**Design Tip #145 Time Stamping Accumulating Snapshot Fact Tables**

May 1, 2012
© Kimball Group. All rights reserved.

[Print this Article](http://www.kimballgroup.com/2012/05/01/design-tip-145-time-stamping-accumulating-snapshot-fact-tables/print/)

In Design Tip #140, I discussed the challenges of designing dimensional schemas for processes of indeterminate length such as a sales pipeline or insurance claims processing. We concluded they are best represented as accumulating snapshot fact tables characterized by one row per pipeline occurrence where each row is updated multiple times over its lifetime. However, because each row is updated, we have an imperfect recording of history. The accumulating snapshot does a great job of telling us the pipeline’s current state, but it glosses over the intermediate states. For example, a claim may move in and out of states multiple times: opened, denied, protested, re-opened, re-closed. The accumulating snapshot is hugely valuable, but there are several things that it cannot do:

* It can’t tell us the details of when and why the claim looped through states multiple times.
* We can’t recreate our “book of business” at any arbitrary date in the past.

To solve both of these problems, we’ll need two fact tables. A transaction fact table captures the details of individual state changes. Then we’ll add effective and expiration dates to the accumulating snapshot fact table to capture its history.

The transaction fact table is straightforward. As described in The Data Warehouse Toolkit, Second Edition (p. 134-135) and the cornerstone article, Fundamental Grains, we often pair the accumulating snapshot fact table with a transaction fact table that contains a row for each state change. Where the accumulating snapshot has one row per pipeline process such as a claim, the transaction fact table has one row per event. Depending on your source systems, it’s common to build the transaction fact table first, and derive the accumulating snapshot from it.

Now let’s turn our attention to the time stamped accumulating snapshot fact table. First of all, not everyone needs to bother with retaining these time stamped snapshots. For most organizations, a standard accumulating snapshot representing the current state of the pipeline, combined with the transaction fact table to show the event details, is ample. However, we’ve worked with several organizations that need to understand the evolution of a pipeline. While it’s technically possible to do that from the transaction data, it’s not child’s play.

One solution to the historical pipeline tracking requirement is to combine the accumulating snapshot with a periodic snapshot: snap a picture of the pipeline at a regular interval. This brute force method is overkill for pipelines that are relatively long in overall duration, but change infrequently. What works best in this case is to add effective and expiration change tracking to the accumulating snapshot.
Here’s how it works:

* Design a standard accumulating snapshot fact table.
* Instead of updating each row as it changes state, add a new row. Our recent designs have been at the daily grain: add a new row to the fact table any day in which something about that pipeline (e.g., claim, sales process, or drug adverse reaction) has changed.
* You need some additional metadata columns, similar to a type 2 dimension:
	+ snapshot start date: the date this row became effective.
	+ snapshot end date: the date this row expired, updated when a new row is added.
	+ snapshot current flag: updated when we add a new row for this pipeline occurrence.

Most users are only interested in the current view, i.e., a standard accumulating snapshot. You can meet their needs by defining a view (probably an indexed or materialized view) that filters the historical snapshot rows based on snapshot current flag. Alternatively, you may choose to instantiate a physical table of current rows at the end of each day’s ETL. The minority of users and reports who need to look at the pipeline as of any arbitrary date in the past can do so easily by filtering on the snapshot start and end
dates.

The time stamped accumulating snapshot fact table is slightly more complicated to maintain than a standard accumulating snapshot, but the logic is similar. Where the accumulating snapshot will update a row, the time stamped snapshot updates the row formerly-known-as-current and inserts a new row. The big difference between the standard and time stamped accumulating snapshots is the fact table row count. If an average claim is changed on twenty days during its life, the time stamped snapshot will be twenty times bigger than the standard accumulating snapshot. Take a look at your data and your business’s requirements to see if it makes sense for you. In our recent designs, we’ve been pleasantly surprised by how efficient this design is. Although a few problematic pipeline occurrences were changed hundreds of times, the vast majority were handled and closed with a modest number of changes.

# Design Tip #140 Is it a Dimension, a Fact, or Both?

November 1, 2011
© Kimball Group. All rights reserved.

[Print this Article](http://www.kimballgroup.com/2011/11/01/design-tip-140-is-it-a-dimension-a-fact-or-both/print/)

For must subject areas, it’s pretty easy to identify the major dimensions: Product, Customer Account, Student, Employee, and Organization are all easily understood as descriptive dimensions. A store’s sales, a telecommunication company’s phone calls, and a college’s course registrations are all clearly facts.

However, for some subject areas, it can be challenging – especially for the new dimensional modeler – to identify whether an entity is a dimension or a fact. For example, an insurance company’s claims processing unit wants to analyze and report on their open claims. “Claim” feels like a dimension, but at the same time, it can behave like a fact table. A similar situation arises with software companies with extended sales cycles: is the sales opportunity a dimension, a fact, or both?

In most cases, the design puzzle is solved by recognizing that the business event you’re trying to represent in the fact table is actually a long-lived process or lifecycle. Often, the business users are most interested in seeing the current state of the process. A table with one row per process – one row per claim or sales opportunity, for example – sounds like a dimension table. But if you distinguish between the entity (claim or sales opportunity) and the process (claim settlement or sales pipeline), it becomes clearer. We need a fact table to measure the process. And many dimension tables to describe the attributes of the entity measured in that process.

This type of schema is implemented as an accumulating snapshot. The accumulating snapshot is less common than transactional and periodic snapshot fact tables. The grain of this type of fact table is one row per process; it has many roles of the date dimension; and the fact table rows are updated multiple times over the life of the process (hence the name accumulating snapshot). You can read more about accumulating snapshot fact tables in The Data Warehouse Toolkit, pages 128-134.

Many of the core dimensions of an accumulating snapshot schema are easy to identify, but there are some challenges to these designs. Long-lived processes tend to have a lot of little flags and codes from the source system that signal various statuses and conditions in the process. These are great candidates for junk dimensions. Expect your accumulating snapshot schema to have several junk dimensions.

I try to avoid creating a dimension with the same number of rows as the accumulating snapshot fact table. This can happen if you have not separated your junk dimensions into logically correlated groupings. In the earlier examples, I also resist designing a Claim or Sales Opportunity text description dimension for as long as I can. Realistically, though, there is usually some bit of detail that the business users absolutely need to see, such as the accident report or description of the sales opportunity. Although such a dimension may be huge, it is likely to be accessed by the business users only after other dimensions have been tightly constrained since there may be no predictable structure to the text entries.

# Design Tip #130 Accumulating Snapshots for Complex Workflows

December 1, 2010
© Kimball Group. All rights reserved.

[Print this Article](http://www.kimballgroup.com/2010/12/01/design-tip-130-accumulating-snapshots-for-complex-workflows/print/)

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.

