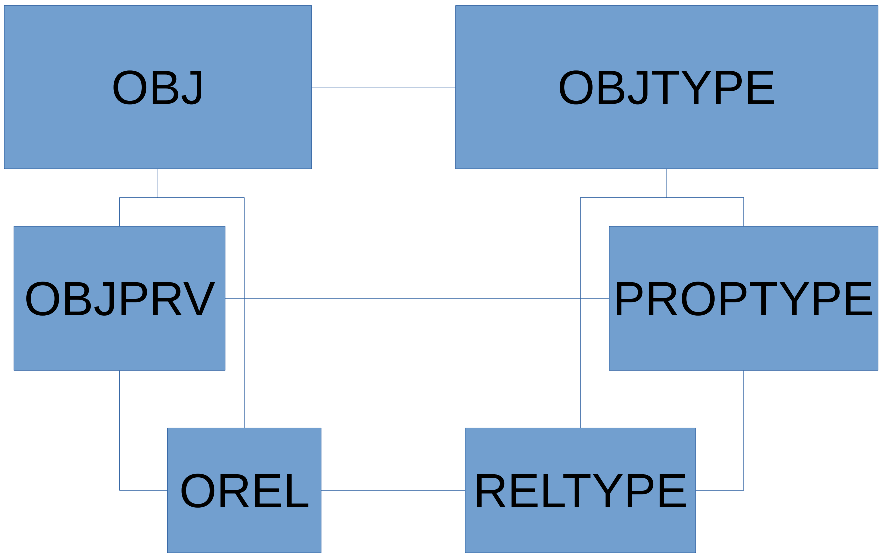
This database design description was invited by an explicit question on c.d.t. It is made up of a mixture of an unknown number of known antipatterns, at least including EAV and its usual nephews metadata-as-data and typeless attributes. See for yourself. Any mathematician will immediately fall for the exceptional beauty of the symmetrical structure.



OBJ has the usual (OBJID, OBJTYPE) attributes, but also additional attribute P1,P2,P3,...,P29,P30. Property 1, Property 2, Property 3, … , Property 29, Property 30. All 30 are VARCHAR(somelength), where the somelength was chosen by the architect at hand such that “it should suffice for everyone” (all properties nullable, all 30 properties indexed by their own index). I believe at one point they had a guideline for the modelers to distribute the use of properties as evenly as possible between P30 and P1, so that these 30 indexes had the best chance of growing to roughly the same size, and physical distribution of the relevant index entries was “optimal”. There was no rule that any given P property had to be designated to any single data type, so any P property column could contain a mixture of DATE, CHAR, TIMESTAMP, DECIMAL, … data types.

OBJTYPE predictably defines the types of object that can be recorded by the system. OBJTYPE also has properties P1, P2, P3, … , P29, P30. But in this table, the P properties contain the name of the property that is held, for objects of this type, in the P1, P2, P3, … columns of OBJ. These names must be properties that are named in PROPTYPE. (But obviously, with the predictable one-to-many relship between PROPTYPE and OBJTYPE, this rule is not enforced except by the “product configuration manager” application.)

So, for each OBJTYPE there are many PROPTYPES. Birthdate, ordernumber, account no, all “properties” in the model are defined here. There was also an indication here of whether history was to be kept of the values objects have had for this property (“Ah and if I change the flag to no history does it then erase the currently existing history ?” A : Of course not, that has to be done by programmer intervention. You shouldn't change such aspects of the model willy-nilly.) There was also an indication here of whether the property concerned a relationship to another object (see later). The inevitable “may be missing” flag for what in SQL become nullable columns, also wasn't forgotten. There was also a “formatter” indication that defined how a data value was to be formatted to a String so it could be recorded as a PROPerty value. I believe this indicator was the name of a Java class that could be dynamically instantiated and had a predefined interface method name that could be invoked. (Of course once defined, you wouldn't want to change these aspects of the data anymore either because otherwise a programmer would have to come in and do a conversion.)

Predictably, for each OBJ, and for each PROPTYPE that is kept for the OBJTYPE that OBJ is of, an entry in OBJPRV (object property value) records the value of the concerned PROPTYPE of the concerned OBJ. This table gets a little bit large and to avoid contention on it for querying, the values for the most “popular” properties are duplicated in one of the 30 OBJ level columns, according to configuration for the object type in OBJTYPE. If history must be kept, then DTST and DTEND columns determine the period of applicability for any such property value to an object. “Old” property value history itself might have been offloaded at regular times to an archive table, I don't know for sure anymore. If the PROPTYPE history flag says “no history”, then obviously DTST and DTEND will both be null. Just to make sure you could certainly never just ask DTST<=NOW() AND DTEND > NOW() and obtain all applicable property values. The PROPTYPE nullable flag determines possible absence of an object property value in the obvious way.

Onto the last two tables. Certain property types reflected the fact of an object type possibly having a relationship with some other object. Now if an object has a relationship with another object, then there are actually two pieces of information to be kept : the type of the relationship (“married to”, “legal representative of”, “shareholder of”, “[order] placed by [customer]”, …, and the piece of information that identifies the other object. The “type of relationship” is indicated by the property type, but they had a problem with the identifier of the other object. In the property value table, it would have to be encoded as a VARCHAR. And in the table where that other object was itself defined, of course it was a BIGINT. Which meant that if they recorded the identity of the other object in the VARCHAR column, then they couldn't enforce referential integrity between the two. And that was a major problem because they regarded integrity of the database as absolutely crucial. So they proceeded differently. For relationship type properties, the identity is recorded in a row in a child table, OREL. I honestly don't remember if the property value in OBJPRV was also made nullable and effectively left null in these cases (wouldn't have surprised me as it obviously keeps the indexes strictly as small as effectively needed, you know), or whether the VARCHAR formats were effectively filled in and just never used. OREL had the linked-to object's identity in the right format for enforcing RI to OBJ.

RELTYPE, to conclude, defined the obvious meta-level stuff for relationship types, that is, the type of object that linked-to objects must be of for any given type of relationship.

One of the memorable moments was when one of the fierce defenders of this model was explaining to me how I had to walk the graph of linked objects, and that it was very important to always keep an eye on whether I was walking from left to right or from right to left, because otherwise I'd probably just get lost. I've had to work with this model to implement a modestly complex business model of billing. After two months of struggling and walking graphs in any direction I could imagine, the task was maybe 10% complete and I was feeling rather seasick. At that time, I was fortunate enough to be presented the occasion to leave.

Oh yes, before I forget. Of course each table had its own surrogate ID and the debugging interface they had for debugging the database (!!!!!!!!!!!!!) was based exclusively on that. After all, programmers are better at dealing with objects and their id's than they are at dealing with business-level information, aren't they ?