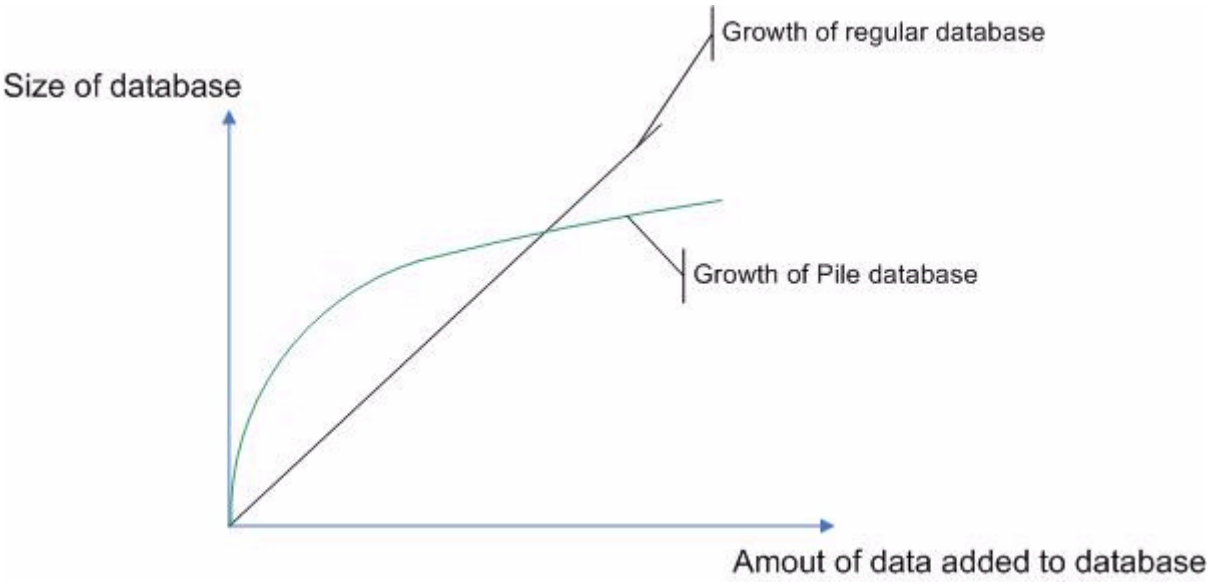


Somewhat confusing on first sight, isn t it? But if you look at it for a while, it becomes very clear: Each letter is only stored once as a terminal value. Then the text is broken into two letter pairs and each pair encoded as relation with the first letter being pointed to by the normative child and the last letter pointed to by the associative child.

On the next level, two of those relations are combined by a new relation, now representing 2 two letter relations, that means 4 letters - and so on.

The read pointers show, where reuse comes into play: In order to encode (or "relationalize") "PER_", to relations need to be combined. However, those relations ("PE" and "R_") already exist from encoding "PETER_" at the beginning of the text. So we save time and space by not encoding them again.

If you think this further, you ll realize, the longer a text is and the more combinations of letters of any length occur repeatedly, the more relations are reuse and less new relations have to be build. That means, the size of a Pile database does not increase linearly but asymptotically!



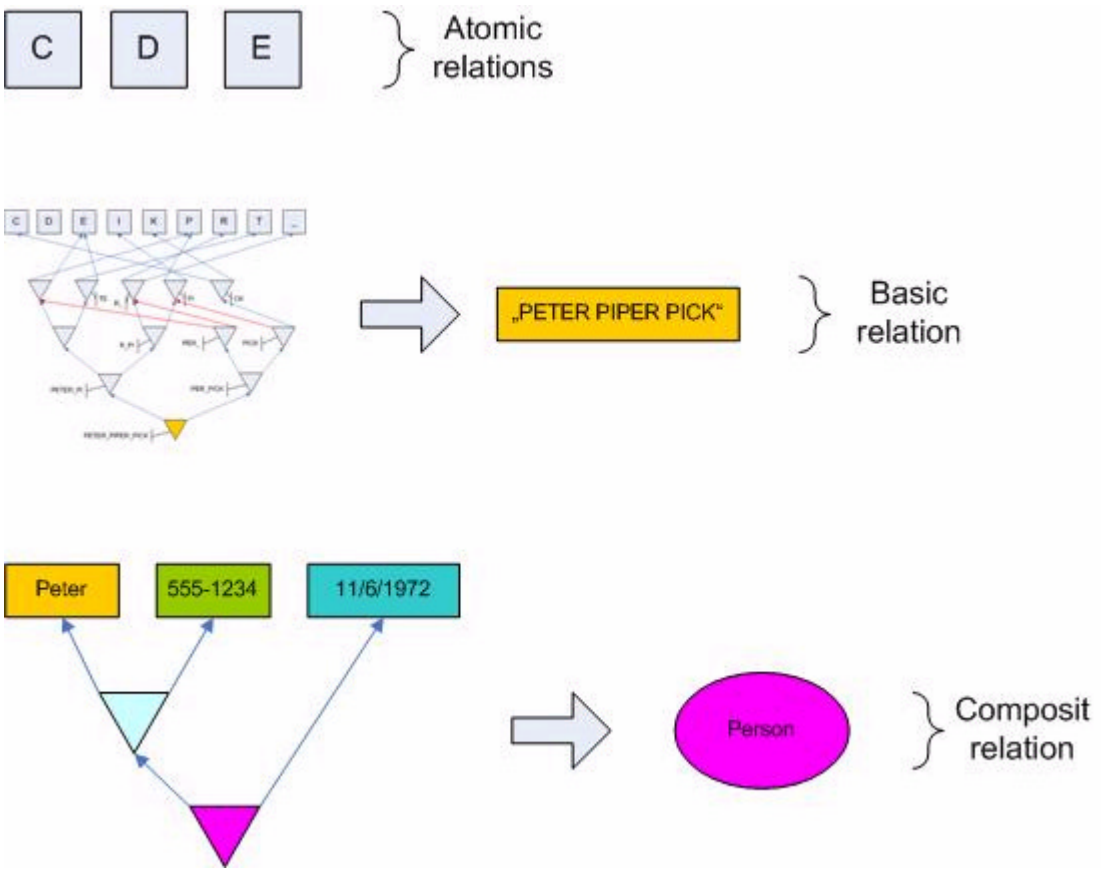
As long as the amount of data is small, Pile sure adds considerable overhead. A tree like the above sure at first requires more bytes in total than the simple text it represents. But as the amount of data increases and patterns start to occur repeatedly, Pile can reuse more and more relations already present.

Warming up: some terminology first

Saving some bytes on hard disks might not give you a real kick. Being able to search through texts very quickly might not be what you're looking for. But what about very flexible database schemas? Or how would you like to never have to think about normalizing data again?

The real coolness of Pile lies in its fundamental data representation which automatically relates entities, if they share some "characteristics", I'd say.

In order to show you that, let me coin some terms:



The letters we saw earlier I ḋ like to call *atomic relations* because they are the roots, the terminal values of any Pile tree. In the end, all other relations directly or indirectly relate atomic relations to each other. (The reason why I still call these terminal values "relations" is, that you can further relationalize them into relations of bits. Each terminal value in the end is just a relation of relation of the "nucleons" 1 and 0.)

Since most of us are not concerned with just storing texts but with storing mundane data entities like addresses or invoices, I think it ṡ necessary to introduce (at least) two levels of abstraction over atomic relations:

Basic relations I call a hierarchy of relations rooted in atomic relations and forming a larger whole. This whole or unit of textual data can be a name or a birth date or a two page memo text. Basic relations so to speak are a bag of very simple relations with a label stuck to it. I use the qualifier of the relation triple as such a label. In Pile notation a triple looks like this: (n [Q] a). n being the normative child, a being the associative child and [Q] being the qualifier. In my pictures I try to use colors for [Q], but sometimes I might assign it an identifier like [name] or [person]. ([Q] of atomic relations and the inner relations making up a basic relation I leave undefined; they are not important for the further discussion. But if you like, you can call the [atom] and [basic].)

When pointing to a basic relation the pointer references the bottom most relation in the upside down tree constituting the basic relation. You can always zoom into a basic relation and see its details.

On the next level above basic relations are *Composite relations*. Composite relations combine several basic relations (or other composite relations) to form an even larger whole. They too have an explicit color, i.e. qualifier.

If you want to map this to the world of classes you can say: basic relations are like text fields and composite relations are like classes containing fields. For a start you could liken the above picture to a class definition like this:

```
class Person
{
  string name;
  string phone;
  string birthdate;
}
```

Then the actual Pile of the depicted composite relation would contain one object:

```
Person p = new Person();
p.name = "Peter";
p.phone = "555-1234";
p.birthdate = #11/6/1972#;
```

Networks of data instead of tables

Ok, ok, enough with theory. Here comes the fun part: Imagine how a Pile database of persons and appointments look like. Here are the composite relations with their basic relations:

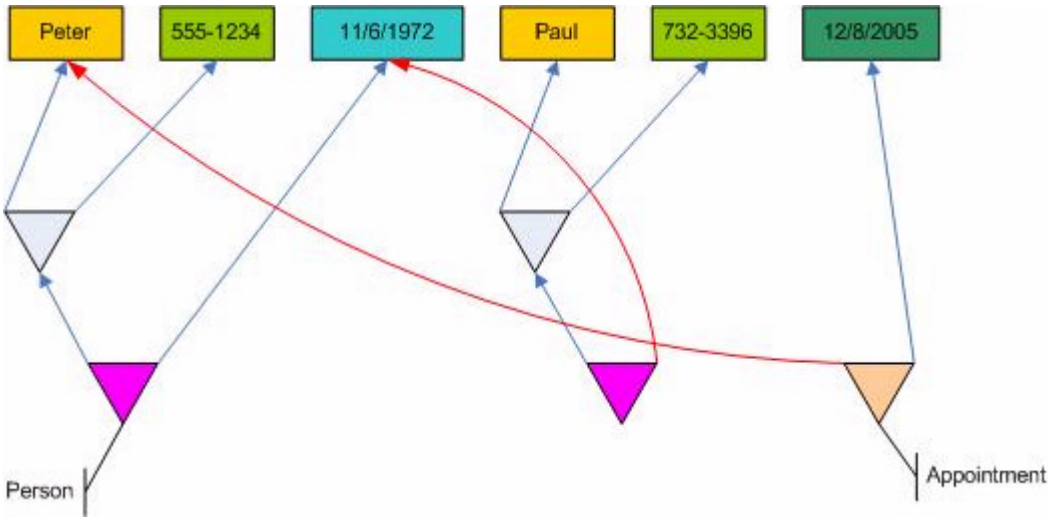
```
Person(name, phone, birthdate)
Appointment(name, date)
```

Think about it for a while...

You see the difference between the world of objects and tables? Remember: Pile doesn t store data redundantly. If you have two Persons and an Appointment like this

```
Person("Peter", "555-1234", "11/6/1972")
Person("Paul", "732-3396", "11/6/1972")
Appointment("Peter", "12/8/2005")
```

the database would look like this:



The name "Peter" and the birthdate "11/6/1972" would only get stored once. However, I don't think this space saving aspect is important, but rather that a Person entity and an Appointment entity share the same data as well as two Person entities share the same data.

Because: sharing data means being linked, being in relation to each other.

Pile thus very, very naturally and implicitly links data. You don't need to set up a specific relation between entities like Person and Appointment using an explicit object reference or a foreign key. The are automatically associated with each other by their data.

Ok, to fully see that, you need one more piece of information: relations in Pile are bi-directional. So if some relation R's normative child is relation C, then not only R "sees" C, but also C "sees" R. C is R's child and R is C's parent. When traversing a Pile tree you can move up the hierarchy from parent to children and there are always only two children to each relation. But you can also traverse the Pile tree down from the children to the parents. However, each child can have an arbitrary number of parents! The basic relation for "Peter" has two parents: a composite Person relation (indirectly via the internal unqualified relation) and a composite Appointment relation.

So, what a Pile Engine does is weaving a network of data instead of putting data in little bins which have to be manually connected. Pile frees you of the need to think about setting up relations in advance like with relational databases. Maybe instead you could say, relations in Pile databases just happen :-) Wherever Composite relations share the same basic relations (or Composite relations) they are automatically connected. That (!) I find very fascinating.

Information modeling the Pile way

This cool automatic association of data has some implications on how you would model data for a Pile database. Instead of starting top-down with classes or tables or Composite relations, you should start with basic relations. Once you know all the entities you want to store in a Pile database (e.g. addresses, invoices, appointments, products, categories...) you identify the basic relations. View them as value types and blobs of text. (I'm not sure yet, if a Pile database should handle allow for other terminal values than mere characters.)

For the above example we would arrive at these basic relations: name, phone, birthdate, date.

If a name in a Person is the same kind of beast as a name in an Appointment, we just need one basic relation for it. Both, Person and Appointment, store some kind of date information. Should we thus use the same basic relation or distinguish between them? I'd say, a birthdate is sufficiently different from an appointment date to justify two basic relations. But an invoice date and the date on a photograph might be viewed as of the same kind.

Having identified the basic relations we already know, that all composite relations sharing them will automagically be connected! I've to say it again: very, very cool!

Next you'd look for composite relations made up of basic relations, e.g. Person and Appointment. Then you could move up the abstraction ladder and look for composite relations combining other composite relations, e.g. Invoice(Person, Shippinginformation, Item).

Keep in mind, Pile's relations are always bi-directional and 1:n! No need to specify cardinality. (I haven't thought much about constraints on relations yet, sorry.)

And that's it! You're finished with your Pile "schema". Lean back and think about the beauty of it: If you come up with some new composite relation or want to rearrange existing ones, no data has to be moved around or copied. You want to add a music library to your ERP Pile database? No problem! Just think about which basic/composite relations you can reuse. You sure can reuse the basic name relation for the artist's name. And you could reuse the basic date relation for the release date of an album. Or you could define an Artist to be Artist(Person, genre).

As soon as you start putting music information in your Pile database it gets automatically associated with your ERP data. Maybe you find out,

your famous artist has already ordered from you, because his name connects him to an invoice :-)

Oops, this leads me to the topic of identity.

Handling identity

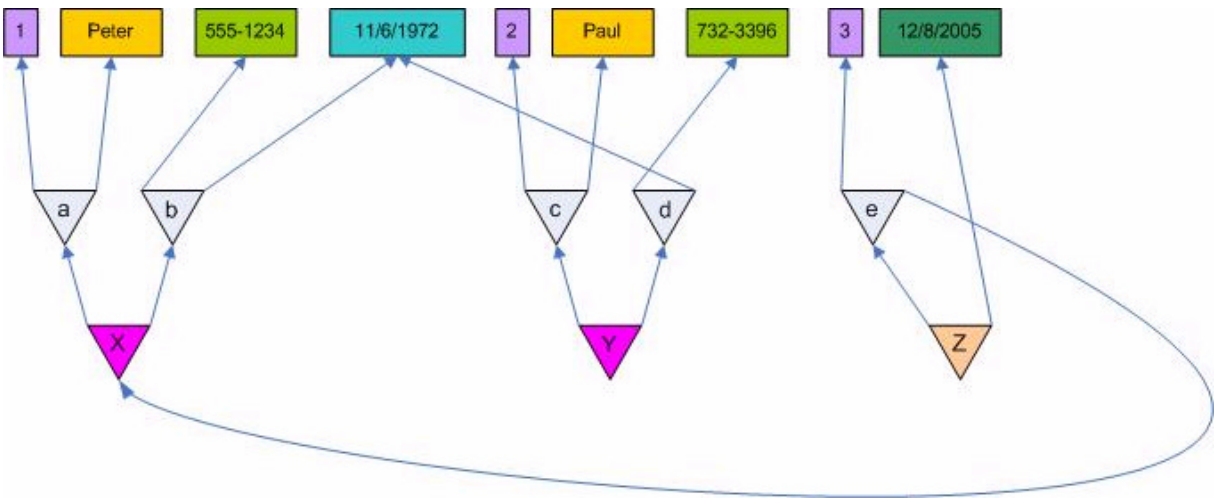
Each relation is identified by its children. All relations are distinct. Well, that's cool for automatically weaving a network of data - but sure can lead to some misunderstandings when not explicitly taken into account.

There are certainly many persons with the name of Peter. How would you know from looking at the above simple Pile database with Persons and Appointments, if an Appointment belongs to this Peter or that Peter? There is no way to tell.

So as with relational databases you need to assign explicit identities to units of data that actually describe some individual entity like a person. I suggest to introduce a basic ID relation for that purpose.

Basic relations: ID, name, phone, birthdate, date
Composite relations: Person(*ID*, name, phone, birthdate),
Appointment(*ID*, *Person*, date)

The new Pile schema defines Persons and Appointments to have an identity and Appointments to be associated with Persons instead of names.



Do you "feel" this to be still different and much simpler than modeling a regular relational database schema? Good! :-) Including a Person in the definition of Appointment is so much more natural than setting up a PK-FK-relation.

Please also note: Even though now Appointment is related to Person instead of name, it is still possible to start with just a name and find all Appointments for it (i.e. Appointments referring to Persons containing this name). Only some levels of indirection have been introduced, but no fundamental switch like from "column in table" to "column in different table and PK-FK-relation" was necessary. Pile just knows relations whereas the traditional relational model knows rows/tables and keys.

Querying a Pile database

Sticking data into a Pile database is one thing. But how to get it out again? How to query for a Person or all Appointments of a Person on a certain day?

As far as I understand, querying a Pile database basically is pattern matching. You hand a relation to the Pile engine, e.g. name("Peter") and, for example, it returns the list of parents of that relation, e.g. a("1", "Peter") (see above picture). The query birthdate("11/6/1972") would return b("555-1234", "11/6/1972") and d("732-3396","11/6/1972"). Asking for c("2", "Paul") would return Y(c,d).

Please remember: relations are defined by their children. So in reality a simple query would just consist of the child relations, e.g. instead of a("1", "Peter") you'd say ("1", "Peter") because "1" and "Peter" taken together are equal to a. That means, a Pile engine should return a full relation including all parents when queried with the "contents" of a relation. Sending query ("1", "Peter") thus would result in a, sending ("2", "Paul") would return c.

(Here we stumble across something that's still a mystery to me: How should a function look like that maps relations (or addresses) n and a to a new relation r(n,a)? How can r = f(n,a) look like? I find this a difficult question. The sample Pile engine seems to have solved this problem, but I don't yet understand the solution. As soon as I do, though, I'll let you know ;-)

With a relation (or a list of relations) in hand as the result of a query, you then can traverse them up and down to your liking, e.g. with relation a in hand, you move up to its parent X and then up to its parent e and from there to Z. Small easy steps to get from a name with an identity to an Appointment.

To form more complex queries, trees of relations need to be assembled, e.g. (("1",*)(*,*)). This query would return X, because it has a normative child whose normative child is "1" (The * I use as wildcard for a relation.) Query ((*,*)(*,"11/6/1972")) would X and Y.

Of course even more complex queries can be expressed using trees. What I don't know yet is, how to best express logical operations like OR/NOT (because a single tree essentially is a query where the operands are ANDed). But there sure is a way... ;-)

Conclusion

That's how far I understand Pile. The existing Pile demo engine does not implement all this, as far as I know. It just works on plain text. But once I have the f(n,a) function, I'm pretty sure, I can set up a little Pile engine for non-text entities pretty quickly. That will be a fun programming experience... :-)

But of course, there's still much to be learned. I've to dig into the Pile literature at www.pileworks.org and talk to the guys making up all the theory behind it. Interestingly I just received an email from one of them which made some things already more clear. For example I learned to see not only trees growing from the bottom up - but the many trees hanging down from the roots. The root "11/6/1972" for example is the root of two trees.

Also I want to think more about how far Pile is related to Prolog and Object Role Modeling and Lisp. When sketching Pile trees/networks or writing them down, sometimes feel reminded of these concepts/technologies. But I have an inkling Pile goes much further...

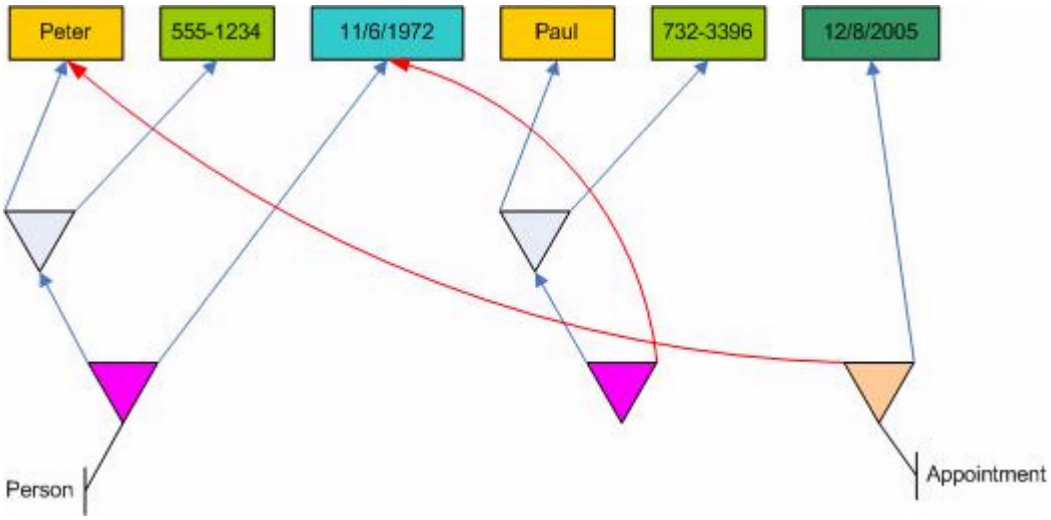
Whether Pile is a revolution or just a nice idea or a solution to just some particular data storage/retrieval problems, I don't know yet. But I think it's cool :-)

(Posted on Thursday, December 08, 2005)

2. Storing Relations Revisited - The Hanging Trees of Pile

Currently no day goes by without me learning something new about Pile - the idea of storing relations instead of data. Yesterday I introduced you to Pile and talked about trees growing bottom up.