**Extreme Status Tracking For Real Time Customer Analysis**

[Print this Article](http://www.kimballgroup.com/2010/06/21/extreme-status-tracking-for-real-time-customer-analysis/print/)

We live in a world of extreme status tracking, where our customer-facing processes are capable of producing continuous updates on the transactions, locations, online gestures, and even the heartbeats of customers. Marketing folks and operational folks love this data because real-time decisions can be made to communicate with the customer. They expect these communications to be driven by a hybrid combination of traditional data warehouse history and up-to-the-second status tracking. Typical communications decisions include whether to recommend a product or service, or judge the legitimacy of a support request, or contact the customer with a warning.

As designers of integrated enterprise data warehouses (EDWs) with many customer-facing processes, we must deal with a variety of source operational applications that provide status indicators or data-mining-based behavioral scores we would like to have as part of the overall customer profile. These indicators and scores can be generated frequently, maybe even many times per day; we want a complete history that may stretch back months or even years. Though these rapidly changing status indicators and behavior scores are logically part of a single customer dimension, it is impractical to embed these attributes in a Type 2 slowly changing dimension. Remember that Type 2 perfectly captures history, and requires you to issue a new customer record each time any attribute in the dimension changes. Kimball Group has have long pointed out this practical conflict by calling this situation a “rapidly changing monster dimension.” The solution is to reduce the pressure on the primary customer dimension by spawning one or more “mini-dimensions” that contain the rapidly changing status or behavioral attributes. We have talked about such mini-dimensions for at least a decade.

In our real-time, extreme status tracking world, we can refine the tried-and-true mini-dimension design by adding the following requirements. We want a “customer status fact table” that is…

* a single source that exposes the complete, unbroken time series of all changes to customer descriptions, behavior, and status;
* minutely time-stamped to the second or even the millisecond for all such changes;
* scalable, to allow new transaction types, new behavior tags, and new status types to be added constantly, and scalable to allow a growing list of millions of customers each with a history of thousands of status changes;
* accessible, to allow fetching the current, complete description of a customer and then quickly exposing that customer’s extended history of transactions, behavior and status; and
* usable as the master source of customer status for all fact tables in the EDW.

Our recommended design is the Customer Status Fact table approach shown in the figure below.



The Customer Status Fact table records every change to customer descriptions, behavior tags, and status descriptions for every customer. The transaction date dimension is the calendar date of the change and provides access to the calendar machinery that lets an application report or constrain on complex calendar attributes such as holidays, fiscal periods, day numbers, and week numbers.

The customer dimension contains relatively stable descriptors of customers, such as name, address, customer type, and date of first contact. Some of the attributes in this dimension will be Type 2 SCD (slowly changing dimension) attributes that will add new records to this dimension when they change, but the very rapidly changing behavior and status attributes have been removed to mini-dimensions. This is the classic response to a rapidly changing monster dimension. The Most Recent Flag is a special Type 1 field that is set to True only for the current valid customer record. All prior records for a given customer have this field set to False.

The customer durable key is what we normally designate as the natural key, but we call it durable to emphasize that the EDW must guarantee that it never changes, even if the source system has a special business rule that can cause it to change (such as an employee number that is re-assigned if the employee resigns and then is rehired). The durable key can be administered as a meaningless, sequentially assigned integer surrogate key in those cases where more than one source system provides conflicting or poorly administered natural keys. The point of the durable key is for the EDW to get control of the customer keys once and for all.

The customer surrogate key is definitely a standard surrogate key, sequentially assigned in the EDW back room every time a new customer record is needed, either because a new customer is being loaded or because an existing customer undergoes a Type 2 SCD change.

The double-dashed join lines shown in the figure are a key aspect of extreme status processing. When a requesting application sets the most recent flag to True, only the current profiles are seen. The customer surrogate key allows joining to the status fact table to grab the precise *current* behavior tags and status indicators. In a real-time environment, this is the first step in determining how to respond to a customer. But the customer durable key can then be used as an alternate join path to instantly expose the complete history of the customer we have just selected. In a real-time environment, this is the second step in dealing with the customer. We can see all the prior behavior tags and status indicators. We can compute counts and time spans from the customer status fact table.

The behavior dimension can be modeled in two ways. The simpler design is a wide dimension with a separate column for each behavior tag type. Perhaps these behavior tags are assigned by data mining applications that monitor the customer’s behavior. If the number of behavior tag types is small (less than 100), this design works very well because query and report applications can discover and use the types at run time. New behavior tag types (and thus new columns in the behavior dimension) can be added occasionally without invalidating existing analysis applications.

A more complex behavior dimension design is needed when a very large and messy set of behavior descriptors is available. Perhaps you have access to a number of demographic data sources covering complicated overlapping subsets of your customer base. Or perhaps you have account application data containing financial asset information that is very interesting but can be described in many ways. In this case, you will need a dimensional bridge table. Kimball Group has described dimensional bridge table designs in previous articles. Search for “bridge tables” (in quotes) at [www.kimballgroup.com](http://www.informationweek.com/news/software/info_management/225700892?pgno=3).

The status dimension is similar to the behavior dimension but can probably always be a wide dimension with a separate column for each status type, simply because this dimension is more under your internal control than the behavior dimension.

The transaction dimension describes what provoked the creation of the new record in the customer status fact table. Transactions can run the gamut from conventional purchase transactions all the way to changes in any of the customer-oriented dimensions, including customer, behavior and status. The transaction dimension can also contain special priority or warning attributes that alert applications to highly significant changes somewhere in the overall customer profile.

The begin and end effective date/times are ultra-precise, full-time stamps for when the current transaction became effective and when the next transaction became effective (superseding the current one). Kimball Group has given a lot of thought to these ultra-precise time stamps and we recommend the following design:

* The grain of the time stamps should be as precise as your DBMS allows, at least down to the individual second. Some day in the future, you may care about time stamping some behavioral change in such a precise way.
* The end effective time stamp should be exactly equal to the begin time stamp of the next (superseding) transaction, not “one tick” less. You need to have a perfect unbroken set of records describing your customer without any possibility of miniscule gaps because of your choice of a “tick”.
* In order to find a customer profile at a specific point in time, you won’t be able to use BETWEEN syntax because of the preceding point. You will need something like:

**#Nov 2, 2009: 6:56:00# >= BeginEffDateTime and #Nov 2, 2009: 6:56:00# < EndEffDateTime**

as your constraint, where Nov 2, 2009, 6:56am is the desired point in time.

The customer status fact table is the master source for the complete customer profile, gathering together standard customer information, behavior tags, and status indicators. This fact table should be the source for all other fact tables involving customer. For example, an orders fact table would benefit from such a complete customer profile, but the grain of the orders fact table is drastically sparser than the customer status fact table. Use the status fact table as the source of the proper keys when you create an orders fact record in the back room. Decide on the exact effective date/time of the orders record, and grab the customer, behavior, and status keys from the customer status fact table and insert them into the orders table. This ETL processing scenario can be used for any fact table in the EDW that has a customer dimension. In this way, you add considerable value to all these other fact tables.

This article has described a scalable approach for extreme customer status tracking. The move toward extreme status tracking has been coming on like an express train, driven both by customer facing processes that are capturing micro-behavior, and by marketing’s eagerness to use this data to make decisions. The customer status fact table is the central switchboard for capturing and exposing this exciting new data source.

