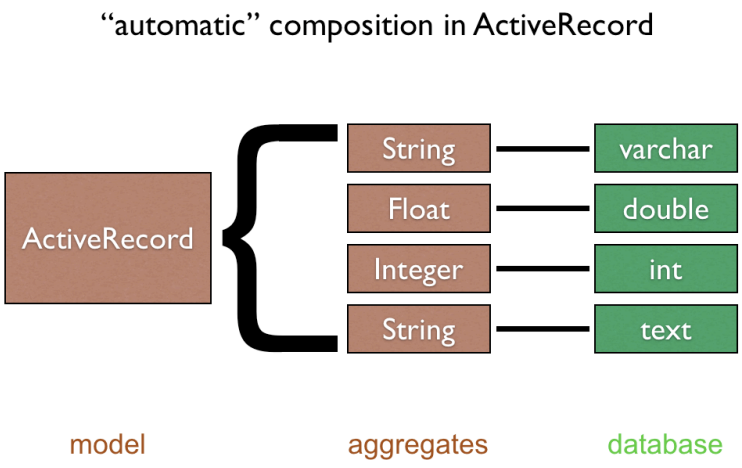
Obscure Data Formats - RailsConf Europe 2007 - By Chad Thatcher

In the ***first part*** of this paper I will **introduce "composition" in ActiveRecord** and **highlight** some of its **benefits** and **uses** in our models. This will be **followed** by a few **examples** to expand the **concept** and put **everything** into **context** when I **demonstrate** the project which I will do in the **second part** of the paper.

I will then **go into** some **detail** about the obscure data **format** the project uses called "**MARC**" and **show** how **composition** is used within the project to **solve** some **difficult** problems encountered.

PART 1 - Composition

Composition or **aggregation** is one of the most powerful **building blocks** of any good **O**bject **O**riented **design**. It is also a key **component** of **object relational modeling** seeing as **fields** are **aggregated** within the model object either **explicitly** or **implicitly**. **Hibernate**, for example, asks us to **explicitly define** our composition through an **HBM xml** definition. With **ActiveRecord** this is **implicit**...



...which is both a **blessing** and a **curse**. **Blessing** in that because of the very **nature** of Rails we **don't** have to **muck** about with **configurations** and this is great because **99%** of the time this would be **wasted** effort. But is is also a little **bit** of a **curse** in that we can sometimes **overlook** an **opportunity** to capitalize on the power of composition.

Lets look at the **features** of **rolling** our own **compositions** in ActiveRecord by using the macro-like class method **"COMPOSED_OF"**

composed_of

- allows us to take control of a models composition by mapping our database fields to objects other than those that are mapped automatically
- we can store data in a simple way, but use it in a complex way - or visa versa
- we hide the details of our data and how it functions
- can be a better alternative to inheritance

*And here are **some of the ways** "composed_of" can be used:*

```
composed_of :property,
  :class => :Aggregate,
  :mapping => "field"

composed_of :field,
  :class => :Aggregate,
  :mapping => "field"

composed_of :property,
  :class => :Aggregate,
  :mapping => [ %w( field, attr ), %w( field, attr ) ]
```

Using the "**composed_of**" method in its most **basic form** we simply **map** a table **field** to an **object** of our choice. And the **name** of our **composition** becomes the **access point**.

We can also **map** an aggregate to a field by **naming** the "composed_of" **property name** the **same** as the **field** name. This is a **subtly** different to the **first** example but in this way we can completely **hide** or **protect** the **attribute**. In the first example the **user** can still **directly set** the attribute by simply **referring** to it rather than its **composed_of** property **name**. In the **last example** referring to the **attribute** will always result in the **aggregate object** being **returned** or **referenced**.

We can also **map several fields** in a single composition to a **single** object. Being able to **manage several** table **fields** in one aggregate is very powerful if in the very least it **gives** us the **ability** to **fetch** or **set** this data in a **compound** way. For example, lets say I am **recording** the **addresses** of **resources** on the net and for one reason or another want to **store** that **URL** in it component **parts**, the server name, a file path and other cgi key/value

pairs. By **using** an **aggregate** I could **get** at those **parts individually** and also deal with the **URL** in its **complete form** and **never** have to **worry** about how to **break it up** or **reconstitute** it. This is what it might look like...

```
class Asset < ActiveRecord::Base
  composed_of :url,
    :class_name => :UrlMagic,
    :mapping => [ %w(server_name server), %w(file_path file) ... ]
end

...

class AssetsController < ApplicationController
  def do_something
    asset = Asset.find(params[:id])
    full_url = asset.url.url
    ...
    server = asset.url.server
    file = asset.url.file
    ...
  end
end
```

```
class UrlMagic
  attr_reader :server, :file

  def initialize(server = nil, file = nil)
    self.server, self.file = server, file
  end

  def url
    self.server + self.file
  end

  def url=(full_url)
    self.server, self.file = full_url.split(/.../)
  end

  def ==(other_url)
    url == other_url.url
  end
end
```