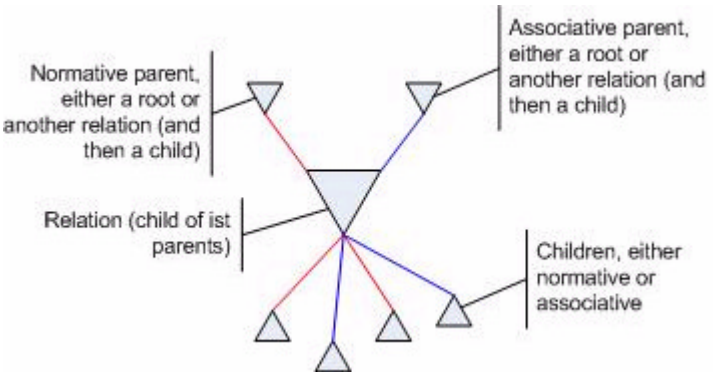
The roots of these trees are at the bottom and the leaves are at the top. Nodes are pointing upwards.



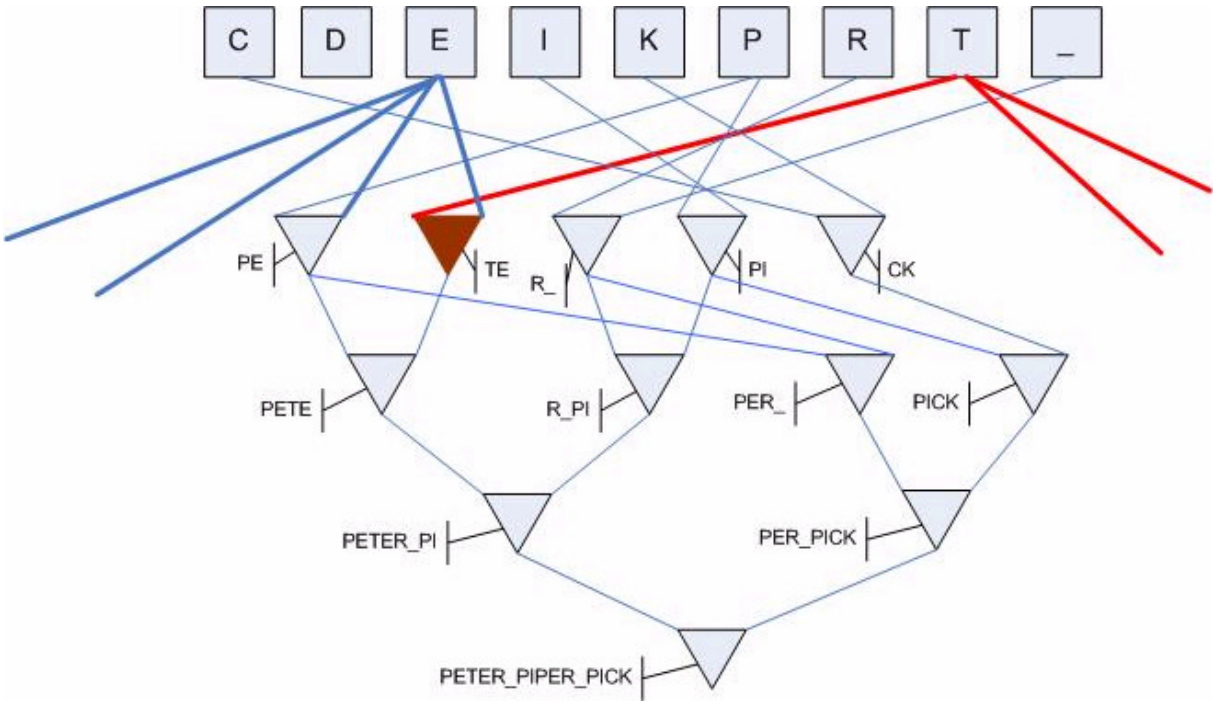
Also I had no answer to the question, how to get to a parent node from its two children. I asked for some function p=f(n,a) (n and a being the left/normative and right/associative children of a node) to "calculate" the parent.

Now, since yesterday I received emails from www.pileworks.org - thanks Miriam! - and have learned, I was on the wrong track. Although my general description is not altogether wrong, it somewhat mixed up things a bit. So let me start to give you an update of my understanding.

First of all I learned to see the tree(s) the right way. They actually start at the top and grow down, it's hanging trees. Each terminal value truly is the root of an arbitrary number of trees. That also means, children are located below parents. And once we switch views we clearly see, although each node has only two parents (because it's a relation connecting two other nodes) it can have any number of children.



Secondly I realized, all query pattern matching always starts at some root/terminal value. That means, when looking for "te" in the Pile for "Peter" I start out with "t" and "e" and want to find the te-node - which is the only child node belonging to both (!) terminal nodes.

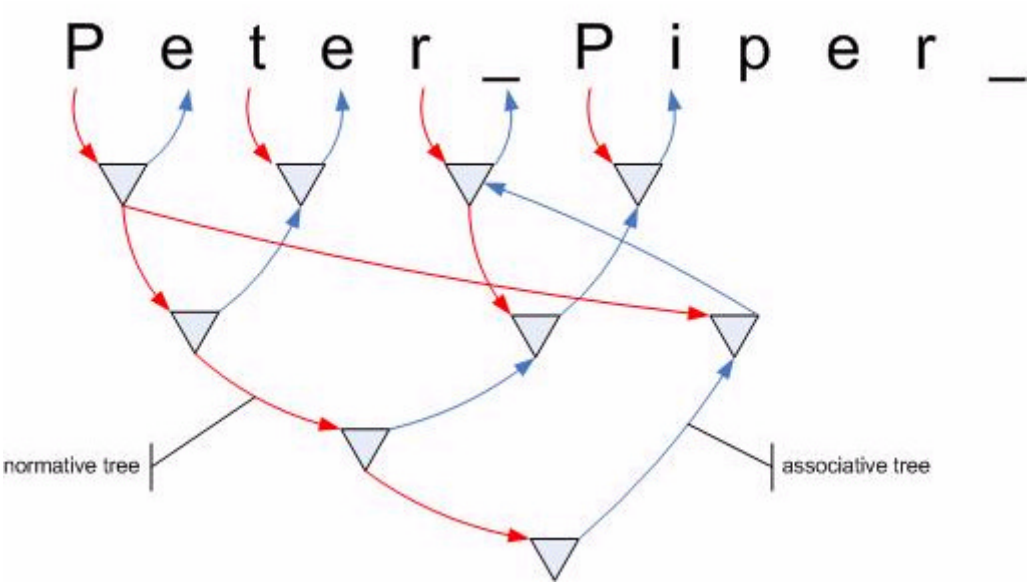


So what looking for ("t", "e") means is: Calculate the intersection of the sets of child nodes of the "t" and the "e" node. The intersection - if not empty - will only contain one node.

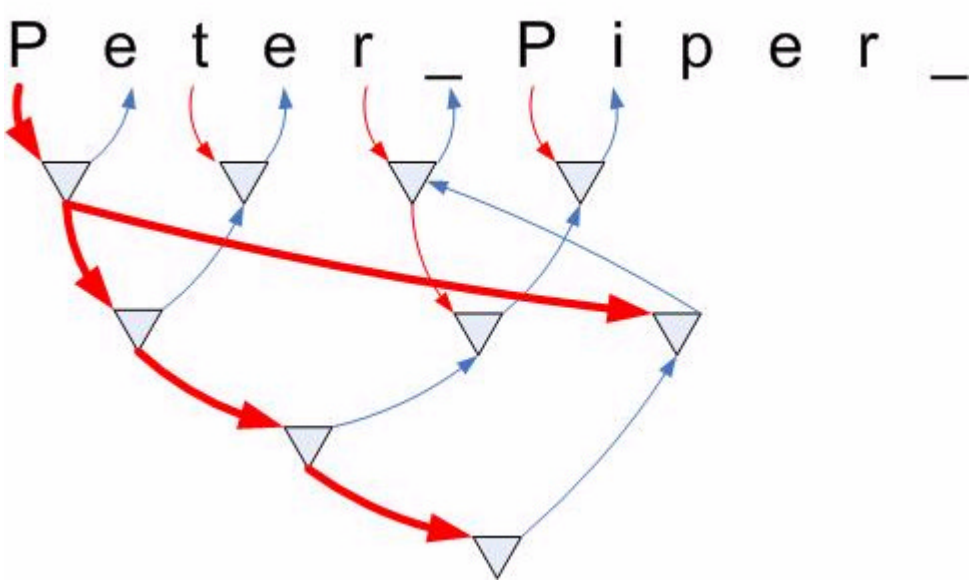
After I understood that, I realized my problem of finding f(n,a) vanished. There is no need for such a function. Finding ("t", "e") is just a matter of looking thru lists of child nodes. A simple programming exercise. I then was able to develop a tiny Pile engine within one or two hours which is able to store texts and search for patterns. The core Pile database code is just some 150 lines of code. That was really fun! (However, I ve to admit I

still have some problems searching for patterns. Looking for "Pe" or "te" in "Peter Piper..." works just fine; but looking for "et" does not, because there is no relation between the two; there are only relations ("P", "e") and ("t", "e"), but not ("e", "t"). Hm... But I'm sure the PileWorks guys will help me out. I sure still lack some understanding of the Pile theory.)

Thirdly I learned from what Miriam wrote and what I saw in a Powerpoint presentation (which I highly recommend to you, if you're interested in Pile) at www.pileworks.org that Pile distinguishes between two kinds of trees: normative trees and associative trees. The implications of this are not yet clear to me, but I need to adapt my depiction of Piles. If you look at the PowerPoint presentation you'll see, the PileWorks guys have their own diagrams. I kind of find it hard to get used to them. So I try to capture the essence of the two trees in a different way.

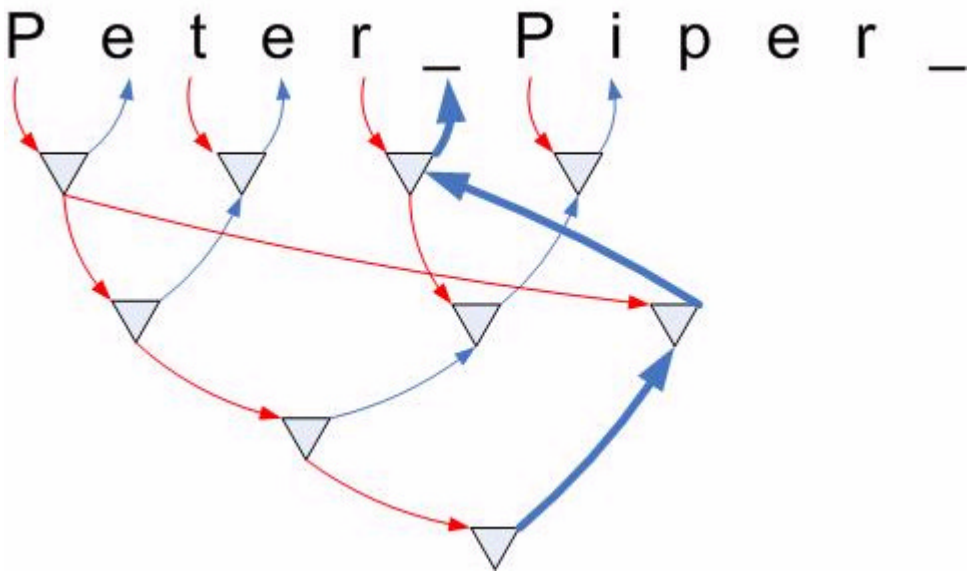


The normative tree is a "real" tree. It starts at a root and spans nodes on lower levels. Build it by following only (!) normative read arrows. The normative tree of "P" for example would be:



The normative tree for "r" on the other hand would contain "r", r_:("r", "_") and r_Pi:(r_, Pi:("P", "i")).

Associative trees point upwards - although the terminal values are still roots and nodes closer to the top are parents. Here's the associative tree for "_" (don't mind there are no branches in it; that's due to the simple example):



The associative tree is build by only following blue lines; either start from a root node and traverse nodes connected by blue lines - or start at the bottom and follow only blue arrows.

Ah, sigh, I feel relieved now that I clarified this ;-) The pictures now look much more interesting, don't they? :-)

However, I'm not sure what the distinction between those kinds of trees actually means. Does it help to solve my pattern matching problem? Should I change the data structures in my tiny Pile engine to reflect the changed direction of the arrows? I've to ponder this a while... If I find out something, I'll let you know.

(posted on Friday, December 09, 2005)

3. Musings On Relations - or: WinFS is Not Enough

Have you had a look at WinFS? No, you should. It's cool. Or maybe I should say: It could be even more cool, if it didn't stop too early.

The basic idea behind WinFS is: set your data free! Unlock the gems hidden in large databases! Microsoft's fundamental insight underlying this is, the future is about relationships between data and you can only set up relationships between separately addressable and accessible units of data.

Within a single relational database you can setup relations between rows in tables. The relationship between smaller informational units (columns, fields) is implicit by arranging them within a single table. But what about relationships between data in different databases? Although that's possible, it's not really what you want to do in your day to day business. Databases thus draw a boundary around data. From a security point of view this is of course beneficial; but from a data reuse point of view, this is a contra productive.

Wouldn't is be nice, though, to not be concerned about the database, the bucket where an address is stored in? Wouldn't it be nice to be able to relate an existing address to a new information you want to store, which not necessarily comes to rest in the same database as the address? Think about linking your latest digital pictures to contacts you already have in Outlook. Think about adding tasks in Outlook to an order workflow process in your ERP program.

All those scenarios are either not possible today or very hard to achieve. That's the reasoning behind why WinFS is needed. With WinFS there is no more Outlook .pst file hiding all those precious contacts, tasks, emails and appointments behind a wall. Instead these items are set free to float around as separately addressable and accessible data units in the file system space. And you can do the same with your own data.

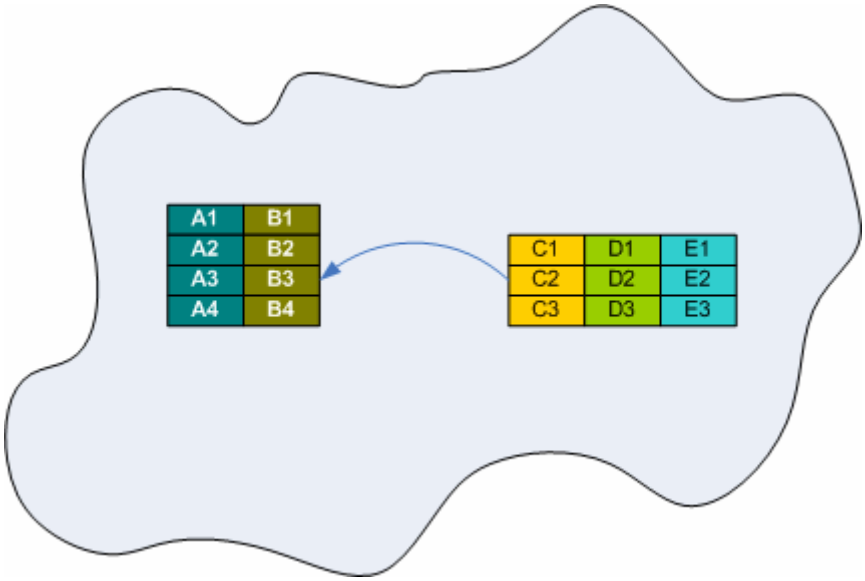
Once there are comparatively small data items floating around rather than being penned up in an ever increasing number of (incompatible) databases, you can start to relate those data items to each other. That's so cool! It let's you reuse information in different contexts/programs instead of reentering (or importing/exporting) it over and over again.

However, although WinFS breaks up the database barriers around the information nuggets it remains in the world of relational databases. No, not just because SQL Server 2005 is the foundation on which WinFS is built. Rather because relationships are still an afterthought and a second class citizen in data modeling.

From data to associations

File systems, XML, RDBMS, ODBMS and also WinFS are all data centric. Data is the main concern. Storing data is the most important task of an RDBMS. Databases are about recording data, making it persistent. Well, that sounds reasonable, doesn't it?

The following picture depicts the current thinking: Data is arranged in fields stuffed into a row. Rows can point at each other. The data items (fields) are related implicitly and explicitly on two levels: implicitly by putting them next to each other in a row and storing all rows of the same kind in a table, explicitly by foreign keys.



The relational calculus is good in describing sets. But it's bad at describing relations between data in different sets. Explicit identities (primary keys) need to be introduced and normalization is needed to avoid update inconsistencies due to duplication of data.

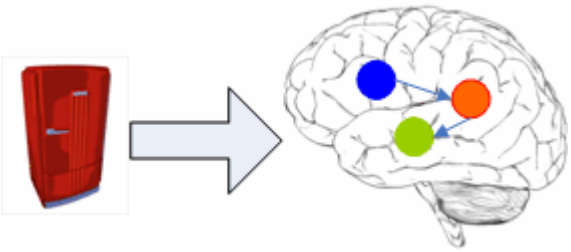
To say it somewhat bluntly: The problem with the relational calculus and RDBMS etc. is the focus on data. It's seems to be so important to store the data, that connecting the data moves to the background.

That might be close to how we store filled in paper forms. But it's so unlike how the mind works.

There is no data stored in your brain. If you look at the fridge in your kitchen, there is no tiny fridge created in your brain so you can take the memory of your fridge with you, when you leave your kitchen.

Instead the fridge is left where it is, right there in your kitchen. However, what is stored in your brain are associations of all kinds. In fact, your

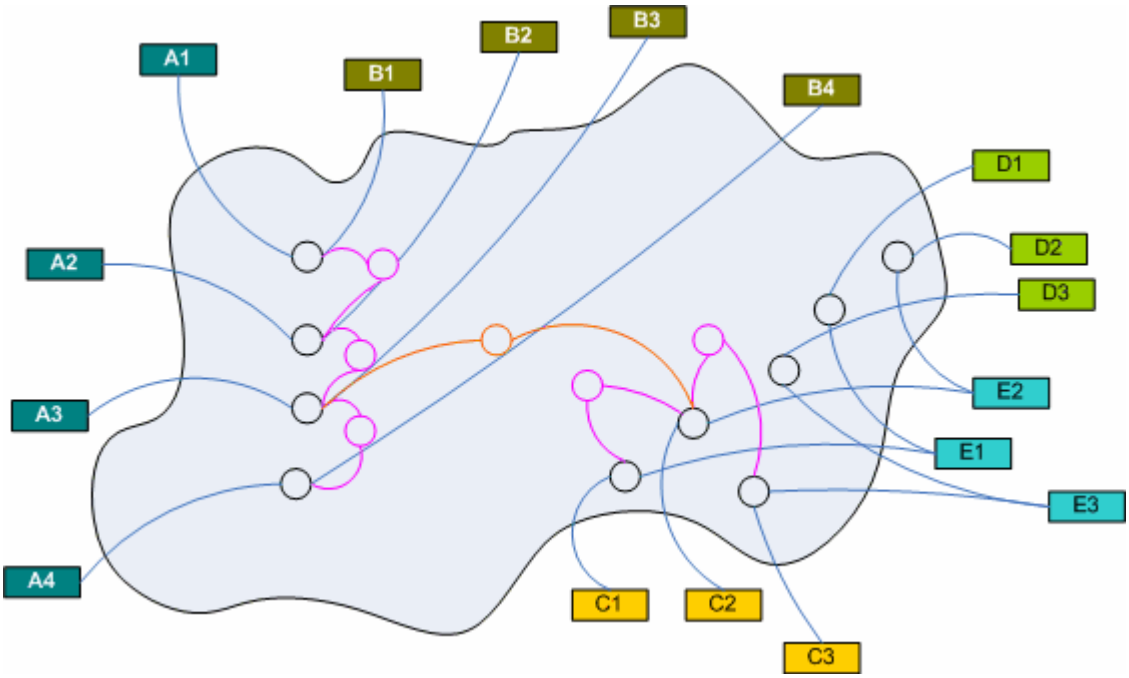
brain can only store "immaterial" associations. (Let's neglect for the moment, that those immaterial associations need to manifest themselves somehow, e.g. electrical signals, chemical substances, or cell growth.)



The fridge causes the brain to setup internally an unknown number of associations. Thus, the brain works just with relations/associations and not with data or "the real things". The brain has its own representations for the data. There is not data in the brain; rather the data itself stays outside the brain.

So "the real thing", the fridge, is not in the brain, but instead some kind of, hm, "token" or handle. Or maybe there is not even a "token" for a whole fridge in the brain, but a large number of handles for parts of a fridge? Or what seems to be even more likely: the brain knows nothing about fridges and fridge parts, but just about very, very simple visual structures like points, edges, colors. So the mental representation of a fridge is a set of relations between such basic structures/concepts. Then the brain does not need "tokens" for real world entities, but just for basic structures/concepts to relate them to each other.

