Paper 3108-2019

Reduce Table Size and Always Normalize! (Right? Maybe Not!)

Darryl Prebble, Prebble Consulting Inc.

ABSTRACT

For years it's been common knowledge that you should normalize your data to maximize your storage space. Of course, this involves creating and maintaining reference tables. What if the tables become out of date? Am I just causing more problems by introducing complexity like this?

In this paper I illustrate a method of automating the normalization of tables to take the work and maintenance out of the equation. I also introduce an alternative to normalization available in Teradata and how it stacks up.

INTRODUCTION

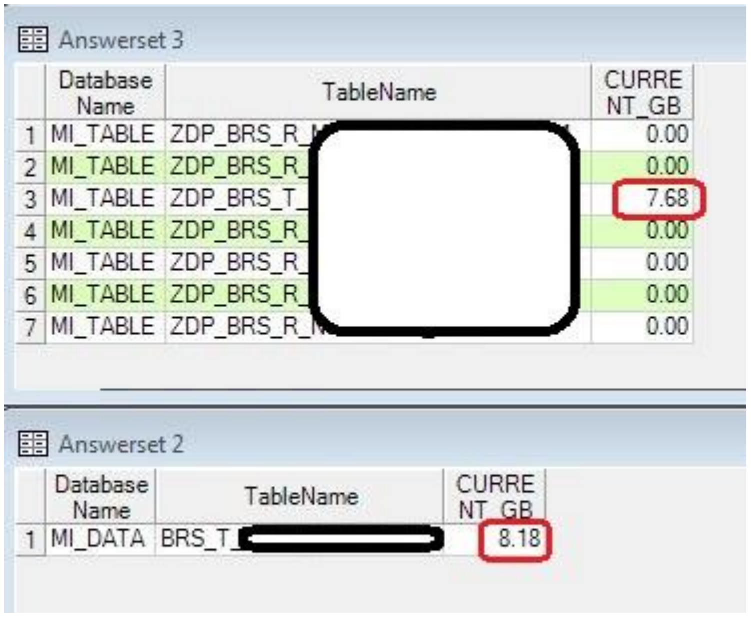
The idea of normalizing your data is a simple one – remove long, repeating, descriptive text fields from a table (taking up lots of space) and replace them with an integer ID. This helps you store the same information using far less storage space.

The practice comes with an admin cost however, in that you have to maintain the supporting reference tables. Often this is done in a manual way and requires ongoing support on the project, which is not ideal.

This paper examines how to automate the process and ends with a small twist – when is Best Practice, not Best Practice?!

WHY?

A simple question, and a good place to start. If you don't already know about data normalization, why should you take the time to learn what it is and why should you implement it? A simple question, with a simple answer shown in Figure 1 below.



Answerset 3

	Database Name	TableName	CURRENT_GB
1	MI_TABLE	ZDP_BRS_R_[redacted]	0.00
2	MI_TABLE	ZDP_BRS_R_[redacted]	0.00
3	MI_TABLE	ZDP_BRS_T_[redacted]	7.68
4	MI_TABLE	ZDP_BRS_R_[redacted]	0.00
5	MI_TABLE	ZDP_BRS_R_[redacted]	0.00
6	MI_TABLE	ZDP_BRS_R_[redacted]	0.00
7	MI_TABLE	ZDP_BRS_R_[redacted]	0.00

Answerset 2

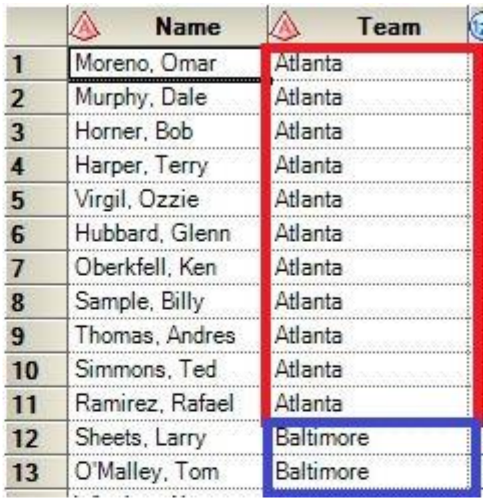
	Database Name	TableName	CURRENT_GB
1	MI_DATA	BRS_T_[redacted]	8.18

Figure 1. Size of table before and after normalization (table names redacted)

In Figure 1, Answerset2 is the "before", while Answerset3 is the "after". An easy savings of over 6%! Doesn't sound like much, but when implemented on a large scale it's a huge savings.

