

Kimball Design Tip #32: Doing The Work At Extract Time

By Ralph Kimball

Our mission as data warehouse designers is to publish our data most effectively. This means placing the data in the format and framework that is easiest for end users and application developers to use.

In the back room of our data warehouse we must counter our natural minimalist tendencies when we are preparing the data for final consumption by the end users and application developers. In many important cases, we should deliberately trade off

- * increased back room processing, and
- * increased front room storage requirements

in exchange for

- * symmetrical, predictable schemas that users understand,
- * reduced application complexity, and
- * improved performance of queries and reports.

Making these tradeoffs should be an explicit design goal for the data warehouse architect. A good data warehouse architect will resist the urge to dump extra analysis and table manipulation on the end users and the application designers. Beware the vendors who argue in favor of complex schemas! But of course these tradeoffs must be chosen judiciously and not overused. Let's look at half a dozen situations where we do just enough work at extract time to really make a difference. We'll also try to draw some boundaries so we know when not to overdo it and spoil the final result. We'll start with some rather narrow examples and gradually expand our scope.

Modeling Events Across Multiple Time Zones

Virtually all measurements recorded in our data warehouses have one or more time stamps. Sometimes these are simple calendar dates but increasingly, we are recording the exact time to the minute or the second. Also, most of our enterprises span multiple time zones, either in the United States, across Europe, across Asia, or around the world. In these cases we have a natural dilemma. Either we standardize all of our time stamps to a single well identified time zone, or we standardize all of our time stamps to the correct local wall clock time when the local measurement event occurred. If we want to convert from GMT to local time, maybe we just figure out which time zone the measurement was taken in, and we apply a simple offset. Unfortunately, this doesn't work. In fact, it fails spectacularly. The rules for international time zones are horrendously complicated. There are more than 500 separate geographic regions with distinct time zone rules. Moral of the story: don't compute time zones in your application, rather add an extra time stamp foreign key in every place you have an existing time stamp, and put the BOTH the standard and local times in your data. In other words, do the work at extract time, not query time, and give up a little data storage.

Verbose Calendar Dimensions

All fact tables in dimensional data warehouse schemas should eschew native calendar date stamps in favor of integer valued foreign keys that connect to verbose calendar dimensions. This recommendation is not based on a foolish dimensional design consistency, but rather it recognizes that calendars are complicated and applications typically need a lot of navigation help. For example,

native SQL date stamps do not identify the last day of the month. Using a proper calendar date dimension, the last day of the month can be identified with a Boolean flag, making applications simple. Just imagine writing a query against a simple SQL date stamp that would constrain on the last days of each month. Here is an example where adding the machinery for an explicit calendar date dimension simplifies queries and speeds up processing by avoiding complex SQL.

Keeping the Books Across Multiple Currencies

The multi time zone perspective discussed in the first example often occurs with the related issue of modeling transactions in multiple currencies. Again, we have two equal and legitimate perspectives. If the transaction took place in a specific currency (say, Swiss francs) we obviously want to keep that information exactly. But if we have a welter of currencies, we find it hard to roll up results to an international total. Again, we take the similar approach of expanding every currency denominated field in our data warehouse to be two fields: local currency value and standard currency value. In this case, we also need to add a currency dimension to each fact table to unambiguously identify the local currency. The location of the transaction is not a reliable indicator of the local currency type.

Contrasting Units of Measure in a Product Pipeline

Most of us think that product pipeline measurements are pretty simple and don't have the complications that financial services, for example, have. Well, spend some time with manufacturing people, distribution people, and retail people, all talking about the same products in the same pipe. The manufacturing people want to see everything in car load lots or pallets. The distribution people want to see everything in shipment cases. The retail people can only see things in individual "scan units". So what do you put in the various fact tables to keep everyone happy? The WRONG answer is to publish each fact in its local unit-of-measure context and leave it to the applications to find the right conversion factors in the product dimension tables! Yes, this is all theoretically possible, but this architecture places an unreasonable burden on the last step of the process where the end users and the application developers live. Instead, present all the measured facts in a single standard unit of measure and then, in the fact table itself, provide the conversion factors to all the other desirable units of measure. That way, applications querying the pipeline data from any perspective have a consistent way to convert all the numeric values to the specific, idiosyncratic perspective of the end user.

Physical Completion of a Profit and Loss (P&L) Design

A profit and loss fact table is very powerful because it presents all the components of revenue and cost at (hopefully) a low level of granularity. After providing this wonderful level of detail, designers sometimes compromise their design by failing to provide all the intermediate levels of the P&L. For instance, the "bottom line profit" is calculated by subtracting the costs from the net revenue. This bottom line profit should be an explicit field in the data, even if it is equal to the algebraic sum of other fields in the same record. It would be a real shame if the user or application developer got confused at the last step and calculated the bottom line profit incorrectly.

Heterogeneous Products

In financial services like banking and insurance, a characteristic conflict often arises between the need to see all of the account types in a single "household" view of the customer, and to see the detailed attributes and measures of each account type. In a big retail bank, there may be 25 lines of business and more than 200 special measures associated with all the different account types. You simply cannot create a giant fact table and giant account dimension table that can accommodate all the heterogeneous products. The solution is to publish the data twice. First create a single core fact table with only the four or five measures, like balance, that are common to all account types. Then, publish the data a second time, with the fact table and account dimension table separately extended for each of the 25 lines of business. Although this may seem wasteful because the huge fact table is effectively published twice, it makes the separate householding and line of business applications simple unto themselves.

Aggregations in General

Aggregations are like indexes: they are specific data structures meant to improve performance.

Aggregations are a significant distraction in the back room. They consume processing resources, add complexity to the ETL suite of applications, and they take up lots of storage. BUT, aggregations remain the single most potent tool in the arsenal of the data warehouse designer to improve performance cost effectively.

Dimensional Modeling In General

By now I hope it is obvious that the most widely used back room tradeoff used for the benefit of the end user is the practice of dimensional modeling. A dimensional model is a second normal form version of a third (or higher) normal form model. Collapsing the snowflakes and other complex structures of the higher normal form models into the characteristic flat dimension tables makes the designs simple, symmetrical and understandable. Furthermore, database vendors have been able to focus their processing algorithms on this well understood case in order to make dimensional models run really fast. Unlike some of the other techniques discussed in this design tip, the dimensional model approach can be applied across almost all horizontal and vertical application areas.