# Design Tip #37: Modeling A Pipeline With An Accumulating Snapshot

June 13, 2002
© Kimball Group. All rights reserved.

[Print this Article](http://www.kimballgroup.com/2002/06/13/design-tip-37-modeling-a-pipeline-with-an-accumulating-snapshot/print/)

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.

# Design Tip #124 Alternatives for Multi-valued Dimensions

June 2, 2010
© Kimball Group. All rights reserved.

[Print this Article](http://www.kimballgroup.com/2010/06/02/design-tip-124-alternatives-for-multi-valued-dimensions/print/)

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:

**Five Alternatives for Better Employee Dimension Modeling**

[Print this Article](http://www.kimballgroup.com/2009/08/17/five-alternatives-for-better-employee-dimension-modeling/print/)

The employee dimension presents one of the trickier challenges in data warehouse modeling. These five approaches ease the complication of designing and maintaining a ‘Reports To’ hierarchy for ever-changing reporting relationships and organizational structures.

Most enterprise data warehouses will eventually include an Employee dimension. This dimension can be richly decorated, including not only name and contact information, but also job-related attributes such as job title, departmental cost codes, hire dates, even salary-related information. One very important attribute of an employee is the identity of the employee’s manager. For any manager, we’d like to work down the Reports To hierarchy, finding activity for her direct reports or her entire organization. For any employee, we’d like to work up the hierarchy, identifying his entire management chain. This Reports To hierarchy presents significant design and management challenges to the unwary. This article describes approaches for including this relationship in a dimensional model. The Employee Dimension.

The basic structure of the Employee dimension is shown in Figure 1. The unique feature of a Reports To hierarchy is that a manager is also an employee, so Employee has a foreign key reference to itself, from Manager Key to Employee Key.



**Figure 1: Basic structure of the Employee dimension and Reports To hierarchy**

Someone new to dimensional modeling might leave the table as it is currently designed as the Manager/Employee relationship is fully described. Assuming you can populate the table, this design will work if an OLAP environment is used to query the data. Popular OLAP tools contain a Parent-Child hierarchy structure that works smoothly and elegantly against a variable-depth hierarchy modeled as shown here. This is one of the strengths of an OLAP tool.

However, if you want to query this table in the relational environment, you’d have to use a CONNECT BY syntax. This is very unattractive and probably unworkable:

* Not every SQL engine supports CONNECT BY.
* Even SQL engines that support CONNECT BY may not support a GROUP BY in the same query.
* Not every ad hoc query tool supports CONNECT BY.

**Alternative 1: Bridge Table using Surrogate Keys**

The classic solution to the Reports To or variable-depth hierarchy problem is a bridge table technique described in *The Date Warehouse Toolkit* (Wiley 2002), p.162-168 and illustrated by Figure 2. The same Employee dimension table as above relates to the fact table through a bridge table.



**Figure 2: Classic relational structure for a Reports To hierarchy**

The Reports To Bridge table contains one row for each pathway from a person to any person below him in the hierarchy, both direct and indirect reports, plus an additional row for his relationship to himself. This structure can be used to report on each person’s activity; the activity of their entire organization; or activity down a specified number of levels from the manager.

There are several minor disadvantages to this design:

* The bridge table is somewhat challenging to build.
* The bridge table has many rows in it, so query performance can suffer.
* The user experience is somewhat complicated for ad hoc use, though we’ve seen many analysts use it effectively.
* In order to drill up — to aggregate information up rather than down a management chain — the join paths have to be reversed.

The major challenge comes when we want to manage Employee and the Reports To hierarchy as a Type 2 dimension — a dimension for which we are tracking history rather than updating in place. This bridge table would still work in theory; the problem is the explosion of Employee and Reports To Bridge records to track the changes.

To understand the problem, look back at Figure 1 and think about it as a Type 2 dimension for a medium-sized company with 20,000 employees. Imagine that the CEO — the top of the hierarchy — has 10 senior VPs reporting to her. Let’s give her a Type 2 change that generates a new row and hence a new Employee Key. Now, how many employees are pointing to her as their manager? It’s a brand new row, so of course no existing rows point to it; we need to propagate 10 new Type 2 rows for each of the senior VPs. The change ripples through the entire table. We end up replicating the complete Employee table because of one attribute change in one row. Even aside from the obvious implication of data volume explosion, simply teasing apart the logic of which rows need to be propagated is an ETL nightmare.

**Alternative 2: Bridge Table with Separate Reports To Dimension**

Tracking the history of changes in a variable depth hierarchy such as an employee Reports To hierarchy is especially challenging when the hierarchy changes are intermingled with other Type 2 changes in the dimension. An obvious solution is to separate the Employee dimension from the Reports To relationship. Simplify Employee by removing the self-referencing relationship, and create a new Reports To dimension, as illustrated in Figure 3.



**Figure 3: Separate Employee and Reports To (or Job) dimensions**

The key elements that distinguish this design from the classic structure are:

* Eliminate the surrogate key for manager from the Employee dimension, and hence the recursive foreign key relationship.
* The Reports To dimension has very few columns: surrogate keys, personnel numbers, and names. The only Type 2 attribute is possibly the Manager Position Number.
* If you’re exclusively using OLAP to query the schema, the bridge table is unnecessary.

If the business users don’t need to track changes in the full reports-to hierarchy, this solution works neatly. Employee is a Type 2 dimension. We see the name of each employee’s manager. If Employee.ManagerName is managed as Type 2 we can easily see the names of all past bosses from the Employee dimension. If Reports To is managed as Type 1 – we’re not tracking changes in the reporting structure – it is no more difficult to populate and maintain than in the classic solution.

If the business users absolutely must see the history of the reporting relationship, this solution will be challenging. We’ve simplified the management problem by separating out the Reports To and Employee dimensions, but if we get a major organizational change we’re still going to have to propagate a lot of new rows in both Reports To and the bridge table.

**Alternative 3: Bridge Table with Natural Keys**

In order to track changes in a Reports To hierarchy for anything other than trivial data volumes, we need a solution that does not use surrogate keys. The classic structure described in Figure 2 works fine at query time, but it’s a maintenance challenge. Our natural key alternative is illustrated in Figure 4.



**Figure 4: Tracking history in the reports-to relationship with a natural key bridge table**

The key elements of this design relative to the classic structure of Alternative 1 are:

* Eliminate the surrogate key for manager from the Employee dimension, and hence the recursive foreign key relationship.
* Include the Employee dimension twice in the schema, once as the employee (linked directly to the fact table), and once as the manager (linked via the bridge table). The Manager dimension table is simply a database view of the Employee dimension.
* The bridge table is built on employee numbers – the natural key carried in the source systems – rather than the data warehouse surrogate keys. It’s like the classic bridge table except that we need start and end dates to uniquely identify each row.
* The propagation of new rows in the bridge table is substantially fewer than before since new rows are added when reporting relationships change, not when any Type 2 employee attribute is modified (as in Figure 2). A bridge table built on natural keys is an order of magnitude easier to manage – though still quite challenging.

