Data Warehouse Design Tips

A relentless collection of more than 130 tips for designing and implementing Data Warehouse projects successfully.

13 March, 2011

Data Warehouse Design Tips

The Kimball Group delivers practical techniques that are:

- Vendor independent
- Reliable,
- real-world guidance, not theory

Written by Kimball Group members, the only practitioners certified by Ralph, these Data Warehouse design tips and best practices can be used for your Data Warehouse projects.

TABLE OF CONTENTS

Design Tip #105	Snowflakes, Outriggers, and Bridges
Design Tip #104	Upgrading your BI Architecture
Design Tip #103	Staffing the Dimensional Modeling Team
Design Tip #102	Server Configuration Considerations
Design Tip #101	Slowly Changing Vocabulary
Design Tip #100	Keep Your Keys Simple
Design Tip #99	Staging Areas and ETL Tools
Design Tip #98	Focus on Data Stewardship

Design Tips 2007

Design Tip #97	Modeling Data as Both a Fact and Dimension Attribute
Design Tip #96	Think Like A Software Development Manager
Design Tip #95	Patterns to Avoid when Modeling Header/Line Item Transactions
Design Tip #94	Building Custom Tools for the DW/BI System
Design Tip #93	Transactions Create Time Spans
Design Tip #92	Dimension Manager and Fact Provider
Design Tip #91	Marketing the DW/BI System
Design Tip #90	Slowly Changing Entities
Design Tip #89	The Real Time Triage
Design Tip #88	Dashboards Done Right
Design Tip #87	Combining SCD Techniques Having It Both Ways

Design Tips 2006

Design Tip #86	Reference Dimensions for Infrequently-Accessed Degenerates
Design Tip #85	Smart Date Keys to Partition Fact Tables
Design Tip #84	Readers' Suggestions on Fact Table Surrogate Keys
Design Tip #83	Resist Abstract Generic Dimensions
Design Tip #82	Pivoting the Fact Table with a Fact Dimension
Design Tip #81	Fact Table Surrogate Keys
Design Tip #80	Dimension Row Change Reason Attributes
Design Tip #79	Dangerously Large Dimension Tables
Design Tip #78	Late Arriving Dimension Rows
Design Tip #77	Warning: Summary Data may be Hazardous
Design Tip #76	Advantages of a 64-bit Server
Design Tip #75	Creating the Metadata Strategy

Design Tips 2005

Design Tip #74	Compliance-Enabled Data Warehouses
Design Tip #73	Relating to Agile Methodologies
Design Tip #72	Business Process Decoder Ring
Design Tip #71	Naming Conventions
Design Tip #70	Architecting Data for MS SQL Server 2005
Design Tip #69	Identifying Business Processes
Design Tip #68	Simple Drill-Across in SQL
Design Tip #67	Maintaining Back Pointers to Operational Sources
Design Tip #66	Implementation Analysis Paralysis
Design Tip #65	Document the ETL System

KIMBALL GROUP KIMBALL UNIVERSITY

www.kimballgroup.com Number 130, December 1, 2010

Design Tip #130 Accumulating Snapshots for Complex Workflows

By Margy Ross

As Ralph described in *Design Tip #37* *Modeling a Pipeline with an Accumulating Snapshot*, accumulating snapshots are one of the three fundamental types of fact tables. We often state that accumulating snapshot fact tables are appropriate for predictable workflows with well-established milestones. They typically have five to ten key milestone dates representing the workflow/pipeline start, completion, and the key event dates in between.

Our students and clients sometimes ask for guidance about monitoring cycle performance for a less predictable workflow process. These more complex workflows have a definite start and end date, but the milestones in between are often numerous and less stable. Some occurrences may skip over some intermediate milestones, but there's no reliable pattern.

Be forewarned that the design for tackling these less predictable workflows is not for the faint of heart! The first task is to identify the key dates that will link to role-playing date dimensions. These dates represent key milestones; the start and end dates for the process would certainly qualify. In addition, you'd want to consider other commonly-occurring, critical milestones. These dates (and their associated dimensions) will be used for report and analyses filtering. For example, if you want to see cycle activity for all workflows where a milestone date fell in a given work week, calendar month, fiscal period, or other standard date dimension attribute, then it should be identified as a key date with a corresponding date dimension table. The same holds true if you want to create a time series trend based on the milestone date. While selecting specific milestones as the critical ones in a complex process may be challenging for IT, business users can typically identify these key milestones fairly readily. But they're often interested in a slew of additional lags which is where things get thorny.

For example, let's assume there are six critical milestone dates, plus an additional 20 less critical event dates associated with a given process/workflow. If we labeled each of these dates alphabetically, you could imagine analysts being interested in any of the following date lags:

A-to-B, A-to-C, …, A-to-Z (total of 25 possible lags from event A)
B-to-C, …, B-to-Z (total of 24 possible lags from event B)
C-to-D, …, C-to-Z (total of 23 possible lags from event C)
…
Y-to-Z

Using this example, there would be 325 (25+24+23+…+1) possible lag calculations between milestone A and milestone Z. That's an unrealistic number of facts for a single fact table! Instead of physically storing all 325 date lags, you could get away with just storing 25 of them, and then calculate the others. Since every cycle occurrence starts by passing through milestone A (workflow begin date), you could store all 25 lags from the anchor event A, then calculate the other 300 variations.

Let's take a simpler example with actual dates to work through the calculations:

Event A (process begin date) - Occurred on November 1
Event B - Occurred on November 2
Event C - Occurred on November 5
Event D - Occurred on November 11
Event E - Didn't happen
Event F (process end date) - Occurred on November 16

In the corresponding accumulating snapshot fact table row for this example, you'd physically store the following facts and their values:

A-to-B days lag - 1
A-to-C days lag - 4
A-to-D days lag - 10
A-to-E days lag - null
A-to-F days lag - 15

To calculate the days lag from B-to-C, you'd take the A-to-C lag value (4) and subtract the A-to-B lag value (1) to arrive at 3 days. To calculate the days lag from C-to-F, you'd take the A-to-F value (15) and subtract the A-to-C value (4) to arrive at 11 days. Things get a little trickier when an event doesn't occur, like E in our example. When there's a null involved in the calculation, like the lag from B-to-E or E-to-F, the result needs to also be null because one of the events never happened.

This technique works even if the interim dates are not in sequential order. In our example, let's assume the dates for events C and D were swapped: event C occurred on November 11 and D occurred on November 5. In this case, the A-to-C days lag is 10 and the A-to-D lag is 4. To calculate the C-to-D lag, you'd take the A-to-D lag (4) and subtract the A-to-C lag (10) to arrive at a -6 days.

In our simplified example, storing all the possible lags would have resulted in 15 total facts (5 lags from event A, plus 4 lags from event B, plus 3 lags from event C, plus 2 lags from event D, plus 1 lag from event E). That's not an unreasonable number of facts to just physically store. This tip makes more sense when there are dozens of potential event milestones in a cycle. Of course, you'd want to hide the complexity of these lag calculations under the covers from your users, like in a view declaration.

As I warned earlier, this design pattern is not simplistic; however, it's a viable approach for addressing a really tricky problem.

www.kimballgroup.com Number 129, November 3, 2010

Design Tip #129 Are IT Procedures Beneficial to DW/BI Projects?

By Joy Mundy

Back when the world was young and data warehousing was new, projects were a lot more fun. Kimball Group consultants (before there was a Kimball Group) were usually called in by the business users, and we'd help them design and build a system largely outside the aegis of corporate IT. In the intervening years -- decades! -- data warehousing has become a mainstream component of most IT organizations. For the most part, this is a good thing: the rigor that formal IT management brings to the DW makes our systems more reliable, maintainable, and performant. No one likes the idea of a BI server sitting under a business user's desk.

However, IT infrastructure is not always helpful to the DW/BI project. Sometimes it gets in the way, or is actively obstructionist. No doubt every little sub-specialty of information technology clamors that it is somehow different or special, but in the case of DW/BI, it's really true.

Specifications. The classic waterfall methodology subscribed to by many IT organizations has you develop a painfully detailed specification that's formalized, agreed to by the business and IT, and then turned over to the developers for implementation. Changes to the specification are tightly controlled, and are the exception rather than the rule.

If you try to apply a waterfall methodology to a DW/BI project, the best you'll end up with is a reporting system. The only things you can specify in sufficient detail are standard reports, so that's what you get: a system to deliver those standard reports. Many specs include a demand to support ad hoc analysis, and sometimes include examples of specific analyses the users would like to be able to do. But within the waterfall methodology it's impossible to clearly specify the boundaries and requirements of ad hoc analyses. So the project team "meets" this requirement by plopping an ad hoc query tool in front of the database.

It's really frustrating for the business users who are asked to write or approve the specification. They know the spec doesn't capture the richness of what they want, but they don't know how to communicate their needs to IT. I was recently in a meeting with a disgruntled business user who glared at the DW manager and said "Just provide me with a system that captures all the relationships in the data." If only that one sentence were sufficient for the poor guy to design a data warehouse.

The data model takes a much more central role in the system design for a data warehouse than for a transaction system. As Bob argued in Design Tip #123, the data model -- developed *collaboratively* with the business users -- becomes the core of the system specification. If the business users agree that any realistic analysis on the subject area can be met through the data in the model, and IT agrees the data model can be populated, the two sides can shake hands over the model. This is quite different from the standard waterfall approach, where the data modeler would take the spec into his cubicle and emerge several weeks later with the fully formed data model.

Naming conventions. Another place I've seen formal IT processes get in the way is in naming the entities in the data warehouse. Of course, this is much less important an issue than the specifications, but I find myself deeply annoyed by naming dogmatisms. Because the DW database is designed for ad hoc use, business users are going to see the table and column names. They are displayed as report titles and column headers, so extremely long column names are problematic.

That said, naming conventions, the use of case words, and minimizing the use of abbreviations are all good ideas. But like all good ideas, temper them with reason. Taken to an extreme, you can end up with absurdly long column names like (and I'm not making this up) CurrentWorldwideTimezoneAreaIdentifier.

Although they give us something to laugh about, the real problem with absurdly long names is that the users won't tolerate them in their reports. They'll always be changing them in the report display, which means they'll use inconsistent names and introduce a new (and stupid) reason for the "multiple versions of the truth" problem. Please let your naming conventions be tempered by common sense.

Dogma. Unless it's Kimball Group dogma. I have heard all sorts of rules instituted by "someone at corporate IT." These include:

- All queries will be served from stored procedures (clearly someone doesn't understand what "ad hoc" means).
- All ETL will be done in stored procedures (All? Why?).
- All database constraints will be declared and enforced at all times (*most* of the time -- sure; but *all* the time?).
- All tables will be fully normalized (no comment).
- There will be no transformations to data in the DW (I don't have any response to this one other than a puzzled expression).

Don't get us wrong… As easy as it is to mock some practices, we believe in professionally developed and managed DW/BI systems, which usually mean IT. The advantages are huge:

- Central skills in data modeling, ETL architecture, development, and operations are greatly leveraged from one project to the next.
- Professional development, including code check-ins, code reviews, ongoing regression testing, automated ETL, and ongoing management.
- Solid bug tracking, release management techniques, and deployment procedures mean an IT-managed DW/BI system should more smoothly roll out improvements to a system already in production.
- Security and compliance.

However, if you're on a project that's suffering under an IT mandate that makes no sense to you, don't be afraid to push back.

KIMBALL GROUP KIMBALL UNIVERSITY

www.kimballgroup.com Number 128, October 6, 2010

Design Tip #128 Selecting Default Values for Nulls

By Bob Becker

Design Tip #43 Dealing with Nulls in the Dimensional Model describes two cases where null values should be avoided in a dimensional model; in these situations, we recommend using default values rather than nulls. This Design Tip provides guidance for selecting meaningful, verbose defaults.

Handling Null Foreign Keys in Fact Tables

The first scenario where nulls should be avoided is when we encounter a null value as a foreign key for a fact table row during the ETL process. We must do something in this case because an actual null value in a foreign key field of a fact table will violate referential integrity; the DBMS may not allow this situation to happen. There are a number of reasons why we have no foreign key:

- There is a data quality issue because the key value provided by the source system is invalid or incorrect.
- The dimension itself is not applicable for the particular fact row.
- The foreign key value is missing from the source data. In some case, this missing data is another data quality issue. In other cases, the foreign key legitimately is not known because the event being tracked has not yet occurred as frequently happens with accumulating snapshot fact tables.

When dealing with null foreign keys, we suggest applying as much intelligence in the ETL process as possible to select a default dimension row that provides meaning to the business users. Do not simply set up a single default row and point all default scenarios to the same row. Consider each condition separately and provide as many default rows as needed to provide the most complete understanding of the data as possible. At a minimum consider the following default rows:

- Missing Value – The source system did not provide a value that would enable looking up the appropriate foreign key. This could indicate a missing data feed in the ETL process.
- Not Happened Yet – The missing foreign key is expected to be available at a later point in time.
- Bad Value – The source provided bad data or not enough data to determine the appropriate dimension row foreign key. This may be due to corrupted data at the source, or incomplete knowledge of the business rules for this source data for this dimension.
- Not Applicable – This dimension is not applicable to this fact row.

Every dimension needs a set of default rows to handle these cases. Usually the ETL team assigns specific values such as 0, -1, -2, and -3 to the keys that describe these alternatives. The choice of the specific key value generally makes no difference, but there is one weird special case. When the calendar date dimension is used as the basis for partitioning a large fact table (say, on the Activity Date of a set of transactions), care must be taken that the Activity Date always has a real value, not one of the exceptional values, since such an exceptional record will get partitioned off to Siberia in the oldest partition if it has a key value of 0!

Handling Null Attribute Values in Dimension Tables

Nulls should also be avoided when we can’t provide a value for a dimension attribute in a valid dimension row. There are a several reasons why the value of a dimension attribute may not be available:

- Missing Value – The attribute was missing from the source data.

- Not Happened Yet – The attribute is not yet available due to source system timing issues.
- Domain Violation – Either we have a data quality issue, or we don't understand all the business rules surrounding the attribute. The data provided by the source system is invalid for the column type or outside the list of valid domain values.
- Not Applicable – The attribute is not valid for the dimension row in question.

Text attributes in dimension tables usually can contain the actual values that describe the null conditions. Try to keep in mind the effect on BI tools downstream that have to display your special null value description in a fixed format report. Avoid tricks we've seen, such as populating the default attributes with a space or meaningless string of symbols like @@@ as these only confuse the business users. Consider the default values for each dimension attribute carefully and provide as much meaning as possible to provide context to the business users.

Numeric attributes in dimension tables will need to have a set of special values. A value of zero often is the best choice because it is usually obvious to the users that it is artificial. Some numeric attributes will present you with a difficult choice if the business users combine these values in numeric computations. Any actual numeric value used to stand in for null (say, zero) will participate in the computation but give misleading results. An actual null value often will cause an error in a computation, which is annoying but at least does not produce a falsely confident result. Perhaps you can program your BI tool to display null numeric dimension attributes with "null" so that you can both report and compute on these attributes without worrying about distorted data.

Finally, these default value choices should be re-used to describe common null conditions across business processes and dimension tables in your dimensional data warehouse.

KIMBALL GROUP KIMBALL UNIVERSITY

www.kimballgroup.com Number 127, September 1, 2010

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.

KIMBALL GROUP KIMBALL UNIVERSITY

www.kimballgroup.com Number 126, August 3, 2010

Design Tip #126 Disruptive ETL Changes

By Ralph Kimball

Many enterprise data warehouses are facing disruptive changes brought about by increased usage by operations and the exploding interest in customer behavior. My impression is that many shops have implemented isolated adaptations to these new forces but haven't lifted their gaze to the horizon to realize that the data warehouse design landscape has changed in some significant ways, especially in the ETL back room.

The overarching change is modifying the EDW to support mixed workload operational applications where low latency data coexists with historical time series and many other customer facing applications and data sources. Yes, of course we talked about operational data even before Y2K, but the early ODS implementations were restricted to tiny answer set fetches of highly constrained transactional questions such as "was the order shipped?" Today's operational users are drastically more demanding.

Here are seven disruptive changes coming from operational requirements. I wouldn't be surprised if you were facing all seven of these at once:

1. *Driving data latency toward zero*. The urge to see the status of the business at every instant is hard to resist. Not everyone needs it, but for sure someone will claim they do. But as you approach zero latency data delivery, you have to start throwing valuable ETL processes overboard, until finally you only have the vertical retrace interval on your display (1/60th of a second) in which to perform useful ETL work. Although this extreme is admittedly ridiculous, now I have your attention. And keep in mind, if you have true zero latency data delivery, the original source application has to provide the computing power to refresh all the remote BI screens. The lesson here is to be very cautious as your users tighten their requirements to approach zero latency delivery.

2. *Integrating data across dozens, if not hundreds, of sources*. Customer behavior analytics is the rage in the operational/BI world, and there is a lot of money chasing behavior data. Operational and marketing people have figured out that almost any data source reveals something interesting about customer behavior or customer satisfaction. I have seen a number of shops struggling to integrate dozens of not-very-compatible customer facing data collection processes.

3. *Instrumenting and actively managing data quality*. After talking idealistically about data quality for fifteen years, the data warehouse community is now turning actively to do something about it. Within the data warehouse, this takes the form of data quality filters that test for exceptional conditions, centralized schemas that record data quality events, and audit dimensions attached to final presentation schemas. Things get really interesting when you try to address this requirement while simultaneously driving data latency toward zero.

4. *Tracking custody of data for compliance*. Maintaining the chain of custody for critical data subject to compliance means that you can no longer perform SCD Type 1 or Type 3 processing on dimension tables or fact tables. Check out Design Tip #74 available at www.kimballgroup.com to understand how to solve this problem.

5. *Retrofitting major dimensions for true Type 2 tracking*. Organizations are revisiting earlier

decisions to administer important dimensions, such as customer, with Type 1 (overwrite) processing when changes are reported. This provokes a significant change in the ETL pipeline, as well as a serious commitment to the use of surrogate keys. Surrogate keys, of course, simplify the creation of conformed dimensions that combine data from multiple original sources.

6. *Outsourcing and moving to the cloud*. Outsourcing offers the promise of having someone else handle the administration, backing up, and upgrading of certain applications. Outsourcing may also be combined with a cloud implementation which may be an attractive alternative for storing operational data that is subject to very volatile volume surges. The right cloud implementation can scale up or down on very short notice.

7. *Harvesting light touch data that cannot be loaded into an RDMS*. Finally, a number of organizations that have significant web footprints are capable of collecting tens or hundreds of millions of web page events per day. This data can grow into petabytes (thousands of terabytes) of storage. Often, when these “light touch” web events are first collected as stateless microevents, their useful context is not understood until much later. For instance, if a web visitor is exposed to a product reference, and then days or weeks later actually buys the product, then the original web page event takes on much more significance. The architecture for sweeping up and sessionizing this light touch data often involves MapReduce and Hadoop technology. Check out Wikipedia for more on these new technologies.

I think that this list of what are today disruptive changes to your ETL pipelines are actually a glimpse of the new direction for enterprise data warehousing. Rather than being exotic outliers, these techniques may well become the standard approaches for handling customer oriented operational data.

www.kimballgroup.com Number 125, June 30, 2010

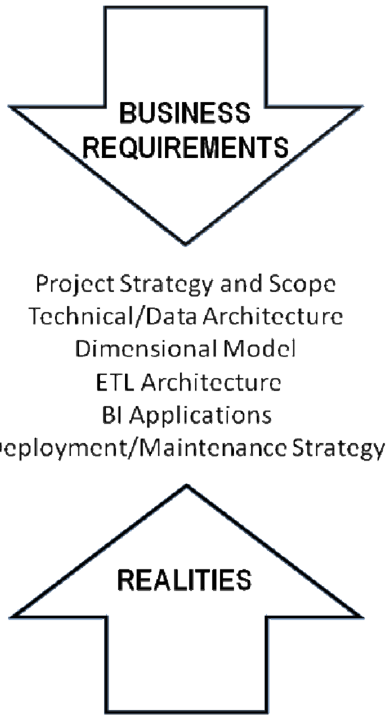
Design Tip #125 Balancing Requirements and Realities

By Margy Ross

If you're a long time reader of the Kimball articles, *Design Tips* and books, you know we feel strongly about the importance of understanding the business's data and analytic requirements.

Suffice it to say that we expect more than merely inventorying the existing reports and data files; we encourage you to immerse yourself in the business community to fully appreciate what business people do and why, and what kind of decisions they make today and hope to make in the future. However, as with many things in life, moderation is prudent. You need to temper the business requirements with numerous realities: the available source data realities, the existing architecture realities, the available resource realities, the political landscape and funding realities, and the list goes on.

Balancing the organization's requirements and realities is a never ending exercise for DW/BI practitioners. It takes practice and vigilance to maintain the necessary equilibrium. Some project teams err on the side of becoming overly focused on the technical realities and create over-engineered solutions that fail to deliver what the business needs. At the other end of the spectrum, teams focus exclusively on the business's needs in a vacuum. Taken to an extreme, these requirements-centric teams fail to deliver because what the business wants is unattainable; more often, the result is a silo point solution to address isolated requirements which perpetuates a potentially inconsistent view of the organization's performance results.


BUSINESS REQUIREMENTS
Project Strategy and Scope
Technical/Data Architecture
Dimensional Model
ETL Architecture
BI Applications
Deployment/Maintenance Strategy
REALITIES

Throughout the key design, development and deployment tasks outlined in our Kimball Lifecycle approach, it's a recurring theme to be constantly mindful of this requisite balancing act. You want to keep the pendulum from swinging too far in one direction where you're exposed to significant delivery and/or business adoption risks.

Most would agree that balance is critical for long-term DW/BI sustainability. However, there’s a trump card for this highwire act. If you can’t unequivocally declare that your DW/BI deliverable has improved the business’s decision-making capabilities, then straddling the requirements and realities is a moot point. Providing an environment that positively impacts the business’s ability to make better decisions must be an overarching, non-negotiable target; delivering anything less is a technical exercise in futility for the organization.

KIMBALL GROUP KIMBALL UNIVERSITY

www.kimballgroup.com Number 124, June 2, 2010

Design Tip #124 Alternatives for Multi-valued Dimensions

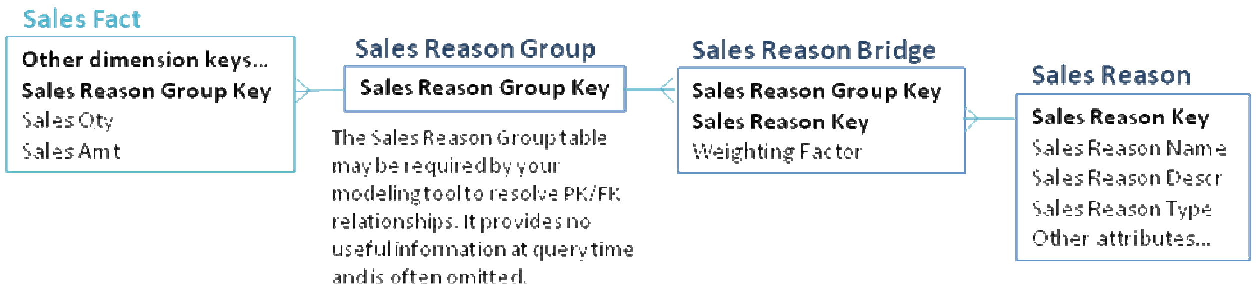
By Joy Munde

The standard relationship between fact and dimension tables is many-to-one: each row in a fact table links to one and only one row in the dimension table. In a detailed sales event fact table, each fact table row represents a sale of one product to one customer on a specific date. Each row in a dimension table, such as a single customer, usually points back to many rows in the fact table.

A dimensional design can encompass a more complex multi-valued relationship between fact and dimension. For example, perhaps our sales order entry system lets us collect information about why the customer chose a specific product (such as price, features, or recommendation). Depending on how the transaction system is designed, it's easy to see how a sales order line could be associated with potentially many sales reasons.

The robust, fully-featured way to model such a relationship in the dimensional world is similar to the modeling technique for a transactional database. The sales reason dimension table is normal, with a surrogate key, one row for each sales reason, and potentially several attributes such as sales reason name, long description, and type. In our simple example, the sales reason dimension table would be quite small, perhaps ten rows. We can't put that sales reason key in the fact table because each sales transaction can be associated with many sales reasons. The sales reason bridge table fills the gap. It ties together all the possible (or observed) sets of sales reasons: {Price, Price and Features, Features and Recommendation, Price and Features and Recommendation}. Each of those sets of reasons is tied together with a single sales reason group key that is propagated into the fact table.

For example, the figure below displays a dimensional model for a sales fact that captures multiple sales reasons:



If we have ten possible sales reasons, the Sales Reason Bridge table will contain several hundred rows.

The biggest problem with this design is its usability by ad hoc users. The multi-valued relationship, by its nature, effectively "explodes" the fact table. Imagine a poorly trained business user who attempts to construct a report that returns a list of sales reasons and sales amounts. It is absurdly easy to double count the facts for transactions with multiple sales reasons. The weighting factor in the bridge table is designed to address that issue, but the user needs to know what the factor is for and how to use it.

In the example we're discussing, sales reason is probably a very minor embellishment to a key fact table that tracks our sales. The sales fact table is used throughout the organization by many user communities, for both ad hoc and structured reporting. There are several approaches to the usability problem presented by the full featured bridge table design. These include:

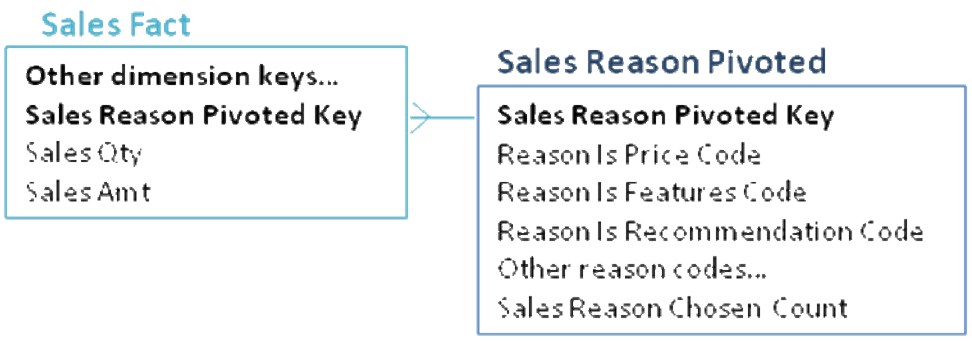
- *Hide the sales reason from most users.* You can publish two versions of the schema: the full one for use by structured reporting and a handful of power users, and a version that eliminates sales reason for use by more casual users.

- *Eliminate the bridge table by collapsing multiple answers.* Add a row to the sales reason dimension table: “Multiple reasons chosen.” The fact table can then link directly with the sales reason dimension. As with all design decisions, the IT organization cannot choose this approach without consulting with the user community. But you may be surprised to hear how many of your users would be absolutely fine with this approach. We’ve often heard users say “oh, we just collapse all multiple answers to a single one in Excel anyway.” For something like a reason code (which has limited information value), this approach may be quite acceptable.

  One way to make this approach more palatable is to have two versions of the dimension structure, and two keys in the fact table: the sales reason group key and the sales reason key directly. The view of the schema that’s shared with most casual users displays only the simple relationship; the view for the reporting team and power users could also include the more complete bridge table relationship.

- *Identify a single primary sales reason.* It may be possible to identify a primary sales reason, either based on some logic in the transaction system or by way of business rules. For example, business users may tell you that if the customer chooses price as a sales reason, then from an analytic point of view, price is the primary sales reason. In our experience it’s relatively unlikely that you can wring a workable algorithm from the business users, but it’s worth exploring. As with the previous approach, you can combine this technique with the bridge table approach for different user communities.

- *Pivot out the sales reasons.* If the domain of the multi-choice space is small -- in other words, if you have only a few possible sales reasons -- you can eliminate the bridge table by creating a dimension table with one column for each choice. In the example we’ve been using, the sales reason dimension would have columns for price, features, recommendation, and each other sales reason. Each attribute can take the value yes or no. This schema is illustrated below:



  This approach solves the fact table explosion problem, but does create some issues in the sales reason dimension. It’s only practical with a relatively small number of domain values, perhaps 50 or 100. Every attribute in the original dimension shows up as an additional column for *each* domain value. Perhaps the biggest drawback is that any change in the domain (adding another sales reason) requires a change in the data model and ETL application.

  Nonetheless, if the multi-valued dimension is important to the broad ad hoc user community, and you have a relatively small and static set of domain values, this approach may be more appealing than the bridge table technique. It’s much easier for business users to construct meaningful queries.

  Clearly the pivoted dimension table doesn’t work for all multi-valued dimensions. The classic example of a multi-valued dimension -- multiple diagnoses for a patient’s hospital visit -- has far too large a domain of possible values to fit in the pivoted structure.

The bridge table design approach for multi-valued dimensions, which Kimball Group has described many times over the past decades, is still the best. But the technique requires an educated user community, and user education seems to be one of the first places the budget cutting axe is applied. In some circumstances, the usability problems can be lessened by presenting an alternative, simpler structure to the ad hoc user community.

KIMBALL GROUP KIMBALL UNIVERSITY

www.kimballgroup.com Number 123, May 5, 2010

Design Tip #123 Using the Dimensional Model to Validate Business Requirements

By Bob Becker

The Kimball Group has frequently written about the importance of focusing on business requirements as the foundation for a successful DW/BI implementation. Design Tip #110 provides a crisp set of dos and don'ts for gathering requirements. However, some organizations find it difficult to land on the right level of detail when documenting the requirements, and then leveraging them to define the scope of a DW/BI development iteration.

Many organizations have formalized rules of engagements for major IT development efforts including a set of structured deliverables used in the development process. This often includes a document such as a functional specification for capturing business requirements. Unfortunately, these documents were originally intended to support operational system development efforts and are typically ineffective for capturing the requirements to build a BI system. It's often difficult to translate DW/BI business requirements into the type of scenarios and detail called for in these templates. In addition, DW/BI requirements are often fuzzy rather than specific; they may be captured in statements such as "measure sales volume every which way" or "slice and dice claims at any level of detail." Sometimes DW/BI requirements are captured as a set of representative analytic questions or a set of predetermined reports with caveats surrounding required flexibility to support ad hoc reporting capabilities. Such requirements make it hard for the business representatives and DW/BI project team to agree on the exact requirements that are in scope or out of scope.

To confront this challenge, organizations should develop the data warehouse bus matrix (see Design Tip #41) and logical dimensional model as deliverables before looping back to finalize the business requirements and scope sign-off. The bus matrix clearly identifies the business processes and associated dimensions that represent the high level data architecture required to support the business requirements. The bus matrix alone can help define scope. The logical dimensional model then describes the details of the DW/BI data architecture: dimension tables, columns, attributes, descriptions, and the beginnings of the source to target maps for a single business process. Including the logical model allows the business representatives to concur that their requirements can be met by the proposed data model. Likewise, it is important that the DW/BI project team commit to the business users that they understand the required data, have performed the necessary technical evaluation including data profiling, and agree that they can populate the logical dimensional model with the anticipated data. Gaining this agreement will enable both the business community and DW/BI project team to reach a common understanding of the business requirements and scope as documented by the logical data model. Once the model has been agreed to, then any business requirement that can't be resolved by the data represented by the logical data model is clearly out of scope. Likewise, any business requirements that can be resolved by the logical data model are in scope.

The logical dimensional model should be developed *jointly* by representatives from all interested groups: business users, reporting teams, and the DW/BI project team. It is critical that the appropriate individuals are represented on the dimensional data model design team as described in Design Tip #103 for this strategy to be effective. Be sure to incorporate thorough reviews of the proposed data models with a wide set of business users and DW/BI project team members to assure that all the business and technical requirements and challenges have been identified (see Design Tip #108).

www.kimballgroup.com Number 122, April 7, 2010

Design Tip #122 Call to Action for ETL Tool Providers

By Warren Thornthwaite

Let me start by stating this Design Tip is an observation on the general state of ETL tools, not any specific ETL product, some of which are better or worse than average.

As I was recently walking through the steps to create a fully functional type 2 slowly changing dimension (SCD) in one of the major ETL tools, it occurred to me that the ETL tools in general have stopped short of achieving their potential. Certainly they continue to add functionality, such as data profiling, metadata management, real-time ETL, and master data management. But for the most part, they are weak on the core functionality required to build and manage a data warehouse database. They haven't completed the job.

Let's look at the slowly changing dimension example that got me started on this topic. The product in question has a wizard that does a reasonable job of setting up the transformations needed to manage both type 1 and type 2 changes on attributes in the target dimension table. It even offers the choice of updating only the current row or all historical rows in the case of a type 1 change. What it doesn't do is manage the type 2 change tracking columns in the way we recommend. When a row changes, you need to set the effective date of the new row and the expiry date of the old row. I also like to update a current row indicator to easily limit a query to just current rows; I realize it's redundant with the value used as the expiration date on the current row (all rows with some distant future expiry date, such as 12/31/9999, are current rows by definition), but the current row indicator is commonly used for convenience and clarity. Unfortunately, this particular wizard will update either the dates or the current row indicator, but not both. So I have to add a step after the wizard to update the current rows.

Another problem with this specific wizard is that it doesn't provide any way to determine which type 2 columns changed to trigger the creation of a new row. This row reason information can be useful to audit the change tracking process; it can also be useful to answer business questions about the dynamics of the dimension. For example, if I have a row reason on a customer dimension, it's easy to answer the question "How many people moved last year?" Without the row reason, I have to join the table back on itself in a fairly complex way and compare the current row to the previous row and see if the zip code column changed. This is frustrating because I know the wizard knows which columns changed, but it's just keeping it secret.

I realize other ETL tools do a better job of automatically handling slowly changing dimensions. What's disappointing about the general state of ETL tools today is not the shortcomings of this particular SCD processing example, but the lack of effective support for most of the other 33 ETL subsystems, many of which we see in use in almost every data warehouse.

For example, junk dimensions are common dimensional modeling constructs. By combining multiple small dimensions into a single physical table, we simplify the model and remove multiple foreign key columns from the fact table. Creating and maintaining a junk dimension is not conceptually difficult, but it is tedious. You either create cross-join of the rows in each dimension table if there aren't too many possible combinations, or you add combinations to the junk dimension as they occur in the incoming fact table. An ETL tool vendor that really wants to help their ETL developer customers should provide a wizard or transformation that automatically creates and manages junk dimensions, including the mapping of junk dimension keys to the fact table.

The same issue applies to mini dimensions which are column subsets extracted out of a large dimension and put into a separate dimension, which is joined directly to the fact row. Like junk dimensions, mini dimensions are not conceptually difficult, but they are tedious to build from scratch every time. Unfortunately, I know of no ETL tool that offers mini dimension creation and maintenance as a standard component or wizard.

ETL vendors are often enthusiastic about demonstrating that their tools can address the 34 subsystems we describe and teach in our design classes. Unfortunately the vendors' responses are typically either PowerPoint slides or demo-ware implemented by an analyst at headquarters, rather than by an ETL developer working in a production environment.

I could go on down the list of subsystems, but you get the idea. Here is a link to an article Bob Becker wrote describing the 34 subsystems. Take a look and see how much time you spend coding these components by hand in your ETL environment. Then ask your ETL vendor what their plans are for making your job easier.

www.kimballgroup.com Number 121, March 3, 2010

Design Tip #121 Columnar Databases: Game Changers for DW/BI Deployment?

By Ralph Kimball

Although columnar RDBMSs have been in the market since the 1990s, the recent BI requirements to track rapidly growing customer demographics in enormous terabyte databases have made some of the advantages of columnar databases increasingly interesting.

Remember that a columnar RDBMS is a “standard” relational database supporting the familiar table and join constructs we are all used to. What makes a columnar RDBMS unique is the way data is stored at the physical level. Rather than storing each table row as a contiguous object on the disk, a columnar RDBMS stores *each table column* as a contiguous object on the disk. Typically, these column specific data objects are sorted, compressed, and heavily indexed for access. Even though the underlying physical storage is columnar, at the user query level (SQL), all tables appear to be made up of familiar rows. Applications do not have to be recoded to use a columnar RDBMS. The beneath-the-cover twist from a row orientation to a column orientation is the special feature of a columnar RDBMS.

A columnar RDBMS offers the application designer and the DBA some design advantages, including:

• Significant database compression. Dimension tables filled with many low cardinality repeated attributes compress significantly, sometimes by 95% or even more. Fact tables with numeric measurement values also compress surprisingly well, often by 70% or more. Columnar databases are very supportive of the dimensional modeling approach.

• The query and storage penalties for very wide tables with many columns are greatly reduced. A single column, especially one with low cardinality, adds very little to the overall storage of a table. More significantly, when a table row is fetched in a query, only the named columns are actually fetched from the disk. Thus if three columns are requested from a 100 column dimension, only about 3% of the row content is retrieved. In a conventional row oriented RDBMS, typically the entire row must be fetched when any part of the row is needed. (More exactly, the disk blocks containing the requested data are fetched into the query buffer and much un-requested data in these blocks comes along for the ride).

• Adding a column to a fact or dimension table in a columnar RDBMS is not an especially big deal since the system doesn’t care about row widths. No more block splits when you expand a “row.” Now the designer can be much more profligate about adding columns to an existing table. For instance, you can now be comfortable with wider fact table designs, where many of the fields in a “row” have null values. In this way, the designer can regularly add new demographics columns to a customer dimension because each column is an independent compressed object in the physical storage. Be aware, however, that as you add more columns to your fact and dimension tables, you have increased responsibility to keep the BI tool user interfaces simple. A tried and true way to keep the user interfaces simple is to deploy multiple logical views for different BI use cases, where each view exposes a set of relevant fields used together. And, we should point out that adding more columns to a table is not an excuse for adding data to a table that violates the grain!

• As I mention in my design classes, SQL is horribly asymmetric in that it allows very complex computations and constraints within records, but makes it virtually impossible to apply computations or constraints across records. Columnar RDBMSs remove some of this applications pressure, not only because they encourage far wider table designs (thereby exposing many fields within the same row to SQL), but also because columnar RDBMSs specialize in cross column access through bitmap indexes and other special data structures. BI tool designers welcome the extended width of fact and dimension tables because within-row constraints and computations are a hundred times easier to set up than the corresponding cross-row versions.

Columnar databases pose provocative tradeoffs for IT. In particular, these databases have had a reputation for slow loading performance, which the vendors have been addressing. Before investing in a columnar database, make sure to carefully prototype loading and updating back room tasks. But in any case, these databases offer some interesting design alternatives.

KIMBALL GROUP KIMBALL UNIVERSITY

www.kimballgroup.com Number 120, February 2, 2010

Design Tip #120 Design Review Dos and Don'ts

By Margy Ross

Over the years, we've described common dimensional modeling mistakes, such as our October '03 "Fistful of Flaws" article in *Intelligent Enterprise*. And we've recommended dimensional modeling best practices countless times; our May '09 "Kimball's Ten Rules of Dimensional Modeling" article has been widely read.

While we've identified frequently-observed errors and suggested patterns, we haven't provided much guidance about the process of conducting design reviews on existing dimensional models. Kimball Group consultants perform numerous design reviews like this for clients as it's a cost-effective way to leverage our experience; here are some practical dos and don'ts for conducting a design review yourself.

Before the design review...

- Do invite the right players. Obviously, the modeling team needs to participate, but you'll also want representatives from the BI development team (to ensure that proposed changes enhance usability) and ETL development team (to ensure that the changes can be populated). Perhaps most importantly, it's critical that folks who are really knowledgeable about the business and their needs are sitting at the table. While diverse perspectives should participate in a review, don't invite 25 people to the party.

- Do designate someone to facilitate the review. Group dynamics, politics, and the design challenges themselves will drive whether the facilitator should be a neutral resource or involved party. Regardless, their role is to keep the team on track to achieving a common goal. Effective facilitators need the right mix of intelligence, enthusiasm, confidence, empathy, flexibility, assertiveness, and the list goes on. Don't forget a sense of humor.

- Do agree upon the scope of the review (e.g., dimensional models focused on several tightly coupled business processes.) Ancillary topics will inevitably arise during the review, but agreeing in advance on the scope makes it easier to stay focused on the task at hand.

- Do block off time on everyone's calendar. We typically conduct dimensional model reviews as a focused 2-day effort. The entire review team needs to be present for the full two days. Don't allow players to float in and out to accommodate other commitments. Design reviews require undivided attention; it's disruptive when participants leave intermittently.

- Do reserve the right space. The same conference room should be blocked for the full two days. Optimally, the room has a large white board; it's especially helpful if the white board drawings can be saved or printed. If a white board is unavailable, have flip charts on hand. Don't forget markers and tape; drinks and food never hurt.

- Do assign homework. For example, ask everyone involved to make a list of their top five concerns, problem areas, or opportunities for improvement with the existing design. Encourage participants to use complete sentences when making their list so it's meaningful to others. These lists should be emailed to the facilitator in advance of the design review for consolidation.

Soliciting advance input gets people engaged and helps avoid “group think” during the review.

During the design review…

- Do check attitudes at the door. While it’s easier said than done, don’t be defensive about prior design decisions. Do embark on the review thinking change is possible; don’t go in resigned to believing nothing can be done to improve the situation.

- Unless needed to support the review process, laptops and smartphones should also be checked at the door (at least figuratively). Allowing participants to check email during the sessions is no different than having them leave to attend an alternative meeting.

- Do exhibit strong facilitation skills. Review ground rules. Ensure that everyone is participating and communicating openly. Do keep the group on track; ban side conversations and table discussions that are out of scope or spirally into the death zone. There are tomes written on facilitation best practices, so we won’t go into detail here.

- Do ensure a common understanding of the current model before delving into potential improvements. Don’t presume everyone around the table already has a comprehensive perspective. It may be worthwhile to dedicate the first hour to walking through the current design and reviewing objectives. Don’t be afraid to restate the obvious.

- Do designate someone to act as scribe, taking copious notes about both the discussions and decisions being are made.

- Do start with the big picture. Just as when you’re designing from a blank slate, begin with the bus matrix, then focus on a single high priority business process, starting with the granularity then moving out to the corresponding dimensions. Follow this same “peeling back the layers of the onion” method with your design review, starting with the fact table, then tackling dimension-related issues. Do undertake the meatiest issues first; don’t defer the tough stuff to the afternoon of the second day.

- Do remind everyone that business acceptance is the ultimate measure of DW/BI success; the review should focus on improving the business users’ experience.

- Do sketch out sample rows with data values during the review sessions to ensure everyone has a common understanding of the recommended changes.

- Do close the meeting with a recap; don’t let participants leave the room without clear expectations about their assignments and due dates. Do establish a time for the next follow-up.

Following the design review…

- Do anticipate that some open issues will remain after the 2-day review. Commit to wrestling these issues to the ground, even though this can be challenging without an authoritative party involved. Don’t fall victim to analysis paralysis.

- Don’t let your hard work gather dust. Do evaluate the cost/benefit for the potential improvements; some changes will be more painless (or painful) than others. Then develop action plans for implementing the improvements.

- Do anticipate similar reviews in the future. Plan to reevaluate every 12 to 24 months. Do try to view inevitable changes to your design as signs of success, rather than failure.

Good luck with your review!

KIMBALL GROUP KIMBALL UNIVERSITY

www.kimballgroup.com Number 119, December 2, 2009

Design Tip #119 Updating the Date Dimension

By Joy Mundy

Most readers are familiar with the basic date dimension. At the grain of a calendar day, we include various labels for calendar and fiscal years, quarters, months, and days. We include both short and long labels for various reporting requirements. Even though the labels in the basic date dimension could be constructed at the time we design a report, we always build the date dimension in advance so that report labels are consistent and easy to use.

The date dimension gets new rows as time marches forward, but most attributes are not subject to updates once a row has been added. December 1, 2009 will, of course, always roll up to December, calendar Q4, and 2009.

However, there are some attributes you may add to the basic date dimension that will change over time. These include indicators such as IsCurrentDay, IsCurrentMonth, IsPriorDay, IsPriorMonth, and so on. IsCurrentDay obviously must be updated each day. The attribute is particularly useful for generating reports that always run for today – or, even better, a report which defaults to today but which can be run for any day in the past. A nuance to consider is the day that IsCurrentDay refers to. Most data warehouses load data daily, so IsCurrentDay should refer to yesterday (or more accurately, the most recent day of data in the system).

You might also want to add attributes to your date dimension that are unique to your business processes or corporate calendar. These are particularly valuable because they cannot be derived using SQL calendar functions. Examples include IsFiscalMonthEnd, IsCloseWeek, IsManagementReviewDay, and IsHolidaySeason.

Some date dimension designs include lag columns. The LagDay column would take the value 0 for today, -1 for yesterday, +1 for tomorrow, and so on. This attribute could easily be a computed column rather than physically stored. It might be useful to set up similar structures for month, quarter, and year. Many reporting tools, including all OLAP tools, include functionality to do prior period kinds of calculations, so the lag columns are often not required.

Developers are sometimes skeptical of the date dimension's value since many of the attributes can be derived by the report developer. There are many excellent reasons to have a date dimension, but by including valuable attributes that cannot be computed, you may avoid the argument altogether.

www.kimballgroup.com Number 118, November 4, 2009

Design Tip #118 Managing Backlogs Dimensionally

By Bob Becker

Certain industries need the ability to look at a backlog of work, and project that backlog into the future for planning purposes. The classic example is a large services organization with multi-month or multi-year contracts representing a large sum of future dollars to be earned and/or hours to be worked. Construction companies, law firms and other organizations with long term projects or commitments have similar requirements. Manufacturers that ship against standing blanket orders may also find this technique helpful.

Backlog planning requirements come in several flavors supporting different areas of the organization. Finance needs to understand future cash flow in terms of expenditures and cash receipts, and properly project both invoiced and recognized revenue for management planning and expectation setting. There are operational requirements to understand the flow of work for manpower, resource management and capacity planning purposes. And the sales organization will want to understand how the backlog will ultimately flow to understand future attainment measures.

Dimensional schemas can be populated when a new contract is signed, capturing the initial acquisition or creation of the contract and thus the new backlog opportunity. In addition, another schema can be created that captures the work delivered against the contract over time. These two schemas are interesting and useful, but by themselves are not enough to support the future planning requirements. They show that the organization has “N” number of contracts worth “X” millions of dollars with “Y” millions of dollars having been delivered. From these two schemas, the current backlog can be identified by subtracting the delivered amount from the contracted amount. Often it is worthwhile to populate the backlog values in another schema as the rules required to determine the remaining backlog may be relatively complex. Once the backlog amount is understood, it then needs to be accurately projected into the future based on appropriate business rules.

The use of another schema we call the “spread” fact table is helpful in supporting the planning requirements. The spread fact table is created from the backlog schema discussed above. The backlog and remaining time on the contract are evaluated and the backlog is then spread out into the appropriate future planning time buckets and rows are inserted into the fact table. For this discussion we’ll assume monthly time periods, but it could just as easily be daily, weekly or quarterly. Thus the grain of our spread fact table will be at the month by contract (whatever is the lowest level used in the planning process). This schema will also include other appropriate conformed dimensions such as customer, product, sales person, and project manager. In our example, the interesting metrics might include the number of hours to be worked, as well as the amount of the contract value to be delivered in each future month.

In addition, we include another dimension called the scenario dimension. The scenario dimension describes the planning scenario or version of the spread fact table’s rows. This may be a value such as “2009 October Financial Plan” or “2009 October Operational Plan.” Thus, if we plan monthly, there will be new rows inserted into the spread fact table each month described by a new row in the scenario dimension. The secret sauce of the spread fact table is the business rules used to break down the backlog value into the future spread time buckets. Depending on the sophistication and maturity of the planning process, these business rules may simply spread the backlog into equal buckets based on the remaining months in the contract. In other organizations, more complex rules may be utilized that evaluate a detailed staffing and work plan incorporating seasonality trends using a complex algorithm to calculate a very precise amount for each future time period in the spread fact table.

By creatively using the scenario dimension, it is possible to populate several spreads each planning period based on different business rules to support different planning assumptions. As indicated in the scenario descriptions above, it may be possible that the financial planning algorithms are different than the operational planning algorithms for a variety of reasons.

The spread fact table is not just useful for understanding the backlog of actual work. Similar planning requirements often surface with other business processes. Another example is planning for sales opportunities that are proposed but have not yet been signed. Assuming the organization has an appropriate source for the future sales opportunities, this would be another good fit for a spread fact table. Again, appropriate business rules need to be identified to evaluate a future opportunity and determine how to spread the proposed contract amounts into the appropriate future periods. This schema can also include indicators that describe the likelihood of winning the opportunity, such as forecast indicators and percent likely to close attributes. These additional attributes will enable the planning process to look at best case/worst case future scenarios. Typically, the sales opportunities spread fact table will need to be populated as a separate fact table than the actual backlog spread as the dimensionality between the two fact tables is typically quite different. A simple drill across query will enable the planning process to align the solid backlog along with the softer projected sales opportunities to paint a more complete picture of what the future may hold for the organization.

www.kimballgroup.com Number 117, September 30, 2009

Design Tip #117 Dealing with Data Quality: Don't Just Sit There, Do Something!

By Warren Thornthwaite

Most data quality problems can be traced back to the data capture systems because, historically, they have only been responsible for the level of data quality needed to support transactions. What works for transactions often won't work for analytics. In fact, many of the attributes we need for analytics are not even necessary for the transactions, and therefore capturing them correctly is just extra work. By requiring better data quality as we move forward, we are requiring the data capture system to meet the needs of both transactions and analytics. Changing the data capture systems to get better data quality is a long term organizational change process. This political journey is often paralyzing for those of us who didn't expect to be business process engineers in addition to being data warehouse engineers!

Do not let this discourage you. You can take some small, productive steps in the short term that will get your organization on the road to improving data quality.

Perform Research

The earlier you identify data quality problems, the better. It will take much more time if these problems only surface well into the ETL development task, or worse, after the initial rollout. And it will tarnish the credibility of the DW/BI system (even though it's not your fault).

Your first pass at data quality research should come as part of the requirements definition phase early in the lifecycle. Take a look at the data required to support each major opportunity. Initially, this can be as simple as a few counts and ratios. For example, if the business folks wanted to do geographic targeting, calculating the percentage of rows in the customer table where the postal code is NULL might be revealing. If 20 percent of the rows don't have a postal code, you have a problem. Make sure you include this information in the requirements documentation, both under the description of each opportunity that is impacted by poor data quality, and in a separate data quality section.

The next opportunity for data quality research is during the dimensional modeling process. Defining each attribute in each table requires querying the source systems to identify and verify the attribute's domain (the list of possible values the attribute can have). You should go into more detail at this point, investigating relationships among columns, such as hierarchies, referential integrity with lookup tables, and the definition and enforcement of business rules.

The third major research point in the lifecycle is during the ETL system development. The ETL developer must dig far deeper into the data and often discovers more issues.

A data quality / data profiling tool can be a big help for data quality research. These tools allow you to do a broad survey of your data fairly quickly to help identify questionable areas for more detailed investigation. However, if you don't have a data quality tool in place, don't stop your research until you find the best tool and the funds to purchase it. Simple SQL statements like:

```
SELECT PostalCode, COUNT(*) AS RowCount
FROM Dim_Customer GROUP BY PostalCode ORDER BY 2 DESC;
```

will help you begin to identify anomalies in the data immediately. You can get more sophisticated later, as you generate awareness and concern about data quality.

It’s a good idea to include the source systems folks in the research process. If they have a broader sense of responsibility for the data they collect, you may be able to get them to adjust their data collection processes to fix the problems. If they seem amenable to changing their data collection processes, it is a good idea to batch together as many of your concerns as possible while they are in a good mood. Source systems folks often aren’t happy at updating and testing their code too frequently. Don’t continuously dribble little requests to them!

Share Findings

Once you have an idea of the data quality issues you face, and the analytic problems they will cause, you need to educate the business people. Ultimately, they will need to re-define the data capture requirements for the transaction systems and allocate additional resources to fix them. They won’t do this unless they understand the problems and associated costs.

The first major chance to educate on data quality problems is as part of the opportunity prioritization session with senior management. You should show examples of data quality problems, explain how they are created, and demonstrate their impact on analytics and project feasibility. Explain that you will document these in more detail as part of the modeling process, and at that point you can reconvene to determine your data quality strategy. Set the expectation that this is work and will require resources.

The dimensional modeling process is the second major education opportunity. All of the issues you identify during the modeling process should be discussed as part of documenting the model, and an approach to remedying the problem should be agreed upon with key business folks.

At some point, you should have generated enough awareness and concern to establish a small scale data governance effort which will become the primary research and education channel for data quality.

Conclusion

Improving data quality is a long, slow educational process of teaching the organization about what’s wrong with the data, the cost in terms of accurate business decision making, and how best to fix it. Don’t let it overwhelm you. Just start with your highest value business opportunity and dive into the data.

www.kimballgroup.com Number 116, September 2, 2009

Design Tip #116 Add Uncertainty to Your Fact Table

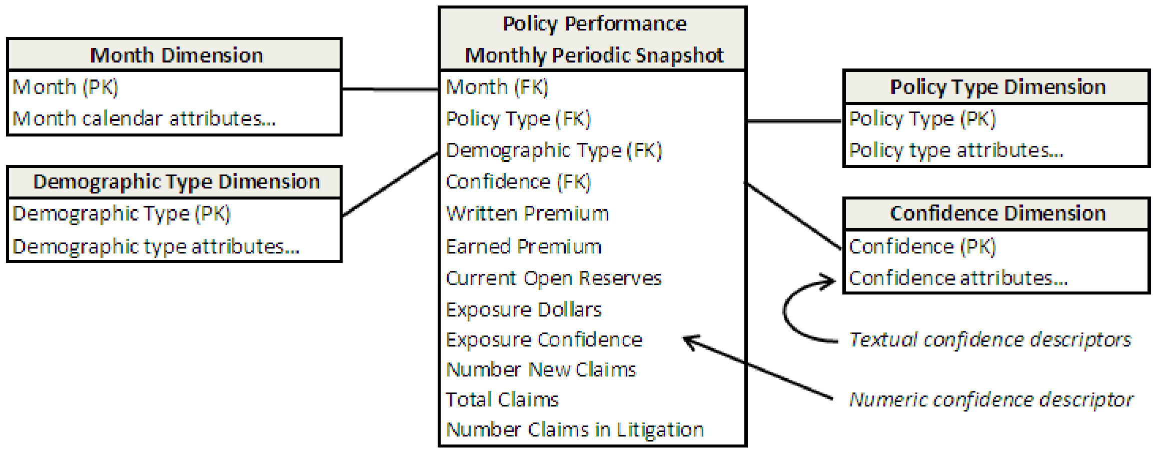
By Ralph Kimball

We always want our business users to have confidence in the data we deliver through our data warehouses. Thus it goes against our instincts to talk about problems encountered in the ETL back room, or known inaccuracies in the source systems. But this reluctance to expose our uncertainties can ultimately hurt our credibility and lessen the business users' confidence in the data if a data quality problem is revealed and we haven't said anything about it.

Almost thirty years ago when I was first exposed to A.C. Nielsen's pioneering data warehouse solution, the Inf*Act data reporting service for grocery store scanner data, I was surprised to see critical key performance indicators in their reports occasionally marked with asterisks indicating a lack of confidence in the data. In this case, the asterisk meant "not-applicable data encountered in the computation of this metric." The key performance indicator nevertheless appeared in the report, but the asterisk warned the business user not to overly trust the value. When I asked Nielsen about these asterisks, they told me the business users appreciated the warning not to make a big decision based on a specific value. I liked this implied partnership between the data provider and business user because it promoted an atmosphere of trust. And, of course, the reminders of data quality issues motivated the data provider to improve the process so as to reduce the number of asterisks.

Today I rarely see such warnings of uncertainty in the final BI layer of our data warehouses. But our world is far more wired to data than it was in 1980. I think it is time we reintroduced "uncertainty" into our fact tables. Here are two places we can add uncertainty into any fact table without changing the grain or invalidating existing applications.

Using property and casualty insurance as an example, one of the fact table's key performance indicators is the exposure dollars for a group of policies with certain demographics. This is an estimate of total liability for the known claims against the chosen set of policies. Certainly the insurance company management will pay attention to this number!



In the above figure, we accompany the exposure dollar value with an exposure confidence metric, whose value ranges between 0 and 1. An exposure confidence value of 0 indicates no confidence in the reported exposure dollars and a value of 1 indicates complete confidence. We assign the exposure confidence value in the back room ETL processes by examining the status of each claim that contributes to the aggregated record shown in the above figure. Presumably a claim associated with a very large exposure but whose claim status is “preliminary estimate” or “unverified claim” or “claim disputed” would lower the overall exposure confidence on the summary record in this fact table. If the doubtful claim’s individual reserve value is given a weight of zero, then the overall reserve confidence metric could be a weighted average of the individual reserve values.

The above figure also shows a confidence dimension containing textual attributes describing the confidence in one or more of the values in the fact table. A textual attribute for the exposure confidence would be correlated with the exposure confidence metric. An exposure confidence between 0.95 and 1 might correspond to “Certain.” An exposure confidence between 0.7 and 0.94 might correspond to “Less Certain,” and an exposure confidence of less than 0.7 might correspond to “Unreliable.” The combination of numeric and textual confidence information in our example allows BI tools to display numeric values in various ways (e.g., using italics for data less than Certain), and allows the BI tool to constrain and group on ranges of confidence.

This example should be a plausible template for providing confidence indicators on almost any fact table. I have found it to be a useful exercise just to imagine what the confidence is for various delivered metrics. And there is no question that the business users will find that exercise useful too.

KIMBALL GROUP KIMBALL UNIVERSITY

www.kimballgroup.com Number 115, August 4, 2009

Design Tip #115 Kimball Lifecycle in a Nutshell

By Margy Ross

A student in a recent Data Warehouse Lifecycle in Depth class asked me for an overview of the Kimball Lifecycle approach to share with their manager. Confident that we'd published an executive summary, I was happy to oblige. Much to my surprise, our only published Lifecycle overview was a chapter in a *Toolkit* book, so this Design Tip addresses the unexpected content void in our archives.

The Kimball Lifecycle approach has been around for decades. The concepts were originally conceived in the 1980s by members of the Kimball Group and several colleagues at Metaphor Computer Systems. When we first published the methodology in *The Data Warehouse Lifecycle Toolkit* (Wiley 1998), it was referred to as the Business Dimensional Lifecycle because this name reinforced three fundamental concepts:

- Focus on adding *business* value across the enterprise
- *Dimensionally* structure the data delivered to the business via reports and queries
- Iteratively develop the solution in manageable *lifecycle* increments rather than attempting a Big Bang deliverable

Rewinding back to the 1990s, our methodology was one of the few emphasizing this set of core principles, so the Business Dimensional Lifecycle name differentiated our approach from others in the industry. Fast forwarding to the late 2000s when we published the second edition of the *Lifecycle Toolkit* (Wiley 2008), we still absolutely believed in these concepts, but the industry had evolved. Our principles had become mainstream best practices touted by many, so we condensed the methodology's official name to simply the Kimball Lifecycle.