WHAT IS NORMALIZATION?

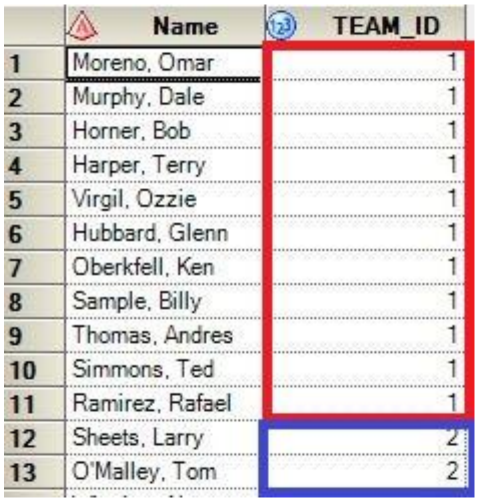
What is the issue that needs solving here? See Figure 2 for an example of the problem.



	Name	Team
1	Moreno, Omar	Atlanta
2	Murphy, Dale	Atlanta
3	Horner, Bob	Atlanta
4	Harper, Terry	Atlanta
5	Virgil, Ozzie	Atlanta
6	Hubbard, Glenn	Atlanta
7	Oberkfell, Ken	Atlanta
8	Sample, Billy	Atlanta
9	Thomas, Andres	Atlanta
10	Simmons, Ted	Atlanta
11	Ramirez, Rafael	Atlanta
12	Sheets, Larry	Baltimore
13	O'Malley, Tom	Baltimore

Figure 2. Sample data from SASHELP.BASEBALL

As shown in red and blue, we have two examples of repeating, descriptive text values that over a large enough dataset will utilize a lot of storage space needlessly. As shown in Figure 3 below, this is what it looks like once normalized.

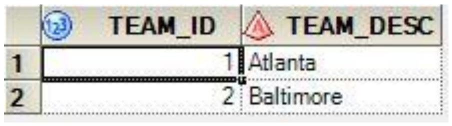


	Name	TEAM_ID
1	Moreno, Omar	1
2	Murphy, Dale	1
3	Horner, Bob	1
4	Harper, Terry	1
5	Virgil, Ozzie	1
6	Hubbard, Glenn	1
7	Oberkfell, Ken	1
8	Sample, Billy	1
9	Thomas, Andres	1
10	Simmons, Ted	1
11	Ramirez, Rafael	1
12	Sheets, Larry	2
13	O'Malley, Tom	2

Figure 3. Sample of data from SASHELP.BASEBALL after normalization

Instead of storing all of those repeating text values, we now only have to store a number. The benefit of this technique increases as the size of the text we're replacing increases as well.

The last step required here is simply to create a reference table as shown in Figure 4.



	TEAM_ID	TEAM_DESC
1	1	Atlanta
2	2	Baltimore

Figure 4. REF_TEAM table created to support normalization of SASHELP.BASEBALL

A simple join on TEAM_ID will retrieve the TEAM_DESC and you end up with the same end-result with all of the storage savings to make your friendly neighborhood DBA happy.

DRAWBACKS?

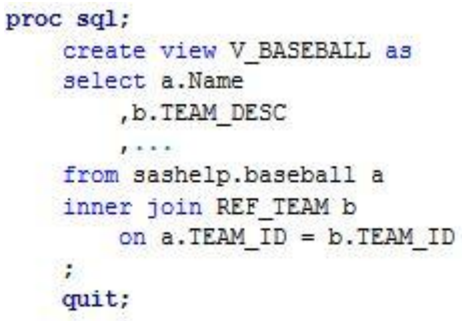
Few things in life are free of course, and normalization is no exception. There are two costs to consider when implementing a solution such as this.