A primary design goal is to be able to find all the fact rows associated with a manager and her entire organization, as the organization was structured at the time of the event measured in the fact table. This is a complicated query:

* From the Manager view of the Employee dimension, find the manager we’re interested in.
* Join to the bridge table to find the personnel numbers and row dates for the employees in her organization.
* Join again to the Employee dimension to find the surrogate Employee Key for the people in the organization.
* Finally, join to the fact table to pick up all facts associated with these employees.
* The joins to the Bridge table and Manager view of Employee must constrain to pick up only the one row that’s in effect as of the time of the fact transaction.

SELECT Manager.ManagerName, Employee.EmployeeName, SUM(FactTable.SomeFact) AS OrganizationalSum

FROM FactTable

INNER JOIN Employee — standard dimensional join

ON (FactTable.EmployeeKey = Employee.EmployeeKey)

INNER JOIN NKBridge — needs a date constraint

ON (Employee.PersonnelNum = Bridge.PersonnelNum

AND Fact.DateKey BETWEEN Bridge.RowStartDate and Bridge.RowEndDate)

INNER JOIN Manager — needs a date constraint

ON (Bridge.MgrPersonnelNum = Manager.MgrPersonnelNum

AND Fact.DateKey BETWEEN Manager.RowStartDate AND Manager.RowEndDate)

WHERE Manager.ManagerName = ‘Name of specific person’

GROUP BY Manager.ManagerName, Employee.EmployeeName

The natural key bridge table approach is unwieldy. Its main advantage is that it’s feasible to maintain. It also avoids breaking out the reporting relationship into a separate dimension, as in Alternative2. Any queries that don’t involve the Reports To structure can drop the bridge table and Manager dimension view. Disadvantages include:

* Query performance is a concern as the queries are complex and the bridge table will grow quite large over time.
* The technique is not appropriate for broad ad hoc use. Only a tiny percentage of power users could ever hope to master the complex query structure.
* The technique relies on dynamic “date-bracketed” joins between the tables, and hence cannot be implemented in OLAP technology.

**Alternative 4: Forced Fixed Depth Hierarchy Technique**

It is tempting to force the structure into a fixed depth hierarchy. Even a very large company probably has fewer than 15-20 layers of management, which would be modeled as 15-20 additional columns in the Employee dimension. You’ll need to implement a method of handling the inevitable future exceptions. A fixed depth employee dimension table is illustrated in Figure 5.



**Figure 5: Forced fixed depth reports to hierarchy**

The Employee Org Level Number tells us what level from the top of the hierarchy we’ll find this employee. Usually we fill in the lower levels with the employee’s name.

At query time, the forced fixed depth hierarchy approach will work smoothly with both relational and OLAP data access. The biggest awkwardness is to train the users to query the Org Level Number first to find out the level where the employee is located – for example Level 5 – and then constrain on that column (Level05 Manager Name). A design that uses this approach must very carefully evaluate whether this two step query procedure is actually workable with a particular query tool and consider the training costs for the business users. Query performance should be substantially better than designs that include a bridge table.

The forced fixed depth approach is maintainable, but you will see a lot of propagation of Type 2 rows. If the entire fixed depth hierarchy is managed as Type2, then a new CEO (Level01 Manager) would result in a new row for every employee. Some organizations compromise by managing the top several levels as Type1.

**Alternative 5: The PathString Attribute**

By now the readers are probably desperate for a recommendation. Two years ago, a clever student in a Kimball University modeling class described an approach that allows complex ragged hierarchies to be modeled without needing to use a bridge table. Furthermore, this approach avoids the Type 2 SCD explosion described in Alternative #1, and it works equally well in both OLAP and ROLAP environments.

The PathString attribute is a field in the Employee dimension that contains an encoding of the path from the supreme top level manager down to the specific employee. At each level of the hierarchy, the nodes are labeled left to right as A, B, C, D, etc. and the entire path from the supreme parent is encoded in the PathString attribute. Every employee has a PathString attribute. The supreme top level manager has a PathString value of “A”. The “A” indicates that this employee is the left most (and only) employee at that level. Two additional columns would hold the level number, and an indicator of whether the employee is a manager or an individual contributor. Figure 6 shows a sample organization chart with PathString values for each node.



**Figure 6: Sample org chart with PathString values**

Users query the tree by creating a filter condition on the PathString column in the Employee dimension. For example, we can find all the people who report (directly or indirectly) to the employee with PathString ACB by filtering WHERE PathString LIKE ‘ACB%’. We can find direct reports by adding a clause AND OrgLevel = 4.

The advantage of the PathString approach is its maintainability. Because of this clever structure, you will see substantially fewer Type 2 rows cascading through the dimension. An organizational change high in the tree – such as creating a new VP organization and moving many people from one node to another – will result in a substantial restatement of the tree. If you’re tracking the organizational structure itself as Type 2, this would mean many new rows in the employee dimension. But it’s still fewer rows than the alternative approaches.

The main disadvantage of the PathString approach is the awkwardness of the business user query experience. This solution will require substantial marketing and education of the user community for it to be palatable.

**Recommendation**

Hopefully when you study these alternatives, you will see one that meets your needs. A Type2 “reports to” or variable depth hierarchy is a challenging beast to include in your DW/BI design. This is particularly true if you want to support ad hoc use of the structure, because you’ll need to balance ease of use and query performance against some very difficult maintenance problems. The decision matrix is complicated by the different capabilities of alternative storage engines, especially the differences between relational and OLAP.

The sad conclusion is that there is no universally great solution to the problem. In order to craft the best solution, you need to have a deep understanding of both your data and your business users’ requirements. We always strive for that understanding, but in this case, it’s imperative.

**Maintaining Dimension Hierarchies**

[Print this Article](http://www.kimballgroup.com/2008/10/27/maintaining-dimension-hierarchies/print/)

Dimension hierarchies and pre-computed aggregations can make or break your data warehouse. Here’s how to design, load and maintain true hierarchies while working around bad data sources and optimizing for usability and performance.

Dimensions are key to navigating the data warehouse / business intelligence system, and hierarchies are the key to navigating dimensions. Any time a business user talks about wanting to drill up, down or into the data, they are implicitly referring to a dimension hierarchy. In order for those drill paths to work properly, and for a large DW/BI system to perform well, those hierarchies must be correctly designed, cleaned, and maintained.

Hierarchies are important not just for usability. They play a huge role in query performance for a modern DW/BI system: aggregations are often precomputed and stored for intermediate hierarchy levels and transparently used in queries. Precomputed aggregations are one of the most valuable tools to improve query performance, but in order for them to work, your hierarchies have to be clean.

**Start with the Design**

The solution to the problem of maintaining hierarchies begins during the design phase. For every substantial dimension, spend time thinking through the hierarchical relationships. Business-user input is absolutely imperative, as is time spent exploring the data.

The first question to resolve is what are the drilldown paths or hierarchies in each dimension? Most dimensions have a hierarchy, even if it’s not coded in the transaction system. A core dimension such as customer, product, account, or even date may have many hierarchies. Date provides a good example that we all understand.