In spite of dramatic advances in technology and understanding during the last couple of decades, the basic constructs of the Kimball Lifecycle have remained strikingly constant. Our approach to designing, developing and deploying DW/BI solutions is tried and true. It has been utilized by thousands of project teams in virtually every industry, application area, business function, and technology platform. The Kimball Lifecycle approach has proven to work again and again.

The Kimball Lifecycle approach is illustrated in Figure 1. Successful DW/BI implementations depend on the appropriate amalgamation of numerous tasks and components; it's not enough to have a perfect data model or best of breed technology. The Lifecycle diagram is the overall roadmap depicting the sequence of tasks required for effective design, development, and deployment.

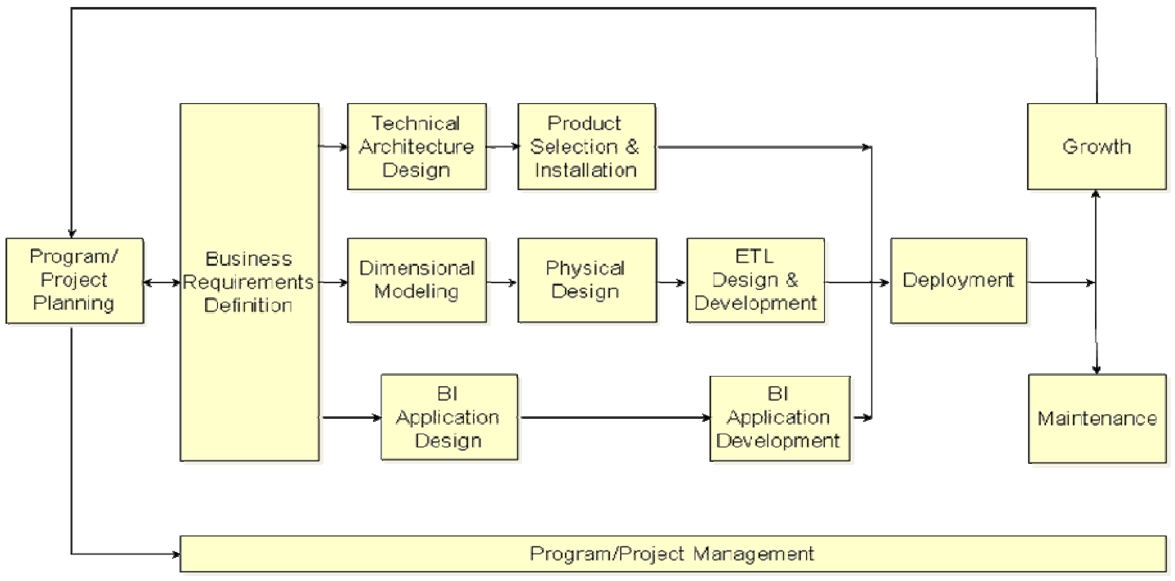


Figure 1 The Kimball Lifecycle diagram.

Program/Project Planning and Management:

The first box on the roadmap focuses on getting the program/project launched, including scoping, justification and staffing. Throughout the Lifecycle, ongoing program and project management tasks keep activities on track.

Business Requirements:

Eliciting business requirements is a key task in the Kimball Lifecycle as these findings drive most upstream and downstream decisions. Requirements are collected to determine the key factors impacting the business by focusing on what business users do today (or want to do in the future), rather than asking “what do you want in the data warehouse?” Major opportunities across the enterprise are identified, prioritized based on business value and feasibility, and then detailed requirements are gathered for the first iteration of the DW/BI system development. Three concurrent Lifecycle tracks follow the business requirements definition.

Technology Track:

DW/BI environments mandate the integration of numerous technologies, data stores, and associated metadata. The technology track begins with system architecture design to establish a shopping list of needed capabilities, followed by the selection and installation of products satisfying those architectural needs.

Data Track:

The data track begins with the design of a target dimensional model to address the business requirements, while considering the underlying data realities. The word Kimball is synonymous with dimensional modeling where data is divided into either measurement facts or descriptive dimensions. Dimensional models can be instantiated in relational databases, referred to as star schemas, or multidimensional databases, known as OLAP cubes. Regardless of the platform, dimensional models attempt to address two simultaneous goals: ease of use from the users’ perspective and fast query performance. The Enterprise Data Warehouse Bus Matrix is a key Kimball Lifecycle deliverable representing an organization’s core business processes and associated common conformed dimensions; it’s a data blueprint to ensure top-down enterprise integration with manageable bottom-up delivery by focusing on a single business process at a time. The bus matrix is tremendously important because it simultaneously serves as a technical guide, a management guide, and a forum for communication with executives.

The dimensional model is converted into a physical design where performance tuning strategies are considered, then the ETL system design and development challenges are tackled. The Lifecycle describes 34 subsystems in the extract, transformation and load process flow grouped into four major operations: *extracting* the data from the source, performing *cleansing and conforming* transformations, *delivering* the data to the presentation layer, and *managing* the backroom ETL processes and environment.

Business Intelligence Track:

While some project members are immersed in the technology and data, others focus on identifying and constructing a broad range of BI applications, including standardized reports, parameterized queries, dashboards, scorecards, analytic models, data mining applications, along with the associated navigational interfaces.

Deployment, Maintenance, and Growth:

The three Lifecycle tracks converge at deployment, bringing together the technology, data and BI applications. The deployed iteration enters a maintenance phase, while growth is addressed by the arrow back to project planning for the next iteration of the DW/BI system. Remember that a DW/BI system is a long term process, not a one-off project!

Throughout the Kimball Lifecycle, there's a recurring theme acknowledging that DW/BI professionals must continuously straddle the business's requirements and the underlying realities of the source data, technology, and related resources. Project teams who focus exclusively on the requirements (or realities) in isolation will inevitably face significant delivery and/or business adoption risks.

Finally, we've said it before, and we'll surely repeat it again. Regardless of your organization's specific DW/BI objectives, we believe your overarching goal should be *business acceptance of the DW/BI deliverables to support decision making*. This target must remain in the bull's eye throughout the design, development, and deployment lifecycle of any DW/BI system.

KIMBALL GROUP KIMBALL UNIVERSITY

www.kimballgroup.com Number 114, July 8, 2009

Design Tip #114 Avoiding Alternate Organization Hierarchies

By Joy Mundy

Most organizations have one or several organizational hierarchies used by sales, finance, and management to roll up data across the enterprise. These organizational hierarchies become key conformed dimensions in the enterprise data warehouse, used for drilling up and down, slicing and dicing, performance-based aggregations, and even security roles. Despite their central position in management reporting, organization hierarchies are often managed – or un-managed – in a haphazard fashion.

You should manage your organization hierarchies centrally and professionally. There should be a clearly defined process for making changes to the organizational structure. Disciplined companies may try to limit major hierarchical changes only at fiscal year boundaries; changing the org structure as seldom as possible helps people mentally track those changes.

Professional management of hierarchies delivers roll-ups that are officially blessed as the truth. The more effective the official hierarchies, the less likely some business users will need to create alternative hierarchies. Unfortunately, in the real world we often see demand for alternative hierarchies: mid-level managers want to see their organization structured differently than the official rollup. In the data warehouse, we're often asked to bring in those alternative hierarchies, to enable hands-on managers to view information in a way that matches their business. It is easy and common to bring in multiple hierarchies, but a better solution – if possible – is to embellish the official hierarchies.

Senior management typically focuses on the top few levels of an organizational hierarchy – typically, the level of detail exposed in external reporting – and unsurprisingly those top levels work well. However, lower rank managers sometimes perceive two types of problems with official hierarchies:

- Small internal organizations that don't need all the levels available to them. You can address this problem by filling in the lowest levels of the hierarchy with meaningful defaults, and designing drillable reports that make it easy for analysts to hide the repetitive structures.

- Large internal organizations that can't fit everything into the available levels. For example, a recent client has 60% of its business in the United States, and the US sales organization had a more complex organization than the non-US business. This situation may be solved by adding one or two levels to the official hierarchy. This obvious solution of adding more levels to the official hierarchy will only be effective if you don't make it too painful for smaller organizations that don't need more levels.

Managing the hierarchies professionally means that one organization – ideally a data administration or data management organization – is in charge of making changes to the rollup structures. Although the central organization is the only one to actually make changes, let the people who will use a piece of the hierarchy be the ones to design its structure. Design a system that distributes ownership broadly across the organization, but still keeps everyone marching to the same drummer.

A good master data management (MDM) system that includes workflow components is the ideal tool to facilitate this process. Assign owners at each node of the hierarchy, and designate additional people who must sign off on any change:

- Owners of immediate parent and child nodes

- Representative(s) from finance
- Representative(s) from data management organization

In addition, designate a larger group of people who are notified of a proposed change which may affect them. If you don't have a MDM tool to manage this process, email works too.

The management of the organizational hierarchies is really not a data warehouse function; all these changes, and the communication required, must be managed upstream of the data warehouse.

If you have distributed management of the hierarchical structures throughout the organization, and included enough levels for all parts of the organization, you will see a greatly reduced demand for alternative hierarchies. Many organizations will get by with one to three official hierarchies. With strong management encouragement to use only official hierarchies, you may be able to eliminate the need for other local structures.

www.kimballgroup.com

Number 113, June 3, 2009

Design Tip #113 Creating, Using, and Maintaining Junk Dimensions

By Warren Thornthwaite

The A junk dimension combines several low-cardinality flags and attributes into a single dimension table rather than modeling them as separate dimensions. There are good reasons to create this combined dimension, including reducing the size of the fact table and making the dimensional model easier to work with. Margy described junk dimensions in detail in Kimball Design Tip #48: De-Clutter with Junk (Dimensions). On a recent project, I addressed three aspects of junk dimension processing: building the initial dimension, incorporating it into the fact processing, and maintaining it over time.

Build the Initial Junk Dimension

If the cardinality of each attribute is relatively low, and there are only a few attributes, then the easiest way to create the dimension is to cross-join the source system lookup tables. This creates all possible combinations of attributes, even if they might never exist in the real world.

If the cross-join of the source tables is too big, or if you don't have source lookup tables, you will need to build your junk dimension based on the actual attribute combinations found in the source data for the fact table. The resulting junk dimension is often significantly smaller because it includes only combinations that actually occur.

We'll use a simple health care example to show both of these combination processes. Hospital admissions events often track several standalone attributes, including the admission type and level of care required, as illustrated below in the sample rows from the source system lookup and transaction tables.

Admit_Type_Source

Admit _ Type_ID	Admit_Type_ Descr
1	Walk-in
2	Appointment
3	ER
4	Transfer

Care_Level_Source

Care_ Level_ ID	Care_Level_ Descr
1	ICU
2	Pediatric ICU
3	Medical Floor

Fact_Admissions_Source

Admit_ Type_ID	Care_ Level_ID	Admission_Count
1	1	1
2	1	1
2	2	1
5	3	1

The following SQL uses the cross-join technique to create all 12 combinations of rows (4x3) from these two source tables and assign unique surrogate keys.

```
SELECT  ROW_NUMBER() OVER( ORDER BY Admit_Type_ID, Care_Level_ID) AS
Admission_Info_Key,
Admit_Type_ID, Admit_Type_Descr, Care_Level_ID, Care_Level_Descr
FROM Admit_Type_Source
CROSS JOIN Care_Level_Source;
```

In the second case, when the cross-join would yield too many rows, you can create the combined dimension based on actual combinations found in the transaction fact records. The following SQL uses outer joins to prevent a violation of referential integrity when a new value shows up in a fact source row that is not in the lookup table.

```
SELECT ROW_NUMBER() OVER(ORDER BY F.Admit_Type_ID) AS
Admission_Info_Key,
F.Admit_Type_ID, ISNULL(Admit_Type_Descr, 'Missing Description')
Admit_Type_Descr,
F.Care_Level_ID, ISNULL(Care_Level_Descr, 'Missing Description')
Care_Level_Descr   -- substitute NVL() for ISNULL() in Oracle
FROM Fact_Admissions_Source F
LEFT OUTER JOIN Admit_Type_Source C ON
      F.Admit_Type_ID = C.Admit_Type_ID
LEFT OUTER JOIN Care_Level_Source P ON
      F.Care_Level_ID = P.Care_Level_ID;
```

Our example Fact_Admissions_Source table only has four rows which result in the following Admissions_Info junk dimension. Note the Missing Description entry in row 4.

Admission_Info_Key	Admit_ Type_ID	Admit_Type_Descr	Care_Level_ID	Care_Level_Descr
1	1	Walk-In	1	ICU
2	2	Appointment	1	ICU
3	2	Appointment	2	Pediatric ICU
4	5	Missing Description	3	Medical Floor

Incorporate the Junk Dimension into the Fact Row Process

Once the junk dimension is in place, you will use it to look up the surrogate key that corresponds to the combination of attributes found in each fact table source row. Some of the ETL tools do not support a multi-column lookup join, so you may need to create a work-around. In SQL, the lookup query would be similar to the second set of code above, but it would join to the junk dimension and return the surrogate key rather than joining to the lookup tables.

Maintain the Junk Dimension

You will need to check for new combinations of attributes every time you load the dimension. You could apply the second set of SQL code to the incremental fact rows and select out only the new rows to be appended to the junk dimension as shown below.

```
SELECT * FROM ( {Select statement from second SQL code listing} ) TabA
WHERE TabA.Care_Level_Descr = 'Missing Description'
OR TabA.Admit_Type_Descr = 'Missing Description'  ;
```

In this example, it would select out row 4 in the junk dimension. Identifying new combinations could be done as part of the fact table surrogate key substitution process, or as a separate dimension processing step prior to the fact table process. In either case, your ETL system should raise a flag and notify the appropriate data steward if it identifies a missing entry.

There are a lot of variations on this approach depending on the size of your junk dimension, the sources you have, and the integrity of their data, but these examples should get you started. Send me an email if you'd like a copy of the code to create this example in SQL Server or Oracle.

www.kimballgroup.com Number 112, May 5, 2009

Design Tip #112 Creating Historical Dimension Rows

By Warren Thornthwaite

The business requirement for tracking dimension attribute changes over time is almost universal for DW/BI systems. Tracking attribute changes supports accurate causal analysis by associating the attribute values that were in effect when an event occurred with the event itself. You will need a set of ETL tasks to assign surrogate keys and manage the change tracking control fields for this Type 2 tracking of historical changes.

One big challenge many ETL developers face is recreating historical dimension change events during the initial historical data load. This is often a problem because the operational system may not retain history. In the worst case, attribute values are simply overwritten with no history at all. You need to carefully search the source systems for evidence of historical values. If you can’t find all the historical data, or the effort to recreate it is significant, you will need to discuss the implications of incomplete history with business folks. Once the decisions are made, you will need to load what you have and properly set the effective date and end date control columns. We’ll talk more about each of these three steps below.

Dig for History

First look at the source tables that directly feed your dimension along with any associated tables. You may find some change tracking system for audit or compliance purposes that isn’t obvious from the direct transaction source tables. We worked on one system that wrote the old customer row into a separate history table anytime a value changed. Once we found that table, it was fairly straightforward to recreate the historical dimension rows.

You may need to broaden your search for evidence of change. Any dates in the dimension’s source tables might be helpful. Your customer dimension source tables may have fields like registration date, cancel date, or contact date. Transaction events might also be helpful. Sometimes the shipment or customer service systems keep a copy of the customer address in their records each time a fact event occurs.

In the worst case, you may need to pull data from the backup system. Pulling daily backups for a set of tables for the last five years is usually not feasible. Explore the possibility of pulling one backup per month, so you will at least be able to identify changes within a 30 day window of when they actually occurred.

Discuss the Options and Implications

The business folks often don’t comprehend the implications of incomplete dimension history without careful explanation and detailed examples. You need to help them understand so they can help make informed decisions. In the worst case, you may not have any change history. Simply associating current dimension values with historical fact events is usually not acceptable from the business perspective. In a case like this, it’s not uncommon to only load data from the point where dimension history is available.

Build the Dimension

You may have to pull several different change events for a given dimension together from several sources. Combine all these rows into a single table and order them by the customer transaction key and the dimension change date which becomes the effective date of the row. The expiration date of each row will depend on the effective date of the next row. You can accomplish the expiration date assignment with a looping function in your language of choice. It's also possible to do it in your database with a cursor or other control structures. Don't worry too much about performance because this is a one-time effort.

Choose a Day-Grain or Minute-Second-Grain
For years we have recommended augmenting a Type 2 dimension with begin and end effective time stamps to allow for precise time slicing of history as described in this design tip. But there is a fussy detail that must be considered when creating these time stamps. You must decide if the dimension changes must be tracked on a daily grain (i.e., no more than one change per day) or on a minute-second grain where many changes could occur on a given day. In the first case, you can set the end expiration time stamp to be one day less than the time stamp of the next dimension change, and your BI queries can constrain a candidate date BETWEEN the begin and end time stamps. But if your dimension changes are to be tracked on a minute-second basis, you don't dare set the end time stamp to be "one tick" less than the next dimension change's time stamp because the "tick" is machine and DBMS dependent, and you could end up with gaps in your time sequence. In this case, you must set the end time stamp to be exactly equal to the time stamp of the next change, and forgo the use of the BETWEEN syntax in your BI queries in favor of GREATER-THAN and LESS-THAN-OR-EQUAL syntax.

Here is a set-based example to assign the expiration data for a day grain customer dimension table that uses the ROW_NUMBER function in Oracle to group all the rows for a given customer in the right order:

```
UPDATE Customer_Master T
          SET T.Exp_Date =
             (SELECT NVL(TabB.Real_Exp_Date, '31-DEC-9999')
 FROM
          (SELECT ROW_NUMBER() OVER(Partition by Source_Cust_ID
          Order by Eff_Date) AS RowNumA, Customer_Key, Source_Cust_ID,
          Eff_Date, Exp_Date
          FROM Customer_Master ) TabA  -- target row
LEFT OUTER JOIN
          (SELECT ROW_NUMBER() OVER(Partition by Source_Cust_ID
          Order by Eff_Date) AS RowNumB, Source_Cust_ID,
          Eff_Date -1  AS
            Real_Exp_Date, Exp_Date --  assumes day grain
          FROM Customer_Master) TabB   -- next row after the target row
ON TabA.RowNumA = TabB.RowNumB - 1
AND TabA. Source_Cust_ID = TabB. Source_Cust_ID
WHERE T.Customer_Key = TabA.Customer_Key);
```

Here is a set-based example of how to assign the expiration data for a minute-second grain customer dimension table that uses the ROW_NUMBER function in Oracle to group all the rows for a given customer in the right order. Note that date field references in the first example have been replaced with date-time field references below:

```
UPDATE Customer_Master T
          SET T.Exp_Date_Time =
             (SELECT NVL(TabB.Real_Exp_Date_Time, '31-DEC-9999 0:0:0')
 FROM
          (SELECT ROW_NUMBER() OVER(Partition by Source_Cust_ID
          Order by Eff_Date_Time) AS RowNumA, Customer_Key, Source_Cust_ID,
          Eff_Date_Time, Exp_Date_Time
          FROM Customer_Master ) TabA  -- The target row
LEFT OUTER JOIN
```

```
            (SELECT ROW_NUMBER() OVER(Partition by Source_Cust_ID
            Order by Eff_Date_Time) AS RowNumB, Source_Cust_ID,
            Eff_Date_Time AS
              Real_Exp_Date_Time, Exp_Date_Time --  assumes minute-second grain
            FROM Customer_Master) TabB   -- next row after the target row
ON TabA.RowNumA = TabB.RowNumB - 1
AND TabA. Source_Cust_ID = TabB. Source_Cust_ID
WHERE T.Customer_Key = TabA.Customer_Key);
```

This design tip gives you detailed guidance to instrument your slowly changing dimensions with precise time stamps during an initial historical load or routine batch load of large numbers of records.

www.kimballgroup.com Number 111, April 1, 2009

Design Tip #111 Is Agile Enterprise Data Warehousing an Oxymoron?

By Ralph Kimball

In today's economy the data warehouse team is caught between two conflicting pressures. First, we need more immediate, impactful results about our customers and our products and services across the enterprise. In other words, integrate the enterprise's data NOW! But second, we need to allocate our scarce people and money resources more wisely than ever before. In other words, make sure that all our designs are extensible and universal, and avoid any future rework. We can't waste a dime!

If we yield to the first pressure to deliver results too quickly, then we deliver one-off spot solutions that pretty quickly get us in trouble. For example, suppose we show the marketing department a prototype application written by one our developers that provides new demographic indicators on some of our existing customers. Suppose that the application is based on a server side implementation driven through a web interface that calls PHP to pull data from a MySQL database on a small development server. At the time of the demo to marketing, anyone can access the application via a browser if they know the URL. Furthermore the testing was done through Internet Explorer. It doesn't seem to work quite correctly under Firefox, but "we'll fix that soon." In spite of the fact that you got marketing's attention, you are asking for trouble. What's wrong with this picture? If you slow down, and count to ten, you will realize that you are about to have a "success disaster." Your application has little or no data quality, governance, security, or extensibility. It also doesn't help that the person who wrote the PHP program has left the company…

On the other hand, if we revert to the safe IT tradition of using a "waterfall" design where we propose a comprehensive system architecture, write a functional specification, evaluate vendors, and design an enterprise data model, we won't deliver useful results in time to help the business. If IT management is competent, we won't be allowed to start such a project!

Is there a middle ground that we might call an "agile enterprise data warehouse?" Remember that agile development calls for small teams, a succession of closely spaced deliverables, acceptance of a cut-and-try mentality, delivery of code rather than documentation, and intensive interaction throughout the project from end users who effectively control the project. For more on the "agile development" movement, go to Wikipedia, and read Margy's 2005 Design Tip #73.

If organized effectively and in the right cultural environment, perhaps any IT initiative can be a candidate for agile development, but let's not get carried away. Here's my recommendation for a very useful project that will yield measurable results quickly, and is designed not to be politically controversial. Best of all, this is a "stealth project" that when successful, will lay the foundation for an architectural revolution in your environment. The acronym for this stealth project is LWDS-MDM: Light Weight Down Stream Master Data Management!

In Design Tip #106, I described the typical downstream MDM function that exists within most data warehouse environments, given that few of us have a full fledged centralized MDM driving our operational systems. The downstream MDM gathers incompatible descriptions of an entity such as Customer and publishes the cleaned, conformed, and deduplicated dimension to the rest of the data warehouse community. The subscribers to this dimension are almost always owners of fact tables who want to attach this high quality conformed dimension to their fact tables so that BI tools around the enterprise can do drill across BI reports and dashboards on the conformed contents of the dimension.

Your job is to assemble a small team of developers and end users to start building the conformed Customer dimension. This is an ideal target for incremental, adaptive development. Remember that the essential content of a conformed dimension is a small set of “enterprise attributes” that have a consistent interpretation across all the members of the dimension and can be used for constraining and grouping by all forms of BI. Furthermore, the agile development team can start by cleaning and conforming Customer records from a limited set of original sources around the enterprise. Don’t try to design the cosmic solution for Customer metadata! Instead, concentrate on making the first set of attributes and sources produce a usable result in two to four weeks! If you read about agile development, you will be shocked and amazed that you are expected to give a working demo of your system to the end users at the end of every such two to four week sprint!

Hopefully you see how this stealth project should be able to grow to become really powerful. Over time, each sprint adds more attributes and more sources. You are chipping away at the big issues of enterprise data integration, including an understanding of user requirements, data quality, and the need for data governance.

Write to me if you go down this path. Let’s take the agile approach away from the pure methodologists who really don’t understand data warehousing, and let’s show the world a real “use case.”

www.kimballgroup.com Number 110, March 4, 2008

Design Tip #110 Business Requirements Gathering Dos and Don'ts

By Margy Ross

Successful data warehouse and business intelligence solutions provide value by helping the business identify opportunities or address challenges. Obviously, it's risky business for the DW/BI team to attempt delivering on this promise without understanding the business and its requirements. This Design Tip covers basic guidelines for effectively determining the business's wants and needs.

First, start by properly preparing for the requirements process:

- Do recruit a lead requirements analyst with the right stuff, including confidence and strong communication and listening skills, coupled with the right knowledge, such as business acumen and vision for BI's potential.
- Do involve the right business representatives representing both vertical (cross management ranks) and horizontal (cross department) perspectives. Don't rely solely on the input from a lone power analyst who's a pseudo-IT professional.
- Don't presume you already know what the business wants because you've supported them for the past decade.
- Don't accept a static requirements document from the business which lists general items such as "we need the ability to quickly drill down into a single version of integrated data."
- Do differentiate between project and program requirements. While the mechanics are similar, there are differences in terms of the participants, breadth and depth of details collected, documentation, and next steps.
- Do get face-to-face with business representatives (or at least voice-to-voice). Don't rely on non-interactive static surveys or questionnaires; don't think producing a 3" binder of existing reports equates to requirements analysis.
- Do prepare interviewees in advance so you don't spend the first 30 minutes of every session conceptually explaining DW/BI.

Secondly, spend your time with the business representatives wisely:

- Do show up at the appointed meeting time. Don't allow your cell phone to ring (or casually try to figure out why it's vibrating).
- Do ask "what do you do (and why)"; don't ask "what do you want" in the data warehouse or pull out a list of data elements to determine what's needed.
- Do ask unbiased open-ended why, how, what if, and what then questions. Do respect the lead interviewer's or facilitator's role in steering the sessions; don't turn it into a questioning free-for-all.
- Do strive for a conversation flow; don't dive into the detailed weeds too quickly.
- Do listen intently to absorb like a sponge; don't allow the requirements team to do too much talking.
- Do conduct some sessions with source system experts to begin understanding the underlying data realities.
- Don't schedule more than four requirements sessions in a day; do allow time between sessions to debrief.
- Do ask the business representatives for measurable success criteria (before thanking them for their brilliant insights).
- Do set reasonable expectations with the business representatives about next steps; don't leave it to their imaginations.

Finally, bring the requirements gathering activities to an orderly conclusion:

- Do synthesize the findings into business process-centric categories. Do review Design Tip #69 to help identify business processes representing manageable units of design and development effort for the DW/BI team.
- Do write down what you heard (along with who said it and why it's required), but don't attempt to reinvent an alternative Dewey Decimal System for classification and cross-footing. Detailed project requirements can often be classified into three categories (data, BI application/report delivery, and technical architecture requirements) which correspond to downstream Lifecycle tasks.
- Don't allow the requirements gathering activities to be overcome by analysis paralysis or process perfection.
- Do present your findings to senior executives to ensure a common understanding, reach consensus about an overall roadmap, and establish next step priorities based on business value and feasibility.

Focusing on business requirements has always been a cornerstone of the Kimball approach; I first delivered a presentation on the topic at an industry conference in 1994. Much has changed in our industry over the past fifteen years, but the importance of effective requirements gathering has remained steadfast. Good luck with your requirements gathering initiative!

www.kimballgroup.com Number 109, February 3, 2008

Design Tip #109 Dos and Don'ts on the Kimball Forum

By Joy Mundy

You've asked (and asked, and asked), and we're finally doing it: we're launching the Kimball Forum! It has gone live already; you can access it from a link right on the Kimball Group's home page. We hope the Kimball Forum will foster a community of dimensional data warehouse practitioners. Many of you have years of experience in business intelligence, and we hope you'll find this a good opportunity to help your colleagues. We at Kimball Group will monitor the forum, we'll chime in sometimes, but we're hoping to develop a vibrant community rather than one that we dominate.

When you first connect to the forum, please take a moment to review the rules. They are summarized here:

- **Keep the discussion polite and civil.** We would rather not delete anyone's posts.
- **Be succinct.**
- **Make an effort to communicate clearly.** If no one can understand your question, no one will answer it.
- **Do not advertise.** That would really annoy us.
- **Don't beg for hand-holding.** This forum is for discussion and analysis, not for answering every question you might dream up. Make an effort to review existing content, both in the Forum and elsewhere on the Kimball website, before posting your question.

Just to get you started on the right foot, here are examples of good questions:

- I am the DW team lead for my company, with a Kimball DW/BI system that's been in production for a few years. We just got a new CIO, who's hired a team of consultants to review all aspects of IT, including the data warehouse. They say our design is inflexible; they say we need an ODS and normalized data warehouse to feed the user-oriented data marts. Can anyone share their experiences to help me argue my position?
- I have a dimension with many alternative rollups or hierarchies. By "many" I mean 20 or so. These rollups are modified, added to, and discontinued fairly frequently (maybe 10 times a year). Although I could make columns in the dimension table for each hierarchy, that design creates problems for ETL. Is there an alternative approach? Although I'm primarily interested in solving this problem for reporting from the RDBMS dimensional data warehouse, I do have some users on OLAP platforms.
- My current business requirements specify that some dimension attributes are Type 1 and some are Type 2. I'm worried about building the system as currently specified, *not* capturing history for Type 1 attributes, and then sometime in the future seeing requirements change. How have others addressed this problem?

Primarily for entertainment purposes, here are some examples of bad questions:

- Attached is the data model for my data warehouse. Can you provide suggestions for improvement?
- What's the difference between SCD-1 and SCD-2?

- I am a master's student, writing my thesis on data warehousing. Can you tell me about it?

Finally, we want to publicly thank Kimball University alum, Brian Jarrett, for initially conceiving and establishing a forum framework that Warren Thornthwaite leveraged for the Kimball Forum. Brian got the ball rolling; Warren kept it in motion.

We hope to see you on the Forum!

www.kimballgroup.com Number 108, December 3, 2008

Design Tip #108 When is the Dimensional Model Design Done?

By Bob Becker

There is a tendency for data warehouse project teams to jump immediately into implementation tasks as the dimensional data model design is finalized. But we'd like to remind you that you're not quite done when you think you might be. The last major design activity that needs to be completed is a review and validation of the dimensional data model before moving forward with implementation activities.

We suggest you engage in a review and validation process with several successive audiences, each with different levels of technical expertise and business understanding. The goal is to solicit feedback from interested people across the organization. The entire DW/BI team benefits from these reviews in the form of a more informed and engaged business user community. At a minimum, the design team should plan on talking to three groups:

- Source system developers and DBAs who can often spot errors in the model very quickly
- Core business or power users who were not directly involved in the model development process
- Broader user community.

Typically the first public design review of the detailed dimensional model is with your peers in the IT organization. This audience is often comprised of reviewers who are intimately familiar with the target business process because they wrote or manage the system that runs it. Likely, they are partly familiar with the target data model because you've already been pestering them with source data questions.

The IT review can be challenging because the reviewers often lack an understanding of dimensional modeling. In fact, most of them probably fancy themselves as pretty proficient third normal form modelers. Their tendency will be to apply transaction processing-oriented modeling rules to the dimensional model. Rather than spending the bulk of your time debating the merits of different modeling disciplines, it is best to be prepared to provide some dimensional modeling education as part of the review process.

When everyone has the basic dimensional modeling concepts down, begin with a review of the bus matrix. This will give everyone a sense for the project scope and overall data architecture, demonstrate the role of conformed dimensions, and show the relative business process priorities. Next, illustrate how the selected row on the matrix translates directly into the dimensional model. Most of the IT review session should then be spent going through each individual dimension and fact table.

Often, the core business users are members of the modeling team and are already intimately knowledgeable about the data model, so a review session with them is not required. However, if they have not been involved in the modeling process, a similar, detailed design review should be performed with the core business users. The core users are more technical than typical business users and can handle more detail about the model. Often, especially in smaller organizations, you can combine the IT review and core user review into one session. This works if the participants already know each other well and work together on a regular basis.

Finally, the dimensional data model should be shared with the broader community of business users. Often, this is a very large audience. In such cases a representative subsection of the users can be selected. This session is as much education as it is design review. You want to educate people without overwhelming them, while at the same time illustrating how the dimensional model supports their business requirements. In addition, you want them to think closely about how they will use the data so they can help highlight any shortcomings in the model.

Create a presentation that starts with basic dimensional concepts and definitions, and then describe the bus matrix as your enterprise DW/BI data roadmap. Review the high level model, and finally, review the important dimensions, like customer and product.

Allocate about a third of the time to illustrate how the model can be used to answer a broad range of questions about the business process. Pull some interesting examples from the requirements documentation and walk through how they would be answered. More analytical users will get this immediately. Reassure the rest of your audience that most of this complexity will be hidden behind a set of easy-to-use structured reports. The point is to show you can answer just about every question they might ask about this business process.

There are usually only minor adjustments to the model once you get to this point. After working so hard to develop the model, the users may not show what you consider to be appropriate enthusiasm. The model may seem obvious to the users and makes sense; after all, it is a reflection of their business. This is a good thing; it means you have done your job well!

KIMBALL GROUP KIMBALL UNIVERSITY

www.kimballgroup.com Number 107, November 6, 2008

Design Tip #107 Using the SQL MERGE Statement for Slowly Changing Dimension Processing

By Warren Thornthwaite

Most ETL tools provide some functionality for handling slowly changing dimensions. Every so often, when the tool isn't performing as needed, the ETL developer will use the database to identify new and changed rows, and apply the appropriate inserts and updates. I've shown examples of this code in the Data Warehouse Lifecycle in Depth class using standard INSERT and UPDATE statements. A few months ago, my friend Stuart Ozer suggested the new MERGE command in SQL Server 2008 might be more efficient, both from a code and an execution perspective. His reference to a blog by Chad Boyd on MSSQLTips.com gave me some pointers on how it works. MERGE is a combination INSERT, UPDATE and DELETE that provides significant control over what happens in each clause.

This example handles a simple customer dimension with two attributes: first name and last name. We are going to treat first name as a Type 1 and last name as a Type 2. Remember, Type 1 is where we handle a change in a dimension attribute by overwriting the old value with the new value; Type 2 is where we track history by adding a new row that becomes effective when the new value appears.

Step 1: Overwrite the Type 1 Changes

I tried to get the entire example working in a single MERGE statement, but the function is deterministic and only allows one update statement, so I had to use a separate MERGE for the Type 1 updates. This could also be handled with an update statement since Type 1 is an update by definition.

```
MERGE INTO  dbo.Customer_Master AS CM
USING Customer_Source AS CS
ON (CM.Source_Cust_ID = CS.Source_Cust_ID)
WHEN MATCHED AND -- Update all existing rows for Type 1 changes
        CM.First_Name <> CS.First_Name
        THEN UPDATE SET CM.First_Name = CS.First_Name
```

This is a simple version of the MERGE syntax that says merge the Customer_Source table into the Customer_Master dimension by joining on the business key, and update all matched rows where First_Name in the master table does not equal the First_Name in the source table.

Step 2: Handle the Type 2 Changes

Now we'll do a second MERGE statement to handle the Type 2 changes.
This is where things get a little tricky because there are several steps involved in tracking Type 2 changes. Our code will need to:

1. Insert brand new customer rows with the appropriate effective and end dates
2. Expire the old rows for those rows that have a Type 2 attribute change by setting the appropriate end date and current_row flag = 'n'
3. Insert the changed Type 2 rows with the appropriate effective and end dates and current_row

flag = ‘y’

The problem with this is it’s one too many steps for the MERGE syntax to handle. Fortunately, the MERGE can stream its output to a subsequent process. We’ll use this to do the final insert of the changed Type 2 rows by INSERTing into the Customer_Master table using a SELECT from the MERGE results. This sounds like a convoluted way around the problem, but it has the advantage of only needing to find the Type 2 changed rows once, and then using them multiple times.

The code starts with the outer INSERT and SELECT clause to handle the changed row inserts at the end of the MERGE statement. This has to come first because the MERGE is nested inside the INSERT. The code includes several references to getdate; the code presumes the change was effective yesterday (getdate()-1) which means the prior version would be expired the day before (getdate()-2). Finally, following the code, there are comments that refer to the line numbers

```
1     INSERT INTO Customer_Master
2     SELECT Source_Cust_ID, First_Name, Last_Name, Eff_Date, End_Date, Current_Flag
3     FROM
4         (  MERGE Customer_Master    CM
5              USING Customer_Source    CS
6             ON (CM.Source_Cust_ID = CS.Source_Cust_ID)
7            WHEN NOT MATCHED THEN
8                INSERT VALUES (CS.Source_Cust_ID, CS.First_Name, CS.Last_Name,
               convert(char(10), getdate()-1, 101), '12/31/2199', 'y')
9          WHEN MATCHED AND CM.Current_Flag = 'y'
10               AND   (CM.Last_Name <> CS.Last_Name )  THEN
11                    UPDATE SET CM.Current_Flag = 'n', CM.End_date = convert(char(10), getdate()-
                   2, 101)
12         OUTPUT $Action Action_Out, CS.Source_Cust_ID, CS.First_Name, CS.Last_Name,
         convert(char(10), getdate()-1, 101) Eff_Date, '12/31/2199' End_Date, 'y'Current_Flag
13         ) AS MERGE_OUT
14    WHERE  MERGE_OUT.Action_Out = 'UPDATE';
```

Code Comments

Lines 1-3 set up a typical INSERT statement. What we will end up inserting are the new values of the Type 2 rows that have changed.

Line 4 is the beginning of the MERGE statement which ends at line 13. The MERGE statement has an OUTPUT clause that will stream the results of the MERGE out to the calling function. This syntax defines a common table expression, essentially a temporary table in the FROM clause, called MERGE_OUT.

Lines 4-6 instruct the MERGE to load Customer_Source data into the Customer_Master dimension table.

Line 7 says when there is no match on the business key, we must have a new customer, so Line 8 does the INSERT. You could parameterize the effective date instead of assuming yesterday’s date.

Lines 9 and 10 identify a subset of the rows with matching business keys, specifically, where it’s the current row in the Customer_Master AND any one of the Type 2 columns is different.

Line 11 expires the old current row in the Customer_Master by setting the end date and current row flag to ‘n’.

Line 12 is the OUTPUT clause which identifies what attributes will be output from the MERGE, if any. This is what will feed into the outer INSERT statement. The $Action is a MERGE function that tells us what part of the MERGE each row came from. Note that the OUTPUT can draw from both the source and the master. In this case, we are outputting source attributes because they contain the new Type 2 values.

Line 14 limits the output row set to only the rows that were updated in Customer_Master. These correspond to the expired rows in Line 11, but we output the current values from Customer_Source in Line 12.

Summary
The big advantage of the MERGE statement is being able to handle multiple actions in a single pass of the data sets, rather than requiring multiple passes with separate inserts and updates. A well tuned optimizer could handle this extremely efficiently.

Design Tip #106 Can the Data Warehouse Benefit from SOA?

By Ralph Kimball

The Service Oriented Architecture (SOA) movement has captured the imagination, if not the budgets, of many IT departments. In a nutshell, organizing your environment around SOA means identifying reusable services, and implementing these services as centralized resources typically accessed over the web. The appeal comes from the promised cost savings of implementing a service only once in a large organization, and making the service independent of specific hardware and operating system platforms because all communications take place via a neutral communications protocol, most often WSDL-SOAP-XML.

Well, pretty much all of these advantages of SOA can be realized, but early SOA pioneers have learned some valuable lessons that give one pause for thought. The names of these lessons are data quality, data integration, and governance. To make long story short, SOA initiatives fail 1) when they sit on a platform of poor quality data, 2) attempt to share data that is not integrated across the enterprise, and 3) are implemented with insufficient thought given to security, compliance, and change management. SOA architects have also learned to "avoid the overly detailed use case." Services meet the goals of SOA architecture when they are simple, conservative in their scope, and not dependent on complex business rules of an underlying application.

So, does SOA have anything to offer data warehousing? Can we identify "abstract services" in the data warehouse world, commonly recognized, simply stated, and independent of specific data sources, business processes, and BI deployments? I think we can.

Consider the relationship between the dimension manager and the fact provider. Remember that a dimension manager is the centralized resource for defining and publishing a conformed dimension to the rest of the enterprise. A master data management (MDM) resource is an ideal dimension manager but few of us are lucky enough to have a functioning MDM resource. More likely, the data warehouse team is a kind of "downstream MDM" function that gathers incompatible descriptions of an entity such as Customer and publishes the cleaned, conformed, and deduplicated dimension to the rest of the data warehouse community. The subscribers to this dimension are almost always owners of fact tables who want to attach this high quality conformed dimension to their fact tables so that BI tools around the enterprise can perform drill across reports on the conformed contents of the dimension. If the vocabulary of this paragraph is unfamiliar, then you have some reading to do! See the *Data Warehouse Lifecycle Toolkit, 2nd Edition* book, or the recent series of introductory articles I have written in the last year for *DMReview*.

Every dimension manager publisher needs to provide the following services to their fact table subscribers. *Fetch* means that fact table provider pulls information from the dimension manager, and *Alert* means that the dimension manager pushes information to the fact providers.

- Fetch specific dimension member (we assume in this and the following steps that the dimension record has a surrogate primary key, a dimension version number, and that the information transmitted is consistent with the security and privacy privileges of the requester).

- Fetch all dimension members.
- Fetch dimension members changed since specific date-time with designated SCD Types 1, 2, and 3.
- Fetch natural key to surrogate key correspondence table unique to fact table provider.
- Alert providers of new dimension release. (A major dimension release requires providers to update their dimensions because Type 1 or Type 3 changes have been made to selected attributes.)
- Alert providers to late arriving dimension members. (Requires fact table provider to overwrite selected foreign keys in fact tables).

These services are generic to all integrated data warehouses. In a dimensionally modeled data warehouse, we can describe the administrative processing steps with great specificity, without regard to the underlying subject matter of the fact or dimension tables. That is why, in SOA parlance, dimensional modeling provides a well-defined reference architecture on which to base these services.

These services seem pretty defensible as meeting SOA design requirements. An interesting question is whether a similar set of abstract services could be defined between the fact provider and the BI client. How about "Alert client to KPI change?" Maybe this is possible. I'll follow this design tip with another if this approach pans out. In the meantime, I'd suggest reading *Applied SOA: Service-Oriented Architecture and Design Strategies* (Rosen, et al, Wiley 2008).

www.kimballgroup.com Number 105, September 3, 2008

Design Tip #105 Snowflakes, Outriggers, and Bridges

By Margy Ross

Students often blur the concepts of snowflakes, outriggers, and bridges. In this Design Tip, we'll try to reduce the confusion surrounding these embellishments to the standard dimensional model.

When a dimension table is snowflaked, the redundant many-to-one attributes are removed into separate dimension tables. For example, instead of collapsing hierarchical rollups such as brand and category into columns of a product dimension table, the attributes are stored in separate brand and category tables which are then linked to the product table. With snowflakes, the dimension tables are normalized to third normal form. A standard dimensional model often has 10 to 20 denormalized dimension tables surrounding the fact table in a single layer halo; this exact same data might easily be represented by 100 or more linked dimension tables in a snowflake schema.

We generally encourage you to handle many-to-one hierarchical relationships in a single dimension table rather than snowflaking. Snowflakes may appear optimal to an experienced OLTP data modeler, but they're suboptimal for DW/BI query performance. The linked snowflaked tables create complexity and confusion for users directly exposed to the table structures; even if users are buffered from the tables, snowflaking increases complexity for the optimizer which must link hundreds of tables together to resolve queries. Snowflakes also put burden on the ETL system to manage the keys linking the normalized tables which can become grossly complex when the linked hierarchical relationships are subject to change. While snowflaking may save some space by replacing repeated text strings with codes, the savings are negligible, especially in light of the price paid for the extra ETL burden and query complexity.

Outriggers are similar to snowflakes in that they're used for many-to-one relationships, however they're more limited. Outriggers are dimension tables joined to other dimension tables, but they're just one more layer removed from the fact table, rather than being fully normalized snowflakes. Outriggers are most frequently used when one standard dimension table is referenced in another dimension, such as a hire date attribute in the employee dimension table. If the users want to slice-and-dice the hire date by non-standard calendar attributes, such as the fiscal year, then a date dimension table (with unique column labels such as Hire Date Fiscal Year) could serve as an outrigger to the employee dimension table joined on a date key.

Like many things in life, outriggers are acceptable in moderation, but they should be viewed as the exception rather than the rule. If outriggers are rampant in your dimensional model, it's time to return to the drawing board given the potentially negative impact on ease-of-use and query performance.

Bridge tables are used in two more complicated scenarios. The first is where a many-to-many relationship can't be resolved in the fact table itself (where M:M relationships are normally handled) because a single fact measurement is associated with multiple occurrences of a dimension, such as multiple customers associated with a single bank account balance. Placing a customer dimension key in the fact table would require the unnatural and unreasonable divvying of the balance amongst multiple customers, so a bridge table with dual keys to capture the many-to-many relationship between customers and accounts is used in conjunction with the measurement fact table. Bridge tables are also used to represent a ragged or variable depth hierarchical relationship which cannot be reasonably forced into a simpler fixed depth hierarchy of many-to-one attributes in a dimension table.

In these isolated situations, the bridge table comes to the rescue, albeit at a price. Sometimes bridges are used to capture the complete data relationships, but pseudo compromises, such as including the primary account holder or top rollup level as dimension attributes, help avoid paying the toll for navigating the bridge on every query.

Hopefully, this Design Tip helps clarify the differences between snowflakes, outriggers, and bridges. As you might imagine, we’ve written extensively on these topics in our *Toolkit* books.

www.kimballgroup.com Number 104, August 7, 2008

Design Tip #104 Upgrading your BI Architecture

By Joy Mundy

The Kimball Lifecycle Methodology describes how to build a rich business intelligence environment to support:

a) Publish standard reporting and scorecards: How's my business?

b) Identify exceptions: What's unusually good or bad?

c) Determine causal factors: Why did something go well or poorly? This step is particularly challenging because these new queries may require new data sources.

d) Model predictive or what-if analysis: How will business look next year?

e) Track actions: What's the impact of the decisions that were made?

Our methodology describes how to start from scratch to build this infrastructure iteratively, from project planning, requirements gathering, architecture, and design through implementation, deployment, and operations.

What do you do if you're stuck at Level (a)? What if you have an infrastructure that supports basic reporting, but is the wrong architecture to enable complex analytics or business user self-service? How do you get to where you want to go? In some ways it's easier to start from a blank slate and do it right the first time; it's easy to be a hero when you're starting from zero. But large companies, and a growing number of medium and even small companies, already have some kind of business intelligence in place. There are additional challenges in moving to a new architecture at the same time you have to maintain the existing system and users.

There are three common unsuccessful BI architectures:

- *Normalized data warehouse with no user-focused delivery layer.* The organization has invested in a BI architecture, but stopped short of the business users. The data warehouse is normalized, which means it's simpler to load and maintain, but not easy to query. Reports are written directly on the normalized structures, and often require very complex queries and stored procedures. In most cases, only a professional IT team can write reports.

- *Normalized data warehouse with mart proliferation.* A common approach to solving the problem of data model complexity is to spin off a data mart to solve a specific business problem. Usually these marts are dimensional (or at least can pass as dimensional in the dark with your glasses off). Unfortunately, they are limited in scope, contain only summary data, and are unarchitected. A new business problem requires a new mart. The better the underlying data warehouse, the easier it is to spin up a new mart. User's ad hoc access is limited to the scenarios that have been cooked into the standalone mart.

- *Mart proliferation directly from transaction systems.* The least effective architecture is to build data marts directly from OLTP systems, without an intermediate DW layer. Each mart has to develop complex ETL processes. Often, we see marts chained together as one mart feeds the next.

In any case, the appropriate solution is to build a conformed, dimensional data delivery area. Gather business requirements and build the logical model for a Kimball conformed dimensional data warehouse.

If you already have a normalized enterprise data warehouse, analyze the gap between the business requirements and the DW contents. You might be able to build relatively simple ETL processes to populate the dimensional DW from the normalized one. For any new business processes and data, determine whether the normalized DW provides value in your environment. If so, continue to integrate and store data there, then dimensionalize and store it again in the dimensional structure. Alternatively, you may find that it makes more sense to integrate and dimensionalize in one ETL process, and phase out the normalized DW. Once the data is in the conformed dimensional model, you'll find that business users have much greater success self-servicing and developing ad hoc queries. Some of those ad hoc queries will push up into exception, causal, and even predictive analysis, and will evolve into BI applications for the broader audience.

If you don't have a normalized data warehouse in place, you probably won't build one. This scenario is more like the "starting from scratch" approach using the Kimball Method. You'll need to gather business requirements, design the dimensional model, and develop the extract, transformation, dimensionalization, and loading logic for the enterprise dimensional data warehouse.

Arguably the biggest challenge in building an upgraded architecture is that your users' expectations are higher. You'll need to keep the existing environment in place and provide modest improvements while the new system is being developed. If you're starting from scratch, you can make users happy by rolling out the new system a little bit at a time. With a BI upgrade or replacement project, your Phase 1 scope is likely going to have to be bigger than we normally recommend to make a splash.

You need to plan for people and resources to maintain the existing environment as well as to perform the new development. We recommend that you devote a team to the new development; if the same people are trying to do the old and the new, they'll find their energies sucked into the constant operational demands of the user community. The entire group will have to expand, and the old team and new team both need business expertise and technical skills.

Once you roll out a core set of data in the upgraded environment, there are two paths you can take. You can go deeper into the initial set of data by building analytic applications that go beyond just publishing basic reports. Or, you can bring in data from additional business processes. With enough resources, you can do both at the same time.

www.kimballgroup.com Number 103, July 1, 2008

Design Tip #103 Staffing the Dimensional Modeling Team

By Bob Becker

It's surprising the number of DW/BI teams that confine the responsibility for designing dimensional models to a single data modeler or perhaps a small team of dedicated data modelers. This is clearly shortsighted. The best dimensional models result from a collaborative team effort. No single individual is likely to have the detailed knowledge of the business requirements and the idiosyncrasies of all the source systems to effectively create the model themselves.

In the most effective teams, a core modeling team of two or three people does most of the detailed work with help from an extended team. The core modeling team is led by a data modeler with strong dimensional modeling skills and excellent facilitation skills. The core team should also include a business analyst who brings a solid understanding of the business requirements, the types of analysis to be supported, and an appreciation for making data more useful and accessible. Ideally, the core team will include at least one representative from the ETL team with extensive source systems development experience and an interest in learning. The data modeler has overall responsibility for creating the dimensional model.

We recommend including one or more analytic business users as part of the core modeling team. These power users add significant insight, helping speed the design process and improving the richness and completeness of the end result. These are the users who have figured out how to get data out of the source system and turn it into information. They typically know how to build their own queries, sometimes even creating their own private databases. They are particularly valuable to the modeling process because they understand the source systems from a business point of view, and they've already identified and created the business rules needed to convert the data from its form in the source system to something that can be used to support the decision making process.

The core modeling team needs to work closely with source system developers to understand the contents, meaning, business rules, timing, and other intricacies of the source systems involved in populating the dimensional model. If you're lucky, the people who actually built or originally designed the source systems are still around. For any given dimensional model, there are usually several source system people you need to pull into the modeling process. There might be a DBA, a developer, and someone who works with the data input process. Each of these folks does things to the data that the other two don't know about.

The DBA implementing the physical database, the ETL architect/developer and the BI architect/developer should also be included in the modeling process. Being actively engaged in the design process will help these individuals better understand the business reasons behind the model and facilitate their buy-in to the final design. Often the DBA comes from a transaction system background and may not understand the rationale for dimensional modeling. The DBA may naturally want to model the data using more familiar design rules and apply third normal form design concepts to normalize the dimensions, physically defeating your dimensional design. ETL designers often have a similar tendency. Without a solid appreciation of the business requirements and justification for the dimensional design, the ETL designer will want to streamline the ETL process by shifting responsibility for calculations to the BI tool, skipping a description lookup step, or taking other shortcuts. Though these changes may save ETL development time, the tradeoff may be an increase in effort or decrease in query performance for hundreds of business users. BI designers can often provide important input into the models that improve the effectiveness of the final BI applications.

Before jumping into the modeling process, take time to consider the ongoing management and stewardship implications of the DW/BI environment. If your organization has an active data stewardship initiative, it is time to tap into that function. If there is no stewardship program, it's time to initiate the process. An enterprise DW/BI effort committed to dimensional modeling as an implementation approach must also be committed to a conformed dimension strategy to assure consistency across multiple business processes. An active data stewardship program can help an organization achieve its conformed dimension strategy.

Although involving more people in the design process increases the risk of slowing down the process, the improved richness and completeness of the design is well worth the additional overhead.

www.kimballgroup.com Number 102, June 3, 2008

Design Tip #102 Server Configuration Considerations

By Warren Thornthwaite

“How many servers do I need?” is a frequently asked question when technology is discussed in our Kimball University classes. The only right answer is the classic "it depends." While this may be true, it’s more helpful to identify the factors upon which it depends. In this Design Tip, I briefly describe these factors, along with common box configurations.

Factors Influencing Server Configurations
Not surprisingly, the three major layers of the DW/BI system architecture (ETL, presentation server, and BI applications) are the primary drivers of scale. Any one of these may need more horsepower than usual depending on your circumstances. For example, your ETL system may be particularly complex with lots of data quality issues, surrogate key management, change data detection, data integration, low latency data requirements, narrow load windows, or even just large data volumes.

At the presentation server layer, large data volumes and query usage are the primary drivers of increased scale. This includes the creation and management of aggregates which can lead to a completely separate presentation server component like an OLAP server. At the query level, factors such as the number of concurrent queries and the query mix can have a big impact. If your typical workload includes complex queries that require leaf level detail, like COUNT(DISTINCT X), or full table scans such as the queries required to create a data mining case set, you will need much more horsepower.

The BI applications are also a major force in determining the scale of your system. In addition to the queries themselves, many BI applications require additional services including enterprise report execution and distribution, web and portal services, and operational BI. The service level requirements also have an impact on your server strategy. If it’s OK to lock out user queries during your load window, you can apply most of your resources to the ETL system when it needs it, and then shift them over to the database and user queries once the load is finished.

Adding Capacity
There are two main approaches to scaling a system. Separating out the various components onto their own boxes is known as *scaling out*. Keeping the components on a single system and adding capacity via more CPUs, memory and disk is known as *scaling up*.

Start with the basic design principle that fewer boxes is better, as long as they meet the need from a data load, query performance and service level perspective. If you can keep your DW/BI system all on a single box, it is usually much easier to manage. Unfortunately, the components often don’t get along. The first step in scaling is usually to split the core components out onto their own servers. Depending on where your bottlenecks are likely to be, this could mean adding a separate ETL server or BI application server, or both. Adding servers is usually cheaper up front than a single large server, but it requires more work to manage, and it is more difficult to reallocate resources as needs change. In the scale up scenario, it is possible to dedicate a portion of a large server to the ETL system, for example, and then reallocate that portion on the fly. At the high end, you can actually define independent servers that run on the same machine, essentially a combination of scaling up and scaling out.

Clustering and server farms offer ways to add more servers in support of a single component. These techniques usually become necessary only at the top end of the size range. Different database and

BI products approach this multi-server expansion in different ways, so the decision on when/if is product dependent.

Just to make it more interesting, your server strategy is also tightly integrated with your data storage strategy. The primary goal in designing the storage subsystem is to balance capacity from the disks through the controllers and interfaces to the CPUs in a way that eliminates any bottlenecks. You don't want your CPUs waiting idly for disk reads or writes.

Finally, once you get your production server strategy in place, you need to layer on the additional requirements for development and test environments. Development servers can be smaller scale, however, in the ideal world, your test environment should be exactly the same as your production system. We have heard from folks in the Kimball community who have had some success using virtual servers for functional testing, but not so much with performance testing.

Getting Help
Most of the major vendors have configuration tools and reference systems designed specifically for DW/BI systems based on the factors we described above. They also have technical folks who have experience in DW/BI system configuration and understand how to create a system that is balanced across the various factors. If you are buying a large system, they also offer test labs where they can set up full scale systems to test your own data and workloads for a few weeks. Search your vendor's website for "data warehouse reference configuration" as a starting point.

Conclusion
Hardware has advanced rapidly in the last decade to the point that many smaller DW/BI systems can purchase a server powerful enough to meet all their needs for relatively little money. The rest of you will still need to do some careful calculations and testing to make sure you build a system that will meet the business requirements.

This design tip focuses on the tangible decision of how much hardware to buy, and how to configure that hardware for the widely varying requirements of ETL, database querying and BI. This is an exciting time to make hardware decisions because of the explosive growth of server virtualization and the eventual promise of cloud computing. However, on reflection it is clear that neither of these approaches makes the problem of hardware configuration go away. DW/BI systems are so resource intensive, with highly specific and idiosyncratic demands for disk storage, CPU power, and communications bandwidth, that you cannot "virtualize it and forget it." Certainly, virtualization is appropriate in certain cases, such as flexing to meet increased demands for service. But those virtual servers still have to sit on real hardware that is configured with the right capacities.

Cloud computing is an exotic possibility that may one day change the DW/BI landscape. But we are still in the "wild ideas" stage of using cloud computing. Only the earliest adopters are experimenting with this new paradigm, and as an industry, we haven't learned how to leverage it yet. Thus far, a few examples of improved query performance on large databases have been demonstrated, but this is just a piece of the DW/BI pie.

Design Tip #101 Slowly Changing Vocabulary

By Margy Ross

Ralph's first article on data warehousing appeared in 1995. During the subsequent 13 years, we've written hundreds of articles and Design Tips, as well as published seven books. Remarkably, the concepts that Ralph introduced in the 1990s have withstood the test of time and remain relevant today. However, some of our vocabulary has evolved slightly over the years. This became readily apparent when we were working on the 2nd edition of *The Data Warehouse Lifecycle Toolkit* which was released in January 2008.

Data Warehouse vs. Business Intelligence

Traditionally, the Kimball Group has referred to the overall process of providing information to support business decision making as *data warehousing*. Delivering the end-to-end solution, from the source extracts to the queries and applications that the business users interact with, has always been one of our fundamental principles; we wouldn't consider building data warehouse databases without delivering the presentation and access capabilities. This terminology is strongly tied to our written legacy; nearly all our *Toolkit* books include references to the data warehouse in their titles.

The term *business intelligence* emerged in the 1990s to refer to the reporting and analysis of data stored in the warehouse. Some misguided organizations had built data warehouses as archival repositories without regard to getting the data out and usefully delivered to the business. Not surprisingly, these data warehouses had failed and people were excited about BI to deliver on the promise of business value.

Some folks continue to refer to *data warehousing* as the overall umbrella term, with the data warehouse databases and BI layers as subset deliverables within that context. Others refer to *business intelligence* as the overarching term, with the data warehouse as the central data store foundation of the overall business intelligence environment.

Because the industry cannot reach agreement, we have been using the phrase *data warehouse/business intelligence (DW/BI)* to mean the complete end-to-end system. Though some would argue that you can theoretically deliver BI without a data warehouse, and vice versa, we believe that is ill-advised. Linking the two in the *DW/BI* acronym reinforces their dependency.

Independently, we refer to the queryable data in your DW/BI system as the *enterprise's data warehouse*, and value-add analytics as *BI applications*. In other words, the *data warehouse is the foundation for business intelligence*.

Data Staging → ETL System

We often refer to the extract, transformation, and load (ETL) system as the back room kitchen of the DW/BI environment. In a commercial restaurant's kitchen, raw materials are dropped off at the back door and transformed into a delectable meal for the restaurant patrons by talented chefs. Much the same holds true for the DW/BI kitchen: raw data is extracted from the operational source systems and dumped into the kitchen where it is transformed into meaningful information for the business. Skilled ETL architects and developers wield the tools of their trade in the DW/BI kitchen; once the data is verified and ready for business consumption, it is appropriately arranged “on the plate” and brought through the door into the DW/BI front room.