There are a lot of **examples** of using **composition** in **ActiveRecord** out on the net.

possible applications

- the "currency" example
- the "address" example
- legacy formats
- mapping or 3d data
- hierarchies like hdf
- bitfields
- digital asset management

The most **common** one being the "**currency**" example wraps **double** fields that hold **money** values and the aggregate gives us **convenient** methods for getting those values with a **precision** of to two decimal places or doing **currency conversions** etc.. The "**address**" example wraps several fields in a model that represent a **postal** address and gives us various conveniences like fetching the address **ready** for **label** printing.

The **examples** so far make it easy to **miss** some of the potential of "composed_of" which really starts to **shine** when your aggregates **manage complex** data or **reflect** into the database or call on **external** services or resources.

And “**composed_of**”, **being composition**, also allows us to **prefer** and use **composition** over **inheritance**.

```
composed_of :asset,  :class_name => :Asset,
:mapping => %w(asset_name, asset_path, asset_type)
```

- one field contains the file name
- another contains the path
- another for media type, ordinarily used for STI
- delegate specific object for given media type

In an **asset** management system, for example, the usual **suspects** might be lined up for managing digital assets like **single table inheritance**. But we could also use **composition** in **preference** to **inheritance** here and wrap up our asset in an aggregate, which in turn would **aggregate** the **appropriate** object required for managing a **particular** media **type**.

In a **3D** application we might have a massive **collection** of **points** and for efficiency want to **store** these in a **single** field or in **groups** of **tuples** instead of running tables with millions and **millions** of **rows** and **costly joins**. **Aggregates** would **help** us do this and provide us with a clean way of **encapsulating** operations we might want to carry out on that data.

A **similar** thing could be done with **mapping** data **provided** you **don’t** want to actually **involve** those values in **searches** but even then, as you will see shortly when I **demonstrate** RISM, there are other viable **solutions** for this.

Another great use for composition is for **storing** large **structured documents** like **XML** when there is no need to **keep** them in an **unraveled** state in the database. An aggregate **implementing DOM** will give us convenient **access** to the **inner parts** of the data.

It is also **provides** a means of keeping our code **DRY**

- structured documents like XML
- keeping DRY
- serialized objects
- entity relationship modeling

when we spot the **same** fields popping up in several places that require some sort of **special handling**.

There is also **potential** in using **serialization** provided we are not **limiting** the **scope** of our project with this kind of **implementation specific** data representation, but this is quickly getting into a much **debated area** and I am not going to get into that.

Or with **something like Entity Relationship Modeling** - the idea that you don't **normalise** you data entities in the RDBMS but rather have one **giant table** and allow the **users** to **define** their **own types** and **rules** around those types. **Composed_of** could be an **excellent tool** for this.

PART II - Case Study

The **project** I have been working on - the International Repository of Musical Sources - was **founded** in **1949** with the aim of **locating** and **documenting** all **surviving musical sources** dating from **earliest** times to about **1800**. Over the years around **420,000** catalogue **records** have been added to the database.

In **2003** RISM UK - run as a joint effort between **Royal Holloway** of the University of London and The **British Library** went **online** providing its part of the collection as a **public resource**. **Many** other countries have done the **same**.