The date dimension often has three or more hierarchies. Novice dimensional modelers will try to create a single hierarchy that goes from day to week, month, quarter, and year. But that just doesn’t work! Weeks do not roll up smoothly to months or even years. There is usually a separate fiscal calendar, and sometimes several others.

Display the hierarchies graphically to review them with the business users. The diagram below shows clearly the different hierarchies and levels that will be available. Notice the attributes that apply at different levels. This picture is a graphical display suitable for communicating with users and among the DW/BI team; it does not represent the table’s physical structure. Get user buy-in on the hierarchies, levels, and names. Equally important, test how much transformation you need to apply to the actual data in order to populate these hierarchical structures.




The familiar date dimension contains lessons that are applicable to the administration of all dimensions:

* **• You can have multiple hierarchies.** Most interesting dimensions have several alternative hierarchies. Work with business users to name columns and hierarchies so that the meaning of each is clear.
* **You must have many-to-one referential integrity between each level:** a day rolls up to one and only one month, month to quarter, and quarter to year.
* If the data warehouse environment (as opposed to the original source) maintains referential integrity with explicit physical tables for each level, then **a unique primary key must be identified at each level.** If these keys are artificial surrogate keys, then they should be hidden from the business users in the final single, flat denormalized dimension table in the presentation layer of the data warehouse. A common error is to think of the key for the month level as month name (January) or month number. The correct primary key is year and month. This is a very common mistake which we encounter in many kinds of dimensions. In a geography dimension, for example, city name alone is not an identifier column; it needs to be some combination of city, state, and perhaps country.
* **Think carefully during the design phase** about whether columns can be reused between hierarchies. You might think that the week hierarchy could share the year column with the calendar hierarchy, but what about the first and last weeks of the year? If our business rule is to have week 1 for a new year start on the first Monday of the year, Week 1 of 2009 starts on January 5. January 1-4 will fall in 2008 for the week hierarchy. You need a separate year-of-week column. Sometimes you do want hierarchies to intersect, but you must be certain that the data will support that intersection.

**Load Normalized Data**

The date dimension hierarchies are easy to load and maintain. Nothing is more predictable than the calendar, and no user intervention is required. Other dimensions are often populated from imperfect source systems, including the least perfect of all: the spreadsheet.

If your source systems are imperfect, managing the hierarchies over time is painful. Optimally, hierarchies should be maintained before the data warehouse — in the transaction system or a master data management (MDM) system. With good normalized source data, the data warehouse will never see malformed data. In the real world, we’re not always so lucky. Data warehouse teams have been managing master data for decades and in many organizations will continue to do so.

Consider a product dimension for a retail store, with a hierarchy that goes from product to brand, category, and department. In this example, the product hierarchy isn’t officially part of the transaction systems, but instead is managed by business users in the Marketing department. When we initially load the data warehouse, our incoming data is as illustrated in table below:




The scenario described here is not ideal: this product dimension is not well maintained by the source systems. Most of it is fine, but notice the last row of data: we have a typo in the category, which breaks referential integrity. The “Ice Creamy” brand in one row rolls up to Frozen Desserts, and in another row to Frozen. This is forbidden.

You should find and fix problems like these early on, before you even start building the ETL system. Your ETL system must implement checks, to confirm that each category rolls to one department, and each brand to one category. But by the time you’re actually loading the historical data, you should have worked with the source systems and business users to fix the data errors.

The real challenge lies with ongoing updates of the dimension table. We don’t have time during nightly processing to have a person examine a suspect row and make an intelligent determination about what to do. If the data arriving at the ETL system’s door is suspect, the ETL system can’t distinguish between bad data and intentional changes. This is one of the hazards of developing a prototype or proof of concept. It’s easy to fix up the data on a one-time basis; keeping it clean over time is hard.

**Maintain True Hierarchies**

Clean source data is essential. True hierarchies are often maintained in normalized tables, as illustrated below. Optimally, this maintenance occurs before the data warehouse proper, either in the source transaction system or a master data management system.


You can write an ETL process to move this nicely structured data into the dimension table; it’s a two-step process. Start at the top of the hierarchy (department), and perform inserts and updates into normalized tables in the staging area. Work down to the leaf level (product). Your staging tables will look similar to the structures in the sample product hierarchy table presented earlier. Once you’ve performed the extract step and have staged all the hierarchical data, write a query to join these tables together and perform standard dimension processing from the staging area into the data warehouse dimension.

The product dimension in the data warehouse should be denormalized into a single flattened dimension table. The normalization illustrated above is the design pattern for the source system and staging areas, not the actual dimension table that users query.

**Address Dirty Sources**

Not everyone has a well-designed source system with normalized hierarchies as illustrated above. It’s common in the DW/BI world for hierarchies to be managed by business users. Transaction systems tend to have only enough information to do their job, and business users often have a legitimate need for alternative, richer rollups and attributes. What can you do?

* **Modify the source systems.** This is extraordinarily unlikely, unless your organization wrote those systems.
* **Buy and implement a master data management (MDM) system that manages the process of defining and maintaining hierarchies.** This is the best solution, though MDM is expensive in terms of software license but especially management commitment and attention.
* **Write an applet to manage a specific user hierarchy.** Keep your design simple, solving only the problem in front of you – for example, the product hierarchy. If you get carried away, you’ll find yourself developing what amounts to a MDM solution.

A true hierarchy has referential integrity between each of its levels. Remember that this is fundamentally a data quality issue that is enforced in the back room or source systems; it’s typically not carried into the presentation area as separate tables or snowflakes of tables. When a dimension has a true hierarchy, you gain two huge benefits:

* **You will be able to define and maintain precomputed aggregations at intermediate levels of the hierarchy.** In other words, you can pre-compute and store an aggregate at the Month level or the Product Brand level. Precomputed aggregations are one of the most important tools for improving query performance in the DW/BI system.
* **You will be able to integrate data at different levels of granularity.** Sometimes data naturally exists at an aggregate level. For example, our store might develop a long-term sales forecast by month and category. We can create a subset dimension at the category level to associate with the forecast facts, and then join together actual and forecast sales, if and only if the product hierarchy is a true hierarchy.

**Make it Perform**

Those with large data warehouses, especially those with large dimensions, need to worry about dimension hierarchies. The performance benefits of precomputed aggregations are tremendous, and they will make or break the usability of the BI/DW system. To realize these benefits, you must implement procedures to maintain hierarchical information correctly in the source system or a master data management system.

In the meantime, users can benefit from navigation paths that look like hierarchies but really aren’t. Business users have legitimate reasons for wanting to group information together, and it’s our job to make that not just possible, but also easy and well-performing. Just make sure that your project has the resources to ensure success!

# Design Tip #62: Alternate Hierarchies

December 7, 2004
© Kimball Group. All rights reserved.

[Print this Article](http://www.kimballgroup.com/2004/12/07/design-tip-62-alternate-hierarchies/print/)

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 Customer Region Hierarchy 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.

