

Design Tip #127 Creating and Managing Mini-Dimensions

By Warren Thornthwaite

I wrote a Design Tip last year on creating junk dimensions; I've decided to extend that article into a series on implementing common ETL design patterns.

This Design Tip describes how to create and manage mini-dimensions. Recall that a mini-dimension is a subset of attributes from a large dimension that tend to change rapidly, causing the dimension to grow excessively if changes were tracked using the Type 2 technique. By extracting unique combinations of these attribute values into a separate dimension, and joining this new mini-dimension directly to the fact table, the combination of attributes that were in place when the fact occurred are tied directly to the fact record. (For more information about mini-dimensions, see [Slowly Changing Dimensions Are Not Always as Easy as 1, 2, 3](#) at Intelligent Enterprise and [Design Tip # 53 – Dimension Embellishments](#).)

Creating the Initial Mini-Dimension

Once you identify the attributes you want to remove from the base dimension, the initial mini-dimension build is easily done using the brute force method in the relational database. Simply create a new table with a surrogate key column, and populate the table using a SELECT DISTINCT of the columns from the base dimension along with an IDENTITY field or SEQUENCE to create the surrogate key. For example, if you want to pull a set of demographic attributes out of the customer dimension, the following SQL will do the trick:

```
INSERT INTO Dim_Demographics
SELECT DISTINCT col 1, col2, ...
FROM Stage_Customer
```

This may sound inefficient, but today's database engines are pretty fast at this kind of query. Selecting an eight column mini-dimension with over 36,000 rows from a 26 column customer dimension with 1.1 million rows and no indexes took 15 seconds on a virtual machine running on my four year old laptop.

Once you have the Dim_Demographics table in place, you may want to add its surrogate key back into the customer dimension as a Type 1 attribute to allow users to count customers based on their current mini-dimension values and report historical facts based on the current values. In this case, Dim_Demographics acts as an outrigger table on Dim_Customer. Again, the brute force method is easiest. You can join the Stage_Customer table which still contains the source attributes to Dim_Demographics on all the attributes that make up Dim_Demographics. This multi-join is obviously inefficient, but again, not as bad as it seems. Joining the same million plus row customer table to the 36 thousand row demographics table on all eight columns took 1 minute, 49 seconds on the virtual machine.

Once all the dimension work is done, you will need to add the mini-dimension key into the fact row key lookup process. The easy way to do this during the daily incremental load is to return both the Dim_Customer surrogate key and the Dim_Demographic surrogate key as part of the customer business key lookup process.

Ongoing Mini-Dimension Maintenance

Ongoing management of the dimension is a two-step process: first you have to add new rows to the Dim_Demographics table for any new values or combinations of values that show up in the incoming

Stage_Customer table. A simple brute force method leverages SQL's set based engine and the EXCEPT, or MINUS, function.

```
INSERT INTO Dim_Demographics
SELECT DISTINCT Payment_Type, Server_Group, Status_Type, Cancelled_Reason,
                Box_Type, Manufacturer, Box_Type_Descr, Box_Group_Descr
FROM BigCustomer
EXCEPT
SELECT Payment_Type, Server_Group, Status_Type, Cancelled_Reason,
        Box_Type, Manufacturer, Box_Type_Descr, Box_Group_Descr
FROM Dim_Demographics
```

This should process very quickly because the engine simply scans the two tables and hash match the results. It took 7 seconds against the full source customer dimension in my sample data, and should be much faster with only the incremental data.

Next, once all the attribute combinations are in place, you can add their surrogate keys to the incoming incremental rows. The same brute force, multi-column join method used to do the initial lookup will work here. Again, it should be faster because the incremental set is much smaller.

By moving the Type 2 historical tracking into the fact table, you only connect a customer to their historical attribute values through the fact table. This may not capture a complete history of changes if customer attributes can change without an associated fact event. You may want to create a separate table to track these changes over time; this is essentially a factless fact table that would contain the customer, Dim_Demographics mini-dimension, change event date, and change expiration date keys. You can apply the same techniques we described in [Design Tip #107](#) on using the SQL MERGE statement for slowly changing dimension processing to manage this table.

NOTE: The original version of this design tip used a MERGE statement to identify new rows for the mini-dimension in the incremental processing step. The use of a MERGE statement in this case has the potential to allow duplicate rows in the mini-dimension. The EXCEPT statement in this updated version does not allow duplicates, and it works faster.