PUSHING WORK DOWN TO THE END-USER

Where previously the end-user could simply query the table and see all the information they wanted, they may not be happy seeing the TEAM_ID in place of the TEAM_DESC. This means that they will have to do a join to REF_TEAM on TEAM_ID to get the full information.

This in itself isn't the end of the world, but what happens as the solution scales upwards with large tables and your normalization is encompassing many fields, not just one?

Luckily, the solution here is simple – create a view that the end-user will query instead of pointing them to the actual table. In the view, you will do all of the joins ahead of time so that your work behind the scenes is completely invisible to the end-user. If done correctly, they'll have no idea you worked so hard to keep your DBAs happy. See Figure 5 below for an example of how to create this type of view.



```
proc sql;
    create view V_BASEBALL as
    select a.Name
        ,b.TEAM_DESC
        ,...
    from sashelp.baseball a
    inner join REF_TEAM b
        on a.TEAM_ID = b.TEAM_ID
    ;
    quit;
```

Figure 5. View to hide your tracks from end-users

When a user queries V_BASEBALL, they'll have no idea that the data has been normalized behind the scenes. Is there a performance hit in the background now that joins have to take place? Yes, but if keys and indexes are properly defined, there shouldn't be an issue for any but the largest of tables.

WHO MAINTAINS THE REFERENCE TABLES?

Often, reference table values are defined by a Business Analyst during the implementation of a project. Perhaps the tables are passed to the development team as an Excel or CSV file to be ingested into the database. This works great for day 1 deployment, but what happens going forward when new values are introduced to the data?

By normalizing the data you've made it critical that new data values are communicated to the development team ahead of time, and likely require on-going developer and BA support.

Or does it…

AUTOMATED SUPPORT OF REFERENCE TABLES

In order to automate the whole thing so that you don't have to continue supporting the project for rest of your life, we can break it all down into three tasks.

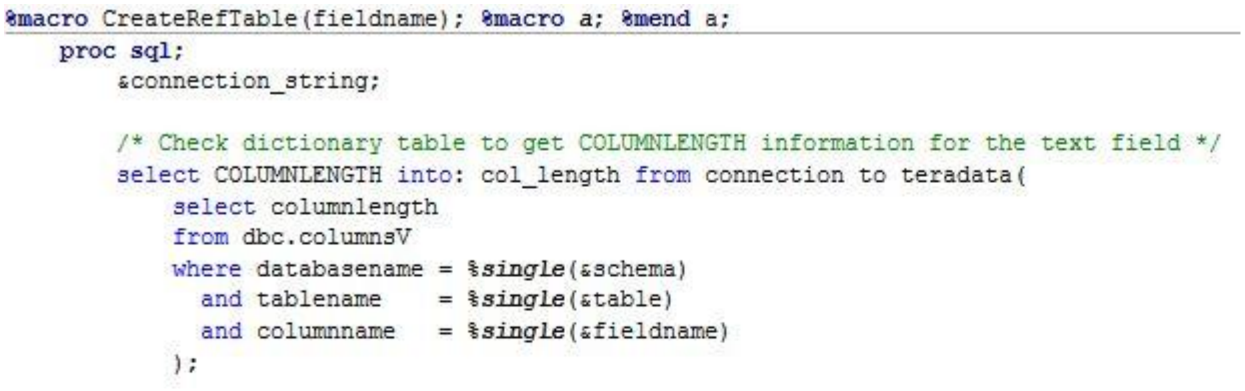
I break down these tasks below using passthrough Teradata SQL, as normalization gets it's best results from large database appliances working with large amounts of data. However, these modular concepts are simple to port to SAS® datasets as well (I started this design with SAS datasets), or any other DB appliance.

1. Creation of reference tables
2. Population of reference tables (both initial load and ongoing)
3. Normalization of main table
    a. Addition of ID fields
    b. Population of ID fields
    c. Removal of descriptive text fields

CREATION OF REFERENCE TABLES

In the process of replacing text descriptions with IDs *from* a reference table, first you must create the reference table and populate it.