In the past, we’ve referred to the ETL system as *data staging*, but we’ve moved away from this terminology. We used data staging to refer to all the cleansing and data preparation that occurred between the source extraction and loading into target databases, however others used the term to merely mean the initial dumping of raw source data into a work zone.

Data Mart → Business Process Dimensional Model

The DW/BI system’s front room must be designed and managed with the business users’ needs front and center. Dimensional models are a fundamental front room deliverable; fact tables contain the metrics resulting from a business process or measurement event, while dimension tables contain the descriptive attributes and characteristics associated with measurement events. *Conformed dimensions* are the master data of the DW/BI environment, managed once in the kitchen and then shared by multiple dimensional models for enterprise integration and consistency.

Historically, we’ve heavily used the term *data mart* to refer to these architected *business process dimensional models*. While data mart is short-and-sweet, the term has been marginalized by others to mean summarized departmental, independent non-architected dataset; unfortunately, this hijacking has rendered the data mart terminology virtually meaningless.

End User Applications → Business Intelligence Applications

It’s not enough to just deliver dimensional data to the DW/BI system’s front room. Some business users are interested in and capable of formulating ad hoc queries, but most will be more satisfied with the ability to execute predefined applications that query, analyze, and present information from the dimensional model. There is a broad spectrum of *BI application* capabilities, from a set of canned static reports to analytic applications that directly interact with the operational transaction systems. In all cases, the goal is to deliver capabilities that are accepted by the business to support and enhance their decision making. Previously, we referred to these templates and applications as *end user applications*, but have since adopted the more current BI application terminology.

While our vocabulary has evolved slightly over the last 13 years, the underlying concepts have held steady. This is a testament to the permanency of our mission: bringing data effectively to business users to help them make decisions. Considering all the other changes in your world during this same timeframe, I think you’ll agree that the evolution of our vocabulary has been very slowly changing in comparison.

Design Tip #100 Keep Your Keys Simple

By Ralph Kimball

Recently a student emailed me with a design dilemma I have seen many times over my career. He said: *Last Friday I was told by my client that their method to assign surrogate keys to type 2 SCDs was the "industry standard." They are maintaining a 1 to 1 relationship between the natural keys and the surrogate keys in the dimension table. Their type 2 SCD in their EDW looks as follows:*

dim_key	*nat_key*	*attribs*	*start_dt*	*end_dt*
1	*ABC*	*Blue*	*1/1/2008*	*3/31/2008*
1	*ABC*	*Red*	*4/1/2008*	*12/31/9999*
2	*DEF*	*Green*	*1/1/2008*	*2/29/2008*
2	*DEF*	*Yellow*	*3/1/2008*	*12/31/9999*

I expected to see a unique surrogate key for each record, not re-using the same surrogate key for each natural key.

Here's my response:

I've seen this story a few times before! It is definitely wrong to make the SCD 2 key be a combination of a fixed key (1-1 with the natural key) plus a date range. All of your keys should be simple integers assigned by the data warehouse at ETL time, after extraction from the original source.

Big problems:
1) performance will definitely be compromised with a multi-field join from the dimension table to the fact table, especially if the fields involved are complex alphanumeric fields or date-time stamps. The best join performance is achieved by using single field integer keys between dimension tables and fact tables.

2) user/application developer understandability will definitely be compromised with the need to constrain the dates correctly. This point is expanded below.

Technical problems (show stoppers but harder to explain):
3) the date ranges in the dimension for each natural key presumably implement an unbroken overall span with no gaps. That is hard and sometimes nearly impossible to do correctly. If the end effective date of one record is exactly equal to the begin effective date of the next record (for that slowly changing member of the dimension) then you can't use BETWEEN in order to pick a specific dimension record because you could land on the exact dividing line and get two records. You must use greater-than-or-equal for the begin date and less-than for the end date which is likely to further compromise performance. You can't make the end-effective-date one "tick" less than the begin-date of the next record in order to use BETWEEN because the "tick" is machine/DBMS dependent and you can get situations in busy environments with data arriving from multiple sources where you lose transactions that fall into the gap. For example, one of my students gets 10,000 financial transactions in the last second of a fiscal period from 40 separate (and incompatible) source systems, with varying formats for the exact transaction times.

4) In situations where the begin and end effective stamps are actually date-TIMES then you have a

serious issue of how accurately does the user/application developer need to constrain the dimension. You would have to have a detailed understanding of intra-day business rules to do this correctly. Also, date-TIME stamps could be very unwieldy, depending on the DBMS.

5) Explicit date-TIME stamps are machine and DBMS dependent and therefore do not port cleanly from one environment to another.

6) In product movement situations as suggested by your sample data, the begin and end stamps MAY NOT STRADDLE the activity date of the fact record. For example, a specific product profile may be valid from Jan 1 to Feb 1, and hence those are the dates in the dimension, but the product may be sold Feb 15. Try explaining that to the users. Even worse, try building an application that requires two different date constraints. Using simple surrogate keys eliminates this objection, since the correct correspondence between dimension and fact records is resolved at ETL time.

7) The end user/developer MUST always constrain the dimension (actually EVERY dimension simultaneously) to an exact instant in time in every query for the rest of his/her life. Eight dimensions equals eight time constraints. And if one of the dimensions has DATE-TIME fields whereas the others have DATE-ONLY fields then you will get wrong answers if you constrain only to a date. Dates are dangerous because SQL is too "smart". A constraint on a DAY works on values within a day and the system doesn't warn you.

8) The style of embedding date ranges in every dimension record hints at a normalized style of modeling in which every many-to-1 relationship is encumbered with date ranges in order to handle time variance. Thus the objection raised in the previous paragraph is much worse if these normalized tables are actually exposed to the application developer or end user. Now, the data constraints would need to appear in every intermediate pair of tables, not just in the final dimension tables attached to fact tables.

9) Finally, any explicit dependence on the content of natural keys coming from a source system fails to anticipate the challenge of integrating data from multiple source systems, each with their own notion of natural keys. The separate natural keys could be of bizarrely different data types or could even overlap!

www.kimballgroup.com Number 99, March 4, 2008

Design Tip #99 Staging Areas and ETL Tools

By Joy Mundy

The majority of the data warehouse efforts we've recently worked with use a purchased tool to build the ETL system. A diminishing minority write custom ETL systems, which usually consist of SQL scripts, operating system scripts, and some compiled code. With the accelerated use of ETL tools, several of our clients have asked, does Kimball Group have any new recommendations with respect to data staging?

In the old days, we would extract data from the source systems to local files, transfer those files to the ETL server, and often load them into a staging relational database. That's three writes of untransformed data right there. We might stage the data several more times during the transformation process. Writes, especially logged writes into the RDBMS, are very expensive; it's a good design goal to minimize writes.

In the modern era, ETL tools enable direct connectivity from the tool to the source database. You can write a query to extract the data from the source system, operate on it in memory, and write it only once: when it's completely clean and ready to go into the target table. Although this is theoretically possible, it's not always a good idea.

- The connection between source and ETL can break mid-stream.
- The ETL process can take a long time. If we are processing in stream, we'll have a connection open to the source system. A long-running process can create problems with database locks and stress the transaction system.
- You should always make a copy of the extracted, untransformed data for auditing purposes.

How often should you stage your data between source and target? As with so many issues associated with designing a good ETL system, there is no single answer. At one end of the spectrum, imagine you are extracting directly from a transaction system during a period of activity – your business is processing transactions all the time. Perhaps you also have poor connectivity between your source system and your ETL server. In this scenario, the best approach is to push the data to a file on the source system, and then transfer that file. Any disruption in the connection is easily fixed by restarting the transfer.

At the other end of the spectrum, you may be pulling from a quiet source system or a static snapshot. You have a high bandwidth, reliable connection between snapshot and ETL system. In this case, even with large data volumes, it may be perfectly plausible to pull directly from the snapshot source and transform in stream.

Most environments are in the middle: most current ETL systems stage the data once or twice between the source and the data warehouse target. Often that staging area is in the file system, which is the most efficient place to write data. But don't discount the value of the relational engine in the ETL process, no matter which ETL tool you're using. Some problems are extremely well suited to relational logic. Although it's more expensive to write data into the RDBMS, think about (and test!) the end-to-end cost of staging data in a table and using the relational engine to solve a thorny transformation problem.

Design Tip #98 Focus on Data Stewardship

By Bob Becker

Delivering consistent data is like reaching the top of Mount Everest for most data warehouse initiatives, and data stewards are the climbers who fearlessly strive toward that goal. Achieving data consistency is a critical objective for most DW/BI programs. Establishing responsibility for data quality and integrity can be extremely difficult in many organizations. Most operational systems effectively capture key operational data. A line in the order entry system will typically identify a valid customer, product, and quantity. Optional fields that may be captured at that point, such as user name or customer SIC code, are sometimes not validated, if they get filled in at all. Operational system owners are not measured on the accuracy or completeness of these fields; they are measured on whether or not the orders get taken, filled, and billed. Unfortunately, many of these operationally optional fields are important analytic attributes. Quality issues or missing values become significantly magnified under the scrutiny of hundreds of analytic business users with high powered query tools.

Identifying and dealing with these issues requires an organizational commitment to a continuous quality improvement process. Establishing an effective data stewardship program is critical to facilitating this effort. The primary goal of a data stewardship program is the creation of corporate knowledge about its data resources to provide legible, consistent, accurate, documented, and timely information to the enterprise. Stewardship is also tasked with ensuring that data is used correctly and to its fullest extent, but only by those individuals authorized to leverage the data. Lack of consistent data across the organization is the bane of many DW/BI system efforts. Data stewardship is a key element in overcoming consistency issues.

Unfortunately, you can't purchase a wonder product to create conformed dimensions and miraculously solve your organization's master data management issues. Defining master conformed dimensions to be used across the enterprise is a cultural and geopolitical challenge. Technology can facilitate and enable data integration, but it doesn't fix the problem. Data stewardship must be a key component of your solution.

In our experience, the most effective data stewards come from the business community. As with technology, the DW/BI team facilitates and enables stewardship by identifying problems and opportunities and then implementing the agreed upon decisions to create, maintain, and distribute consistent data. But the subject matter experts in the business are the ones rationalizing the diverse business perspectives and driving to common reference data. To reach a consensus, senior business and IT executives must openly promote and support the stewardship process and its outcomes, including the inevitable compromises. For more information on how to set up a data stewardship initiative, take a look at this column: Data Stewardship 101 (*Intelligent Enterprise*, June 1, 2006).

Design Tip #97 Modeling Data as Both a Fact and Dimension Attribute

By Ralph Kimball

In the dimensional modeling world, we try very hard to separate data into two contrasting camps: numerical measurements that we put into fact tables, and textual descriptors that we put into dimension tables as "attributes". If only life were that easy…

Remember that numerical facts usually have an implicit time series of observations, and usually participate in numerical computations such as sums and averages, or more complex functional expressions. Dimension attributes, on the other hand, are the targets of constraints, and provide the content of "row headers" (grouping columns) in a query.

While probably 98% of all data items are neatly separated into either facts or dimension attributes, there is a solid 2% that don't fit so neatly into these two categories. A classic example that we have taught for years is the price of a product. Is this an attribute of the product dimension or is this an observed fact? In our opinion, this one is an easy choice. Since the price of a product often varies over time and over location, it becomes very cumbersome to model the price as a dimension attribute. It should be a fact. But it is normal to decide this rather late in the design process.

A more ambiguous example is the limit on a coverage within an automobile insurance policy. The limit is a numerical data item, say $300,000 for collision liability. The limit may not change over the life of the policy, or it changes very infrequently. Furthermore, many queries would group or constrain on this limit data item. This sounds like a slam dunk for the limit being an attribute of the coverage dimension.

But, the limit is a numeric observation, and it can change over time, albeit slowly. One could pose some important queries summing or averaging all the limits on many policies and coverages. This sounds like a slam dunk for the limit being a numeric fact in a fact table.

Rather than agonizing over the dimension versus fact choice, simply model it BOTH ways! Include the limit in the coverage dimension so that it participates in the usual way as a target for constraints and the content for row headers, but also put the limit in the fact table so it can participate in the usual way within complex computations.

This example illustrates some important dimensional modeling themes:

- Your design goal is ease-of-use, not elegance. In the final step of preparing data for consumption by end users, we should be willing to stand on our heads to make our BI systems understandable and fast. That means 1) transferring work into the ETL back room, and 2) tolerating more storage overhead in order to simplify the final data presentation.

- In correctly designed models, there is never a meaningful difference in data content between two opposing approaches. Stop arguing that "you CAN'T do the query if you model it that way". That is almost never true. The issues that you should focus on are ease of application development, and understandability when presented through an end user interface.

www.kimballgroup.com Number 96, October 31, 2007

Design Tip #96 Think Like A Software Development Manager

By Warren Thornthwaite

For most organizations, the vast majority of users get to the DW/BI system through the BI applications, including standard reports, analytic applications, dashboards, and operational BI; all these applications provide a more structured, parameter driven, relatively simple means for people to find the information they need. In Design Tip #91 on marketing the DW/BI system, we described the BI applications as the product of the DW/BI system and emphasized that these products must be valuable, usable, functional, high quality, and they must perform well.

Most of these characteristics are formed in the BI application design and development process. Throughout application development, testing, documenting, and rollout, it is very helpful to pretend to be a professional development manager from a consumer software product company. Actually, it's not pretending. Real software development managers go through the same steps as the folks responsible for delivering the BI applications. The best software development managers have learned the same lessons:

- The project is 25 percent done when your developer gives you the first demo of the working application. The first demo from a proud developer is an important milestone that you should look forward to, but seasoned software development managers know that the developer has only passed the first unit test. The second 25 percent is making the application pass the complete system test, where all the units are working. The third 25 percent is validating and debugging the completed system in a simulated production environment. The final 25 percent is documenting and delivering the system into production.

- Don't believe developers who say their code is so beautiful that it is self-documenting. Every developer must stay on the project long enough to deliver complete, readable, high-quality documentation. This is especially true for any interactions or algorithms that are directly exposed to the users.

- Use a bug tracking system. Set up a branch in your bug tracking or problem reporting system to capture every system crash, every incorrect result, and every suggestion. A manager should scan this system every day, assigning priorities to the various problems. An application cannot be released to the user community if there are any open priority 1 bug reports.

- Place a very high professional premium on testing and bug reporting. Establish bug-finding awards. Have senior management praise these efforts. Make sure that the application developers are patient with business users and testers.

- Be proactive with the bug reports you collect from users and testers. Acknowledge receipt of every reported bug, allow the users and testers to see what priority you have assigned to their reports and what the resolution status of their reports is, and then fix all the bugs.

These lessons are especially important in organizations with large user communities. The same is true when creating operational BI applications that will be used by a large number of operational users. You will probably not meet everyone individually in these user communities, so your product had better be great.

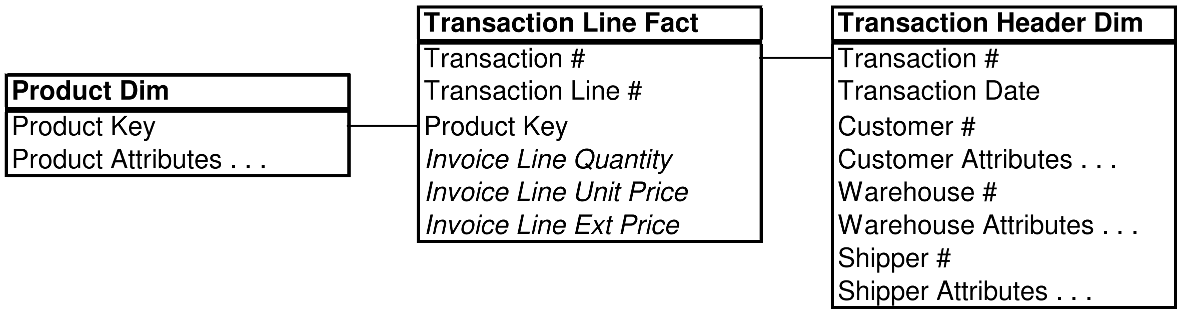
www.kimballgroup.com Number 95, October 2, 2007

Design Tip #95 Patterns to Avoid when Modeling Header/Line Item Transactions

By Margy Ross

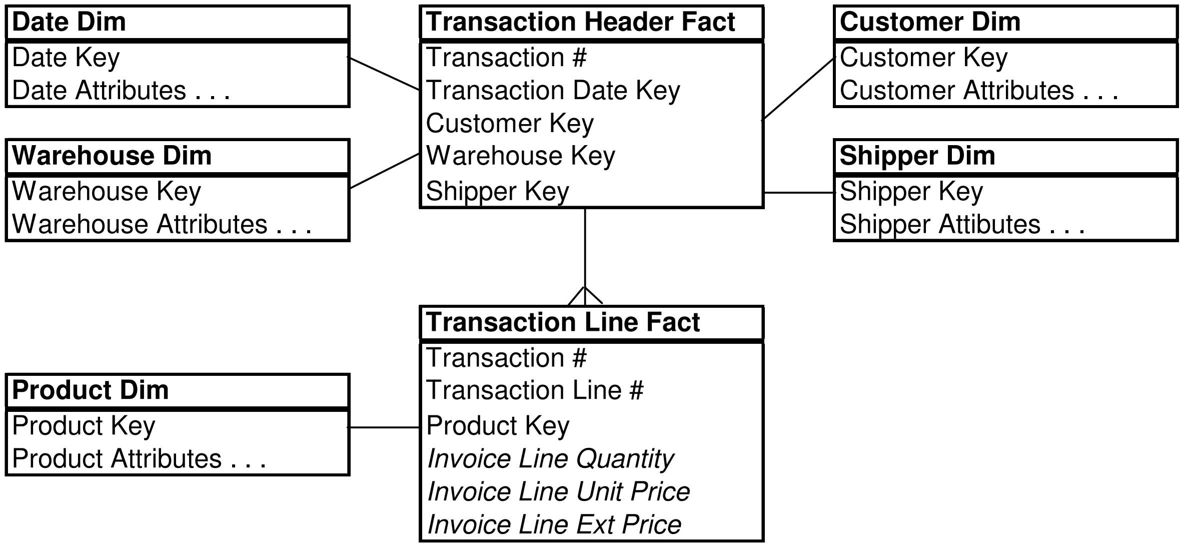
Many transaction processing systems consist of a transaction header “parent” with multiple line item “children.” Regardless of your industry, you can probably identify source systems in your organization with this basic structure. When it’s time to model this data for DW/BI, many designers merely reproduce these familiar operational header and line constructs in the dimensional world. In this Design Tip, we describe two common, albeit flawed, approaches for modeling header/line item information using invoicing data as a case study. Sometimes visualizing flawed designs can help you more readily identify similar problems with your own schemas.

Bad Idea #1
In this scenario, the transaction header file is virtually replicated in the DW/BI environment as a dimension. The transaction header dimension contains all the data from its operational equivalent. The natural key for this dimension is the transaction number itself. The grain of the fact table is one row per transaction line item, but there’s not much dimensionality associated with it since most descriptive context is embedded in the transaction header dimension.



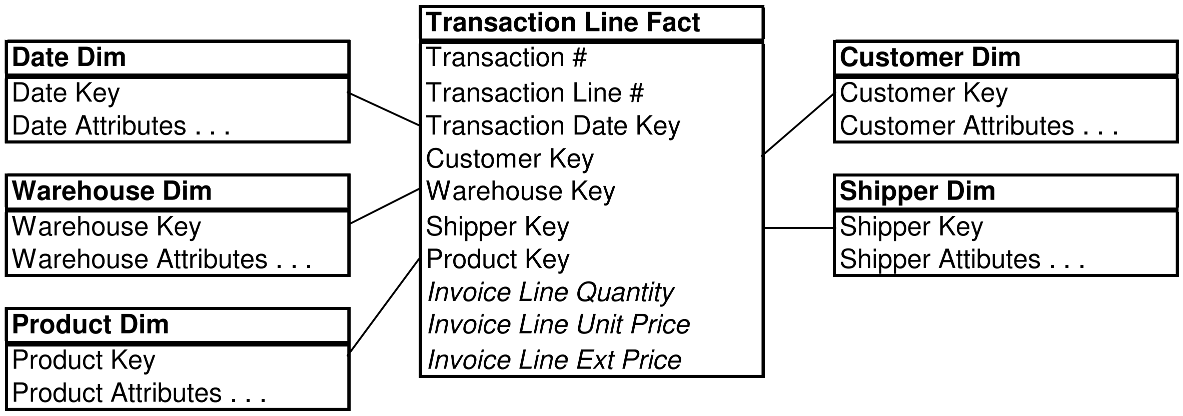
While this design accurately represents the parent/child relationship, there are obvious flaws. The transaction header dimension is likely very large, especially relative to the fact table itself. If there are typically five line items per transaction, then the dimension is 20% as large as the fact table. Usually there are orders of magnitude differences between the size of a fact table and its associated dimensions. Also, dimensions don’t normally grow at nearly the same rate as the fact table. With this design, you’d add one row to the dimension table and an average of five rows to the fact table for every new transaction. Any analysis of the transaction’s interesting characteristics, such as the customer, warehouse, or shipper involved, would need to traverse this large dimension table.

Bad Idea #2
In this example, the transaction header is no longer treated as a monolithic dimension but as a fact table instead. The header’s descriptive information associated is grouped into dimensions surrounding the header fact. The line item fact table (identical in structure and granularity as the first diagram) joins to the header fact based on the transaction number.



Once again, this design accurately represents the parent/child relationship of the transaction header and line items, but there are still flaws. Every time the user wants to slice-and-dice the line facts by any of the header attributes, they'll need to join a large header fact table to an even larger line fact table.

Recommended Structure for Header/Line Item Transactions
The second scenario more closely resembles a proper dimensional model with separate dimensions uniquely describing the core descriptive elements of the business, but it's not quite there yet. Rather than holding onto the operational notion of a transaction header "object," we recommend that you bring all the dimensionality of the header down to the line items.



Once again, this model represents the data relationships from the transaction header/line source system. But we've abandoned the operational mentality surrounding a header file. The header's natural key, the transaction number, is still present in our design, but it's treated as a degenerate dimension.

For more information about transaction header/line schemas, you can refer back to Design Tip #25 where Ralph describes the allocation of header-level facts down to the line level granularity.

www.kimballgroup.com Number 94, September 4, 2007

Design Tip #94 Building Custom Tools for the DW/BI System

By Joy Mundy

There is a large and diverse market for products to help build your DW/BI system and deliver information to your business users. These range from DBMSs – relational and OLAP – to ETL tools, data mining, query, reporting, and BI portal technologies. What role could there possibly be for custom tools in such a rich environment?

Most of the custom tools we've seen have supported back room operations, such as metadata management, security management, and monitoring. For example, you could be capturing information about who's logging into the system, and how long queries are taking. The simplest custom monitoring tool would be a set of predefined reports to display historic trends and real time activity.

But the best tools will let the users initiate an action. One recent client had an unusual requirement where business users performed complex analysis and then submitted jobs to the ETL system. Each job could take anywhere from a few minutes to a quarter hour to run, depending on how busy the system was. The business users submitted their jobs at the end of the day, and then hung around the office until they were certain their data processed correctly and was ready for the next morning. The DW/BI team developed a straightforward tool that monitored the jobs that were submitted to the system. Users could see where their jobs were in the processing queue, get a good idea of how long they might take, and – best of all – users could cancel their own jobs if they realized they made a mistake in the data preparation step. This tool was particularly nice and developed by a skilled programmer, but a less fancy tool could be pulled together in a matter of a few weeks.

Most DW/BI teams use a variety of products from multiple vendors. Custom tools will be most useful at the transition points between different technologies. This holds true even if your DW/BI system is built largely on a single platform; there are always gaps between the components. Metadata management is one place where we might need to write a bit of custom glue. In the absence of an integrated platform with complete and synchronized metadata between design, relational and OLAP databases, business intelligence layer, and standard reports—a platform we still hope to see some day—there will always be a place for a custom tool to bridge those metadata ponds. A very simple tool might consist of a few scripts to synchronize metadata stores. But we have seen customers with web-based applications that let a business analyst update and synchronize metadata, such as business descriptions.

Other examples of custom tools that we've seen at our clients include:

- Report publishing workflow – manage the process for creating a new standard report, including ensuring the report's definition is approved by appropriate representatives from both business and IT.

- Security management – programmatically issue the commands to add users to the system with a user interface for assigning them to specific roles. This is particularly valuable for data-driven security systems such as those driven from an organizational structure and for security systems that span multiple databases.

- Dimension hierarchy management – enable business users to remap dimension hierarchies, such as which products roll up to a product subcategory and category, or which general ledger accounts are aggregated together.

Don't be overwhelmed! Many DW/BI teams build no custom tools, or only a few very primitive tools. But there are some very effective programming environments on the market, not to mention inexpensive software development houses that you can hire. Don't be afraid to be creative. It's often the case that a very modest investment in some custom tools can greatly improve the manageability of your DW/BI system. The best tools are the ones that make the business users happier, by giving them more control over the DW/BI system that is, by all rights, theirs rather than yours.

www.kimballgroup.com Number 93, July 30, 2007

Design Tip #93 Transactions Create Time Spans

By Ralph Kimball

In the dimensional modeling world we talk about three kinds of fact table grains: transaction, periodic snapshot, and accumulating snapshot. If you are careful to make sure every fact table you design is based on one grain, and doesn't mix different grains, amazingly these three choices are enough to design every fact table in your data warehouse.

The transaction grain is used for measurements at a specific instant. When you make a deposit into your bank account, a transaction record is created. The record exists only if the event actually happens. We don't know when or if any more transaction events will occur. The next one could be one second later or next year.

In spite of the seemingly ephemeral nature of transactions, each transaction leaves behind a powerful legacy. At the moment of the transaction, the state of the business process is frozen and remains frozen until the next transaction occurs. As a result of your deposit, your bank account has a new balance, which remains at exactly the same value until the next transaction occurs. Thus each transaction against your bank account implicitly defines a time span until the next transaction occurs. During this time span, all aspects of your account are frozen. Wouldn't it be interesting to be able to quickly query history for bank accounts at a particular random point in time and get all the balances at that moment?

Time spans can be represented in a transaction grain fact table by adding begin and end datetime stamps to each record, and in the case of the bank account, adding a field representing the account balance that results from the transaction. These are not foreign keys to any kind of time dimension. Rather, these are full datetime fields. The begin datetime, of course, is the exact moment when the transaction takes effect. The end datetime is initially set to a maximum datetime far in the future, such as Dec 31, 9999, 11:59:59 pm. It is important to set the end datetime to a real value, not NULL, so that BETWEEN comparisons always return a value.

The end datetime remains at the artificial maximum value until the next transaction occurs. Then you must set this field to the next transaction datetime minus the smallest increment of time supported by your datetime stamp. Then the next transaction record is loaded into the fact table as described in the previous paragraph. You now have an unbroken string of time spans which can be queried using any date and time you desire.

Keep in mind the following caveats. First, your transactions must be complete. It won't work to include just customer deposits and withdrawals, while omitting other bank initiated transactions. Second, not every transaction record creates a useful unique time span. Bank account transactions fit the time span approach beautifully, but grocery store cash register transactions probably don't.
So the next time you define a transaction grained fact table, give some thought to the extra flexibility that these implicit time spans can offer.

www.kimballgroup.com Number 92, June 26, 2007

Design Tip #92 Dimension Manager and Fact Provider

By Bob Becker

Conformed dimensions are the glue that ties together your enterprise data warehouse. To facilitate and manage the conforming process, we have identified two additional fundamental responsibilities for the DW/BI team: the dimension manager and fact provider. Typically these functions are performed by the ETL team working closely with the data stewardship organization.

The dimension manager is a centralized authority who prepares and publishes conformed dimensions to the data warehouse community. We first introduced this new data warehouse role and its partner role, the fact provider, in *The Data Warehouse ETL Toolkit*. A conformed dimension is by necessity a centrally managed resource: each conformed dimension must have a single, consistent source. It is the dimension manager's responsibility to administer and publish the conformed dimension(s) for which he has responsibility. There may be multiple dimension managers in an organization. The dimension manager's responsibilities include the following ETL processing:

- Implement the common descriptive labels agreed to by the data stewards and stakeholders during the dimension design.
- Add new rows to the conformed dimension for new source data, generating new surrogate keys.
- Add new rows for Type 2 changes to existing dimension entries (true physical changes at a point in time), generating new surrogate keys.
- Modify rows in place for Type 1 changes (overwrites) and Type 3 changes (alternate realities), without changing the surrogate keys.
- Update the version number of the dimension if any Type 1 or Type 3 changes are made.
- Replicate the revised dimension simultaneously to all fact table providers.

It is easier to manage conformed dimensions in a single tablespace in a single DBMS instance on a single machine because there is only one copy of the dimension table. However, managing conformed dimensions becomes more difficult in multiple tablespace, multiple DMBS, or multi-machine distributed environments. In these situations, the dimension manager must carefully manage the simultaneous release of new versions of the dimension to every fact provider. Each conformed dimension should have a version number column in each row that is overwritten in every row whenever the dimension manger releases the dimension. This version number should be utilized to support any drill across queries to assure that the same release of the dimension is being utilized. *The Data Warehouse ETL Toolkit* has more details on how to instrument this drill-across capability incorporating the dimension release version number.

The fact table provider is responsible for receiving conformed dimensions from the dimension managers. The fact provider owns the administration of one or more fact tables and is responsible for their creation, maintenance and use. If fact table are used in any drill across applications then by definition the fact provider must be using conformed dimensions provided by the dimension manager. The fact provider's responsibilities are more complex. They include:

- Receive or download replicated dimension from the dimension manager. In an environment where the dimension can not simply be replicated but must be updated locally, the fact provider must process dimension records marked as new and current to update current key maps in the surrogate key pipeline and also process any dimension records marked as new but postdated.
- Add all new records to fact tables after replacing their natural keys with correct surrogate

keys.

- Modify records in all fact tables for error correction, accumulating snapshots, and late arriving dimension changes.
- Remove aggregates that have become invalidated and/or recalculate affected aggregates. If the new release of a dimension does not change the version number, aggregates have to be extended to handle only newly loaded fact data. If the version number of the dimension has changed, the entire historical aggregate may have to be recalculated.
- Quality assure all base and aggregate fact tables. Be satisfied that the aggregate tables are correctly calculated.
- Bring updated fact and dimension tables on line.
- Inform end users that the database has been updated. Tell users if major changes have been made, including dimension version changes, postdated records being added, and changes to historical aggregates.

In order for your conformed dimension strategy to be effective over the long haul, your conformed dimensions must remain in sync across all fact tables. Formally establishing dimension managers and fact providers proactively assigns responsibilities to specific individuals to ensure a sustainable conformed environment over time. We recommend that you assign a dimension manager to each major conformed dimensions and a fact provider for each fact table. Certainly, in many organizations, an individual may be responsible for multiple dimension or fact tables.

KIMBALL GROUP KIMBALL UNIVERSITY

www.kimballgroup.com Number 91, May 30, 2007

What's New

In this Design Tip, Warren describes classic marketing 101 concepts and applies them to the ongoing marketing of your DW/BI system.

We have a full line-up of Kimball University classes scheduled for June. Now's your opportunity to get practical, in-depth education from the Kimball Group!

We're bringing both Dimensional Modeling and ETL Architecture in Depth to Minneapolis during mid June. We haven't taught classes in Minnesota since 2003; Margy and Bob are both excited about teaching in their hometown with me.

Joy and Warren are returning to Seattle with their Microsoft Data Warehouse class on June 12th. This is a very popular venue for this course, so reserve your seat early.

Finally, in late June, we're teaching a Data Warehouse Lifecycle class in Anaheim, just in time to coincide with summer vacations.

We have a limited block of rooms reserved at every hotel hosting our classes, but you need to make your reservation early if you want to obtain a room at the discounted rate.

As always, thanks for your ongoing support and interest in the Kimball Method!

Regards,

Ralph

Learn from the *Toolkit* Experts in upcoming Kimball University classes!

Dimensional Modeling in Depth

Jun 12-15	Minneapolis, MN
Aug 14-17	Seattle, CA (early discount ends 6/29)

Microsoft Data Warehouse in Depth

Jun 12-15	Seattle, WA
Jul 31-Aug 3	San Diego, CA (early discount ends 6/15)

ETL Architecture in Depth

Jun 18-21	Minneapolis, MN

Data Warehouse Lifecycle in Depth

Jun 26-29	Anaheim, CA
Aug 7-10	New York City, NY (early discount ends 6/22)

Upcoming Events

Architecture of Data Quality
web seminar featuring Ralph
Jun 5

Sponsored by: INFORMATICA

Implementing a Scalable Enterprise DW/BI Solution seminar featuring Warren

Jun 5 Seattle

Sponsored by: hp invent Microsoft intel

Design Tip #91 Marketing the DW/BI System

Marketing is often dismissed by technical folks. When someone says "oh, you must be from marketing," it's rarely meant as a complement. This is because we don't really understand what marketing is and why it's important. In this design tip, we'll review classic marketing concepts and explore how we can apply them to the DW/BI system.

It might be more palatable to think of marketing as education. Marketers educate consumers about product features and benefits, while generating awareness of a need for those features and benefits. Marketing gets a bad name when it's used to convince consumers of a need that isn't real, or sell a product that doesn't deliver its claimed features and benefits. But that is a different article. Really great marketing, when effectively focused on the value delivered, is hugely important.

Before you start creating your marketing program, you should have a clear understanding of your key messages: what are the mission, vision, and value of your DW/BI system? Marketing 101 has focused on the four Ps, Product, Price, Placement, and Promotion, for at least the last 30 years or so.* We'll look at each of these factors in the context of the DW/BI system and direct you to additional information where it is available.

Product
As far as the business community is concerned, the DW/BI products are the information needed for decision making and the BI applications and portal through which the information is delivered. Our products must excel in the following five areas:

- Value – meet the business needs identified in the business requirements process
- Functionality – product must work well
- Quality – data and calculations have to be right
- Interface – be as easy as possible to use and look good
- Performance – work in a reasonable timeframe as defined by the users

Price
Most users don't pay for the DW/BI system directly. The price they pay is the effort it takes to get the information from the DW/BI system compared to other alternatives. There is an upfront cost of learning how to use the BI tool or the BI applications, and an ongoing cost of finding the right report or building the right query for a particular information need. You must lower the price as much as possible by first creating excellent products that are as easy to use as possible. Then offer a full set of training, support and documentation, including directly accessible business metadata, on an ongoing basis.

Placement
In consumer goods, placement is obvious: the product has to be on the store shelf or the customer can't buy it. For us, placement means our customers are able to find the information they need when they need it. In other words, you must build a navigation structure for the BI applications that makes sense to the business folks. Additionally, tools like search, report metadata descriptions and categories, and personalization capabilities can be extremely helpful. For additional information, see Design Tip #58: BI Portal and my February 2006 Intelligent Enterprise article, "Standard Reports: Basics for Business Users" at www.kimballgroup.com.

Promotion
Every customer contact you have is a marketing opportunity. TV ads are not an option, not counting YouTube, but you do have several promotion channels:

- BI applications – These are what people use the most. Names are important: having a good acronym for the DW/BI system can leave a good impression. Every report and application should have a footer indicating that it came from the DW/BI system and a logo in one of the upper corners. Ultimately, if you create a good product, the name and logo will become marks of quality – your brand.
- BI portal – This is the main entry point for the DW/BI system. It has to meet the same requirements as the BI applications.
- Regular communications – Know who your stakeholders are and which communications vehicle works best for each. Your ongoing communications plan will include status reports, executive briefings, and user newsletters. Consider webcasts on specific topics if that's an option in your organization.

- Meetings, events, and training – Any public meeting where you can get a few minutes on the agenda is a good thing. Briefly mention a recent successful business use, remind people of the nature and purpose of the DW/BI system, and tell them about any upcoming plans or events. Host your own events, like User Forum meetings, every six to nine months or so.

Ongoing marketing is a key element of every successful DW/BI system. The more you keep people informed about the value you provide them, they more they will support your efforts.

* A fifth P, People, is sometimes included. In recent years, there's been a push to replace the Marketing 4 Ps with 4 Cs: Customer solution, Customer cost, Convenience and Communication.

Warren Thornthwaite
warren@kimballgroup.com

www.kimballgroup.com Number 90, April 30, 2007

Design Tip #90 Slowly Changing Entities

By Ralph Kimball

From time to time we get asked whether the techniques for handling time variance in dimensions can be adapted from the dimensional world to the normalized world. In the dimensional world we call these techniques “slowly changing dimensions” or SCDs. In the normalized world we might call these “slowly changing entities” or SCEs.

We’ll show in this design tip that while it is possible to implement SCEs, in our opinion it is awkward and impractical to do so.

Consider, for example, an employee dimension containing 50 attributes. In a dimensional data warehouse, this employee dimension is a single flat table with 56 columns. Fifty of the columns are copied from the original source. We assume that these source columns include a natural key field, perhaps called Employee ID, which serves to reliably distinguish actual employees. The six additional columns are added to support SCDs in the dimensional world, and these columns include a surrogate primary key (used to join to fact tables), change date, begin-effective date-time, end-effective date-time, change reason, and most-recent-flag. The detailed use of these fields has been described many times in our design tips and Toolkit books.

Now let’s review what we do when we are handed a changed record from the source system. Suppose that an individual employee changes office location, affecting the values of five attributes in the employee record. Here are the steps for what we call Type 2 SCD processing:

1. Create a new employee dimension record with the next surrogate key (adding 1 to the highest key value previously used). Copy all the fields from the previous most-current record, and change the five office location fields to the new values.
2. Set the change date to today, the begin-effective date-time to now, the end-effective date-time to December 31, 9999 11:59 pm, the change reason to “Relocation”, and the most recent flag to True.
3. Update the end-effective date-time of the previously most-current record for that employee to now minus 1 second, and change the most recent flag to False.
4. Begin using the new surrogate key for that employee in all subsequent fact table record entries.

A slightly more complicated case arises if a hierarchical attribute affecting many employees is changed. For example if, if an entire sales office is deemed to be assigned to a new division, then every employee in that sales office needs to undergo the Type 2 steps described above, where the sales division fields are the target of the change.

Now what if we have a fully normalized employee database? Remember that normalization requires that every field in the employee record that does not depend uniquely on the employee natural key must be removed from the base employee record and placed in its own table. In a typical employee record with 50 fields, only a small number of high cardinality fields will depend uniquely on the natural key. Perhaps 40 of the fields will be low cardinality fields “normalized out” of the base employee record into their own tables. Perhaps 30 of these fields will be independent of each other and cannot be stored together in the same entity. That means the base employee record must have 30 foreign keys to these entities! There will be additional physical tables containing lower cardinality fields two or more levels away from the base employee table. Instead of maintaining one natural key and one surrogate primary key as in the dimensional world, the DBA and application designer must maintain and be aware of more than 30 pairs of natural and surrogate keys in order to track time variance in the normalized world.

We'll assume that every entity in the normalized database contains all six SCE navigation fields, analogous to the SCD design.

To make the office location change described above to an individual employee profile, we must perform all the administrative steps listed above, for each affected entity. But even worse, if there is a change to any field removed more than one level from the base employee record, the DBA must make sure that the SCE processing steps are propagated downward from the remote entity through each intermediate entity all the way to the base employee table. The base employee table must also support the primary surrogate key for joining to the fact table, unless the normalized designer opts to omit all surrogate keys in favor of only constraining on the begin- and end-effective date-time stamps. (We think that eliminating all surrogate keys is a performance and applications mistake of the first magnitude). All of these comments are also true for the second change we described above, for the division reassignment of the sales office.

Processing time variance in a fully normalized database involves other nightmares too complicated to describe in detail in this design tip. For example, if you are committed to a correctly normalized physical representation of your data, then if you discover a business rule that changes a presumed many-to-1 relationship in your data to many-to-many, then you must undo the key administration and physical table design of the affected entities. These changes also require user queries to be reprogrammed! These steps are not needed in the dimensional world. As we have said many times, all of the actual processing to validate and administer the original input data must be performed on pre-normalized (i.e., flat) data anyway. A second serious problem for handling time variance in the normalized world is how to administer late arriving dimensional data. Instead of creating and inserting one new dimension record as in the SCD case, we must create and insert new records in every affected entity and all of the parents of these entities back to the base employee record.

As we often remark, a clever person who is a good programmer can do anything. You can make SCEs work in both the ETL back room and the BI front room if you are determined, but since the final data payload is identical to the simpler dimensional SCDs, we respectfully suggest that you win the Nobel prize on a different topic.

www.kimballgroup.com Number 89, March 20, 2007

Design Tip #89 The Real Time Triage

By Ralph Kimball

Asking business users if they want "real time" delivery of data is a frustrating exercise for the BI system designer. Faced with no constraints, most users will say "that sounds good, go for it!" This kind of response is almost worthless. One is left wondering if the user is responding to a fad.

To avoid this situation we recommend dividing the real time design challenge into three categories, which we will call Daily, Frequently, and Instantaneous. We will use these terms when we talk to end users about their needs, and we will design our data delivery pipelines differently for each of these choices.

Instantaneous means that the data visible on the screen represents the true state of the source transaction system at every instant. When the source system status changes, the screen responds instantly and synchronously. An instantaneous real time system is usually implemented as an EII (Enterprise Information Integration) solution, where the source system itself is responsible for supporting the update of remote user's screens, and servicing query requests. Obviously such a system must limit the complexity of the query requests because all the processing is done on the legacy application system. EII solutions typically involve no caching of data in the ETL pipeline, since EII solutions by definition have no delays between the source systems and the users' screens. EII technologies offer reasonable light weight data cleaning and transformation services, but all these capabilities must be executed in software since the data is being continuously piped to the users' screens. Most EII solutions also allow for a transaction protected write back capability from the users' screens to the transactional data. In the business requirements interviews with end users, you should carefully assess the need for an instantaneous real time solution, keeping in mind the significant load that such a solution places on the source application, and the inherent volatility of instantaneously updated data. Some situations are ideal candidates for an instantaneous real time solution. Inventory status tracking may be a good example, where the decision maker has the right to commit inventory to a customer that is available in real time.

Frequently means that the data visible on the screen is updated many times per day but is not guaranteed to be the absolute current truth. Most of us are familiar with stock market quote data that is current to within 15 minutes but is not instantaneous. The technology for delivering frequent real time data (as well as the slower daily real time data) is distinctly different from instantaneous real time delivery. Frequently delivered data is usually processed as micro-batches in a conventional ETL architecture. This means that the data undergoes the full gamut of change data capture, extract, staging to file storage in the ETL back room of the data warehouse, cleaning and error checking, conforming to enterprise data standards, assigning of surrogate keys, and possibly a host of other transformations to make the data ready to load into a ROLAP (dimensional) star schema, or an OLAP cube. Almost all of these steps must be omitted or drastically reduced in an EII solution. The big difference between frequently and daily delivered real time data is in the first two steps: change data capture and extract. In order to capture data many times per day from the source system, the data warehouse usually must tap into a high bandwidth communications channel such as message gram traffic between legacy applications, or an accumulating transaction log file, or low level database triggers coming from the transaction system every time something happens. As a designer, the principal challenge of frequently updated real time systems is designing the change data capture and extract parts of the ETL pipeline. If the rest of the ETL system can be run many times per day, then perhaps the design of these following stages can remain batch oriented.

Daily means that the data visible on the screen is valid as of a batch file download or reconciliation from the source system at the end of the previous working day. A few years ago a daily update of the data warehouse was considered aggressive, but in 2007 daily data would be the most conservative choice. There is a lot to recommend daily data! Quite often processes run on the source system at the end of the working day that correct the raw data. When this reconciliation becomes available, that is the signal that the data warehouse can perform a reliable and stable download of the data. If you have this situation, you should explain to the end users what compromises they will experience if they demand instantaneous or frequently updated data. Daily updated data usually involves reading a batch file prepared by the source system, or performing an extract query when a source system readiness flag is set. This, of course, is the simplest extract scenario, because you take your time waiting for the source system to be ready and available. Once you have the data, then the downstream ETL batch processing is similar to that of the frequently updated real time systems, but it only needs to run once per day.

The business requirements gathering step is crucial to the design process. The big decision is whether to go instantaneous, or can you live with frequently or daily. The instantaneous solutions are quite separate from the other two, and you would not like to be told to change horses in midstream. On the other hand, you may be able to gracefully convert a daily real time ETL pipeline to frequently, mostly by altering the first two steps of change data capture and extract. If you would like to study all 34 steps of the ETL pipeline including the ones mentioned in this design tip, please check out the Data Warehouse ETL Toolkit book and our ETL in Depth class, both of which are described on our website www.kimballgroup.com.

www.kimballgroup.com Number 88, February 22, 2007

Design Tip #88 Dashboards Done Right

By Margy Ross

With their graphically appealing user interfaces, dashboards and their scorecard cousins are demo superstars. Dashboards have really grabbed the attention of senior management since they closely align with the way these people operate. What's not to like about the promise of performance feedback on every customer or supplier facing process in the organization at a glance. It's no wonder execs are enthused.

But there's a dark side to dashboard projects. They're extremely vulnerable to runaway expectations. They're risky due to the cross organizational perspective inherent in most dashboard designs. And they can be a distraction to the DW/BI team; rather than focusing on the development of an extensible infrastructure, dashboard projects often encourage a data triage where key performance indicators from a multitude of processes are plucked and then pieced together using the systems equivalent of chewing gum and duct tape.

Dashboards and scorecards done right are layered on a solid foundation of detailed, integrated data. Anything less is ill-advised. Dashboards based on manually collected, pre-aggregated, standalone subsets of data are unsustainable in the long run.

If you have an existing data warehouse that's populated with the requisite detailed, integrated data, then you should tackle any proposed dashboard development project with gusto. Dashboards present a tangible opportunity to deliver on the promise of business value derived from your data warehouse. The dashboard interface appeals to a much broader set of users than traditional data access tools. In addition, dashboards provide a vehicle for going beyond rudimentary, static reporting to more sophisticated, guided analytics.

But what do you do when your executives are clamoring for a sexy dashboard, but there's no existing foundation that can be reasonably leveraged? Facing a similar predicament, some of you have bootstrapped the dashboard development effort. And it may have been initially perceived as a success. But then middle managers start calling because their bosses are monitoring performance via the dashboard, yet there's no ability for them to drill into the details where the true causal factors of a problem are lurking. Or management starts to question the validity of the dashboard data because it doesn't tie to other reports due to inconsistent transformation/business rules. Or the users determine they need the dashboard updated more frequently. Or your counterpart who supports another area of the business launches a separate, similar but different dashboard initiative. The quick bootstrapped dashboard will be seriously, potentially fatally, stressed from the consequences of bypassing the development of an appropriate infrastructure. Eventually you'll need to pay the price and rework the initiative.

While it's perhaps less politically attractive at first, a more sustainable approach would be to deliver the detailed data, one business process at a time, tied together with conformed master dimensions, of course. As the underlying details become available, the dashboard would be incrementally embellished to provide access to each deployment of additional information. We understand this approach doesn't deliver the immediate "wow" factor and requires executive patience. While executives may not naturally exhibit a high degree of patience, most are also reluctant to throw away money on inevitable rework caused by taking too many shortcuts. Having an honest conversation with business and/or IT management so they fully understand the limitations and pitfalls of the quick-and-dirty dashboard may result in staunch converts to the steadier, more sustainable approach.

Those of us longer in the tooth remember EIS, or Executive Information Systems, that blossomed briefly in the 1980s. EIS suffered from exactly the same problem we are discussing here. The carefully prepared executive KPIs were not supported by solid detailed data that could withstand drill down. Any good executive is going to ask “why?” And that’s when the data warehouse and its dashboards need to sit on a solid foundation.

www.kimballgroup.com Number 87, January 24, 2007

Design Tip #87 Combining SCD Techniques: Having It Both Ways

By Joy Mundy

I've spoken with many warehouse designers who are torn between conflicting business requirements when structuring their dimensions. Some business users have a compelling requirement for a dimension attribute to track history as a Type 2 attribute, while other users are equally clear they want that same attribute to restate history as Type 1. What to do?

One approach is to construct queries so that we join the dimension table to itself, returning the dimensional attributes for the current row together with the measured numeric facts for all history, such as:

```
SELECT CurrCust.CustomerName AS CurrentCustomerName,
CurrCust.City AS CurrentCustomerCity,
SUM(SalesFacts.SalesAmount) AS TotalSales
FROM Customer CurrCust
INNER JOIN Customer CustType2 ON
  (CurrCust.CustBusinessKey = CustType2.CustBusinessKey)
  AND CurrCust.IsCurrentInd = 'Y'
    INNER JOIN SalesFacts ON
      SalesFacts.CustomerKey = CustType2.CustomerKey
GROUP BY CurrCust.CustomerName, CurrCust.City
```

This query returns an answer set with three columns: current customer name, current customer city and sales for that customer across all time. This SQL has been tested and is correct! This is the easiest solution to implement on the design and ETL side because it requires no additional effort. But our design goal is seldom centered on what's easiest for the ETL system to implement, and this approach has two significant downsides. First and most important, it's not something we'd expect most business users to be able to do. Second, that self-join on the dimension table will slow down query performance.

What we're really doing in this self-join approach is adding the current version of each Type 2 attribute as an additional column in the dimension table. We're doing this virtually at query time, but a nicer approach is simply to do it once at ETL time. In other words, if you have a Type 2 attribute that some people want to see as Type 1, then put it in your dimension table twice. Although this may seem to violate the dictum that we have a single version of the truth, that problem is neatly solved by renaming the Type 1 attribute as *current*, such as Current Customer City. It's easy to educate the business users that the current attributes are the versions of the attribute as they are today; the other attribute with the similar name tracks history. This is a very simple solution.

Most warehouse designers are comfortable with this approach as long as there are only a handful of Type 2 attributes in the dimension. But what if your dimension table is very wide, for example a customer dimension with a hundred attributes, and many or most of those attributes are tracked as Type 2? Doubling the width of the dimension table with a hundred additional *current* versions of each attribute is unappealing. It's particularly awkward if your user access technology doesn't have a way to group attributes into display folders when the user is browsing the data structures.

In the extreme situation where your dimension table is long and wide – you have millions of customers and hundreds of attributes about those customers – you may choose to create two dimensions: Customer and Current Customer. Customer would be the normal dimension with many Type 2 attributes, and would contain multiple rows for each customer as their attributes change over time. Current Customer would contain only the current view of each customer's attributes, and would have one row per customer. Current Customer would be a pure Type 1 dimension. Fact tables would need to have a key for both Customer and Current Customer. This technique is further described in our *"Slowly Changing Dimensions are not always as Easy as 1, 2, 3"* article published in Intelligent Enterprise.

These two approaches roll neatly into an OLAP solution. The simple approach of adding the current attributes to the dimension table would translate to additional attributes of the Customer dimension in the OLAP database. If your only access into the data is through the OLAP layer, you can actually get away with the first technique we mentioned: use a query or define a view that performs the self-join on the dimension table. Most OLAP tools would only execute this query at the time the dimension is processed, and it would probably perform adequately. If instead you decide to create and manage a separate Current Customer dimension table, it would show up in your OLAP tool as a separate dimension. It should work as well in OLAP as it does in the relational database.

www.kimballgroup.com Number 86, December 15, 2006

Design Tip #86 Creating a Reference Dimension for Infrequently-Accessed Degenerates

By Bob Becker

In Design Tips #81 and #84, Ralph and I explored several circumstances when it may be advantageous to create a surrogate key for fact tables. This design tip leverages that earlier discussion and introduces a concept called the reference dimension where we stash rarely used fact table elements, such as degenerate dimension reference numbers, in a separate table that's linked to the fact table either through a regular dimension surrogate key or the fact table's surrogate key.

While degenerate dimensions (Design Tip #46) often play an important role in supporting reporting and analytic requirements, some degenerate dimensions are not analytically valuable; they are included in the schema for reference purposes only. They help provide occasional ties back to the operational source systems, support audit, compliance or legal requirements, or are included simply because "we might need it someday." The result can be a fact table that contains a large number of degenerate dimensions with perhaps only two or three that are truly important and interesting. In healthcare, for example, the degenerate dimensions might reference the provider network contract number, fee schedule number, and claim microfiche number. Since fact tables are the largest tables in our schema often containing hundreds of millions or billions of rows, we'd like to keep them as tight as possible. It certainly seems wasteful to populate the fact table with ten or more large alphanumeric degenerate dimensions especially if they are not typically used for reporting or analytics. This is where the reference dimension can be helpful.

The concept of the reference dimension is to break the fact table apart, moving the seldom used degenerate dimension values to a separate reference dimension table with the fact table or reference dimension surrogate key preserving the relationship between the tables. The analytically-valuable degenerate dimensions should not be moved to the reference dimension; they need to be retained in the fact table where they can be used most effectively. It is very important to move only the degenerate dimensions that are not used to support analytic or reporting requirements. While it is generally not considered good dimensional modeling practice to create a dimension with a potential one-to-one relationship with the fact table, in this situation we will accept it as a reasonable tradeoff. The important advantage we gain is dramatically reducing the length of the fact table row resulting in a tighter design that will perform better. This design tradeoff works because the reference dimension should very seldom actually join back to the fact table. If the design assumptions were correct and the degenerate dimensions moved to the reference dimension are not required to support reporting and analytics, most users will never use the reference dimension. There should be just a few occasional investigatory requirements that need to access this information.

In the rare occasion when some component of the reference table is required, it will be necessary to join it to the fact table which may be an expensive and slow running query. User expectations when using the reference dimension need to be carefully managed. Frequent complaints about the performance of the reference dimension likely means some date element needs to migrate back to the fact table as it's important for supporting reporting or analytic needs.

Caution: This design tip should not to be construed as granting design teams permission to build large dimensions with potentially one-to-one relationships with the fact table. Rather, the reference dimension is a specific design response to a particular situation that a design team may confront. We would not expect to see reference dimensions in most dimensional designs. They should be the rare exception rather than the rule.

www.kimballgroup.com Number 85, November 1, 2006

Design Tip #85 Using Smart Date Keys to Partition Large Fact Tables

By Warren Thornthwaite

I've recently had two people ask if it's OK to use a meaningful key for the Date dimension; an integer of the form YYYYMMDD. In one case, my recommendation was no; in the other it was yes. In the no case, the user's goal was directly to provide users and applications with a key in the fact table they could recognize and query directly thereby bypassing the Date dimension. We describe why this actually reduces usability in Design Tip #51. This approach could also hurt performance in some database platforms.

The yes case involved partitioned fact tables. Partitioning allows you to create a table that is segmented into smaller tables under the covers, usually by date. Thus you can load data into the current partition and re-index it without having to touch the whole table. Partitioning dramatically improves load times, backups, archiving old data, and even query performance. It makes it possible to build terabyte data warehouses.

So why does partitioning lead us to consider a smart surrogate key in the Date dimension? It turns out updating and managing partitions is a fairly tedious, repetitive task that can be done programmatically. These programs are much easier to write if the table is partitioned on date and the date key is an ordered integer. If the date key follows a date-style pattern you can also take advantage of date functions in the code.

In SQL Server 2005, for example, the initial definition of a simple partitioned table that holds the first three months of sales data for 2006 might look like this:

```
CREATE PARTITION FUNCTION SalesPartFunc (INT)  AS RANGE RIGHT
FOR VALUES (10000000, 20060101, 20060201, 20060301, 20060401)
```

The values are breakpoints and define six partitions, including a partition to hold the non-date entries in the table (DateKey<10000000), and empty partitions on either side of the data to make it easier to add, drop and swap partitions in the future.

Prior to loading the fourth month of 2006, we would add another partition to hold that month's data. This hard coded version splits the empty partition by putting in the next breakpoint, in this case, 20060501:

```
ALTER PARTITION FUNCTION PFMonthly () SPLIT RANGE (20060501)
```

The following Transact SQL code automatically generates the command using a variable called @CurMonth. It is a bit convoluted because it converts the integer into a date type for the DATEADD function, then converts it to VARCHAR(8) to concatenate it into the SQL statement string. Finally, the EXEC command executes the string.

```
DECLARE
  @CurMonth INT, @DateStr Varchar(8), @SqlStmt VARCHAR(1000)
SET @CurMonth = 20060401
SET @DateStr = CONVERT(VARCHAR(8),DATEADD(Month, 1, CONVERT(datetime,
        CAST(@CurMonth AS varchar(20)), 112)),112)
```

```
SET @SqlStmt = 'ALTER PARTITION FUNCTION PFMonthly () SPLIT RANGE (' + _
        @DateStr + ')'
EXEC (@SqlStmt)
```

So, by using a smart YYYYMMDD key, you still get the benefits of the surrogate key and the advantage of easier partition management. We recommend using your front end metadata to hide the foreign keys in the fact table to discourage your users from directly querying them. If you would like a working example of this code for SQL Server 2005 or Oracle 10g, please send me a note.

As a final note, remember that this logic surrounding the date key cannot be replicated for any other dimension such as Customer or Product. Only the Date dimension can be completely specified in advance before the database is created. Calendar dates are perfectly stable: they are never created or deleted!

www.kimballgroup.com Number 84, October 3, 2006

Design Tip #84 Reader Suggestions for Fact Table Surrogate Keys

By Ralph Kimball and Bob Becker

In July, Bob Becker wrote in Design Tip #81 about fact table surrogate keys. To refresh your memory, he described the circumstances where it makes sense to create a single field in a fact table that is a classic surrogate key, in other words, a simple integer field assigned sequentially as fact table records are created. Such a surrogate key has no internal structure or obvious meaning. Bob pointed out that these surrogate keys had a number of useful purposes, including disambiguating otherwise identical fact records that could arise under unusual business rules, as well as providing a way to confidently deal with suspended load jobs in the ETL back room.

Since July, a number of readers have written to us suggesting additional clever ways in which fact table surrogate keys can be exploited. The reader should note that these concepts primarily provide for behind-the-scene improvements in query performance or ETL support.

Reader Larry pointed out that in Oracle his experience was that a fact table surrogate key can make the database optimizer happier in a situation where declaring the key (the combination of fields that guarantee a row is unique) of the fact table otherwise would require enumerating a large number of foreign keys to dimensions. Some optimizers complain when a large number of B-Tree indexes are defined. With a surrogate key, a single B-Tree index could be placed on that field, and separate bitmap indexes placed on all the dimension foreign keys. Another reader, Eric, similarly reports that in Microsoft SQL Server "Another reason to use a surrogate key with a clustered index on a fact table is to make the primary key of each fact table row smaller, so that the nonclustered indexes defined on the fact table (which contain these primary key values as row identifiers) will be smaller."

Larry also reports that "If I have a user who complains about a report or query being wrong, I can often simply add in the surrogate key column to the report or query, forcing the results to show EVERY row that contributes to it. This has proven useful for debugging." In a similar vein, he also uses the surrogate key as an efficient and precise way to identify a specific fact record that he may wish to point out to the ETL development team as a example of a problem.

And finally Larry reports that "In healthcare (payers), it seems that there is a lot of late arriving dimension data, or changes to dimension data. This always seems to happen on a type two dimension. Often this has caused me to need to update the dimension rows, creating new rows for a prior period forward. A simple query of the fact table can return a list of the surrogate keys for affected rows. Then this can be used to limit the retroactive update to fact rows to only those that need to be touched. I have seen this technique improve this kind of update substantially."