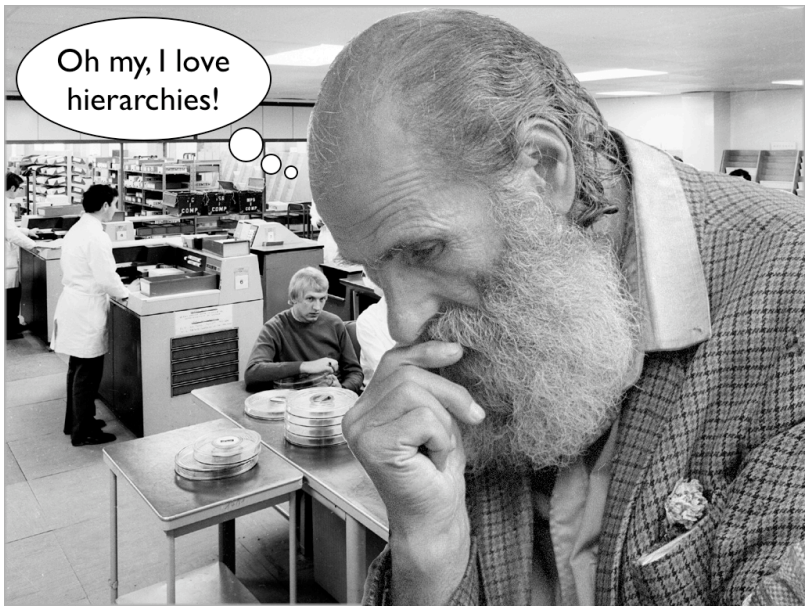
The **final** UK and Irish **contribution** to the project will be around **110,000** catalogue **records** and the online database may eventually **include** the **contributions** of **other** countries.

- the final contribution from UK & Ireland to the RISM project will be roughly 110,000 catalogue records
- RISM UK online may eventually include contributions from other countries

Because of the **age** of the project and the fact that **95%** of all of the projects **sources** come from **libraries**, **archives** and **museums**, its not surprising that it uses **MARC** as its data storage format. So **what** exactly **is MARC**?

MARC stands for **MA-chine Readable Catalog-ing**.

MARC was **invented** in the **First Age of Comput-ing** when **IT departments looked like this** and IT professionals still wore **tweed!**



MARC is an **international standard** for storing **bibliographic** records. Much of the **design** of MARC was **influenced** by those little **index cards** we still find in most libraries today. At the time **hierarchical** databases were the **dominant** means of storing data, and **relational** databases **didn't** even **exist** apart from being a **theoretical** subject found in **PhDs**. So **MARC** is a **hierar-chical** format and over the **decades** as it was **adapted** by **different** countries and **organizations** took on a number **variations**. Here is an **example** record of one of **todays** most **dominant** MARC **formats** - "**MARC 21**":

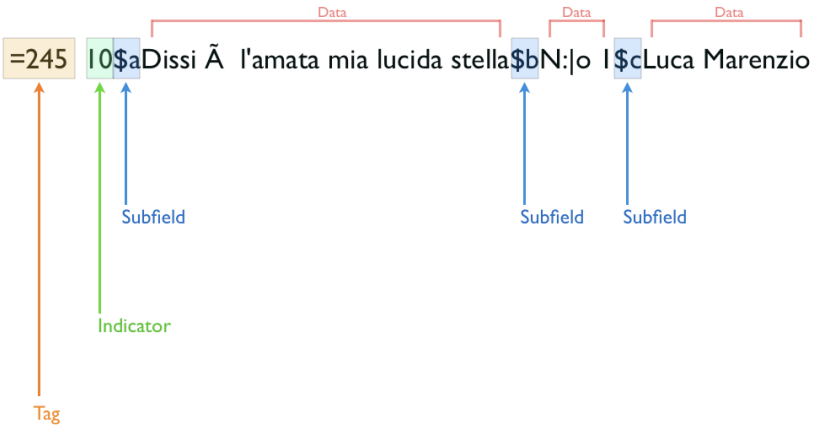
```
=000  00000ndd#a22000005a#4500
=001  20040806149626
=003  UkLU-RH
=005  20010101111111.0
=008  010101q17501800en ||||||||||||||||ita|d
=033  \\$a1585----
=040  \\$aUkLU-RH
=100  1\$aMarenzio, Luca$d1553c-1599
=240  10$aDissi a l'amata mia lucida stella$mV (4)$rF major
=245  10$aDissi Ã  l'amata mia lucida stella$bN:|o 1$cLuca Marenzio
=260  \\$a[S.l.]$b[s.n.]$c18th century, second half
=300  \\$a4 ms. parts: p. 1-2$c11 x 14 cm
=508  \\$aText author: Giovanni Battista Moscaglia
=518  \\$aPrinted in Marenzio's Madrigali a quatro voci. Libro Primo (Roma 1585)
=580  \\$aSee also the general description of the collection containing this entry 20040806149625
=590  \\$aS, A, T, B
=594  \\$aVSol 1111
=596  \\$aRISM A/I, M 578
=598  \\$aCanto, Alto, Tenore, Basso
=650  00$aVocal quartets, Unaccompanied
=700  1\$aMoscaglia, Giovanni Battista$d1550c-1587p$4lyr
=710  2\$aBritish Library$bMadrigal Society Collection$k(Manuscript)
=740  0\$aDissi a l'amata mia lucida stella
=773  0\$w20040806149625$7nnda
=787  \\$a20040806149933
=787  \\$a200408061530 94
=789  \\$a1.1.1$bS$eC-1$fbB$gc$h'1F/G/4-2A"4C/'BAGA+/8A$i20040806149626$nF
=852  \\$aGB Lms$eLondon (Great Britain), Madrigal Society Collection$pA.1-4$x(Manuscript)
```