Ok, why am I telling you all this? What does this have to do with WinFS? Well, it's about a completely different way to deal with data (or things). To map what the brain does to the software world means, removing the data from the "system" leaving only associations:



Within the "system" there are just associations and associations between associations. The data is outside the "system". Compared to our traditional thinking this kind of "system" is homogeneous. There are only associations. That's it. There is no distinction between associations and data or different kinds of associations (implicit vs. explicit). Associations or relations are first class citizens in this kind of "system".

And since there are no different kinds of data and no more "data buckets" like tables or columns, any association can be associated with any other association.

When you define an RDBMS schema you explicitly set up which kind of data (rows) can be connected to which other kind of data. You try to foresee what could possibly make sense in terms of associating data. Well, that's what the Outlook team did in the past. They said: Well, we think, users want to associate a contact with an appointment or an email with a task. So we stuff everything in a nice little database.

But then, users thought differently. All of a sudden, they wanted to associate an Outlook contact with an invoice - without success, because the Outlook developers had thought they could foresee the future usage of certain data.

This dawned on Microsoft and they now come up with WinFS. Great! Or not?

No, not so great, although still technologically cool. Because WinFS still requires you to think in pretty large bins of data (e.g. a contact, an appointment). Although you can set up relations between those smaller

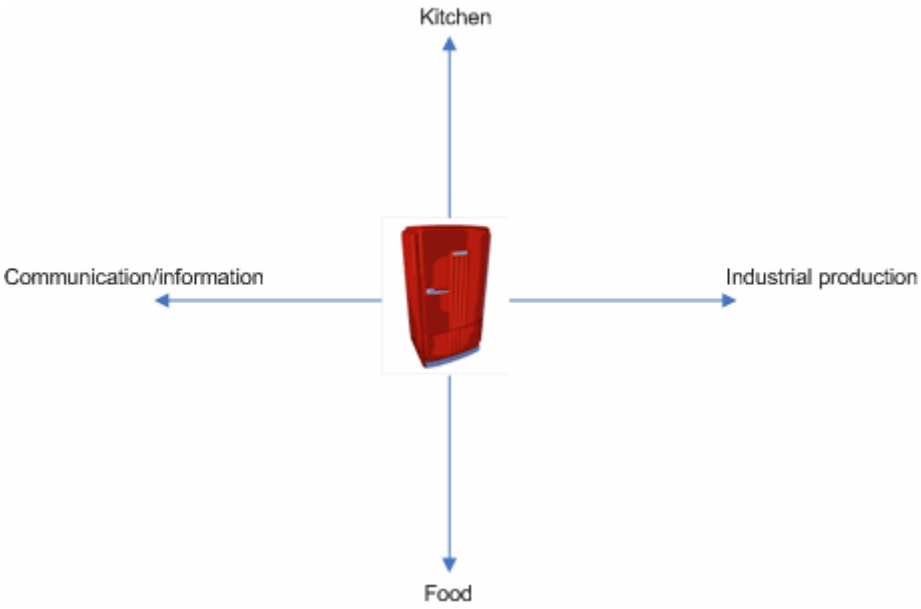
bins, WinFS still is about data first - and only then come associations between data. It's a heterogeneous system.

Your brain, on the other side, is homogeneous: the brain knows only about associations. Because that's the only way to deal with an unpredictable world where you cannot foresee how "things" might look and behave and how you might want to associate fine grained basic concepts like points or coarse grained concepts like fridges with each other. The brain knows about causality/time, points, edges, space, that's probably pretty much it. Those concepts/structures are its *roots*. All else is just associations between those roots and other associations. Billions, trillions of them. And it works :-)

So why stop where WinFS stops? Why not take WinFS to the max? Why not radically chance of view of the database world? How about association bases or connection bases instead of data bases?

A world of associations

The gain of a new view on how to deal with data would be an explosion of possible associations. When you look at your fridge, you immediately can see it in different contexts: there is the context of "kitchen" where the fridge is one of many appliances, then there is the fridge as a manufactured product pointing to a history of industrial production, then there is the context of "food" which the fridge keeps, then there is the context of "information" because you put post-it! notes on the fridge's door, and so on...



The fridge is at the origin of a multi-dimensional space of contexts. Many different contexts intersect in a fridge. That's so natural to all of us... so why not treat data the same?

Switching to a new view on dealing with data is thus a switch from one context to multiple contexts. In an associative system and data unit (external to the system) can exist in any number of contexts, just depending on the associations between it and other data units or other associations.

So if associations are the real value of data, because they put them "in perspective" aka into different contexts, then how to get more out of an associative system? Well, by forming as many associations as possible (or as makes sense for a certain observer).

Since the number of possible associations is determined by the number of data units, it's best to see to maximize their number first. And that's exactly where WinFS falls short.

Although WinFS promotes disassembling databases into their rows (objects, e.g. contacts, tasks), the resulting data units not only stay within the system, but are also still fairly coarse grained. A whole contact can be associated with a whole appointment.

But why stop there? Why disassemble the data further in order to be able to generate even more associations? Who's able to foresee that associating a whole task with a whole invoice is all that users ever need?

Maybe I want to navigate (by traversing the maze of associations) from a single date in an appointment to contacts with this date as a birthdate? Why not reuse names from contacts in the context of appointments? And I mean just names.

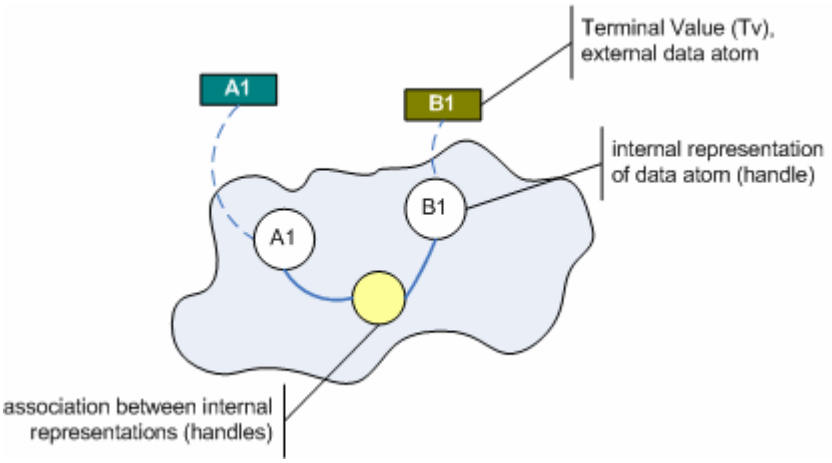
What this would mean is blowing up those WinFS data units (objects) into very small pieces, data atoms. Each atom being some data unit which cannot be split into smaller pieces.

Single letters come to mind as candidates for a data atom. (The bit values 1 and 0 would be the true data atoms, but even though it would be possible to build a "system" on them, since letters are just associations between 1s and 0s, I find this low level a bit unwieldy.) Pictures might be larger data atoms because their individual bytes might indeed make no sense in other associations - but who knows.

In the end, an associative base system should be data atom agnostic. If might know, data atoms are streams of bytes and might offer to store them as is. But then... why should it know about data atoms? They are of no use within (!) the system. So an associative system should provide just

one operation concerning data atoms: create a handle for a data atom, if you ask it to.

The associative system then looks like this:



Whatever is outside the system, the system does not care about. However, in order to setup associations with the outside data atoms, the system has to have some kind of internal representation, that's why the system needs to be able to generate - *ex nihilo* so to speak - handles for external data atoms (or terminal values). What those handles mean, which terminal values they stand for, whether it's a single letter or a multi-megabyte picture, the associative system does not know.

Conclusion

Now, think about the implications for a while...

Such kind of associative base, an AB instead of a DB if you want, would not store data, but rather would generate data from data atoms as needed.

Take a text like the Bible: If you defined the 256 ASCII characters as to be the atoms, then there would be no bible text data, but just some 800,000 associations between those 256 terminal values and other associations. (I know this figure, because I've implemented such a system in C# and loaded the 4.5 MB King James Bible into the AB.)

Still, though, I can losslessly generate the complete Bible text upon request from those associations. It's just a matter of recursive descend in a binary tree. But what's more important is, no combination of letters would need to be stored twice in such an AB. Each association could be unique. No more duplication of data.

This, though, not only leads to maybe saving some disk space, but it means, when looking for the pattern "Enoch" I immediately get all contexts in which Enoch appears in the Old Testament. Starting to look for patterns from the handles for their terminal values immediately leads to all associations which connect to those patterns.

But this is only a simple example and you might say, hey, this is what full text database searches are for. And you're right! However, a full text database stores the data twice: once as the data, and once all the major words in the index. Also a full text database usually limits you to searching for words. If you want to look for arbitrary patterns, e.g. "o b" in the text of Hamlet, then you're lost. A full text search engine would not return "to be". For an AB engine, though, this would make no difference. And that's important, for example, in searching for gene sequences in the field of bio informatics.

I can understand, though, if you find it difficult to switch your thinking from data centric to associations only. It took me 2-3 weeks and I'm still working on it. But the potential of this switch seems to be huge! Each day I learn something new. It almost feels as if I'm in love :-) I'm almost blocked from doing other work, because my mind reels with the possibilities and implications. That's the reason, why I needed to write this blog entry. I needed to get this out of my head to move on.

Just yesterday I talked to a developer of an ODBMS about all this. Fortunately I was able to depict all this to him on the phone - and he immediately grasped the idea. He even corrected me when I thought about maybe defining whole data fields (e.g. a name, a birthdate, a zip-code) as data atoms to gain performance from having a "regular" database engine to index them. He said, no, that's not necessary, because all those values (consisting of characters) can be indexes using associations within (!) the AB. And he's right! I felt so relieved: Such an index would be just another context in with terminal values appear.

The beauty of an association only system is very striking, I think. So while WinFS is a cool idea compared to today's situation, WinFS is but a small step towards really setting data free to be associated in a million ways like in our brains.

(Posted on Tuesday, December 20, 2005)

4. Beyond WinFS - Let Associations Rule! - Or: An Introduction to Pile for Mere Mortals

Yesterday I wrote about why I think WinFS is a step forward - but in the end will not really solve the problem of the penned up data. WinFS´ data units (the objects, e.g. contacts, appointments and what not) simply are still too coarse grained. And even though WinFS puts relations between data units (or entities) more into the center of the stage, it is still bound by the general world view of "data is most important". However, as long as data is in the focus, associations between data are second. And as long as associations are second, we ll have a hard time to map the multi contextuality of real world entities to software.

Ok, enough said about the limits of current databases technologies. You rightfully ask, how could a different view of the world look like in software. I ve already written about where I see a promising alternative approach rising above the horizon. See my blog postings on "Storing relations instead of data" and "The hanging trees of Pile". Unfortunately these were early descriptions of something I did not fully understand. Although I m no "master of Pile" I guess my understanding has evolved and I can now speak with some more confidence, since I ve implemented a working Pile engine in C#.

So what I d like to do is provide you with a more systematic introduction to the world of Pile which is all about this new view on handling information. Some two weeks ago I met Erez Elul, the "inventor" of Pile, and Miriam Bedoni his friend who form the main Pile think tank :-) and are constant sources of new insights into the world of associations. They corrected my understanding and deepened it, so I feel confident I can now present the Pile framework for thinking about associative systems in a much clearer way.

Erez (on the left) trying hard to explain some Pile intricacies to my old unenlightened self ;-)

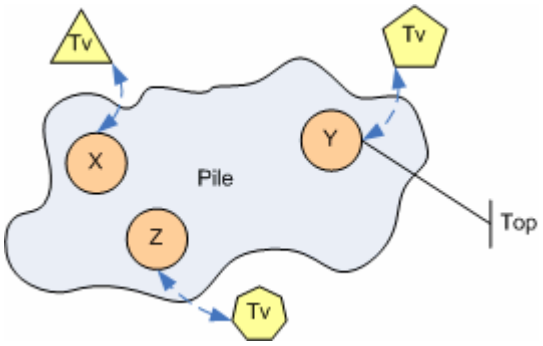
On word of warning, though: I ll deviate from the terminology and notation Erez and Pile Systems are using and I ll add stuff, where I think it makes the whole topic more palatable for mere mortals like you and me who have been raised in a data(!)base world. But I hope, once you look at the original documentation about Pile (for example at www.pileworks.org) you ll find your way around.

Ok, now let s start our journey thru the Pile universe...

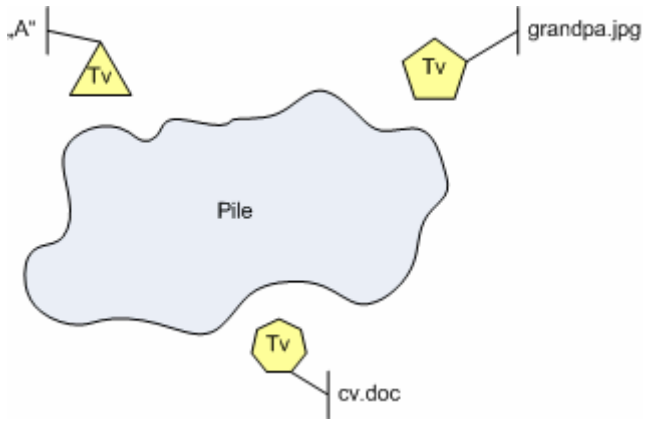
Terminal values and Tops

When I first heard about Pile and "storing data without data" but rather generating data from associations when needed, I asked myself, if that was possible. Where should the data I want to see on my screen come from, if it s nowhere in the system? It took a moment until I understood, there still needs to be data when working with associations, but it s just the few data atoms which are needed. Whenever you need the "full data" you assemble it from the data atoms by traversing associations.

Pile calls the data atoms as described in yesterday s posting *Terminal Values* (Tv). A Tv is not part of a Pile (that s how a system of associations is called) or Pile base (as opposed to data base); a Tv is always outside the Pile and a Pile engine (the software managing associations) does not know anything about it. It is ignorant about whether a Tv is a single character or a .jpg-File.



Although the actual data is outside a Pile, there needs to be some way to build associations with it. So there needs to be some representation of Tvs within a Pile. These representations are called *Tops*.



The name Top is supposed to suggest, they are at the very top of the many trees of associations existing in a Pile. Where those trees are I ll explain in a minute.

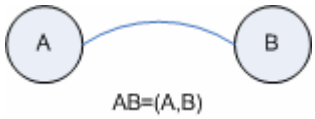
Each Tv can only be represented by one Top. And the arrow between Tv and Top in the above picture means: There is a way to get from a particular Tv to its Top and vice versa. But mind you, this mapping between Tv and Top is not part of a Pile or the Pile engine! It s the responsibility of a so called Pile agent. But more of that later.

The letters in the Top circles stand for some kind of identity. In order to relate "things" they need to be identifiable so each "thing" in Pile has an address, also called handle. More of that later.

Relations aka associations

Pile is not about data, but about relationships. It only knows relationships. Nothing more, nothing less. Pile calls the connection between two "things" relation. However, I prefer the term association, because whenever you talk about relations, people think you talk about relational databases. To avoid this misunderstanding I ll often use association instead of relation. But in the end they are interchangeable.

Pile s relations connect two "things" and are bidirectional.

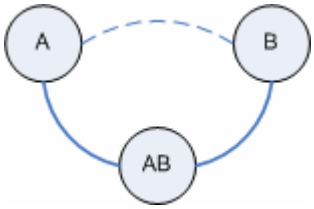


A relation thus can be described by a duple consisting of the left and right "thing": the association between A and B can be written as (A, B). And to give this association a name of its own you can call it AB :-)

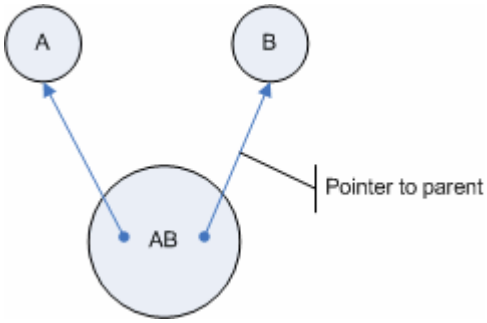
Now, what are those "things" that are connected by relations in Pile? Well, those "things" are... relations. Pile only knows relations and thus can only connect relations. That's part of the beauty of Pile.

A and B are relations as are X, Y, Z in the above picture. Tops are relations too. However, Tops are special in so far as they don't really connect anything within a Pile. You can think of a Top as an empty relation, e.g. X=(,).

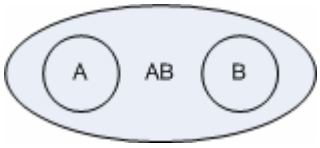
Now comes the twist: Each relation itself again is a relation! The above line between A and B thus is only a logical connection. In reality A and B are connected by a third "entity" called AB:



However, in this picture there are again solid lines between relations suggesting new relations. So maybe the most accurate way of depicting Pile relations is like this:

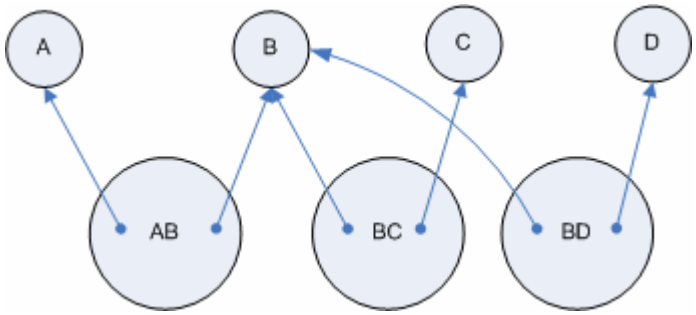


Each relation contains the knowledge of the relations it connects. You could think of this knowledge as pointers to the associated entities. Or you could view it as a containment relationship:



This also makes obvious another property of Pile relations: a relation always only has 2 parents, e.g. A and B for AB.

However, every relation can be parent to an arbitrary number of other relations:

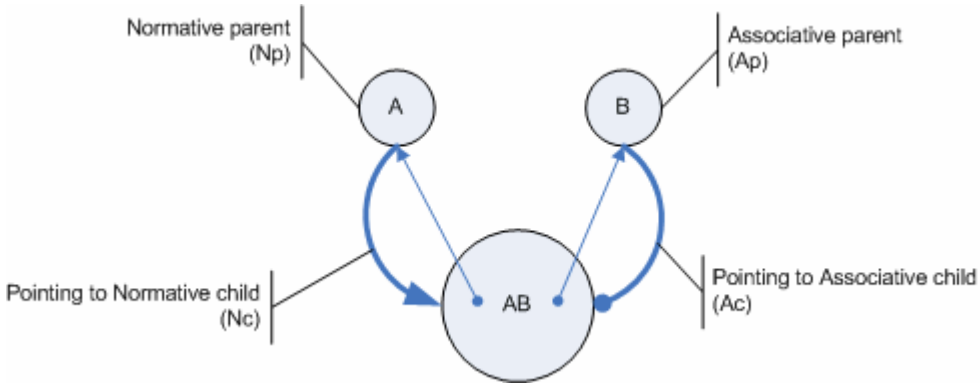


Here relation B is parent to AB, BC and BD.

Tops don t have parents. They are mapped to their Tv by some other means unknown to Pile. Pile just recognizes Tops as what they are thru the lack of parent pointers.

Defining hierarchies and linking hierarchies

Relations in Pile are not only bidirectional and themselves again parents for new relations, but also directed. Pile relations have a start and an end. The parents play different roles. Or how the Pile theory calls it: relations are related in different *manners*.



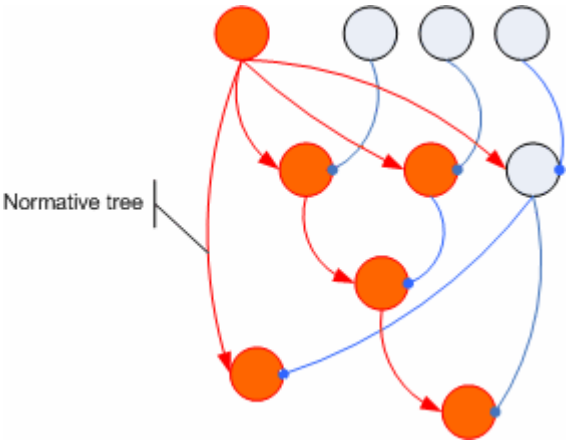
The parent-child relationship between two relations can be in so called *normative manner* or in *associative manner*. The *Normative parent* (Np) of a relation is the origin of it, the *Associative parent* (Ap) is the destination. So you could read a relation from left to right: A to B or A before B or whatever. (Erez even sees time encoded in Pile relations - but that s a part I don t understand yet. He probably alludes to a reading of relations like "A causing B".)

Although the "true pointers" are from a child relation to its two parent relations, the logical connections are from the parents to their children (thick lines in above picture). Each relation can be Np or Ap to an arbitrary number of relations who are then its *Normative child* (Nc) and *Associative child* (Ac) relations. And each relation has exactly one Np and one Ap (except for Tops).