Reader Norman offers an interesting query application depending on fact table surrogate keys. He writes "I wanted to offer up an additional reason for having surrogate keys in the fact tables. This is in cases where it is necessary to join two fact records together in a single query for the purpose of performing calculations across related rows. The preference would be to do these calculations in the ETL and then store them in the fact records, but I have had some requirements where the possible number of stored calculations could have been immense (and thus would have greatly increased the size of the fact tables) and/or the calculations were important yet infrequently performed by the query application, thus needing to be supported with relatively high query speeds, but not at the expense of calculating and storing the calculations. Storing "next record" surrogate keys in the fact table supports these types of requirements." Following Norman's interesting insight, we would assume a fact table row contains a field called NextSurrogateKey which contains the surrogate key of the desired companion record. Constraining on this value immensely simplifies the SQL which otherwise would have to repeat all the dimension constraints of the first record. You would just have an embedded SELECT statement which you would use in a computation as if it were a variable, like (SELECT additional_fact from fact table b where b.surrogatekey = NextSurrogateKey).

And finally reader Dev writes about a similar applications technique where instead of using a fact table surrogate key to link to an adjacent record in the same fact table, he embeds the surrogate key of a related record that resides in another fact table. His example linked a monthly grained fact table that records health care plan measures to a second fact table that records client group benchmarks. Like Norman's example, embedding the surrogate key of the second fact table in the first allowed the applications to be far simpler. Bob and I recommend that, like Norman's example, the link from one fact table to another be handled with an explicit SELECT clause rather than a direct join between fact tables. All too often, we have seen correct but weird results from SQL when two fact tables are joined directly, because of tricky differences in cardinality between the two tables. The explicit SELECT statement should eliminate this issue.

Clearly, while these last two examples enable joins between fact table rows, we are NOT advocating creating surrogate keys for your fact tables to enable fact table to fact table joins. In most circumstances, delivering results from multiple fact tables should use drill across techniques described in Design Tip #68. Keep in mind that in this design tip we are discussing advanced design concepts to support unique business requirements. These concepts need to be carefully considered and tested before being implemented in your environment.

www.kimballgroup.com Number 83, September 7, 2006

Design Tip #83 Abstract Generic Dimensions

By Margy Ross

Childhood guessing games sometimes rely on the distinction of “person, place or thing” for early mystery-solving clues. Some modelers use these same characterizations in their data models by creating abstract person, place and/or thing (typically referred to as product) tables. While generalized tables appeal to the purist in all of us and may provide flexibility and reusability advantages for a data modeler, they often result in larger, more complicated dimension tables in the eyes of the business user.

Let’s consider a generic person or party dimension. Since our employees, customers and supplier contacts are all people, we should store them in the same dimension, right? Similarly, the same argument could be made for the geographic locations where our internal facilities, customers and suppliers reside. While a single table approach might seem clean and logical to data modelers and IT application developers, this abstraction often appears completely illogical to most business analysts for several reasons:

1) We collect and know vastly different information about our own internal entities than we do about external entities. Using a generic model means that some attributes are either nonsensical and/or unpopulated.

2) Generic attribute labels don’t supply adequate meaning and context for the business users and BI applications. For example, if a report illustrates sales by state, it’s unclear whether that refers to the state where the store is located or the state where the customer is located. It’s far preferable if the attribute is clearly labeled to denote its full meaning.

3) Lumping all varieties of people, places or products that our business interacts with inevitably results in larger dimension tables than if they were divided into more discreet logical entities.

Some of you might be familiar of a technique called “role-playing” where the same physical dimension table simultaneously serves multiple roles in the same fact table, such as the date dimension appearing as two uniquely labeled dimensions for the ship date and request date in a single fact table. Other role-playing examples might include the origin and destination airports, or the servicing and authorizing employees, or the dealer that sold the car versus the dealer maintaining it. In each of these examples, a single row in the dimension table could serve in multiple capacities. Role-playing is a very different concept than creating a generic dimension table that’s potentially the Cartesian product of all possible parties.

It’s worth noting that while generic dimensions are not appropriate for the dimensional model viewed by the business, we’re certainly not opposed to their usage in operational systems where they’re behind the scenes and not visible to the business. However, in most legacy situations, the operational source systems do not treat persons, places or things homogenously. It’s more likely that descriptive information about our facility locations, customer locations and supplier locations comes from a variety of different source systems. Attempting to create a generic dimension table from these various sources may create onerous integration challenges for the ETL team without any payback from a business user perspective.

Finally, let’s not forget the dimensional modelers’ mantra – easy to use and fast query performance.

In most cases, abstract dimension tables fail to deliver on either front because the abstraction process reduces clarity for the business users and inevitably creates a larger table. They should be avoided in the dimensional models that are presented to the business in your DW/BI architecture.

www.kimballgroup.com Number 82, August 2, 2006

Design Tip #82 Turning the Fact Table on its Head

By Joy Mundy

The grain of the fact table most often comes directly from the grain of the transaction table from which the data is sourced. Occasionally it makes sense to pivot the facts so that we actually create *more* fact table rows than there are rows in the source.

This counterintuitive occurrence is most likely when the source system isn't a transaction system, such as one that captures sales events, but an analytic system like a forecasting, promotion, or financial analysis system. For example, let's say we're building a fact table to hold budget and actual data. Our source table was designed to support the budget process; it contains financial information by Month, Account and Department with facts ActualAmt and BudgetAmt. The beginning data modeler starts to create a similar structure in the data warehouse database. But interviews with the business community help us understand that there are several versions of budget. We need for the data warehouse to track several drafts of the budget during the budget development process.

The solution here is to create a fact table with four dimensions: Month, Account, Department and Scenario. The new Scenario dimension would have a handful of rows, including "Actual," "Budget Draft 1 FY07," and "Final Budget FY07." Our fact table contains only one measure, Amount. The fact table is normalized to be "long and skinny" rather than "short and fat." The new structure is more flexible and easily accommodates an arbitrary number of draft and final budgets.

It's easy for an outsider to see this solution. It's surprising how difficult it can be, while in the trenches, to pull your mind away from the structure you've been handed and think about creative alternatives. Here are some hints that you should think about this approach to pivot the fact table and add a new dimension.

1. Excessive number of facts. What does excessive mean? A hundred facts in a fact table are excessive; ten is not. Somewhere in the middle, perhaps around thirty measures, you cross into the grey area towards excessive.
2. Naming conventions to group measures. If you have a ton of facts, your fact column names probably use prefixes and suffixes to help users find the facts they're looking for.
3. Many measures in a row are null. Of the, say, 100 facts that could apply to a row, only a subset of them tend to be populated at any one time.

If all these conditions are true, consider normalizing the fact table by creating a fact dimension. The fact dimension could contain several columns that would help users navigate the list of facts. Admittedly, our long-skinny fact table will have a lot more rows than the short-fat one. It will use somewhat more disk space as well, though depending on the sparsity of facts, it may not be all that much bigger. The biggest downside is that many users want to see the facts on a row. It would require pretty good SQL skills to formulate the query to unpivot the data back to get several measures on the same row in a report. Luckily we don't need to write this SQL; a query or reporting tool can do this work for us. And if we use an OLAP database as the user presentation layer, this notion of a fact dimension is entirely natural.

www.kimballgroup.com Number 81, July 6, 2006

Design Tip #81 Fact Table Surrogate Key

By Bob Becker

Meaningless integer keys, otherwise known as surrogate keys, are commonly used as primary keys for dimension tables in data warehouse designs. Our students frequently ask us - what about fact tables? Should a unique surrogate key be assigned for every row in a fact table? Although for the logical design of a fact table, the answer is no, surprisingly we find a fact table surrogate key may be helpful at the physical level. We only recommend creating surrogate keys for fact tables when certain special circumstance described in this design tip apply.

As a quick reminder, surrogate keys are meaningless (aka not-meaningful, artificial, sequence numbers, warehouse, etc.) keys, typically defined as an integer data type, and sequentially assigned by the data warehouse team to serve as the primary keys of the dimension tables. Surrogate keys provide a number of important benefits for dimensions including avoiding reliance on awkward "smart" keys made up of codes from the dimension's source systems, protecting the data warehouse from changes in the source systems, enabling integration of data from disparate source systems, support for type 2 slowly changing dimensions attributes, space savings in the fact tables when these dimension keys are embedded in the fact tables as foreign keys, and improved indexing and query performance.

But in a fact table, the primary key is almost always defined as a subset of the foreign keys supplied by the dimensions. In most environments this composite key will suffice as the primary key to the fact table. There is typically no advantage of assigning a surrogate key to the fact rows at a logical level because we have already defined what makes a fact table row unique. And, by its nature, the surrogate key would be worthless for querying.

However, there are a few circumstances when assigning a surrogate key to the rows in a fact table is beneficial:

1. Sometimes the business rules of the organization legitimately allow multiple identical rows to exist for a fact table. Normally as a designer, you try to avoid this at all costs by searching the source system for some kind of transaction time stamp to make the rows unique. But occasionally you are forced to accept this undesirable input. In these situations it will be necessary to create a surrogate key for the fact table to allow the identical rows to be loaded.
2. Certain ETL techniques for updating fact rows are only feasible if a surrogate key is assigned to the fact rows. Specifically, one technique for loading updates to fact rows is to insert the rows to be updated as new rows, then to delete the original rows as a second step as a single transaction. The advantages of this technique from an ETL perspective are improved load performance, improved recovery capability and improved audit capabilities. The surrogate key for the fact table rows is required as multiple identical primary keys will often exist for the old and new versions of the updated fact rows between the time of the insert of the updated row and the delete of the old row.
3. A similar ETL requirement is to determine exactly where a load job was suspended, either to resume loading or back put the job entirely. A sequentially assigned surrogate key makes this task straightforward.

Remember, surrogate keys for dimension tables are a great idea. Surrogate keys for fact tables are not logically required but can be very helpful in the back room ETL processing.

Design Tip #80 Adding a Row Change Reason Attribute

By Warren Thornthwaite

We are firm believers in the principle that business requirements drive the data model. Occasionally, we'll work with an organization that needs to analyze Type 2 changes in a dimension. They need to answer questions like "How many customers moved last year?", or "How many new customers did we get by month?" which can be difficult with the standard Type 2 control columns. When this happens, we add a control column called RowChangeReason to the design. I was recently approached by someone who realized their business could use a RowChangeReason column and asked if it was possible to add one on to an existing dimension. In this design tip, we'll describe a bulk update technique for retroactively adding a RowChangeReason column to any of your dimensions.

In its simplest version, the RowChangeReason column contains a two character abbreviation for each Type 2 column that changed in a given row. For example if LastName and ZipCode changed, the RowChangeReason would be "LN ZP". You can certainly use more characters if you like, but make sure to put a space between the abbreviations. Although this is not meant to be a user queryable column, it does let us easily identify and report on change events. A question like "How many people changed zip codes last year?" could be answered with the following SELECT statement:

```
SELECT COUNT(DISTINCT CustomerBusinessKey)
FROM Customer
WHERE RowChangeReason LIKE '%ZP%'
AND RowEffectiveDate BETWEEN '20050101' AND '20051231'
```

The LIKE operator and wildcards make the order of the entries unimportant. The RowChangeReason column allows us to answer a lot of interesting questions about behaviors in the dimension table.

We'll use the simple customer dimension shown in Figure 1 as our example. In this table, FirstName is tracked as Type 1, and the other business attributes are tracked as Type 2. Since the table has Type 2 attributes, it also has the necessary Type 2 control columns: RowEffectiveDate, RowEndDate and IsRowCurrent (which we realize is redundant, but we like the convenience). It also includes the new column we are adding on called RowChangeReason.

Customer
CustomerKey
CustomerBusinessKey
FirstName
LastName
City
State
Zip
RowEffectiveDate
RowEndDate
IsRowCurrent
RowChangeReason

Figure 1 – A simple customer dimension

The process to bulk update the new RowChangeReason column takes two passes: One for the new row for each business key, and one for all subsequent changed rows. The first pass joins the dimension to itself using an outer join, treating the table as the current row and the alias as the prior row. The query then finds all the new rows by constraining on all those entries that don't have prior rows using the IS NULL limit in the WHERE clause.

```
UPDATE Customer
SET RowChangeReason = 'NW'
FROM Customer LEFT OUTER JOIN
   Customer PE ON   -- Prior Entry Customer table alias
   Customer.CustomerBusinessKey =
     PE.CustomerBusinessKey
   AND
   Customer.RowEffectiveDate = PE.RowEndDate+1
WHERE PE.CustomerBusinessKey IS NULL
```

The second pass is a bit more complicated because it needs to create a RowChangeReason code for all the rows that have been added to the dimension due to a change in a Type 2 attribute. In this case, we use an inner join between the current row and the prior row which automatically excludes the new rows. We'll also use a CASE statement to generate the string of abbreviations which identifies each of the four Type 2 columns that actually had a change from their prior values. Finally, we'll concatenate the abbreviations together to make the entry for the RowChangeReason column.

```
UPDATE Customer
SET RowChangeReason = Query1.RowChangeReason
FROM
  (SELECT NE.CustomerBusinessKey, NE.RowEffectiveDate,
    (CASE WHEN NE.LastName <> PE.LastName
      THEN 'LN ' ELSE '' END) +
    (CASE WHEN NE.City <> PE.City
      THEN 'CT ' ELSE '' END) +
    (CASE WHEN NE.State <> PE.State
      THEN 'ST ' ELSE '' END) +
    (CASE WHEN NE.Zip <> PE.Zip
      THEN 'ZP ' ELSE '' END) RowChangeReason
  FROM Customer NE INNER JOIN  -- NE for new entry
       Customer PE ON   -- PE for prior entry
       NE.CustomerBusinessKey = PE.CustomerBusinessKey AND
       NE.RowEffectiveDate = PE.RowEndDate + 1
  ) Query1
INNER JOIN Customer ON
     Customer.CustomerBusinessKey = Query1.CustomerBusinessKey AND
     Customer.RowEffectiveDate = Query1.RowEffectiveDate
```

This SQL will work for the one-time process of adding on a RowChangeReason column if you left it off your initial design. Of course, you will still need to add the appropriate comparison logic to your ETL programs to fill the RowChangeReason column moving forward.

www.kimballgroup.com Number 79, May 10, 2006

Design Tip #79 When Do Dimensions Become Dangerous?

By Ralph Kimball

In many organizations, either the customer or product dimensions can have millions of members. Especially in the early days of data warehousing, we regarded these large dimensions as very dangerous because both loading and querying could become disastrously slow. But with the latest 2006 technology, fast processors, and gigabytes of RAM, do we have to worry any more about these large dimensions, and if so, when does a dimension become dangerous?

How big can a dimension get? Consider a typical wide customer dimension describing account holders in a large bank. Suppose there are 30 million account holders (customers) and we have done a good job of collecting 20 descriptive and demographic attributes for each customer. Assuming an average width of 10 bytes for each field, we start with 30 million X 20 X 10 = 6 GB of raw data as input for the ETL data loader. Of course, if we collect 100 attributes instead of 20, our dimension is five times as big, but let's avoid that one for now.

Although OLAP vendors may disagree, I think a 30 million row dimension puts us solidly in dangerous territory for OLAP deployments. Remember that usually any Type 1 or Type 3 changes to a dimension in an OLAP system forces all OLAP cubes using that dimension to be rebuilt. Be careful with Type 1 and Type 3 changes, if you can even build a cube with such a large dimension!

Relational systems do not have this sensitivity to Type 1 and Type 3 changes, but a properly supported ROLAP system needs to place an index (typically a bitmap index) on every field in the dimension. In most relational systems, this level of indexing adds as much as a FACTOR of 3 to the storage size of the dimension. Now we are up to as much as 18 GB for our customer dimension.

Most serious relational deployments should be able to support this 18 GB dimension at query time if the dimension is not also being updated. But two danger scenarios lurk close by: update frequency and slowly changing dimensions.

If your 30 million member dimension requires thousands of insertions, deletions, and updates per week, you need to plan this administration very carefully. The problem is that you can't afford to drop all the indexes every time you load and you probably can't find a way to partition the dimension in order to make the dimension updating more efficient. So you have to perform these admin actions in the presence of all your indexes. Perhaps if you are lucky, you can get your database to defer index updates until after a batch load process runs to completion. You need to study these options carefully, and consider batching certain kinds of updates together.

Perhaps the biggest threat to our dimension, however, comes from Type 2 tracking of attribute changes. Remember that this means every time any attribute in a customer record is updated, we do not overwrite; rather we issue a new record with a new surrogate key. If the average customer record undergoes two updates per year, then in three years, our 30 million rows become 180 million rows, and the 18 GB of storage becomes 108 GB. All the preceding discussions get worse by a factor of six. In this case, I would try very hard to split the customer dimension into at least two pieces, isolating the "rapidly changing" attributes (such as demographics) into an abstract demographics dimension. This takes most of the Type 2 pressure off of the original customer dimension. In a financial reporting environment this is an effective approach because our fact tables are normally periodic snapshots, where we can guarantee that the customer key and the demographics key have a target record in the fact table every reporting period. We have described this process of splitting a dimension many times over the years under the topic “rapidly changing monster dimensions”.

Some of you readers may have successfully wrangled a dangerous dimension of this size or even larger. Please email me describing your technology and your approach and I’ll share it in a Design Tip.

www.kimballgroup.com Number 78, April 1, 2006

Design Tip #78 Late Arriving Dimension Rows

By Bob Becker

Your ETL system may need to process late arriving dimension data for a variety of reasons. This design tip discusses the scenario where the entire dimension row routinely arrives late, perhaps well after impacted fact rows have been loaded.

For example, a new employee may be eligible for healthcare insurance coverage beginning with their first day on the job and be issued a valid insurance card with a valid patient ID. However, the employer may not provide detailed enrollment information to their healthcare insurance provider for several weeks; it may take several more weeks before the new employee is entered into the insurer's operational systems. Of course, the new employee may require health care during this time and submit claims using their patient ID. In this case, the insurer's data warehouse ETL system will receive claim fact row input with a valid patient ID that doesn't have an associated row in the patient dimension – yet. The timing of the enrollment business process and patient setup in the insurer's operational systems naturally runs slower than the claims submission business process.

There are several possible ETL reactions to this problem. The ETL system can reject the fact rows to a suspense file that is reworked on a frequent basis to load the fact rows only after complete patient information has been received and the missing dimension row created. The downside to this approach is the claims fact table will not fully represent the insurance company's true financial exposure.

Alternatively, the ETL system can designate a single dimension row with a description such as "patient unknown" and load the impacted fact rows with the foreign key pointing to this common dimension row. After the patient dimension rows have been created, the affected fact rows need to be revisited and updated with the appropriate patient foreign key. This approach requires retaining the incoming claims fact input in a suspense file to help determine which fact rows need to be attached to the new patient rows. This is an appropriate solution when the incoming natural keys in the fact data input aren't reliable or need to be researched/corrected and thus can't be used to create valid, unique dimension rows.

In the case of the healthcare insurer, the fact row data almost certainly contains a valid patient natural key – it just hasn't been reflected in the patient dimension table, but most likely will at some point. In this situation, we prefer an ETL solution that creates and inserts a new row in the dimension table with only the surrogate key and valid natural key as a placeholder for each unique patient. All incoming fact rows will be loaded (rather than put into a suspense state) with the foreign key linking to the placeholder dimension row for each unique patient natural key. Later, as the business activities and operational source systems catch up, the complete dimension row attributes are populated for the patient. At that time, the placeholder row simply becomes the permanent dimension row. No changes are required to the fact rows as the patient foreign keys in the claims facts will not need to change.

Of course, there are complications to be considered. A detailed discussion is outside the scope of this design tip, but briefly, there may be type 2 version changes to the placeholder dimension row that warrant destructive changes to the foreign key in the associated fact rows. Likewise, if it is determined that a placeholder row was not required for some reason, the placeholder row will need to be deleted or expired and affected fact rows updated. As always, if we change data in the data warehouse, we need to assure the organization's audit and compliance requirements are satisfied.

www.kimballgroup.com Number 77, March 6, 2006

Design Tip #77 Warning – Summary Data Alone Is Hazardous To Your Health

By Margy Ross

An overarching false statement about dimensional models is that they're only appropriate for summarized information. Some people maintain that data marts with dimensional models are intended for managerial, strategic analysis and therefore should be populated with summarized data, not operational details.

We strongly disagree! Dimensional models should be populated with the most detailed, atomic data captured by the source systems so business users can ask the most detailed, precise questions possible. Even if users don't care about the particulars of a single transaction or sub-transaction, their "question of the moment" requires summarizing these details in unpredictable ways. Of course, database administrators may pre-summarize information, either physically or via materialized views, to avoid on-the-fly summarization in every case. However, these aggregate summaries are performance-tuning complements to the atomic level, not replacements.

If you restrict your dimensional models to summarized information, you will be vulnerable to the following shortcomings:

- Summary data naturally pre-supposes the typical business questions. When the business requirements change, as they inevitably will, then both the data model and ETL system must change to accommodate new data.
- Summary data limits query flexibility. Users run into dead ends when the pre-summarized data can't support an unanticipated inquiry. While you can roll up detailed data in unpredictable ways, the converse is not true – you can't magically explode summary data into its underlying components.

When critics authoritatively state that "dimensional models pre-suppose the business question, are only appropriate for predictable usage, and are inflexible," they're conveying the hazards of pre-summarization, not dimensional modeling. If dimensional models contain atomic data, as we advocate, then business users can roll-up or drill-down in the data ad infinitum. They can answer previously unexpected questions without any change to the database structures. When new attribute or measure details are collected by the source system, the atomic data model can be extended without disrupting any existing business intelligence (BI) applications.

Some people advocate an approach where the atomic data is stored in a normalized data model, while summary data is stored dimensionally. The atomic details are not completely ignored in this scenario; however accessing them for user consumption is inherently restricted. Normalized structures remove data redundancies to process transactional updates/inserts more quickly, but the resulting complexity causes navigational challenges and typically slower performance for BI reports and queries. While normalization may save a few bytes of storage space, it defeats the users' ability to seamlessly and arbitrarily traverse up, down and across the detailed and summary data in a single interface. In the world of DW/BI, query functionality/performance trumps disk space savings.

As the architect Mies van der Rohe is credited with saying, "God is in the details." Delivering dimensional models populated with the most detailed data possible ensures maximum flexibility and extensibility. Delivering anything less in your dimensional models undermines the foundation necessary for robust business intelligence (and is hazardous to the health and well-being of your overall DW/BI environment).

www.kimballgroup.com Number 76, February 9, 2006

Design Tip #76 Creating the Advantages of a 64-bit Server

By Joy Mundy

Data Warehouse / Business Intelligence systems love memory. This has been true for decades, since 64 MB was a lot of system memory. It remains true today, when 64 GB is a lot of system memory.

Memory is the most common bottleneck affecting the performance of a DW/BI system. Adding more memory is often the easiest way to improve system performance. DW/BI systems have a strong affinity for 64-bit hardware. The improved processing performance is nice, but 64-bit is particularly important to us because of the vastly larger addressable memory space. In-memory operations are orders of magnitude faster than operations that need to access a lot of disk.

All components of the DW/BI system benefit from additional memory. Memory helps the relational engine answer queries and build indexes much faster. The ETL system can be a significant memory hog. Good design for an ETL system performs transformations in a pipeline and writes data to disk as seldom as possible. OLAP technology uses memory during processing when the cube is computing aggregations and during query time. Even the reporting application can use significant memory. The query that underlies a report uses the memory of the underlying relational or OLAP engine. And if you're using a reporting server to manage and render reports, you may be surprised at how much memory it can consume.

High end hardware for both Windows and Unix systems are dominated by 64-bit, as you can see by reviewing the TPC-H benchmark results. But 64-bit is a no-brainer for DW/BI systems of any size. You can buy a commodity server with four dual core 64-bit processors, 16 GB of memory, and 700 GB of storage for about $25,000. That kind of hardware should easily support a smallish DW/BI system of several billion fact rows and dozens of users.

A server like we've described is particularly compelling for the single vendor "stack" such as Microsoft SQL Server 2005. Microsoft charges a flat rate per processor, no matter how many components of the product you install, or even whether your system is 64-bit or has dual core processors. Many smaller organizations are deploying their DW/BI system on a single server that has the relational database engine, Integration Services for ETL, Analysis Services for OLAP and data mining, and Reporting Services for report management and rendering.

A single-server solution is particularly compelling if you can sequence processing to use system resources efficiently. For example, use Integration Services and the relational engine to load the relational data warehouse. When the heavy ETL work is done, use Analysis Services and the relational engine to process incremental cube updates. Next, pre-execute and cache the big, popular, and demanding reports. This step will use Reporting Services and Analysis Services or the relational engine, depending on where the reports are sourced from. During the workday, rely primarily on Analysis Services to support ad hoc analysis and on-demand reporting. Integration Services is not busy during the day at all, and by pre-caching reports you're reducing the daytime load on Reporting Services and the relational engine.

Although we've characterized the 64-bit hardware as a no-brainer even for small to medium systems, there are some things you need to worry about. The first question is the chip architecture. Those of us who've grown comfortable with the simplicity of the "Wintel" world are faced with new decisions: Itanium, x64, or AMD64? These are fundamentally different chips, and the operating system and application code (such as SQL Server) must be separately compiled, tested, and supported. You're betting that the operating system and database software will continue to support your chosen chipset for the expected life of your hardware. Consider software requirements beyond the core server software, such as specialized ETL functionality or database management tools.

A secondary issue is the architecture of your development and test environments. Your DW/BI test system must use the same architecture as your production system. In an ideal world, the development database servers would also use the same architecture, but it's common to use cheaper systems for development. As an aside, it's amazingly common to use a different architecture for the test system—including not having a test system at all—but that's just asking for trouble.

For a large DW/BI system, 64-bit is the only way to go. But even for smaller systems, a modest investment in a 64-bit server and a decent amount of memory will pay for itself in system performance. We used to make fun of people who'd buy bigger hardware so they don't have to tune their DW/BI system. But in the case of 64-bit and large memory, it's a sensible thing to do.

www.kimballgroup.com Number 75, January 13, 2006

Design Tip #75 Creating the Metadata Strategy

By Warren Thornthwaite

In most cases, metadata is a neglected area of the DW/BI system; in a few cases, it's an over-engineered monstrosity. In this design tip we offer an approach to dealing with metadata that we believe is a reasonable compromise between having little or no managed metadata and building an enterprise metadata system. Our recommendation concentrates on business metadata first, making sure it is correct, complete, maintained, and accessible to the business users. Once that's done, provide a way to view the other major metadata stores. Here's a straight-forward, business-value based approach:

1. Use whatever tools you have to survey your system to identify and list the various locations, formats, viewers, editors, owners, and uses of metadata. Where there aren't any tools, you will need to create query or programmatic access to the metadata sources so you can explore and track them.
2. Identify and/or define missing metadata elements that need to be captured and managed. These are typically business elements that will be used more broadly and therefore need to be updated and distributed throughout the system.
3. Once you have a solid list of metadata elements in place, decide on the master location for each. This is the location where the element will be stored and edited. It is the source for any copies needed by other parts of the system. It might be in the relational database for some elements, in the front end tool for others, or even in your organization's repository tool. Try to use all available pre-existing metadata structures like description fields before adding your own metadata tables.
4. Create systems to capture and maintain any business or process metadata that does not have a home. These can be simple front ends that let the user directly edit metadata in its master location. You'll want some data quality checks and a good metadata backup system in place, just in case.
5. Create programs or tools to share and synchronize metadata as needed. This primarily involves copying metadata from its master location to whatever subsystem needs it. The goal is to use the metadata in the master locations to fill in the description, source, business name, and other fields in all the tables and object models all the way out to the front end tools. If the master location is populated right from the start as part of the design and development process, the metadata will be easier to synchronize and maintain on an ongoing basis. Note that copying the metadata from one location to another is an ideal task for your ETL system.
6. Educate the DW/BI team and key business users about the importance of metadata and the metadata strategy. Work with the data steward to assign metadata creation and updating responsibilities.
7. Design and implement a delivery approach for getting business metadata to the user community. Typically, this involves sourcing your front end tool's metadata structures from the master metadata locations. Often, it helps to create a simple metadata repository for

business metadata and provide users with a way to browse the repository to find out what's available in the DW/BI system.

8. Manage the metadata and monitor usage and compliance. Make sure people know the information is out there and are able to use it. Make sure the metadata is complete and current. A large part of the baseline metadata effort is spent building reports and browsers so people can look at the metadata. Managing the metadata means looking at it regularly and making sure it is complete and current.

Even though this is the balanced strategy between nothing and too much, it is still a fair amount of work. Make sure there's time in your project plan's development tasks to capture and manage metadata, including separate tasks for the above steps. And finally, make sure someone on the DW/BI team is assigned the role of Metadata Manager and owns the responsibility for creating and implementing the metadata strategy.

www.kimballgroup.com Number 74, December 16, 2005

Design Tip #74 Compliance-Enabled Data Warehouses

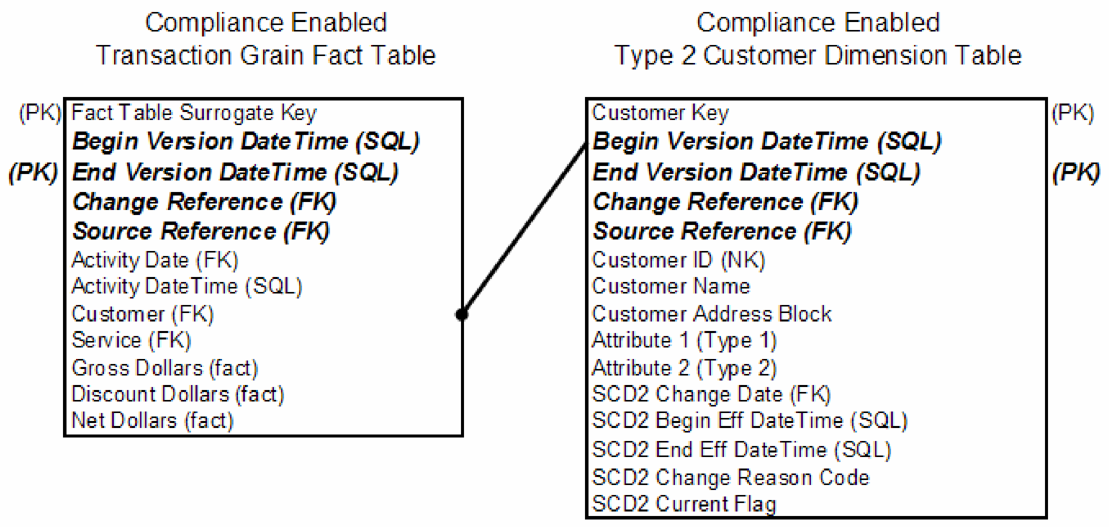
By Ralph Kimball

I often describe compliance in the data warehouse as “maintaining the chain of custody” of the data. In the same way a police department must carefully maintain the chain of custody of evidence in order to argue that the evidence has not been changed or tampered with, the data warehouse must also carefully guard the compliance-sensitive data entrusted to it from the moment it arrives. Furthermore the data warehouse must always be able to show the exact condition and content of such data at any point in time that it may have been under the control of the data warehouse. Finally when the suspicious auditor is looking over your shoulder, you need to link back to an archived and time stamped version of the data as it was originally received, which you have stored remotely with a trusted third party. If the data warehouse is prepared to meet all these compliance requirements, then the stress of being audited by a hostile government agency or lawyer armed with a subpoena should be greatly reduced.

The big impact of these compliance requirements on the data warehouse can be expressed in simple dimensional modeling terms. Type 1 and Type 3 changes are dead. Long live Type 2. In other words, all changes become inserts. No more deletes or over-writes.

In plain English, the compliance requirements mean that you cannot actually change any data, for any reason. If data must be altered, then a new version of the altered records must be inserted into the database. Each record in each table therefore must have a begin-timestamp and an end-timestamp that accurately represents the span of time when that record was the “current truth.”

The following figure shows a compliance-enabled fact table connected to a compliance-enabled dimension table. The fields shown in bold-italics in each table are the extra fields needed for compliance tracking. The fact table and the dimension table are administered similarly.



The ***Begin Version DateTime*** and ***End Version DateTime*** fields describe the span of time when the record in question was the “current truth.” The ***Change Reference*** field is a foreign key pointing to a Change Reference dimension (not shown) that describes the change status of that record. The ***Source Reference*** field is a foreign key pointing to a Source Reference dimension (not shown) that shows the off-site trusted third party location where the hash-encoded and time stamped version of this record has been stored since the moment when it was created.

The first time that records are loaded into either of the tables, the ***Begin Version DateTime*** is set to the time of the load, and the ***End Version DateTime*** is set to an arbitrary date far in the future, such as midnight, December 31, 9999. The Change Reference would describe the change status as “initial load.” If and when a Type 1 correction needs to be made to either facts in the fact table or Type 1 or 3 attributes in the dimension table, a new record is inserted into the respective table with the same surrogate key, but a new pair of ***Version DateTimes***. Thus, the true primary key of the tables has become the combination of the original primary key field and the ***End Version DateTime***, as shown in the figure. When the new record is inserted, the prior most recent ***End Version DateTime*** field must be changed to this new load ***DateTime***. This two-step dance of adjusting the ***Begin*** and ***End DateTime*** stamps is familiar to data warehouse architects from the normal Type 2 dimension processing.

The presence of existing Type 2 attribute processing in the dimension does not change anything. Only Type 1 and Type 3 updates invoke the special steps described in this Design Tip. Type 2 processing proceeds as it always has, by introducing new dimension member records with new surrogate primary keys, and by administering the familiar SCD Type 2 metadata fields shown in the bottom portion of the dimension in the figure.

If any of you are uncomfortable with the seeming introduction of a two part key in the dimension table (a violation of the standard dimensional design that insists on a single field surrogate key), then think about it in the following way. Although technically one must view the tables as now having two-part primary keys, think about this schema as still behaving like typical one-part key designs ***after the version has been constrained***. That’s the point of this Tip. First, you choose a version ***as of*** a certain date, then you run your typical queries with the same old keys as always.

Before concluding this Design Tip, let’s not overlook the BIG benefit of this approach! When the hostile government agency or lawyer armed with a subpoena demands to see what has happened to the data, your tables are fully instrumented to comply. If you must reveal the exact status of the data as of, say, January 1, 2004, then you just constrain this date to be between the ***Begin*** and ***End Version DateTimes*** of all the tables. Presto – the database reverts to that moment in history. And, of course, any subsequent changes made to that data are fully explained by the ***Change Reference***. Finally, you can prove that the evidence (oops, I mean data) hasn’t been tampered with by using the ***Source Reference***.

Good luck with your compliance.

Design Tip #73 Relating to Agile Methodologies

By Margy Ross

I've fielded several questions recently regarding agile development methodologies. People seem to want a quick binary response: do we support and approve of agile methods or not? Unfortunately, our reaction is not so clearly black-and-while. One thing for certain is that the agile approach has enthusiastic supporters. Tackling the topic via a Design Tip might be akin to discussing someone's religion after having read a bit about it on the Internet – likely a fool-hardy proposition, but we'll jump in regardless.

First of all, what is agile software development? As with most things related to information technology, agile development takes on slightly different meanings depending on who you talk to or what you read. In general, it refers to a group of methodologies, including Extreme Programming, SCRUM, Adaptive Software Development and others, which share a common focus on iterative development and minimizing risk by delivering new functionality in short timeframes, often measured in weeks. These approaches were initially referred to as 'light-weight methodologies' in contrast to more regimented, documentation-intensive traditional methods. The term 'agile' was adopted in 2001 when a group of prominent thought leaders convened to discuss the common threads of their methodologies. The group published the Agile Manifesto (www.agilemanifesto.org) to encapsulate their shared beliefs and the non-profit Agile Alliance was established. Scott Ambler, author of several books on the subject, sums it up in a sound bite: "Agile is an iterative and incremental (evolutionary) approach to software development which is performed in a highly collaborative manner with 'just enough' ceremony."

There are many principals or tenets of the agile approach that resonate and tightly align with the Kimball Method's standard techniques:

- Focus on the primary objective of delivering business value. This has been our mantra for decades.
- Value collaboration between the development team and stakeholders, especially business representatives. Like the agile camp, we strongly encourage a close relationship and partnership with the business.
- Stress the importance of ongoing face-to-face communication, feedback and prioritization with the business stakeholders. While the Kimball Method encourages some written documentation, we don't want the burden to be overly onerous (or pointless).
- Adapt quickly to inevitably-evolving requirements.
- Tackle development of re-usable software in an iterative, incremental manner with concurrent, overlapping tasks. As an aside, standardizing on the 'agile' nomenclature was a marketing triumph; wouldn't you rather be agile than spiraling or stuck in a waterfall (methodology)?

So what's the bottom line? In reality, one size seldom fits all, despite what the label claims. From my vantage point, there's a time and place for agile techniques when creating a DW/BI system. They seem to naturally fit with the front-end business intelligence layer. Designing and developing the analytic reports and analyses involves unpredictable, rapidly changing requirements. The developers often have strong business acumen and curiosity, allowing them to communicate effectively with the business users. In fact, we suggest that BI-focused team members cohabitate with the business so they're readily available and responsive; this in turn encourages more business involvement. It's reasonable to deliver functionality in a matter of weeks. At the other end of the spectrum, the real-world wrangling of the data is inherently more complex and dependent on order. While we support reducing ETL kitchen development time, the essential tasks realistically take months, not weeks, in our experience.

One final word of caution: some DW/BI development teams have naturally fallen into the trap of creating analytic or reporting solutions in a vacuum. In most of these situations, the team worked with a small set of users to extract a limited set of source data and make it available to solve their unique problems. The outcome is often a stand-alone data stovepipe that can't be leveraged by others or worse yet, delivers data that doesn't tie to the organization's other analytic information. We encourage agility, when appropriate, however building isolated data sets must be avoided. As with most things in life, moderation and balance between extremes is almost always prudent.

www.kimballgroup.com Number 72, October 7, 2005

Design Tip #72 Business Process Decoder Ring

By Bob Becker

In Design Tip # 69, *Identifying Business Processes*, Margy discussed the importance of recognizing your organization's business processes and provided guidelines to spot them. We dive into more details here.

Focusing on business processes is absolutely critical to successfully implement a DW/BI solution using the Kimball Method. Business processes are the fundamental building block of a dimensional data warehouse. We suggest you build your data warehouse iteratively, business process at a time. You may wonder, "What's so magical about business processes? How does identifying the business process help in our dimensional modeling activities?". The answer is that correctly identifying the business processes launches the entire dimensional design. The lightly-held secret is that each business process will result in at least one fact table. Essentially, identifying the business processes identifies the fact tables to be built.

It's not unusual for a single business process to result in more than one fact table. This occurs most frequently when the process involves heterogeneous products, also know as super types and sub types. In this case, several similar but separate fact tables will be spawned. Health care provides a good example. The Paid Claims business process may result in three fact tables: professional claims (e.g., doctors' office visits), institutional claims (e.g., hospital stays) and drug claims.

The consequences of incorrectly identifying the business process are designs that never come to fruition or (worse) a fact table designed at an inappropriate level of detail. Given the volumes we've written on the topic, it should go without saying (although you wouldn't know it by reading some of the recent analyst reports that have crossed our desks) that we advocate implementing data at its most atomic level of detail in your dimensional data warehouse.

A good test of whether you have accurately identified a business process is to state the fact table's grain. If you can succinctly state the grain, you are in good shape. On the other hand, if there is confusion regarding the level of detail of a single row in the fact table, you're not there yet. Most likely you are mixing more than a single business process. You need to step back and look carefully for additional business processes that may be involved.

Listening to your business users' requirements is the best way to develop an understanding of the business processes. Unfortunately, as the business requirements unfold, they don't take the shape of business processes as we describe them. The business users typically describe analytic requirements: the types of analysis and decisions they want to make using data in their terms. For example, an actuarial analyst in a health care organization may describe their needs for ratings analysis, utilization trend reporting, and the claim triangles they rely on. But none of these analytic requirements seem to describe a business process. The key is to decode the requirements, decomposing them into the appropriate business processes. This means digging a bit deeper to understand the data and operational systems that support the analytic requirements. Further analysis in our example ultimately shows that all three analytic requirements are served by data from the business process: paid claims.

Sometimes the analytic requirements are more challenging to decode. Often the most valuable analytic requirements described by business users cross multiple business processes. Unfortunately, it's not always obvious that multiple business processes are involved. The decoding process is more difficult because it requires decomposing the analytic requirements into all of the unique types and sources of data required. In our health care example, underwriting requires a loss ratio analysis to support renewal decisions. Digging further into the details, we can determine the loss ratio analysis compares clients' premium revenue against their claim expenses to determine the ratio between revenues and expenses. In this case, two business processes are required to support the analytic requirement: paid claims and premium billing.

After reflecting on this design tip you should be able to discern the challenges of creating a corporate dashboard. A dashboard is not a single business process; rather it is a display mechanism for presenting the results of many or most of the business processes in the organization. Unfortunately, the dashboard is usually presented (and accepted) by the DW/BI team as a single analytic requirement.

www.kimballgroup.com Number 71, September 2, 2005

Design Tip #71 The Naming Game

By Warren Thornthwaite

The issue of field naming rears its ugly head while you're creating the dimensional data model. Naming is complex because different people have different meanings for the same name, like revenue, and different names with the same meaning, like sales. The difficulty comes from human nature: most of us don't want to give up what we know and learn a new way. The unenviable task of determining names typically falls on the data steward. If you are responsible for dealing with this political beast, you will find the following three-step approach helpful. Steps 1 and 2 generally happen before the model is presented to the business users. Step 3 usually happens after business users have seen and understand the model.

Step 1—Preparation

Begin by developing skills at thinking up succinct, descriptive, unique names for data elements. Learn your organization's (and team's) naming conventions. Study the table and column names in the various systems. If you don't have established naming conventions, now's a good time to do so. A common approach is to use a column name standard with three parts:

PrimeWord_ZeroOrMoreQualifiers_ClassWord

The prime word is a categorization word that often corresponds to the entity the column is from, and in some cases, qualifiers may not be necessary. So the field in the Sales Fact table that represents the amount sold might be Sales_Dollar_Amount. You can research different naming conventions on the Internet. Here are some links to get you started:

https://dwr.ais.columbia.edu/info/Data%20Naming%20Standards.html
http://www.ss64.com/orasyntax/naming.html

Step 2—Creating a Solid Starting Point Name Set

During the modeling process, work with the modeling team (including a representative or two from the business) to draft an initial set of names and the rationale. Once the model is near completion, hold a review session with the modeling team to make sure the names makes sense -- this is also good practice for the next step.

In addition to the review session, it helps to have one-on-one meetings with the key stakeholders. This typically includes the core business users and any senior managers whom you have a sense might have an opinion. If their preferred name for any given column is different from your suggested name, try to figure out why. Help them be clear on their definition of the data element by asking them to explain what the term means to them. Look for missing qualifiers and class words to clarify the meaning. For example, a sales analyst would be interested in Sales numbers, but it turns out that this Sales number is really Sales_Commissionable_Amount, which is different from Sales_Gross_Amount and Sales_Net_Amount.

The resulting name set should be used by the data modeling team to update the current version of the data model. Keep track of the alternative names for each field and the reasons people offered for their preferred choices. This will be helpful in explaining the derivations of the final name set.

Step 3—Building Consensus

Once you have a solid, tested name set, and the core users have seen the data model presentation, gather all the stakeholders in a conference room for at least half a day (count on more if you have a lot of columns or a contentious culture) and work through it. Start from the high level model and progress through all the columns, table by table. Generally, there have been enough iterations of model reviews and naming discussions so that many of the issues have already been resolved and the remaining issues are reasonably well understood.

The goal of this session is to reach consensus on the final name set. Often this means someone has to accept the will of the majority and let go of their favorite name for a given column. It is surprising how emotional this can be. These names represent how we view the business, and people feel pretty strongly about getting them “right.” Don’t let people get out of the room without reaching agreement if it is at all possible. If you have to reconvene on the same issues, it will take extra time to re-hash the various arguments.

Once you have reached agreement on the final name set, document it carefully and take it back to the data modelers so they can work it into the final data model.

www.kimballgroup.com Number 70, August 3, 2005

Design Tip #70 Architecting Your Data For Microsoft SQL Server 2005

By Joy Mundy

Like many of you, Warren Thornthwaite and I (Joy) have been working with the pre-release versions of Microsoft SQL Server 2005. We're also putting finishing touches on our new book, The Microsoft Data Warehouse Toolkit (Wiley, December 2005). One of the issues we've grappled with, for the book and our consulting SQL Server clients, is whether you should implement your Microsoft-based DW/BI system in the relational database, Analysis Services, or both? The answer in most cases is both.

For years people have been building relational data warehouses, so obviously that's possible. Microsoft has added some interesting functionality into the Analysis Service 2005 dimensional database engine that makes it possible—in theory at least—to implement the dimensional database directly from a transactional system, skipping the relational data warehouse. But is that a good idea?

Before I answer that question, I'm going to argue that in any case your Microsoft DW architecture should include Analysis Services. For successful ad hoc access against even a simplified dimensional model, you need a user-oriented layer that streamlines navigation, performs complex calculations and aggregate navigation, and manages data security. For many years people have used relational techniques like views and client-side query tools to deliver this functionality. A dimensional engine like Analysis Services provides an attractive alternative via the following features:

- User-oriented metadata
- Complex analytics stored in the database
- Richness of the analytic language (MDX rather than SQL)
- Query performance
- Aggregate management
- Rich security model
- Server-based—important for scalability, performance, and sharing of metadata across multiple query tools.

Although Analysis Services has become a popular component of SQL Server 2000, there are still several common objections to using Analysis Services:

- Scalability
- Duplication of data
- Changing existing end-user applications.

We find only the third objection to be broadly compelling. Worries about scalability and data duplication have been addressed effectively in SQL Server 2005, and shouldn't prevent the vast majority of implementations from reaping the very real benefits of a data warehouse / business intelligence system built on Analysis Services.

If I've convinced you that Analysis Services is a vital part of your Microsoft data warehouse architecture, your next question may be: why do you even need to store the dimensional data in the RDBMS? You aren't required to do so: Microsoft provides several mechanisms for populating Analysis Services cubes directly from non-dimensional source systems. Why go to the trouble and expense of populating the relational store in addition to Analysis Services? Here's why:

- Conformation of dimensions and facts. In a hypothetical, simple example you could conform data on the way into the Analysis Services database. In the real world, you will have to update and delete some data in the ETL pipeline, and you really want to do this in a relational database.

- Disaster recovery. The tools and knowledge for managing a relational database for easy recovery are better than those for Analysis Services, though management tools are much improved in the new version.
- Comfort. DBAs and power users are very familiar with SQL and relational databases, and may violently resist the elimination of the relational layer.
- Query flexibility. If you want to modify an Analysis Services database, you usually need to redeploy and reprocess a large chunk of the database. It's much easier to "join and go" in the relational world.
- Future flexibility. The notion of eliminating the relational data warehouse and populating the Analysis Services database directly from transaction systems may sound elegant. But if you choose this architecture, you're sailing on a Microsoft ship with a non-refundable ticket.

There are a few scenarios where the best choice is to skip the relational storage of the dimensional data. But these are edge cases which I'll write about in a future Design Tip. Most of us, most of the time, should plan to store and manage the dimensional data in the relational database, and use that store to feed Analysis Services. Think of the Analysis Services layer primarily as metadata for the dimensional engine, with an ephemeral data cache—similar in spirit to relational indexes—that greatly improves query performance.

www.kimballgroup.com Number 69, July 5, 2005

Design Tip #69 Identifying Business Processes

By Margy Ross

Readers who follow the Kimball approach can often recite the 4 key decisions when designing a dimensional model: identify the business process, grain, dimensions and facts. While this sounds straightforward, teams often stumble on the first step. They struggle to articulate the *business process* as it's a term that seems to take on different meaning depending on the context. Since the business process declaration is the first stake in the ground when designing a dimensional model, we want to eliminate confusion in our context.

First, let's begin by discussing what a business process is not. When designing a dimensional model, the business process does not refer to a business department, organization or function. Likewise, it shouldn't refer to a single report or specific analysis.

For a dimensional modeler, the business process is an event or activity which generates or collects metrics. These metrics are performance measurements for the organization. Business analysts inevitably want to scrutinize and evaluate these metrics by a seemingly limitless combination of filters and constraints. As dimensional modelers, it's our job to present these metrics in an easy-to-understand structure that responds quickly to unpredictable inquiries.

When identifying the business process for dimensional modeling, some common characteristics and patterns often emerge.

1) Business processes are typically supported by an operational system. For example, the billing business process is supported by a billing system; likewise for the purchasing, ordering, or receiving business processes.

2) Business processes generate or collect unique measurements with unique granularity and dimensionality used to gauge organizational performance. Sometimes the metrics are a direct result from the business process. Other times, the measurements are derivations. Regardless, the business processes deliver the performance metrics used by various analytic processes. For example, the sales ordering business process supports numerous reports and analytics, such as customer analysis, sales rep performance, and so on.

3) Business processes are frequently expressed as action verbs with the associated dimensions as nouns describing the who, what, where, when, why and how related to the process. For example, the billing business process results will be sliced-and-diced and analyzed by date, customer, service/product, and so on.

4) Business processes are usually triggered by an input and result in output that needs to be monitored. For example, an accepted proposal is input to the ordering process which results in a sales order and its associated metrics. In this scenario, the business process is sales ordering; you'll have an orders fact table with the sales order as a potential degenerate dimension and the order amounts and counts as facts. Try to envision the general flow from input into a business process, resulting in output metrics. In most organizations, there's a series of business processes where outputs from one process become inputs to the next. In

the parlance of a dimensional modeler, these business processes will result in a series of fact tables.

5) Analysts sometimes want to drill across business processes, looking at the result of one process alongside the results of another. Drilling across processes is certainly viable if the dimensions common to both processes are conformed.

Determining your organization's core business processes is critical to establishing your overall framework of dimensional models. The easiest way to determine these processes is by listening to the business users. Which processes generate the performance metrics they're most interested in monitoring? At the same time, the data warehouse team should be assessing the realities of the source environment to deliver the data coveted by the business.

One final comment… It should go without saying that the ever-popular dashboard is NOT a business process; it presents the performance results of numerous individual business processes.

www.kimballgroup.com Number 68, June 3, 2005

Design Tip #68 Simple Drill-Across in SQL

By Warren Thornthwaite

Drill-across refers to the process of querying multiple fact tables and combining the results into a single data set. A common example involves combining forecast data with actual data. The forecast data is typically kept in a separate table, captured at a different level of detail than the actual data. When a user wants a report that compares actual and forecast by customer, the query needs to go against two fact tables. (Note: Data from the two fact tables can only be combined if they are built using conformed dimensions. The Customer, Date and any other shared dimensions must be exactly the same in both star schemas.)

The most efficient way to combine data from the two fact tables is to issue separate queries against each fact table, then combine the two results sets by matching up their shared attributes. This tends to work best because most database optimizers recognize the single star query and quickly return the two results sets.

The following SQL is used to drill-across two star schemas, Actual Sales and Forecast, both with Customer and Date dimensions. The query uses SELECT statements in the FROM clause to create two sub-queries and join their results together, exactly as we'd like. Even if you don't have to write the SQL yourself, you'll get a sense for what your BI tool might be doing.

```
SELECT Act.Customer, Act.Year, Act.Month, Actual_Amount, Forecast_Amount
FROM
-- Subquery "Act" returns Actuals
 (SELECT Customer_Name AS Customer, Year, Month_Name AS Month,
    SUM(Sale_Amount) Actual_Amount
    FROM Sales_Facts A
    INNER JOIN Customer C
    ON A.Customer_Key = C.Customer_Key
    INNER JOIN Date D
    ON A.Sales_Date_Key = D.Date_Key
   GROUP BY Customer_Name, Year, Month_Name) Act
INNER JOIN
-- Subquery "Fcst" returns Forecast
 (SELECT Customer_Name AS Customer, Year, Month_Name AS Month,
    SUM(Forecast_Amount) Forecast_Amount
    FROM Forecast_Facts F
    INNER JOIN Customer C
    ON F.Customer_Key = C.Customer_Key
    INNER JOIN Date D
    ON F.Sales_Date_Key = D.Date_Key
  GROUP BY Customer_Name, Year, Month_Name) Fcst
-- Join condition for our small result sets
  ON Act.Customer = Fcst.Customer
  AND Act.Year = Fcst.Year
  AND Act.Month = Fcst.Month
```

This should perform almost as fast as doing the two individual queries against the separate fact tables because the join is on relatively small subset of data that’s already in memory. The results should look like this:

Act sub-query:

Customer	Year	Month	Actual_Amount
Big Box	2005	May	472,394
Small Can	2005	May	1,312,034

Fcst sub-query:

Customer	Year	Month	Forecast_Amount
Big Box	2005	May	435,000
Small Can	2005	May	1,257,000

Final drill-across query results:

Customer	Year	Month	Actual_Amount	Forecast_Amount
Big Box	2005	May	472,394	435,000
Small Can	2005	May	1,312,034	1,257,000

For relational-based star schemas, many front-end BI tools can be configured to issue the separate SQL queries through their metadata, or at least in their user interface. Many OLAP engines do this through a “virtual cube” concept that ties the two underlying cubes together based on their shared dimensions.

Remember, if you do not have rigorously enforced conformed dimensions, you may not be comparing apples to apples when you drill-across!

www.kimballgroup.com Number 67, May 4, 2005

Design Tip #67 Maintaining Back Pointers to Operational Sources

By Ralph Kimball

Our data warehouses are increasingly oriented toward tracking detailed customer transactions in near real time. And as Patricia Seybold points out in her wonderful book, *Customers.com* (Times Business, 1998), managing customer relationships means having access to the data from all the “customer facing processes” in an organization.

The combination of keeping the detail behind all the customer facing processes, but at the same time providing an integrated view, presents an interesting challenge for the ETL architect. Suppose we have a typically complex customer oriented business with fifteen or more customer facing systems including store sales, web sales, shipments, payments, credit, support contracts, support calls, and various forms of marketing communications. Many of these systems create their own natural key for each customer, and some of the systems don’t do a particularly good job of culling out duplicated entries referring to the same customer. There may be no reliable single customer ID used across all customer facing source systems.

The ETL architect faces the daunting task of de-duplicating customer records from each separate source system, matching the customers across the systems, and surviving the best and most reliable groups of descriptive attributes “cherry picked” from each of the systems. I describe the details of de-duplicating, matching, and surviving in ETL subsystem #8 in my ETL classes and book.

The dilemma for the ETL architect is that even after producing a perfect final single record for the customer, the end user analyst may be unable to trace backward from the data warehouse to a set of interesting transactions in just one of the source systems. The de-duplication and survival steps of preparing the final clean customer master may make subtle changes in names, addresses, and customer attributes that decouple the data warehouse from the original dirty transactions back in the source systems.

The recent demand by the end user community to make all customer transaction detail available in the data warehouse means that we need somehow to carry forward all the original source IDs for the customer into the final customer master dimension. Furthermore, if the source systems have generated duplicate records for the same customer (which we find and fix in the ETL pipeline), we need to store all of the original duplicate source system IDs in the customer master dimension. Only by maintaining a complete set of back pointers to the original customer IDs can we provide the level of trace-back service that the end user analysts are demanding.

I recommend creating a single cross reference table to hold all the original customer IDs. This table has the fields

Data Warehouse Natural Customer Key
Source System Name
Source System Customer ID

The Data Warehouse Natural Customer Key is a special natural key created by the data warehouse! We need such a permanent, unchanging key in the data warehouse master customer dimension to unambiguously identify Slowly Changing Type 2 versions of a given customer.

This table can be queried directly, constraining the Data Warehouse Natural Customer Key, or by joining this field to the same field in the master customer dimension. In both cases, the Source System fields will give the complete list of back pointers.

This design has the advantage that simply by adding data rows to our little cross reference table, it flexes gracefully to handle messy duplicated versions of customer IDs in the source systems, as well as the incorporation of new source systems at various points in time.

www.kimballgroup.com Number 66, April 11, 2005

Design Tip #66 Implementation Analysis Paralysis

By Bob Becker

Many data warehouse teams lean heavily toward the doing side. They leap into implementation activities without spending enough time and energy to develop their data models, identify thorough business rules, or plan their data staging processes. As a result, they charge full speed ahead and end up re-working their processes, delivering bad or incomplete data, and generally causing themselves difficulty.

Other project teams have the opposite challenge. These teams are committed to doing their homework in all the critical areas. They are focused on data quality, consistency, completeness and stewardship. However, these project teams sometimes bog down on issues that should have been resolved long ago. Of course, this impasse occurs at the worst time – the promised implementation dates are rapidly approaching and design decisions that should be well into implementation remain unresolved.

Naturally, the outstanding issues involve the most difficult choices and the project team disagrees on the best solutions. The easy issues have already been resolved and the solutions for the more difficult issues don't come as easily. Despite copious amounts of time spent in research, data profiling, design meetings and informal discussions, nothing seems to move the team closer to a decision on the best approach. The project sits at a crossroads unable to move forward. By this time, fear has usually taken hold of the project team. The pressure is on.

One helpful approach is the use of an arbitrator (a trusted individual from outside the project team) to help move the team ahead. The outstanding issues are identified and meetings scheduled with the arbitrator and interested stakeholders. All participants must agree that a final decision will be made during these sessions. The arbitrator should establish a time box to limit discussion on each issue. Discuss the pros and cons of each approach one last time; the arbitrator makes the ultimate decision if the team can't reach consensus.

Another approach is to defer the unresolved issues until a future implementation after further research and discussion have identified an appropriate solution. The downsides to this approach are that the business requirements may not allow the issues to be deferred; postponing resolution may simply delay the inevitable without any significant gain.

There is a delicate balance between planning and doing in the data warehouse world. The goal is to identify reasonable solutions, not necessarily perfect solutions, so the team can transition from planning to implementation. There may still be more to learn, but the implementation process is often more effective at revealing the weak spots in the plan so they can be reinforced than any amount of talking and planning. In fact, for many of the hard choices, much of the information needed to make good choices can only be gained through trial and error.

Clearly, we are not advocating a casual, ad hoc approach to implementing the data warehouse. But we recognize that sometimes you must be pragmatic and move forward with less than ideal solutions that may need to be revisited to achieve your overall goals.

www.kimballgroup.com Number 65, March 6, 2005

Design Tip #65 Document Your ETL System

By Joy Mundy

Whether you use an ETL tool or hand-code your ETL system, it's a piece of software like any other and needs to be documented. As your data warehouse evolves, the ETL system evolves in step; you and your colleagues need to be able to quickly understand both the entire system architecture and the gritty details.

There's a widespread myth that ETL tools are self-documenting. This is true only in comparison with hand-coded systems. Don't buy into this myth: you need to develop an overall, consistent architecture for your ETL system. And, you need to document that system. Yes, writing a document.

The first step in building a maintainable ETL system is to STOP and think about what you're doing. How can you modularize the system? How will those modules fit together into an overall flow? Develop your system so that you use a separate package, flow, module (or whatever your tool calls it) for each table in the data warehouse. Write a document that describes the overall approach – this can be a few pages, plus a screenshot or two.