**Help for Hierarchies**

[Print this Article](http://www.kimballgroup.com/1998/09/02/help-for-hierarchies/print/)

**Helper tables handle dimensions with complex hierarchies**

In last month’s column, I talked about the difficult modeling situation in which one dimension you want to attach to a fact table takes on multiple values for each fact table record. The most vexing aspect of the problem was that you often don’t know how many values the dimension takes on until you see the data itself. The example I discussed was from the healthcare industry where one patient could have multiple diagnoses. The solution to last month’s problem was to create a special helper table between the fact table and the dimension table. This helper table created a many-to-many link between the fact table and the dimension table. As a purist star-join dimensional designer, I like to avoid such situations, but there are a few compelling circumstances, such as the multiple-diagnosis example, where the physical world demands modeling a many-to-many relationship between a fact table and a dimension table.

This month I’ll tackle another real-world modeling situation where the solution will turn out to be another helper table between the fact table and the dimension table. In this case, however, it is not because of a many-to-many relationship. This time the dimension has a complex hierarchical structure of variable depth, and I want to navigate this structure with my dimensional model and standard SQL.



Figure 1. A revenue reporting data mart showing the Revenue fact table whose grain is the indivdual invoice line item, as well as the Customer dimension table whose grain is the individual billed customer.

Consider the simple business situation shown in Figure 1. You can imagine that the fact table represents the revenue from consulting services that a fictitious company called Big Nine Consultants sells to various corporate clients. The grain of the fact table is the line item on each consulting invoice sent to one of the corporate clients. The fact table has a straightforward set of dimensions, including:

* Date (of service)
* Customer
* Consulting Service
* Consulting Manager
* Project Status
* Invoice Number (degenerate).

I call the Invoice Number degenerate because when you try to make a normal dimension from this key, you discover that you have already used all the interesting information in the other dimensions that might otherwise have been stored with this key. This result is characteristic of dimensional models. Very often the invoice number, or the bill of lading number, or the ticket number, is a degenerate dimension. You want these keys in the design because they are the basis for grouping line items on a particular invoice or ticket, but you don’t need to bother creating a dimension when there are no attributes for such keys.

In Figure 1, the main attention is focused on the Customer dimension, which I have shown in detail. With this schema design you can run all sorts of interesting queries by constraining and grouping various attributes in the Customer dimension. You can add up consulting revenue and hours billed for any configuration of Customer. The Customer table joins directly to the fact table.

Perhaps, as you are working on the design of this consulting invoices data mart, a user interview participant points out that the consulting services for the largest and most complex customers are sold at several different organizational levels. This user would like to create reports that show total consulting sold not only to individual departments, but also to divisions, subsidiaries, and overall enterprises; the report still must correctly add up the separate consulting revenues for each organization structure. Figure 2, shows a simple organizational structure, where each node in the tree is a Customer consulting services are sold to.



Figure 2. A schematic diagram of customer organizations that Big Nine Consultants sells consulting services to. Big Nine sells to all of these customers, and they are hierarchically related.

Figure 1 does not contain any information about how these separate Customers relate to each other. A simple computer science approach to storing such information would add a Parent Key field to the Customer dimension. The Parent Key field would be a recursive pointer that would contain the proper key value for the parent of any given customer. A special null value would be required for the topmost Customer in any given overall enterprise. Although this simple recursive pointer lets you represent an arbitrary organizational tree structure of any depth, there is a killer problem that defeats its use in your data warehouse.

The problem is that you cannot use the recursive pointer with SQL to join the dimension table with the fact table and add up the consulting revenues or hours for a set of organizations, as in Figure 2. NSI-standard SQL makes no attempt to deal with recursive pointers, and even such facilities as Oracle’s CONNECT BY do not let you use a join in the same SQL statement as CONNECT BY. Thus in Oracle, although you can enumerate an organizational hierarchy defined via a recursive pointer field in a dimension table, you cannot add anything up by joining that dimension table to a fact table.

Instead of using a recursive pointer, you can solve this modeling problem by inserting a helper table between the dimension table and the fact table, as shown in Figure 3. Amazingly enough, you don’t have to make any changes to either the dimension table or the fact table; you just rip the join apart and insert the helper table.



Figure 3. I have inserted a helper table in between the fact and dimension table that lets me navigate the organizational hierarchy.

The helper table contains one record for *each separate path from each node in the organization tree to itself and to every node below it*. There are, then, more records in the helper table than there are nodes in the tree. In Figure 3 we need a total of 43 records in the helper table. See if you can work this out.

Each record in the helper table contains the fields:

* Parent Customer Key
* Subsidiary Customer Key
* Depth From Parent
* Lowest Flag
* Topmost Flag.

If you are descending the tree from certain selected parents to various subsidiaries, you join the dimension table to the helper table and the helper table to the fact table with the joins as shown in Figure 3. The Depth From Parent field counts how many levels the subsidiary is below the parent. The Lowest Flag field is True only if the subsidiary has no further nodes beneath it. The Topmost Flag field is True only if the parent has no further nodes above it.

The beauty of this design is that you can place any normal dimensional constraint against the Customer dimension table and the helper table will cause all the fact table records for the directly constrained customers plus all their subsidiaries to be correctly summarized. In other words, you can use your standard relational databases and your standard query tools to analyze the hierarchical structure.

If the field Depth From Parent is equal to one, then only the immediate subsidiaries of the directly constrained customers will be summarized. If the Lowest Flag is True, then only the lowest subsidiaries of the directly constrained customers will be summarized.

The joins shown in Figure 3 let you summarize an organizational structure downward from the directly constrained nodes. By reversing the sense of the joins (for example, connecting the customer dimension primary key to the subsidiary customer key), you can move up the organizational structure instead. When Depth From Parent is equal to one, then you are referring to the immediate parent of a directly constrained customer. When Topmost Flag is True, you have selected the supreme parent of a directly constrained customer.

You can generalize this scheme by adding Begin Effective Date and End Effective Date to each record in the helper table. In this way, you can represent changing organizational structures. When a group of nodes is moved from one part of an organizational structure to another, such as with an acquisition, only the records that refer to paths from *outside parents into the moved structure* need to be changed. All records referring to paths entirely within the moved structure are unaffected. This is an advantage over other tree representation schemes where all the nodes in the tree need to be numbered in a global order. Also, these other representation schemes generally do not preserve the ability of standard SQL to summarize the results in an associated fact table the way this scheme does.

If you have an organization where a subsidiary is jointly owned by two or more parents, then you can add a Weighting Factor field to the helper table. Strictly speaking this is no longer a tree. I call this an *irregular tree*. In irregular trees with joint ownership situations, you identify those nodes with two or more direct parents. The fraction of ownership by each parent is identified, and the sum of the fractions for that jointly owned node must be equal to one. Now every helper table record from any parent that terminates at the jointly owned node or crosses through the jointly owned node must use the proper Weighting Factor. The Weighting Factor, if present in the design, must then be multiplied at query time against all additive facts being summarized from the fact table. In this way, the correct contributions to consulting revenue and total hours (in the original example) will be added up through the tree.

