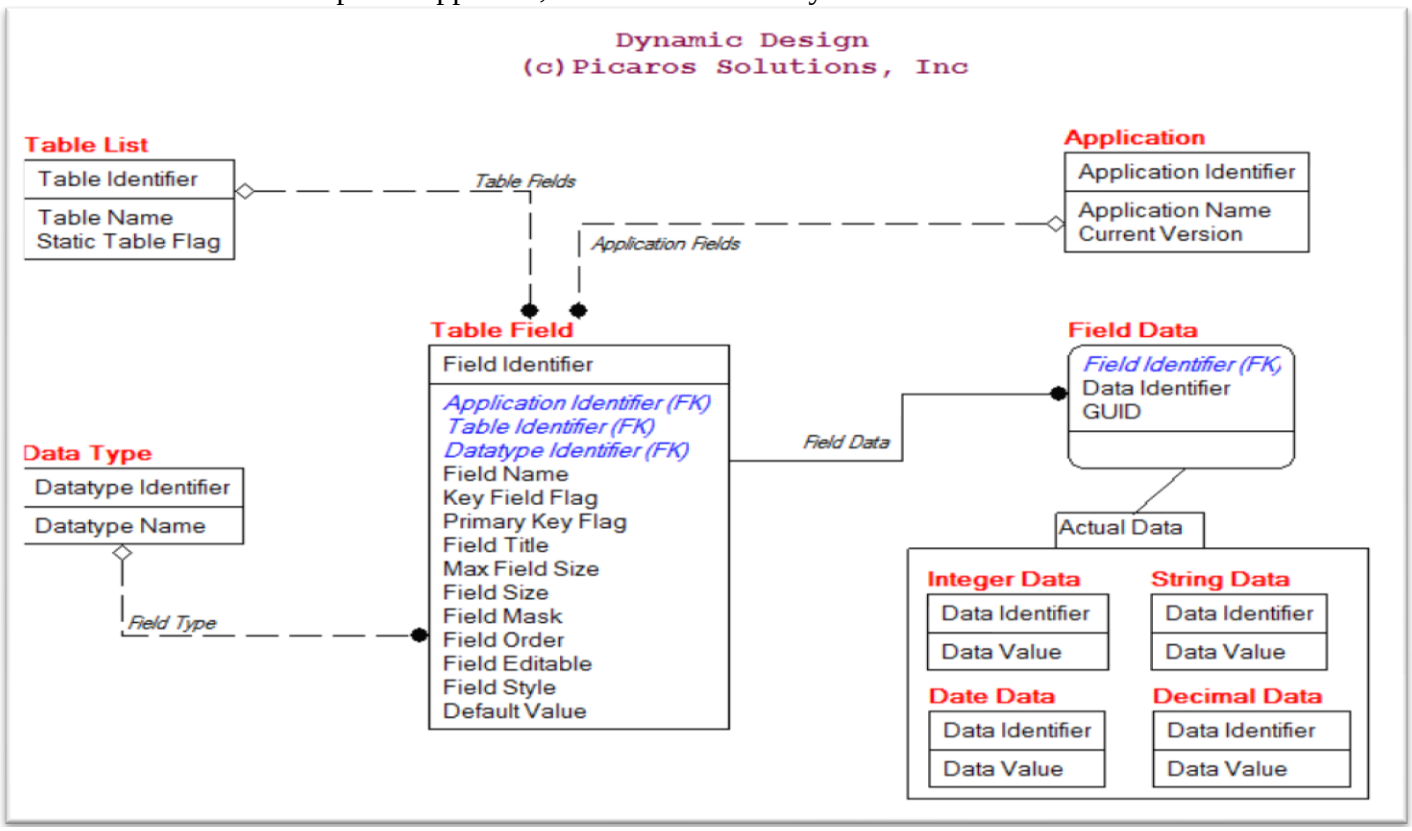
"Dynamic" Metadata Based Database Design

One situation that we often face when dealing with a dynamic environment is how to build solutions that accommodate dynamic changing structures and this is a conundrum from both an operational and warehousing perspective. Situations like changing structure of incoming data, the need to customize software packages for different clients and so forth. While there is no real silver bullet, a metadata based approach can pave the way. However, there are some considerations to keep in mind:

- The metadata based approach can be very effective for operational systems where one has the advantage of using (custom) programming to dynamically interpret the model and produce relevant results – most ERP systems today do a similar thing. Although, having seen some in action, sometimes I wonder at the wisdom behind the complications (read mess) they create
- Reporting tools could be used effectively in ad-hoc environments. Clearly, canned reporting cannot be expected to be realistically dynamic beyond a point
- For analytics using BI tools, it would be a very tall order, atleast in my opinion. These tools would work well in architected environments based of solid warehouse or datamart designs, not really trying to interpret metadata

Let us look at a simplistic approach, a skeleton model if you will:



Please note that as I illustrate, this is still high-level obviously, so there will be gaps in my statements, I am not even trying to be completely exhaustive in my commentary as it is mostly intuitive here on.