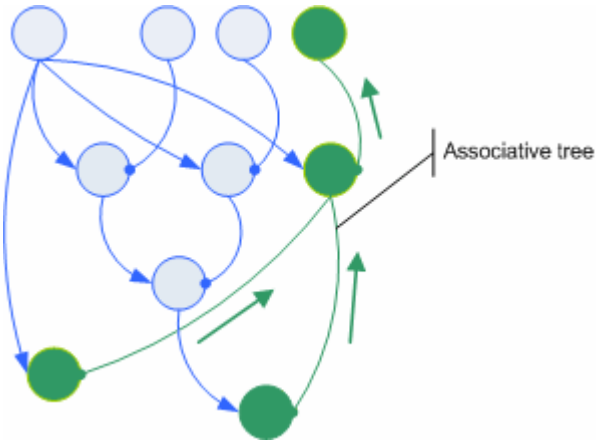
It took me quite some time to see why the Pile guys called the manners like this. But in the meantime I guess it's fair to explain it like this:

- Following the path from parent to child relations along the normative manner defines a hierarchy, a context.
- Following the path from child to parent along the associative manner links relations within a context with other contexts.

When you start at some relation and follow all paths from it to Nc and further down from them to the next level of Nc and so on, you're traversing a tree. This tree is called the *Normative tree* and can be thought of as holding together relations with regard to a common concern. We'll see in a minute, where such trees originate and how they can be used.



Now when you start at a relation and follow the route of it associative children and the Ac etc. then you get a different tree. It too starts at a single relation, but can also be read from bottom to top. When you follow the direction of the green arrows which means you follow the direction of the child relation which connects Np with Ap, then you'll find, the associative tree links relations within a normative tree with its top relation.



An associative tree can thus be viewed as a link between or bridge across contexts spanned by different normative trees.

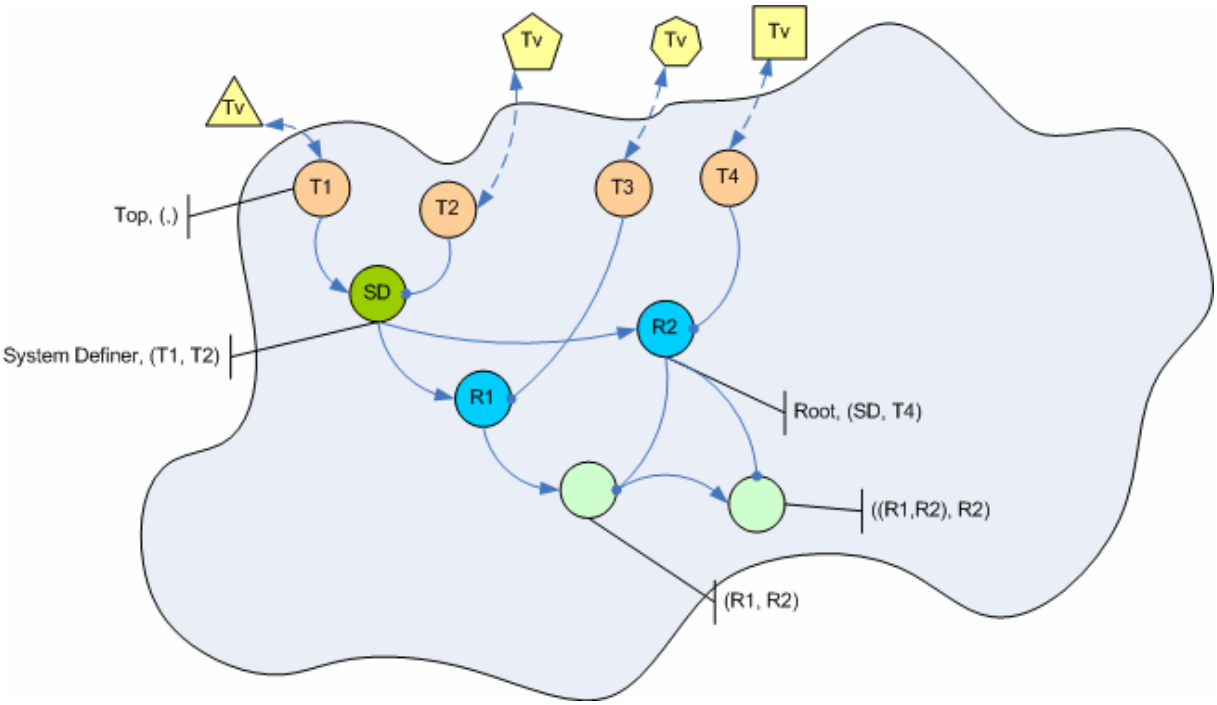
Does that make sense? No, I guess not :-) For you it's probably just terminology and theory without a relation (*sic!*) to the real world. But I hope you bear with me, because eventually I hope I can show you how all this can be put to work. Just suspend your disbelief for another moment and try to see it this way: In order to work with new concepts you need words to describe their parts, their structure. Otherwise no communication about the concepts is possible. For the world of RDBMS the terminology is well established. But for the world of associative bases or Pile it is not. That's why we need to climb up this learning curve first.

Types of relations

Ok, so what do we have? There are Tvs, there are Tops, there are (bidirectional but directed) relations. What can we make of them? We can relate those terms in a systematic way to see patterns. And patterns bring order to chaos, they help to avoid getting overwhelmed by details.

As it turns out, we can classify relations of being one of five types according to the type of their parent relations:

- Top: no parents, representation of a Tv
- (Top, Top): called System Definer (SD)
- (Np, Top): called (Normative) Root (Nr)
- (Top, Ap): called (Associative) Root (Ar)
- (Np, Ap)



The above picture shows the most important four types of relations. (Top, Ap) I haven t stumbled across in my work with Pile, so I left it out. I guess, you ll have no difficulties depicting it yourself.

The SD is somewhat artificial, but nevertheless important. Although its parents are roots, they usually don t have meaning in the outside world. You can pick some arbitrary Tvs for them. The reason for an SD to exist is not to represent a connection between outside data, but to define a normative tree! Each SD defines a context or (logical) system. It is the origin of a normative tree. Later on we ll see how this can be used to model a net of associations for full text search and other applications.

The Root relations then are representations of terminal values within a context. Roots "draw" Tops into the system of a SD. As far as I understand, when you use a Pile to describe data, your associations never are between Tops or SD and Tops, but always between Roots.

Most relations in a Pile are of the last type: (Np, Ap). None of the parents is a Top. You can think of them as "being somewhere in the middle" of the Pile trees.

Categories for relations

There is one last piece missing in the basic conceptual Pile puzzle: logical categories for relations. In order to be able to distinguish between relations and to "draw circles" around relations somehow belonging to

each other, you can assign each relation a so called *Qualifier* (Q). A relation thus is fully defined as a triple: (Np [Q] Ap).

Alternatively you can think of Q as the color or taste of a relation. It has no meaning to the Pile engine, but needs to be interpreted. A Q without an observer is of no use. A Q thus is assigned and interpreted by a Pile agent.

Uses for Qualifiers can be: categorize all relations describing lines in a text, color all relations according to whether the belong to information or meta-information.

Qualifiers are essential to bound searches within a Pile. They serve as demarcations for certain kinds of information, e.g. when searching for a text pattern you can stop traversal of the associations with Q "text" when hitting a relation of category "line".

When to use Qualifiers and when to use relations to Tops in order to assign some kind of type to a set of relations, I don t fully understand yet. But my guess is: This is an area where some research needs to be invested.

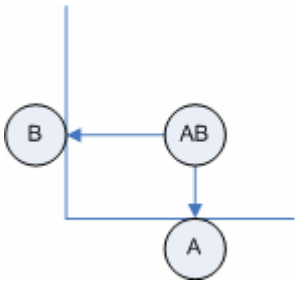
The Pile space - implementing relations

Ok, now we need to transition from theory to reality. How can implement a Pile? Well, you could say, let each relation be an object with two references to parents and many references to children, e.g.

```
class Relation
{
   Qualifier q;
   Relation Np, Ap;
   List<Relation> NcList, AcList;
}
```

Yeah, you could do that. But that wouldn t let your memory consumption explode. Erez thus suggest to step back and think about a more efficient mapping of relations.

What is a relation? Or let me call it a node or object in a Pile. A node is just a point in a two-dimensional space. It s coordinates are its parents.



The parent A and B define the point AB=(A,B) in a two-dimensional coordinate system. In so far it seems to suggest itself to represent a relation as a number on an axis in a Cartesian coordinate system.

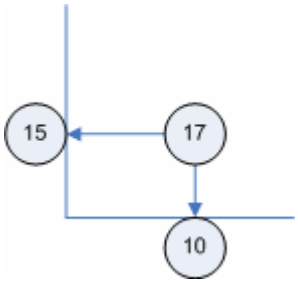
Let A=10, B=15, then AB=(10,15).

And indeed, a relation needs to be nothing more than a unique number - as long as this number somehow encodes the relations it associates with one another.

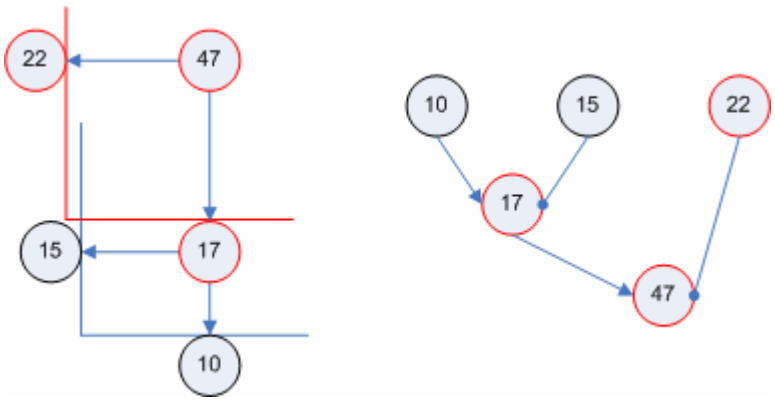
Now, how about the relation ABC=(AB, C)? With C=22 that would be ((10,15), 22). How to depict that in a 2D-space? Or what about D=25, E=34 and (AB, DE)=((10,15), (25,34))? Where would you locate (10,15) or (25,34) on the x- or y-axis? Not possible, you're right.

But here comes the rescue:

Assign the point of a relation in 2D-Space a value, a handle...



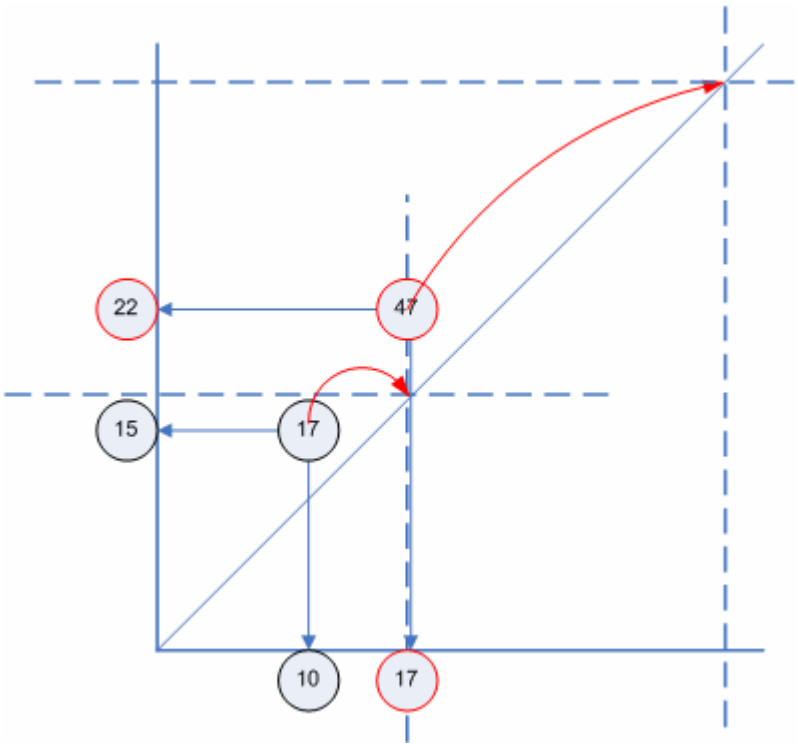
...and use that value as a coordinate, whenever the relation is used as a parent.



The above picture shows AB (with a handle value of 17) as the Np of another relation (with the handle value of 47).

Since a relation can be normative or associative parent of any number of child relations, it's handle can serve as x- or y-coordinate for those children. (The x-coordinate represents the normative manner, the y-coordinate represents the associative manner.)

Now, because the same handle value can be used on the x- and y-axis it effectively is a point on the 45° diagonal, where x- and y-coordinates are the same:

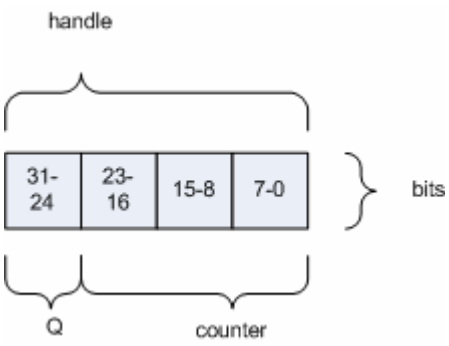


AB=(A,B) is 17=(10,15) and with C=22 leads to ABC=(17,22)=47. Then, when ABC is called to be a parent in another relation, its value 47 is either used as the x-coordinate or the y-coordinate, depending on whether it's supposed to be the normative or the associative parent.

Pretty ingenious, eh? ;-) By mapping 2 coordinates into 1 Pile avoids a nesting problem as in ((10,15),22) and let's you represent relations as simple numbers in a 2D Cartesian coordinate system. No object orientation needed.

But what about Qualifiers, you might ask. Good question! Qualifiers are encoded in the relation's handle. The handle value has n bits of which the

most significant m < n bits are reserved for Q. Current implementations of Pile engines use 32-bit integers for handles and allocate 8 bits for Q:



That means, there can be 256 different "colors" for relations and within each Q category some 16 million (2^24-1) relations. Sounds enough for a start, I'd say :-) But if you like, choose the number of bits per handle and for Q as needed.

Since the value of a handle is independent of the values of the handles of the parent relations, a Pile engine can "calculate" it any way it likes. Why then not say: for each Q the value of the remaining bits (24 in the above example) is viewed as a counter. For each new relation with a particular Q, the Q's handle counter is incremented, packed together with the Q value and returned as the handle for the relation.

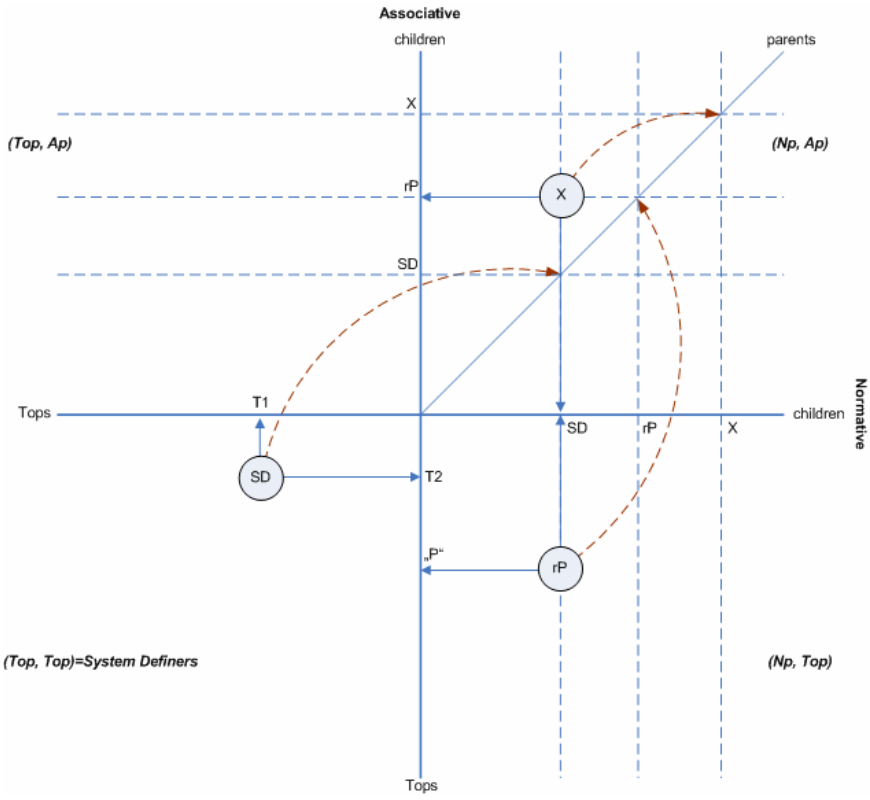
Here's how I do this in my C# Pile engine:

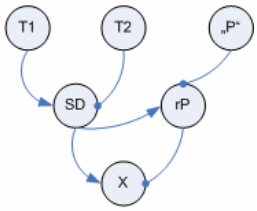
```
private int[] qualifierCounters = new int[256];

public int CreateHandle(byte qualifier)
{
   return ((int)qualifier) << 24 | ++qualifierCounters[qualifier];
}
```

With handles and a coordinate system in hand, we now can think about how to integrate all this into one homogeneous picture. How can we map the different types of relations into the 2D Pile space? Here's Erez doing it for us (with some attendees of the recent Pile workshop in Berlin):

It really took me a while before I understood what he was drawing... but in the end I think it's so simple and straightforward as to signify something important. His Pile space looks like this:





So far I've shown only the top right quadrant of the coordinate system. But in fact it's made up by all 4 quadrants, each for one of the non-Top types of relations.

The negative parts of the axises are reserved for Top relations. They don't have parents. So when you combine two Tops to form a System Definer relation you get a point in the lower left quadrant, e.g. SD=(T1, T2). T1 is the Np, so it's located on the x-axis, T2 is the Ap, so you find it on the y-axis. This quadrant usually only contains very few relations.

The point SD, as explained above, gets assigned it's own handle, SD, which is a positive value and defines a point on the parent diagonal in the upper right quadrant. This point necessarily lies on this diagonal, because it can be used as the x- or y-parent coordinate of another relation depending on whether SD is the Np or Ap.

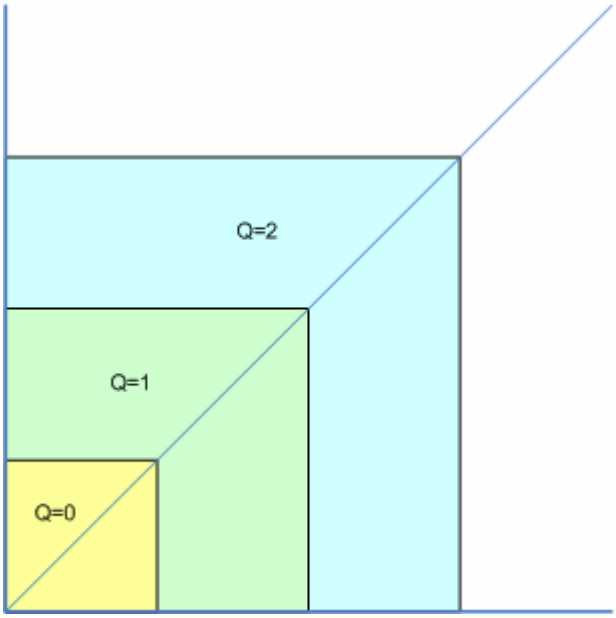
Take the relation between Top for Tv "P" and the SD for example. rP=(SD, "P") is a normative root since its Np is not a Top, but its Ap is. To locate the rP relation in the 2D space the value of its Np is used as the x-coordinate and the value of its Ap is taken as the y-coordinate. Then rP is assigned a value of its own which locates it on the parent diagonal.

Normative roots are located in the lower right quadrant. Ar are located in the upper right quadrant.

Now if you want to relate the SD with rP to and create relation X, you use SDs handle as the x-coordinate and the rP handle as the y-coordinate since X=(SD,rP). Then X is assigned a value of its own. This type of relation - (Np,Ap) - is the most common one. So the top right quadrant will be pretty densely populated.

And all the handles of all relations lie on the parent diagonal! Or to say it differently: All relations are located on a single line starting at (0,0) and ending at (2^32-1,2^32-1). Or even simpler: all relations are in the range of 0 to 2^32-1.

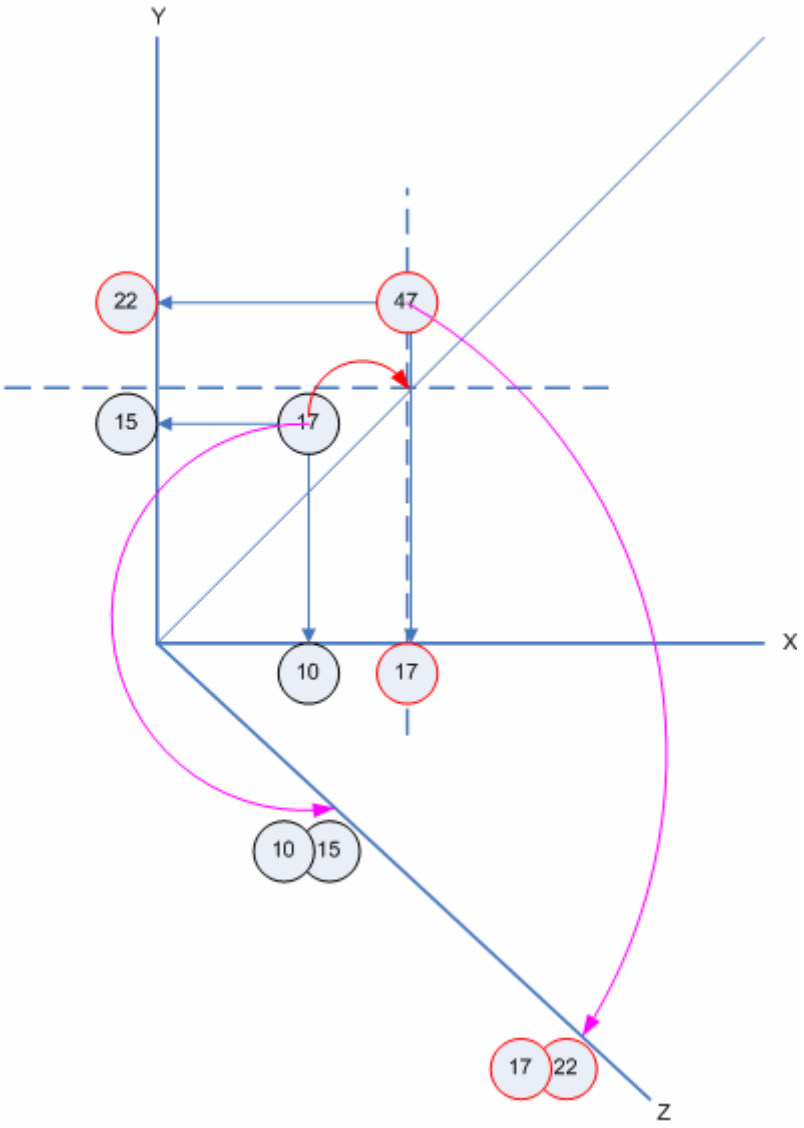
But the distribution of handles on this line is not equal. The line (or diagonal) is segmented by Q!