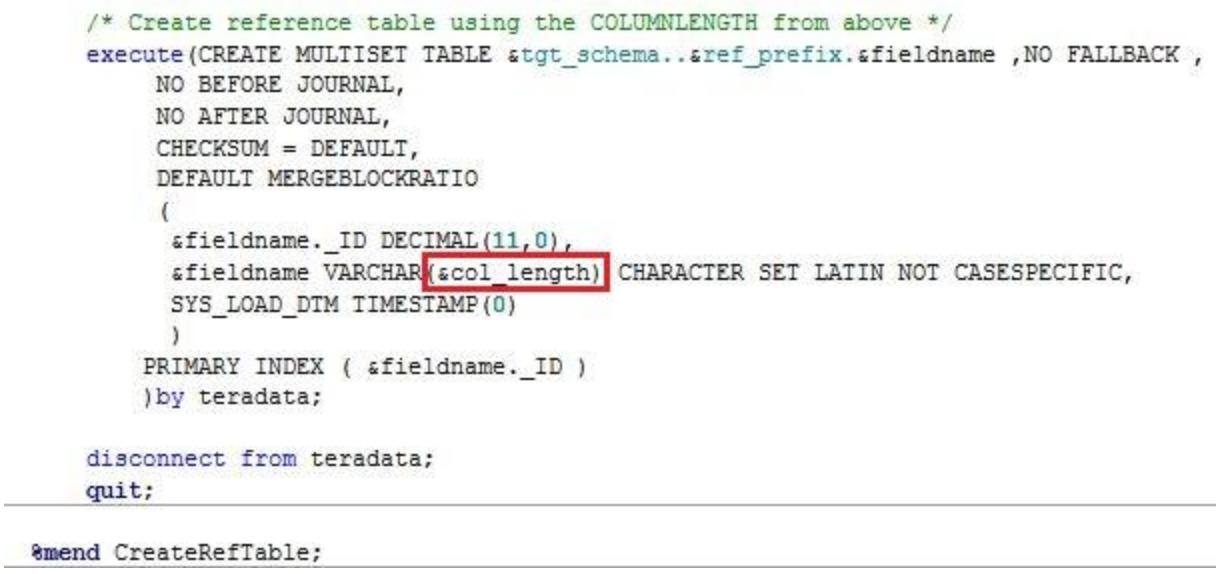
In order to figure out how to create the reference table we turn to the dictionary tables to find out how big the text field needs to be. Note that the code below in Figure 6 has been macro-ized so that it can be called easily to create as many reference tables as needed.



```
%macro CreateRefTable(fieldname); %macro a; %mend a;
    proc sql;
        &connection_string;

        /* Check dictionary table to get COLUMNLENGTH information for the text field */
        select COLUMNLENGTH into: col_length from connection to teradata(
            select columnlength
            from dbc.columnsV
            where databasename = %single(&schema)
              and tablename    = %single(&table)
              and columnname   = %single(&fieldname)
            );
```

Figure 6. Use the dictionary tables to get metadata information

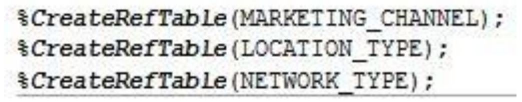
Once you know how big of a text field you need to make, you can go ahead and create the empty table as seen here in Figure 7, using the macro variable *col_length* created above.



```
/* Create reference table using the COLUMNLENGTH from above */
execute(CREATE MULTISET TABLE &tgt_schema..&ref_prefix.&fieldname ,NO FALLBACK ,
     NO BEFORE JOURNAL,
     NO AFTER JOURNAL,
     CHECKSUM = DEFAULT,
     DEFAULT MERGEBLOCKRATIO
     (
      &fieldname._ID DECIMAL(11,0),
      &fieldname VARCHAR(&col_length) CHARACTER SET LATIN NOT CASESPECIFIC,
      SYS_LOAD_DTM TIMESTAMP(0)
      )
    PRIMARY INDEX ( &fieldname._ID )
    )by teradata;

disconnect from teradata;
quit;

%mend CreateRefTable;
```

Figure 7. Create the empty reference table based on metadata information

Creating multiple reference tables based on multiple fields is now easy with this macro, as seen in Figure 8.



