Thursday, April 18, 2002 The Premier Publication for Business Intelligence and Analytics

Data Warehousing in the Insurance Industry, A Strategic Advantage: Designing High-Performance DSS Databases

By Stephen Brobst

Published in DM Review in June 1998

One of the biggest challenges in building decision support database structures is balancing the tradeoffs between high performance and flexibility in the physical design. Data Warehousing 101 quickly teaches us that access patterns are radically different in a decision support environment than in a traditional on-line transaction processing (OLTP) environment. However, sometimes we are too eager to forget the fundamentals of Database Design 101 related to the sins of denormalization and the implications of deviating from third normal form database structures. The result is all too often a massively denormalized set of database structures for which flexibility and adherence to the business model within an organization are sacrificed to the design principle of avoiding joins at all costs. This is a bad thing. Or sometimes it is merely an issue of not bothering to untangle data quality issues propagating from the source systems.

The reality is that neither fully denormalized nor fully normalized database structures are usually the right answer for a well designed decision support database. Each opportunity for denormalization must be approached as an engineering design decision - with quantification of costs and benefits determining the appropriate solution. These decisions are particularly difficult with data models in the insurance industry due to the complexity of the business and the sophistication of analyses typically required. Unfortunately, the naive approach of denormalizing into a single "fact" table is rarely appropriate because the relationships between core entities in the model are more complicated than in most industries. But it looked so easy in the text book example using a retail data model, right? Yes, but this is insurance and insurance is hard.
The key point here is that the business should be driving the technology decisions - not the other way around. If the relevant business questions cannot be answered in a reasonable way against the database structures, re-think the design. With this in mind, it is useful to review some of the basic rules of designing to third normal form database structures and to point out some of the common mistakes frequently made in denormalizing data warehouse database structures.

First normal form

To comply with the first normal form, the domain of each attribute in a relation must include only *atomic* (simple, indivisible) values. That is, there should be no overloading in the semantics of an attribute and each attribute should stand on its own.
The most frequent violations of the first normal form in data warehouse database structures typically fall into one of three categories:

1. Re-defines within an attribute domain
2. Multiple domains combined into the same attribute
3. Repeating group structures

Re-defines within an attribute occur when the value in column A of a table is used to determine the way in which some other column B should be intepreted in a business context. Column B is said to be re-defined by the value in column A. This data encoding technique is often found in older COBOL files structures and was typically a way to re-use space efficiently in fixed length file structures. For example, the status code on the policy table on one data warehouse we recently encountered required a different interpretation based on the policy type. A status code of "L" for a long-term disability policy (policy type of "LTD") meant that the policy had lapsed, whereas the same status code value of "L" meant "free look" for a life policy (policy type of "LIF"). Overloading the meaning of an attribute so that decoding based on the value of another attribute is required makes the database very unfriendly to end users. This situation arose because the records corresponding to different policy types were extracted from distinct source systems. Each source system represented an independent "silo" implementation and defined its own (often overlapping in inconsistent ways with other legacy systems) encodings for the policy status code values. While it is certainly possible (in most cases) to hide this type of problem by embedding the decode of a re-defined attribute into a view structure, this is like sweeping dirt under a rug. The recommended solution is to define a consistent set of domain values for each attribute in a relation and then implement a set of transformation rules to be applied against the source systems data to ensure atomic domain values.

Multiple domains combined into the same attribute is somewhat similar to re-defines within an attribute domain except that the attribute is overloaded without the help of another attribute. Usually, this violation of first normal from is implemented with an assumption of direct dependency from one domain value to another. For example, in the group_type_cd column in one source system that we recently encountered the domain included "LGE", "MED", and "SML" to indicate large, medium or small group types. Yet, the same field also contained values of "ASO" to indicate a self-funded group type that required "Administrative Services Only." The underlying assumption was that only large groups would be self-funding with an administrative services only group type. While this assumption is generally true, there was no business rule in the company that precluded a medium or small group from self-funding. Combining domains from *group size* and *funding arrangement* into a single attribute imposed constraints in the physical database design that did not exist in the business. Of course, one could imagine conjuring up a single set of domain values to handle all combinations of group size and funding arrangement, but this becomes cumbersome very quickly. The recommended solution is to define a separate attribute (column) for each distinct domain in the relation rather than to combine domain values. Again, this will often require up-front work in the transformation rules in translating from source system data into the target domain values - but is well worth the effort in terms of extensibility and generality of the data warehouse design.