What we now have is a mapping of A and B to AB. We can move through a Pile from the Tops down... We can easily get from two parents to the child relation connecting them.

But when traversing networks of associations we surely also want to go in the other direction. So we need a way to locate the Np and Ap of a given relation.

This is done by introducing a z-axis into the picture. Each handle value is also located on the z-axis and its "point" there does not contain a handle, but it x-/y-coordinates: With r=(x,y) there is (x,y)=z(r).



So if you got a relation in your hands take the handle and go to the z-axis. Lookup the value for this child handle: it's a pair of handles, the x- and y-parent-coordinates of the child handle. Whereas a point in the 2D coordinate system holds 1 handle value, a point on the z-axis contains 2 handle values. Once I show you some code, you'll see it's very easy.

Pausing for a moment

That's pretty much it. More you don't need to implement you first Pile engine. Let's pause for a moment.

Sure, there are lots of open questions and many things to research. But the general concept of an AB, an associative base, or a Pile as Pile System calls it, are laid out. The world of associations and even their most basic representation has been defined. No more data to be seen. It's all outside the system. Inside are just relations and relations between relations.

Stop thinking in data, start thinking in associations or relations.

What's next? Next I'll show you my small Pile engine and talk about Pile agents. Stay tuned!

(Posted on Wednesday, December 21, 2005)

5. Data is Just an Abstraction of Associations - or: A Philosophical Interlude

In my previous posting I've described the Pile view of the "world of data": focus on associations, not data. Today I wanted to put meat on this theoretical skeleton and show you code I wrote. But first let me insert a little philosophical deviation. It might help to distinguish Pile from other (more or less novel) approaches to "data management" - and either draw more fire from the RDBMS camp or drive them off altogether :-) We'll see...

In the past days while trying to explain Pile to others I stumbled across two insights I'd like to share with you. Hope they help you as much as they helped me. Please give them a chance.

Elementary particles instead of bricks

We're all used to seeing the world through data centric glasses. First come tables and rows and objects and files. Then come relationships between those data entities. That kind of thinking is so ingrained in our brains (and education) it's hard to shed it. But do we need to? Well, since we can solve so many problems looking through this pair of glasses, it seems as if it's sufficient. Sure, to be content with this conceptual tool because it solves problems, is ok. That's the purpose of any world view and theory or model.

But ability to solve problems does not mean being the only possible interpretation of the world or any inherent "truth".

So once you hit a wall with an established world view (or theory or model) it seems prudent to question it. This questioning must be allowed at any time. Whoever denies this is dogmatic - that's how I understand scientific thinking. Asking questions all the time is at the very heart of science.

One of the questions I asked the established data models was: What's your granularity? And my impression is: the granularity when talking of objects and tables/rows is very, very coarse. Tables/rows and objects and files are bricks to build a data world with. And that's great! So many buildings can be constructed using bricks.

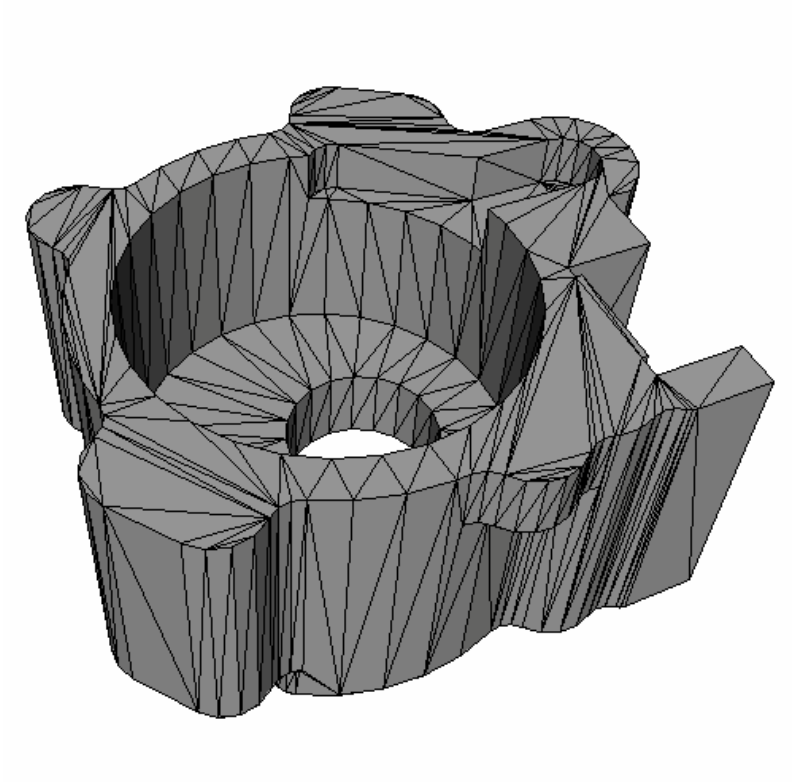
But let's be honest: In the end you can only do so much with bricks. There are inherent limits to what you can do with them. You can try to ignore those limitations, but finally you will succumb to them.

Now look at the world. Is it built from bricks? No, it's not bricks, but something much, much smaller. Elementary particles are the building block of our world. And even though there are not so many different

elementary particles (proton, neutron, electron to constitute atoms), the world is so, so, so rich in materials and forms. How can that be? Well, that's because elementary particles can be related to each other in an infinite number of ways. It's the associations between the few different elementary particles which makes possible the variety of the universe.

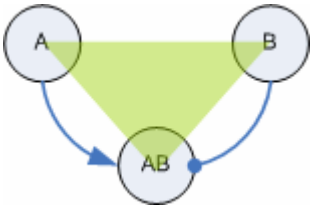
Now, when I say, objects and tables/rows are bricks, what are the elementary particles of data structures? It's associations or relations like in Pile. With objects or files you can build only so much. But when modeling (data) structures with just Pile relations, you can build everything. There's no limit to what you can model.

Or look at the field of 3D graphics: There is only so much you can model with spheres or toruses or cubes. But you can model any shape using triangles! They are the most basic building blocks for geometric structures/bodies, they are the elementary particles of 3D models:



(Source: http://www.computing-objects.com/en/meshtools_gallery_3dr.html)

Now look at Pile's relations. They are triangles too! A relation connects two "things" and itself again is a new "thing". I'd say, you cannot get more basic than that - and be equally simple and powerful.



You might say, "Hey, why do I need AB in this picture? Why not draw a straight line between A and B? Wouldn't that be even more simple?" You're right, that would look simpler, but it would introduce a data - relation dichotomy. "Data" (A and B) would be different from the relation (the connecting line) between them. In so far I'd say, in the end this would not be simpler, because there would be more concepts than in Pile which only knows relations.

My bottom line: Pile describes the most fundamental model for informational structures there is. So in essence it would be wrong to say "Hey, you can model a Pile using an RDBMS!". Rather you should think "I can model an RDBMS or a file system using Pile relations!" That's a fundamental shift in thinking!

As important this focus switch is, of course it is important to have concepts and structures on a higher level of abstraction. You want to have bricks to build houses and not arrange elementary particles. So to have RDBMS and files and objects is a good thing. But it's also important to know they are just convenient abstractions. It's like looking at a fridge and using a fridge - but knowing in the end it's just made up of elementary particles.

Sometimes it's good to see things as high level entities - and sometimes it's good to see them as aggregations of low level entities. The ability to switch perspective is valuable, I'd say.

Data is an abstraction

Once I had this picture of associative triangles in my mind, I then thought about the nature of data. What is data anyway, if it's so important to us? If you look at Wikipedia´s definition you read: "data are numbers, word, images etc." or "statement[s] accepted at face value". And I'd add: data are units of meaning as a statement is supposed to mean something. (Let's leave context dependency out of the picture for the moment.) A number is a "unit of meaning", a letter is, an image file too.

Now, what is data made of? Just 1 and 0. Numbers, letters, files are just sequences of bits. You determine where a unit of meaning starts and ends. You determine the interpretation of a sequence of bits.

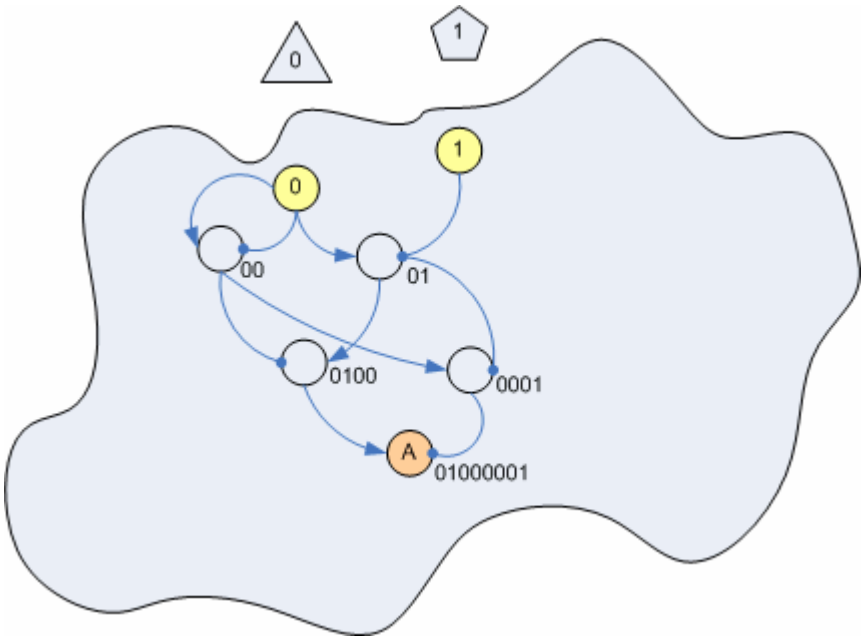
1/0, On/Off themselves are no data. They only become data when you interpret them. By themselves they are only "signals" (or a signal and a

missing signal). So a number or a letter is a sequence of signals. Data emergence from this sequence only if you assign it meaning, e.g. taking 8 signals and saying "This is a byte and it means 'A'."

Datum "A" thus is this sequence of signals: 01000001 (hex 41). And interpreting these signals as "A" is moving to a higher level of abstraction. You give up detail to get a more manageable entity, e.g. a letter instead of a sequence of signals.

So I'd say: There is no data. Data is just a necessary and useful abstraction. What there is are only signals and associations of signals.

With regard to Pile that means: There need not even be any data outside a Pile. The very "atoms" of a Pile are the only two signal values 1 and 0. From them anything you want can be built.



Only 1 and 0 are left outside the (world of) Pile. And since they are so basic, you could even get rid of them and make them axiomatic concepts of Pile as the only really needed Terminal values.

Can you see now, why I think, Pile is so fundamental? Data is just an abstraction and Pile's relations are enough to tie together the basic signals of 1 and 0 to form larger units which then can be interpreted as data in any way you like.

Ok, now, enough with philosophy! Next time I'm gonna show you down to earth C# code. How's that?

(Posted on Friday, December 23, 2005)

6. Folding the Informational Space - or: How Pile Lets You Transcend Hierarchies of Data

After some philosophical digressions now on to "More matter, less art" as Hamlet's mother says. Let's step up the ladder of abstraction and look at how to put the elementary particle, those associations, to some tangible use. How about implementing a little full text search application? That's what I did to check my understanding of Pile. Although there exists such an application at pileworks.org, I thought it would help me more to try to build it myself instead of just studying existing code.

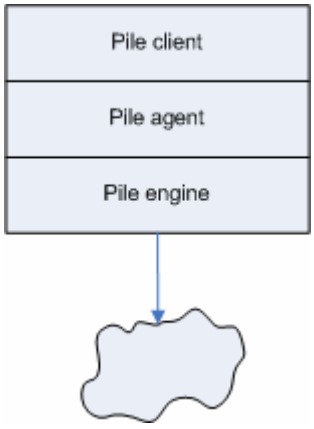
General software architecture

How should a solution based on Pile look like in terms of architecture? There are some guidelines that sound reasonable:

There should be a *Pile engine* which manages the relations, but does not know anything about what they are relating with regard to the real world. The Pile engine can fetch handles, create new relations, or retrieve children of relations. It's like a generic database engine which only knows of tables and rows and constraints, but nothing about invoices or customers.

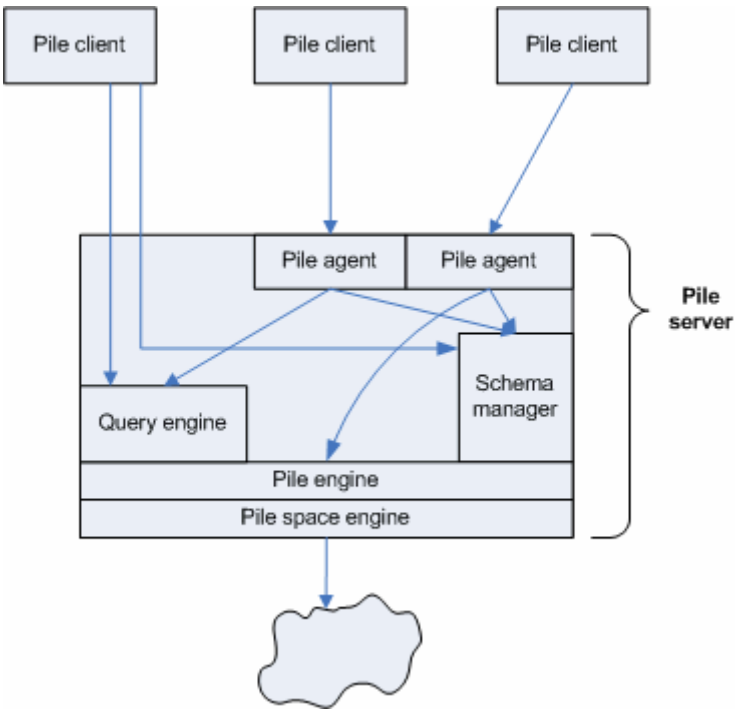
Then on top of a Pile engine sits a *Pile agent* who uses the engine to manage domain specific information. A Pile agent so to speak models the Pile clay. It is the observer who assigns meaning to the relations managed by the engine. A Pile agent defines Terminal values and uses Qualifiers for his purposes.

And then there is the *Pile client* who uses a Pile agent to accomplish some task.



This layered architecture looks pretty reasonable, I'd say. But you might ask, where the difference is between a Pile agent and a Pile client. The reason for the distinction and for calling the Pile agent not just "Pile access layer" but "agent" is its complexity. A Pile is so fundamental, its building blocks - the associations - so tiny and fine grained, so that it needs more than an "access layer" to collect information from a Pile. The agent at least currently defines not only the schema for a Pile but also algorithms for traversing it. There is no Pile query language to help - yet.

In the future, though, this probably will change. I envision Pile to move to an architecture like this:



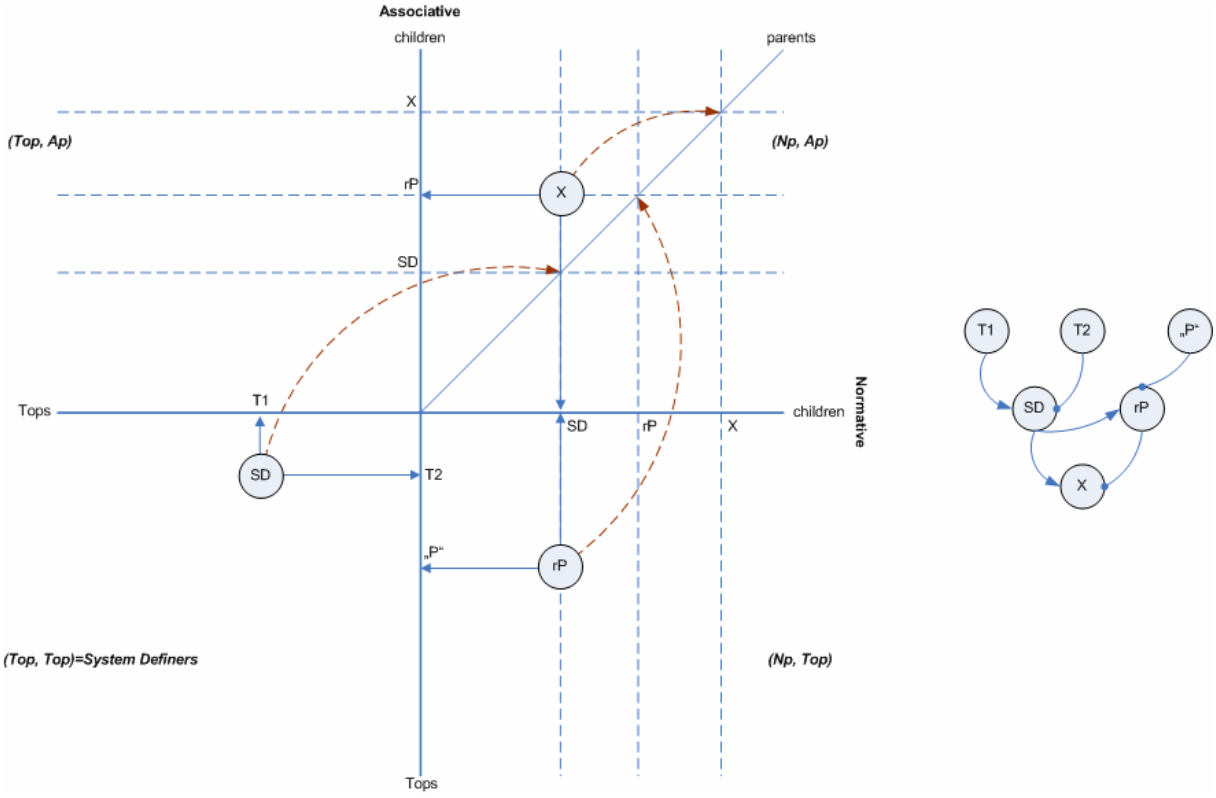
A *Pile server* will provide help for querying a Pile base as well as structuring it. Associations in the end are too fine grained as to always think on that level. Arbitrary levels of abstractions need to be definable (see the below section on Pile structures and wormholes). Constraints need to be set up. In so far Pile of course needs to look like an ordinary database.

A Pile client then can use the query engine and a schema manager to access associations. Query engine and schema manager are generalized Pile agents. Since traversing the mesh of relations in a Pile requires many small steps through the Pile space, all operations need to run close to the Pile, i.e. in a server process (or embedded in a client process). Only results should be send back to clients.

In addition sometimes it might be necessary to work with a Pile in very problem domain specific ways and traverse it directly. That's when custom Pile agents are still needed - but they too should run close to the engine in the same process like stored procs in relational databases.

Please note I split the former Pile engine in two: a low level *Pile space engine* and a high level *Pile engine*.

The Pile space engine for me is the entity to manage handles and the 2D space and create child handles from x-/y-parent-handles. The Pile space engine does not really know about relations and manners and normative/associative parents. It's just the manifestation of the 2D coordinate system I described at the end of [my earlier posting]:

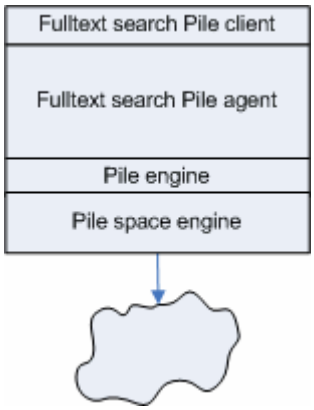


When switching from in-memory operation to a persistent Pile implementation (hopefully) the Pile space engine is the only part that needs to change. At least that's how it worked out when I implemented my Pile application and switched the Pile space engine to work with a flat file database API (from VistaDb).

On top of the Pile space engine sits the Pile engine. It knows about relations and manners and all and adds some convenience operations to the lower level API (see below for details).

But of course, the above architecture of a Pile server is just a sketch. I m no expert in implementing such kind of infrastructure software. But I imagine the depicted components to at least present in such a system. And I m sure Pile needs to be implemented as a true server, if multiple users need to be supported. In case just a single user (single thread) wants to work with a Pile - like in the following prototype -, the whole agent-engine-space stack of layers can run in the client process as embedded components.