You can also use this approach, up to a point, to model manufacturing parts explosions. The tree structures of manufacturing parts explosions can be manipulated to fit the examples discussed in this article. You can even represent a repeated subassembly in the same way as a jointly owned subsidiary, although in this case you don’t need a Weighting Factor because the subassembly is really repeated, not shared. The main limitation in using this approach for manufacturing parts explosions is the sheer number of subassemblies and parts present in a big example. A huge parts explosion with hundreds of thousands or millions of parts would almost certainly result in a helper table with “more records than there are molecules in the universe.” At some point, this helper table becomes infeasible.

This column is my last in this role as “Data Warehouse Architect” for *DBMS* magazine. Next month the magazine and I are going to be reborn as *Intelligent Enterprise* and the “Warehouse Architect,” respectively. I am writing a special expanded column for the inaugural issue of the new magazine on the Brave New Requirements we now face in designing data warehouses. Looking forward to seeing you there. Bye for now.

**Design Tip #22: Variable Depth Customer Dimensions**

April 9, 2001
© Kimball Group. All rights reserved.

[Print this Article](http://www.kimballgroup.com/2001/04/09/design-tip-22-variable-depth-customer-dimensions/print/)

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.

# Design Tip #105 Snowflakes, Outriggers, and Bridges

September 3, 2008
© Kimball Group. All rights reserved.

[Print this Article](http://www.kimballgroup.com/2008/09/03/design-tip-105-snowflakes-outriggers-and-bridges/print/)

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.

# Wrangling Behavior Tags

[Print this Article](http://www.kimballgroup.com/2002/05/09/wrangling-behavior-tags/print/)

In my previous column, I argued that behavior was the new marquee application of the 2000s (“[Behavior: The Next Marquee Application](http://www.kimballgroup.com/2002/04/16/behavior-the-next-marquee-application/),” April 16, 2002). As we’re entering the third decade of data warehousing, we have progressed beyond the shipments-and-share applications of the ’80s, past the customer-profitability applications of the ’90s, to this new focus on individual customer behavior.

I also pointed out that the granularity of the data has increased by roughly a factor of 1,000 each decade. The megabyte databases of the ’80s gave way to gigabytes in the ’90s. Gigabytes are clearly giving way to terabytes in the 2000s. As I remarked in the previous column, the transition to terabyte databases caught me a little by surprise in the last few years because I was hoping that we had finally stopped growing our databases once we recorded every atomic sales transaction in our largest businesses.

But our databases are still growing without bounds because we’re recording more and more subtransactions in advance of the sales transaction. Even if the customer eventually makes only a single purchase, we might capture all the behavior that led up to it. When we extract data from all the customer-facing processes of the business, we see physical visits to brick-and-mortar stores, Web site page requests from e-store visits, calls to support lines, responses to mailings, receipt records of HTML emails containing Web bugs that report back the display of the email on the end user’s screen, product deliveries, product returns, and payments made either by regular mail or online. The flood of data from all the customer-facing processes surrounds and explains the final solitary sales transaction. The scary thing about all these subtransactions is that there’s no obvious barrier or limit to the amount of data you might collect.

Well, it’s nice that we have all this data available for describing customer behavior, but how can we boil the terabytes down to simple, understandable behavior tracking reports?

In the previous column I described how our data mining colleagues can assign behavior tags to complex patterns of subtransactions. I’ll describe a simple, classic example. Let’s use our standard data warehouse reporting techniques to summarize three customer behavior metrics: recency, frequency, and intensity (RFI).

Recency is a measure of how recently you’ve interacted with the customer in any transaction or subtransaction. The metric of recency is the number of days elapsed since the last interaction. Similarly, frequency is a measure of how often you’ve interacted with the customer. 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 you might decide that the total number of Web pages visited is a good measure of intensity, too.

All the RFI measures can be subdivided into separate measures for each customer-facing process, but I’ll keep this example simple.

Now for every customer, we compute the RFI metrics for a rolling time period, such as the latest month. The result is three numbers. Imagine plotting the RFI results in a three-dimensional cube with the axes Recency, Frequency, and Intensity.

Now you call in your data mining colleagues and ask them to identify the natural clusters of customers in this cube. You really don’t want all the numeric results; you want the behavioral clusters that are meaningful for your marketing department. After running the cluster identifier data mining step, you might find eight natural clusters of customers. After studying where the centroids of the clusters are located in your RFI cube, you’re 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, hasn’t been seen recently

G: Frequent window shopper, mostly unproductive

H: Other

You 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. Imagine that you’re 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

This time series shows that the business 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, you’ve seen a propensity for John to start returning products. Not only is this recent behavior costly, but you worry that John will become disenchanted with your products and end up in the F behavior category!

How can you structure your data warehouse to pump out these kinds of reports? And how can you pose interesting constraints on customers to see only those who’ve gone from cluster A to cluster B in the most recent time period?

You 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 you 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 (Type 3).

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’s no simple way to perform a “straddle constraint” across two records.

In this example, you’re 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, choice 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 because a new behavior tag measurement doesn’t by itself generate a new record. Second, complex straddle constraints are easy because the relevant fields are in the same record. And third, you can easily associate the complex straddle constraints with the 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 doesn’t affect any existing applications.

Even though you’ve packed the 24-month time series of behavior tags into each customer record, you may still have other reasons for administering the customer record as a true Type 2 slowly changing dimension (SCD). In other words, you would still introduce another customer record for a specific customer if something else significant in that customer’s profile changed. In this case, you would copy the 24 behavior tag attributes with their contents over to the new record. And when you developed a behavior tag for a new month, you would visit all the records for a given customer, filling in that field, even in the old records. This is an example of a hybrid SCD, where you’re both partitioning history (Type 2) and supporting alternate realities with the succession of behavior tags (Type 3).

Using the techniques outlined in this column, you can boil down terabytes of subtransactional behavior data into a simple set of tags, with help from your data mining colleagues. You can then package the tags into a very compact and useful format that supports your high-level ease-of-use and ease-of-application-development objectives. We’re now ready to pump out all sorts of interesting behavior analyses for your marketing end users.

# Human Resources Data Marts

[Print this Article](http://www.kimballgroup.com/1998/02/02/human-resources-data-marts/print/)

**Design guidelines for querying and analyzing employee data.**

It is easy in the data mart design business to get lulled into a kind of “additive complacency,” where every data source looks like retail sales. In the simple world of retail sales all the transactions are little pieces of revenue that always add up across all the dimensions, and the dimensions themselves are tangible, physical things like Product, Store, and Date.

I frequently get asked, “Well, what about something like human resources? Most of the facts arenýt additive. Most of the facts arenýt even numbers, but they are clearly changing all the time, like numeric facts. How do I model that?” Actually, human resources data marts are a very good application for dimensional modeling. With a single design we can address just about all the major analysis and reporting needs. We just have to be careful about what is a dimension and what is a fact.