The **21** in the name stands for **21st century** believe it or not.

It can be quite **frightening** to look at at **first sight** but if we drill into it a bit, its elements become quite obvious. A MARC record is **made** up of **several tags** and you can see those **stand out** at the be-ginning of **each line**.

Here is one of these **tags** in **isolation**:

Each **tag** is **represented** by a **number** which signifies its **function**. There are a number of **definitions** floating around for the various MARC **formats** for what these tag numbers represent.

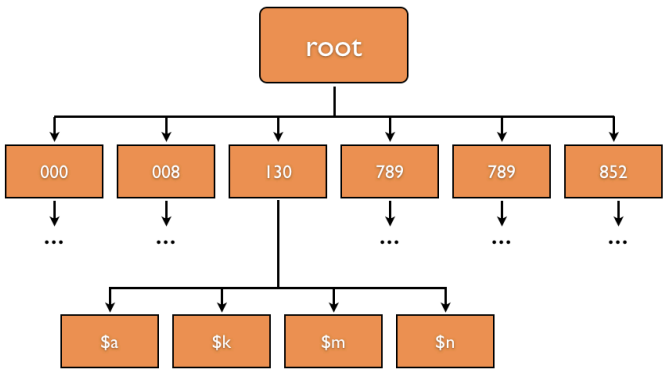


This is a **245** tag which contains a "**title**", "**subtitle**" and any **personal names** that might actually **appear** on the **manuscript** or document. The **indicator** is there for historical reasons and there is **no** need to go into any **detail** for this paper.

Then comes the "**field**" - all the **remaining data**. Bear in mind that the **terminology** used in the day was **slightly different** to what we are used to today. Here the "field" **represents** the **whole data** part of the tag which includes the **subfields** it can be broken down into.

Subfields have a **dollar sign** for a prefix, in this particular **format**, followed by a single **letter** or **number** which signifies the **function** or **meaning** of the **data** that follows.

With that **fresh in mind** here is **MARC** in its hierarchical layout.

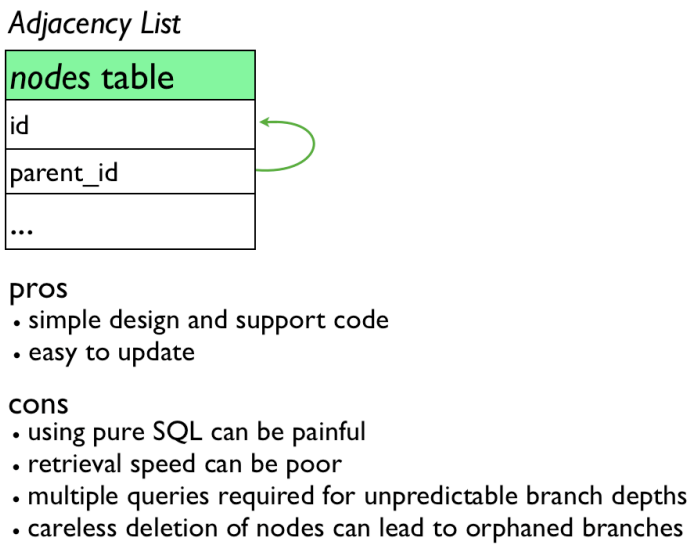


Its important to **realise** that the **order** in which many of these tags **appear** is very important. For example, the **789** tag carries **musical** scores so **if** there is **more** than **one instance** of these tags, they fall **out of order** they would **not make sense** - because the musical bars themselves would be out of order.

And the **order** of **some subfields** can also be **significant** for various other reasons.

Anyway I think thats **enough** on the MARC **format** because it is **like** many **other** hierarchical data structures which we are all **familiar** with. I just wanted you to **feel** a little bit of my **pain**.