Here s a picture of the architecture of my Pile prototype:



Pile space engine and Pile engine are pretty simple (both some 130 lines of code only). Most code is in the Pile agent and its methods for searching strings. The client on top again is a thin layer.

But before delving into the code of the Pile (space) engine, let me say a couple of words on schemas for associative bases.

Data modeling with elementary particles

When you give associations a chance to take center stage, than probably the first question you ll ask yourself is "What s the Terminal values for my Pile?" At least, that s what happens to me all the time ;-) We re all so data focused that we first ask for the data. But that s ok, I guess.

Now, is there a simple answer to that question? I d say yes and it is: It depends :-) What Tvs to choose depends on how much potential for connectivity you want to have in your Pile.

The magic of Pile is to be able to connect everything with everything, since there is only one "thing" to connect to and to connect with: relations. It s not data you connect, it s relations.

That means: whatever is within (!) your data, hidden in your Terminal values, cannot be connected. Terminal values are black boxes, blobs, opaque entities. You can relate them to each other, but that's it.

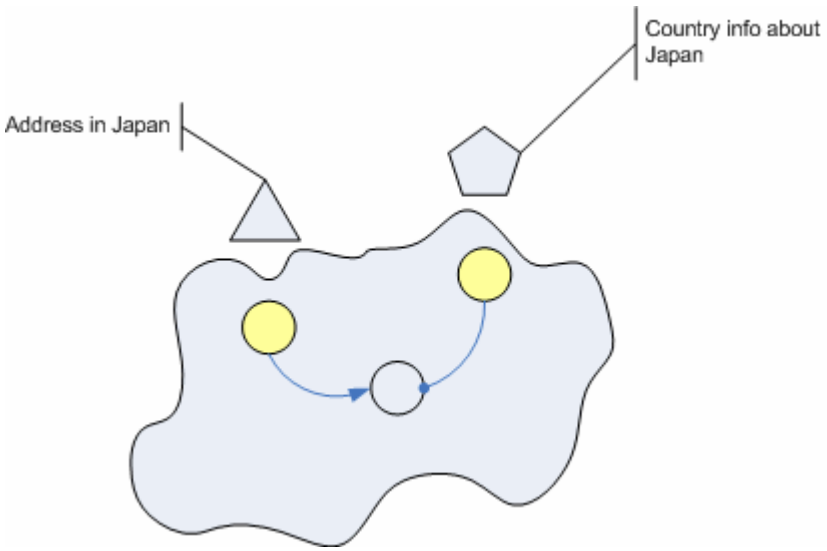
If you like, you can choose to make a whole address (with street address, city, zip, country) a Terminal value. That's perfectly fine for Pile. (In fact, Pile does not distinguish between an address, a 10 MB image file, or a single character as Terminal values.) But mind you: If you choose an address as a Terminal value you loose any chance of "sharing data" between several addresses (e.g. you need to store the same city many times in different address data blobs). And you loose any chance of automatically and implicitly connecting, say, addresses and statistical data on the worlds countries.

Ok, what do I mean by that? Let's take a simple relational database schema containing addresses and country info:

Addresses(street_address, city, zip, country_name)
Countries(country_name, number_of_inhabitants)

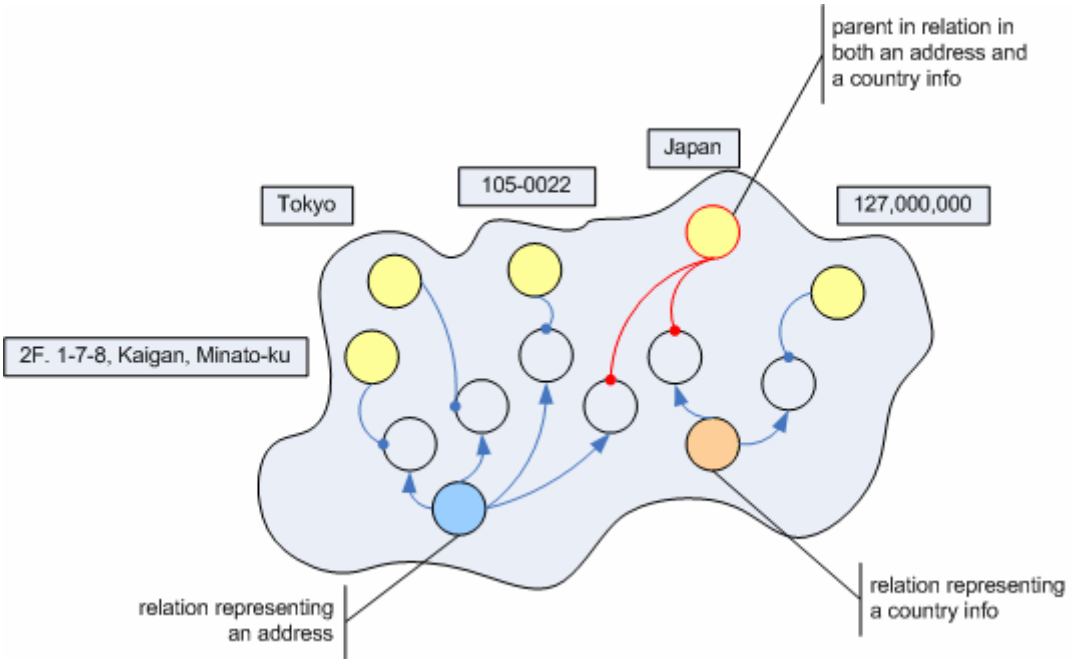
To relate Addresses and Countries you need to set up an explicit link between the two tables. You'd need to make country_name the primary key of table Countries and Addresses.country_name would become a foreign key. (I assume country names to be unique.) Country names would now be stored in each Address row and each Countries row (and also in any indexes needed).

How would this scenario look in Pile if you chose to make addresses and countries Terminal values?



In order to relate the address to the country info, you'd need to set up an explicit relation. That's like working with PK-FK-pairs in relational databases. Also the data (country name) would get stored at least twice: once in the address Tv, once in the country info Tv.

However, if you choose more fine grained Terminal values, Pile gets a chance to blossom. For example, you could define each attribute of an address or country info to be a Terminal value:



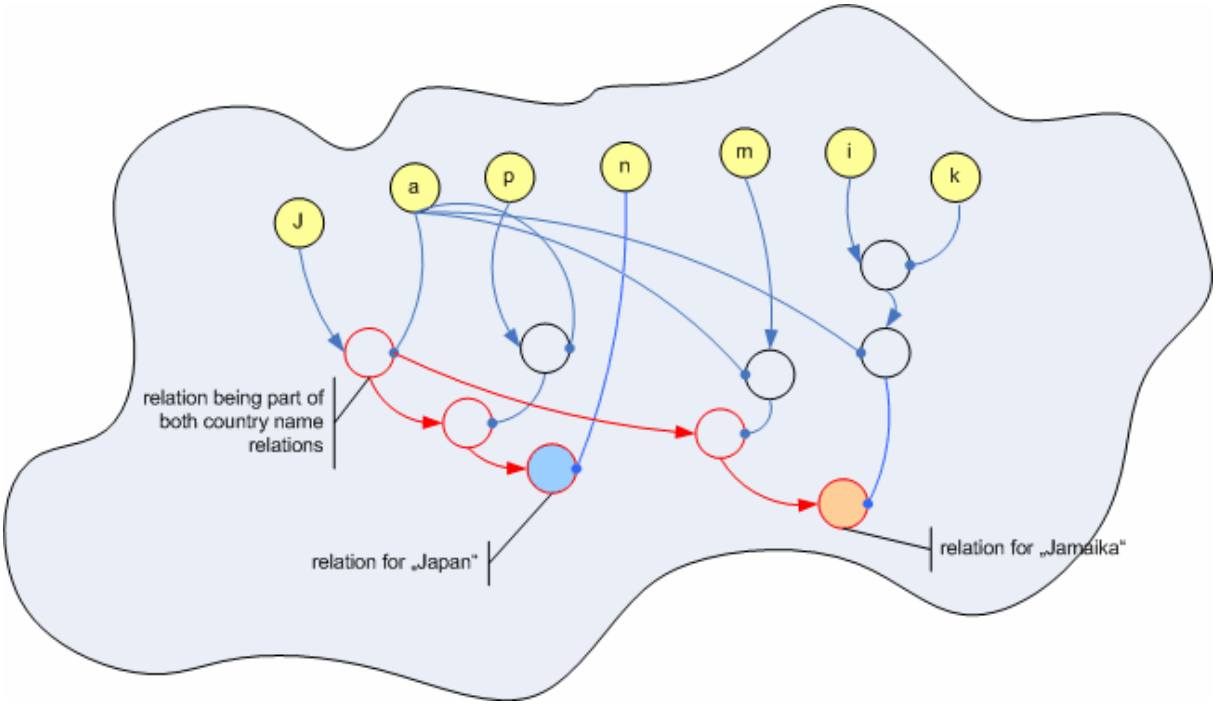
Each value for a street address, city, zip code, country name, and number of inhabitants would need to be stored outside the Pile only once! And within the Pile the Top for a country name would automatically connect addresses with the info on their countries, because the country name is part of both.

This sounds a bit like in relational databases, but please mind the differences: 1. Each value for an attribute would need to be stored only once. 2. No indices are needed to speed up linking of addresses and country infos, since if you have the handle to a country name, you immediately have the handles into all the addresses and country info "entities" (via the associative tree in Pile, see the red lines above). 3. There is no limit as to how fine grained you choose your Terminal values. 4. There is no fixed physical boundary around "groups" of Terminal values or groups of such groups etc. like rows or tables which would limit your ability to connect.

I'm sure you agree with above claims 1. and 2.

But what about 3.? Choosing city, zip code, or country name as Terminal values is just one possible choice. It would have been equally well possible to choose singe letters as Terminal values or - as I explained yesterday - just the bits 1 and 0. The more fine grained you choose your Tvs, the smaller their number, and the higher the potential for connections between parts of informational units on a higher level of abstraction.

As an example choose single characters as Tvs, so there are just 256 Tvs which should be enough for any database you want to set up using a Pile. (Think like in the XML world: any data item can be expressed as plain text.) With those 256 Tvs you can set up relations for the above city, zip, and country name which now exist only within the Pile and not outside anymore. However, whereas before the country names "Japan" and "Jamaika" would have been stored as two distinct external Tvs, now they are represented by two distinct internal relations sharing (!) a parent relation for the letter combination "Ja".



When searching for the letter combination "Ja" you hit the relation they are connected by. You then can check this relation, which entities on a higher level of abstraction (e.g. country name or city) they are part of. Those higher level entities are thus automatically connected without you even explicitly defining this connection. It's simply there, because you chose very fine grained Terminal values.

It's up to you to use such kind of implicit connections of formerly disconnected entities.

Which brings me to 4. What about larger structures than Terminal values? Well, you choose them too as you see fit for your scenario. There is no limit where you can draw boundaries around a set of relations and define them to be some kind of distinguishable entity. You just need some way to tell, where the boundaries of such entities run. But that's a matter of an observer (a Pile agent). Or to say it differently: Where a relational database defines just three levels of containers for data (field, row, table), a Pile can container any number of nested and coexisting containers with an arbitrary structure.

Using formal languages to shape a Pile

A schema definition for Pile thus looks more like the definition of a formal language, I'd say. You first define your Terminal values. Again: you can choose any granularity you want. You can even mix different categories of data, e.g. letters and image files to represent DTP documents. But let me stick with letters for now:

Terminal values:
A ::= "A" .
B ::= "B" .
C ::= "C" .
...
0 ::= "0" .
1 ::= "1" .
...
letter ::= A | B | C | ... .
string ::= { Letter } .
digit ::= 0 | 1 | 2 | ... .
integer ::= Digit { Digit } .

(The Terminal values can be compared with the lexical level of a formal language.)

Next you define *Value Relations* which are made up just from Tvs only, e.g.

Value Relations:
streetName ::= string .
cityName ::= string .
zipCode ::= integer .
countryName ::= string .
numberOfInhabitants ::= integer .

No you step up to the next level of abstraction where you compose higher level relations which could be called *Composite Relations*. For a start they are made up of Value Relations:

Address ::= streetName cityName zipCode countryName .
CountryInfo ::= countryName numberOfInhabitants .

But Composite Relations can also contain other Composite Relations, for example:

Contact ::= contactName Address .
Manufacturer ::= Contact .
Product ::= productName qtyInStock price Manufacturer .
LineItem ::= Product qtyOrdered .
Invoice ::= Contact { LineItem } .

(Value Relations and Composite Relations can be compared to the syntax level of a formal language definition.)

You see what I mean? You can arbitrarily nest Composite Relations. On each level you just define which other relations the Composite Relation is made up of. To store data for your invoicing software, you'd thus define the schema for a Pile by coming up with your own formal language, with a root production like this:

InvoicingSystemSchema ::= { Contact } { Product } { Invoice } { CountryInfo }.

There are no predefined containers, no limits to the number of levels of nested containers or levels of abstraction. And the beauty of it is, each and every container is automatically linked to other containers sharing the same Value Relations or even parts of Value Relations (e.g. letter combinations).

If this reminds you of XML, you're right. An XML document can be viewed as a sentence in a language defined by an XML Schema. The above example could be written like:

```
<InvoicingSystem>
   <Contact>
      <contactName>Acme Inc.</contactName>
      <Address>
         <streetName>...</streetName>
         ...
      </Address>
   </Contact>
   ...
   <Product>
      <productName>...</productName>
      ...
      <Manufacturer>
         <Contact>
```

```
            <contactName>Acme Inc.</contactName>
            ...
         </Contact>
      </Manufacturer>
   </Product>
   ...
</InvoicingSystem>
```

This sure would work, but look at the <Contact>-elements under the root and within the <Manufacturer>-element. If you wanted to avoid storing contacts twice, you needed to set up explicit references like this:

```
<InvoicingSystem>
   <Contact ID="1234">
      <contactName">Acme Inc.</contactName>
      <Address>
         <streetName>...</streetName>
         ...
      </Address>
   </Contact>
   ...
   <Product>
      <productName>...</productName>
      ...
      <Manufacturer>
         <ContactRef ID="1234" />
      </Manufacturer>
   </Product>
   ...
</InvoicingSystem>
```

You might find this quite natural - but in fact it's cumbersome, since you need to think about such dependencies explicitly.

The folded Space of Pile

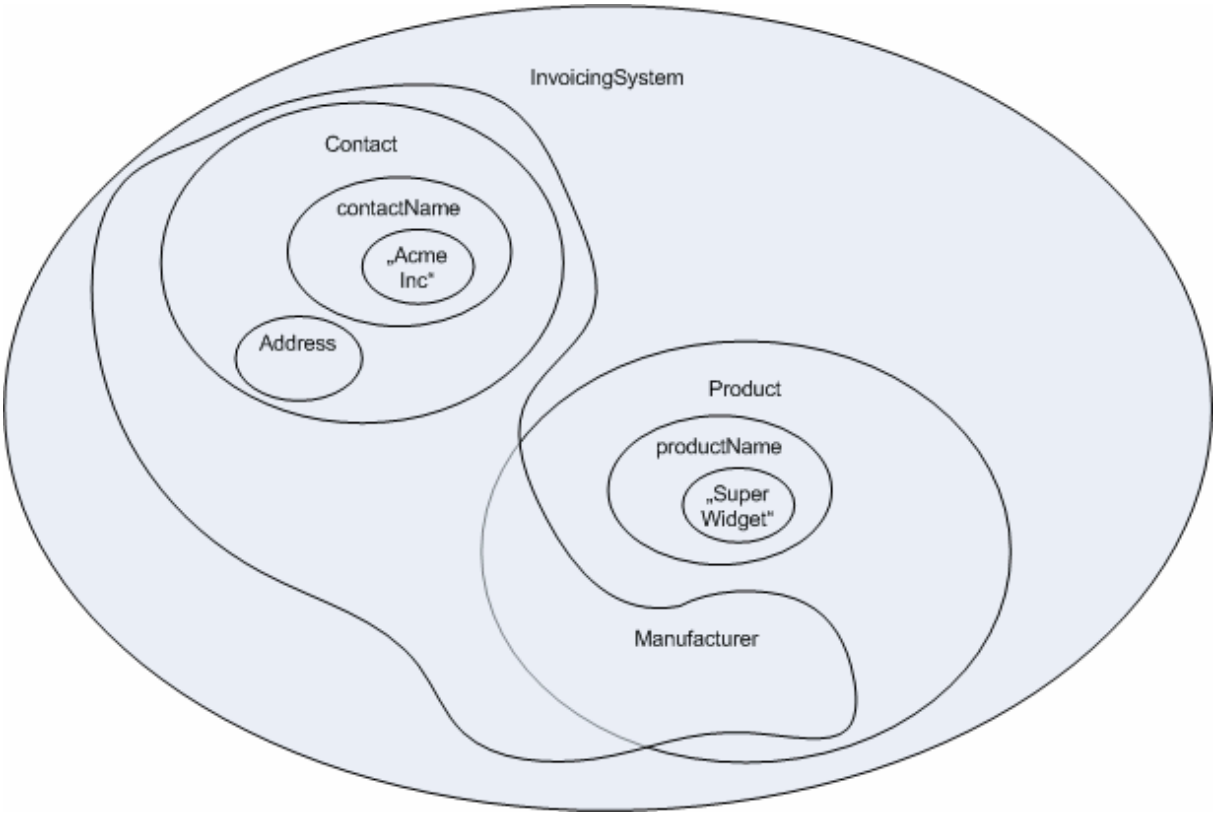
In Pile, on the other hand, such redundancies *do not even exist*, so you don't have to circumvent them. Pile always stores the same information only once. Or to say it more generally: Any relation or relation of relations or relation of relations or relations and so on can only exist once in a Pile.

To store the string "Wash", e.g. as part of a contactName (e.g. "Washington Redskins") in a Pile you set up the relations Wa=("W", "a"), sh=("s", "h"), Wash=(Wa, sh). And to store an address you set up the following relations:

1234=("834 Dupont Circle NW", "**Wash**ington, DC")

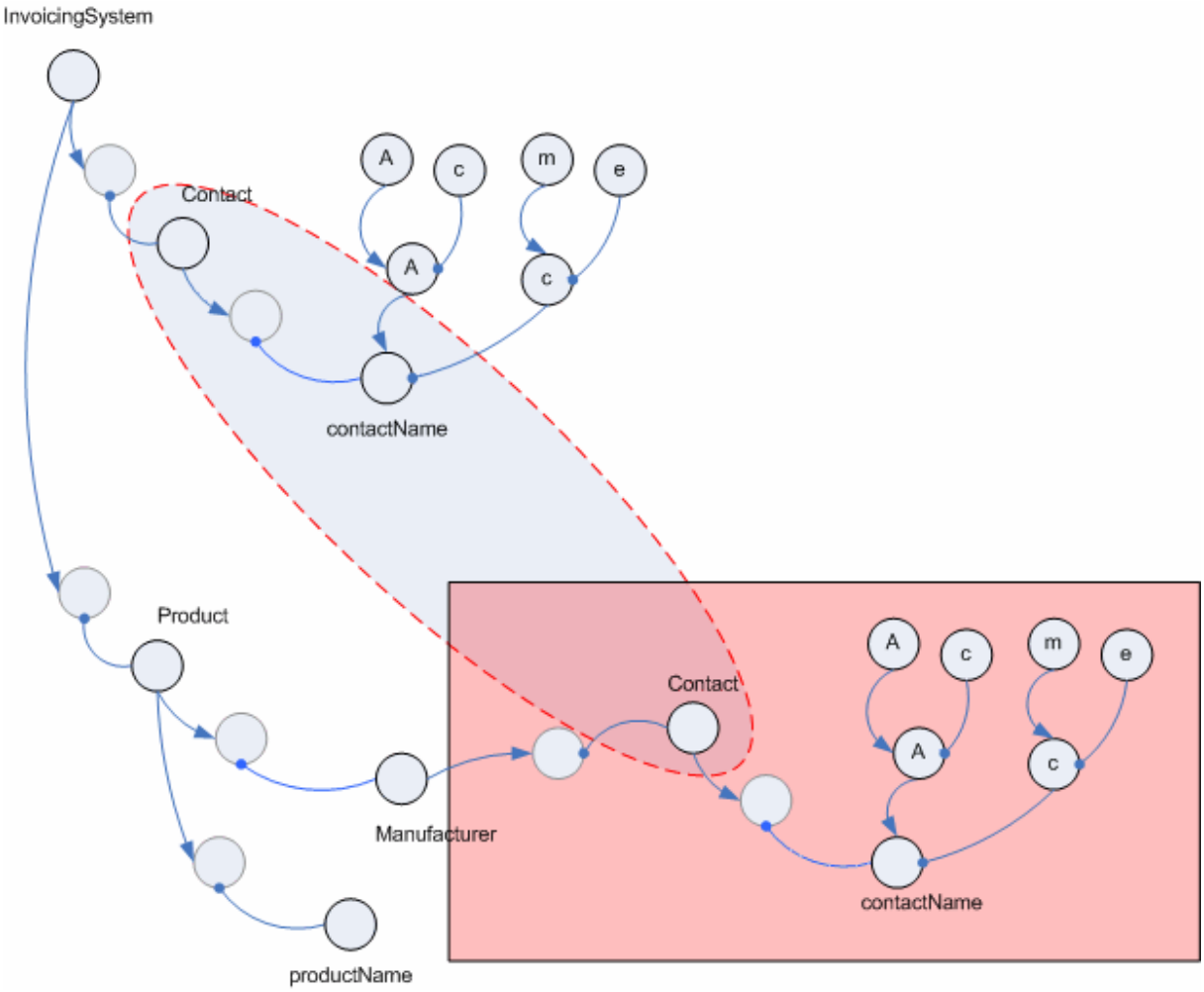
8234=("20099", "USA")
7729=(1234, 8234)

The Wash-relation from the contactName will be reused in the city of the address, since it too starts with "Wash" and the whole address relation 7729 will be reused whenever another contact is created for the same location. This might be a tad difficult to visualize, but you should try. It's worthwhile. Here's a sketch of how the XML sample could look like in Pile:



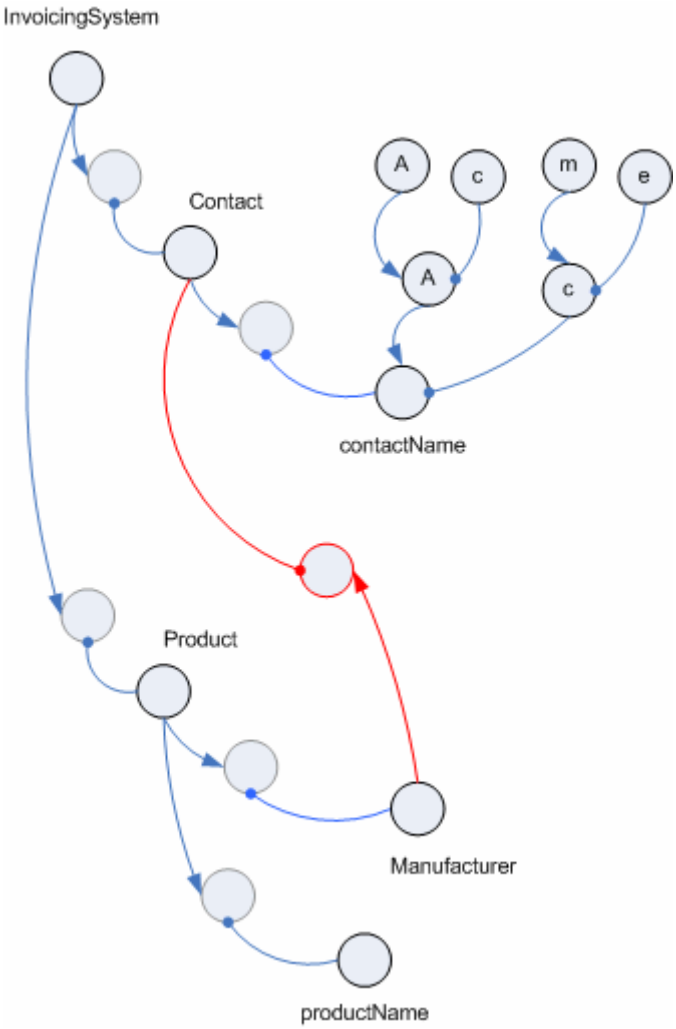
The same (!) Contact exists on several levels of the hierarchy: once below the top Composite Relation and once within the Manufacturer Composite Relation. And, really, it's the same Contact. It's not a copy and there is no special reference, it's just the same. And you cannot avoid it.