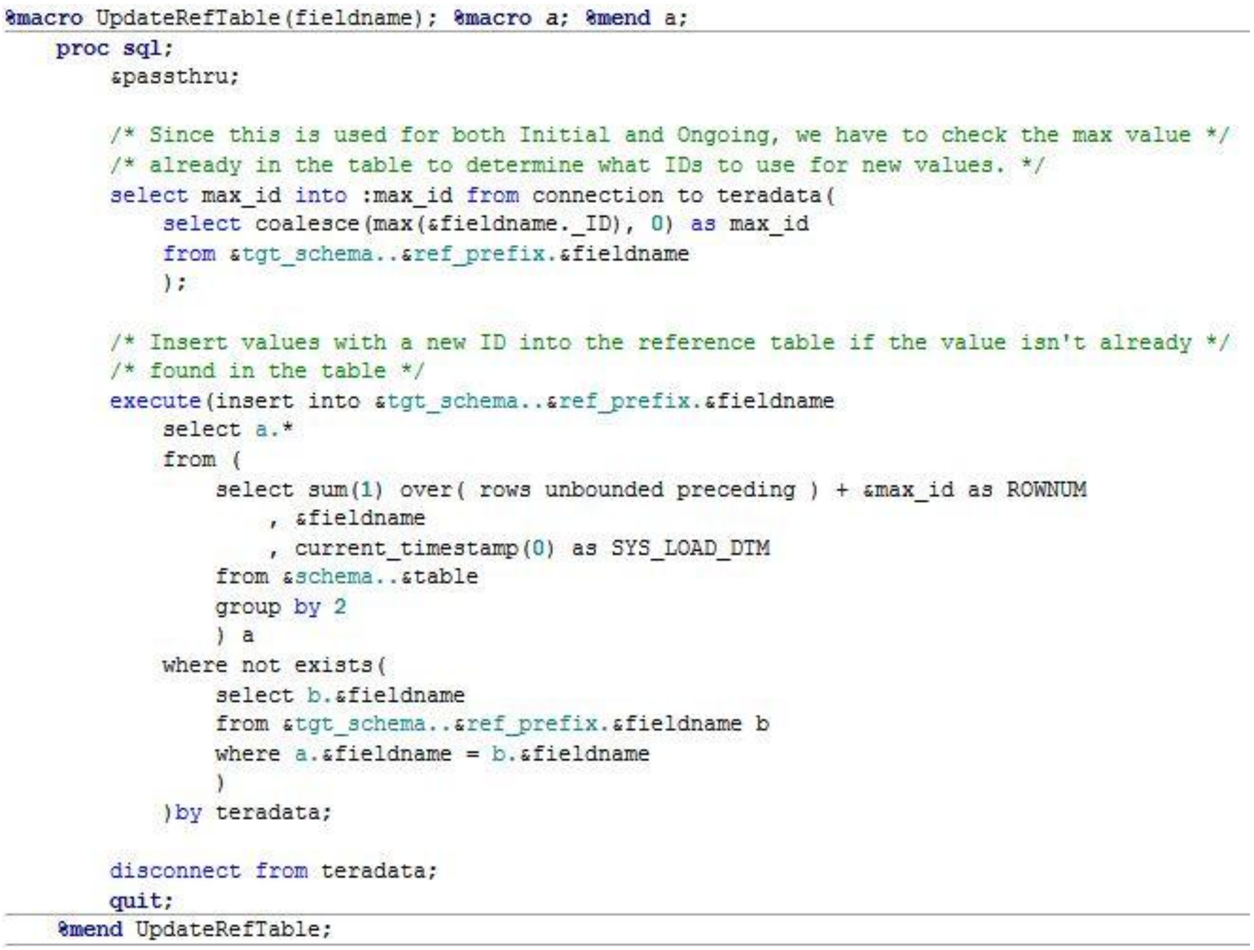
```
%CreateRefTable(MARKETING_CHANNEL);
%CreateRefTable(LOCATION_TYPE);
%CreateRefTable(NETWORK_TYPE);
```