One of the most common violations of first normal form is the repeating group structure. For example, suppose there is a requirement to keep track of the historical face value amount of annuities with monthly snapshots. One way to do this (shown in Figure 1) is to create one record for each year that the policy is on file and keep twelve buckets to keep track of monthly face value amounts on each record. These buckets are referred to as a repeating group structure because the face amount value is stored multiple times with an additional date semantic associated implicitly to each specific column in the table. The normalized design (shown in Figure 2) stores a separate row for each face value amount kept on a monthly basis.

There are many tradeoffs implicit in these two designs. The most obvious is in space allocation. Assuming that it takes 8 bytes to store the policy_id, 2 bytes for the year, 4 bytes for a face value amount, and 7 bytes for a date, each row in the denormalized table requires 58 bytes (ignoring other overhead) versus 19 bytes for each row in the normalized table. However, the normalized design requires the storage of twelve times as many rows to represent one year of face value amount snapshots. Thus, if we wish to store three years of history for ten million policies, we require approximately 1.74GB of storage for the denormalized design versus 6.84GB for the normalized design. In reality, storage for the normalized design is probably a bit overstated because policies not issued or closed on year boundaries do not need to store a row for every month outside of the active dateson the policy, whereas in the denormalized design if a policy is active at any time during the year all twelve buckets will be allocated. Different DBMS engines will treat the storage of the "unused" buckets with varying efficiency. Never-the-less, the storage penalty will still be at least a factor of three for the normalized design.

However, there are some hidden tradeoffs in the two designs. First, database updates with new face amounts will likely be at least three to ten times faster using the normalized design versus the denormalized design because load append of new rows will almost always be faster than updating buckets in existing rows because logging and other overheads can be avoided when simply inserting new rows. The extent to which the performance differentiation is closer to three or ten times better with the normalized design depends mainly on the amount of indexing on the table (since indices on policy_id and snapshot_dt would need to be maintained for each row inserted in the normalized design but not when a bucket gets updated in the denormalized design). The second hidden tradeoff is ease of access to the data for decision support query workloads.

Consider a query in which the end user seeks the average face value amount over the first six months after issue for all policies issued in 1996. One might assume that the denormalized design would be an easier target for generating the appropriate SQL. However, this is definitely not the case. The normalized design requires less than ten lines of SQL code (see Example 1) to answer the question, whereas the denormalized design requires many times that (see Example 2) and rather sophisticated use of the SQL Case function. It is pretty obvious which piece of code would be easier to develop and maintain. However, with more and more SQL being generated by front-end tools (as opposed to human programmers) it is less important to talk about how many lines of code is required than to consider the tools' ability to obtain correct and efficient answers to end user

queries. In this light, it is almost always the case that the better front-end tools will prefer the normalized design because it provides a more accurate representation of the actual business model within the tool meta data when adherence to the logical model is preserved.

Second normal form

To comply with the second normal form, every non-prime attribute must be *fully functionally dependent* on the primary key within a relation. In other words, attributes should be dependent on the full primary key within a relation. The frequently encountered violation of the second normal form is when attributes describe only part of the primary key. Consider the table structure shown in Figure 3. This table represents the assignment of agents to sales offices. The table includes a start and termination date because agents may be assigned to multiple offices over time (perhaps even at the same time). The primary key of this relation is comprised of agent_id, office_id, and start_dt. The violation of second normal form arises from the fact that office name and agent name are stored as part of this table, but are not fully functionally dependent on the primary key. Office_nm depends only on office_id and agent_nm depends only on agent_id. This means that agent_nm is stored redundantly for every row in the table which assigns an agent to an office. Similarly, office_nm is also stored redundantly in the table. Any time data is stored redundantly, there is an opportunity for data quality issues as synchronization becomes a problem across the multiple data copies.

The recommended fix for this situation is to split the denormalized table into its fundamental entities (agent and office) with an appropriate associative entity to capture the entity relationships. The cost for this solution is that additional join workload is created to bring together the agent and office attributes using the three table join as indicated in Figure 4. On the other hand, storage requirements are reduced because agent and office attributes are not stored redundantly. More importantly, however, opportunities for data inconsistency are avoided and adherence to the logical model for the business is preserved. Adherence to the logical model will make it possible to construct a more accurate semantic data model within the meta data for leading query tools on the marketplace.

Third normal form

To comply with third normal form, a relation must be in second normal form *and* every non-prime attribute must be non-transitively dependent on the primary key. The human translation of this techno-speak is that all attributes must be directly dependent on the primary key without implied dependencies through other attributes of the relation. The typical violation of the third normal form is when attributes present in a relation describe attributes other than the primaty key. For example, the relation in Figure 5 shows a policy relation with attributes about the insured (name, birth date) embedded into the relation. These attributes are dependent on the insured_id rather than the primary key (policy_id) for the table. A transitive dependence of these attributes from the insured_id to the policy_id is implied, and results in redundant storage of the attributes related to the insured individual whenever multiple claims are stored.