On any level of the abstraction hierarchy in a Pile be it relations between Terminal values or relations between Value Relations or Composite Relations each combination of two parents can and will only exist once. This is so unlike any other data storage model, I can only describe it using an analogy from physics or chemistry:



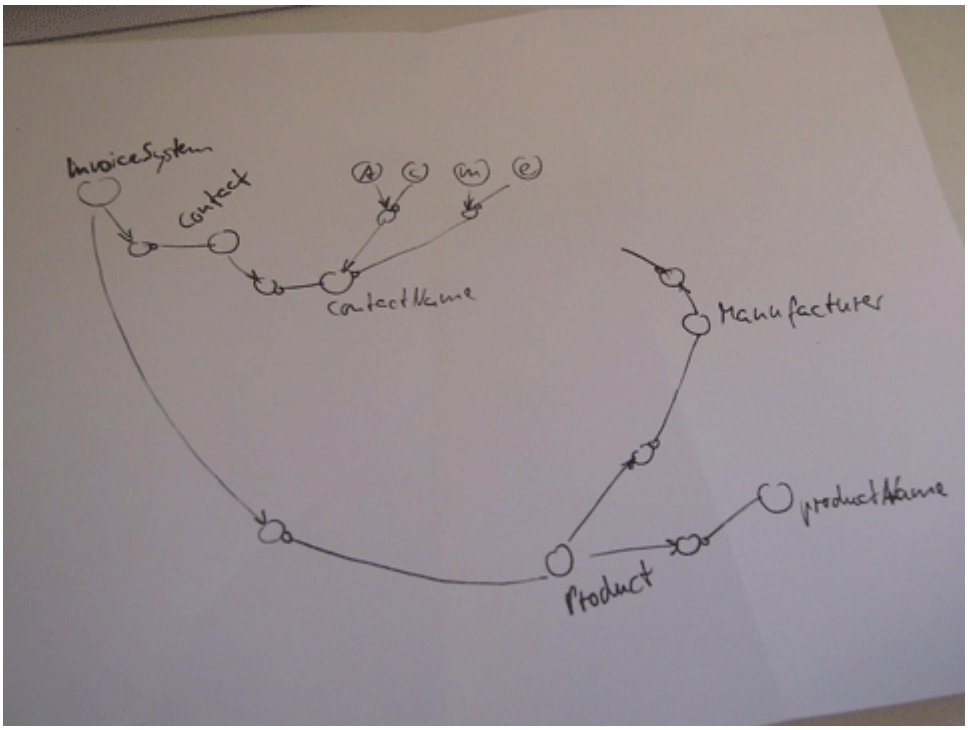
In Pile the informational space is folded like protein molecules are folded to reach a lower energy state or like the space-time continuum being folded in the presence of black holes. Yeah, maybe that's something you can more easily picture: think of Pile as a space where every single relation can be a "worm hole" between different parts of a Pile :-) In Pile everything is unique.

The Contact that's connected to the Manufacturer in the above picture in fact does not exist separately, since it's the same as the Contact hanging directly beneath the InvoicingSystem relation. So the above picture is wrong, in reality the Pile looks like this:



The Manufacturer relation re-uses the existing Contact. And this re-use, the connection between remote parts of a Pile can be thought of as "folding the Pile space" as the following animation tries to depict (I hope you can read my handwriting :-) but it's the same Pile as above):

(Editor's note: since the animation cannot be reproduced in this print document, please see the original blog at http://weblogs.asp.net/ralfw/archive/2005/12/26/434002.aspx)

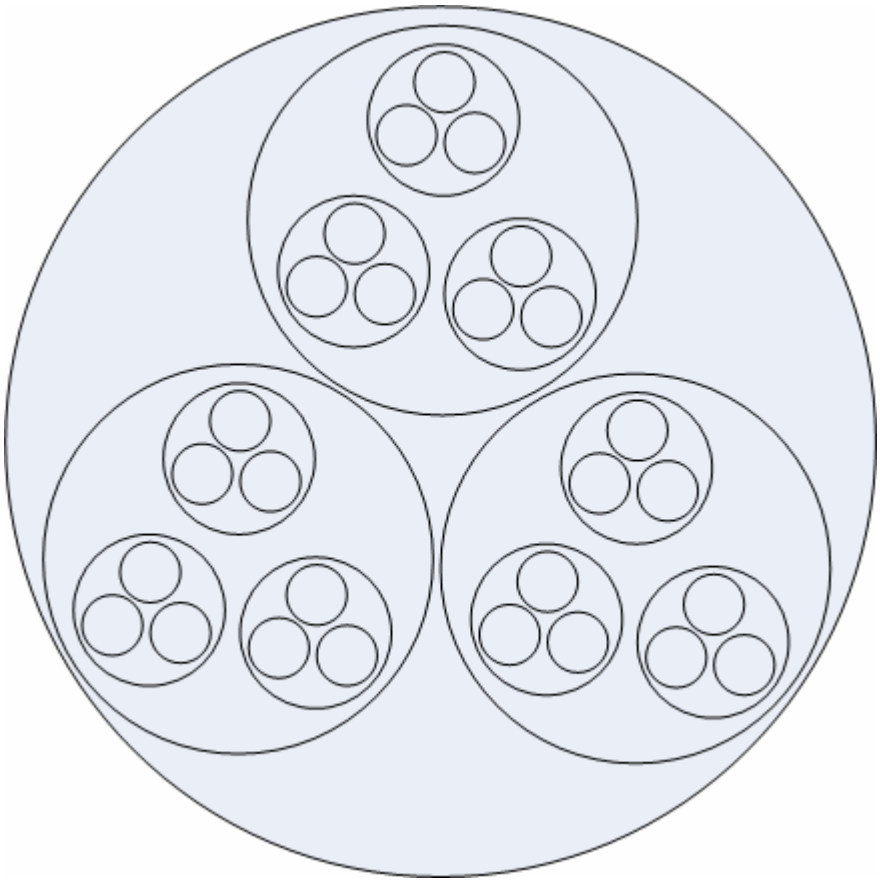


The Manufacturer so to speak is forced to connect to the existing Contact to "minimize the overall energy level" ;-) of the Pile.

Transcending hierarchical systems

A formal language defines a "universe" of valid sentences. Each sentences can be described as a tree. Within the sentence there are an arbitrary number of levels of abstraction, e.g. Terminal values, Value Relations, Composite Relations, composites of composites etc. Each "unit" then is at the same time part and whole: a streetName is a whole with regard to the Terminal values and is part of an Address. An Address is a whole with regard to streetName and city, and part of a Contact.

The property of being whole and part at the same time was termed *Holon* (assembled from *holos* (Greek for "whole") and *on* as the suffix for "part of" as in prot*on*) by Arthur Koestler in the 1960s in his book "The Ghost in the Machine" and can be depicted like this:

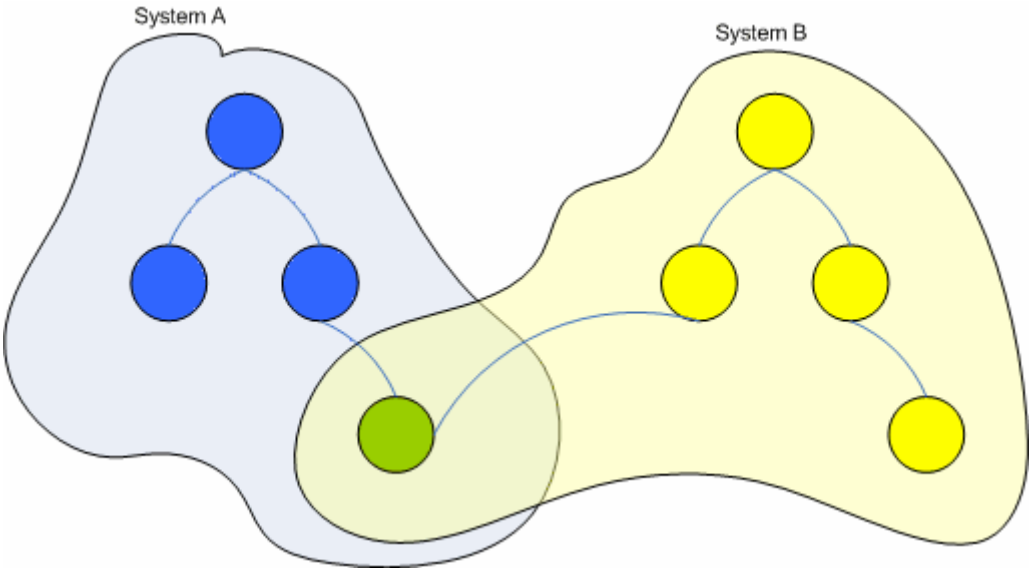


Each circle stands for a holon. Each holon contains smaller holons and at the same time lives within a container holon. Such a system of holons is called a *holarchy*, a hierarchy of holons.

A Pile can container any number of holarchies. Now, the interesting thing is, each holon can be part of many container holons as the first picture in section "The folded Space of Pile" shows. But not only that: each holon can also be part of many different holarchies!

Since Pile does not define "physical containers" like XML-elements or relational tables/rows, all holon boundaries are just logical. And since each holon as a relation can only exist once in a Pile, holons in completely different holarchies of a Pile containing the same "looking" holon in fact contain the same holon. The only prerequisite for this is, different holarchies share some notions. This might be the case on the level of Terminal values (e.g. holarchies A and B using single characters as Tvs) or this might be on a higher level of abstraction (e.g. holarchies A and B both using the Value Relation streetName).

Pile thus allows you to focus on one informational hierarchy like any other data model. You then traverse a holarchy only according to it formal language. But at the same time Pile gives you the chance to transcend a hierarchy and switch focus to any other which it shares relations with.



If a logic defines a context or system, that means a boundary between what belongs to the system and what is external, then Pile is truly poly-logical - meaning, many systems can be defined by different logics - but as long as those systems share some more or less basic notions, entities are not confined to one system, but can simultaneously exist in many systems or context.

That brings me back to how humans view the world. There are not fridges and cars and pens in our heads. There are just associations. And those associations can exist in many different contexts as I described in a previous posting. Since Pile by its nature can is multi-contextual, it seems to be a quite natural model to express the real world in a machine.

You can the smallest possible information elementary particles you like, all associations are unique, you choose the levels of abstractions, all containers are logical and of "same right", connections between different parts of a Pile are automatic, even connections between different "logical systems" are automatic as long as they share some concepts.

To me that sounds like a beautifully simple model to build information systems on.

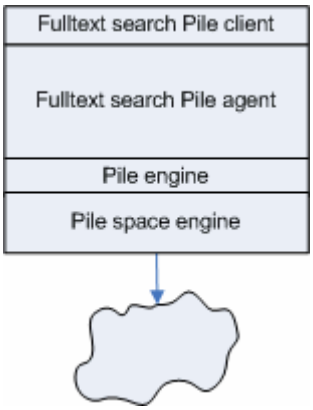
Sorry, I somewhat got carried away. This has become a long posting and I need to pause for a moment. But I promise, next time I ll show you my Pile engine. Stay tuned!

(Posted on Monday, December 26, 2005)

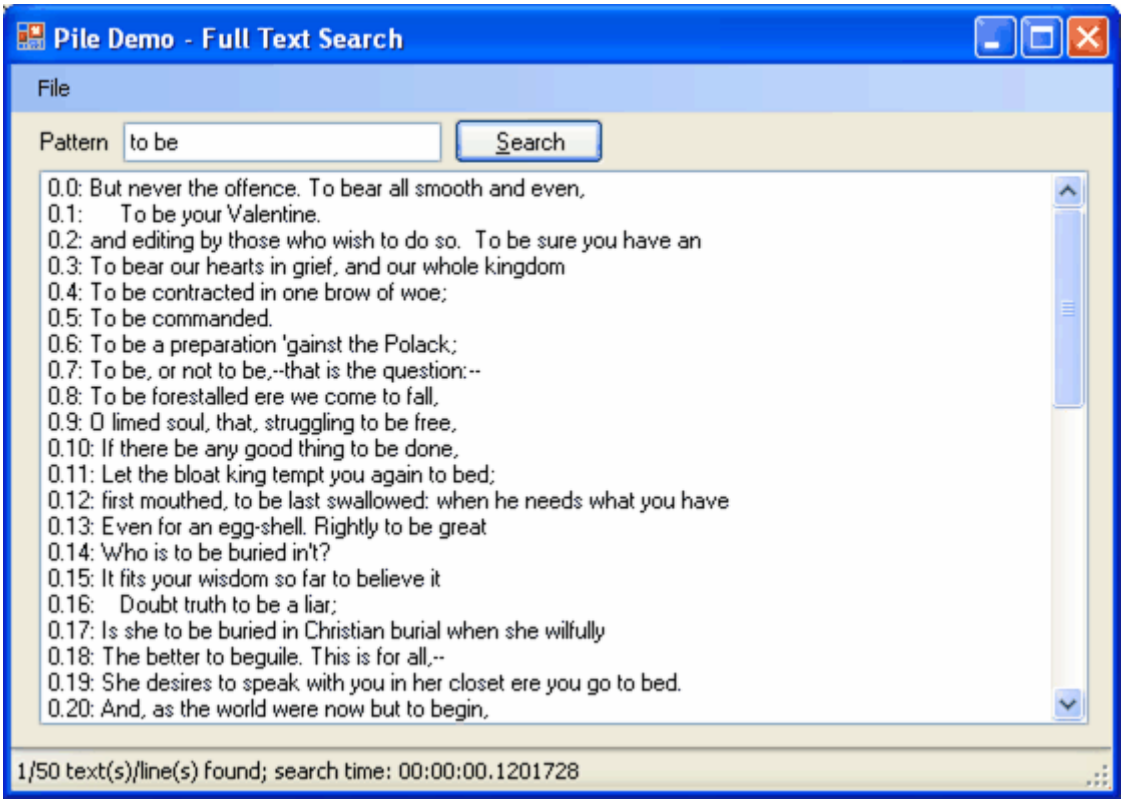
7. Building an associative (data)base - or: The Pile thinking applied

Ok, finally - after another digression on Pile fundamentals (and I guess, more needs to be said, but not right now) - I'll quickly describe my prototype implementation of a Pile engine as well as a Pile agent/client to use it for searching in plain text files.

As stated earlier, the general architecture for my little full text search solution is simple:



There is just one application (process) hosting the GUI front-end...



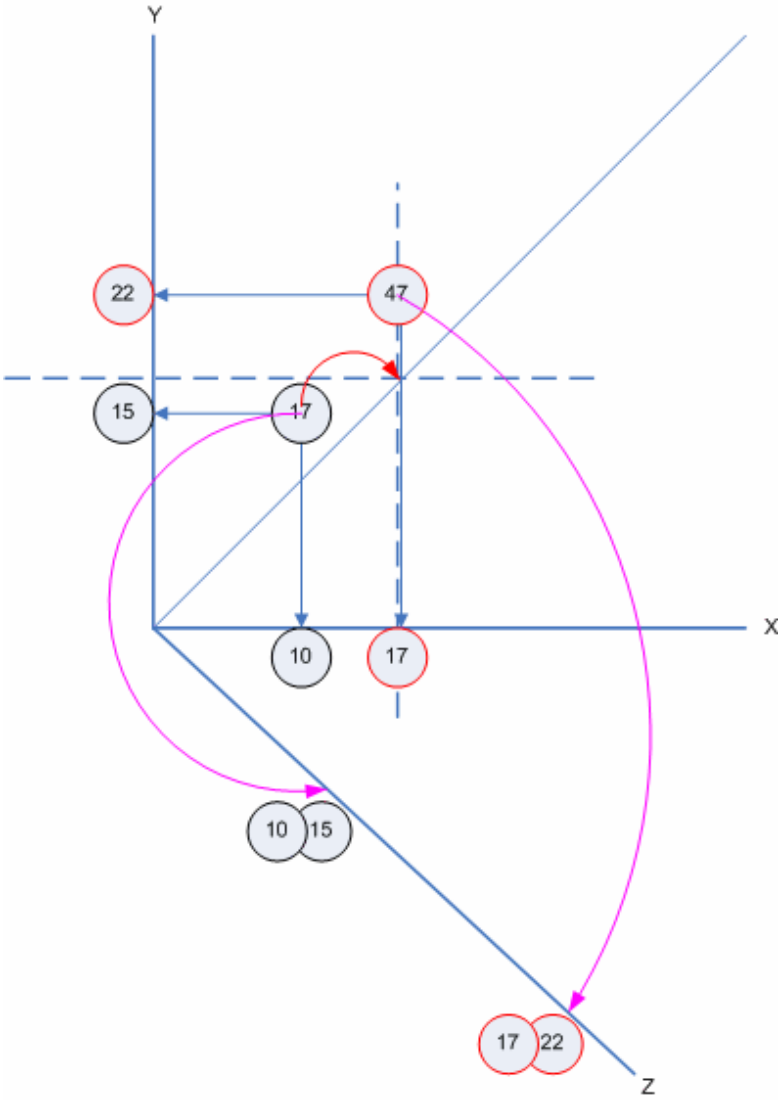
...the Pile agent which actually does the full text search and the basic Pile management code split into a Pile engine and a Pile space engine.

Of course much can be improved in my code. So take it as a first prototype and venture into the realm of Pile. It implements just an in-memory Pile and is not fit for multithreading. But it helped me to switch my thinking from data to associations.

Let's start from the bottom up...

A prototype Pile space engine

The Pile space engine is the low level component encapsulating a Pile. The Pile space engine provides an interface to manage the 2D Pile handle space described in my previous posting. Remember the coordinate system?



To view relations as points in this 2D space with an additional 3rd axis to map them back to their parents, is as fundamental as you can get, I think. Each relation is a point addressed by an x and y coordinate handle(r)=(handle(x), handle(y)). Each such point in the 2D space has a value, its handle, which also encodes the relation's Qualifier.

x and y are the parent relations of r. And the x/y-coordinates are the handles of the parents.

To map a relation back to its parents the z-axis is introduced. Each point on it corresponds to a relation and contains the handles of its parents. (handle(x), handle(y)) = z(handle(r)).