So **how** is this **stored?** Well, there are **several** well **known** techniques for **storing** hierarchical or network models in **RDBMSs** and most **perform** well on the **whole**. Perhaps the most **common** technique is the "**adjacency list**" the one that is used by the "**acts_as_tree**" plugin:



This particular **technique** has a **foreign key** like "**parent_id**" pointing to the id the **parent node** in the same table. This kind of **representation** is **easy** to **update** but does run into **trouble** when there are a **large numbers** of nodes in the hierarchy, because of the fairly **heavy use** of self **joins**. This was actually the technique **used** in the **previous incarnation** of the **RISM** project.

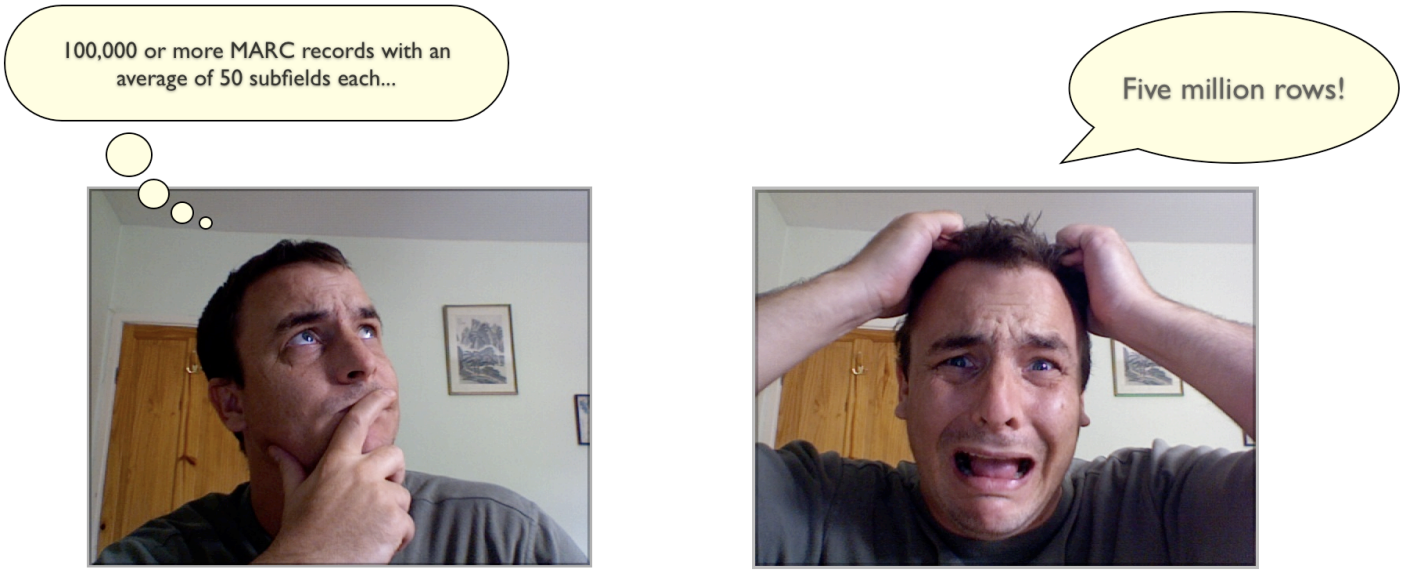
This is **not** a **presentation** on how to **represent** hierarchies in a **relational** model, nor is it an **argument for** or **against** doing this but I thought I would just **go over** adjacency lists quickly so things will make more sense when I discuss later how the **previous incarnation** of the RISM project **struggled** with this technique. And to understand this better lets **look** at some of the projects **requirements**:

RISM's MARC structure requirements:

- Huge number of nodes
- Multiple occurrences of some tags and subfields
- Accurate ordering of the above
- Frequent changes to tags and subfields

In my experience of **dealing** with such a **massive hierarchies** in a **relational** model is that you are either **fighting** with **slow selects** or **heavy** row **updates**. If your hierarchy is **small** and or **not often updated**, then storing it in an RDBMS is still a **viable** and **preferable** option. If your hierarchy is both **large** and **updated often**, like the RISM project, then you may encounter **difficulties**.