Design a template module and group like activities together. The template should clearly identify which widgets are associated with extracts, transformations, lookups, conformation, dimension change management, and final delivery of the target table. Then, document this template flow in painstaking detail, including screenshots. The documentation should focus on what's going on, not on the detailed properties of each step or task.

Next, use the templates to build out the modules for each dimension and fact table. If you can control layout within your ETL tool, make the modules look similar, so people can look in the top-left for the extract logic, and can more easily understand the squiggly mess in the middle. The modules for each dimension table should look really similar to each other; likewise for fact tables. Remember: it's only a *foolish* consistency that's the hobgoblin of small minds. The table-specific documentation should focus on what's different from the standard template. Don't repeat the details; highlight what's important. Pepper your ETL system with annotations, if your ETL tool supports them.

Finally, your ETL tool may support some form of self-documentation. Use this feature, but consider it an appendix to the real document as it's either relatively lame (screenshots) or overwhelmingly detailed (all the properties of all the objects); it's not, in our experience, particularly useful.

www.kimballgroup.com Number 64, February 8, 2005

Design Tip #64 Avoid DW/BI Isolation

By Margy Ross

Perhaps it's just coincidental, but several people have asked a similar question recently. "Should the DW or BI team gather requirements from the business?" Honestly, this question makes the hair start to stand up on the back of my neck. I'm concerned that too many organizations have overly compartmentalized their data warehouse and business intelligence teams.

Of course, some of this division is natural; especially when the resources allocated to DW/BI grows as the environment expands, creating obvious span of control issues. Also, separation of labor allows for specialization. Viewing the overall DW/BI environment as analogous to a commercial restaurant, some team members are highly-skilled in kitchen food preparation while others are extremely attentive to the needs of the restaurant patrons, ensuring their return for a subsequent visit. There are likely few waiters that should suddenly don the chef's garb, and vice versa.

Despite the distribution of responsibilities, the kitchen and front rooms of a restaurant are tightly entwined. Neither can be successful on its own. The best chefs need a well-trained, well-oiled front room machine; the most attractive dining room requires depth and quality from the kitchen. Only the complete package can deliver consistent, pleasurable dining experiences (and sustainability as a restaurant). That's why the chef and wait staff often huddle to educate and compare notes before a meal rush.

In the world of DW/BI, we've observed some teams take a more isolationist approach. Matters are further complicated by the complexities and realities of organizational culture and politics. There may be a kitchen and dining area, but there's no swinging door between the two. It's like there's a transom (above eye level) where orders and plates are flung back and forth, but the two teams of specialists aren't communicating or cooperating. In this scenario, you end up with data models that can't reasonably be populated. Or data models that don't address the diners' needs and/or leverage their tools. Or diners' tools that are overtaxed or slow-performing because they're repeatedly doing the work that could have been done once in the kitchen and shared throughout the organization. In the worse case, the wall becomes so impenetrable that the BI dining room substitutes a different kitchen (or creates their own) to source meals.

The data warehouse should be the foundation for effective business intelligence. Too many people have focused on one without the other. Sure, you can create a data warehouse without concern for business intelligence, and vice versa, but neither situation is sustainable for long. Isolationism is not a healthy approach for building and supporting the DW/BI environment. Even if you don't report into the same management structure, collaboration and communication are critical.

www.kimballgroup.com Number 63, January 11, 2005

Design Tip #63 Building a Change Data Capture System

By Ralph Kimball

The ETL data flow begins with transferring the latest source data into the data warehouse. In almost every data warehouse, we must transfer only the relevant changes to the source data since the last transfer. Completely refreshing our target fact and dimension tables is usually undesirable.

Isolating the latest source data is called “change data capture” and is often abbreviated CDC in high level architecture diagrams. The idea behind change data capture seems simple enough: just transfer the data that has been changed since the last load. But building a good change data capture system is not as easy as it looks.

Here are the goals I have for capturing changed data:

- Isolate the changed source data to allow selective processing rather than complete refresh
- Capture all changes (deletions, edits and insertions) made to the source data including changes made through non-standard interfaces
- Tag changed data with reason codes to distinguish error corrections from true updates
- Support compliance tracking with additional metadata
- Perform the change data capture step as early as possible, preferably before bulk data transfer to data warehouse

The first step in change data capture is detecting the changes! There are four main ways to detect changes:

1) **Audit columns**. In most cases, the source system contains audit columns. Audit columns are appended to the end of each table to store the date and time a record was added or modified. Audit columns are usually populated via database triggers that are fired off automatically as records are inserted or updated. Sometimes, for performance reasons, the columns are populated by the front end application instead of database triggers. When these fields are loaded by any means other than database triggers, you must pay special attention to their integrity. You must analyze and test each of the columns to ensure that is a reliable source to indicate changed data. If you find any NULL values, you must to find an alternative approach for detecting change. The most common environment situation that prevents the ETL process from using audit columns is when the fields are populated by the front-end application and the DBA team allows “back-end” scripts to modify data. If this is the situation in your environment, you face a high risk that you will eventually miss changed data during your incremental loads.

2) **Database log scraping**. Log scraping effectively takes a snapshot of the database redo log at a scheduled point in time (usually midnight) and scours it for transactions that affect the tables you care about for your ETL load. Sniffing involves a “polling” of the redo log, capturing transactions on-the-fly. Scraping the log for transactions is probably the messiest of all techniques. It’s not rare for transaction logs to “blow-out,” meaning they get full and prevent new transactions from occurring. When this

happens in a production transaction environment, the knee-jerk reaction for the DBA responsible is to empty the contents of the log so the business operations can resume, but when a log is emptied, all transactions within them are lost. If you've exhausted all other techniques and find log scraping is your last resort for finding new or changed records, persuade the DBA to create a special log to meet your specific needs.

3) **Timed extracts**. With a timed extract you typically select all of the rows where the date in the Create or Modified date fields equal SYSDATE-1, meaning you've got all of yesterday's records. Sounds perfect, right? Wrong. Loading records based purely on time is a common mistake made by most beginning ETL developers. This process is horribly unreliable. Time-based data selection loads duplicate rows when it is restarted from mid-process failures. This means that manual intervention and data cleanup is required if the process fails for any reason. Meanwhile, if the nightly load process fails to run and misses a day, a risk exists that the missed data will never make it into the data warehouse.

4) **Full database "diff compare."** A full diff compare keeps a full snapshot of yesterday's database, and compares it, record by record against today's database to find what changed. The good news is that this technique is fully general: you are guaranteed to find every change. The obvious bad news is that in many cases this technique is very resource intensive. If you must do a full diff compare, then try to do the compare on the source machine so that you don't have to transfer the whole database into the ETL environment. Also, investigate using CRC (cyclic redundancy checksum) algorithms to quickly tell if a complex record has changed.

This design tip offers only a teaspoon sip of the issues surrounding change data capture. To dig deeper, here are two suggestions. Read the new *Data Warehouse ETL Toolkit* book I wrote with Joe Caserta for more detail on each of the above alternatives. Second, come to my class for ETL system designers. For change data capture, the students and I will build an aggregated set of requirements from around the room as if we were all part of one IT shop. Should be fun! The next ETL class is in San Jose in only three weeks. Go to our web site for details.

www.kimballgroup.com Number 62, December 7, 2004

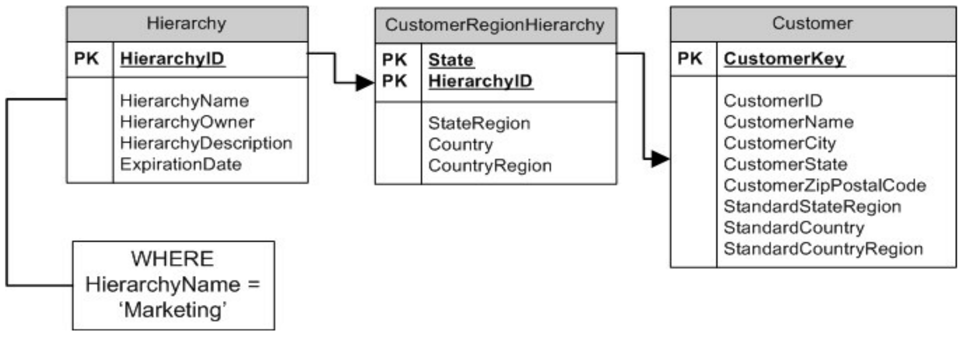
Kimball Design Tip #62: Alternate Hierarchies

By Warren Thornthwaite

Different users often want to see data grouped in different ways. In the simplest case, one department, like Marketing, wants to see customers grouped into one hierarchy and another department, like Sales, wants to see it grouped into an alternate hierarchy. When it really is this simple, it makes sense to include both hierarchies in the Customer dimension table and label them appropriately. Unfortunately, there are only so many alternate hierarchies you can build into a dimension before it becomes unusable.

The need to more flexibly accommodate alternate hierarchies occurs when several departments want to see things their own way, plus they want to see multiple versions of their own way. In this case, we generally work with the users to define the most common way data will be grouped. This becomes the standard or default hierarchy in the base dimension. Any other commonly used hierarchies are also built into the dimension to maintain simplicity for the users.

We then provide an alternate hierarchy table that allows users to roll the data up based on their choice of the available alternate hierarchies. The figure shows an example alternate hierarchy bridge table called CustomerRegionHierarchy for rolling up geographical regions.



Each hierarchy in the alternate hierarchies table must include the entire hierarchy from its starting point where it joins to its associated dimension up to the top. In this case, the CustomerRegionHierarchy table starts at the State level and goes up from there. It is certainly possible to start from a lower level of detail, ZipPostalCode for example, but it would make the bridge table larger and might not add any benefit. On the other hand, if there’s a requirement to create alternative groupings of zip codes within states, the bridge hierarchy table obviously has to start at the zip level.

To simplify reporting and analysis, the bridge table includes the definition of the standard hierarchy. This choice then becomes the default in all structured reports, allowing users to switch between the standard and alternative hierarchies. The creation of a separate Hierarchy table helps simplify maintenance with one row for each hierarchy, but increases the visual complexity. This table could be denormalized back into the bridge table.

The CustomerRegionHierarchy table should be used in structured reports or by expert users. Joining it to the Customer table will cause over counting unless the HierarchyName is constrained to a single hierarchy. All structured reports that provide access to an alternate hierarchies table should be built using the default hierarchy and should require the selection of a single hierarchy.

The alternate hierarchies table is an example of added value in the warehouse. These kinds of custom groupings are commonly used in business, but the definitions are not often centralized or managed in any formal way. Usually they live in a spreadsheet (or dozens of spreadsheets) on desktops.

www.kimballgroup.com Number 61, October 28, 2004

Kimball Design Tip #61: Handling All The Dates

By Bob Becker

It's not unusual to identify dozens of different dates, each with business significance that must be included in a dimensional design. For example, in a financial services organization you might be dealing with deposit date, withdrawal date, funding date, check written date, check processed date, account opened date, card issued date, product introduction date, promotion begin date, customer birth date, row effective date, row load date and statement month.

The first thing to know is not all dates are created equal and handled the same way. Many dates end up as date dimension foreign keys in the fact tables. Most of the remaining dates become attributes of other dimensions. Finally, some dates are included in the design to facilitate ETL processing and/or auditing capabilities.

Assume our financial services company is designing a fact table integrating checking account transactions, such as deposit, ATM and check transactions. Each fact row includes a transaction type dimension to identify the transaction it represents, as well as a transaction date dimension. The business meaning of the date (such as check transaction date, ATM transaction date or deposit transaction date) is defined by the transaction type dimension. In this case, we would not include three separate date keys in the fact table since only one would be valid for a given row.

In other situations, a single transaction represented by one row in the fact table can be defined by multiple dates, such as the transaction event date and transaction posted date. In this case, both dates will be included as uniquely-named dimension foreign keys. We would use role playing to physically build one date dimension with views to present logically unique date dimensions.

A recent client “generalized" their transaction schema to include all transactions across multiple business processes. The schema for this generalized design propagated a number of date dimensions into the transaction fact table having a null or not applicable value. In our financial services case study, this would be akin to generalizing all checking, credit card, savings and mortgage transactions into a single fact table.

All these fact rows represent transactions in a general sense, but they result from different business measurement processes. Remember, we advocate designing your schema by business process, usually a fact table per business process. Checking transactions are very different from funding a mortgage loan. The two processes have unique metrics and dimensionality, resulting in separate fact tables, each with its own uniquely defined and labeled dates associated with it.

Obviously, we also include a date dimension in periodic snapshot schema reflecting the time period for the row, such as snapshot month. These are shrunken subset date dimensions that conform with our core date dimension.

Many business-significant dates will be included as attributes in dimension tables. Account open date would be included in the account dimension. Likewise, customer birth date, product introduction date and promotion begin date belong in their respective dimensions of customer, product and promotion.

When dates are dimension table attributes, we need to consider their reporting and analysis usage.

Is it enough to know the actual date an account was opened or should we also include attributes for account opened year, account opened month, and account opened month/year? These additional attributes improve the business users' ability to ask interesting analytic questions by grouping accounts based on the year and/or month the account was opened.

In order to support more extensive date-related analysis of these dimension attributes, you can incorporate a robust date dimension as an outrigger to the dimension table. In this case, we include the surrogate key for the applicable date in our dimension rather that the date itself, then use a view to declare unique business-appropriate column labels. This technique opens up all of the rich attributes of our core date dimension for analysis. However, remember that extensive use of outrigger dimensions can compromise usability and performance. Also, be careful that all outrigger dates fall within the date range stored in the standard date dimension table.

There are additional dates to help the data warehouse team manage the ETL process and support audit-ability of the data. Dates such as row effective date, row expiration date, row loaded date or row last updated date should be included in each dimension table. While these dates may not need to be user accessible, they can prove invaluable to the data warehouse team.

Kimball Design Tip #60: Big Shifts Happening in BI

By Ralph Kimball

At last week's Business Intelligence Perspectives Conference hosted by *Computerworld* magazine in Palm Springs, two interesting themes were very evident that signaled some big shifts in the business intelligence (BI) world.

1. Compliance is a Free Pass for BI

A number of speakers marveled at how the new regulatory compliance requirements for financial disclosures, especially the Sarbanes Oxley Act, were opening the pocketbooks of companies to upgrade their BI environments. One speaker said, "All you have to do is mention compliance and the funding proposal is approved." But most of the speakers expressed simultaneous concern that no one knows just what the compliance requirements really mean. Not only are the requirements not spelled out in concrete database technical terms, but it appears that the practical impact of the compliance requirements may have to be played out in the courts, with IT departments defending their practices as "commercially diligent" and responsible.

Obviously, most IT departments interested in meeting the compliance requirements will try to err on the conservative side. And, of course, Sarbanes Oxley is not the only game in town. There are probably a dozen overlapping financial reporting statutes with similar requirements, depending on where you do business.

A conservative approach to meeting most compliance requirements would suggest the ability to:
- Prove (backward) lineage of each final measure and KPI appearing in any report
- Prove (forward) impact of any primary or intermediate data element on a final report
- Prove input data has not been changed
- Prove final measures and KPIs are derived from original data under documented transformations
- Document all transforms, present and past
- Maybe: re-run old ETL pipelines
- Maybe: show all end user and administrative accesses of selected data

This list is grist for ten more Design Tips!

2. Sequential Behavior Analysis is BI's Mount Everest

Some of the most interesting and scary case studies at the BI conference were descriptions of trolling huge databases to answer customer behavior questions. Andreas Weigend, Stanford Professor and former Chief Scientist at Amazon, described a study done at Amazon finding the delay (in days) between a customer first clicking on a product and then eventually buying that product. This is immensely difficult. Since most clicks do not result in purchases, you have to wait until a purchase is made and then look backward hours, days, or weeks in the blizzard of records in the clickstream to find the first click by that customer on that product.

The potential volume of data that entities like Amazon want to look through is staggering. Amazon stores every link exposure in their historical data. A link exposure is the presence of a link on a displayed page. It doesn't mean the user clicked the link. Amazon is capturing terabytes of link exposure data each day!

Link exposures are just the beginning of a biblical flood of data, which will really get serious when RFIDs are deployed down to the individual stock item level. Not only is the volume of data horrifying, but the data is often captured in different servers, each representing “doorways” at different locations and times. These challenges raise the question whether the relational model and the SQL language are even appropriate. Yet the ad hoc questions people want to ask of this data demand the same kind of access that relational databases have been so successful in providing against much smaller data sources.

These two big shifts in BI have different personalities. The first (compliance) is like ballast, and the second (behavior) is like a balloon. But in my opinion, both are real and permanent. They will keep us busy.

www.kimballgroup.com Number 59, September 14, 2004

Kimball Design Tip #59: Surprising Value of Data Profiling

By Ralph Kimball

Data profiling is something of a quiet little corner of data warehousing. I suspect that most of us think of data profiling as something you do after most of the ETL system has been built. In this view, data profiling checks for small anomalies in the data that may require cleanup before the real production data is delivered. Finding these anomalies would seem to save the data warehouse team from little surprises after going into production.

During the past year I have dug deeply into the back room ETL processes required to build a data warehouse while working on a new ETL book with Joe Caserta. Perhaps the biggest revelation of the whole project has been discovering how undervalued data profiling is in the average data warehouse project.

What is data profiling?

Data profiling is the systematic up front analysis of the content of a data source, all the way from counting the bytes and checking cardinalities up to the most thoughtful diagnosis of whether the data can meet the high level goals of the data warehouse.

Data profiling practitioners divide this analysis into a series of tests, starting with individual fields and ending with whole suites of tables comprising extended databases. Individual fields are checked to see that their contents agree with their basic data definitions and domain declarations. It is especially valuable to see how many rows have null values, or have contents that violate the domain definition. For example, if the domain definition is “telephone number” then alphanumeric entries clearly represents a problem. The best data profiling tools count, sort, and display the entries that violate data definitions and domain declarations.

Moving beyond single fields, data profiling then describes the relationships discovered between fields in the same table. Fields that implement a key to the data table can be displayed, together with higher level many-to-1 relationships that implement hierarchies. Checking what should be the key of a table is especially helpful because the violations (duplicate instances of the key field) are either serious errors, or reflect a business rule that has not been incorporated into the ETL design.

Relationships between tables are also checked in the data profiling step, including assumed foreign key to primary key relationships and the presence of parents without children.

Finally, data profiling can be custom programmed to check complex business rules unique to a business such as verifying that all the preconditions have been met for granting approval of a major funding initiative.

Hopefully as I’ve been describing the “features” of data profiling, you have thinking that data profiling really belongs at the very beginning of a project, where it could have a big effect on design and timing. In fact, I have come to the conclusion that data profiling should be the mandatory “next step” in every data warehouse project after the business requirements gathering. Here are the deliverables of data profiling that I have come to appreciate during my recent ETL research project:

- A basic “Go – No Go” decision on the project as a whole! Data profiling may reveal that the data on which the project depends simply does not contain the information from which the hoped for decisions can be made. Although this is disappointing, it is an enormously valuable outcome.
- Data quality issues that come from the source system that must be corrected before the project can proceed. Although slightly less dramatic than canceling the whole project, these corrections are a huge external dependency that must be well managed for the data warehouse to succeed.
- Data quality issues that can be corrected in the ETL processing flow after the data has been extracted from the source system. Understanding these issues drives the design of the ETL transformation logic and exception handling mechanisms. These issues also hint at the manual processing time that will be needed to resolve data problems each day.
- Unanticipated business rules, hierarchical structures, and FK-PK key relationships. Understanding the data at a detailed level flushes out issues that will permeate the design of the ETL system.

Finally a big benefit of data profiling that perhaps should be left unstated (at least while justifying the data warehouse to executives) is that data profiling makes the implementation team look like they know what they are doing. By correctly anticipating the difficult data quality issues of a project up front, the team avoids the embarrassing and career-shortening surprises of discovering BIG problems near the end of a project.

Kimball Design Tip #58: The BI Portal (also known as the Data Warehouse Web Site)

By Warren Thornthwaite

The success of a data warehouse/business intelligence system depends on whether or not the organization gets value out of it. Obviously, people have to use the environment for the organization to realize value. Since the BI portal is the primary point of interaction (the only interaction in many cases), the BI team needs to ensure it's a positive experience.

Too often, BI portal home pages focus largely on the history of the data warehouse, the current status of the load process, or who's on the data warehouse team. These are interesting bits of information, but typically not what BI users are looking for. The BI portal is the user interface to the data warehouse. It must be designed with the user community's needs foremost in mind. There are two basic web design concepts that help: density and structure.

Density
The human mind can take in an incredible amount of information. The human eye is able to resolve images at a resolution of about 530 pixels per inch at a distance of 20 inches (R. N. Clark). Compare this with the paltry 72 pixels per inch resolution of the typical computer screen. Our brains rapidly process information looking for the relevant elements. This combination of visual acuity and mental capacity is what kept our ancestors from being removed from the gene pool by various threats; from predators to low hanging branches to a knife in a bar fight. The browser gives us such a low resolution platform that we have to use it as carefully and efficiently as possible. This means we should fill the BI portal pages with as much information as possible. But we can't just load it hundreds of unordered descriptions and links.

Structure
Our brain can handle all this information only if it is accompanied by an organizing structure. Since the primary reason users come to the BI portal is to find information, a great percentage of the home page should be dedicated to categorizing standardized reports and analyses in a way that makes sense to people. Generally we've found the best way to organize the BI portal is around the organization's core business processes. The business process categories allow users to quickly identify the relevant choice. Within each category, there are detailed sub-categories, allowing the user to quickly parse through the home page to find information that is interesting to them.

For example, a web site for a university data warehouse/BI system might have the following report categories (business processes) on its home page:

Admissions	*Employee Tracking*	*Finance*
Alumni Development	*Enrollment*	*Research Grants*

Each of these might link to another page that provides additional descriptions and links to pages with reports on them. We can increase the information density by pulling some of the lower level categories up to the home page:

Admissions	*Employee Tracking*	*Enrollment*
- Application Stats	- Headcount	- Registration
- Offers and Acceptance	- Benefits and Vacation	- Instructors & Classes
- Financial Aid	- Affirmative Action	- Degrees & Majors

Increasing the density in this manner helps define each category and refine the choices before the user has to click. One way to test your BI portal home page is to measure the percentage of the visible page (full screen browser on an average sized monitor) dedicated to providing users with access to information. It should be at least 50%. Some information design folks believe the target should be closer to 90% "substance."

More Structure
Categories help structure the content, but the web site needs a physical structure as well. The web site needs to have a standard look-and-feel, typically based on the organization's overall page layout, so people can navigate the site with ease.

More Content
Although the main point of the BI portal is to provide access to the standardized reports, it must offer much more than just reports. In addition to the categories and reports lists, we need to provide access to a whole range of tools and information, including:

- Search tool that indexes every report, document and page on the BI web site
- Metadata browser
- Online training, tutorials, example reports and help pages
- Help request system and contact information
- Status, notices, surveys, installs, and other administrative info
- Perhaps a support-oriented news/discussion group
- Personalization capabilities that allow users to save reports or report links to their own page

This information all goes in the lower right corner, the least valuable real estate on the screen (at least in English where we read from left to right and top to bottom).

Building an effective BI portal is an incredible amount of work, but it is a key link in the data warehouse value chain. Every word, header, description, function and link included on the portal needs to communicate the underlying DW/BI content. You should do a design review and test the BI portal with users, asking them to find certain reports and other information. Make sure you don't build a weak link.

www.kimballgroup.com Number 57, August 2, 2004

Kimball Design Tip #57: Early Arriving Facts

By Ralph Kimball

Data warehouses are usually built around the ideal normative assumption that measured activity (the fact records) arrive in the data warehouse at the same time as the context of the activity (the dimension records). When we have both the fact records and the correct contemporary dimension records, we have the luxury of bookkeeping the dimension keys first, and then using these up-to-date keys in the accompanying fact records.

Basically three things can happen when we bookkeep the dimension records.

1) If the dimension entity (say Customer) is a new member of the dimension, we assign a fresh new surrogate dimension key.
2) If the dimension entity is a REVISED version of a Customer, we use the Type 2 slowly changing dimension technique of assigning a new surrogate key and storing the revised Customer description as a new dimension record.
3) Finally if the Customer is a familiar, unchanged member of the dimension, we just use the dimension key we already have for that Customer.

For several years, we have been aware of special modifications to these procedures to deal with Late Arriving Facts, namely fact records that come into the warehouse very much delayed. This is a messy situation because we have to search back in history within the data warehouse to decide how to assign the right dimension keys that were in effect when the activity occurred at the right point in the past. See the Intelligent Enterprise article (*Backward in Time*) on this subject at www.intelligententerprise.com/000929/webhouse.jhtml.

If we have Late Arriving Facts, is it possible to have Early Arriving Facts? How can this happen? Are there situations where this is important?

An early arriving fact takes place when the activity measurement arrives at the data warehouse without its full context. In other words, the statuses of the dimensions attached to the activity measurement are ambiguous or unknown for some period of time. If we are living in the conventional batch update cycle of one or more days latency, we can usually just wait for the dimensions to be reported to us. For example, the identification of the new customer may come in a separate feed delayed by several hours. We may just be able to wait until the dependency is resolved.

But if we are in a real time data warehousing situation in which the fact record must be made visible NOW, and we don't know when the dimensional context will arrive, we have some interesting choices. Our real-time bookkeeping needs to be revised, again using Customer as the problem dimension.

1) If the natural Customer key on the incoming fact record can be recognized, then we provisionally attach the surrogate key for the existing most recent version of that Customer to the fact table, but we also hold open the possibility that we will get a revised version of this Customer reported to us at a later time. At that point, we 2) add the revised Customer record to the dimension with a new surrogate key and then go in and destructively modify the fact record's foreign key to the Customer table. 3) Finally, if we believe that the Customer is new, we assign a new Customer surrogate key with a set of dummy attribute values in a new Customer dimension record. We then return to this

dummy dimension record at a later time and make Type 1 (overwrite) changes to its attributes when we get more complete information on the new Customer. At least this step avoids destructively changing any fact table keys.

There is no way to avoid a brief provisional period where the dimensions are “not quite right.” But these bookkeeping steps try to minimize the impact of the unavoidable updates to keys and other fields. If these early arriving records are all housed in a “hot partition” pinned in memory, then aggregate fact table records should not be necessary. Only when the hot partition is conventionally loaded into the static data warehouse tables at the end of the day (and when the dimensions have caught up with the facts) do you need to build the aggregates.

www.kimballgroup.com Number 56, July 12, 2004

Kimball Design Tip #56: Dimensional Modeling for Microsoft Analysis Services

By Joy Mundy

Over the past few years, an increasing number of our students and clients have been asking us about Microsoft SQL Server 2000 Analysis Services (AS). Analysis Services, as a server based OLAP product, fits very well with the dimensional modeling techniques that the Kimball Group has long espoused. But is that fit perfect? Are there things you should be aware of as you're designing a dimensional data model that will be accessed from an AS cube? Of course there are!

The first and strongest connection between an Analysis Services cube and a relational dimensional model is that the cube should be built from a relational dimensional model. The relational dimensional source can be in any relational database, though extracts perform better from some databases than others.

Dimensional models in a relational platform and AS cubes both have dimension and facts; both love surrogate keys; both emphasize the importance of using well thought out hierarchies to assist navigation through the dimensional data. Conformed dimensions in the Kimball vernacular are called shared dimensions in AS. More advanced techniques like junk dimensions, dimension roles and factless fact tables all translate smoothly.

The most important difference between the dimensional world on a relational platform and Analysis Services OLAP is how dimensions behave. The more familiar and comfortable you are with relational-based dimensions, the weirder you'll find AS dimensions. The problem is that they look so similar to each other on first glance, but actually behave quite differently. What we consider a single relational dimension, like customer, typically becomes several dimension-like structures in the cube. Any attribute or hierarchy you want to slice on must have its own dimension in the cube. For example, if you want to compare sales by gender by geographic region, you would need a gender dimension and geography dimension hierarchy in the cube, where these could both be part of the Customer dimension in the relational-based dimensional structures.

In the example just described, you don't have to change your relational dimensional model. In AS, you would define a geography hierarchy in the Customer dimension. You would first create the Gender attribute as a member property in the Customer dimension, and then explicitly convert that member property to a virtual dimension. This conversion is simple to do, requiring only a few mouse clicks, but it seems strange and unnecessary to those of us who are deeply familiar with the way the relational model behaves. Microsoft has recognized this issue and announced that dimensions will behave much more like relational dimensions in the next version of AS.

We'll quickly summarize some other differences and common "gotchas" you may encounter when working with Analysis Services.

The Good:

- Parent-Child dimensions are sometimes easier to manage and query in AS than relational dimensional. In relational-based dimensional, you would navigate a ragged hierarchy by using a bridge table. In Analysis Services, you would simply model the relational dimension with a parent-child structure and create the AS dimension as "parent-child."
- Analysis Services holds all dimension members in memory, so it's best to stay under 5-7 million

total members (across all dimensions) on the 32-bit platform. You can stretch that number, but with reasonably priced 64-bit systems available, it hardly seems worth the trouble.
- You can define multidimensional expressions to calculate anything on any level of the cube. A powerful example of a calculation is defining inventory balances to be semi-additive across time. On the downside, it requires some thought and effort to define complex calculations correctly.
- Incremental cube processing requires the ability to identify the new rows added to the fact table. Tagging the fact rows with an audit dimension or other metadata allows them to be filtered for incremental cube processing.
- Type 2 slowly changing dimensions are processed smoothly by AS.
- Analysis Services dimensions can be built from either a star or snowflake schema. If queries go through AS, it doesn't matter much how the dimension tables are structured on the relational side, though stars are generally easier to maintain. Of course, if queries from other tools will be issued directly against the relational-based dimensional model, stars typically deliver better ease of use and query performance than snowflakes. As in the relational dimensional model, AS dimensions must correspond to the level of granularity. For example, forecast data at the brand level will require a brand-level dimension while the actual data may be at the SKU level.

The Not So Good:
- Type 1 slowly changing dimension dimensions can be a problem if the column being updated is part of a dimension hierarchy. If the updated Type 1 attribute is simply a member property or a virtual dimension, there's no issue. The problem with restating history on a hierarchical attribute is the aggregates. Analysis Services faces exactly the same problem that you do if you were to managing the aggregates by hand in the relational database. There are several approaches which are discussed in the Performance Guide reference listed below.
- Many to many dimensions are ugly in AS 2000. We've seen people use a variety of techniques to make them work, but none of them is particularly appealing.
- There is no way, short of fully reprocessing a cube partition, to update a fact row in an AS cube. If your data volumes are small, you can fully reprocess the entire cube. With large data volumes, you may be able to isolate changeable rows in a partition (perhaps a partition for the most recent 30 days) and fully reprocess only that partition. Partitions require the expensive Enterprise Edition.

Here are some pointers to Microsoft-published Analysis Services content:
Analysis Services Performance Guide
Analysis Services Operations Guide
Creating Large-Scale, Highly Available OLAP Sites
Advantages of 64-bit to SQL Server 2000 Enterprise Edition BI Customers
SQL Server 2000 (64-bit) Analysis Services: Why Migrate and What to Expect

Kimball Design Tip #55: Exploring Text Facts

By Bob Becker

In this Design Tip, we return to a fundamental concept that perplexes numerous dimensional modelers: text facts (also referred to as fact indicators, attributes, details or notes).

Some of you may be rightfully saying that text facts are a dimensional modeling oxymoron. However, we frequently field questions from clients and students about indicator, type or comment fields that seem to belong in the fact table, but the items are not keys, measurements, or degenerate dimensions (see Design Tip #46 at www.kimballgroup.com).

Generally, we recommend not modeling these so-called text facts in the fact table, but rather attempt to find an appropriate home for them in a dimension table. You don't want to clutter the fact table with several mid-sized (20 to 40 byte) descriptors. Alternatively, you shouldn't just store cryptic codes in the fact table (without dimension decodes), even though we are quite certain EVERYONE knows the decodes already.

When confronted with seemingly text facts, the first question to ask is whether they belong in another dimension table? For example, customer type likely takes on a single value per customer and should be treated as a customer dimension attribute.

If they don't fit neatly into an existing core dimension, then they should be treated as either separate dimensions or separate attributes in a junk dimension. It would be straightforward to build small dimension tables that assigned keys to all the payment or transaction types, and then reference those keys in the fact table. If we get too many of these small dimension tables, you should consider creating a junk dimension. We discussed junk dimensions in Design Tip #48 last year. There are several considerations when evaluating whether to maintain separate dimensions or to lump the indicators together in a junk dimension.

- Number of existing dimension foreign keys in the fact table. If you're nearing 20 foreign keys, then you'll probably want to lump them together.
- Number of potential junk "combination" rows, understanding that the theoretical combinations likely greatly exceed the actual encountered combinations. Ideally we'd keep the size of the junk dimension to less than 100,000 rows.
- Business relevance or understanding of the attribute combinations. Do the attributes have so little to do with each other that users are confused by the forced association in a junk dimension?

Finally, what should you do when the supposed "fact" is a verbose, free-form text field that takes on unlimited values, such as a 240-byte comment field? Profiling the field, then parsing and codifying would make it most useful analytically, but that's almost always easier said than done.

It's been our experience that if the field is truly free-form, it is seldom accessed analytically. Usually these comment fields are only valuable to support a detailed investigation into suspicious transactions on an occasional basis. In this event, you'll want to put the text into a separate dimension rather than carrying that extra bulk on every fact record.

KIMBALL GROUP KIMBALL UNIVERSITY

www.kimballgroup.com Number 54, May 10, 2004

Kimball Design Tip #54: Delivering Both Historical and Current Perspectives

By Margy Ross

As with most things in life, change is inevitable with dimension attributes. Most Design Tip readers are familiar with the three basic slowing changing dimension (SCD) techniques:

Type 1: Overwrite the attribute

Type 2: Add another dimension row

Type 3: Add another dimension attribute

If this is news to you, take a look at Ralph's April 22, 2003 Intelligent Enterprise column, "Soul of the Data Warehouse Part 3: Handling Time."

When dimension attributes change, we are often asked to preserve the historically-accurate values, as well as provide the ability to roll-up historical facts based on the current characteristics. Demand for this capability is growing as the business intelligence community matures analytically. Twenty years ago, analysts were satisfied with dimension tables that were refreshed (overwritten) with current attributes at every load. Then the pendulum swung to completely and accurately capture every change using SCD Type 2. Now more people want to have their cake (or steak) and eat it, too.

We discussed a hybrid approach for supporting this requirement several years ago with Design Tip #15: Combining SCD Techniques. In that Tip, we issued Type 2 rows to capture historical attribute changes, with one or more complementary Type 3 "current" attributes on each row. The Type 3 attribute is overwritten (treated as a Type 1) for the current and all previous Type 2 rows, so analysts can query on either the historically-accurate attributes or current assignments. More flexibility is delivered with this hybrid SCD approach, albeit with added complexity.

Physically, this technique could be implemented in a single dimension table with two columns (historical "as was" and current) for each attribute requiring this flexibility. Alternatively, you could handle all the current attributes in an outrigger table joined to the natural key of the dimension (such as the Employee ID), as opposed to the dimension surrogate key. The outrigger contains just one row of current data for each natural key in the dimension table; the attributes are overwritten whenever change occurs. The same natural key likely appears on multiple Type 2 dimension rows with unique surrogate keys. In the spirit of ease-of-use, the core dimension and outrigger of current values may appear as one via a view, however this approach is unacceptable if performance is negatively impacted.

If you have a large dimension table with a hundred attributes requiring historical and current tracking, the above techniques can become onerous. In this situation, you should consider including the dimension natural key as an additional fact table foreign key. You now have two similar, yet very different dimension tables associated with the fact data. First, the dimension surrogate key joins to a typical dimension with historically-accurate Type 2 data. These attributes filter or group facts by the attribute values in effect when the fact data was loaded. Secondly, the dimension natural key (or a static reference key) joins to a dimension table with just the current Type 1 values. The column labels in this table should be prefaced with "current" to reduce the risk of user confusion. These dimension attributes are used to summarize or filter facts based on the current profile, regardless of the values in effect when the fact row was loaded.

In all the situations described in this Design Tip, the query answer set may be quite different depending on which dimension table attributes are constrained or grouped by. Different results are inevitable because different questions are being asked. However, given the vulnerability to error or misinterpretation, this capability is often reserved for the segment of the user community that understands these inherent differences.

Kimball Design Tip #53: Dimension Embellishments

By Bob Becker

When developing dimensional models, we strive to create robust dimension tables decorated with a rich set of descriptive attributes. The more relevant attributes we pack into dimensions, the greater the users' ability to evaluate their business in new and creative ways. This is especially true when building a customer-centric dimension.

We encourage you to embed intellectual capital in dimensional models. Rather than applying business rules to the data at the analytical layer (often using Excel), derivations and groupings required by the business should be captured in the data so they're consistent and easily shared across analysts regardless of their tools. Of course, this necessitates understanding what the business is doing with data above and beyond what's captured in the operational source. However, it's through this understanding and inclusion of derived attributes (and metrics) that the data warehouse adds value.

As we deliver a wide variety of analytic goodies in the customer dimension, we sometimes become victims of our own success. Inevitably, the business wants to track changes for all these interesting attributes. Assuming we have a customer dimension with millions of rows, we need to use mini-dimensions to track customer attribute changes. Our old friend, the type 2 slowly changing dimension technique, isn't effective due to the large number of additional rows required to support all the change.

The mini-dimension technique uses a separate dimension(s) for the attributes that frequently change. We might build a mini-dimension for customer demographic attributes, such as own/rent home, presence of children, and income level. This dimension would contain a row for every unique combination of these attributes observed in the data. The static and less frequently changing attributes are kept in our large base customer dimension. The fact table captures the relationship of the base customer dimension and demographic mini-dimension as the fact rows are loaded.

It is not unusual for organizations dealing with consumer-level data to create a series of related mini-dimensions. A financial services organization might have mini-dimensions for customer scores, delinquency statuses, behavior segmentations, and credit bureau attributes. The appropriate mini-dimensions along with the base customer dimension are tied together via their foreign key relationship in the fact table rows. The mini-dimensions effectively track changes and also provide smaller points of entry into the fact tables. They are particularly useful when analysis does not require consumer-specific detail.

Users often want to analyze customers without analyzing metrics in a fact table, especially when comparing customer counts based on specific attribute criteria. It's often advantageous to include the currently-assigned surrogate keys for the customer mini-dimensions in the base customer dimension to facilitate this analysis without requiring joins to the fact table. A simple database view or materialized view provides a complete picture of the current view of the customer dimension. In this case, be careful not to attempt to track the mini-dimension surrogate keys as type 2 slowly changing dimension attributes. This will put you right back at the beginning with a large customer dimension growing out of control with too frequent type 2 changes.

Another dimension embellishment is to add aggregated performance metrics to the customer dimension, such as total net purchases last year. While we normally consider performance metrics to be best handled as facts in fact tables (and they should certainly be there!), we are populating them in the dimension to support constraining and labeling, not for use in numeric calculations. Business users will appreciate the inclusion of these metrics for analyses. Of course, populating these attributes in our dimension table places additional demands on the data staging system. We must ensure these aggregated attributes are accurate and consistent.

An alternative and/or complementary approach to storing the actual aggregated performance metrics is grouping the aggregated values into range buckets or segments, such as identifying a credit card customer as a balance revolver or transactor. This is likely to be of greater analytic value than the actual aggregated values and has the added benefit of assuring a consistent segment definition across the organization. This approach works particular well in combination with the mini-dimension technique.

Kimball Design Tip #52: Let's Improve Our Operating Procedures

By Joy Mundy

In my career I've been able to review a lot of data warehouses, in various stages of their lifecycles. I've observed that, broadly speaking, we are not very good about operating the data warehouse system with anything like the rigor that the transaction system guys expect of their systems. In all fairness, a data warehouse is not a transaction system, and few companies can justify a 24x7 service level agreement for data warehouse access. But come on guys, do we have to look like Keystone Kops in an emergency? As we all know, Bad Things Happen — especially as a data warehouse is downstream of every other system in your company.

Operating a data warehouse in a professional manner is not much different than any other systems operations: follow standard best practices, plan for disaster, and practice. Here are some basic suggestions, based on my observations from actual deployments.

Negotiate a service level agreement with the business users. The key here is to negotiate, and then to take that SLA seriously. The decision about service level must be made between the executive sponsor and the data warehouse team leader, based on a thoughtful analysis of the costs and benefits of ratcheting up the SLA towards high availability. The basic outlines of the SLA need to be negotiated early in the project, as a requirement for high availability may significantly change the details of your physical architecture.

Use service accounts for all data warehouse operations. You'd think it would go without saying that all production operations should use a specified service account with appropriate permissions. But I've long lost count of how many times I've seen production loads fail because the DBA left the company and her personal account becomes inactive.

Isolate development from testing from production. Again, it should go without saying. Again, apparently it doesn't. I've observed two main barriers against teams' being rigorous about Dev/Test/Prod procedures: cost and complexity.

The hardware and software costs can be significant, because the best practice is to configure a test system identically to its corresponding production system. You may be able to negotiate reduced software licensing costs for the test system, but the hardware vendors are seldom so accommodating. If you have to skimp on hardware, reduce storage first, testing with a subset of historical data. Next I'd reduce the number of processors. As a last resort, I'd reduce the memory on the test machine. I really hate to make these compromises, because processing and query performance might change in a discontinuous fashion. In other words, the test system might behave substantially differently with reduced data, processors, or memory. The development hardware systems are usually normal desktop machines, although their software should be virtually identical to Test and Prod. Coerce your software vendors to provide as many development licenses as you need at near-zero cost. I think all Dev licenses should cost less than $100. (Good luck!)

Everything that you do to the production system should have been designed in Dev and the deployment script tested on Test. Every operation on the backend should go through rigorous scripting and testing, whether deploying a new data mart, adding a column, changing indexes, changing your aggregate design, modifying a database parameter, backing up, or restoring. Centrally managed front room operations like deploying new query and reporting tools, deploying new corporate reports, and changing security plans, should be equally rigorously tested, and scripted if your frontend tools allow it.

When you are scripting operations, parameterize connection information like ServerName into configuration files, if your tools permit. (If they don't permit it, excoriate your vendor until they add such a feature.) Script everything you possibly can, and then check those scripts into source control.

Data warehouse software vendors do not make it easy for you to do the right thing. The front-end tools and OLAP servers are particularly bad about helping – or even permitting – the development of scripts for incremental operations. It is very challenging to coordinate the rollout of a new subject area across RDBMS, ETL system, analysis, and reporting systems. Be very careful, and test, test, test!

Keep on top of service packs, hot fixes, and product upgrades. The appropriate people on the DW team should be responsible for monitoring upgrades, including fixes, for all products used in the data warehouse system. Set a recurring weekly appointment to browse the appropriate websites to see what's available, and evaluate how important each fix or release is to your project. You shouldn't install every fix as soon as it comes out, but you should be educated about them. If you're proactive, you can save your expensive calls to Tech Support for problems that haven't already been solved.

Develop playbooks for all operations. A playbook contains step-by-step instructions for performing an operation such as restoring a database or table, or deploying a new data mart, or adding a new column to a table. You should develop generic playbooks, and then customize that playbook for each operation you plan to perform in production. For example, if you are changing a database parameter, write down, in reasonable detail, the steps to follow. Then test the playbook on the Test system before applying to Prod. This is absolutely vital if you are performing an operation through a tool's GUI rather than via a script.

Operations is not the fun part of data warehousing. But with good planning and practice, you can meet the inevitable snafu with calm and deliberation, rather than hysteria.

www.kimballgroup.com Number 51, February 1, 2004

Kimball Design Tip #51: Latest Thinking On Time Dimension Tables

By Ralph Kimball

Virtually every fact table has one or more time related dimension foreign keys. Measurements are defined at specific points of time and most measurements are repeated over time.

The most common and useful time dimension is the calendar date dimension with the granularity of a single day. This dimension has surprisingly many attributes. Only a few of these attributes (such as month name and year) can be generated directly from an SQL date-time expression. Holidays, work days, fiscal periods, week numbers, last day of month flags, and other navigational attributes must be embedded in the calendar date dimension and all date navigation should be implemented in applications by using the dimensional attributes. The calendar date dimension has some very unusual properties. It is one of the only dimensions that is completely specified at the beginning of the data warehouse project. It also doesn't have a conventional source. The best way to generate the calendar date dimension is to spend an afternoon with a spreadsheet and build it by hand. Ten years worth of days is less than 4000 rows.

Every calendar date dimension needs a Date Type attribute and a Full Date attribute. These two fields comprise the natural key of the dimension table. The Date Type attribute almost always has the value "date" but there must be at least one record that handles the special non-applicable date situation where the recorded date is inapplicable, corrupted, or hasn't happened yet. The foreign key references in the fact table in these cases must point to a non-date date in the calendar date table! You need at least one of these special records in the calendar date table, but you may want to distinguish several of these unusual conditions. For the inapplicable date case, the value of the Date Type is "inapplicable" or "NA". The Full Date attribute is a full relational date stamp, and it takes on the legitimate value of null for the special cases described above. Remember that the foreign key in a fact table can never be null, since by definition that violates referential integrity.

The calendar date primary key ideally should be a meaningless surrogate key but many ETL teams can't resist the urge to make the key a readable quantity such as 20040718 meaning July 18, 2004. However as with all smart keys, the few special records in the calendar date dimension will make the designer play tricks with the smart key. For instance, the smart key for the inapplicable date would have to be some nonsensical value like 99999999, and applications that tried to interpret the date key directly without using the dimension table would always have to test against this value because it is not a valid date.

In some fact tables time is measured below the level of calendar day, down to minute or even second. One cannot build a time dimension with every minute second of every day represented. There are more than 31 million seconds in a year! We want to preserve the powerful calendar date dimension and simultaneously support precise querying down to the minute or second. We may also want to compute very precise time intervals by comparing the exact time of two fact table records. For these reasons we recommend a design with a calendar date dimension foreign key and a full SQL date-time stamp, both in the fact table. The calendar day component of the precise time remains as a foreign key reference to our familiar calendar day dimension. But we also embed a full SQL date-time stamp directly in the fact table for all queries requiring the extra precision. Think of this as special kind of fact, not a dimension. In this interesting case, it is not useful to make a dimension with the minutes or seconds component of the precise time stamp, because the calculation of time intervals across fact table records becomes too messy when trying to deal with separate day and time-of-day dimensions. In previous Toolkit books, we have recommended building such a dimension with the minutes or seconds component of time as an offset from midnight of each day, but we have come to realize that the resulting end user applications became too difficult, especially when trying to compute time spans. Also, unlike the calendar day dimension, there are very few descriptive attributes for the specific minute or second within a day.

If the enterprise has well defined attributes for time slices within a day, such as shift names, or advertising time slots, an additional time-of-day dimension can be added to the design where this dimension is defined as the number of minutes (or even seconds) past midnight. Thus this time-of-day dimension would either have 1440 records if the grain were minutes or 86,400 records if the grain were seconds. The presence of such a time-of-day dimension does not remove the need for the SQL date-time stamp described above.

Kimball Design Tip #50: Factless Fact Tables? Sounds Like Jumbo Shrimp?

By Bob Becker

Factless fact tables appear to be an oxymoron, similar to jumbo shrimp. How can you have a fact table that doesn't have any facts? We've discussed the basics of factless fact tables several times in our books and articles. In this design tip, we use a factless fact table to complement our slowly changing dimension strategies.

As you probably recall, a factless fact table captures the many-to-many relationships between dimensions, but contains no numeric or textual facts. They are often used to record events or coverage information. Common examples of factless fact tables include:

- Identifying product promotion events (to determine promoted products that didn't sell)
- Tracking student attendance or registration events
- Tracking insurance-related accident events
- Identifying building, facility, and equipment schedules for a hospital or university

For more information on factless fact tables, see Ralph's earlier articles at http://www.intelligententerprise.com/db_area/archives/1999/991602/warehouse.shtml and http://www.dbmsmag.com/9609d05.html.

In today's design tip, imagine we are working on a design for a large business-to-consumer company (pick your favorite consumer-oriented industry – airline, insurance, credit card, banking, communications, or web retailer). The company does business with tens of millions of customers. In addition to the typical requirements for transaction schema to track consumer behavior and periodic snapshot schema to trend our consumer relationships over time, our business partners need the ability to see a customer's exact profile (including dozens of attributes) at any point in time.

Long-time readers may remember Ralph discussing a similar situation in Design Tip 13 (http://ralphkimball.com/html/designtips/2000/designtip13.html). He outlined a technique where the dimension itself captures profile change events as a slowly changing dimension Type 2, rather than creating a fact table to capture the profile transactions. However, we are not likely to use the DT #13 technique in the current scenario given the huge data volumes (millions of customer rows) and potentially volatile changes (dozens of attributes).

Let's assume we design a base customer dimension (with minimal SCD Type 2 attributes), along with four "mini" dimensions to track changes to customer credit attributes, customer preferences, market segmentation/propensities, and customer geography. The five foreign keys are included in the transaction-grained fact table, as well as the monthly snapshot. These foreign keys represent the customer's "state" when the fact row is loaded. So far so good, but we still need to support customer profiling at any point in time. We consider using another periodic snapshot fact table, loaded daily for every customer to capture the point-in-time relationship of the customer dimension and associated mini-dimensions. This translates into loading tens of millions of snapshots nightly with several years of history. We quickly do the math and decide to evaluate other alternatives.

About now you're thinking "That's great, but what about the jumbo shrimp?" We can use a factless fact table to capture the relationship between the customer dimension and mini-dimensions over time. We load a fact row in the factless fact table whenever there is a Type 2 change to the base customer dimension or a change in the relationship between the base dimension and the mini-dimensions. The factless fact table contains foreign keys for the base customer dimension and each of the four mini-dimensions when the row is loaded. We then embellish this design with two dates, row effective and row expiration, to locate a customer's profile at any point in time. We might also add a simple dimension to flag the current customer profile, in addition to a change reason dimension to indicate what caused a new row to be loaded into the factless fact table.

www.kimballgroup.com Number 49, September 15, 2003

Kimball Design Tip #49: Off The Bench

By Margy Ross

Intelligent Enterprise published an article entitled "The Bottom-Up Misnomer" in their latest September 17th issue. I wrote this article with Ralph several months ago. While it's been working through the publishing pipeline, an industry newsletter has sparked more bottom-up versus top-down discussion. Everyone seems to feel qualified to explain the Kimball approach. Unfortunately, they sometimes spread misunderstandings and continue to blur the issues. While we are certainly not the expert source for a detailed explanation of the corporate information factory (CIF), we do feel it's our responsibility to clarify our methods rather than watching from the sidelines.

When we wrote The Data Warehouse Lifecycle Toolkit, we referred to our approach as the Business Dimensional Lifecycle. In retrospect, we should have probably just called it the Kimball Approach as suggested by our publisher. We chose the Business Dimensional Lifecycle label instead because it reinforced our core tenets about successful data warehousing based on our collective experiences since the mid-1980s.

1) First and foremost, you need to focus on the business. If you're not enabling better business decision-making, then you shouldn't bother investing resources in data warehouses and business intelligence. Focusing on the business does NOT imply that we encourage the development of isolated data stores to address specific departmental business needs. You must have one eye on the business' requirements, while the other is focused on broader enterprise data integration and consistency issues.

2) The analytic data should be delivered in dimensional models for ease-of-use and query performance. We recommend that the most atomic data be made available dimensionally so that it can be sliced-and-diced "any which way." As soon as you limit the dimensional model to pre-summarized information, you've limited your ability to answer queries that need to drill down into more details.

3) While the data warehouse will constantly evolve, each iteration should be considered a project lifecycle consisting of predictable activities with a finite start and end.

Somewhere along the line, we were tagged as being a "bottom-up" approach. Perhaps this term was associated because of our strong alignment with the business. Unfortunately, the label fails to reflect that we strongly recommend the development of an enterprise data warehouse bus matrix to capture the relationships between the core business processes / events and core descriptive dimensions BEFORE development begins. These linkages ensure that each project iteration fits into the larger puzzle.

Finally, we believe conformed dimensions (which are logically defined in the bus matrix and then physically enforced through the staging process) are absolutely critical to data consistency and integration. They provide consistent labels, business rules/definitions and domains that are re-used as we construct more fact tables to integrate and capture the results from additional business processes / events.

So these are the concepts that we hold near and dear. I know I'm biased, but I frankly don't see that they warrant debate. If you want to learn more, check out the Intelligent Enterprise "Bottom-Up" article at http://www.ralphkimball.com/html/articlesfolder/articlesbydate.html . Better yet, join us for a Kimball University class where you'll learn from and interact directly with the people who developed the techniques.

Kimball Design Tip #48: De-Clutter With Junk (Dimensions)

By Margy Ross

When developing a dimensional model, we often encounter miscellaneous indicators and flags that don't logically belong to the core dimension tables. These unattached attributes are usually too valuable to ignore or exclude. Designers sometimes want to treat them as facts (supposed textual facts) or clutter the design with numerous small dimensional tables. A third, less obvious but preferable, solution is to incorporate a junk dimension as a holding place for these flags and indicators.

A junk dimension is a convenient grouping of flags and indicators. It's helpful, but not absolutely required, if there's a positive correlation among the values. The benefits of a junk dimension include:

- Provide a recognizable, user-intuitive location for related codes, indicators and their descriptors in a dimensional framework.
- Clean up a cluttered design that already has too many dimensions. There might be five or more indicators that could be collapsed into a single 4-byte integer surrogate key in the fact table.
- Provide a smaller, quicker point of entry for queries compared to performance from constraining directly on these attributes in the fact table. If your database supports bit-mapped indices, this potential benefit may be irrelevant, although the others are still valid.

An interesting use for a junk dimension is to capture the context of a specific transaction. While our common, conformed dimensions contain the key dimensional attributes of interest, there are likely attributes about the transaction that are not known until the transaction is processed.

For example, a healthcare insurance provide may need to capture the context surrounding their claims transactions. The grain for this key business process is one row for each line item on a claim. Due to the complexities of the healthcare industry, similar claims may be handled quite differently. They may design separate junk dimensions to capture the context of how the claim was processed, how it was paid, and the contractual relationship between the healthcare providers at the time of the claim.

There are two approaches for creating junk dimensions. The first is to create the junk dimension table in advance. Each possible, unique combination generates a row in the junk dimension table. The second approach is to create the rows in the junk dimension on the fly during the extract, transformation, and load (ETL) process. As new unique combinations are encountered, a new row with its surrogate key is created and loaded into the junk dimension table.

If the total number of possible rows in the junk dimension is relatively small, it is probably best to create the rows in advance. On the other hand, if the total number of possible rows in the junk dimension is large, it may be more advantageous to create the junk dimension as unique rows are encountered. One of the junk dimensions encountered in the recent healthcare design had over 1 trillion theoretical rows, while the actual number of observed rows was tens of thousands. Obviously, it did not make sense to create all the theoretically possible rows in advance. If the number of rows in the junk dimension approaches or exceeds the number of rows in the fact table, the design should be clearly re-evaluated.

Since a junk dimension includes all valid combinations of attributes, it will automatically track any changes in dimension attributes. Therefore slowly changing dimension strategies do not need to be considered for junk dimensions.

For more information, see Ralph's Intelligent Enterprise article at http://www.intelligententerprise.com/000320/webhouse.shtml. Junk dimensions are referred to as mystery dimensions in this article.

Kimball Design Tip #47: Relationship Between Strategic Business Initiatives and Business Processes

By Bill Schmarzo

One of the questions I frequently field in my Analytics Workshop is "what is the relationship between an organization's strategic business initiative (which is where the business is focused) and the business process (which is the foundation upon which I build the data warehouse)?"

Strategic business initiatives are organization-wide plans, championed by executive leadership, to delivery significant, compelling and distinguishable financial or competitive advantage to the organization. A strategic business initiative typically has a measurable financial goal and a 12 to 18 month delivery timeframe. Understanding the organization's strategic business initiatives is the starting point for an analytic applications project as it ensures that the analytics project is delivering something of value - or relevance - to the business community.

Meanwhile, a business process is the lowest level of activity that the business performs, such as taking orders, shipping, invoicing, receiving payments, handling service calls and processing claims. We are particularly interested in the metrics resulting from these business processes as they support a variety of analytics. For example, the business process might be retail sales transactions from the point-of-sale system. From that core business process and resulting data, we could embark on a slew of analytics such as promotion evaluation, market basket analysis, direct marketing effectiveness, price-point analysis, and competitive analysis. Business process data is the foundation upon which we build the data warehouse.

So the business is focused on the strategic business initiatives and the data warehouse / analytics team is focused on business processes. Doesn't that cause a huge disconnect? Actually, no. As part of the business requirements gathering process, the data warehouse / analytics team needs to break down or decompose the strategic business initiative into its supporting business processes.

Imagine a row / column matrix where you have business processes as the row headers (just like in the enterprise data warehouse bus architecture matrix) and strategic business initiatives as the column headers. The intersection points in the matrix mark where the business process data is necessary to support the strategic business initiatives. Instead of adding confusion, the integration of strategic business initiatives and business process provides more clarity as to where to begin the analytics project and why. It maintains the tried and true implementation approach of building your data warehouse one business process at a time, reducing time to delivery and eliminating data redundancy, while delivering the foundation necessary to support those initiatives that the business has deemed important.

www.kimballgroup.com Number 46, June 3, 2003

Kimball Design Tip #46: Another Look At Degenerate Dimensions

By Bob Becker

We are often asked about degenerate dimensions in our modeling workshops. Degenerate dimensions cause confusion since they don't look or feel like normal dimensions. It's helpful to remember that according to Webster, "degenerate" refers to something that's 1) declined from the standard norm, or 2) is mathematically simpler.

A degenerate dimension (DD) acts as a dimension key in the fact table, however does not join to a corresponding dimension table because all its interesting attributes have already been placed in other analytic dimensions. Sometimes people want to refer to degenerate dimensions as textual facts, however they're not facts since the fact table's primary key often consists of the DD combined with one or more additional dimension foreign keys.

Degenerate dimensions commonly occur when the fact table's grain is a single transaction (or transaction line). Transaction control header numbers assigned by the operational business process are typically degenerate dimensions, such as order, ticket, credit card transaction, or check numbers. These degenerate dimensions are natural keys of the "parents" of the line items.

Even though there is no corresponding dimension table of attributes, degenerate dimensions can be quite useful for grouping together related fact tables rows. For example, retail point-of-sale transaction numbers tie all the individual items purchased together into a single market basket. In health care, degenerate dimensions can group the claims items related to a single hospital stay or episode of care.

We sometimes encounter more than one DD in a fact table. For example, an insurance claim line fact table typically includes both claim and policy numbers as degenerate dimensions. A manufacturer could include degenerate dimensions for the quote, order, and bill of lading numbers in the shipments fact table.

Degenerate dimensions also serve as a helpful tie-back to the operational world. This can be especially useful during data staging development to align fact table rows to the operational system for quality assurance and integrity checking.

We typically don't implement a surrogate key for a DD. Usually the values for the degenerate dimension are unique and reasonably sized; they don't warrant the assignment of a surrogate key. However, if the operational identifier is a unwieldy alpha-numeric, a surrogate key might conserve significant space, especially if the fact table has a large number of rows. Likewise, a surrogate key is necessary if the operational ID is not unique over time or facilities. Of course, if you join this surrogate key to a dimension table, then the dimension is no longer degenerate.