How such a Pile space is implemented, I though, should be hidden within a component or at least behind an interface. So I set up this C# interface to describe all necessary operations to create and traverse relations in the Pile space:

```
namespace pile.engine32.contracts
{
    public interface IPileHandleSpace
    {
        int CreateHandle(byte qualifier);
        void SetHandle(int xHandle, int yHandle, int h);

        int[] GetCorrdinatesForHandle(int h);
        int GetHandleByCoordinates(int xHandle, int yHandle);
        int[] FindHandlesByCoordinate(int coordHandle, bool
followNormativeManner, byte qualifierFrom, byte qualifierTo);

        byte GetQualifier(int handle);
    }
}
```

CreateHandle() creates a new handle for a given Qualifier out of nothing. The Pile space engine does not set up a relation by itself, but just manages some 2D space of handles. So there is no need to pass informations on parents of a new relation to *CreateHandle()*.

SetHandle() stores a handle at the given coordinates in the space. Also it registers the handle on the z-axis to be able to get at its parents.

GetCoordinatesForHandle() retrieves the parent handles for a handle. The int-array contains to items - or is empty is the handle does not exist.

GetHandleByCoordinates() retrieves the handle at a specific x/y-coordinate in the Pile space.

FindHandlesByCoordinate() retrieves for a given handle all the handles to which it is a x- or y-coordinate. Whether *coordHandle* is to interpreted as an x- or y-coord signifies *followNormativeManner*. It the flag is *true*, the handle is the x-coord, otherwise the y-coord. The Qualifier parameters are supposed to constrain the handles returned to a range of Qualifier values. Example: Retrieve all handles, where *coordHandle* is the x-coord (*followNormativeManner == true*) and the Qualifier is between 10 and 12.

Relations are represented by 32-bit integers in my Pile implementation. Qualifiers are 8-bit values.

For each Qualifier the Pile space engine keeps a counter. Creating a new handle thus is just a matter of incrementing this counter.

```
private int[] qualifierCounters = new int[256];

public int CreateHandle(byte qualifier)
{
    return ((int)qualifier) << 24 | ++qualifierCounters[qualifier];
}
```

Since the 2D space is only sparsely populated with handles, it cannot be represented by a 2D-array. I thus chose a two level data structure consisting of dictionaries:

private Dictionary<int, Dictionary<int, int>> xAxis = new Dictionary<int,Dictionary<int,int>>();
private Dictionary<int, Dictionary<int, int>> yAxis = new Dictionary<int, Dictionary<int, int>>();

xAxis[xHandle] stores a list of y-handles together with the handles both are pointing to:

rHandle = xAxis[xHandle][yHandle]

This is sufficient for *GetHandleByCoordinate()* and *SetHandle()* and *FindHandlesByCoordinate()* with *followNormativeManner==true*. If *followNormativeManner* is *false*, though, a second axis data structure is needed to be able to retrieve handles via their y-parent-coordinate.

rHandle = yAxis[yHandle][xHandle]

The z-axis is very simple to model:

```
private Dictionary<int, int[]> zAxis = new Dictionary<int,int[]>();
```

This implementation of a Pile space works pretty well as a proof-of-concept. To become production ready, though, many things have to be improved. No thread-safety yet, no persistence yet, no transactions, no

security yet, no deletions yet etc. Also the data structures need to become more efficient; currently lots of RAM is needed to hold all the relations for a simple text. But that's all a matter of optimization. My concern for now just was to show (myself), how a Pile engine in general could work.

Now, that I know, for many improvements it will be sufficient to change just the implementation of the Pile space engine. As long as no changes to the interface are needed, layers on top of it will not be affected. (In fact I already implemented a second Pile space engine which replaced all in-memory data structures with simple database tables. It used the table-oriented API of VistaDb, a nice small SQL database. Unfortunately, though, this implementation of a persistent Pile space engine was very naive and had very poor performance. However, it showed at least, how easy it is, to replace on Pile space engine implementation with another. Since then, though, I've learned a bit about how a Pile space can be partitioned as to be able to swap in and out smaller pieces. But that's maybe a topic of another posting...)

A prototype Pile engine

The layer above the Pile space engine knows about relations and manners. It also knows about parents and children and Terminal values. Most of the methods of the Pile engine, though, are very thin and just more of less delegate their work to the Pile space engine. The so to speak just translate between terminologies, from relations to Pile space and back.

```
namespace pile.engine32
{
    public class PileEngine
    {

        public class ParentRelations
        {
            private int nParent, aParent;

            internal ParentRelations(int nParent, int aParent)
            {
                this.nParent = nParent;
                this.aParent = aParent;
            }

            public int NormativeParent
            {
                get
                {
                    return nParent;
                }
            }

            public int AssociativeParent
            {
```

```
                get
                {
                    return aParent;
                }
            }
        }


        public int CreateTop()...

        public ParentRelations GetParentRelations(int
childRelation)...

        public int GetChildRelation(int normativeParentRelation, int
associativeParentRelation)...

        public byte GetRelationQualifier(int relation)...

        public int CreateChildRelation(int normativeParentRelation,
int associativeParentRelation, byte qualifier)...
        public int CreateChildRelation(int normativeParentRelation,
int associativeParentRelation, byte qualifier, out bool
childAlreadyExisted)...

        public int[] FindNormativeChildRelations(int
normativeParentRelation)...
        public int[] FindNormativeChildRelations(int
normativeParentRelation, byte qualifier)...
        public int[] FindNormativeChildRelations(int
normativeParentRelation, byte qualifierFrom, byte qualifierTo)...

        public int[] FindAssociatveChildRelations(int
associativeParentRelation)...
        public int[] FindAssociatveChildRelations(int
associativeParentRelation, byte qualifier)...
        public int[] FindAssociatveChildRelations(int
associativeParentRelation, byte qualifierFrom, byte qualifierTo)...
    }
}
```

CreateTop() returns a relation without parents. It's called by a Pile agent to create a representation for a Terminal value or to create a System Definer. The Pile agent needs to keep track of which handle belongs to which Tv.

CreateChildRelation() on the other hand creates a new relation from two parent relations. Because two particular parent relations can only be related once by a child relation, the method returns this child relation if both parents should have been associated before:

```
public int CreateChildRelation(int normativeParentRelation, int
associativeParentRelation, byte qualifier, out bool
childAlreadyExisted)
```

```
{
    int h = phs.GetHandleByCoordinates(normativeParentRelation,
associativeParentRelation);
    if (h == 0)
    {
        h = phs.CreateHandle(qualifier);
        phs.SetHandle(normativeParentRelation,
associativeParentRelation, h);
        childAlreadyExisted = false;
    }
    else
        childAlreadyExisted = true;

    return h;
}
```

The method first checks, if the child relation already exists. If not, it creates a new handle for it and stores it at the coordinates of the parent relations.

GetChildRelation() returns the child relation of both parents - or 0 if it does not exist.

GetRelationQualifier() extracts the Qualifier from a relation's handle.

Find...ChildRelations() returns an array of relations in the specified manner and optionally matching the Qualifiers passed in. The *Find...()* methods are important when searching in a Pile, because often many relations need to be traversed to check, if the lead to the relation looked for.

A prototype full text search Pile agent

The Pile agent sits on top of the Pile engine and implements a domain specific solution: search for patterns in plain text files. Only the Pile agent knows, which System Definers, Terminal values and Qualifiers are needed. Usage of the *PileAgent* class whose methods you see below is very simple. To load texts into the Pile call one of the *StoreText()* methods:

```
PileAgent pa = new PileAgent();
int tFox = pa.StoreText("The quick brown fox\r\njumps over the lazy
dog.");
int tPeter = pa.StoreText("Peter Piper\r\npicked a pack\r\nof
pickled peppers.");
```

Pile calls loading data into a Pile *assimilating* it. It kinda gets "sucked in" and "shreddered" and so to speak looses its identity. You can't see it anymore as one unit of data. There are only relations of which some are new and others have existed before. But be assured, you can retrieve the text from the Pile, anytime you want :-) What is retrieved, though, is not the original text, but something that looks exactly like it. The text you get

back is generated from the associations in the Pile like a picture in a computer game is generated from an internal model of the game's world.

For the Pile agent call one of the *Retrieve...()* methods. For an assimilated text pass in the handle you got back from *StoreText()*:

```
Console.WriteLine(pa.RetrieveText(tFox));
```

Of course the purpose of a full text search Pile agent is searching for patterns and returning all texts containing it. To do this, call *FindTexts()* and get only the lines with the pattern generated:

```
TextFound[] tf = pa.FindTexts("peter", false);

foreach(TextFound t in tf)
    foreach(int l in t.LinesHandles)
        Console.WriteLine(pa.RetrieveLine(l));
```

A *TextFound* instance represents one texts in which the pattern was found with all the lines containing it. Text as well as lines are returned as relation handles.

This is a minimal API for a full text search Pile agent. No methods to delete texts, no methods to enumerate the contents of the Pile, no persistence. But for loading texts and searching them it´s ok. Here´s the list of all public (and some private) methods:

```
public int StoreText(System.IO.TextReader tr)...
public int StoreText(string text)...
private int StoreLine(string text)...

public string RetrieveText(int handleOfText)...
public string RetrieveLine(int handleOfLine)...
private System.Text.StringBuilder ConstructLine(int handle)...
private void ConstructStringFromRelations(System.Text.StringBuilder
sb, int handle)...

public class TextFound
{
    ...

    public int TextHandle...
    public int[] LineHandles...
}

public TextFound[] FindTexts(string pattern, bool
caseSensitiveComparison)...
```

If you want to try the Pile prototype application now, download it, assimilate some texts, and check out how long it takes to search for patterns of several sizes. You can find suitable texts at Project Gutenberg

(e.g. Hamlet, Macbeth, King James Bible (4.5 MB), or The Koran (1,44 MB)) or take the RFC of your choice (e.g. HTML, or SMTP).

Text representation

You now know how to use the Pile agent. But you sure are also interested in how it works and uses the Pile engine. So let me start with how texts are represented in the Pile.

As Terminal values I've chosen the well known basic 256 characters. For each the Pile agent creates a Top as well as a Root. The Roots, so to speak, are the representations of the Tops within a specific context. This context is spanned by a System Definer.

The Pile agent has two System Definers: One to keep track of assimilated texts, one for the low level representation of text lines. (I'm not satisfied with this implementation anymore, though. But it works :-) So I leave it as is for now.)

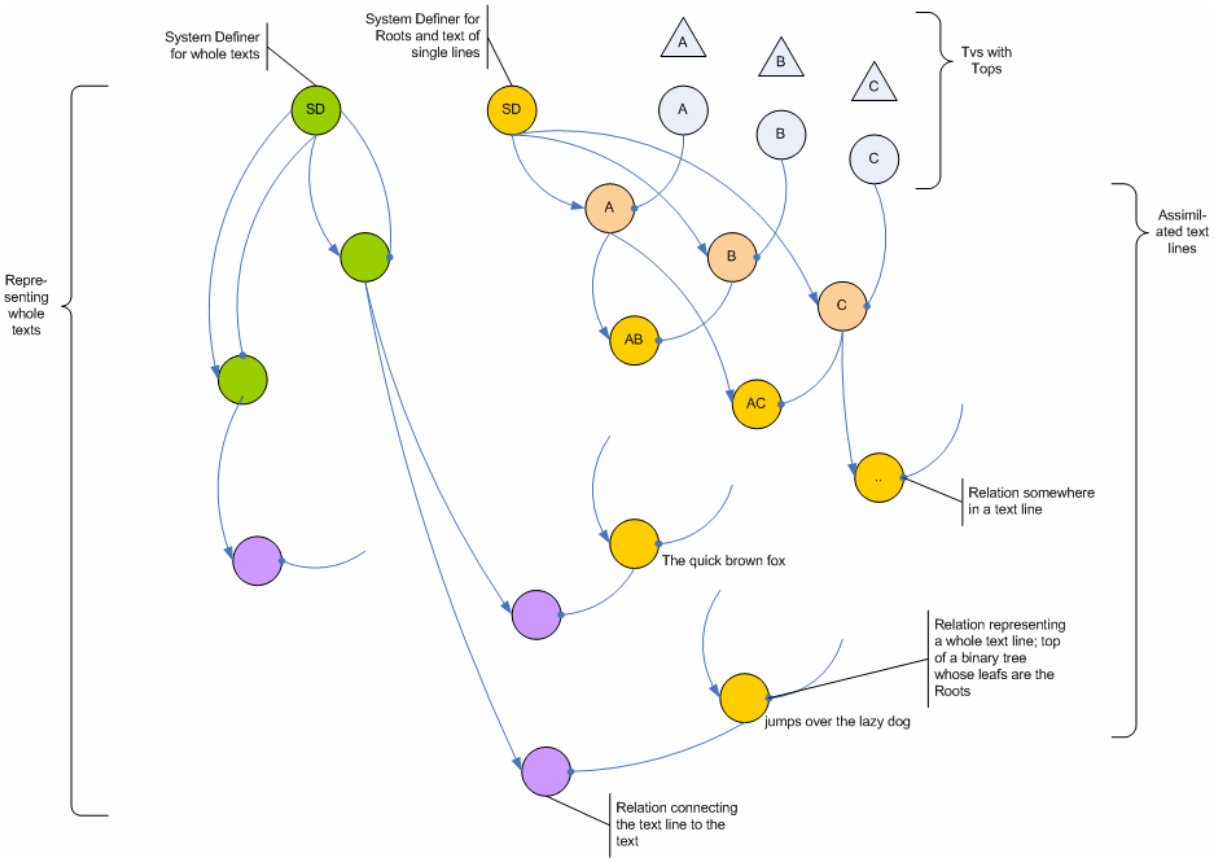
Even though when implementing the Pile agent I did not define a formal schema, you can think of it like this:

Composite Relations:
FullTextPile ::= { Text } .
Text ::= { line } .

Value Relations and Terminal values:
line ::= { letter } .
letter ::= "A" | "B" | ... | "a" | "b" | ... .

The "mesh of relations" then consists of three parts:

1. There are the Terminal values outside the Pile with their internal representations, the Tops.
2. There are Roots and the relations built on top of them to represent individual lines of text. Their tree starts at the *sdLetterRoots* System Definer.
3. There are the relations for texts and lines starting at the *sdText* System Definer.



As you can see, there are two "logics" or contexts or systems in one Pile. One system concerned with handling raw texts lines represented by a single relation on top of a binary tree. And one system concerned with managing texts and lines - and connecting those lines to the text line relations in the other system. The purple relations in the above picture so to speak are transcending the text logic to reach into the basic letter logic.

Retrieving text

To retrieve text from the above associative structure is easy. If you have a handle of a text or line in your hand, you just recursively traverse the tree of relations down to the Roots or Tops and assemble it letter by letter. See ConstructStringFromRelations(), it´s just some 15 lines of code. Very easy. *RetrieveLine()* and *RetrieveText()* build on top of this method.

Storing text

Storing text or assimilating it into the Pile is a little more difficult, but still straightforward. *StoreLine()* does it in some 50 lines of code. *StoreText()* builds on it and just splits the text to store into individual lines and binds together the assimilated lines under a relation for the whole text. Since there is no order in the texts, all text relations have the *sdTexts* System Definer as their normative as well as associative parent.

StoreLine() first creates relations for pairs of letters in the line. For "The quick brown fox" this would be:

Th=("T", "h")
e_=("e", "_")
qu=("q", "u")
ic=("i", "c")
etc.

Then it creates relations for pairs of those relations, e.g.

The_=(Th, e_)
quic=(qu, ic)
etc.

This process of pairing is continued, until only one relation representing the whole text line is reached. It's the top of a binary tree and is returned from *StoreLine()* to be connected to the relation representing the whole text.

Two things are to note here:

1. If a relation already exists, e.g. Th or quic, because it has been created while assimilating some other text, it is reused!
2. Representing a letter of 1 byte with a handle of 4 bytes and connecting two letters with a relation resulting in 12 bytes (for all three handles) sounds like wasting space. But in fact in the long run space is saved. Consider the King James Bible: It's some 4.5 million bytes, but requires only some 800,000 relations amounting to some 3.2 million bytes. Even though the prototype Pile application uses up much more space, this example should show, using relations is not per se wasting precious bytes.

Searching for text

Search for patterns in the assimilated texts certainly is the most difficult part of the Pile agent. It took me some 200 lines of code and a couple of days to get it right (or to be precise: almost right ;-) *FindTexts()* works pretty well, but is not error free yet). A detailed description I'll leave for some other time, but to get you started in analysing my code, let me explain the basic approach, which is characteristic for Pile applications:

When searching for a pattern, e.g. "the", you start at the bottom of the binary trees representing the texts. That means, you start at the Roots of the system.

The algorithm of *FindTexts()* can be sketched like this:

- Set up a list of relations for the characters in the pattern for look for. That means, find the Root for the letters "t", "h", and "e". The list is called *followers*.
- Take the first relation and move it to a list of *firsts*.

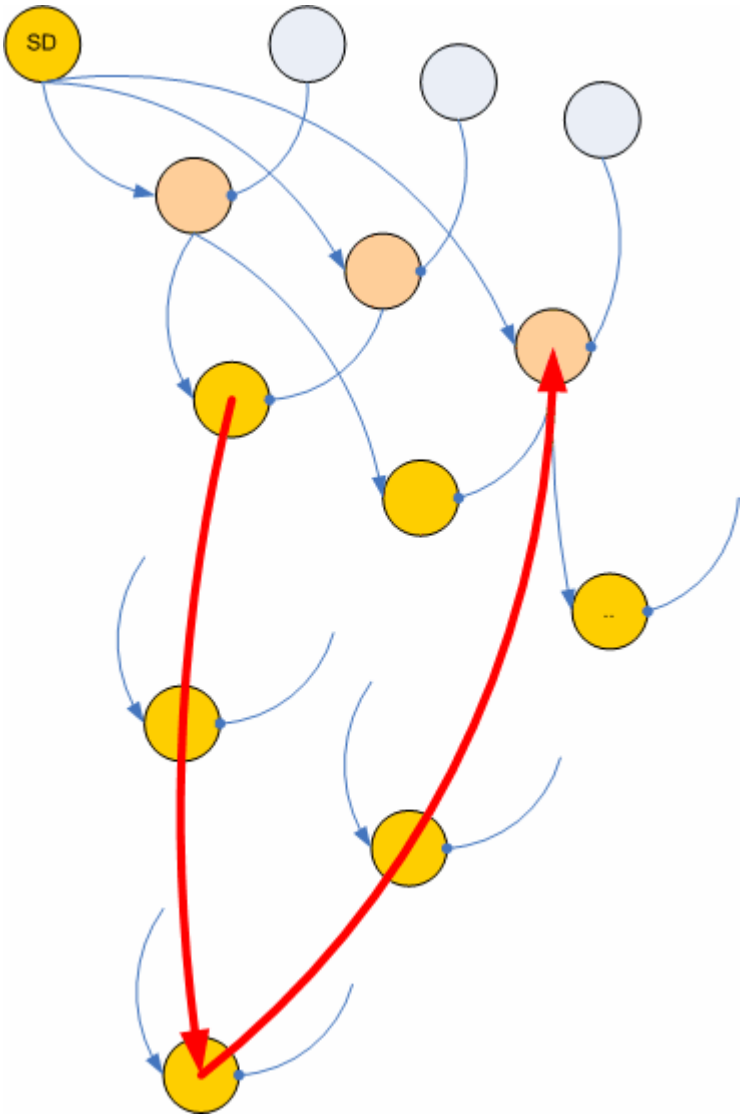
While searching, the pattern is split in two parts: On the left are the relations already found matching the first n letters of the pattern, e.g. the relation th, since "th" has already been found. On the right are the remaining letters that still need to be matched and which follow the first letters, e.g. "e".

firsts needs to be a list, since there might be many locations in a Pile, where a sub pattern occurs, e.g. many lines in which "th" is present. And each of these locations has to be checked, if it also matches the following character.

With *firsts* and *followers* in hand...

Repeat as long as there are follower characters
    Repeat for each relation in *firsts*
        Get the next follower
        Find a path from the current *firsts´* relation to the follower
        If there exists such a path, then remember the relations at the top of this path and add them to the *firsts* list for the next round

The difficult part is "Find a path". This really made me scratch my head a lot. You ll find my solution in the method *SearchValley()*. The algorithm is called *valley search* since you start from some relation closer to the roots, wander the binary tree of relations within a line down closer to the top, then wander it up again to the roots.



You see the "valley" in the picture: the red arrows sketching the search path are going down and up again, like through a valley.

This Valley Search is not so easy, because you need to consider going down relations along the associative and then normative manner and then going back up along the associative and then normative manner. If there was just one line of text and one location for a pattern within this line, then all would be pretty easy. But many locations in many lines... you really need to keep track in your algorithm of where you are, which path you follow. Recursion helps, though :-)

But let the details of the search algorithm not deter you from the basic search paradigm: When searching in a Pile you start at the Roots! That's the fundamental insight you should take away from here.

Download

You can download the prototype Pile application from my website here: http://eh17.s5.domainkunden.de/download/prototype.pile.engine.zip

Check it out, toy around with it. I hope you find your way around in my code. I at least tried to structure it so it makes sense to you.

You need Microsoft Visual Studio 2005 (!) for it to compile. But I guess the express edition, which is very cheap or altogether free, will do.

Let me know what you think, if you like: info@ralfw.de

I guess, this will be the last posting on Pile for a while. I need to learn more about it. I need to move on to other projects I unduly neglected because I ḿ so infatuated with Pile ;-) But after having written all this I now kinda feel relieved.

(Posted 27.12.2005)