Figure 8. Creating multiple reference tables with macro call

Further to this solution is the ability to programmatically figure out what fields should be normalized based on the field's cardinality, then having this macro called within. In other words, automating the entire process. I won't give away all of my secrets, so I leave this to the user to solve themselves.

POPULATION OF REFERENCE TABLES

Population of the reference tables for both Initial and Ongoing loads is the same logic. This is helpful, because as I've laid out earlier in the Drawbacks section, ongoing support requirements is an issue to be taken into consideration when normalizing your data. See Figure 9 below.

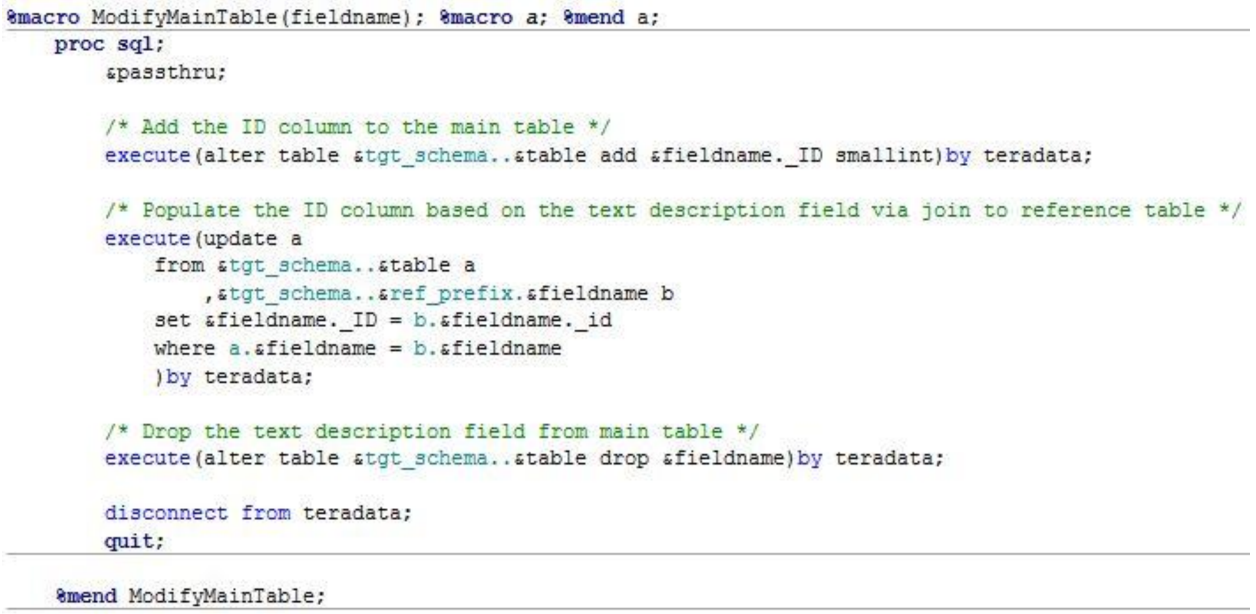


```
%macro UpdateRefTable(fieldname); %macro a; %mend a;
    proc sql;
        &passthru;

        /* Since this is used for both Initial and Ongoing, we have to check the max value */
        /* already in the table to determine what IDs to use for new values. */
        select max_id into :max_id from connection to teradata(
            select coalesce(max(&fieldname._ID), 0) as max_id
            from &tgt_schema..&ref_prefix.&fieldname
            );

        /* Insert values with a new ID into the reference table if the value isn't already */
        /* found in the table */
        execute(insert into &tgt_schema..&ref_prefix.&fieldname
            select a.*
            from (
                select sum(1) over( rows unbounded preceding ) + &max_id as ROWNUM
                    , &fieldname
                    , current_timestamp(0) as SYS_LOAD_DTM
                from &schema..&table
                group by 2
                ) a
            where not exists(
                select b.&fieldname
                from &tgt_schema..&ref_prefix.&fieldname b
                where a.&fieldname = b.&fieldname
                )
            )by teradata;

        disconnect from teradata;
        quit;
    %mend UpdateRefTable;
```