- **Application** is not really a stretch assuming this is to be used as a common central model across applications, or it (or a variant) can be used for something else, a collection of some sort
- **Table List** is the basic list of tables that an application would need
    - Static Table Flag **would indicate that the is table is part of core design I am extending**
    - **The attributes listed here is just a sample, one can customize this as needed obviously**
- **Data Type** is the full list of available data types supported by the specific DBMS engine being used – one could think of making this exhaustive to avoid portability issues – afterall this is just data
- **Table Field** is the list of attributes per table along with specifications – highly customizable
- **Actual Data** is where the field data would be "stored" – albeit it will only be a pointer to the real place where data in its needed native format is stored
    - Point to note here is there is an implied one-to-one relation between the number of data types and the number of **Actual Data** tables – 4 in the above example – we can be clever how this is done physically – for instance **String Data** can be used for multiple data types
    - One would also question as to why not store everything as string/text and convert as needed
    - Storing it in native format has advantages instead of constantly converting the data for (say) performing mathematical operations on it. Also, when we denormalize the structure, we do not have to first create the structure and load data, a SQL like "CREATE XYZ AS SELECT…" or "SELECT * INTO XYZ"

While the above is simplistic, it should not be used (I wouldn't) to encompass the entire database design – we could capture an entire database design within this simple design, subject to "beefing" it up ofcourse in a relevant manner providing more usable attributes. What I would use this for, is to extend a static design that I would do in a normal situation assuming no "dynamic" need. The static design would encompass the core functionality that is fundamental to the application. Although, other than performance arguments can come up in an OLTP environment, I cannot think of a reason as to why not.

***The above solution is purely a (kind of – I wouldn't call my column names like the above) physical manifestation – the modeler still needs to logically capture the functionality needed to enable population of the metadata into the above tables*.**

While this design is useful to capture data in a changing/dynamic environment with some clever programming, it is not readily accessible for reporting purposes. To enable such a design to be usable for reporting (without having to jump hoops and this means interpreting metadata about the available tables and fields), we have to think in terms of "flattening" the structures & data.

- A cleverly written stored procedure or program can create a table with proper datatypes and push all the data into it, so it would look like a real table that we would have normally designed – this table would change structurally if we change the design, so controls would need to be established
- In applications where maintenance is involved, update/delete functionality can be carried out with some clever programming to account for integrity

Let me illustrate this with an example:

Application

Application Identifier	Application Name	Current Version
1	Dummy	1.0
…		

Table List

Table Identifier	Table Name	Static Table Flag
1	TABLE1	Y
2	TABLE2	N
…		

Data Type

Datatype Identifier	Datatype Name
1	Date
2	Decimal
3	Integer
4	String
…	

Table Field ***(Keeping it simple for this illustration)***

Field Identifier *(Non-intelligent)*	Application Identifier	Table Identifier	Datatype Identifier	Field Name	Field Order	…..
101	1	1	3	FIELD11	11	
102	1	1	3	FIELD12	12	
103	1	1	3	FIELD13	13	
104	1	1	1	FIELD14	14	
1201	1	2	3	ID	1	
1202	1	2	4	CODE	2	
1203	1	2	4	NAME	3	

Field Data

Field Identifier	Data Identifier	GUID	Comments
101	1	…	This bunch will be for one record
102	2	…	
103	3	…	
104	1/1/2013	…	
…			
1201	1	…	This bunch will be for one record
1202	SAMP1	…	
1203	Sample Code 1	…	
…			

- **Instead of pointer values for Data Identifier and create more illustration tables, I am putting the actual value here for simplicity**
- **As I stated before, this is not exhaustive, for instance it does not explain how primary keys will be maintained etc**

F_TABLE1 (Flat Table)

ORIGINAL1	OTHER9FIELDS	FIELD11	FIELD12	FIELD13	FIELD14
…		1	2	3	1/1/2013
…		11	22	33	12/31/2013

F_TABLE2 (Flat Table)

ID	CODE	NAME
1	SAMP1	Sample Code 1
2	SAMP2	Sample Code 2
…		

- If **TABLE1** is part of my static design as indicated by Static Table Flag, I can still extend it with changing situations (see the value of 11 in the field Order indicating there are 10 other fields in this table) – this will be especially useful if there is a lot of code written around **TABLE1** and there is a risk of breaking things if the table structure is altered
  - Augmenting a table design can be tricky in terms of how data will be updated for the new fields, but not something programming cannot handle
- **F_TABLE1** is what is exposed to reporting and other tools and by the time **F_TABLE1** is created any complexity arising out of metadata is eliminated. Any code written around non-static tables should be written in a mandated dynamic style
- Key thing to ensure (structure) changes do not have an impact is to always add new fields at the end – just like an "alter table" would work by ensuring that the attribute **Field Order** is controlled
  - While fields can be dropped seamlessly, it should be handled with caution, in case reports have been created using them
  - Due diligence as always would definitely be warranted

- There is no shame ☺ in flattening it, afterall the purpose is for the data to be usable, most likely by power-users for adhoc reporting or even to setup templates that regular users can quickly use
- By ensuring the order of fields is not altered, any new dynamic fields that make it in will (or should) not impact the downstream functionality and new tables/fields can be used only with some controlled effort

The flattened tables are pretty easy to use as can be seen, changing environments can be controlled and reporting/analytic solutions can be built based on these. In fact, if architected designs are the norm for BI tools and they can withstand some underlying structure changes in terms of new fields (which clearly will not be available within these tools till they are added), then this approach can be used even with analytic tools.

That is a lot of talk (or written words), so one can wonder if it is "proven"?

As I stated in the beginning, many ERP tools do use some approach to ensure they can customize their tools for different customers – however, from what I could gather over time, they tend to alter physical tables in place (like TABLE1 above) and dynamic SQL is what drives their UI code.

The manner in which I have built solutions using the above simplistic model in two separate situations:

- Ages ago, when I used to be a power programmer myself and the cost of a data modeling tool in my venerable hometown was almost-prohibitory – I used a rudimentary design along these lines (a little more constricted as I had a specific purpose) to mock the functionality I would get from a modeling tool to build a data dictionary application to establish data (naming, domain etc) standards and also used it to produce a lot of standard database code (stored procedures and triggers) to be used with the application code for manipulating the data
- More recently, I used a design closer to the above to provide functionality to a suite of software products being built around a common data set – this ofcourse was extend-the-core-design path and I did have to programmatically flatten the table as a POC

So, basically the answer is – yes, it is eminently doable!