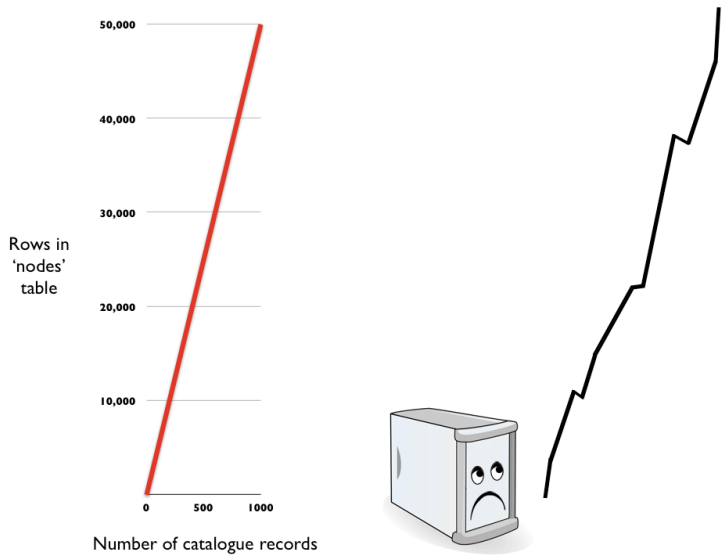
Given that **each** MARC **record** has an average of **50** subfields, and hence **50 nodes** in any given branch, I needed to make one **careful consideration**: **How many** manuscripts would eventually be **catalogued** in RISM and hence **how many records** would there be representing **each child** in the hierarchy?



Now **five million** rows may not seem like a massive **amount**. After all there are **databases** out there running **happily** with far more than this. But 5 million was **just** the **estimate** coming off the starting block. **Combine** this kind of volume with choppy, **sporadic updates** to the data set and the more **common techniques** for storing hierarchies in a relational database become **unsuitable**.

Besides, **updating** or **inserting** records in a table of 5 **million rows** on a good day is going to be a **slow performer**. There was also the small **fact** that the "**content**" field that stored the data of **each subfield** had to be a text, or blob in order to house the data of some of the **larger** subfields like "**descriptions**". Given that a "node" table would be **managing** all nodes in a **generic** way, **even for** nodes whose **data** is entirely **predictable** and **measurable** would have their data bunged into a text column. **Efficiency suffers**. And **future growth** leaves is predictably **steep**.

And you're **asking** your **hardware** to climb a **mountain**...



You may be thinking that a **solution** would be to just **chuck cheap** iron at the problem, but this is a **publicly funded** project and thats a **big ask** no matter how cheap the iron. The other issue is that this **application** needs to **run** as a "local version" on small **laptops**, and I will explain why in a minute.

So I started to **look** at the **motivation** behind **storing** these MARC records as **hierarchies** in the **database** and found that the only real **requirement** behind it was to be able to **search** the content of individual nodes. And there are a **number** of **solutions** that could take care this, most **notably** full text indexing with **FERRET** so I began to explore using this to free up this limitation.

I still needed to be **able** to **manipulate** a MARC record **as a tree** to get at **individual subfields** or **add** or **delete** them and **composition** seemed like a good **candidate** so I experimented with "**COMPOSED_OF**". After a few prototypes I found that it would do nicely.

Here is **how composed_of** is **used** in **RISM**. It simply **maps** a field called "**source**" in the manuscripts table **to** the **Marc** class.

```
composed_of  :marc,
  :class_name => :Marc,
  :mapping => 'source'
```

Marc class is:

- a special MarcNode (the root node)
- contains a tree of MarcNode objects
- natively aware of the MARC format
- lazy
- database aware
- imports/exports different MARC formats

"**Marc**" **is** a **MarcNode** which is a **straightforward node** type that you **find** in many solutions for **representing** trees. Here **Marc** is a **descendant** of

```
class Manuscript < ActiveRecord::Base
  composed_of :marc, :class_name => :Marc, :mapping => %w(source)
  ...
end


class Marc < MarcNode
  ...
end


class MarcNode
  ...
end
```

MarcNode because it **provides** a few **extra services** and **behaves** slightly **differently** being the **root** node.

Each MarcNode, of course, **contains** a **tree** of other MarcNodes which **represent** the **hierarchy** of the MARC record found in the "SOURCE" field.

It **knows** the MARC **format** and can **parse** it easily and **quickly**, populating its MarcNode tree along the way.