Figure 9. Population of reference tables with macro call

MODIFY MAIN TABLE

With the reference tables created, the last step is to replace the values in the main table with the representative IDs. This is done by adding ID fields, populating them, then dropping the text description fields, as seen in Figure 10 below.



```
%macro ModifyMainTable(fieldname); %macro a; %mend a;
    proc sql;
        &passthru;

        /* Add the ID column to the main table */
        execute(alter table &tgt_schema..&table add &fieldname._ID smallint)by teradata;

        /* Populate the ID column based on the text description field via join to reference table */
        execute(update a
            from &tgt_schema..&table a
                ,&tgt_schema..&ref_prefix.&fieldname b
            set &fieldname._ID = b.&fieldname._id
            where a.&fieldname = b.&fieldname
            )by teradata;

        /* Drop the text description field from main table */
        execute(alter table &tgt_schema..&table drop &fieldname)by teradata;

        disconnect from teradata;
        quit;

    %mend ModifyMainTable;
```

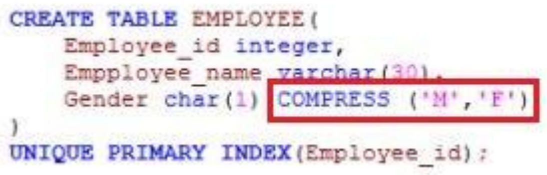
Figure 10. Modification of main table

This finalizes the normalization process, as you have replaced large, repeating text descriptions with integer IDs instead. A simple join allows you to reintegrate the text descriptions at run-time in a view with nobody the wiser! This is why data normalization is considered Best Practice for your database environment.

AN ALTERNATIVE

As alluded to in the title of this paper, there is an alternative to the Best Practice in this case! In Teradata you can "fake" normalization in a way that achieves very similar results called Multi Value Compression (MVC).

To use the basis of understanding that has been developed above, this method replaces the text values with IDs just as normalization does, but stores the reference table (ie. Mappings) in the table header. See below in Figure 11 for an example of MVC.



```
CREATE TABLE EMPLOYEE(
    Employee_id integer,
    Empployee_name varchar(30),
    Gender char(1) COMPRESS ('M','F')
)
UNIQUE PRIMARY INDEX(Employee_id);
```

Figure 11. Teradata MVC

The advantage here is clear – all of the advantages of normalization, without any of the upkeep. You do not have to create reference tables. You do not have to create views to join all the data back together again. You do not have to maintain the reference tables going forward!

So what's the catch? The catch is maintenance. Going forward when there are new values introduced in your data, those values will not be compressed. The data values in MVC need to be baked right into the table DDL itself as shown in Figure 11. So while your table will not break as new values are introduced to the data going forward, the storage savings will suffer. Best Practice for getting around this is archive the table, drop it and then recreate with the new values listed in the table DDL. Then you insert the data into the new table from the backup. Whether this solution works for you in your environment is something you'll have to work out with your DBAs and Governance team, but the benefits are clear especially if you do not anticipate large volumes of new data values coming through.

The normalization method outlined above takes care of maintenance going forward and is hands-off once implemented.

CONCLUSION

The benefit of normalizing your data is clear, especially in today's age of massive data volumes. The methods as outlined in this paper can help minimize overhead and support, removing any counter-arguments to implementing this technique in your datamart.

Also, if the situation allows, Teradata MVC is a wonderful alternative though you must be aware of it's limitations. As a rule, it works best when you anticipate no new values to be introduced to your data (ie. Provinces or States).

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Darryl Prebble
Prebble Consulting Inc.
dprebble@gmail.com