During design reviews, we sometimes find a dimension table growing proportionately with the fact table. As rows are inserted into the fact table, new rows are also inserted into a related dimension table, often at the same rate as rows were added to the fact table. This situation should send a red flag waving. Usually when a dimension table is growing at roughly the same rate as the fact table, there is a degenerate dimension lurking that has been missed in the initial design.

www.kimballgroup.com Number 45, March 28, 2003

Kimball Design Tip #45: Techniques For Modeling Intellectual Capital

By Bill Schmarzo

One of the biggest challenges to deploying analytics is that the resulting intellectual capital gets buried inside the tools that are used to build the analytics. Once the intellectual capital gets locked inside one of these tools, it becomes difficult to share that decision making logic with users who might be using different business intelligence, query and reporting tools.

I define intellectual capital as the organizational best practices for a specific well-defined business activity. For example, what is the best way to analyze the introduction of a new product line extension such as the new flavor of tooth paste, in the consumer packaged goods market? This would include not only the data and metrics, but also what constitutes normal performance for each of the supporting metrics including the "baseline". Any performance outside of 2 to 3 standard deviations from the baseline would be considered exceptional. I would then try to understand which of the driving metrics such as the price gaps versus competitive set, sell-through, quantity and quality of retail support, any out-of-stocks or distribution metrics, inventory-on-hand, displays, coupons) were outside of acceptable guidelines.

Successful organizations learn to embed their intellectual capital into their data warehouse design using common dimensional modeling techniques instead of locking it into the tools. And these organizations leverage the 5-stage analytic lifecycle (publishing reports, identifying exceptions, determining causal factors, modeling alternatives and tracking actions) in the business requirements process to tease out their intellectual capital requirements.

In the first stage, publishing reports, the intellectual capital foundation starts with ensuring that you have the "right" data with the "right" grain. This stage is critical for setting the data warehouse foundation with standard metrics, conformed dimensions, common attributes, and the most atomic grain for supporting a common vocabulary across the organization. Dimensional modeling techniques that are useful in the publish reports stage include named hierarchies that support standard "state of the business" reporting as well as real time partitions to support real time reporting needs.

In the exceptions stage, the intellectual capital shows up as the richness and robustness of the dimension tables. For example, let's say that you are trying to identify the specific business entities that are causing performance problems. Dimensional modeling techniques like drill down with atomic grain, and drill across supported by the bus architecture, enable the users to identify those areas of the business that are the sources of exceptional performance.

In the determining causal factors stage, dimensional modeling techniques such as consolidated marts and accumulating snapshots help to understand the causes of exception performance. For example, let's say that we're trying to understand processing bottlenecks for business activities such as claims processing or order tracking. An application could be built using a data analysis tool to retrieve the milestone dates and calculate the time gaps. But instead we'd recommend designing that logic into the data warehouse using the accumulating snapshot technique so that the users can see the processing pipeline in one simple database format, regardless of which tool they are using.

In the evaluating alternatives and tracking actions stages, dimensional modeling techniques such as mini-dimensions play a role in capturing and employing the analytic results. For example, let's say that your analytic process yields customer scores for up-selling, cross-selling, private label purchases, credit card fraud, web site visits, email responses, and customer attrition. We might want to track these scores over time in order to analyze the impact of a marketing program on these customer scores. We can make it easier to track and share this intellectual capital by designing these scores into the data warehouse design using demographic mini-dimensions instead of burying them in unpredictable places in the schema. We can also use architectural techniques such as a hot response cache to rapidly deliver the results in the operational environment, thereby "closing the analytic loop."

There are many practical dimensional modeling techniques that can be used to ensure that the intellectual capital that results from the analytic lifecycle gets designed back into the data warehouse instead of being locked into the tools used to build the analytics. This dramatically improves the organization's ability to capture, share and re-use their intellectual capital.

Kimball Design Tip #44: Don't Be Overly Reliant On Your Data Access Tool's Metadata

By Bob Becker

"Oh, we'll handle that in the tool" is the refrain we sometimes hear from design teams. Instead, whenever possible, we suggest you invest the effort to architect as much flexibility, richness, and descriptive information directly into your dimensional schemas as possible rather than leaning on the capabilities of the tool metadata as a crutch.

Today's business intelligence tools provide robust metadata to support a wide range of capabilities, such as label substitution, predefined calculations, and aggregate navigation. These are useful features for your business community. But, we need to be judicious in the use of features the tools provide. Too often design teams take shortcuts and rely on the data access tool metadata to resolve issues that are better handled in our dimensional models. The end result is business rules that become embedded in the tool metadata rather than in our schemas. We also see design teams utilize tool metadata to provide code lookups and indicator descriptors in a misguided effort to keep their schema smaller and tighter.

The biggest drawback to these shortcuts is the dependence on the end user tool metadata to enforce business rules. If we rely on the tool metadata to implement business rules, every user must access the data via the "supported" tool to guarantee business users are presented with "correct" data. Users that want or need to use another access method are forced to recreate the business rules buried in the tool metadata to be assured of correct results.

As data warehouse developers, we need to protect against situations where business users might see different results depending on the tool they elect to use. Regardless of how they access the data warehouse data, users should get the same high quality, consistent data.

You may be thinking, "Fine, then we'll force all users to access the data warehouse through our supported tool." However, this approach will inevitably fall apart. There are a number of reasons an individual may need to access the data warehouse through some other means, bypassing the supported tool and therefore any business rules enforced via its metadata. These scenarios may not exist in your organization at the time you are developing your schema, but rest assured; one of them will arise during your watch:

- An IT professional (perhaps you) may elect to use SQL directly against the data warehouse data to resolve a complex query or audit data.
- Your organization may develop analytic applications based on custom written SQL-based queries directly against the data warehouse.
- Statistical modeling tools and/or data mining tools may need to directly access the data warehouse data.
- A sophisticated user armed with Microsoft Access (or another non-supported tool) may be granted direct access to the data warehouse.
- There may become a need to supplement the data warehouse with multidimensional "cubes" drawn directly from the data warehouse.
- Your organization may select another end user tool, yet not replace the current tool.

None of this should be construed as an argument against leveraging the capabilities of your data access tool. Rather the key is that when in doubt or confronted with a choice, we prefer the design choice that places a capability as close to the data as possible to assure that the capability is available to as wide an audience as possible.

Kimball Design Tip #43: Dealing With Nulls In The Dimensional Model

By Warren Thornthwaite

Most relational databases support the use of a null value to represent an absence of data. Nulls can confuse both data warehouse developers and users because the database treats nulls differently from blanks or zeros, even though they look like blanks or zeros. This design tip explores the three major areas where we find nulls in our source data and makes recommendations on how to handle each situation.

Nulls as Fact Table Foreign Keys
We encounter this potential situation in the source data for several reasons: either the foreign key value is not known at the time of extract, is (correctly) not applicable to the source measurement, or is incorrectly missing from the source extract. Obviously, referential integrity is violated if we put a null in a fact table column declared as a foreign key to a dimension table, because in a relational database, null is not equal to itself.

In the first case, especially with an accumulating snapshot fact table, we sometimes find columns tracking events which have not yet occurred. For example, in an orders tracking accumulating snapshot, a business might receive an order on the 31st, but not ship until the next month. The fact table's Ship_Date will not be known when the fact row is first inserted. In this case, Ship_Date is a foreign key to the date dimension table, but will not join as users expect if we leave the value as null. That is, any fact reporting from the date table joined on Ship_Date will exclude all orders with a null Ship_Date. Most of our users get nervous when data disappears, so we recommend using a surrogate key, which joins to a special record in the date dimension table with a description like "Data not yet available."

Similarly, there are cases when the foreign key is simply not applicable to the fact measurement, such as when promotion is a fact table foreign key, but not every fact row has a promotion associated with it. Again, we'd include a special record in the dimension table with a value such as "No promotion in effect."

In the case where the foreign key is missing from the source extract when it shouldn't be, you have a few options. You can assign it to another special record in the appropriate dimension with a meaningful description like "Missing key," or assign a specific record such as "Missing key for source code #1234," or write the row out to a suspense file. In all cases, you will need to troubleshoot the offending row.

Nulls as Facts
In this case, the null value has two potential meanings. Either the value did not exist, or our measurement system failed to capture the value. Either way, we generally leave the value as null because most database products will handle nulls properly in aggregate functions including SUM, MAX, MIN, COUNT, and AVG. Substituting a zero instead would improperly skew these aggregated calculations.

Nulls as Dimension Attributes
We generally encounter dimension attribute nulls due to timing or dimension sub-setting. For example, perhaps not all the attributes have been captured yet, so we have some unknown attributes

for a period of time. Likewise, there may be certain attributes that only apply to a subset of the dimension members. In either case, the same recommendation applies. Putting a null in these fields can be confusing to the user, as it will appear as a blank on reports and pull-down menus, and require special query syntax to find. Instead, we recommend substituting an appropriately descriptive string, like "Unknown" or "Not provided."

Note that many data mining tools have different techniques for tracking nulls. You may need to do some additional work beyond the above recommendations if you are creating an observation set for data mining.

Kimball Design Tip #42: Combining Periodic And Accumulating Snapshots

By Ralph Kimball

Normally we think of the accumulating snapshot and the periodic snapshot as two different styles of fact tables that we must choose between when we are building a fact table around a data source. Remember that a periodic snapshot (like the monthly summary of a bank account) is a fact table that records activity during a repeating predictable time period. Periodic snapshot records are generally repeated each reporting period as long as the thing being measured (like the account) is in existence. Periodic snapshots are appropriate for long running processes that extend over many reporting periods.

Accumulating snapshots, on the other hand, are used for short processes that have a definite beginning and end, such as an order being filled. For an order, we would usually make a record for each line on the order, and we would revisit the record making updates as the order progressed through the pipeline. The accumulating snapshot is by definition a snapshot of the most recent state of something and therefore the dimensional foreign keys and the facts are, in general, over-written as time progresses.

The simplest implementation of an accumulating snapshot does not give you intermediate points in the history of, for example, an order.

There are at least three ways to capture this intermediate state:

1. Freeze the accumulating snapshots at regular intervals such as month end. These periodic snapshots should probably be in a separate fact table by themselves to keep applications from getting too complicated. Ironically, this approach comes in the back door to mimic a real-time interpretation of a periodic snapshot (where you create a hot rolling current month), but that's another story. The frozen snapshots of the orders can now reflect the use of Type 2 SCDs for the dimensions (like Customer). As in any periodic snapshot, the good news is that you know you have a record for that order each month the order is active. The bad news is that you only see the snapshots of the order at month ends.

2. Freeze the accumulating snapshot and store it in a second fact table if and only if a change to the order occurs. This gives the complete history of an order. It has the same number of records as option 3, below.

3. Maintain a full transaction grain fact table on the order lines. Add a transaction dimension to this fact table to explain each change. This is "fully general" in that you can see every action that has occurred on an order, but be careful. Some of the transactions are not additive over time. For instance, if a line item on an order is cancelled and two other line items are substituted for the original one, it is a complex calculation to correctly reconstruct the order at an arbitrary point in time after these transactions. That's why option #2 may be the best if you need to see every intermediate state of a complete order.

If you are interested in taking a deeper look at all three kinds of fact tables, read my article, "Fundamental Grains," in the Ralph Kimball Group article archived on our website at www.kimballgroup.com. Look in the Advanced Fact Table Topics section.

www.kimballgroup.com

Number 41, November 6, 2002

Kimball Design Tip #41: Drill Down Into A More Detailed Bus Matrix

By Margy Ross

Many of you are already familiar with the data warehouse bus architecture and matrix given their central role in building architected data marts. Ralph's most recent Intelligent Enterprise article (http://www.intelligententerprise.com/021030/517warehouse1_1.shtml) reinforces the importance of the bus architecture. The corresponding bus matrix identifies the key business processes of an organization, along with their associated dimensions. Business processes (typically corresponding to major source systems) are listed as matrix rows, while dimensions appear as matrix columns. The cells of the matrix are then marked to indicate which dimensions apply to which processes.

In a single document, the data warehouse team has a tool for planning the overall data warehouse, identifying the shared dimensions across the enterprise, coordinating the efforts of separate implementation teams, and communicating the importance of shared dimensions throughout the organization. We firmly believe drafting a bus matrix is one of the key initial tasks to be completed by every data warehouse team after soliciting the business' requirements.

While the matrix provides a high-level overview of the data warehouse presentation layer "puzzle pieces" and their ultimate linkages, it is often helpful to provide more detail as each matrix row is implemented. Multiple fact tables often result from a single business process. Perhaps there's a need to view business results in a combination of transaction, periodic snapshot or accumulating snapshot perspectives. Alternatively, multiple fact tables are often required to represent atomic versus more summarized information or to support richer analysis in a heterogeneous product environment.

We can alter the matrix's "grain" or level of detail so that each row represents a single fact table (or cube) related to a business process. Once we've specified the individual fact table, we can supplement the matrix with columns to indicate the fact table's granularity and corresponding facts (actual, calculated or implied). Rather than merely marking the dimensions that apply to each fact table, we can indicate the dimensions' level of detail (such as brand or category, as appropriate, within the product dimension column).

The resulting embellished matrix provides a roadmap to the families of fact tables in your data warehouse. While many of us are naturally predisposed to dense details, we suggest you begin with the more simplistic, high-level matrix and then drill-down into the details as each business process is implemented. Finally, for those of you with an existing data warehouse, the detailed matrix is often a useful tool to document the "as is" status of a more mature warehouse environment.

www.kimballgroup.com Number 40, October, 2002

Kimball Design Tip #40: Structure Of An Analytic Application

By Bill Schmarzo

A comprehensive analytic application environment needs to support a framework that moves users beyond standard reports. The environment needs to proactively “guide” users through the analysis of a business situation, ultimately helping them make an insightful and thoughtful decision. The goals of this analytic application lifecycle are to:

- Proactively “guide” business users beyond basic reporting
- Identify and understand exceptional performance situations
- Capture decision-making “best practices” for each exceptional performance situation
- Share the resulting “best practices” or intellectual capital across the organization

We break the process into five distinct stages:

1. PUBLISH REPORTS -- provides standard operational and managerial “report cards” on the current state of the business.
2. IDENTIFY EXCEPTIONS -- reveals the exceptional performance situations (both over- as well as under-performance) to focus attention.
3. DETERMINE CAUSAL FACTORS-- seeks to understand the “why” or root causes behind the identified exceptions.
4. MODEL ALTERNATIVES -- provides a backdrop to evaluate different decision alternatives.
5. TRACK ACTIONS -- evaluates the effectiveness of the recommended actions and feeds the decisions back to both the operational systems and data warehouse (against which Stage 1 reporting will be conducted), thereby closing the loop.

Let’s walk through each of these stages in a bit more detail to understand their objectives and impact on our data warehouse architecture.

Stage 1: PUBLISH REPORTS. Standard operational and managerial reports are the starting point for the analytic applications lifecycle. These reports look at the current results versus plan or previous periods in order to provide a report card on the state of the business (e.g., “Market share is up 2 points, but profitability is down 10%”).

Data warehouse requirements in the PUBLISH REPORTS stage focus on improving the presentation layer and includes presentation technologies such as dashboards, portals and scorecards. While many data warehouse implementations successfully deliver reports, they stop at the PUBLISH REPORTS stage and declare success.

Stage 2: IDENTIFY EXCEPTIONS. This stage focuses on the identification of “what’s the matter?” or “where are the problems?” phase of the analysis. The focus of this stage is to identify the exceptions to normal performance as well as the opportunities. Most business managers have asked the data warehouse team to replicate a stack of reports in the data warehouse, when in reality, they really want the parts of the reports marked with highlighters and yellow stickies. The exceptions stage is essential in helping users wade through the deluge of data to focus on the opportunities offering the best business return; those deserving the most attention.

The IDENTIFY EXCEPTIONS stage implies new capabilities such as broadcast servers which distribute alerts to users' devices of choice based upon exception "triggers," and visualization tools to view the data in different more creative ways such as trend lines, geographical maps or clusters.

Stage 3: DETERMINE CAUSAL FACTORS. This stage tries to understand the "why" or the root causes of the identified exceptions. Identifying reliable relationships and interactions between variables that drive exceptional performance is the key.

Successfully supporting users' efforts in this stage will require our data warehouse architecture to include additional software such as statistical tools and/or data mining algorithms (i.e., association, sequencing, classification, and segmentation) to quantify cause-and-effect.

Stage 4: MODEL ALTERNATIVES. In this stage we build on cause-and-effect relationships to develop models for evaluating decision alternatives.

The ability to perform what-if analysis and simulations on a range of potential decisions is considered the final goal when following a typical analytic applications lifecycle. We hope to successfully answer strategic questions such as: What happens to my market share, revenue and units sold if I achieve a greater than 10% price differential versus my top 2 competitors? Or, what are the impacts upon my inventory costs if I can achieve 5% sales forecast accuracy instead of my usual 10%? Our data warehouse architecture will need to accommodate additional technologies in the MODEL ALTERNATIVES stage including statistical tools and data mining algorithms for model evaluation, such as sensitivity analysis, Monte Carlo simulations, and optimizations (goal seeking).

Stage 5: TRACK ACTIONS. This stage tracks the effectiveness of the decisions recommended by the previous stage. Ideally, we can enable a "closed loop" process and feed the recommended actions back to the operational system. The effectiveness of the decisions should be captured and analyzed in order to continuously fine-tune the analysis process, business rules and models.

The TRACK ACTIONS stage places additional demands on our data warehouse architecture. We need to enable effective closed loop capabilities back to the operational systems and the data warehouse. We also recommend enhancing existing dimensional models or building performance management tracking data marts to record and track the results of specific business decisions to determine which decisions worked and which ones didn't. Emerging technologies applicable in this area include broadcast servers which will soon enable users to respond with recommended actions from their device of choice (e.g., e-mail, PDA, pagers, WAP phones) not just deliver alerts.

This Design Tip is only a "teaspoon sip" introducing the structure of an analytic application. A more comprehensive column on this topic will appear in Intelligent Enterprise magazine in a few weeks.

Kimball Design Tip #39: Bus Architecture Foundation For Analytic Applications

By Bill Schmarzo

In our previous analytic application design tip, we explored the industry drivers behind the current analytic application hoopla. One of those drivers is the broad acceptance of dimensional modeling as a mature, business-centric data warehouse discipline. Dimensional models promote analytic exploration by providing a high performance and easy-to-use environment to identify performance exceptions and their causal factors.

There are two additional requirements placed on the data warehouse architecture to effectively support analytic applications.

First, the data warehouse bus architecture and its enabling conformed dimensions must serve as the cornerstone of the data warehouse architecture. Ralph and Margy's recent book, "The Data Warehouse Toolkit 2nd Edition," defines the data warehouse bus architecture as "the architecture for the data warehouse's presentation area based upon conformed dimensions and facts. Without adherence to the bus architecture, a data mart is a standalone stovepipe application." For more information about bus architecture and conformed dimensions, see Ralph's 1999 article from Intelligent Enterprise magazine.

Remember, we advocate implementing dimensional models focusing on business processes (e.g., orders, shipments, inventory, and payments), not on business departments (e.g., sales, marketing, manufacturing or finance). The bus architecture and conformed dimensions are a necessary prerequisite for analytic applications because the most interesting, high-value analytic applications require integrating metrics from multiple business processes. For example, a customer lifetime value analysis requires data from orders (to determine revenue and margin contributions), receivables (to determine collections costs), sales pipeline (to determine cost of sales), shipments (to determine manufacturing costs, particularly in build-to-order businesses), logistics (to determine distribution costs), and service calls (to determine support costs).

The key metrics from each of these business processes would be stored in separate dimensional models. The bus architecture allows us to traverse between the different business process models to pull together an integrated view of the customer. In a query tool, we use multi-pass SQL to perform the traversal. This cross-business process integration would be very challenging without a bus architecture using conformed dimensions.

Secondly, "Consolidated" data marts are often also required to support analytic applications. A consolidated data mart is an integrated superset of single business process-oriented dimensional models. It brings together the necessary data from multiple business processes into a single consolidated mart to support a suite of analytical applications. For example, a customer profitability consolidated mart might bring together data from orders, invoicing, solicitations and customer service in order to support customer attrition, customer promotional effectiveness, and customer cross-sell analytic applications.

Business user ease-of-use and performance are the key drivers behind the use of consolidated marts. Forcing users to become expert at multi-pass SQL, being dependent upon an IT professional, or "joining on the screen" on their local PC to integrate data from multiple business process marts is unrealistic. Unfortunately, many query and reporting tools do not provide adequate "drill across" support to navigate between multiple business process marts. The integrated consolidated mart provides a single analytic view (a single star schema) that is easier for users, provides high performance, and is supported by most query and reporting tools. In a sense, a consolidated data mart is the next evolutionary step beyond separate data marts with conformed dimensions. The advantage of a consolidated data mart is that multi-pass SQL techniques are not required (because all the information has been consolidated into a single fact table), but the disadvantage is the cost and the delays associated with building the consolidated tables from the constituent separate data marts.

The move from today's data warehouse architecture to one that can support tomorrow's analytic applications is a natural evolution. It requires adherence to well-known and documented dimensional modeling concepts such as the data warehouse bus architecture, conformed dimensions, and consolidated data marts. Without the appropriate architecture, accessing and analyzing data from multiple business processes in support of these high-business-value analytic applications becomes foreboding.

Kimball Design Tip #38: Analytic Application? What's That!?

By Bill Schmarzo

There's lots of talk about the new "toast of the town" for the data warehouse community ... analytic applications. Analytic applications promise to deliver new business value to organizations that have already made major investments in their data warehouse infrastructure and skills. But what do we mean when we say "analytic application"? And what are the recent industry developments that have led to all the analytic applications hoopla?

In its simplest form, an analytic application is really nothing more than what most data warehouse practitioners have been trying to do for the past couple of decades with decision support systems. These folks would say that an analytic application is a repeatable, guided decision process to analyze business performance. I'd add two other characteristics - "collaborative" and "delivers immediate business value" - to that definition. Examples of analytic applications include vendor performance assessment, customer profitability analysis, and product pipeline analysis. Each of these analytic applications brings together the elements of business requirements, immediate decision making processes, and data. The application then allows us to understand and positively impact business performance.

So the underlying driver for analytic applications remains the demand to "manage by the numbers." But analytic applications have become more implementable in the last year. There have been a number of industry developments fanning the flames of excitement including:

- The widespread adoption of packaged operational systems for enterprise resource planning (SAP, Oracle, PeopleSoft, Lawson, Great Plains), customer relationship management (Siebel, Clarify, Onyx), and supply chain management (i2, Manugistics). This has lead to the standardization of many operational business processes. These packaged systems provide a ready source of standard, relatively clean, and relatively accessible operational data.
- The broad acceptance of dimensional modeling as a refined, extensible, and user-centric data warehouse discipline.
- The emergence of packaged dimensional models for business processes such as finance, human resources, distribution, manufacturing, sales, service, and marketing. These are actually prepackaged analytic application packages intended for end user groups.
- The packaging of source-to-target mappings for the leading packaged operational systems by vendors like Acta, Cognos, Informatica, PeopleSoft and SAP that reduces the time and effort required to extract and transform the data from packaged operational systems and load the data into the packaged dimensional models.
- The efforts by industry organizations (FASB, SCOR, APICS) to standardize a whole range of business performance metrics. This in turn is facilitating the development of prepackaged reports.
- The homogenizing of the query and reporting tools market, and the ability of these tools to take advantage of dimensional models. Many of the tools offer premium capabilities ONLY if the underlying schemas are dimensional.

- The demands of the extended enterprise (customers and suppliers outside the immediate organization) for access to key corporate metrics as a point of competitive advantage.
- The growing realization by data warehouse teams that we must focus on more than just the extracting and organizing of data to achieve real business results. Managing by the numbers is more than just looking at the numbers.

So hopefully we can see the many industry forces at work to nurture the acceptance and drive the viability of analytic applications. But to take advantage of these industry developments, it is vital that we understand the interrelationship between the analytic application and its user community profiles and preferences, the decision-making processes, and the data and dimensional model requirements. We have found that the following approach helps us bring all of this together:

1. Start with a clear definition and understanding of the analytic application, including its relationship to the organization's strategic business initiatives and financial drivers.

2. Frame the analytic application with a guided decision-making process that is designed to guide users beyond simple management and operational reporting so that they use the analytic application for immediate decision making.

3. Capture the user community requirements, usage profiles, and preferences including roles, responsibilities and expectations.

4. Identify the data and dimensional modeling requirements (facts, calculations, dimensions, dimensional attributes, grain, and aggregation strategy) necessary to support the structured decision making process.

5. Match the feature lists and capabilities of various vendors' packaged analytic applications to see which of them fit an 80-20 rule both for the ETL and the final reporting parts of the application. Ideally 80% of your needs are met without custom development, and the remaining 20% of your needs can be added "gracefully" to the vendor's package.

In my next design tip, I'll dig deeper underneath a typical analytic application to show you the architectural impact on your data warehouse environment.

Kimball Design Tip #37: Modeling A Pipeline With An Accumulating Snapshot

By Ralph Kimball

In the Intelligent Enterprise article Fundamental Grains (www.intelligententerprise.com/db_area/archives/1999/993003/warehouse.shtml) I described the three different types of grains that all fact tables seem to fall into. Remember that a fact table is a place where we store measurements emanating from a particular business process. Dimension tables surround and describe these measurements.

Remember also that the grain of a fact table is the definition of exactly what does a fact record represent. It is certainly true that the KEY of a fact table implements the grain, but frequently the clearest declaration of the grain is a business concept rather than a technical list of foreign keys. For instance, the grain of a fact table representing an order taking process may be "line item on the order" whereas the technical definition of the key to that table may turn out to be "invoice number BY product BY promotion".

In all the thousands of fact table designs I have seen and looked at, they all have sorted themselves into three fundamental grains:

1. The TRANSACTION grain, that represents a point in space and time;
2. The PERIODIC SNAPSHOT grain, that represents a regular span of time repeated over and over; and
3. The ACCUMULATING SNAPSHOT grain, that represents the entire life of an entity.

The accumulating snapshot fact table is unusual in a number of ways. Unlike the other grains, the accumulating snapshot usually has a number of Time dimensions, representing when specific steps in the life of the "accumulating entity" take place. For example, an order is

1) created,
2) committed,
3) shipped,
4) delivered,
5) paid for, and maybe
6) returned.

So the design for an orders accumulating snapshot fact table could start off with six time keys, all being foreign keys to views on a single date-valued dimension table. These six views of the date table are called "roles" played by the date table and they are semantically independent as if they were separate physical tables, because we have defined them as separate views.

The other unusual aspect of the accumulating snapshot fact table is that we revisit the same records over and over, physically changing both foreign keys and measured facts, as the (usually short) life of the entity unfolds. The orders process is a classic example.

Now that we have reminded ourselves of the salient design issues for accumulating snapshot fact tables, let's apply this design technique to a pipeline process. We'll use the student admissions pipeline, but those of you interested in sales pipelines should be able to apply this design to your situation easily.

In the case of admissions tracking, prospective students progress through a standard set of admissions hurdles or milestones. We're interested in tracking activities around no less than 15 key steps in the process, including 1) receipt of preliminary admissions test scores, 2) information requested (via web or otherwise), 3) information sent, 4) interview conducted, 5) on-site campus visit, 6) application received, 7) transcript received, 8) test scores received, 9) recommendations received, 10) first pass review by admissions, 11) application reviewed for financial aid, 12) final decision from admissions, 13) student accepted, 14) student admitted and 15) student enrolled. At any point in time, managers in the admissions and enrollment departments are interested in how many applicants are at each stage in the pipeline. It's much like a funnel where many applicants enter the pipeline, but far fewer progress through to the final stage. Managers also want to analyze the applicant pool by a variety of characteristics. In this admissions example, we can be confident that there is a very rich Applicant dimension filled with interesting demographic information.

The grain of the accumulating snapshot is one row per applicant. Because this is an accumulating snapshot, we revise and update each applicant's unique record in the fact table whenever one of the steps is completed.

A key component of the design is a set of 15 numeric "facts", each a 0 or 1 corresponding to whether the applicant has completed one of the 15 steps listed above. Although technically these 15 1/0 facts could be deduced from the 15 date keys, the additive numeric facts make the application elegant and easy to use with almost any query or reporting tool.

As an extra goodie, we add four more numeric additive facts representing "lags" or time gaps between particularly important steps in the process. These include

Information Requested	==>	Sent lag
Application Submitted	==>	Complete lag
Application Submitted	==>	Final Decision lag
Final Decision	==>	Accept or Decline lag

These lag facts are both good diagnostics for picking out stalled applications, but they help the managers tune the process by identifying bottlenecks.

Our final fact table design looks like

Preliminary Test Score Receipt Date Key (FK)
Information Requested Date Key (FK)
Information Sent Date Key (FK)
Interview Conducted Date Key (FK)
On-Site Campus Visit Date Key (FK)
Application Submitted Date Key (FK)
Transcript Received Date Key (FK)
Test Scores Received Date Key (FK)
Recommendations Received Date Key (FK)
Admissions First Pass Review Date Key (FK)
Reviewed for Financial Aid Date Key (FK)
Admissions Final Decision Date Key (FK)
Applicant Decision Received Date Key (FK)
Admitted Date Key (FK)
Enrolled Date Key (FK)
Applicant Key (FK)
Admissions Decision Key (FK)
Preliminary Test Score Receipt Quantity
Information Requested Quantity
Information Sent Quantity
Information Requested-Sent Lag

Interview Conducted Quantity
On-Site Campus Visit Quantity
Application Submitted Quantity
Transcript Received Quantity
Test Scores Received Quantity
Recommendations Received Quantity
Application Complete Quantity
Application Submitted-Complete Lag
Admissions First Pass Review Quantity
Reviewed for Financial Aid Quantity
Admissions Final Decision Quantity
Application Submitted-Final Decision Lag
Accepted Quantity
Decline Quantity
Final Decision-Accepted/Decline Lag
Admitted Quantity
Enrolled Quantity

Interesting design! Imagine how easy it would be to summarize the state of the pipeline at any point in time. Although the records are obviously wide, this is not an especially big table. If you are a big state university with 100,000 applicants per year, you would only have 100,000 records per year. Assume the 17 foreign keys are all 4 byte integers (nice surrogate keys), and the 21 quantities and lags are 2 byte tiny integers. Our fact table records are then 17 x 4 + 21 x 2 = 110 bytes wide. This makes about 11 MB of data per year in this fact table. Check my math. Actually this is a common outcome for accumulating snapshot fact tables. They are the smallest of the three types, by far.

I would like to hear from you about other pipeline processes where you have either already applied this modeling technique or you realize that it would fit your situation well. Write to me in the next week or two and I'll talk about good examples I receive in the next design tip.

www.kimballgroup.com

Number 36, May 5, 2002

Kimball Design Tip #36: To Be Or Not To Be (Centralized)

By Margy Ross and Bob Becker

Contrary to William Shakespeare and some data warehouse industry pundits, that's NOT the question.

In this article, we discuss an issue faced by maturing data mart/warehouse environments. While some organizations are newcomers to the data warehouse party, others have been at this for quite a while. As the market matures, the cause of data warehouse "pain" within the IT organization is bound to evolve. Recently, centralization has been promoted as the latest miracle elixir. Centralization is claimed to turn independent, disparate data marts into "gold" by reducing administrative costs and improving performance. While centralization "may" deliver some operational efficiency, it does not inherently address the larger issues of integration and consistency.

If your data warehouse environment has been developed without an overall strategy, you are probably dealing with multiple, independent islands of data with the following characteristics:

- Multiple, uncoordinated extracts from the same operational source
- Multiple variations of the same information with inconsistent naming conventions and business rules
- Multiple analyses illustrating inconsistent performance results

Some analysts have tarnished the reputation of data marts by attributing this multitude of data warehousing sins to the mart approach. That's a gross generalization that fails to reflect the benefits that many organizations have realized with their architected data marts. The problems we listed above are the result of a non-existent, poorly defined, or inappropriately executed strategy and can exist with any architectural approach including the enterprise data warehouse, hub-and-spoke and distributed/federated marts.

We can all agree that isolated sets of analytical data warrant attention. Clearly, they are inefficient and incapable of delivering on the business promise of data warehousing. Standalone databases may be easier to initially implement, but without a higher-level integration strategy they are dead-ends that continue to perpetuate incompatible views of the organization. Merely moving these renegade data islands onto a centralized box is no silver bullet if you dodge the real issue: data integration and consistency. A centralization approach that fails to deal with these ills is guilty of treating the symptoms rather than the disease. While it may be simpler to just brush integration and consistency under the carpet due to the political or organizational challenges associated with them, these are the tickets to true business benefit from the data warehouse. It's hard work, but the business pay-off is worth it. In the vernacular of dimensional modeling, this means focusing on the data warehouse bus architecture and conformed dimensions/facts.

As we've previously described, the data warehouse bus architecture is a tool to establish the overall data integration strategy for the organization's data warehouse. It provides the framework for integrating your organization's analytic information. The bus architecture is documented and communicated via the Data warehouse bus matrix (as Ralph described in an Intelligent Enterprise article - www.intelligententerprise.com/db_area/archives/1999/990712/webhouse.shtml). The matrix rows represent the core business processes of the organization, while the matrix columns reflect the common, conformed dimensions.

Conformed dimensions are the means for consistently describing the core characteristics of your business. They are the integration points between the disparate business processes of the organization, ensuring semantic consistency between the processes. There may be valid business reasons for not conforming dimensions. For example, if you are a diversified conglomerate that sells unique products to unique customers through unique channels. However, for most organizations, the key to integrating disparate data is organizational commitment to the creation and use of conformed dimensions throughout your data warehouse architecture, regardless of whether data is centralized or distributed physically.

As we warned earlier, centralization without integration may only throw more fuel on the pre-existing problems. Management may be convinced that buying a new box to house the myriad of existing data marts/warehouses will deliver operational efficiency. Depending on the amount of money they're willing to spend on a centralized hardware platform, it may even positively impact performance. However, these IT benefits are insignificant compared to the business potential from integrated data. Centralization without data integration and semantic consistency will distract an organization from focusing on the real crux of the problem. Inconsistent data will continue to flummox the organization's decision-making ability.

We are well aware that moving to a data warehouse bus architecture will require organizational willpower and the allocation of scarce resources. No one said it would be easy. In fact some industry analysts state that it can't be done. However, our clients' experiences prove otherwise.

We've outlined the typical tasks involved in migrating disparate data to a bus architecture with conformed dimensions. Of course, since each organization's pre-existing environment varies, the list would need to be adjusted to reflect your specific scenario.

- Document the existing data marts/warehouses in your organization, noting the inevitable data overlaps.
- Conduct a high-level assessment of the organization's unmet business requirements.
- Gather key stakeholders to develop a preliminary data warehouse bus matrix for your organization.
- Identify a dimension authority or stewardship committee for each dimension to be conformed.
- Design the core conformed dimensions by integrating and/or reconciling the existing, disparate dimension attributes. Realistically, it may be overwhelming to get everyone to agree on every attribute, but don't let that bring this process to a crashing halt. You've got to start walking down the path toward integration.
- Gain organization agreement on the master conformed dimension(s).
- Develop an incremental plan for converting to the new conformed dimension(s).

Formulating the bus architecture and deploying conformed dimensions will result in a comprehensive data warehouse for your organization that is integrated, consistent, legible and well performing. You'll be able to naturally add data marts with confidence that it will integrate with the existing data. Rather than diverting attention to data inconsistencies and reconciliations, your organization's decision-making capabilities will be empowered with consistent, integrated data.

www.kimballgroup.com Number 35, March 27, 2002

Kimball Design Tip #35: Modeling The Spans

By Ralph Kimball

In the past couple of years I have seen an increased demand for applications that need to ask questions about time spans. One person captured it nicely when he said "each record in my fact table is an episode of constant value over a region of time". Time spans can start and stop at arbitrary points in time. In some cases time spans link together to form an unbroken chain, in other cases the time spans are isolated, and in the worst cases the time spans overlap arbitrarily. But each time span is represented in the database by a single record. To make these variations easier to visualize, let's imagine that we have a database filled with atomic transactions, such as deposits and withdrawals from bank accounts. We'll also include open account and close account transactions. Each transaction implicitly defines an episode of constant value over a region in time. A deposit or a withdrawal defines a new value for the account balance that is valid until the next transaction. This time span could be one second or it could be many months. The open account transaction defines the status of the account as continuously active over a time span until a close account transaction appears.

Before we propose a database design, let's remind ourselves of some of the time span questions we want to ask. We'll start off by limiting our questions to a granularity of individual days, rather than parts of days like minutes and seconds. We'll return to minutes and seconds at the end. The easy questions we have always been good at answering include:

- Show all the transactions that occurred within a given time span.
- Determine if a selected transaction occurred within a given time span.
- Define time spans using complex calendar navigation capabilities including seasons, fiscal periods, day numbers, week numbers, pay days, and holidays.

For these cases all we need is a single time stamp on the transaction fact table record. The first question picks up all the transactions whose time stamp is in an interval specified in the user's query. The second question retrieves the time stamp from a selected transaction and compares it to the interval. The third set of questions replaces the simple time stamp with a calendar date dimension filled with lots of helpful calendar attributes. This date dimension is connected to the fact table through a standard foreign key - primary key join. This is all vanilla dimensional design and only requires a single time key in the fact table record to represent the required time stamp. So far, so good.

By the way, when using complex calendar navigation the queries become much easier if the verbose date dimension includes first day and last day markers for each defined span of time, such as "last day of quarter". This field would have the value "N" for all the days except the last day of the applicable quarter. The last day would have the value "Y" in the special field. These markers allow the complex business time spans to be easily specified in the queries. Note that the use of a verbose date dimension means the application is not navigating a time stamp on the fact table. More about this in the last section.

A second moderately hard category of time span questions include

- Show everyone who was a customer at some point within a time span.

- Show the last transaction for a given customer within a time span.
- Show the balance of an account at an arbitrarily selected point in time.

We'll continue to make the simplifying assumption that all the time spans are described by calendar days, not by minutes and seconds. It is possible to answer all these questions with the single time stamp design given above but this approach requires complex and inefficient queries. For instance, to answer the last question, we would need to search the set of account transactions for the latest transaction at or prior to the desired point in time. In SQL, this would take the form of a correlated sub-SELECT embedded in the surrounding query. Not only is this probably slow, but the SQL is not readily produced by end user tools.

For all of these moderately hard time span questions, we simplify the applications enormously by providing twin time stamps on each fact record, indicating the beginning and end of the time span implicitly defined by the transaction.

With the twin time stamp design, we knock off the above three example questions easily:

1. Search for all the open account transactions whose begin date occurs on or prior to the end of the time span, and whose end date occurs on or after the beginning of the time span. 2. Search for the single transaction whose begin date is on or before the end of the time span and whose end date is on or after the end of the time span. 3. Search for the single transaction whose begin date is on or before the arbitrary point in time and whose end date is on or after the arbitrary point in time.

In all these cases the SQL uses a simple BETWEEN construct. The "value between two fields" style of this SQL is indeed allowable syntax. I learned this recently.

When we use the twin time stamp approach, we have to be honest about one major drawback. In almost all situations, we have to visit each fact table record twice. Once when we first insert it (with an open ended end timestamp), and once more when a superceding transaction occurs that actually defines the real end timestamp. The open ended end time stamp probably should be a real value somewhere out in the future, so that applications don't trip over null values when they try to execute the BETWEEN clause.

We have saved the hardest questions to the end: time spans to the second. In this case we ask the same basic questions as in the first two sections, but we allow the time span boundaries to be defined to the nearest second. In this case, we'll put the same two twin time stamps in the fact record, but we have to give up our connection to a robust time dimension. Both our beginning and ending time stamps must be conventional RDBMS date/time stamps. We must do this because we usually cannot create a single time dimension with all the minutes or all the seconds over a significant period of time. Dividing the time stamp into a day component and a seconds-in-a-day component would make the BETWEEN logic horrendous. So for these ultra-precise time spans we live with the limitations of SQL date/time semantics and give up the ability to specify seasons or fiscal periods to the nearest second.

If you are really a diehard, you could consider four time stamps on each transaction record if your time spans are accurate to the second. The first two would be RDBMS date/time stamps as described in the preceding paragraph. But the third and fourth would be calendar day (only) foreign keys connecting to a verbose calendar day dimension as in the first two sections of this article. That way you can have your cake and eat it too. You can search for ultra-precise time spans, but you can also ask questions like "Show me all the power outages that occurred on a holiday".

Even with these powerful techniques, I am sure there are some tricky time span questions I haven't considered. I collect these tricky puzzles. E-mail new ones to me at ralph@kimballgroup.com.

Kimball Design Tip #34: You Don't Need An EDW

By Ralph Kimball

"EDW" stands for enterprise data warehouse, but more to the point, it is the name for a design approach. The EDW approach differs materially from the Data Warehouse Bus Architecture approach. The EDW embodies a number of related themes that need to be contrasted individually from the DW Bus approach. It may be helpful to separate logical issues from physical issues for a moment.

Logically, both approaches advocate a consistent set of definitions that rationalize the different data sources scattered around the organization. In the case of the DW Bus, the consistent set of definitions takes the specific form of conformed dimensions and conformed facts. With the EDW approach, the consistency seems much more amorphous. You must take it on faith that if you have a single highly normalized E/R model of all the enterprise's information, you then know how to administer hundreds or thousands of table consistently. But, overlooking this lack of precision, one might argue that the two approaches are in agreement up to this point. Both approaches strive to apply a unifying coherence to all the distributed data sources.

A side issue of the EDW enterprise data model is that frequently these models are idealized models of information rather than real models of data sources. Although the exercise of creating the idealized information model is useful, I have seen a number of these big diagrams that never get populated. I have also written a number of articles trying to shed a harsh light on the related claim that the big normalized models encapsulate an organization's "business rules". At best, the normalized models enforce SOME of the DATA rules (mostly many-to-1 relationships), and almost NONE of what a business procedures expert would call business rules. The explanatory labeling of the join paths on an E/R diagram rarely if ever is carried into the code of the backroom ETL processes or the front room query and report writing tools.

Even if we have a tenuous agreement that both approaches have the same goal of creating a consistent representation of an organization's data, as soon as you move into physical design and deployment issues, the differences between the EDW and the DW Bus become really glaring.

As most of you know, the conformed dimensions and conformed facts take on specific forms in the DW Bus architecture. Conformed dimensions are dimensions that have common fields, and the respective domains of the values in these fields are the same. That guarantees that you can perform separate queries on remote fact tables connected to these dimensions and you will be able to merge the columns into a final result. This is of course, drill across. I have written extensively on the steps required to administer conformed dimensions and conformed facts in a distributed data warehouse environment. I have never seen a comparable set of specific guidelines for the EDW approach. I find that interesting because even in a physically centralized EDW, you have to store the data in physically distinct table spaces, and that necessitates going through the same logic as the replication of conformed dimensions. But I have never seen systematic procedures described by EDW advocates for doing this. Which tables do you synchronously replicate between table spaces and when? The DW Bus procedures describe this in great detail.

The flat 2NF (second normal form) nature of the dimensions in the DW Bus design allows us to administer the natural time variance of a dimension in a predictable way (SCD types 1, 2, and 3).

Again, in the highly normalized EDW world, I have never seen a description of how to do the equivalent of slowly changing dimensions. But it would seem to require copious use of time stamps on all the entities, together with a lot more key administration than the dimensional approach requires. By the way, the surrogate key approach I have described for administering SCDs actually has nothing to do with dimensional modeling. In an EDW, the root table of a snowflaked "dimension" would have to undergo exactly the same key administration (using either a surrogate key or a natural key plus a date) with the same number of repeated records if it tracked the same slowly changing time variance as the DW Bus version.

The flat 2NF nature of dimensions in the DW Bus design allows a systematic approach to defining aggregates, the single most powerful and cost effective way to increase the performance of a large data warehouse. The science of dimensional aggregation techniques is intimately linked to the use of conformed dimensions. The "shrunken" dimensions of an aggregate fact table are perfectly conformed subsets of the base dimensions in the DW Bus architecture. The EDW approach, again, has no systematic and documented approach for handling aggregates in the normalized environment or giving guidance to query tools and report writers for how to use aggregates. This issue interacts with "drilling down" described below.

The EDW architecture is both logically and physically centralized, something like a planned economy. Maybe this is unfair, but I think this approach has the same subtle but fatal problem that a planned economy has. It sounds great up front, and the idealistic arguments are hard to refute before the project starts. But the problem is that a fully centralized approach assumes perfect information "a priori" and perfect decision making afterward. Certainly, with planned economies, that was a major reason for their downfall. The DW Bus architecture encourages a continuously evolving design with specific criteria for "graceful modification" of the data schemas so that existing applications continue to function. The symmetry of the dimensional design approach of the DW Bus allows us to pinpoint exactly where new or modified data can be added to a design to preserve this graceful character.

Most importantly, a key assumption built into most EDW architectures is that the centralized EDW "releases" data marts. These data marts are often described as "built to answer a business question". Almost always this comes from inappropriate and premature aggregation. If the data mart is only aggregated data, then of course there will be a set of business questions that cannot be answered. These questions are often not the ones asking for a single atomic record, but rather questions that ask for a precision slice of large amounts of data. A final, unworkable assumption of the EDW is that if the user wants to ask any of these precise questions, they must leave the aggregated dimensional data mart and descend into the 3NF atomic data located in the back room. EVERYTHING is wrong with this view, in my opinion. All of the leverage we gave developed in the DW Bus is defeated by this hybrid architecture: drilling down through conformed dimensions to atomic data; uniform encoding of slowly changing dimensions; the use of performance enhancing aggregates; and the sanctity of keeping the back room data staging area off limits to query services.

Anyway, as you probably know, the Data Warehouse Lifecycle Toolkit book is devoted to the task of building an "enterprise" data warehouse, but in a distributed fashion based on the DW Bus architecture. If you would rather read free articles on these topics then here are some relevant links to topics raised in this design tip, in reverse chronological order. You might want to read them starting with the oldest!

* The Anti-Architect: www.intelligententerprise.com/020114/502warehouse1_1.shtml
* Managing Helper Tables: www.intelligententerprise.com/010810/412warehouse1_1.shtml
* Joint Effort (administering a distributed DW): www.intelligententerprise.com/010524/webhouse1_1.shtml
* Backward in Time: www.intelligententerprise.com/000929/webhouse.shtml
* Enforcing the (Business) Rules: www.intelligententerprise.com/000818/webhouse.shtml
* There are No Guarantees (in an E/R model): www.intelligententerprise.com/000801/webhouse.shtml
* The Matrix (planning a distributed DW Bus architecture):

www.intelligententerprise.com/db_area/archives/1999/990712/webhouse.shtml
* The Data Webhouse Has No Center: www.iemagazine.com/db_area/archives/1999/991307/warehouse.shtml
* Coping With the Brave New Requirements: www.intelligententerprise.com/db_area/archives/1998/9811/warehouse.shtml
* Brave New Requirements for Data Warehousing: www.intelligententerprise.com/db_area/archives/1998/9810/warehouse.shtml
* Pipelining Your Surrogates:
* www.dbmsmag.com/9806d05.html
* Surrogate Keys:
* www.dbmsmag.com/9805d05.html
* Is Data Staging Relational?
* www.dbmsmag.com/9804d05.html
* Bringing Up Supermarts (this article describes the DW Bus architecture before we adopted the term "Bus"): www.dbmsmag.com/9801d14.html
* Aggregate Navigation With (Almost) No Metadata www.dbmsmag.com/9608d54.html

www.kimballgroup.com Number 33, February 10, 2002

Kimball Design Tip #33: Using CRM Measures As Behavior Tags

By Ralph Kimball

Let's describe a simple classic example of assigning behavior tags to complex patterns of micro transactions arising from our customer facing processes like call centers, web site visits, delivery systems, and payment reconciliation systems. We'll use our standard data warehouse reporting techniques to summarize three customer behavior metrics: recency, frequency, and intensity. Recency is a measure of how recently have we interacted with the customer. Let's take a broad perspective and count any transaction from all the customer facing processes we mentioned above. The actual metric of recency is the number of days elapsed since the last contact with the customer.

Similarly, frequency is a measure of how often we have interacted with the customer, again taking the broad perspective of all the customer facing processes. And finally, intensity is a numeric measure of how productive the interactions have been. The most obvious measure of intensity is the total amount of purchases, but perhaps we think the total number of web pages visited is a good measure of intensity, too.

All the of the RFI (recency, frequency, intensity) measures can be subdivided into separate measures for each customer facing process, but we'll keep this example simple.

Now for every customer, we compute their RFI metrics for a rolling time period, such as the latest month. The result is three numbers. We imagine plotting the RFI results in a three dimensional cube with axes Recency, Frequency, and Intensity.

Now we call in our data mining colleagues and ask them to identify the natural clusters of customers in this cube. We really don't want all the numeric results: what we want are behavioral clusters that are meaningful for our marketing department. After running the cluster identifier data mining step, we find, for example, eight natural clusters of customers. After studying where the centroids of the clusters are located in our RFI cube, we are able to assign behavior descriptions to the eight behavior clusters:

A: High volume repeat customer, good credit, few product returns
B: High volume repeat customer, good credit, but many product returns
C: Recent new customer, no established credit pattern
D: Occasional customer, good credit
E: Occasional customer, poor credit
F: Former good customer, not seen recently
G: Frequent window shopper, mostly unproductive
H: Other

We can view the tags A through H as text facts summarizing a customer's behavior. There aren't a lot of text facts in data warehousing but these behavior tags seem to be a pretty good example. We can imagine developing a time series of behavior tag measurements for a customer over time with a data point each month:

John Doe: C C C D D A A A B B

What do you think of this time series? We successfully converted John Doe from a new customer, to an occasional customer, and then to a very desirable high volume repeat customer. But in recent months we have seen a propensity for John to start returning products. Not only is this recent behavior costly, but we worry that John will become disenchanted with our products and eventually end up in the F behavior category!

This little time series is pretty revealing. How can we structure our data warehouse to pump out these kinds of reports? And how can we pose interesting constraints on customers to see only those who have gone from cluster A to cluster B in the most recent time period?

We can model this time series of textual behavior tags in several different ways. Each approach has identical information content but they differ significantly in ease of use. Let's assume we generate a new behavior tag for each customer each month. Here are three approaches:

1) Fact table record for each customer for each month, with the behavior tag as a textual fact.
2) Slowly changing customer dimension record (Type 2) with the behavior tag as a single attribute (field). A new customer record is created for each customer each month. Same number of new records each month as choice #1.
3) Single customer dimension record with a 24 month time series of behavior tags as 24 attributes.

Choices 1 and 2 both have the problem that each successive behavior tag for a given customer is in a different record. Although simple counts will work well with these first two schemes, comparisons and constraints are difficult. For instance, finding the customers who had crossed from cluster A to cluster B in the last time period would be awkward in a relational database because there is no simple way to perform a "straddle constraint" across two records.

In this example we are influenced very much by the predictable periodicity of the data. Every customer is profiled each month. So, even though the behavior tag is a kind of text fact, design choice number 3 looms as very effective. Placing the time series of behavior tags in each customer record has three big advantages. First, the number of records generated is greatly reduced, since a new behavior tag measurement does not by itself generate a new record. Second, complex straddle constraints are easy because the relevant fields are in the same record. And third, we can easily associate the complex straddle constraints with our complete portfolio of customer facing fact tables by means of a simple join to the customer dimension.

Of course, modeling the time series as a specific set of positional fields in the customer dimension has the disadvantage that once you exhaust the 24 fields, you probably need to alter the customer dimension to add more fields. But, in today's fast changing environment, perhaps that will give you an excuse to add to the design in other ways at the same time! At least this change is "graceful" because the change does not impact any existing applications.

We have succeeded in boiling down terabytes of transactional behavior data into a simple set of tags, with help from our data mining colleagues. We then have packaged the tags into a very compact and useful format that supports our high level ease-of-use and ease-of-application-development objectives. We are now ready to pump out all sorts of interesting behavior analyses for our marketing end users.

Kimball Design Tip #32: Doing The Work At Extract Time

By Ralph Kimball

Our mission as data warehouse designers is to publish our data most effectively. This means placing the data in the format and framework that is easiest for end users and application developers to use.

In the back room of our data warehouse we must counter our natural minimalist tendencies when we are preparing the data for final consumption by the end users and application developers. In many important cases, we should deliberately trade off

* increased back room processing, and
* increased front room storage requirements

in exchange for

* symmetrical, predictable schemas that users understand,
* reduced application complexity, and
* improved performance of queries and reports.

Making these tradeoffs should be an explicit design goal for the data warehouse architect. A good data warehouse architect will resist the urge to dump extra analysis and table manipulation on the end users and the application designers. Beware the vendors who argue in favor of complex schemas! But of course these tradeoffs must be chosen judiciously and not overused. Let's look at half a dozen situations where we do just enough work at extract time to really make a difference. We'll also try to draw some boundaries so we know when not to overdo it and spoil the final result. We'll start with some rather narrow examples and gradually expand our scope.

Modeling Events Across Multiple Time Zones
Virtually all measurements recorded in our data warehouses have one or more time stamps. Sometimes these are simple calendar dates but increasingly, we are recording the exact time to the minute or the second. Also, most of our enterprises span multiple time zones, either in the United States, across Europe, across Asia, or around the world. In these cases we have a natural dilemma. Either we standardize all of our time stamps to a single well identified time zone, or we standardize all of our time stamps to the correct local wall clock time when the local measurement event occurred. If we want to convert from GMT to local time, maybe we just figure out which time zone the measurement was taken in, and we apply a simple offset. Unfortunately, this doesn't work. In fact, it fails spectacularly. The rules for international time zones are horrendously complicated. There are more than 500 separate geographic regions with distinct time zone rules. Moral of the story: don't compute time zones in your application, rather add an extra time stamp foreign key in every place you have an existing time stamp, and put the BOTH the standard and local times in your data. In other words, do the work at extract time, not query time, and give up a little data storage.

Verbose Calendar Dimensions
All fact tables in dimensional data warehouse schemas should eschew native calendar date stamps in favor of integer valued foreign keys that connect to verbose calendar dimensions. This recommendation is not based on a foolish dimensional design consistency, but rather it recognizes that calendars are complicated and applications typically need a lot of navigation help. For example,

native SQL date stamps do not identify the last day of the month. Using a proper calendar date dimension, the last day of the month can be identified with a Boolean flag, making applications simple. Just imagine writing a query against a simple SQL date stamp that would constrain on the last days of each month. Here is an example where adding the machinery for an explicit calendar date dimension simplifies queries and speeds up processing by avoiding complex SQL.

Keeping the Books Across Multiple Currencies
The multi time zone perspective discussed in the first example often occurs with the related issue of modeling transactions in multiple currencies. Again, we have two equal and legitimate perspectives. If the transaction took place in a specific currency (say, Swiss franks) we obviously want to keep that information exactly. But if we have a welter of currencies, we find it hard to roll up results to an international total. Again, we take the similar approach of expanding every currency denominated field in our data warehouse to be two fields: local currency value and standard currency value. In this case, we also need to add a currency dimension to each fact table to unambiguously identify the local currency. The location of the transaction is not a reliable indicator of the local currency type.

Contrasting Units of Measure in a Product Pipeline
Most of us think that product pipeline measurements are pretty simple and don't have the complications that financial services, for example, have. Well, spend some time with manufacturing people, distribution people, and retail people, all talking about the same products in the same pipe. The manufacturing people want to see everything in car load lots or pallets. The distribution people want to see everything in shipment cases. The retail people can only see things in individual "scan units". So what do you put in the various fact tables to keep everyone happy? The WRONG answer is to publish each fact in its local unit-of-measure context and leave it to the applications to find the right conversion factors in the product dimension tables! Yes, this is all theoretically possible, but this architecture places an unreasonable burden on the last step of the process where the end users and the application developers live. Instead, present all the measured facts in a single standard unit of measure and then, in the fact table itself, provide the conversion factors to all the other desirable units of measure. That way, applications querying the pipeline data from any perspective have a consistent way to convert all the numeric values to the specific, idiosyncratic perspective of the end user.

Physical Completion of a Profit and Loss (P&L) Design
A profit and loss fact table is very powerful because it presents all the components of revenue and cost at (hopefully) a low level of granularity. After providing this wonderful level of detail, designers sometimes compromise their design by failing to provide all the intermediate levels of the P&L. For instance, the "bottom line profit" is calculated by subtracting the costs from the net revenue. This bottom line profit should be an explicit field in the data, even if it is equal to the algebraic sum of other fields in the same record. It would be a real shame if the user or application developer got confused at the last step and calculated the bottom line profit incorrectly.

Heterogeneous Products
In financial services like banking and insurance, a characteristic conflict often arises between the need to see all of the account types in a single "household" view of the customer, and to see the detailed attributes and measures of each account type. In a big retail bank, there may be 25 lines of business and more than 200 special measures associated with all the different account types. You simply cannot create a giant fact table and giant account dimension table that can accommodate all the heterogeneous products. The solution is to publish the data twice. First create a single core fact table with only the four or five measures, like balance, that are common to all account types. Then, publish the data a second time, with the fact table and account dimension table separately extended for each of the 25 lines of business. Although this may seem wasteful because the huge fact table is effective published twice, it makes the separate householding and line of business applications simple unto themselves.

Aggregations in General
Aggregations are like indexes: they are specific data structures meant to improve performance.

Aggregations are a significant distraction in the back room. They consume processing resources, add complexity to the ETL suite of applications, and they take up lots of storage. BUT, aggregations remain the single most potent tool in the arsenal of the data warehouse designer to improve performance cost effectively.

Dimensional Modeling In General

By now I hope it is obvious that the most widely used back room tradeoff used for the benefit of the end user is the practice of dimensional modeling. A dimensional model is a second normal form version of a third (or higher) normal form model. Collapsing the snowflakes and other complex structures of the higher normal form models into the characteristic flat dimension tables makes the designs simple, symmetrical and understandable. Furthermore, database vendors have been able to focus their processing algorithms on this well understood case in order to make dimensional models run really fast. Unlike some of the other techniques discussed in this design tip, the dimensional model approach can be applied across almost all horizontal and vertical application areas.

www.kimballgroup.com Number 31, November 19, 2001

Kimball Design Tip #31: Designing A Real Time Partition

By Ralph Kimball

Even though the time gap between the production OLTP systems and the data warehouse has shrunk in most cases to 24 hours, the rapacious needs of our marketing users require the data warehouse to fill this gap with real time data.

Most data warehouse designers are skeptical that the existing ETL (extract-transform-load) jobs can simply be sped up from a 24 hour cycle time to a 15 minute cycle time. Data warehouse designers are responding to this crunch by building a real time partition in front of the conventional static data warehouse.

Requirements for the Real Time Partition
To achieve real time reporting we build a special partition that is physically and administratively separated from the conventional static data warehouse tables. The real time partition is actually a separate table subject to special update rules and special query rules.

The real time partition ideally should meet the following tough set of requirements. It must:

- contain all the activity that has occurred since the last update of the static data warehouse.
- link as seamlessly as possible to the grain and content of the static data warehouse fact tables
- be so lightly indexed that incoming data can be continuously "dribbled in"
- support high performance querying

In the dimensional modeling world there are three main types of fact tables: transaction grain, periodic snapshot grain, and accumulating snapshot grain. Our real time partition has a different structure corresponding to each type.