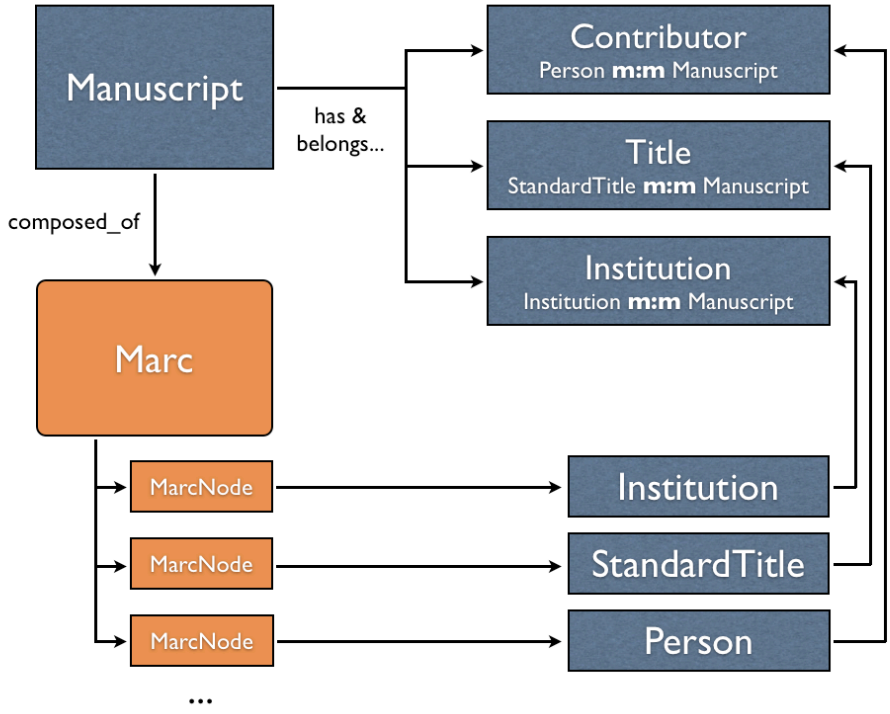
It is **database aware**, and I will explain in a moment why this is.

And it can **handle** various **other formats** of MARC apart from MARC 21 and is able to **export** to and **import** from these formats.

There was one other **big requirement** in RISM which was to **manage** certain **subfields via** other database **tables** in the system. For example, **PEOPLE**. They **didn't want** the cataloguers creating **several copies** of the same person **across** different **manuscript** records. This is **easy** to do when **different** cataloguers are **free typing** and **misspelling** data. So the project wanted **only one occurrence** of **Johann Sebastian Bach** for example, because there is **only one**. The **same** would go for many other **values** in the system, like **standardised titles**, **terms** and **institutions**.

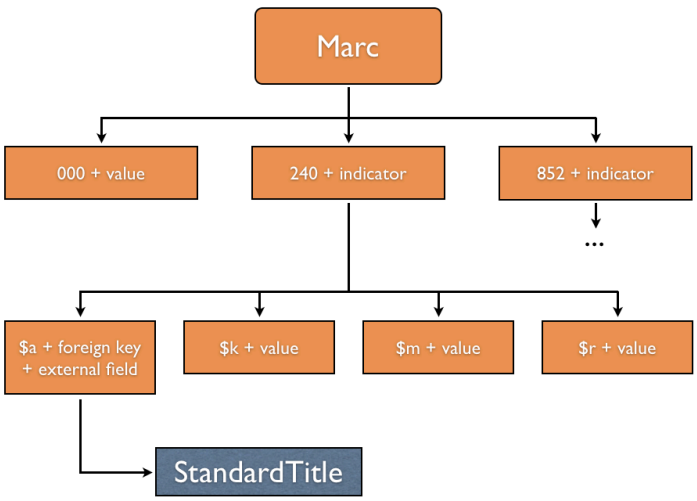
This **means** there is a **normal** database **model** with separate **tables** for **managing** these **entities** in a central place.

The MarcNode **tree adds** another **facet** if you will by **living** on the **boundary between** what would be considered the **normal** "**database**" and its **old world MARC** representation.



Here we can see that the **main** model in the system - **MANUSCRIPT** - is **related** in the **normal** way to the **other** models of the system. Anything that is a **part** of the normal data **model** is in **blue** and the unorthodox **Marc** and **MarcNodes** in **orange**. **Various MarcNodes** then also **reference** parts of the database.

This might **seem** a little back-to-front or **messy** but it **works** very well provided the boundaries of responsibility are well observed.