To frame the problem, letýs describe a typical human resources environment. We assume that we are the human resources department for a large enterprise with more than 100,000 employees. Each employee has a complex human resources profile with at least 100 attributes. These attributes include all the standard human resources descriptions including date of hire, job grade, salary, review dates, review outcomes, vacation entitlement, organization, education, address, insurance plan, and many others. In our large organization, there is a constant stream of transactions against this employee data. Employees are constantly being hired, transferred, promoted, and having their profiles adjusted in various ways.

In our design, we will address three fundamental kinds of queries run against this complex human resources data. In our first kind of query, we want to report summary statuses of the entire employee base on a regular (monthly) basis. In these summaries we want counts, instantaneous totals, and cumulative totals, including such things as number of employees, total salary paid during the month, cumulative salary paid this year, total and cumulative vacation days taken, vacation days accrued, number of new hires, and number of promotions. Our reporting system needs to be extremely flexible and accurate. We want these kinds of reports for all possible slices of the data, including time slices, organization slices, geographic slices, and any other slices supported in the data. Remember the basic tenet of dimensional modeling: If you want to be able to slice your data along a particular attribute, you simply need to make the attribute appear in a dimension table. By using the attribute as a row header (with SQL GROUP BY) you automatically “slice.” We demand that this database support hundreds of different slicing combinations.

The hidden reporting challenge in this first kind of query is making sure that we pick up all the correct instantaneous and cumulative totals at the end of each month, even when there is no activity in a given employeeýs record during that month. This prohibits us from merely looking through the transactions that occurred during the month.

In our second kind of query, we want to be able to profile the employee population at any precise instant in time, whether or not it is at the end of a month. We want to choose some exact date and time at any point in our organizationýs history and ask how many employees we have and what their detailed profiles were on that date. This query needs to be simple and fast. Again, we want to avoid sifting through a complex set of transactions in sequence to construct a snapshot for a particular date in the past.

Although in our first two queries we have argued that we cannot depend directly on the raw transaction history to give us a rapid response, in our third kind of query we demand that every employee transaction be represented distinctly. In this query, we want to see every action taken on a given employee, with the correct transaction sequence and the correct timing of each transaction. This detailed transaction history is the “fundamental truth” of the human resource data and should provide the answer to every possible detailed question, including questions not anticipated by the original team of data mart designers. The SQL for these unanticipated questions may be complex, but we are confident the data is there waiting to be analyzed.

In all three cases, we demand that the employee dimension is always a perfectly accurate depiction of the employee base for the instant in time specified by the query. It would be a huge mistake to run a report on a prior month with the current monthýs employee profiles.

Now that we have this daunting set of requirements, how on earth can we satisfy all of them and keep the design simple? Amazingly, we can do it all with a single dimensional schema with just one fact table and a powerful dimension table called the Employee Transaction dimension. Take a moment to study Figure 1.



Figure 1. The Human Resources Data Mart dimensional model. The fact table is a snapshot that contains monthly numeric summaries that are difficult to calculate from the underlying transactions. The employee transaction dimension table contains a record for every transaction performed on every employee record. All of the employee attributes starting with Name actually consist of multiple fields, indicated by a plus sign.

The human resources data mart consists of a fairly ordinary looking fact table with three dimensions: employee transaction, month, and organization. We show all three dimensions in Figure 1, although we only explode the employee transaction table in detail because that is the interesting part of the design. The month table contains the usual descriptors for the corporate calendar, at the grain of the individual month. The organization dimension contains a description of the organization that the employee belongs to at the close of the relevant month.

The employee transaction dimension table contains a complete snapshot of the employee record for each individual employee transaction. The employee transaction key is an artificial key made during the extraction process, and should be a sequentially assigned integer, starting with 1. Resist the urge to make this a smart key consisting of employee ID, transaction code, and effective date/time. All these attributes are valuable, but they are simply attributes in the employee transaction record, where they participate in queries and constraints like all the other attributes.

The employee ID is the normal human resources “EMP ID” that is used in the production system. The transaction description refers to the transaction that created this particular record, such as Promotion or Address Change. The transaction date/time is the exact date and time of the transaction. We assume that these date/times are sufficiently fine grained that they guarantee uniqueness of the transaction record for a given employee. Therefore, the true underlying key for this dimension table is employee ID plus transaction date/time.

A crucial piece of the design is the second date/time entry: transaction end date/time. This date/time is exactly equal to the date/time of the next transaction to occur on this employee record, whenever that may be. In this way, these two date/times in each record define a span of time during which the employee description is exactly correct. The two date/times can be one second apart (if a rapid sequence of transactions is being processed against an employee profile), or the two date/times can be many months apart.

The current last transaction made against an employee profile is identified by the Last Transaction Flag being set to True. This approach allows the most current or final status of any employee to be quickly retrieved. If a new transaction for that employee needs to be entered, the flag in this particular record needs to be set to False. I never said that we donýt update records in the data warehouse. The transaction end date/time in the most current transaction record can be set to an arbitrary time in the future.

Some of you may object to the storage overhead of this design. Even in a pretty large organization, this approach doesnýt lead to ridiculous storage demands. Assume we have 100,000 employees and that we perform 10 human resources transactions on them each year. Assume further that we have a relatively verbose 2,000-byte employee profile in the employee transaction record. Five years worth of data adds up to 5 X 100,000 X 10 X 2,000 bytes, or just 10GB of raw data. If your definition of employee transaction is much more fine grained so that a job promotion requires dozens of tiny low-level transactions, then you might consider creating a small set of super transactions like Job Promotion in order to make the data sizing realistic. Admittedly, this makes the extraction task more complex.

This compact design satisfies our three categories of queries beautifully. The first kind of query for fast high-level counts and totals uses the fact table. All the facts in the fact table are additive across all the dimensions except for the facts labeled as balances. These balances, like all balances, are semiadditive and must be averaged across the time dimension after adding across the other dimensions. The fact table is also needed to present additive totals like salary earned and vacation days taken.

The particular employee transaction key used in a fact table record is the precise employee transaction key associated with the stroke of midnight on the last day of the reporting month. This guarantees that the month-end report is a correct depiction of all the employee profiles.

The second query is addressed by the employee transaction dimension table. You can make a time-based cut through the employee database by choosing a specific date and time and constraining this date and time to be equal to the transaction date/time and less than the transaction end date/time. This is guaranteed to return exactly one employee profile for each employee whose profile was in effect at the requested moment. The query can perform counts and constraints against all the records returned from these time constraints.

The third kind of query can use the same employee transaction dimension table to look in detail at the sequence of transactions against any given employee.

Some of you may be wondering if the employee transaction dimension table isnýt really a kind of fact table because it seems to have a time dimension. While technically this may be true, this employee transaction table mainly contains textual values and is certainly the primary source of constraints and row headers for query and report-writing tools. So it is proper to think of this table as a dimension table that serves as the entry point into the human resources data mart. The employee transaction table can be used with any fact table in any data mart that requires an employee dimension as long as the notion of employee key is extended to be the employee transaction key. This design is really an embellishment of the standard slowly changing dimension we routinely use when dimensions like Product change at unexpected intervals. The key idea that makes this human resources database fit into our familiar dimensional framework is making each dimension record an individual employee transaction and then tying these records to precise moments in time.