Transaction Grain Real Time Partition
If the static data warehouse fact table has a transaction grain, then it contains exactly one record for each individual transaction in the source system from the beginning of "recorded history". The real time partition has exactly the same dimensional structure as its underlying static fact table. It only contains the transactions that have occurred since midnight, when we loaded the regular data warehouse tables. The real time partition should be almost completely un-indexed, because we need to maintain a continuously "open window" for loading. We avoid building aggregates on this table because we want a minimalist administrative scenario during the day.

We attach the real time partition to our existing applications by drilling across from the static fact table to the real time partition. Time series aggregations (e.g., all sales for the current month) will need to send identical queries to the two fact tables and add them together.

In a relatively large retail environment experiencing 10 million transactions per day, the static fact table would be pretty big. Assuming that each transaction grain record is 40 bytes wide (7 dimensions plus 3 facts, all packed into 4 byte fields), we accumulate 400 MB of data each day. Over a year this would amount to about 150 GB of raw data. Such a fact table would be heavily indexed and supported by aggregates. But the daily tranch of 400 MB (the real time partition) could be pinned in memory. Forget indexes, except may be a B-Tree index on the primary key to support record inserts! Forget aggregations! Our real time partition can remain biased toward very fast loading performance but at the same time provide speedy query performance.

Periodic Snapshot Real Time Partition
If the static data warehouse fact table has a periodic grain (say, monthly), then the real time partition can be viewed as the current hot rolling month. Suppose we are a big retail bank with 15 million accounts. The static fact table has the grain of account by month. A 36 month time series would result in 540 million fact table records. Again, this table would be extensively indexed and supported by aggregates in order to provide good performance. The real time partition, on the other hand, is just an image of the current developing month, updated and overwritten continuously as the month progresses. Semi-additive balances and fully additive facts are adjusted as frequently as they are reported. In a retail bank, the "core" fact table spanning all account types is likely to be quite narrow, with perhaps 4 dimensions and 4 facts, resulting in a real time partition of 480 MB. The real time partition again can be pinned in memory.

Query applications drilling across from the static fact table to the real time partition have slightly different logic compared to the transaction grain. Although account balances and other measures of intensity can be trended directly across the tables, additive totals accumulated during the current rolling period may need to be scaled upward to the equivalent of a full month to keep the results from looking anomalous.

Finally, on the last day of the month, hopefully the accumulating real time partition can just be loaded onto the static data warehouse as the most current month, and the process can start again with an empty real time partition.

Accumulating Snapshot Real Time Partition
Accumulating snapshots are used for short lived processes like orders and shipments. A record is created for each line item on the order or shipment. In the main fact table, this record is updated repeatedly as activity occurs. We create the record for a line item when the order is first placed, then we update it whenever the item is shipped, delivered to the final destination, paid for, and maybe returned. Accumulating snapshot fact tables have a characteristic set of date foreign keys corresponding to each of these steps.

In this case it is misleading to call the main data warehouse fact table "static" because this is the one fact table type that is deliberately updated, often repeatedly. But let's assume that for query performance reasons, this update occurs only at midnight when the users are off-line. In this case the real time partition will consist of only those line items that have been updated today. At the end of the day, the records in the real time partition will be precisely the new versions of the records that need to be written onto the main fact table either by inserting the records if they are completely new, or overwriting existing records with the same primary keys.

In many order and shipment situations, the number of line items in the real time partition will be significantly smaller than the first two examples. For example, the biggest dog and cat food manufacturer in the United States processes about 60,000 shipment invoices per month. Each invoice may have 20 line items. If an invoice line has a normal lifetime of two months and is updated five times in this interval, then we would see about 7500 line items created or updated on an average working day. Even with the rather wide 80 byte records typical of shipment invoice fact tables, we only have 600KB of data in our real time partition. This will obviously fit in memory. Forget indexes and aggregations on this real time partition.

Queries against an accumulating snapshot with a real time partition need to fetch the appropriate line items from both the main fact table and the partition, and can either drill across the two tables by performing a sort merge (outer join) on the identical row headers, or can perform a union of the rows from the two tables, presenting the static view augmented with occasional supplemental rows in the report representing today's hot activity.

In this column I have made what I hope is a strong case for satisfying the new real time requirement with specially constructed, but nevertheless familiar, extensions of our existing fact tables. If you drop nearly all the indexes and aggregations on these special new tables, and pin them in memory, you should be able to get the combined update and query performance that you need.

Kimball Design Tip #30: Put Your Fact Tables On A Diet

By Ralph Kimball

In the middle of the 1990s, before the internet, it appeared that the data explosion might finally abate. At that time we were learning how to capture every telephone call, every item sold at a cash register, every stock transaction on Wall Street, and every policy transaction in huge insurance companies. It's true that we often didn't store a very long time series of some of these data sources in our data warehouses, but there was a feeling that maybe we had reached a kind of physical limit to the granularity of the data. Maybe we had at last encountered the true "atoms" of data.

Well, that view was obviously wrong. We now know that there is no limit to the amount of data we can collect. Every measurement can be replaced a whole series of more granular sub-measurements. On the web, in the web logs we see every gesture made by a visitor BEFORE they check out and purchase a product. We have now replaced the single product purchase record with a dozen or a hundred behavior tracking records. The worst thing is that our marketing people love these behavior tracking records, and want to do all sorts of analysis on them.

Just wait until GPS data capture systems get embedded in our cars and our credit cards and our telephones. Every human being could eventually generate one or more records every second, 24 hours per day!

Although we cannot stop this avalanche of data, we have to try to control it, or we will spend too much money on disk storage. Many of our current data sizing plans are based on quick estimates. In many cases, these estimates seriously overstate our storage needs. The result may be either a decision to buy far too much storage, or to cancel our plans for analyzing available data.

In a dimensional modeling world, it is easy to see that the culprit is always the fact table. The high frequency, repeated measurements in our businesses are stored in the fact table. The fact table is surrounded by geometrically smaller dimension tables. Even a huge customer dimension table with millions of records will be much smaller than the biggest fact table.

By paying fanatic attention to the design of our fact tables, we can often slim them down significantly. Here are the guidelines:

1) Replace all natural foreign keys with the smallest integer (surrogate) keys possible.
2) As part of #1, replace all date/time stamps with integer surrogate keys.
3) Combine correlated dimensions into single dimensions where possible.
4) Group tiny low cardinality dimensions together, even if uncorrelated.
5) Take all text fields out of the fact table. Make them dimensions. Especially comment fields.
6) Replace all long integer and floating point facts with scaled integers, wherever possible.

As an example suppose we are a large telephone company processing 300 million calls per day. We could easily do a data sizing plan for tracking all these calls over a 3 year period based on the following assumptions:

Date/Time = 8 byte date-time stamp
Calling Party Phone Number = 10 byte string

Called Party Phone Number = 15 byte string (to handle international numbers)
Local Provider Entity = 10 byte string
Long Distance Provider Entity = 10 byte string
Added Value Service Provider Entity = 10 byte string
Dialing Status = 5 byte string (100 possible values)
Termination Status = 5 byte string (100 possible values)
Duration fact = 4 byte intege
Rated Charge fact = 8 byte float

Each call is an 85 byte record in this design. Storing three years of this raw data, with no indexes, would require 27.9 terabytes. Assuming a constant stream of data, we would need 776 GB of new storage per month just for the raw data!

Obviously, there is wasted fat in the above record. Let's really turn the screws on this one and see how well we can do. Using the guidelines from above, we can code the same information as follows:

Date = 2 byte tiny integer
Time of Day = 2 byte tiny integer
Calling Party Phone Number = 4 byte integer surrogate key
Called Party Phone Number = 4 byte integer surrogate key
Local Provider Business Entity = 2 byte tiny integer surrogate key
Long Distance Provider Entity = 2 byte tiny integer surrogate key
Added Value Service Provider = 2 byte tiny integer surrogate key
Status = 2 byte tiny integer surrogate key (combination of Dialing and Termination)
Duration fact = 4 byte integer
Rated Charge fact = 4 byte scaled integer

We have made a few assumptions about the data types supported by your particular database. We have assumed that the 65,536 possible 2 byte tiny integer keys are enough to support each of the dimensions where listed above.

With this design, the raw data space required by our fact table becomes 9.2 terabytes, a saving of 67%! Our monthly data growth for raw data has dropped to 256 GB.

While these numbers are still big, they give us some breathing room. Be a fanatic about designing your fact tables conservatively. Put them on a diet.

Kimball Design Tip #29: Graceful Modifications To Existing Fact and Dimension Tables

By Ralph Kimball

Despite the best plans and the best intentions, the data warehouse designer must often face the problem of adding new data types or altering the relationships among data after the data warehouse is up and running. In an ideal world we would like such changes to be "graceful" so that existing query and reporting applications continue to run without being recoded, and existing user interfaces "wake up" to the new data and allow the data to be added to queries and reports.

Obviously, there are some changes that can never be handled gracefully. If a data source ceases to be available and there is no compatible substitute, then the applications depending in this source will stop working.

But can we describe a class of situations where changes to our data environment can be handled gracefully?

The predictable symmetry of our dimensional models comes to our rescue. Dimensional models are able to absorb some significant changes in the source data and in our modeling assumptions without invalidating existing applications. Let's list as many of these changes as we can, starting with the simplest.

1. NEW DIMENSIONAL ATTRIBUTES. If, for example, we discover new textual descriptors of a product or a customer, we add these attributes to the dimension as new fields. All existing applications will be oblivious to the new attributes and will continue to function. Most user interfaces should notice the new attributes at query time. Conceptually, the list of attributes available for constraining and grouping should be displayed in a query tool or a reporting tool via an underlying query of the form SELECT COLUMN_NAME FROM SYS_TABLES WHERE TABLE_NAME = 'PRODUCT'. This kind of user interface will continuously "adjust" when new dimension attributes are added to the schema. In a slowly changing dimension (SCD) environment, where slightly changed versions of the dimension are being maintained, care must be taken to assign the values of the new attributes correctly to the various versions of the dimension records. If the new attributes are available only after a specific point in time, then "N.A." (not available) or its equivalent must be supplied for old dimension records.

2. NEW TYPES OF MEASURED FACTS. Similarly, if new measured facts become available we can add them to the fact table gracefully. The simplest case is when the new facts are available in the same measurement event and at the same grain as the existing facts. In this case, the fact table is altered to add the new fact fields, and the values are populated into the table. In a ideal world, an ALTER TABLE statement can be issued against the existing fact table to add the new fields. If that is not possible, then a second fact table must be defined with the new fields and the records copied from the first. For truly huge fact tables, large groups of records may have to be moved from one table to the other to keep from storing the giant table in its entirety twice. If the new facts are only available from a point in time forward, then true null values need to be placed in the older fact records. If we have done all of this, old applications will continue to run undisturbed. New applications using the new facts should behave reasonably even if the null values are encountered. The users may have to be trained that the new facts are only available from a specific point in time forward.

A more complex situation arises when new measured facts are not available in the same measurement event as the old facts, or if the new facts occur naturally at a different grain. If the new facts cannot be allocated or assigned to the original grain of the fact table, it is very likely that the new facts belong in their own fact table. It is almost always a mistake to mix grains of measurements or mix disjoint kinds of measurements in the same fact table. If you have this situation, you need to bite the bullet and find a query tool or report writer that is capable of multi-pass SQL so that it can access multiple fact tables in the same user request. Tools like Cognos, Business Objects, and Microstrategy are quite capable of handling multi-pass SQL.

3. NEW DIMENSIONS. A dimension can be added to an existing fact table by adding a new foreign key field and populating it correctly with values of the primary key from the new dimension. For example, a weather dimension can be added to a retail sales fact table if a source describing the weather is available, for instance, at each selling location each day. Note that we are not changing the grain of the fact table. If the weather information is only available from a point in time forward, then the foreign key value for the weather dimension must point to a record in the weather dimension whose description is "weather unavailable".

4. A DIMENSION BECOMING MORE GRANULAR. Sometimes it is desirable to increase the granularity of a dimension. For instance, a retail sales fact table with a store dimension could be modified to replace the store dimension with an individual cash register dimension. If we had 100 stores, each with an average of 10 cash registers, the new cash register dimension would have 1000 records. All of the original store attributes would be included in the cash register dimension because cash registers roll up perfectly in a many to 1 relationship to stores. The store attributes could also be modeled physically as a snowflaked outrigger dimension connected to the cash register dimension. Notice that when we increase the granularity of the store==>cash register dimension, we must increase the granularity of the fact table. The fact table will become 10 times as large, in our example. There is probably no alternative but to drop the fact table and rebuild it. Although this change is a big complex change, it is graceful! All the original applications are unaffected. The store totals all look the same and all queries will return the same results. It may run more slowly because there are ten times as many records in the fact table, but we would probably build a store aggregate anyway! This would be the original fact table, now playing the role of an anonymous aggregate table.

5. ADDITION OF AN EXPLICIT HIERARCHY RELATING TWO DIMENSIONS. We may have a situation where two dimensions turn out to have a hierarchical many to 1 relationship but for various reasons we keep the dimensions separate. Normally we would combine hierarchically related entities into the same dimension but if one or both of the dimensions exists in its own right as an independent "conformed" dimension, we may wish to keep them separate. For instance, in insurance we may have a policy dimension and a customer dimension. If every policy has exactly one customer, then policy rolls up to customer. However we probably would not embed the customer information in the policy dimension because the customer dimension will certainly be a major dimension in its own right in our overall data warehouse bus architecture. The customer dimension will need to connect to many other fact tables, in many cases where there is no policy dimension possible. In spite of this need to keep the dimensions separate, some designers will add a very useful CustomerPolicy key to some of their fact tables where the new CustomerPolicy dimension is exactly the marriage of Customer and Policy. This dimension contains the combination of Customers and Policies and can be reliably queried at all times to explore this relationship, independent of the residency of any fact table. The addition of this new combined dimension key poses the same administrative issues as the addition of a new dimension, described in paragraph #3 above. Since it is simply a new dimension, it passes the gracefulness test.

6. ADDITION OF A COMPLETELY NEW SOURCE OF DATA INVOLVING EXISTING DIMENSIONS AS WELL AS UNEXPECTED NEW DIMENSIONS. Almost always, a new source of data has its own granularity and its own dimensions. All of you dimensional designers know the answer to this one. We sprout a brand new fact table. Since any existing fact tables and dimension tables are untouched, by definition, all the existing applications keep chugging along. Although this case seems almost trivial, the point here is to avoid cramming the new measurements into the existing fact tables. A single fact table always owns a single kind of measurement expressed with a uniform grain.

This design tip has tried to define a taxonomy of unexpected changes to your data environment to give you a way to sort through various responses, and to recognize those situations where a graceful change is possible. Since redoing queries and reports is hugely expensive and probably disruptive to the end users, our goal is to stay on the graceful side of the line.

KIMBALL GROUP KIMBALL UNIVERSITY

www.kimballgroup.com Number 28, September 9, 2001

Kimball Design Tip #28: Avoiding Catastrophic Failure Of The Data Warehouse

By Ralph Kimball

The tragic events of September 11 have made all of us re-examine our assumptions and our priorities. We are forced to question our safety and security in ways that would have seemed unthinkable just weeks ago.

We have been used to thinking that our big, important, visible buildings and computers are intrinsically secure, just because they are big, important, and visible. That myth has been shattered. If anything, these kinds of buildings and computers are the most vulnerable.

The devastating assault on our infrastructure has also come at a time when the data warehouse has evolved to a near production-like status in many of our companies. The data warehouse now drives customer relationship management, and provides near real time status tracking of orders, deliveries, and payments. The data warehouse is often the only place where a view of customer and product profitability can be assembled. The data warehouse has become an indispensable tool for running many of our businesses.

Is it possible to do a better job of protecting our data warehouses? Is there a kind of data warehouse that is intrinsically secure and less vulnerable to catastrophic failure?

I have been thinking about writing on this topic for some time, but suddenly the urgency is crystal clear. Let us list some important threats that can result in a sustained catastrophic failure of a data warehouse, and what kinds of practical responses are possible.

Catastrophic Failures

Destruction of the facility – A terrorist attack can level a building or damage it seriously through fire or flooding. In these extreme cases, everything on site may be lost, including tape vaults, and administrative environments.

Deliberate sabotage by a determined insider – The events of September 11 showed that the tactics of terrorism include the infiltration of our systems by skilled individuals who gain access to the most sensitive points of control. Once in the position of control, the terrorist can destroy the system, logically and physically.

Cyberwarfare – It is not news that hackers can break into systems and wreak havoc. The recent events should remove any remaining naïve assumptions that these incursions are harmless, or “constructive” because they expose security flaws in our systems. There are skilled computer users among our enemies, who are actively attempting today to access unauthorized information, alter information, and disable our systems.

Single point failures (deliberate or not) – A final general category of catastrophic failure comes from undue exposure to single point failures, whether the failures are deliberately caused or not. If the loss of a single piece of hardware, a single communication line, or a single person brings the data warehouse down for an extended period of time, then we have a problem with the architecture.

Countering Catastrophic Failures

Distributed architecture – The single most effective and powerful approach for avoiding catastrophic failure of the data warehouse is a profoundly distributed architecture. The “enterprise data warehouse” must be made up of multiple computers, operating systems, database technologies, analytic applications, communication paths, locations, personnel, and on-line copies of the data. The physical computers must be located in widely separated locations, ideally in different parts of the country or around the world. Spreading out the physical hardware with many independent nodes greatly reduces the vulnerability of the warehouse to sabotage and single point failures. Implementing the data warehouse simultaneously with diverse operating systems (e.g., Linux, Unix, and NT) greatly reduces the vulnerability of the warehouse to worms, social engineering attacks, and skilled hackers exploiting specific vulnerabilities.

Parallel communication paths – Even a distributed data warehouse implementation can be compromised if it depends on too few communication paths. Fortunately, the Internet is a robust communication network that is highly parallelized and continuously adapts itself to its own changing topology. The Internet is locally vulnerable if key switching centers (where high performance web servers attach directly to the Internet backbone) are attacked. Each local data warehouse team should have a plan for connecting to the Internet if the local switching center is compromised. Providing redundant multi-mode access paths such as dedicated lines and satellite links from your building to the Internet further reduces vulnerability.

Extended storage area networks (SANs) – A SAN is typically a cluster of high performance disk drives and backup devices connected together via very high speed fiber channel technology. Rather than being a file server, this cluster of disk drives exposes a block level interface to computers accessing the SAN that make the drives appear to be connected to the backplane of each computer. SANs offer at least three huge benefits to a hardened data warehouse. A single physical SAN can be 10 kilometers in extent. This means that disk drives, archive systems and backup devices can be located in separate buildings on a fairly big campus. Second, backup and copying can be performed disk-to-disk at extraordinary speeds across the SAN. And third, since all the disks on a SAN are a shared resource for attached processors, multiple application systems can be configured to access the data in parallel. This is especially compelling in a true read-only environment.

Daily backups to removable media taken to secure storage – We’ve known about this one for years, but now it’s time to take all of this more seriously. No matter what other protections we put in place, nothing provides the bedrock security that offline and securely stored physical media provide.

Strategically placed packet filtering gateways – We need to isolate the key servers of our data warehouse so that they are not directly accessible from the local area networks used within our buildings. In a typical configuration, an application server composes queries which are passed to a separate database server. If the database server is isolated behind a packet filtering gateway, the database server can be configured to only receive packets from the outside world coming from the trusted application server. This means that all other forms of access are either prohibited, or they must be locally connected to the database server behind the gateway. This means that DBAs with system privileges must have their terminals physically attached to this inner network, so that their administrative actions and passwords typed in the clear cannot be detected by packet sniffers on the regular network in the building.

Role enabled bottleneck authentication and access – Data warehouses can be more easily compromised if there are too many different ways to access them, and if security is not centrally controlled. Note that I didn’t say centrally located, rather I said centrally controlled. An appropriate solution would be an LDAP (Lightweight Directory Access Protocol) server controlling all outside-the-gateway access to the data warehouse. The LDAP server allows all requesting users to be authenticated in a uniform way, regardless of whether they are inside the building or coming in over the Internet from a remote location. Once authenticated, the directory server associates the user with a named role. The application server then makes the decision on a screen by screen basis as to whether the authenticated user is entitled to see the information based on the user's role. As our data warehouses grow to thousands of users and hundreds of distinct roles, the advantages of this bottleneck architecture become significant.

There is much we can do to harden our data warehouses. In the past few years our data warehouses have become too critical to the operations of our organizations to remain as exposed as they have been. We have had the wakeup call.

I have written extensively on the above topics. The design of distributed architectures and the discussions of packet filtering gateways and role enabled security are covered comprehensively in the Data Warehouse Lifecycle Toolkit (Wiley, 1998). The application of SANs to data warehouses is described in my IE article of March 8, 2001 “Adjust Your Thinking for SANs” which can be found in the article archive on my web site at www.kimballgroup.com.

www.kimballgroup.com Number 27, August 1, 2001

Kimball Design Tip #27: Being Offline As Little As Possible

By Ralph Kimball

If you update your data warehouse each day, you have a characteristic scramble when you take yesterday's data offline and bring today's data online. During that scramble, your data warehouse is probably unavailable. If all your end users are in the same time zone, you may not be feeling much pressure, as long as you can run the update between 3 and 5 am. But, more likely, if your end users are dispersed across the country or around the world, you want to be offline absolutely as little as possible, because in your case, the sun never sets on the data warehouse. So, how can you reduce this downtime to the bare minimum?

In this design tip, we'll describe a set of techniques that will work for all of the major relational DBMSs that support partitioning. The exact details of administering partitions will vary quite a bit across the DBMSs but you will know what questions to ask.

A partition in a DBMS is a physical segment of a DBMS table. Although the table has a single name as far as applications are concerned, a partitioned table can be managed as if it is made up of separate physical files. In this design tip, I assume your partitioning allows

* moving a partition, but not the whole table, to a new storage device
* taking a partition, but not the whole table, offline
* dropping and rebuilding any index on the partition, but not the whole table
* adding, deleting, and modifying records within a designated partition
* renaming a partition
* replacing a partition with an alternate copy of that partition

Your DBMS lets you partition a table based on a sort order that you specify. If you add data on a daily basis, you need to partition your fact tables based on the main date key in the fact table. In other design tips I have mentioned that if you are using surrogate (integer) keys then you should make sure that the surrogate keys for your date dimension table are assigned in order of the true underlying dates. That way, when you sort your fact table on the date surrogate key, all the most current records cluster in one partition.

If your fact table is named FACT, you also need an unindexed copy called LOADFACT. Actually in all the following steps we're only talking about the most current partition, not the whole table! Here are the steps to keep you offline as little as possible. We go offline at step 4 and come back online at step 7.

1. Load yesterday's data into the LOADFACT partition. Complete quality assurance pass. 2. When done loading, make a copy of LOADFACT named COPYLOADFACT. 3. Build indexes on LOADFACT.

4. Take FACT offline (actually just the most current partition). 5. Rename most current FACT partition to be SAVEFACT. 6. Rename LOADFACT (partition) to be most current FACT partition. 7. Bring FACT online.

8. Now clean up by renaming COPYLOADFACT to be the new LOADFACT. You can resume

dribbling incoming data into this new LOADFACT. 9. If all is well, you can delete SAVEFACT.

So, we have reduced the offline interval to just two renaming operations in steps 5 and 6. Almost certainly, these renaming operations will be faster if the physical size of the most current partition is as small as possible.

There is no question that this scenario is an idealized goal. Your data warehouse will have additional complexities. The main complexities you will have to think through include

* limitations to the partitioning capability of your DBMS.
* need to load old, stale data into your fact table that would disrupt the "most current" assumption
* handling associated aggregate fact tables

In the next design tip, I'll extend the ideal case to cover some of these messy situations. But for a couple of weeks, we can all pretend that life is simple... I would be especially interested in hearing about how your DBMS handles partitioning and whether you have been able to get this scheme to work. I'll publish your comments. Write to me at ralph@ralphkimball.com before the next design tip.

Kimball Design Tip #26: Adding An Audit Dimension To Track Lineage And Confidence

By Ralph Kimball

Whenever we build a fact table containing measurements of our business, we surround the fact table with "everything we know to be true". In a dimensional model, this everything-we-know is packaged in a set of dimensions. Physically we insert foreign keys, one per dimension, into our fact table, and connect these foreign keys to the corresponding primary keys of each dimension. Inside each dimension (like product or customer) is a verbose set of highly correlated text-like descriptors representing individual members of the dimension (like individual products or customers).

We can extend this everything-we-know approach to our fact table designs by including key pieces of metadata that are known to be true when an individual fact record is created. For instance, when we make a fact table record, we should know such things as

1. what source system supplied the fact data (multiple descriptors if multiple source systems).
2. what version of the extract software created the record.
3. what version of allocation logic (if any) was used to create the record.
4. whether a specific "N.A. encoded" fact field actually is unknown, impossible, corrupted, or not-available-yet.
5. whether a specific fact was altered after the initial load, and if so, why.
6. whether the record contains facts more than 2, 3, or 4 standard deviations from the mean, or equivalently, outside various bounds of confidence derived from some other statistical analysis.

The first three items describe the lineage (provenance) of the fact table record. In other words, where did the data come from? The last three items describe our confidence in the quality of data for that fact table record.

Once we start thinking this way, we can come up with a lengthy list of metadata items describing data lineage and data quality confidence. But for the purpose of this design tip, we'll stop with just these six. A more elaborate list can be found in the discussion of Audit Dimensions in the Data Warehouse Lifecycle Toolkit.

Although these six indicators could be encoded in various ways, I prefer text encoding. Ultimately we are going to constrain and report on these various audit attributes, and we want our user interfaces and our report labels to show as understandable text. So, perhaps the version of the extract software (item #2) might contain the value "Informatica release 6.4, Revenue extract v. 5.5.6". Item #5 might contain values such as "Not altered" or "Altered due to restatement".

The most efficient way to add the lineage and confidence information to a fact table is to create a single Audit foreign key in the fact table. A 4-byte integer key is more than sufficient, since the corresponding Audit dimension could have up to 4 billion records. We won't need that many!

We build the Audit dimension as a simple dimension with seven fields:

Audit Key (primary key, 4 byte integer)

Source System (text)
Extract Software (text)
Allocation Logic (text)
Value Status (text)
Altered Status (text)
Out of Bounds Status (text)

In our backroom ETL (extract-transform-load) process, we track all of these indicators and have them ready at the moment when the fact table record is being assembled in its final state. If all six of the Audit fields already exist in the Audit dimension, we fetch the proper primary key from the Audit dimension and use it in the Audit dimension foreign key slot of the fact table. If we have no existing Audit dimension record applicable to our fact table record, we add one to the maximum value of the Audit dimension primary key, and create a new Audit dimension record. This is just standard surrogate key processing. Then we proceed as in the first case. In this way, we build up the Audit dimension over a period of time.

Notice that if we are loading a large number of records each day, almost all of the records will have the same Audit foreign key, since presumably nearly all of the records will be "normal". We can modify the processing in the previous paragraph to take advantage of this by caching the Audit key of the "normal" record, and skipping the lookup for all normal records.

Now that we have built the Audit dimension, how do we use it?

The beauty of this design is that the lineage and confidence metadata has now become regular data, and can now be queried and analyzed along with the other more familiar dimensions. There are two basic approaches to decorating your queries and reports with Audit dimension indicators.

The "poor man's" approach simply adds the desired Audit attributes directly to the Select list of an SQL query. In other words in a simple sales query like

```
SELECT PRODUCT, SUM(SALES)
```

you augment the query to read

```
SELECT PRODUCT, VALUE_STATUS, SUM(SALES), COUNT(*)
```

Now your report will sprout an extra row whenever an anomalous data condition arises. You will get a count that lets you judge how bad the condition is. Notice that before you did this design, the NA (null) encoded data values just dropped silently out of your report without raising an alarm, because SUM ignores null values.

The "rich man's" approach performs full fledged drill across queries producing separate columns (not separate rows) with more sophisticated indicators of lineage or quality. For instance, our simple sales query in the above example could be embellished to produce report headings across each row like

PRODUCT >> SUM(SALES) >> PERCENT OF DATA FROM OLD SOURCE SYSTEM >> PERCENT OF DATA CORRUPTED

and so on. Front end tools like Cognos, Business Objects, and Microstrategy are capable of these drill across reports, using "multi-pass SQL".

I would be interested in hearing about any of your designs where you have converted metadata into data and made a dimension out of it. Write to me at ralph@kimballgroup.com.

www.kimballgroup.com Number 25, July 1, 2001

Kimball Design Tip #25: Designing Dimensional Models For Parent-Child Applications

By Ralph Kimball

The parent-child data relationship is one of the basic, fundamental structures in the business world. An invoice (the parent) has many line items (the children). Other obvious examples besides invoices include orders, bills of lading, insurance policies, and retail sales tickets. Basically, any business document with an embedded repeating group qualifies as a parent-child application, especially when the embedded line items contain interesting numerical measurements like dollars or physical units.

Parent-child applications are of extreme importance to data warehousing because most of the basic control documents that transfer money and goods (or services) from place to place take the parent-child form.

But a parent-child source of data, like invoices, presents a classic design dilemma. Some of the data is only available at the parent level and some only available at the child level. Do we need two fact tables in our dimensional model or can we do it with just one? And what do we do with the data that is only available at the parent level when we want to drill down to the child level?

Let's imagine a typical product sales invoice. The overall invoice is created by the sales agent for our company and is directed at a specific customer. Each line item on the invoice represents a different product sold to the customer.

The parent level data includes

- Date of overall invoice (dimension)
- Sales agent (dimension)
- Customer (dimension)
- Payment terms (dimension)
- Invoice number (degenerate dimension, see below)
- Total net price from the line items (additive fact)
- Total invoice level discounts representing promotional allowances (additive fact)
- Total freight charges (additive fact)
- Total tax (additive fact)
- Grand total (additive fact)

The child level data includes

- Product (dimension)
- Promotion (dimension)
- Number of units of product (additive fact)
- Unit price of product (see below)
- Extended gross price (units X price) (additive fact)
- Promotional discount for this specific product (see below)
- Extended net price (units X (unit price - promotional discount)) (additive fact)

as well as the "context" from the overall invoice (four dimensions, above).

Given our dimension and fact hints, we would seem to be done. We have two nice fact tables. The parent invoice fact table has 4 dimensions and 5 facts, and the child line item fact table has 6 dimensions and 3 facts.

But our design is a failure.

We can't roll up our business by product! If we constrain by a specific product, we don't know what to do with invoice level discounts, freight charges and tax. All of our higher level views of the business by customer, sales agent, and promotions are forced to omit the product dimension.

In most businesses, this is unacceptable.

There is only one way to fix this problem. You have to take the invoice level data and allocate down to the line item level. Yes, there is some controversy in doing this allocation, and yes, you must make some arbitrary decisions, but the alternative is not being able to analyze your business in terms of your products.

We will replace the two fact tables with a single fact table, whose grain is the invoice line item. In other words, we will consistently drop to the most atomic child level when we do a parent-child dimensional design.

Remember from our discussion of Choosing the Grain (design tip #21), that we can "decorate" a measurement with everything that is known to be true at the time of the measurement. So our single line item grain fact table has the following dimensions:

- Date of overall invoice (dimension)
- Sales agent (dimension)
- Customer (dimension)
- Payment terms (dimension)
- Product (dimension)
- Promotion (dimension)

What do we do with the invoice number? It is certainly single-valued, even at the line item level, but we have already "exposed" everything we know about the invoice in our first four dimensions. We should keep the invoice number in the design but we don't need to make a dimension out of it because that dimension would turn out to be empty. We call this characteristic result a "degenerate dimension".

- Invoice number (degenerate dimension).

Now our facts for this line item child fact table include

- Number of units of product (additive fact)
- Gross extended product price (units X price) (additive fact)
- Net extended product price (units X (unit price - promotional discount)) (additive fact)
- Allocated invoice level discounts representing promotional allowances (additive fact)
- Allocated freight charges (additive fact)
- Allocated tax (additive fact)

We don't include the unit prices or discounts as physical facts because we can always divide the extended amounts by the number of units in our reporting application to get these non-additive quantities.

We can instantly recover the exact invoice level amounts by simply adding up all the line items under a specific invoice number. We don't need the separate invoice parent fact table because it is only a simple aggregation of our more granular line item child fact table. We have in no way compromised the invoice totals by performing the allocations down to the line item.

And, best of all, we can now smoothly roll up our business to the highest levels, slicing by product, and including the allocated amounts to get a true picture of our net revenues.

This technique of descending to the line item level and building a single granular fact table is at the heart of data warehouse modeling. It is our way of making good on the promise that we can "slice and dice the enterprise data every which way".

If you have further questions or comments about parent-child modeling situations, e-mail me at ralph@kimballgroup.com and I'll discuss them in an upcoming design tip.

Kimball Design Tip #24: Designing Dimensional In A Multinational Data Warehouse

By Ralph Kimball

If you are managing a multinational data warehouse you may have to face the problem of presenting the data warehouse content in a number of different languages. What parts of the warehouse need translating? Where do you store the various language versions? How do you deal with the open-endedness of having to provide more and more language versions?

There are many design issues in building a truly multinational data warehouse. I tried to cover them comprehensively in the Data Webhouse Toolkit book, so in this design tip we will focus only on how to present "language switchable" results to the end users, and we will not deal with

- international currencies,
- numeric punctuation,
- time zones,
- name and address parsing, or
- telephone number representations.

Our goal will be to switch cleanly among an open ended number of language representations, both for ad hoc querying and for viewing of standard reports. We also want to drill across a distributed multinational data warehouse that has implemented conformed dimensions.

Clearly, the bulk of our attention must focus on our dimensions. Dimensions are the repositories of almost all the text in our data warehouses. Dimensions contain the labels, the hierarchies, and the descriptors of our data. Dimensions drive the content of our user interfaces, and dimensions provide the content of the row and column headers in all of our reports.

The straightforward approach is to provide 1-to-1 translated copies of each dimension in each supported language. In other words, if we have a product dimension originally expressed in English, and we want French and German versions, we copy the English dimension row by row and column by column, preserving the keys and numeric indicators, while translating each textual attribute.

But we have to be careful. In order to preserve the user interfaces and final results of the English version, both the French and German product dimensions would have to also preserve the same

- 1-to-1 and many-to-1 relationships, and
- grouping logic, both for ad hoc reports and building aggregates.

The explicitly understood 1-to-1 and many-to-1 relationships should definitely be enforced by a entity-relationship model in the backroom staging area. That part is easy. But a subtle problem arises when doing the translations to make sure that no two distinct English attributes end up being translated into the same French or German attribute. For example, if the English words "scarlet" and "crimson" were both translated to the German word "rot", certain report totals would be different between the English and German versions. So we need an extra ETL step that verifies we have not introduced any duplicate translations from distinct English attributes.

The big advantage of this design is scalability, since we can always add a new language version without changing table structures, and without reloading the database.

We can allow the French or German end user to drill across a far flung distributed data warehouse if we REPLICATE the translated versions of the dimensions to all the remote data marts. Remember that when we drill across distributed data marts, we execute the primary queries remotely. That is why the translated dimensions need to reside on each target database.

When a French or German end user launches a drill across query, each remote data mart must use the correct translated dimensions. This will be handled easily enough by the original French/German application that formulates the request. Notice that each remote database must support "hot swappable dimensions" that allow this dimension switching to take place from query to query as different language requests are made. This is easy in a relational environment and may be tough in an OLAP environment.

Although we have accomplished a lot with this design, including a scalable approach to implementing a distributed multi-language data warehouse, we still have some unsolved issues that are just plain hard:

1) we cannot easily preserve sort orders across different language versions of the same report. Certainly we cannot make the translated attributes sort in the same order as the root language. If preserving sort orders is required, then we would need a hybrid dimension carrying both the root language and the second language, so that the SQL request could force the sort to be preserved in the root language, but show the second language as unsorted row labels. This is messy and results in double-size dimensions, but probably could be made to work.

2) if the root language is English, we probably will find that almost every other language results in translated text that is longer than the English. Don't ask me why. But this presents problems for formatting user interfaces as well as finished reports.

3) finally, if our set of languages extends beyond English and the main European languages, then even the 8-bit Extended ASCII character set will not be enough. All of the participating data marts would need to support the 16-bit UNICODE character set. Remember that our design needs the translated dimensions to reside on the target machines.

The design of a distributed multi-language data warehouse is a fascinating and important problem. I'd like to hear about interesting approaches and intractable problems. Write to me.

www.kimballgroup.com Number 23, May 2, 2001

Kimball Design Tip #23: A Rolling Prediction Of The Future, Now And In The Past

By Ralph Kimball

>>>> Original message from Richard to Ralph describing his problem <<<<

Hi Ralph & Co,

Help! I'm really stuck but it is an interesting design question for anyone who feels up to the challenge:

How do I predict the future? I need a nice simple (ha ha) fact table design...

I have an 'Account' dimension table in a data mart that has a 'Status' field, let's say this status is "OVERDUE". I also know the 'Status_Effective_Date', lets say this is "May-2nd-2001".

The business user wants to know the following : FOR THE NEXT MONTH, HOW MANY DAYS WILL THE ACCOUNT HAVE BEEN AT THE STATUS OF 'OVERDUE'?

If TODAY is "May-5th-2001", the user wants to see the following:

5th May: 4 days
6th May: 5 days
7th May: 6 days
etc... up to 4th-Jun-2001 - This is a rolling month in the future starting TODAY.

They then want to count all the accounts in the account table and group them by day bandings at status 'OVERDUE' to see the following:

5th May: 100 accounts - 4 to 6 days at 'OVERDUE'
6th May: 78 accounts - 4 to 6 days at 'OVERDUE'
etc

5th May: 200 account - 7 to 9 days at 'OVERDUE'
6th May: 245 account - 7 to 9 days at 'OVERDUE'
etc

I also need to travel back in time and do the same for any point in time + 1 month, however many accounts will have the 'Status_Ineffective_Date' then set and will no longer be at Status 'OVERDUE', but they will have a historical record in the account table to show when they were at this status via an SCD Type 2.

If anyone has similar experience with such a problem or know a design that could supports this, please let me know.... my brain hurts.

Regards,
Richard.

>>>> Ralph's Response to Richard <<<<

Hi Richard,

well I took a quick look at this. Maybe the following will work.

Assume your account dimension table has the following fields (as you described):

```
ACCOUNT_KEY
STATUS
STATUS_EFFECTIVE_DT
```

Now build a another table with the following fields

```
STATUS_EFFECTIVE_DT
STATUS_REPORTING_DT
DELTA
```

where DELTA is just the number of days between the effective date and the reporting date. You need a record in this table for every combination of EFFECTIVE date and REPORTING date your users could possibly be interested in. If you have EFFECTIVE dates going back one year (365 dates) and you want to report forward one year then you would need about 365*365 rows in this table, or about 133,000 rows. You may have other assumptions. Rather large but maybe workable if you have lots of RAM.

Anyway, join this second table to the first on the STATUS_EFFECTIVE_DT.

Then your first query should be satisfied with

```
SELECT STATUS_REPORTING_DT, DELTA
FROM .
WHERE STATUS = 'OVERDUE'
AND STATUS_REPORTING_DT BETWEEN 'May 5, 2001' and "June 4, 2001'
ORDER BY STATUS_REPORTING_DT
```

To get your banding report, maybe you can build a third table (a banding table) with fields

```
BAND_NAME
UPPER_DELTA
LOWER_DELTA
```

You join this table to the second where

```
DELTA <= UPPER_DELTA
DELTA > LOWER_DELTA
```

Your SQL is something like

```
SELECT BAND_NAME, COUNT(*)
FROM (all three tables joined as described)
WHERE STATUS_REPORTING_DT BETWEEN 'May 5, 2001' and "June 4, 2001'
AND  DELTA <= UPPER_DELTA
AND DELTA > LOWER_DELTA
ORDER BY UPPER_DELTA
GROUP BY BAND_NAME
```

I describe this value banding approach in the Lifecycle Toolkit on pages 251-252. Try this on a small example in Access. Let me know how it turns out.

Good luck,

Ralph Kimball

>>>> Richard's Follow-up Response After He Tried The Solution <<<<

Hi Ralph,

I'm very pleased to say that your recommendations worked! - Thank you. I tried it out on a few sample rows in SQL Server 7 and the results look good. I've now taken it to the next level to include a Status_Ineffective_Date so we can exclude accounts that stopped the 'OVERDUE' status somewhere within the reporting period. Maybe you would like to use this is one of your Design Tip mails?

Firstly, I slightly modified your SQL to the following:

```
SELECT
  COUNT(account.Account_Id),
  days_overdue.reporting_date
FROM
  account,
  days_overdue
WHERE
  ( account.Status_Effective_Date=days_overdue.effective_date  )
  AND  (
  days_overdue.reporting_date  BETWEEN  '05/05/2001' AND '06/04/2001'
  AND  days_overdue.delta  >=  10 {an arbitrary number or banding that the user
decides at run time}
  )
GROUP BY
  days_overdue.reporting_date
```

The key to the above SQL is to constrain on 'delta' (which we call 'days_overdue'), or else the query pulls back every account for every day that the reporting period is for, and the COUNT is always the same number. The constraint can be =, <, > or BETWEEN, but must be present.

Secondly, to exclude accounts that are no longer overdue (we assume we know the date they will pay up or have paid up!) we simply include a status_ineffective_date in the account table and can do the following:

```
SELECT
  COUNT(account.Account_Id),
  days_overdue.reporting_date
FROM
  account,
  days_overdue
WHERE
  ( account.Status_Effective_Date=days_overdue.effective_date  )
  AND  (
  days_overdue.reporting_date  BETWEEN  '05/05/2001' AND '06/04/2001'
  AND  days_overdue.delta  >=  10
```

```
                AND account.Status_Ineffective_Date  >  days_overdue.reporting_date
                )
              GROUP BY
                days_overdue.reporting_date
```

You would of course have to set the ineffective_date for all currently overdue accounts to something in the future like 01/01/3000. This also fits perfectly with the SCD type 2 where we are creating a new row in the account dimension each time the status changes, to give us a complete historical comparison!!

Thank you very much for your help and I wish you all the best,
Richard Tomlinson.

>>>> Final Postscript <<<<

This was a fun little puzzle. It's great to receive these but if you send me a request, PLEASE be understanding! I certainly can't deal with all of them, and in some cases the results won't be so serendipitous.

Kimball Design Tip #22: Variable Depth Customer Dimensions

By Ralph Kimball

The customer dimension is probably the most challenging dimension in a data warehouse. In a large organization, the customer dimension can be

1) huge, with millions of records
2) wide, with dozens of attributes
3) slowly changing, but sometimes quickly changing

To make matters worse, in the biggest customer dimensions we often have two categories of customers which I will call Visitor and Customer.

The Visitor is anonymous. We may see them more than once, but we don't know their name or anything about them. On a web site, all we have for a Visitor is a cookie that tells us they have returned. In a retail operation, a Visitor engages in an anonymous transaction. Perhaps we have a credit card number or a simple shopper identity card, but we assume here that no meaningful demographic data accompanies a Visitor.

The Customer, on the other hand, is reliably registered with us. We know the Customer's name, address, and as much demographics as we care to elicit directly from them or purchase from third parties. We have a shipping address, a payment history, and perhaps a credit history with each Customer.

Let's assume that at the most granular level of our data collection, 80% of the fact table measurements involve Visitors, and 20% involve Customers. Let us further assume that we accumulate simple behavior scores for Visitors consisting only of Recency (when was the last time we saw them), Frequency (how many times have we seen them), and Intensity (how much business have we done with them). So in this simple design we only have three attributes/measures for a Visitor.

On the other hand let's assume we have 50 attributes/measures for a Customer, covering all the components of location, payment behavior, credit behavior, directly elicited demographic attributes, and third party purchased demographic attributes.

Let's make some rather specific and limiting assumptions for the sake of a clean design. We can relax some of these assumptions later...

First, let's combine Visitors and Customers into a single logical dimension called Shopper. We will give the true physical Visitor/Customer a single permanent Shopper ID, but we will make the key to the table a surrogate key so that we can track changes to the Shopper over time. Logically, our dimension looks like

attributes for both Visitors and Customers:

Shopper Surrogate Key	<== simple integer assigned sequentially with each change
Shopper ID	<== permanent fixed ID for each physical shopper

Recency Date	<== date of last visit, Type 1: overwritten
Frequency	<== number of visits, Type 1: overwritten
Intensity	<== total amount of business, e.g., sales dollars, Type 1

attributes for Customers only:

5 name attributes	<== first, middle, last, gender, greeting
10 location attributes	<== address components
5 payment behavior attributes	
5 credit behavior attributes	
10 direct demographic attributes	
15 purchased demographic attributes	

One strong assumption we have made here is to include the Recency, Frequency, and Intensity information as dimensional attributes rather than as facts, and also to continuously overwrite them as time progresses (Type 1 slowly changing dimension). This assumption makes our Shopper dimension very powerful. We can do classic shopper segmentation directly off the dimension without navigating a fact table in a complex application. See the discussion of Recency-Frequency-Intensity segmentation in my Webhouse Toolkit book, starting on page 73.

If we assume that many of the final 50 Customer attributes are textual, we could have a total record width of 500 bytes, or more.

Suppose we have 20 million Shoppers (16 million Visitors and 4 million registered Customers). Obviously we are worried that in 80% of our records, the trailing 50 fields have no data! In a 10 gigabyte dimension, this gets our attention.

This is a clear case where, depending on the database, we may wish to introduce a snowflake.

In databases with variable width records, like Oracle, we can simply build a single shopper dimension with all the above fields, disregarding the empty fields issue. The majority of the shopper records, which are simple Visitors, remain narrow, because in these databases, the null fields take up zero disk space.

But in fixed width databases, we probably don't want to live with the empty fields for all the Visitors, and so we break the dimension into a base dimension and a snowflaked subdimension:

Base:

Shopper Surrogate Key	<== simple integer assigned sequentially with each change
Shopper ID	<== permanent fixed ID for each physical shopper
Recency Date	<== date of last visit, Type 1: overwritten
Frequency	<== number of visits, Type 1: overwritten
Intensity	
Customer Surrogate Key	<== new field to link to the snowflake

Snowflake:

Customer Surrogate Key	<== 1:1 matching field for those shoppers who are Customers 5
name attributes	
10 location attributes	
5 payment behavior attributes	
5 credit behavior attributes	
10 direct demographic attributes	
15 purchased demographic attributes	

In a fixed width database, using our previous assumptions, the base Shopper dimension is 20 million X 25 bytes = 500 MB, and the snowflake dimension is 4 million X 475 bytes = 1.9 gigabytes. We have saved 8 gigabytes by using the snowflake.

If you have a query tool that insists on a classic star schema with no snowflakes, then hide the snowflake under a view declaration.

This is the basic foundation for a variable depth customer dimension. I have left lots of issues on the table, including

- how to administer the changing attributes in the Customer portion of the dimension
- how to not lose the history of the recency-frequency-intensity measures given we are overwriting them.
- how to add hierarchical relationships to all of this if the customers are organizations
- how to deal with security and privacy design constraints that may be imposed on top of all this

Stay tuned for the next Design Tip...

Write to me with questions or comments about variable depth customer dimensions.

Kimball Design Tip #21: Declaring The Grain

By Ralph Kimball

The most important step in a dimensional design is declaring the grain of the fact table. Declaring the grain means saying EXACTLY what a fact table record represents. Remember that a fact table record captures a measurement. Example declarations include:

* an individual line item on a customer's retail sales ticket as measured by a scanner device
* an individual transaction against an insurance policy
* a line item on a bill received from a doctor
* an individual boarding pass used by someone making an airplane flight

When you make such a grain declaration, you can have a very precise discussion of which dimensions are possible and which are not. For example, a line item of a doctor's bill (example #3) arguably would have the following dimensions:

* Date (of treatment)
* Doctor (maybe called "provider")
* Patient
* Procedure
* Primary Diagnosis
* Location (presumably the doctor's office)
* Billing Organization (an organization the doctor belongs to)
* Responsible Party (either the patient, or the patient's legal guardian)
* Primary Payer (often an insurance plan)
* Secondary Payer (maybe the responsible party's spouse's insurance plan) and quite possibly others.

If you have been following this example, I hope you have noticed some powerful effects from declaring the grain. First, we can visualize the dimensionality of the doctor-bill-line-item very precisely and we can have Yes/No discussions relative to our data sources about whether a dimension can be attached to this data. For example, we probably would exclude "treatment outcome" from this example because most medical billing data doesn't tie to any notion of outcome.

BUT, a general E/R oriented "data model" of doctor visits might well include treatment outcome. After all, in an abstract sense, doesn't every treatment have an outcome???

The discipline of insisting on the grain declaration at the beginning of a dimensional design keeps you from making this kind of mistake. A model of billable doctor visits that included treatment outcome would look like a dimensional model but it wouldn't be implementable. This is my main gripe with many of the current offerings of "standard schemas" in books and CDs. Since they have no grain discipline, they often combine entities that don't exist together in real data sources.

A second major insight from the doctor-bill-line-item grain declaration is that this very atomic grain gives rise to a lot of dimensions! We have listed 10 dimensions above, and those of you who are experts in health care billing probably know of a couple more. It is an interesting realization that the smaller and more atomic the measurement (fact table record), the more things you know for sure. And hence the more dimensions! This is another way of explaining why atomic data resists the "ad hoc attack" by end users. Atomic data has the most dimensionality and so it can be constrained and rolled up in every way that is possible for that data source. Atomic data is a perfect match for the dimensional approach.

All of the grain declarations listed at the beginning of this design tip represent the lowest possible granularity of their respective data sources. These data measurements are "atomic" and cannot be divided further. But it is quite possible to declare higher level grains for each of these data sources that represent aggregations of atomic data:

* all the sales for a product in a store on a day
* insurance policy transaction totals by month by line of business
* charged amount totals by treatment by diagnosis by month
* counts of passengers and other flight customer satisfaction issues by route by month

These higher levels of aggregation will almost always have fewer, smaller dimensions. Our doctor example might end up with only the dimensions of

Month
Doctor
Procedure
Diagnosis

It would be undesirable in an aggregated fact table to include all the original dimensions of the atomic data because you would usually end up with very little aggregation!

Since useful aggregations necessarily shrink dimensions and remove dimensions, this leads to the realization that aggregated data always needs to be used in conjunction with its base atomic data, because aggregated data has less dimensional detail. Some authors get confused on this point, and after declaring that datamarts necessarily consist of aggregated data, they criticize the datamarts for "anticipating the business question". All of this misunderstanding goes away when aggregated data is made available TOGETHER with the atomic data from which it is derived.

The most important result of declaring the grain of the fact table is anchoring the discussion of the dimensions. But declaring the grain allows you to be equally clear about the measured numeric facts. Simply put, the facts must be true to the grain. In our doctor example, the most obvious measured fact would be "billed amount". Other facts relating to the specific treatment received by that patient at that time are possible. But "helpful" facts like amount billed year-to-date to this patient for all treatments are not true to the grain. When fact records are combined in arbitrary ways by a reporting application, these untrue-to-the-grain facts produce nonesensical, useless results. They need to be left out of the design. Calculate such aggregate measures in your application.

In summary, try to do your dimensional designs using the following four steps, in order:

1. decide on your sources of data
2. declare the grain of the fact table (preferably at the most atomic level)
3. add dimensions for "everything you know" about this grain
4. add numeric measured facts true to the grain.

Write to me with questions or comments about declaring the grain.

www.kimballgroup.com Number 20, February 27, 2001

Kimball Design Tip #20: Sparse Facts And Facts With Short Lifetimes

By Ralph Kimball

Fact tables are built around numerical measurements. When a measurement is taken, a fact record comes into existence. The measurement can be the amount of a sale, the value of a transaction, a running balance at the end of a month, the yield of a manufacturing process run, or even a classic laboratory measurement. If we record several numbers at the same time, we can often put them together in the same fact record.

We surround the measurement(s) with all the things we know to be true at the precise moment of the measurement. Besides a time stamp, we often know things like customer, product, market condition, employee, status, supplier, and many other entities depending on the process supplying us the measurement.

We package all the things we know into descriptive text-laden dimension records and connect the facts to the dimension records through a foreign key / primary key (FK/PK) relationship.

This leads to the classic organization of a fact table (shown here with N dimensions and two facts called Dollars and Units):

```
dimkey1 (FK)
dimkey2 (FK)
dimkey3 (FK)
..
dimkeyN (FK)
Dollars
Units
```

The Dollars and Units fields are reserved placeholders for those specific measurements. This design carries the implicit assumptions that

1) these two measures are usually present together,
2) these are the only measures in this process
3) there are lots of measurement events, in other words, it is worthwhile to devote this fixed format table to these measures.

But what happens when all three of these assumptions break down? This happens frequently in complex financial investment tracking where every investment instrument has idiosyncratic measures. It also happens in industrial manufacturing processes where the batch runs are short and each batch type has a host of special measures. And finally, clinical and medical lab environments are dominated by hundreds of special measurements, none of which occur very frequently. All three of these examples can be described as "sparse facts".

You can't just extend the classic fact table design to handle sparse facts. You would have an unworkably long list of fact fields, most of which would be null in a given record.

The answer is to add a special "fact dimension" and shrink the list of actual numeric facts down to a single AMOUNT field:

```
dimkey1 (FK)
dimkey2 (FK)
dimkey3 (FK)
..
dimkeyN (FK)
factkey (FK) <== additional dimension
Amount
```

The "fact dimension" describes the meaning of the measurement amount. It contains what used to be the field name of the fact, as well as the unit of measure, and any additivity restrictions. For instance, if the measurement is an inventory-like (or balance-like) fact, then it may be fully additive across all the dimensions except time. But if it is a full blown intensity measurement like temperature, then it is completely non-additive. Summarizing across non-additive dimensions requires averaging, not summing.

This approach is elegant because it is superbly flexible. You add new measurement types just by adding new records in the fact dimension, not by altering the structure of the table. You also eliminate all the NULLs in the classic design because a record only exists if the measurement exists.

But there are some significant tradeoffs. You may be generating a LOT of records. If some of your measurements give you 10 numeric results, now you have 10 records rather than the single record you had in the classic design. For extremely sparse situations, this is a great compromise. But as the density of the facts grows in the dimensional space you have created, you start papering the universe with records. At some point you have to return to the classic format.

This approach also makes applications more complicated. Combining two numbers that have been taken as part of a single measurement event is more difficult because now you have to fetch two records. SQL makes this awkward because SQL likes to do arithmetic WITHIN a record, not across records. And you have to be very careful that you don't mix incompatible Amounts in a calculation, since all the numeric measures exist within the single Amount field.

But these tradeoffs are clearly worth it if you live in the investment world, the manufacturing world, or the clinical/laboratory world.

Write to me about variations you have used on this theme and I'll talk about them in a future Design Tip.

www.kimballgroup.com Number 19, February 5, 2001

Kimball Design Tip #19: Replicating Dimensions Correctly

By Ralph Kimball

The secret of building a distributed data warehouse is using conformed dimensions. In a distributed data warehouse many separate sources of measurements are maintained by different departments. These measurements are usually presented in "fact tables". One department may measure item manufacturing results, and another may measure item inventory. A third department may measure item sales, and a fourth may measure item comments and complaints. Clearly all these departments have a common interest in "item". We can build a distributed data warehouse if we can get these four departments to agree on the definition of items.

Actually, we need say this more strongly. We can build a distributed data warehouse if these four departments all use the conformed item dimension. And yes, these departments need to conform all other dimensions they have in common, such as time and customer. But we will focus only on the item dimension in this discussion.

The simplest way to conform the item dimension is for all four departments to use the same, identical item dimension table. Same keys, same attributes, same everything. Each of the four departments, of course, must then convert any private item keys in their fact tables to the public surrogate keys used in the conformed item dimension table. I described this surrogate key pipeline in an article which can be found at www.dbmsmag.com/9806d05.html.

A more complex way to use the conformed item dimension is to allow one of the departments to use a subset of the item dimension table. Suppose the department measuring item comments only records the comments at the brand level, not at the individual item level. It would be acceptable for this departments to use a shrunken version of the item table that only carries information down to the brand level. Of course, this would force any "drill-across" application that was combining information from this data mart with other data marts to seek the lowest common level of the various item dimension tables, which in this case would be at the brand, not individual item, level.

The real payoff of using conformed dimensions is being able to drill across separate data marts (fact tables). If you can constrain and group on the same item characteristics in each separate data mart, you can then line up the separate answer sets using the row headers that come out of the item dimension table. So, on one report line you can show item production, item inventory, and item sales at a very detailed level, and if you move up to the brand level, you can include counts of item complaints.

Drilling across is the key conceptual step in using a distributed data warehouse, and avoiding the need to have it centralized.

But administering a conformed dimension requires special discipline. The overall organization needs a "dimension authority", in this case an item czar. This dimension authority is responsible for maintaining the item dimension and replicating it successfully to all the data mart clients who make any use of item in their fact tables. We need to take seriously the task of replicating the dimension and enforcing its consistent use.

It would be a disaster if we drilled across several data marts accumulating results for a report, when half of the data marts had yesterday's version of the dimension and half had today's. The results would be insidiously wrong. The row labels would not mean the same thing if any of the definitions of any of the reporting attributes had been adjusted. For example, if a category manager had changed the definition of one of the item categories, the reported results across these out-of-synch row headers would be wrong. And yes, category managers have the authority to change category labels and many other attributes in the item dimension.

A similar issue arises when any of the datamarts use an aggregate navigator that automatically substitutes a compressed item dimension table and an associated compressed fact table at the time the user specifies a query. For example, if the user asks for a "share" of a specific item to an entire item category, we usually perform two queries against the fact table and take the ratio in order to compute the share. The first query requests a very specific product and cannot use the compressed aggregate tables. But the second query is just getting the category total and is a prime candidate for aggregate navigation.

The moral of this story is that if the dimension authority has released a new item table, then all the aggregate tables affected by changes made in the item table must be adjusted. If some low level items were moved from one existing category to another, then not only is the item table changed, but any fact table at the category level would have to be adjusted.

We can summarize the two big responsibilities for correctly replicating dimensions:

1) All client data marts must deploy the replicated dimension simultaneously so that any end user drilling across these data marts will be using a consistent set of dimension attributes,

and

2) All client data marts must remove aggregates affected by changes in the dimension, and only make these aggregates available to the end users when they have been made completely consistent with the base fact tables and the new rollup logic.

This topic of Dimensional Replication is criterion #9 in my recommended list of dimensionally friendly criteria. Both Microsoft and Cognos have posted detailed responses to all twenty criteria that make up their "dimensionally friendly systems". Check out their submissions to see what they did about dimensional replication. We are expecting more vendors to rate their systems against these twenty criteria. You can find the complete set of criteria and the vendor responses at www.ralphkimball.com/html/dimension.html.

If you are an end user and you would like to rate your overall system perhaps supplied by several vendors, please contact me. I would be interested in adding such ratings to the web site.

www.kimballgroup.com Number 18, January 21, 2001

Kimball Design Tip #18: Taking The Publishing Metaphor Seriously

By Ralph Kimball

In this design tip I want to share a perspective that I take very seriously, and in some ways is the foundation for all my work in data warehousing. It is the publishing metaphor. Consider the following scenario.

