**The Illusive 1–2–1 (1-to-1) Relationship**

**& The Data Factory Model**

As Data Base Administrators we deal with the concepts of Entities and the Relationships between them – which in a normalized database are represented by Tables and the Foreign Keys between them.

Relationships are known to be of 3 types:

1. One-to-one
2. One-to-many
3. Many-to-many

Most normalized databases are arranged by using the later 2 relationship kinds. (We also know that in order to implement a Many-to-many relationship between 2 entities (tables) a third intermediate entity is needed to form a two one-to-many relationships architecture).

In this article I'm going to focus on the One-to-One relationship type which is the rarest used and somewhat overlooked, hence referred to herein as the 'Illusive' 1-2-1.

What might be a DBA's first intuitive thought when confronted with the 1-2-1 relationship concept can be expressed as:

*"If two entities form a one-to-one relationship between them then they are actually the same so why separate them in the first place?"*

Well, that's true for the most part. Still, there are 2 main reasons to structure a database architecture using 1-2-1 relationships:

1. Separating "sensitive" data from "ordinary" data.
2. Creating an "inheritance like" design

The first situation is quit self explanatory. In order to secure the user's sensitive data - such as her credit card number or medical status - a DBA might want to physically separate it from the "ordinary" demographics data (and optionally encrypt and/or use restrictive permissions and/or other means of protection).

The second scenario is what we're concerned with in this article.

**The database side**

In order to delve into the situation let's first demonstrate an example to make things clearer.

Suppose you're working as a DBA for an organization that was commissioned by the city's municipality to build an application to manage the data on the city's transportation system.

A design meeting was held in which it was decided to build the database layer to support the application in an inheritance like architecture that looked like this:



Figure 1 – An inheritance like architecture

DMLing the data

In order to select rows from a 'concrete' table we create a view that joins the table together with its 'ancestors':

CREATE VIEW [dbo].[V\_Buses]

AS

SELECT T\_Buses.\*,

T\_PublicV.\*,

T\_Vehicles.\*

FROM dbo.T\_Buses

INNER JOIN dbo.T\_PublicTransportationVehicles AS T\_PublicV

ON T\_Buses.BusID = T\_PublicV.PublicTransportationVehicleID

INNER JOIN T\_Vehicles

ON T\_PublicV.PublicTransportationVehicleID = T\_Vehicles.VehicleID

Script 1 – A view joins the 'concrete' table with it's ancestor 'abstract' tables

Since the view selects from a join on multiple tables trying to do an insert, update or delete will generate an error :

View or function 'dbo.V\_Buses' is not updatable because the modification affects multiple base tables.

So in order to take DML operations we need to create an 'INSTEAD OF' trigger on the view and use the INSERTED and DELETED virtual table's values to manually perform the operation:

**The Application Layer Terminology**

This chapter is for those who wish to extend their broader comprehension of the General Data Factory N-Tier Model by getting familiarized with the concepts and terminology used in departments of the factory other then the database layer :

* The User Interface in the far end (aka 'the Client'/'the Shop')
* The Application layer (aka 'the Server'/'the Business Rules layer')
* The Packaging and Shipment layer (intermediate between the Server and the Client – think of delivery trucks - optional)
* The Data Access layer (intermediate between the Server and the DB – think of forklifts - optional)

Application layer developers use classes to describe the architecture of the system being built.

Classes act somewhat like templates in that they are used to generate object instances out of them. For example, the Cake class is instantiated in C# by the use of the 'new' keyword:

Cake objCake = new Cake ();

Classes have Properties (data members), Methods (Ingoing function calls) and Events (Outgoing function calls).

The ordinary one-to-many relationship in the database is resembled in the application layer as a property of the 'one' side typed as a collection of class instances of the 'many' side (in C#):

Public Class Cake

{…

Private CakeIngrediantsCollection m\_colCakeIngrediants = new CakeIngrediantsCollection (this.CakeID);

…

Public property get CakeIngrediants

{

Return m\_colCakeIngrediants;

}

}

Classes can also have hierarchies, that is, a class can act as a layer/node in a structure in which it inherits from a base class (known as "Super Class") and have decedents (known as "sub class").

A base class can be declared 'abstract' which means no object instances can be generated from it and its sole purpose is to capture and encapsulate a set of properties and functions for sub classes to inherit.

If the base class only declares it's public functions and properties and does not contains they're definitions it is said to be a 'pure abstract' class.

The derived sub class can also be declared 'abstract'. If the 'abstract' keyword is omitted from a derived sub class it is said to be a concrete class and has to supply the definitions for any ancestor pure abstract class's methods and properties.

A concrete class actually creates a 1-2-1 relationship with all of its ancestors up the inheritance chain.