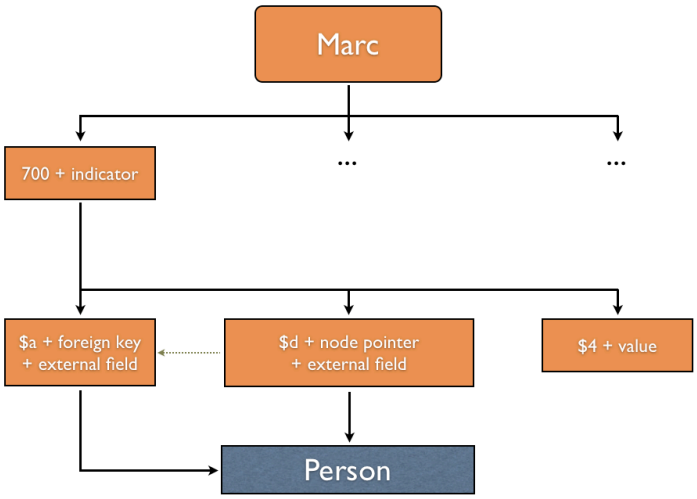


I **chose UUIDs** as **pointers** to the real database **rows** because of the need to **run RISM** as a **local version** on small **laptops** carried by **cataloguers** who would find themselves **cataloguing** in some **remote library** without internet access. The **UUIDs keep** the **remote synchronization robust** by **avoiding** ID **conflicts** when the cataloguer finally gets home or back to the British

Library and **establishes** a **link** with the server.

To **make** things even more **delightful**, some **subfields** are **dependent** on other subfields for **fetching** their real data from the database.

Here the **A subfield** which represents a persons **full name** contains the **reference** to a Person **model**. The **D** subfield which **represents** that persons **life dates** references the same Person model through its **master** subfield **A**.



Because all of the **relationships** between the tables are **maintained** as a **normal** data **model** I get all the **advantages** of that as well. If the **spelling** of a persons **name** is **changed** for example, I **needn't** do anything to **change** the **contents** of the **MARC records** because they are holding references.

However! I still **need** to **consult** the **database** through the normal means when **entities** are **updated** in order to **find** out which **manuscripts** have been affected by the change. I do this is **simply** so I can **find** out which **manuscripts** I need to ask to **reindex** themselves in **Ferret**.

```
=000  00000ndd#a22000005a#4500

=008  010101q17501800en ||||||||||||||||ita|d

=033  \\$a1585----

=100  1\$aMarenzio, Luca$d1553c-1599

=240  10$aDissi a l'amata mia lucida stella$mV (4)$rF major

=300  \\$a4 ms. parts: p. 1-2$c11 x 14 cm

=773  0\$w20040806149625$7nnda

=789  \\$a1.1.1$bS$eC-1$fbB$gc$h'1F/G/4-2A"4C/'BAGA+/8A$i20040806149626$nF

=852  \\$aGB Lms$eLondon (Great Britain), Madrigal Society Collection$pA.1-4$x(Manuscript)
```

I use a **homegrown Ferret solution** because I unfortunately couldn't use the "**acts_as_ferret**" plugin as it does **not cater** for this kind of situation where I am effectively **breaking** up one **column** into many **parts**. Luckily, **each MarcNode knows** whether or not it should be **indexed** and if so **how**...

Some subfields like the **008** tag contains data that needs to be **broken** down further - in this case **years** which need to be **range searchable**.

789h subfields contain an **incipit** representation of the first few bars of the music. This is a **textual** representation of the **music** in a **notation** called **Plaine & Easy** invented so that you could use the **characters** found on a **typewriter** to represent music. These **incipits** need to be **broken** down in all sorts of clever **ways** to allow people to search for **parts** of a musical piece or **similar** melodies etc etc.

On top of **quick** and **advanced search** I am also using ferret to **drive** the manuscripts **lister**. This kind of **search over query** for data **navigation** is becoming more and more **prevalent** as a **cheap** and **effective** solution to **unacceptable response** times. And it is **difficult** to **ignore** other **benefits** such as easy **filtering**.

To give you an idea of the **difference between** the **old** project and the shiny **new** one heres some stats:

	old	new
language/framework	php / homegrown	ruby on rails
lines of code (excluding framework)	8,010	1,874
avg. lister response time	8 s	0.03 s
avg. search response time	42 s	0.03 s

In conclusion, taking control of your model's composition can open up a world of possibilities and provides you with a way of working outside of the normal paradigm. Ferret can also prove to be a nice complement to composed_of allowing you to cater to some requirements which would otherwise force you to follow a strict 3NF design.

Thanks for ~~listening~~ reading!

By Chad Thatcher