The normalized design for the claim table would break out the attributes of the insured individual into a separate insured table corresponding to this entity (as shown in Figure 6). In reality, an associative table would probably be implemented because there are typically multiple individuals associuated to a policy (insured, beneficiary, etc.). Beware, however, the data scrubbing needed to define a single insured record from across multiple policy records requires a significant amount of work. Matching name and address information will often be required when definitive keys (such as social security number) are not easily available from the source systems. This becomes even trickier when matching name and address across insured, beneficiary, and other sources of customer (or prospect) information. The tightness or looseness with which the matching heuristics are implemented can dramatically affect the usefulness of the resultant data. Despite the complexities involved in obtaining individualized customer records, we view this as a critical milestone in moving from a product oriented focus to a customer centric view of the data. You cannot really know who your customers are unless this information is culled from the policy master file and individualized into a clean view of each (unique) individual associated with one or more policies.

When is a little bit of sin a good thing?

The desire to deliver high-performance decision support capability against massive amounts of data with limited resources for hardware budgets dictates an occasional bending of the purist rules of normalization. Despite the advantages of normalization, selective deviation from these rules often delivers performance benefits that outweigh other sacrifices. A proper denormalization decision is one derived from careful consideration of all tradeoffs involved in its deployment. The primary tradeoffs typically involve: (1) performance, (2) storage cost, (3) flexibility, and (4) maintenance. Denormalization will usually be aimed at increasing performance at the cost of additional storage. However, it is also critical to assess the impacts upon flexibility in answering sophisticated end user queries and accurately representing the business. It is almost always a mistake to lose access to detailed data or business relationships as the result of a denormalization decision. Maintenance costs in terms of database refreshes and implications upon data quality management should also be considered.

There are three forms of denormalization typically deployed in a decision support database: (1) pre-join denormalization, (2) column replication or movement, and (3) pre-aggregation. Pre-join denormalization takes tables which are frequently joined and "glues" them together into a single table. This avoids the performance impact of frequent joins, yet typically increases storage requirements. Column replication or movement is a special case of pre-join denormalization in which only some of the columns from one table are pre-joined to another table. Assuming that only a small number of frequently accessed columns need to be replicated or moved in the denormalization, this technique affords the benefits of avoiding frequent joins without paying the full storage cost of a pre-join denormalization.

Pre-aggregation denormalization involves taking frequently accessed aggregate values and pre-computing them into physical tables within the database. This technique can provide huge performance advantages in avoiding frequent aggregation of detailed data. The storage implications are usually small compared to the size of the detailed data, but can become quite large if many multi-dimensional summaries are constructed. However, the overhead in maintaining the aggregate values should not be under-estimated. The cost of keeping the summary tables up-to-date, especially if re-alignment of business definitions such as geography or product hierarchies is frequent, can be quite burdensome. For pre-aggregation denormalization to be beneficial, it is critical to have business agreement on standardized definitions for aggregate values. Note that the cost of updating an aggregate row is typically ten times higher than the cost of inserting a new row into a detail table (transactional update cost versus bulk loading cost). An aggregate must be used very frequently to justify the maintenance costs associated with pre-aggregation denormalization. However, the performance benefits can be significant when standard summaries are established which reflect commonly accessed business metrics.

Whatever design decisions are taken, it is very important to recognize that summary values should *not* replace the detailed data in the decision support environment. Adhoc and more sophisticated queries are almost guaranteed to require the detailed data for drill-though and discovery based analysis.

Conclusion

In a perfect world with infinitely fast machines and flawlessly intelligent cost-based optimizers, denormalization would never be discussed. However, in the reality in which we design very large databases witin the insurance industry, selective denormalization is sometimes required. The key to success in deploying denormalization techniques is to treat each design choice as an engineering decision with a quantification of benefits and costs. Keep in mind, as well, that machines are getting faster and database optimizers are getting better by the day - so try to design toward the future with a minimum of compromises to the logical model of your business in order to preserve the quality of data and query flexibility in the physical design.

Stephen Brobst is an internationally known expert on high-end data warehouse implementations. He performed his masters and Ph.D. research in parallel processing architectures at the Massachusetts Institute of Technology and teaches the high-performance data warehouse design course at TDWI. Brobst can be reached at sbrobst@alum.mit.edu.