Imagine that you have been asked to take over responsibility for a high quality magazine. You have been named editor-in-chief and you have been given broad latitude to manage the content, style, and delivery of this magazine.

If you approach this responsibility thoughtfully, in my opinion you should do the following 12 things:

* identify your readers demographically
* find out what the readers want in this kind of magazine
* identify the "best" readers who will renew their subscriptions and buy products from the magazine's advertisers
* find potential new readers, and make them aware of the magazine
* choose the magazine content most appealing to the target readers
* make layout and rendering decisions that maximize the pleasure of the readers
* uphold high quality writing and editing standards, and adopt a consistent presentation style
* continuously monitor the accuracy of the articles and the advertiser's claims
* keep the reader's trust
* develop a good network of writers and contributors
* draw in advertising and run the magazine profitably
* keep the business owners happy

If you do a good job with all these responsibilities, I think you will be a great editor-in-chief! Conversely, go down through the list and imagine what happens if you omit any single item. Ultimately your magazine would have problems.

While these responsibilities may seem obvious, let's list some dubious items that should be non-goals:

* build the magazine around the technology of a particular printing press
* put most of your management energy into the printing press operational efficiencies
* use a highly technical and complex writing style that many readers may not understand
* use an intricate and crowded layout style that is difficult to read and navigate

The lesson for magazine publishing is that serving the readers effectively is the whole ball game. By building the whole business on the foundation of serving the readers, your magazine is likely to be successful.

The point of this metaphor, of course, is to draw the parallel between being a conventional publisher and being a data warehouse project manager.

I am convinced that the correct job description for a data warehouse project manager is "publish the right data". Your main responsibility is to serve your readers who are your end users. While you will certainly use technology to deliver your data warehouse, the technology is at best a means to an end. The technology and the techniques you use to build your data warehouses should not show up directly in your top 12 responsibilities, but the appropriate technologies and techniques will become much more obvious if your over-riding goal is to effectively publish the right data.

Let's recast the 12 magazine publishing responsibilities as data warehouse responsibilities:

* understand your end users by business area, job responsibilities, and computer tolerance
* find out the decisions the end users want to make with the help of the data warehouse
* identify the "best" end users who make effective decisions using the data warehouse
* find potential new end users, and make them aware of the data warehouse
* choose the most effective, actionable subset of the data to present in the data warehouse, drawn from the vast universe of possible data in your organization
* make the end user screens and applications MUCH simpler and more template driven, explicitly matching the screens to the cognitive processing profiles of your end users
* make sure your data is accurate and can be trusted, labeling it consistently across the enterprise
* continuously monitor the accuracy of the data and the content of the delivered reports
* keep the end user's trust
* continuously search for new data sources, and continuously adapt the data warehouse to changing data profiles and reporting requirements
* take a portion of the credit for end user decisions made using the data warehouse, and use these successes to justify your staffing, software, and hardware expenditures
* keep the end users, end user executives, and your boss happy

If you do a good job with all these responsibilities, I think you will be a great data warehouse project leader! Conversely, go down through the list and imagine what happens if you omit any single item. Ultimately your data warehouse would have serious problems.

I urge you to contrast this view of a data warehouse project manager's job with your own job description. Chances are the above list is much more oriented toward end user and business issues, and may not even sound like a job in IT. But in my opinion, that is what makes this job interesting. Write to me with your reactions.

Kimball Design Tip #17: Populating Hierarchy Helper Tables

By Lawrence Corr

This month's tip follows on from Ralph's September 1998 article 'Help for Hierarchies' (www.dbmsmag.com/9809d05.html) which addresses hierarchical structures of variable depth which are most often represented in relational databases as recursive relationships sometimes known as 'pigs ears' or 'fish hooks'.

Below is the definition of a simple company dimension which contains such a recursive relationship between the foreign key PARENT_KEY and primary key COMPANY_KEY

```
Create table COMPANY (
 COMPANY_KEY         INTEGER NOT NULL,
 COMPANY_NAME        VARCHAR2(50),
 PARENT_KEY          INTEGER);
```

While this is efficient for storing information on organizational structures it is not possible to navigate or rollup facts within these hierarchies using the non-procedural SQL that can be generated by commercial query tools. Ralph's original article describes a helper table similar to the one below which contains one record for each separate path from each company in the organization tree to itself and to every subsidiary below it which solves this problem.

```
Create table COMPANY_STRUCTURE (
 PARENT_KEY                 INTEGER NOT NULL,
 SUBSIDIARY_KEY             INTEGER NOT NULL,
 SUBSIDIARY_LEVEL           INTEGER NOT NULL,
 SEQUENCE_NUMBER            INTEGER NOT NULL,
 LOWEST_FLAG                CHAR(1),
 HIGHEST_FLAG               CHAR(1),
 PARENT_COMPANY             VARCHAR2(50),
 SUBSIDIARY_COMPANY         VARCHAR2(50));
```

The last two columns in this example which denormalize the company names into this table are not strictly necessary but have been added to make it easy to see what's going on later.

The following PL/SQL stored procedure demonstrates one possible technique for populating this 'hierarchy explosion' table from the COMPANY table on
Oracle:

```
CREATE or Replace procedure COMPANY_EXPLOSION_SP as

CURSOR Get_Roots is

select  COMPANY_KEY ROOT_KEY,
        decode(PARENT_KEY, NULL,'Y','N') HIGHEST_FLAG,
        COMPANY_NAME ROOT_COMPANY
```

```
from COMPANY;

BEGIN

For Roots in Get_Roots
LOOP
        insert into COMPANY_STRUCTURE
        (PARENT_KEY,
         SUBSIDIARY_KEY,
         SUBSIDIARY_LEVEL,
         SEQUENCE_NUMBER,
         LOWEST_FLAG,
         HIGHEST_FLAG,
         PARENT_COMPANY,
         SUBSIDIARY_COMPANY)
        select
          roots.ROOT_KEY,
          COMPANY_KEY,
          LEVEL - 1,
          ROWNUM,
          'N',
          roots.HIGHEST_FLAG,
          roots.ROOT_COMPANY,
          COMPANY_NAME
        from
          COMPANY
         Start with COMPANY_KEY = roots.ROOT_KEY
         connect by prior COMPANY_KEY = PARENT_KEY;
END LOOP;

update COMPANY_STRUCTURE
   SET LOWEST_FLAG = 'Y'
where not exists (select * from COMPANY
where PARENT_KEY = COMPANY_STRUCTURE.SUBSIDIARY_KEY);

COMMIT;
END;  /* of procedure */
```

This solution takes advantage of Oracle's CONNECT BY SQL extension to walk each tree in the data. While CONNECT BY is very useful within this procedure it could not be used by an ad hoc query tool. If the tool could generate this syntax to explore the recursive relationship it can not in the same statement join to a fact table. Even if Oracle was to remove this somewhat arbitary limitation, the performance at query time would probably be not too good.

The following fictional company data will help you to understand the COMPANY_STRUCTURE table and COMPANY_EXPLOSION_SP procedure:

```
/* column order is Company_key,Company_name,Parent_key */ insert into company values (100,'Microsoft',NULL); insert into company values (101,'Software',100); insert into company values (102,'Consulting',101); insert into company values (103,'Products',101); insert into company values (104,'Office',103); insert into company values (105,'Visio',104); insert into company values (106,'Visio Europe',105); insert into company values (107,'Back Office',103); insert into company values (108,'SQL Server',107); insert into company values (109,'OLAP Services',108); insert into company values (110,'DTS',108); insert into company values (111,'Repository',108); insert into company values (112,'Developer Tools',103); insert into company values (113,'Windows',103); insert into company values
```

```
(114,'Entertainment',103); insert into company values (115,'Games',114); insert into
company values (116,'Multimedia',114); insert into company values (117,'Education',101);
insert into company values (118,'Online Services',100); insert into company values
(119,'WebTV',118); insert into company values (120,'MSN',118); insert into company values
(121,'MSN.co.uk',120); insert into company values (122,'Hotmail.com',120); insert into
company values (123,'MSNBC',120); insert into company values (124,'MSNBC Online',123);
insert into company values (125,'Expedia',120); insert into company values
(126,'Expedia.co.uk',125);
/* End example data */
```

The procedure will take the 27 COMPANY records and create 110 COMPANY_STRUCTURE records make up of one big tree (Microsoft) with 27 nodes and 26 smaller trees. For large datasets, you may find that the performance can be enhanced by adding a pair of concatenated indexes on the connect by cloumns. In this example one on COMPANY_KEY,PARENT_KEY and the other on PARENT_KEY,COMPANY_KEY.

If you want to visualize the tree structure textually the following query displays an indented subsidiary list for Microsoft:

```
select LPAD( ' ', 3*(SUBSIDIARY_LEVEL)) || SUBSIDIARY_COMPANY from
COMPANY_STRUCTURE order by SEQUENCE_NUMBER where parent_key = 100
```

The SEQUENCE_NUMBER has been added since the original article, it numbers nodes top to bottom, left to right. It allows the correct level 2 nodes to be sorted below their matching level 1 nodes.

For a graphical version of the organization tree take a look at VISIO 2000 Enterprise Edition which has a database or text file driven organization chart wizard. With the help of VBA script, a view on the COMPANY_STRUCTURE table and a fact table it might automate the generation of just the HTML pages you want.

I would be very grateful if someone could email me with DB2 UDB and Microsoft SQL Server procedures to create the same results. More efficient Oracle implementations would be interesting too. The best examples will be credited and included in a future Intelligent Enterprise column.

www.kimballgroup.com Number 16, December 8, 2000

Kimball Design Tip #16: Hot Swappable Dimensions

By Ralph Kimball

Criterion #18 in the list of Dimensionally Friendly Criteria defines a "hot swappable dimension", which is a dimension with two or more alternative versions. If the dimension is hot swappable, any of the alternative versions of the dimension can be chosen at query time.

There are a number of situations where alternative versions of the same dimension can be very useful. Here are three interesting situations:

1) An investment banking house makes available to its clients a large fact table that tracks stocks and bonds on a daily basis over a several year period. The "investment dimension" in this fact table provides information about each stock and bond. But this investment dimension is customized to each client accessing the fact table so that they can each describe and group the investments in interesting and proprietary ways. The different versions of the investment dimension may be completely different including incompatible attribute names and different hierarchy schemes. All clients use the same fact table (hence it only needs to be stored in one place) but each client uses their own investment dimension table as the basis for analyzing the price movements of the stocks and bonds. Viewed from the database server, the clients are busily hot-swapping the investment dimension with each query.

2) A retail bank creates a single large fact table that records the month end balances of all the account types in the bank, including checking, savings, mortgage, credit card, personal loans, small business loans, certificates of deposit, student loans, and others. This is a classic case of "heterogeneous products" because the detailed descriptions of each of these account types are wildly different. There is no single description template that can adequately deal with the complexities of all these account types. Therefore we build a simplified account dimension that is meant to join to all the accounts uniformly. We use this simplified account dimension when we are doing cross-selling and up-selling analyses and are looking at the overall portfolio of a customer. But when we restrict our attention to a single account type (e.g., mortgages), we swap in a drastically wider (more fields) dimension that only contains mortgage related attributes. We can do this when we are confident that we have restricted the analysis to just one kind of account. If we have 20 lines of business, we have 21 account dimensions: 1 simplified dimension describing all the accounts and 20 extended dimensions describing disjoint sets of similar accounts.

3) A manufacturer wishes to make its shipments fact table available to its trading partners, but needs to shield the orders of the partners from each other. In this case each partner gets their own version of the partner dimension with only their own name appearing in plain text. All the other partners show as "OTHER". Additionally, a mandatory weighting factor field in the dimension is set to 1 for the intended partner and is set to 0 for all others. This weighting factor is uniformly multiplied against all facts in the fact table. In this way, a single shipments fact table can be used to support competitive trading partners in a secure way.

Hot swapping dimensions is straightforward in a standard relational database since the joins between tables can be specified at query time. But if referential integrity is required between the dimension tables and the fact table, then every swappable version of the dimension must contain the full key set and hence the full set of dimension records. In this case if the swappable dimension is being used to restrict the access to the fact table (as in examples 2 and 3), the restricted rows of the dimension table must contain dummy or null values.

Hot swapping of dimensions is more of a challenge for OLAP systems where the identity of the dimension is built deeply into the fabric of the OLAP data cube. To see how Cognos (PowerPlay) and Microsoft (Analysis Services) handle this hot swappable dimension criterion in their OLAP products, see their white papers at www.ralphkimball.com/html/dimension.html.

Kimball Design Tip #15: Combining SCD Techniques

By Margy Ross

The acronym, SCD, is a keyword in a dimensional modeler's vernacular. As most of you know, SCD is short-hand for slowly changing dimensions.

There are several well-documented techniques for dealing with slowly changing dimension attributes. Briefly, with SCD Type 1, the attribute value is overwritten with the new value, obliterating the historical attribute values. For example, when the product roll-up changes for a given product, the roll-up attribute is merely updated with the current value. Using Type 2, a new record with the new attributes is added to the dimension table. Historical fact table rows continue to reference the old dimension key with the old roll-up attribute; going forward, the fact table rows will reference the new surrogate key with the new roll-up thereby perfectly partitioning history. Finally, with Type 3, attributes are added to the dimension table to support two simultaneous roll-ups - perhaps the current product roll-up as well as "current version minus one", or current version and original.

In my experience, data warehouse teams are often asked to preserve historical attributes, while also supporting the ability to report historical performance data according to the current attribute values. None of the standard SCD techniques enable this requirement independently. However, by combining techniques, you can elegantly provide this capability in your dimensional models.

We'll begin by using the SCD workhorse, Type 2, to capture attribute changes. When the product roll-up changes, we'll add another row to the dimension table with a new surrogate key. We'll then embellish the dimension table with additional attributes to reflect the current roll-up. In the most current dimension record for a given product, the current roll-up attribute will be identical to the historically accurate "as was" roll-up attribute. For all prior dimension rows for a given product, the current roll-up attribute will be overwritten to reflect the current state of the world. If we want to see historical facts based on the current roll-up structure, we'll filter or summarize on the current attributes. If we constrain or summarize on the "as was" attributes, we'll see facts as they rolled up at that point in time.

We've described a hybrid approach that combines the three fundamental SCD techniques. We're creating new rows to capture change (Type 2), adding attributes to reflect an alternative view of the world (Type 3), which are overwritten for all earlier dimension rows for a given product (Type 1). As a student recently suggested, perhaps we should refer to this as Type 6 (2+3+1)…

www.kimballgroup.com Number 14, October 5, 2000

Kimball Design Tip #14: Reporting Balances On Arbitrarily Chosen Days In A Transaction Fact Table

By Ralph Kimball

Design Tip #13 showed how to attach a slowly changing dimension table (the account table in a bank) to a fast moving transaction grained fact table (the account transactions). We saw how the slowly changing dimension was itself something like a fact table because it was the target of a set of transactions that modified the account profiles.

In this Design Tip, we switch our focus to the big fact table that records all the transactions against the bank account. Let's boil this fact table down to a really simple design for discussion purposes:

Date Key (FK)
Account Key (FK)
Transaction Type Key (FK)
Transaction Sequence Number
Final Flag
Amount
Balance

Here's what the fields contain: Date Key = surrogate key pointing to daily grain calendar dimension; Account Key = surrogate key pointing to account table; Transaction Type Key = surrogate key pointing to a small table of allowed transaction types; Transaction Sequence Number = continuously increasing numeric sequence number running for the lifetime of the account; Final Flag = TRUE if this is the last transaction on a given day, FALSE otherwise; Amount = amount of this transaction; Balance = resulting account balance after the transaction.

Like all transaction grained tables, this one only has a record in it if a transaction was performed. If an account was quiet for two weeks, say October 1 through 14, there will be ZERO records in the fact table for this account in this time span. But suppose we want to ask what all the account balances were on October 5?

In this case we need to look for the most recent previous fact record for each account on or before our requested date.

Here's some tested SQL that does the trick:

```
SELECT a.acctnum, f.balance
FROM fact f, account a
WHERE f.account_key = a.account_key
AND f.final_flag = 'True'
AND f.date_key =
  (SELECT MAX(g.date_key)
   FROM fact g
   WHERE g.account_key = f.account_key
   AND g.date_key IN
      (SELECT t.date_key
                FROM time t
                WHERE t.fulldate <= 'October 5, 2000'))
```

If you study this SQL you probably have a couple of questions!

Question 1. RALPH, how could you use a time surrogate key as the basis for a constraint??? The whole point of surrogate keys is that they have no semantics.

Answer 1. Yes, except for the time surrogate key which is a set of integers running from 1 to N. For a completely separate reason, we have already placed a predictable ordering on the time surrogate key. We need the time surrogate key to be ordered so we can impose physical partitioning on this large fact table based on this key. This neatly segments the physical table so we can perform discrete administrative actions on certain ranges of times, like moving to off-line storage, or dropping and rebuilding indexes. This time dimension is the ONLY dimension that has any logic to the surrogate keys and is the only one we dare place application constraints on. We use this ordering to advantage in the above SQL when we find the most recent prior end-of-day transaction.

Question 2. Why did you go to the trouble of linking off to the time table when you could have just constrained a conventional time stamp in the fact table? Then we wouldn't need a surrogate key (seems like a lot of trouble...).

Answer 2. Putting in a time stamp instead of the surrogate key introduces a whole set of problems we solved with surrogate keys, including null dates, inapplicable dates, and hasn't-happened dates. We have discussed all of this in other places. But more important, a naked date stamp in the fact table doesn't let us do realistic complex time constraints. What if instead of October 5, the request had been for the balances on "3rd quarter Federal Reserve reporting date" whatever that is. In this case, the above SQL is barely altered. Just replace the last line with the appropriate constraint in the time dimension table for this special calendar event.

Question 3. Isn't this design very sensitive to backdated transactions being inserted into the fact table?

Answer 3. That is a good question. This fact table must be COMPLETE and ACCURATE. Every transaction against the account must appear in this table or else the running balance cannot be computed. Certainly a late arriving transaction record would require sweeping forward from the point of insertion in that account and incrementing all the balances and all the transaction sequence numbers. Note that we haven't explicitly used the transaction sequence number in this discussion, although something is needed in this design to reliably reconstruct the true sequence of transactions and to provide the basis of a key for the fact table (account_key + date_key + sequence number). I like the sequence number rather than a time-of-day stamp because differences between the sequence numbers are a valid measure of account activity.

I have a number of additional astounding insights about this design but you have to come to Maui to hear what they are. Just kidding. Write to me with comments.

Kimball Design Tip #13: When A Fact Table Can Be Used As A Dimension Table

By Ralph Kimball

Fact tables come in three main flavors. The grain of a fact table can be an individual transaction, where a fact table record represents an instant in time. Or, the grain can be a periodic snapshot, representing a predictable duration of time like a week or a month. Or finally, the grain can be an accumulating snapshot, representing the entire history of something up to the present. I discussed these three types in depth in an IE article, which you can find at www.intelligententerprise.com/993003/warehouse.shtml.

The first fact table type, the instantaneous transaction, may give us an opportunity to capture the description of something at an exact moment. Suppose that we have a series of transactions against the customer information in your bank account. In other words, an agent in the bank periodically makes changes to your name, address, phone number, customer classification, credit rating, risk rating, and other descriptors. The transaction grained fact table that captures these transactions might look like

Cust Info Transaction Date (FK)	<<== foreign key
Account (SK)	<<== surrogate key
Responsible Agent (FK)	<<== foreign key
Cust Info Transaction Type (FK)	<<== foreign key
Account Number	<<== bank's production "key"
Name	<<== text fact(s)
Address (several fields)	<<== text fact(s)
Phone Number	<<== text fact
Customer Classification	<<== text fact
Credit Rating	<<== non additive numeric fact
Risk Rating	<<== non additive numeric fact
.. other customer descriptors	

This is a typical design for a fact table where the "measurements" recorded by the customer information transactions are changes made to textual values, such as the name, address, and other textual fields listed above. Such a fact table blurs the distinction between a fact table and a dimension table because this fact table is filled with discrete textual values and non additive numeric values that cannot be summarized, but are instead the targets of end user constraints.

Three of the four keys to this fact table are simple foreign keys (FKs) connecting to conventional dimension tables. These include the transaction date, the responsible agent, and the name of the transaction itself. The production account number is not a data warehouse join key, but rather is the bank's constant identifier for this customer account.

The remaining key is the surrogate account key. In other words, it is simply a sequentially assigned number that uniquely identifies this transaction against this account. BUT, here is the subtle point that is the secret of this whole design. This account surrogate key therefore uniquely represents this snapshot of this account at the moment of the customer info transaction, and continues to accurately describe the account until the next customer info transaction occurs at some indeterminate time in the future.

So, to make a long story short, we can use the account surrogate key as if it were a typical Type 2 SCD (slowly changing dimension) key, and we can embed this key in any OTHER fact table describing account behavior. For example, suppose that we also are collecting conventional account transactions like deposits and withdrawals. We'll call these "balance transactions" to distinguish them from the customer information transactions. This second fact table could look like

Balance Transaction Date (FK)	<<== foreign key
Balance Transaction Time of Day (FK)	<<== foreign key
Account (SK)	<<== surrogate key
Location (FK)	<<== foreign key
Balance Transaction Type (FK)	<<== foreign key
Amount	<<== additive fact
Instant Balance	<<== semi-additive fact

When we make one of these balance transaction fact records, we carefully consult our account transaction/fact table and pick out the right surrogate key to use. Normally when we process today's records, we just use the most recent surrogate key for the account. This design then perfectly links every balance transaction to the right account profile described in our first fact table. Or is it a dimension table???

Well, I hope this has got you thinking. I wrote a DBMS magazine article on human resources databases that used a similar design approach. You can find it at http://dbmsmag.com/9802d05.html. See also the section "Time Stamping the Changes in a Large Dimension" in the Lifecycle Toolkit book, page 233. Send me your questions and comments and I'll devote the next Design Tip to your responses.

See you in Maui. We'll discuss this design as we watch the sun set over Molokai…

Kimball Design Tip #12: Accurate Counting With A Dimensional Supplement

By Warren Thornthwaite

In the previous design tip we talked about performing accurate counts within a dimension. We can add even more value to these counts when we introduce a separate table with additional attributes that intersects with the dimension table.

We recently loaded a simple example of this kind of "supplemental" table that maps zip code to Media Market Areas (MMA). Our Marketing folks were interested in seeing how our customers break out by MMA compared to the overall population. In other words, we want to know where are we getting strong geographical penetration, and where are we not doing as well. If this supplemental data proves valuable to the organization, we would go ahead and add it into the Customer dimension as an additional attribute. But first, we'll want to do some initial queries to make sure it is worth the effort.

To run these queries, we join the supplemental table to our customer table and do customer counts by MMA. However, we have to be careful because the two sets do not overlap 100%. There are some zip codes in the MMA table with no corresponding customers, and there are some customers whose zip codes have no corresponding MMA. An inner join will undercount both sides of the query. We can use the following two tables to illustrate this:

Media_Market_Areas

Zip	MMA
94025	SF-Oak-SJ
94303	SF-Oak-SJ
97112	Humboldt
98043	Humboldt
00142	Gloucester

Current_Customer

Customer_Key	Zip
27	94303
33	94025
47	24116
53	97112
55	94025

If we'd like to see how many Customers we have by MMA, an inner join gives us the following:

MMA	Count(Customer_Key)
Humboldt	1
SF-Oak-SJ	3

The inner join is an "equal" join. Since there is no MMA for zip 24116, the query undercounts our subscriber base giving us 4 customers when we really have 5. We also lose information on the other side of the join because the results don't tell us the MMAs where we have zero penetration (e.g., Gloucester). Re-writing the query with a full outer join gives us the
following:

MMA	Count(Customer_Key)
NULL	1
Gloucester	0
Humboldt	1
SF-Oak-SJ	3

Now we are counting all 5 of our customers, and we see we have no customers in Gloucester. We could use an IFNULL function on the character column to replace those NULLs with friendlier values, like 'MMA Unknown'. Note that what you count makes a big difference in your results. In our case, we counted Customer_Key. If we had count(*), we would have gotten a total of 7 items, since the * counts rows, and the full results set has 7 rows. If we had counted (MMA_to_Zipcode.Zip_Code), we would have returned a count of 6 because 94025 is counted twice.

You need to be cautious using outer joins because you can easily undermine the logic by putting a constraint on one of the participating tables. A constraint on Customer_Age < 25 would yield a report with exactly the same structure and headings but reduced counts. If the report is not explicitly labeled, it would be misleading.

We've discovered that combining the case statement with the sum function is a great trick to get counts of various subsets of the full results set from both sides of the outer join in a single pass. Using the data above, we could create a query that gives us total counts for all three areas of the data set. In the Select list you could write

```
Sum(case when Media_Market_Area.Zip IS NULL then 1 else 0 end) AS
Customer_Count_with_No_MMA,

Sum(case when count(customer_key) = 0 then 1 else 0 end) as
MMA_Count_with_No_Customers,

Sum(case when not(Media_Market_Area.Zip IS NULL or count(customer_key) = 0) then 1
else 0 end as Count_MMAs_with_Customers)
```

This gives you three columns: the count of customers with no MMAs, the count of MMAs with no customers, and a count where they match. Essentially, the constraints are built into the CASE statement, so they don't limit the results of the outer join.

www.kimballgroup.com Number 11, August 6, 2000

Kimball Design Tip #11: Accurate Counts Within A Dimension

By Warren Thornthwaite

Any dimension table with rich descriptive attributes often becomes the direct target of queries independent of any fact table. For example, we do various counts against our Customer table to answer questions like how many customers do we have by payment type, or state, or gender, or service plan, etc. on a daily basis. Simple counts against a static dimension are obvious, but our lives become more interesting when we try to retrieve these counts from a slowly changing dimension.

Counts on a Slowly Changing Dimension (SCD)
The problem is making sure we don't over count. In the customer SCD, we'll have multiple rows per customer, so a simple query that counts customer by state will end up over counting any customer who has had any changes (and therefore more than one record). One might be inclined to do a COUNT DISTINCT of the production customer key if it is available. The problem with this is if the attribute you are counting by has changed, like a move from one state to another, you may still over count because the customer key may be unique by state. We need some way to limit the SCD to one row per customer. We can do this for the most recent version of each customer record if there is a 'Current_Row' flag in the dimension. This approach will give us counts for the most current status of our customers. You can even make a standalone Customer Count table, view, or pre-defined query that only returns current rows for generating current counts.

Counts Over Time
Current counts are always useful, but the real trick is to get counts as of any particular date in history. Understanding how values change over time is one of the primary purposes of the data warehouse. Knowing we currently have 2,311 customers is good; being able to compare this with the count from a year ago is even better. You must have a slowly changing dimension to get these kinds of historical counts. For example, if you needed to know how many customers you had at the end of 1999, you could constrain the Row_Begin_Date <= '12/31/1999' AND Row_End_Date >= '12/31/1999'. This limits the results set to only those rows that were "active" on 12/31/1999. (The choice of comparison operators depends on how you set your begin and end dates.) We have assumed in this example that when a customer dimension record is changed, the row_end_date of the first record is one day less than the row_begin_date of the second record, and multiple changes within a single day are not allowed in the data.

If you really want to get fancy, instead of directly constraining date fields in the customer dimension table, you can use the full power of the Period dimension table to supply the target date, or even multiple target dates. To do this, you use the same comparison operators to create two unequal joins between the Date field of your Period table and the begin and end dates of the customer table, and constrain the Date field in the Period table equal to the target date. You can then include the Date field in the select list to see the date the counts apply to. To return counts for multiple dates in the same query, like the last day of the month for a year, remove the limit on the Date field of the Period table and add a constraint on the Month_End_Flag = 'y'. The SQL would look something like this:

```
SELECT Period.Date, Customer.State, COUNT(Customer.Customer_Key) FROM Customer,
Period WHERE Customer.Row_Begin_Date <= Period.Date AND Customer.Row_End_Date
```

```
        >= Period.Date AND Period.Month_End_Flag = 'y'
```

Note that this type of unequal join between the Period table and dimension table can pose a real challenge for the database engine with large dimensions. In our case, we have bitmapped indexes on both date fields, and we get pretty good performance.

Clearly, these types of queries are non-trivial to construct, and we would encourage you to shield most of your user community from query gymnastics of this nature. In fact, if requests for counts by certain attributes are common, it may make sense to create a summary fact table that includes a count for every existing combination of attributes by day. Then we are back to a simple dimensional model where each attribute is a dimension, and the fact table has at least one field that is the count of customers by attribute combination by day.

Kimball Design Tip #10: Is Your Data Correct?

By Ralph Kimball

A common problem in the data warehouse backroom is verifying that the data is correct. Is the warehouse an accurate image of the production system? Was this morning's download complete? Could some of the numbers be corrupted?

There is no single technique for validating a data load because there is so much variability in the sources of the data. If you are downloading an unchanged image of a production source, preserving the original granularity, then you can probably run a "flash report" on the production system with up-to-the minute totals, and then you can recapitulate the same report on the data warehouse. In this case, you "know" the answer beforehand and the two results should match to the last decimal place.

But it is more common not to have a known baseline of data. Maybe you are receiving the individual sales transactions from 600 retail stores every night. You can certainly perform a gross count on the number of stores reporting, but how can you apply some additional judgment to say whether the data is "probably correct"?

Using the 600 stores as an example, let's look at the sales totals for each department in each store each morning and ask if today's new numbers are "reasonable". We will decide that today's sales total is reasonable if it falls within 3 standard deviations of the mean of the previous sales totals for that department in that store. Arrghh. Statistics. But hang in there: it's not too bad. We chose 3 standard deviations because in a "normal" distribution, 99% of the values lie within 3 standard deviations above or below the mean.

I'll describe the process in words. After that I'll include some SQL. You can just read the words if you want to get the main drift.

In order to make this process run fast, you want to avoid looking at the complete time history of old data when you are calculating the standard deviation. You can avoid looking at the complete time history by keeping three accumulating numbers for each department in each store in a special table used only in the data validity pass. You need to keep the number of days being accumulated, the accumulating sum of each day's sales (by department by store) and the accumulating sum of the SQUARE of each day's sales (by department by store). These could be kept in a little stand alone accumulating department table. The grain of this table is department by store and the three numeric fields NUMBER_DAYS, SUM_SALES and SUM_SQUARE_SALES are all Type 1 attributes that are overwritten each day. You can update all three of these fields just by adding the next day's values to the ones already there. So if you have 600 stores and 20 departments in each store, this table has 12,000 rows but does not grow over time. The table also carries the store names and department names in each row.

Now, using this accumulating department table, you look at all 12,000 department totals in this morning's data load, and kick out this morning's numbers that are more than 3 standard deviations from the mean. You can chose to examine the individual unusual numbers if there aren't too many, or you can reject the entire load if you see more than a threshold number of sales totals more than 3 standard deviations from the mean.

If this morning's load passes muster, then you release the data to your end users, and then you update the accumulating department table to get ready for tomorrow's load.

Here's some untested SQL that may possibly work. Remember that the standard deviation is the square root of the variance. The variance is the sum of the squares of the differences between each of the historical data points and the mean of the data points, divided by N-1, where N is the number of days of data. Unfortunately, this formulation requires us to look at the entire time history of sales, which although is possible, makes the computation unattractive. But if we have been keeping track of SUM_SALES and SUM_SQUARE_SALES, we can write the variance as (1/(N-1))*(SUM_SQUARE_SALES - (1/N)*SUM_SALES*SUM_SALES). Check my algebra.

So if we abbreviate our variance formula with "VAR" then our data validity check looks like

```
SELECT s.storename, p.departmentname, sum(f.sales)
FROM fact f, store s, product p, time t, accumulatingdept a WHERE

(first, joins between tables…)
f.storekey = s.storekey and
f.productkey = p.productkey and
f.timekey = t.timekey and
s.storename = a.storename and
p.departmentname = a.departmentname and

(then, constrain the time to today to get the newly loaded data…) t.full_date = #July 13,
2000# and

(finally, invoke the standard deviation constraint…)
HAVING
ABS(sum(f.sales) - (1/a.N)*a.SUM_SALES) > 3*SQRT(a.VAR)
```

where we expand VAR as in the previous explanation and use the "a." prefix on N, SUM_SALES and SUM_SQUARE_SALES. We have assumed that departments are groupings of products and hence are available as a rollup in the product dimension.

Embellishments on this scheme could include running two queries: one for the sales MORE than three standard deviations above the mean, and another for sales LESS than three standard deviations below the mean. Maybe there is a different explanation for these two situations. This would also get rid of the ABS function if your SQL doesn't like this in the HAVING clause.

If you normally have significant daily fluctuations in sales (e.g., Monday and Tuesday are very slow compared to Saturday), then you could add a DAY_OF_WEEK to the accumulating department table and constrain to the appropriate day. In this way you don't mix the normal daily fluctuations into our standard deviation test.

Kimball Design Tip #9: Processing Slowly Changing Dimensions During Initial Data Load: A Pragmatic Compromise

By Lawrence Corr

Last month's tip clearly defined the type 2 slowly changing dimension technique and the proper use of surrogate keys. This month we tackle the vexing issue of handling slowly changing dimensions during the initial load of a new subject area within a data warehouse. This would occur when you have brought a new measurement (fact) source into an existing data warehouse. Dimensions like product, customer, and time are probably already defined and have a rich history reflecting many "slow changes".

First I'll describe the regular ETL (extract-transform-load) processing that would happen each night:

Dimensions are processed first, any new records from the production source are inserted into the dimension tables and are assigned the next surrogate key in sequence. Existing dimension records that have changed since the last load of the warehouse are detected and the nature of their change examined. An established slowly changing dimension policy will decide, based on which fields have changed, whether the current dimension record should be destructively updated and the old values lost (type 1) or that a new dimension record possessing the same natural ID should be created using the next surrogate key in sequence (type 2).

Following all these intricate comparisons and decisions on each dimension, it is time to face the bulk load of the facts. Here speed is of the essence. The natural IDs must be stripped off the fact records and replaced with the correct surrogate keys as rapidly as possible. Ideally we want to take advantage of the in-memory lookup processing offered by the majority of modern ETL tools. Small lookup tables for each dimension that translate natural IDs to surrogate keys are built from the dimension tables in the warehouse RDBMS using statements similar to the ones below:

```
Select customer_ID, max(customer_key) from customer
 group by customer_ID
```

or

```
Select customer_ID, customer_key from customer
Where current = 'Y'
```

By doing so, the lookup tables will contain the surrogate keys that match the new facts and will support rapid hash lookups.

However, this simple and efficient lookup technique will not be sufficient if we are loading a significant span of historical facts in this first load. For example, imagine the situation where we are initially loading two years worth of transactions and we are 'blessed' with having the same two years of customer master file history. The two may never have met on a report before but we wish to match them precisely in this new data mart and perfectly partition history.

The dimension lookup would have to contain the historic surrogate keys for each customer detail change during the two year time period. The simple hash lookup comparable to the SQL:

```
Select customer_key from customer_lookup CL
 where CL.customer_ID = factfile.customer_id
```

must be replaced with:

```
Select customer_key from customer_lookup CL
 where customer_ID = factfile.customer_id and factfile.transaction_date between
CL.effective_date and CL.end_date
```

The 'between date logic' required here will slow the processing of a several hundred million row fact load with ten such dimensions to a crawl. Added to this, you now have two very different fact load applications. If you ever get this initial load to run to completion, you have to throw it away and then build, test and maintain the simpler version that will incrementally load the fact table from now on.

While being something of a compromise, the answer to this problem has an appealing simplicity. Don't build the complex bulk job, only build the incremental job! This technique provides a far truer version of the past than simply attaching all historic facts to the current dimension records and vowing to slowly change from now on, which is often the strategy adopted when faced with the initial bulk load. Here's what I suggest you do:

Take the two-year load and break it into one hundred and four jobs each representing one week's worth of transactions and run them sequentially in rapid succession. Begin each job by loading only the dimensional history relevant for that week, i.e., where the dimension detail effective_dates are less than or equal to the minimum transaction date. While this means a little additional logic in the dimension maintenance stages, the fact load can run unaltered. It can do so because the simple lookup tables contain the maximum surrogate keys applicable for the load period. You have just run 104 incremental loads. The compromise is that some fact records will be attached to dimension records that are up to a week out of date. In most cases this is a small margin of error because the dimensions are changing slowly.

The techniques for maintaining dimensions and building lookup tables are described in more detail in Ralph's June 1998 Data Warehouse Architect article "Pipelining Your Surrogates" (www.ralphkimball.com/html/articles.html)

Kimball Design Tip #8:
Perfectly Partitioning History With The Type 2 Slowly Changing Dimension

By Ralph Kimball

The Type 2 slowly changing dimension (SCD) approach provides a different kind of partitioning. You could call this a logical partitioning of history. In the Type 2 approach, whenever we encounter a change in a dimension record, we issue a new record, and add it to the existing dimension table. A simple example of such a change is a revised product description, where something about the product, such as the packaging type, changes but the basic Stock Number (e.g., the bar code) does not change. As keepers of the data warehouse, we have taken a pledge to perfectly track history and so we must track both the new product description as well as the old.

The Type 2 SCD requires special treatment of the dimension key, the product key in our example. We must assign a generalized key because we can't use the same Stock Number as the key. This gives rise to the whole discussion of assigning anonymous surrogate keys, which has been discussed extensively elsewhere. See the articles www.dbmsmag.com/9805d05.html and www.dbmsmag.com/9806d05.html for the complete story on surrogate keys.

Stop and think for a moment about how you have been using the dimension key up to the point where you make the new dimension record described above. Before today, you have been using the "old" surrogate key whenever you have created a fact table record describing some product activity.

Today, two things happen. First, we assume that the changed packaging type goes into effect with the new fact table data that we receive today. Second, this means that after we create the new dimension record with its new surrogate key, then we use that surrogate key with all of today's new fact records.

WE DON'T GO BACK TO PREVIOUS FACT RECORDS TO CHANGE THE PRODUCT KEY.

The old product dimension record still points correctly to all the previous historical data, and the new product dimension record will now point to today's records and subsequent records, until we are forced to make another Type 2 change.

This is what we mean when we say that the Type 2 SCD perfectly partitions history. If you visualize this, then you really understand this design technique.

Notice that when you constrain on something in the dimension like the product name, which is unaffected by the change in the packaging type attribute, then you automatically pick up both the old and new dimension records, and you automatically join to all of the product history in the fact table. It is only when you constrain or group on the packaging type attribute that SQL smoothly divides the history into the two regimes.

This discussion is the heart of the Type 2 approach. The Type 2 approach can be augmented and made even more powerful by carefully placing time stamps in the dimension, but these time stamps are extra "goodies" that are not needed for the basic partitioning of history. See the article www.dbmsmag.com/9802d05.html for one application of a time stamped SCD.

Kimball Design Tip #7: Getting Your Data Warehouse Back On Track

By Margy Ross and Bob Becker

During the past year, we've repeatedly observed a pattern with maturing data warehouses. Despite significant effort and investment, some data warehouses have fallen off course. Project teams (or their user communities) are dissatisfied with the warehouse deliverables - the data's too confusing, it's not consistent, queries are too slow, etc. Teams have devoured the data warehousing best sellers and periodicals, but are still unsure how to right the situation (short of jumping ship and finding new employment).

If this situation sounds familiar, take the following self-check test to determine if the four leading culprits are undermining your data warehouse. Consider each question carefully to honestly critique your warehouse situation. In terms of corrective action, we recommend tackling these fundamental concerns in sequential order, if possible.

1. **Have you proactively gathered requirements for each iteration of the data warehouse from business users and aligned the data warehouse implementation with their top priorities?**

   This is the most prevalent problem for aging data warehouses. Somewhere along the line, perhaps while overly focused on data or technology, the project lost sight of the real goal to serve the information needs of business users.

   As a project team, you must always focus on the users' gain. If the team's activities don't provide benefit to the business users, the data warehouse will continue to drift. If you're not actively engaged in implementing solutions to support users' key business requirements and priorities, why not? Revisit your plans to determine and then focus on delivering to the users' most critical needs.

2. **Have you developed a Data Warehouse Bus Matrix?**

   The Matrix is one of the data warehouse team's most powerful tools. Use it to clarify your thinking, communicate critical points of conformance, establish the overall data roadmap, and assess your current progress against the long-term plan. If you're unfamiliar with the Matrix, refer to Ralph's 12/7/99 Intelligent Enterprise article at http://www.intelligententerprise.com/990712/webhouse.shtml.

3. **Is management committed to using standardized conformed dimensions?**

   Conformed dimensions are absolutely critical to the long-term viability of a data warehouse. We find many warehouse teams are reluctant to take on the socio-political challenges of defining conformed dimensions. In all honesty, it's extremely difficult for a data warehouse team to establish and develop conformed dimensions on its own. Yet the team can't ignore the issue and hope it will resolve itself. You'll need management support for conformed dimensions to help navigate the organizational difficulties inherent in the effort.

4. **Have you provided atomic data in dimensional models to users?**

Data shortcomings are often at the root of data warehouse course adjustments - it's either the wrong data, inappropriately structured or prematurely summarized. Focusing on business requirements will help determine the right data; then the key is to deliver the most atomic data dimensionally. Unfortunately, it's tough to gracefully migrate from data chaos to this nirvana. In most cases, it's best to bite the bullet and redeploy. Teams sometimes resort to the seemingly less drastic approach of sourcing from the current quagmire, however, the costs are inevitably higher in the long run. Often the granularity of the existing data precludes this alternative due to premature summarization.

In summary, if your data warehouse has fall off course, it won't magically right itself. You'll need to revisit the basic tenets of data warehousing. Listen to users to determine your target destination, get a map, establish a route, and then follow the rules of the road to get your data warehouse back on track.

www.kimballgroup.com Number 6 April 10, 2000

Kimball Design Tip #6: Showing The Correlation Between Dimensions

By Ralph Kimball

One of the questions I get asked frequently is "how can I represent the correlation between two dimensions without going through the fact table?" Often the designer follows up with the question "can I create a little joiner table with just the two dimension keys and then connect THAT table to the fact table?"

Of course, in a classic dimensional model, we only have two choices. Either the dimensions are modeled independently and the two dimension keys occur together ONLY in the fact table, or else the two dimensions are combined into a single super-dimension with a single key. So when does the designer choose separate dimensions and when does the designer combine the dimensions?

To be more concrete, let us imagine that the two dimensions are Product and Market in a retail setting. Suppose that the fact table of interest records actual sales of Products in the various Markets over Time. Our desire to represent the correlation between the Product and Market dimensions is based on the suspicion that "Products are highly correlated with Markets in our business". This sentence is the key to the whole design question.

If Products are extremely correlated with Markets, then there may be a 1-to-1 or a many-to-1 relationship between Products and Markets. In this case, combining the two dimensions makes eminent sense. The combined dimension is only as big as the larger of the two dimensions. Browsing (looking at the combinations of values) within the combined dimension would be useful and fast. Interesting patterns would be apparent.

But rarely do Product and Market have such a nice relationship. At least three factors intrude that eventually make us pull these two dimensions apart.

1. The 1-to-1 or many-to-1 relationship may not literally be true. We may have to admit that the relationship is really many-to-many. When most Products are sold in most Markets, it becomes obvious that we need two dimensions because otherwise our combined dimension becomes huge and starts to look like a Cartesian product of the original dimensions. Browsing doesn't yield much insight.

2. If the relationship between Product and Market varies over Time, or under the influence of a fourth dimension like Promotion, then we have to admit that the combined dimension is in reality some kind of fact table itself!

3. There are more relationships between Product and Market than simply the retail sales. Each business process involving Product and Market will turn out to generate its own fact table. Good candidates include Promotion Coverage, Advertising, Distribution, and Manufacturing. Creating a combined Product-Market dimension exclusively around retail sales would make some of these other processes impossible to express.

The point of this Design Tip is to encourage you to visualize the relationship between the entities you choose as dimensions. When entities have a fixed, time-invariant strongly correlated relationship, they should be modeled as a single dimension. In most other cases, your design will be simpler and smaller when you separate the entities into two dimensions.

Don't avoid the fact table! Fact tables are incredibly efficient. They contain only dimension keys and measurements. They contain only the combinations of dimensions that occur in a particular process. So when you want to represent the correlation between dimensions, remember that the fact table was created exactly for this purpose.

Kimball Design Tip #5: Surrogate Keys For The Time Dimension

By Ralph Kimball

I get several data warehouse design questions each day. Because so many of them are serious and interesting, I try to answer them. But if it appears to be an assignment from a college professor, I politely decline!

Here's the question:

A consultant we had recently proposed a time dimension that looks rather different from the ones you design.

His time dimension structure was:

Key	varchar2 (8)
StartDate	date or date/time
EndDate	date or date/time

Sample data was:

Key	StartDate	EndDate
xmas99	25Nov99	06Jan00
1qtr99	01Jan99	31Mar99
newyrsdy	01Jan00	01Jan00
01Jan00	01Jan00	01Jan00

What is your view on a time dimension with this structure? For what type of scenario / business would you find this a good, viable alternative?

Here's how I responded:

I don't think I like that time dimension very much if indeed it is a dimension.

I expect a time dimension to describe the time context of a measurement expressed in a fact table. In database terms there needs to be a time-valued foreign key in every fact table record that points out to a specific record in the time dimension.

It is very important for applications simplicity that the fact table be of uniform granuarity. In other words, all the records in the fact table should represent measurements take at a daily level, a weekly level, or a monthly level, for instance.

Your proposal has time dimension records of varying grain and it appears that they overlap. If you have a measurement record that occurs on a particular date, and these "time dimension" records overlap, which one of the records do you choose for a particular fact table record?

In a uniform grain fact table you can use the associated time dimension to constrain in a simple way on many different spans of time. A time dimension table with entries for every discrete day is very flexible because in this table you can simultaneously represent all the useful time groupings you can think of.

A typical daily grain time table with an U.S. perspective could have the following structure:

> Time_Key (surrogate key; simple integers assigned from 0 to N)
> Time_Type (Normal, Not_Applicable, Hasnt_Happened_Yet, Corrupted)
> SQL_Time_Stamp (8 byte date stamp for Type=Normal, Null otherwise)
> Day_Number_in_Month (1..31)
> Day_Number_in_Year (1..366)
> Day_Number_in_Epoch (an integer, positive or negative)
> Week_Number_in_Year (1..53)
> Week_Number_in_Epoch (an integer, positive or negative)
> Month_Number_in_Year (1..12)
> Month_Number_in_Epoch (an integer, positive or negative)
> Month_Name (January, ..., December, can be derived from
> SQL_Time_Stamp) Year (can be derived from
> SQL_Time_Stamp) Quarter (1Q, ..., 4Q) Half (1H, 2H)
> Fiscal_Period (names or numbers, depending on your finance department)
> Civil_Holiday (New Years Day, July 4, ..., Thanksgiving, Christmas) Workday (Y, N)
> Weekday (Y, N)
> Selling_Season (Winter Clearance, Back to School, Christmas Season) Disaster (Hurricane Hugo, ..., Northridge Earthquake)

In this daily time table you make a record for each day of the year and you populate each field (shown above) with the relevant values for that day. All the special navigation fields like Fiscal_period and Selling_season let you define arbitrary spans of time for these special items. For example, you can constrain Selling_Season = "Back to School" and would automatically get all the days from August 15 to September 10.

In your proposed design you show the keys of the time dimension table with values like "xmas99" and "1qtr99". These are smart keys. Smart keys are dangerous in a data warehouse dimension table for several reasons. The generation of these keys is held hostage to the rules for their syntax. It is tempting to write applications and user interfaces that would make these keys visible to someone. If there is a "1qtr99" are you guaranteeing that there is a "2qtr99"? And what do you do when you need one of the Not-Applicable situations for a date stamp?

We have discussed the assignment of surrogate keys in other forums, but we really mean what we say here: the keys in the time dimension must have no applications significance. They are just integers and you can't compute with them.

Kimball Design Tip #5: Follow-up To Surrogate Keys For The Time Dimension

Posted May 21, 2000

I would like to share with you some useful comments I received about Design Tip #5, where I outlined a preferred design for a time dimension, and said that the primary key for this dimension should just be a simple integer, not a true date stamp.

Several people, who otherwise agreed with this approach, said that nevertheless it can be very useful for the surrogate keys (1,2,3, ..., N) to be assigned in the correct order corresponding to the dates in the associated time dimension records. This allows any fact table to be physically partitioned on the basis of the surrogate time key. Physically partitioning a large fact table on the basis of time is a very natural approach in any case, because it allows old data to be removed gracefully, and it allows the newest data to be re-indexed and augmented without disturbing the rest of the fact table, if you are using the table partitioning features of your database.

Also, since I have occasionally pointed out that Microsoft SQL Server is the only high end serious database that still does not support table partitioning, I was pleased to see that table partitioning is now a feature of SQL 2000.

www.kimballgroup.com Number 4 February 21, 2000

Kimball Design Tip #4:
Super Fast Change Management Of Complex Customer Dimensions

By Ralph Kimball

Many data warehouse designers have to deal with a difficult customer dimension that is both wide and deep. A customer dimension may have 100 or more descriptive attributes, and may have millions of rows. Sometimes the "customers" are health insurance policy claimants, and other times they are owners of motor vehicles. But the design issues are the same.

Often in these situations the data warehouse receives a complete updated copy of the customer dimension as frequently as once per day. Of course, it would be wonderful if only the "deltas" (the changed records) were delivered to the data warehouse, but more typically the data warehouse has to find the changed records by carefully searching the whole file. This comparison step of each field in each record between yesterday's version and today's version is messy and slow.

Here's a technique that accomplishes this comparison step at blinding speeds and has the added bonus of making your ETL program simpler. The technique relies on a simple CRC code that is computed for each record (not each field) in the incoming customer file. More on CRC's in a moment. Here are the processing steps:

1. Read each record of today's new customer file and compute that record's CRC code.
2. Compare this record's CRC code with the same record's CRC code from yesterday's run, which you saved. You will need to match on the source system's native key (customer ID) to make sure you are comparing the right records.
3. If the CRC codes are the same, you can be sure that the entire 100 fields of the two records exactly match. YOU DON'T HAVE TO CHECK EACH FIELD.
4. If the CRC codes differ, you can immediately create a new surrogate customer key and place the updated record in the data warehouse customer dimension. This is a Type 2 slowly changing dimension (SCD). Or, a more elaborate version could search the 100 fields one by one in order to decide what to do. Maybe some of the fields trigger an overwrite of the data warehouse dimension record, which is a Type 1 SCD.

If you have never heard of a CRC code, don't despair. Your ETL programmer knows what it is. CRC stands for Cyclic Redundancy Checksum and it is a mathematical technique for creating a unique code for every distinguishable input. The CRC code can be implemented in Basic or C. Most introductory computer science textbooks describe the CRC algorithm. Also look on the Google search engine (www.google.com) for "CRC code" or "checksum utility".

I would like to hear about any interesting techniques you have developed like this for your dimensional data warehouses. Send me an e-mail describing your technique.

KIMBALL GROUP KIMBALL UNIVERSITY

www.kimballgroup.com Number 3 January 30, 2000

Kimball Design Tip #3: Focus On Business Process, Not Business Departments!

By Margy Ross

One of the most prevalent fallacies in our industry is that data marts are defined by business department. We've seen countless data warehouse architecture diagrams with boxes labeled "Marketing Data Mart," "Sales Data Mart," and "Finance Data Mart." After reviewing business requirements from these departments, you'd inevitably learn that all three organizations want the same core information, such as orders data. Rather than constructing a Marketing data mart that includes orders and a Sales data mart with orders, etc., you should build a single detailed Orders data mart which multiple departments access.

Focusing on business processes, rather than business departments allows you to more economically deliver consistent information throughout the organization. If you establish departmentally-bound marts, you'll duplicate data. Regardless if the source is an operational system or centralized data warehouse, multiple data flows into the marts invariably result in data inconsistencies. The best way to ensure consistency is to publish the data once. A single publishing run also reduces the extract-transform-load development effort, on-going data management burden, and disk storage requirements.

We understand that it can be tricky to build a process-centric data mart given the usual departmental funding. You can promote the process concept by scrutinizing the unnecessary expense associated with implementing and maintaining the same (or nearly the same) large fact tables in multiple data marts. Even if organizational walls exist, management typically responds to savings opportunities.

So how do you go about identifying the key business processes in your organization? The first step is to listen to your business users. The performance metrics that they clamor to analyze are collected or generated by a business process. As you're gathering requirements, you should also investigate key operational source systems. In fact, it's easiest to begin by defining data marts in terms of source systems. After you've identified the data marts based on individual business processes and source systems, then you can focus on marts that integrate data across processes, such as a vendor supply chain, or all the inputs to customer profitability or customer satisfaction. We recommend that you tackle these more complex (albeit highly useful) multi-process marts as a secondary phase.

Of course, it will come as no surprise to hear that you must use conformed dimensions across the data marts. Likewise, we strongly suggest drafting a Data Warehouse Bus matrix up-front to establish and communicate your overall mart strategy. Just don't let the rows of your matrix read "Marketing," "Sales," and "Finance."

Kimball Design Tip #2: Multiple Time Stamps

By Ralph Kimball

The most frequent question I get in my classes and e-mails is how to handle multiple time stamps on a fact table record. Although the correct and immediate answer is "make each time stamp a Time dimension", it is worth describing this approach carefully because it nicely illustrates a whole set of modern data warehouse design techniques, which I will emphasize in CAPS.

First, choose the **FUNDAMENTAL GRAIN** of your fact table. Every fact table I have ever designed has been a transaction grain, a periodic snapshot grain, or an accumulating snapshot grain. See the article www.intelligententerprise.com/993003/warehouse.shtml for more on fundamental grains. When you understand the fundamental grain, you will be able to judge what the meaning and relevance of multiple time stamps is to that grain. Multiple time stamps arise most frequently when the fact record is an accumulating snapshot, where the fact record represents the complete history of an order line, for example. The multiple time stamps may represent
1) original order date
2) requested delivery date
3) actual ship date
4) actual delivery date, and
1. 5) return date.

Second, for the above order line example, create five **ROLES** for a single underlying Time dimension. See the article www.dbmsmag.com/9708d05.html for more on roles. This means five fields in the fact table, where each field is a good foreign key linking to a dimension. The single underlying Time dimension is "exposed" to the fact table through five views, which makes each instance of the Time dimension semantically independent, as they must be.

Third, make sure that the actual foreign keys in the fact table are proper **SURROGATE KEYS**. In other words, they aren't literal SQL date/time stamps, but rather are simple anonymous integers. Resist all urges to put meaning or ordering in these keys! See the articles www.dbmsmag.com/9805d05.html and www.dbmsmag.com/9806d05.html for more on surrogate keys. If you think carefully about our order line example, you will have to agree that some order line records must contain "unknown" or "hasn't happened yet" time stamps. This is one of the classic reasons for using surrogate keys.

If your time stamps are accurate to the minute or second, then you need to split the calendar day off from the time of day and make them separate dimensions. We'll discuss variations on these time stamp designs in a future design tip.

www.kimballgroup.com Number 1 December 15, 1999

Kimball Design Tip #1: Guidelines For An Expressive Clickstream Data Mart

By Ralph Kimball

The clickstream is the record of page events collected by a web server. In the raw data, there is one record for every click made by a visitor that the web server can detect. The clickstream contains unprecedented detail about every "gesture" made by a visitor to the web site.

The clickstream data source is huge. Even moderately busy commercial web servers may generate 100 million page event records each day. We must reduce the volume of data to manageable proportions for our most important analyses. In this design tip we will seek a way to NOT crawl through 100 million records, while still keeping a useful level of detail to analyze web visitor behavior.

In the raw clickstream data there are hints of a number of interesting dimensions, including Calendar Day, Time of Day, Visitor, Page Object Requested, Referring Context (what prior page contained the link clicked), and Action (basically either Get Object from the web server, or Post Object to the web server).

The recommended grain of the clickstream behavior fact table is

> One Fact Record = One Visitor Session.

If an average session consists of 20 page events, then the number of fact records in our example is reduced to 5 million per day, which is comparable to the experience of medium sized retailer data warehouses.

The recommended dimensionality of this fact table is

* Web Server Day (calendar date as recorded by the web server)
* Web Server Time (seconds since midnight recorded by the web server marking the start of the session)
* Visitor Day (calendar date as experienced by the visitor)
* Visitor Time (seconds since midnight recorded by the visitor marking the start of the session)
* Visitor (generic name "Visitor" for anonymous visitors, unique system generated name for unregistered visitors who have accepted a cookie, and true name for registered visitors)
* Starting Page (identity of first page in session: the page that attracted the visitor from elsewhere on the web)
* Ending Page (identity of the last page in session: maybe this is a session killer)
* Referring Context (the URL of the page the visitor came from, if available)
* Session Diagnosis (a simple descriptive tag indicating what kind of session this was)

The recommended numeric facts in this design are:

* Number pages visited
* Total dwell time (something of an estimate, because we can't account for the visitor's real activities)

This design can be a very powerful base from which to evaluate visitor behavior on a web site. The most important dimension is the Session Diagnosis dimension. You must have a sophisticated back room ETL (Extract-Transform-Load) process to create good session diagnoses out of the detailed page event sequences.

For further reading on these subjects, download the following free article from the Intelligent Enterprise magazine archive: www.intelligententerprise.com/990501/warehouse.shtml

Another article will appear in January 2000 in Intelligent Enterprise discussing the Page and Session Diagnosis dimensions in more detail.

Follow-up to Design Tip #1: Guidelines For An Expressive Clickstream Data Mart

January 7, 2000

I had a number of interesting comments from the first design tip, which recommended a dimensional design for clickstream data. Several people asked me why the design tip recommended a fact table grain of a record = a complete session, when the January 5, 1999 Intelligent Enterprise article found at www.intelligententerprise.com/990501/warehouse.shtml recommended a grain of the individual page event. These people asked me if I had changed my mind.

No I have not changed my mind, but I understand the problem better. There are at least three useful grains at which to represent clickstream data:

1) Fact record = individual page event. This level, described in the IE article, can give detailed maps and trajectories of every web visit, if you keep every record. But for very busy sites, there is too much data. You will spend all your time and money collecting and storing the data rather than analyzing it. Several people told me that statistical sampling techniques with as little as 1% of the total volume of data could very usefully depict site usage patterns that would lead to important decisions about the use of the web site, even if all their individual visitors were not present in the data. I like this suggestion very much. You will probably need a professional statistician to help you choose a robust small sample of your data.

2) Fact record = each complete visitor session. This is the level I discussed in the first design tip. In this case, you can realistically strive for complete coverage of all visitors, although you are not seeing their complete maps and trajectories taken through your web site. But you can do extensive demographic and web site effectiveness analyses. Remember that you have the Entry Page, the Exit Page, and the Session Diagnosis dimensions.

3) Fact record = web page by calendar day. This grain is one of several similar rollup levels that can be useful for seeing the total pattern of hits in various parts of your web site. Clearly the advantage of this grain is the sharply reduced size of the data, but like any aggregate fact table, you have suppressed several of the behavioral dimensions like Visitor and Session Diagnosis.

I guess the answer to the grain question is that eventually you want them all. Just like most of the other data warehouses we build.