Separating Data and Processing
or
Building Databases for Systems Yet to Come

Matt Ohmes
Cognos Corporation
2301 East Lamar Blvd. #416
Arlington, Texas 76006

Introduction

This paper had a long and difficult birth.

Like most programmers during the last 10 years, I had been told of the glories of "normalized" data structures. But also, like most programmers, I did not really know WHY it was important to normalize. It was simply accepted as a theoretical maxim that "good" database designs were normalized. It was also accepted as common sense that "real programmers" didn't normalize; real programmers optimized!

There did not seem to be a practical advantage to normalization. It made it more difficult to program, maintain, and fine tune applications. On the other hand, if you just made a few adjustments to the database, here and there; things got so much simpler.

Since I considered myself a "real programmer", I never paid much attention to all this theoretical talk. After a while though, I began to notice some problems with my database designs. Oh, they worked all right for the original application, but the designs caused nothing but problems for anything new.

Several years and a few minor disasters later, I decided to re-analyze my approach. Why was the database that was so "perfect" for one application, "perfectly awful" for the next? Finally, the light dawned; "normalization"!

I had fine tuned and "tweaked" myself into my a corner for the last time. From that point on, all my data bases would be perfect paragons of normalized form!

Several more years and minor disasters later, I realized that normalization alone, was also not the answer. It was a technique. A set of rules that tended to yield a desired result. The result is easier to state than achieve: The "ideal" database is one that allows any number of applications to effectively and efficiently access and manipulate it. To achieve this result, the designers and programmers must, as much as is possible and practical, separate the database from the processes that acts upon it.

This paper addresses many of the pitfalls that I have encountered while learning the ins and outs of database design. It also lists some rules of thumb that I use in my data structures. Hopefully it will help you sidestep some of my "minor disasters".

What is Normalization?

Since this paper is not intended to be a formal discussion of normalization, I won't spend much time on the normalization process or theory. But, I will give a general description of 1st, 2nd, and 3rd "normal form" structures.

The initial requirement for any level of normalization states that all records (or "tuples" if your relational) must be uniquely identified. This unique identifier is commonly called the "primary key" and can be either a single field or combination of fields. Note that Image Detail sets cannot have a unique key item, but each record can still be different from others in the set; through a combination of fields.

First Normal Form stipulates no repeating groups. Simply stated, this means getting rid of all arrays. This applies to both explicit arrays (eg. Item MONTH occurs 12) and implicit arrays (eg. Item JAN, Item FEB, etc.).

Actually, First Normal Form prohibits a variable number of repeating fields in a record, but this restriction is generally expanded to include any arrays. Arrays are usually eliminated by creating a new file in which each record would take the place of a single, occupied array occurrence.

Second Normal Form must satisfy First Normal Form, plus all items must be functionally dependent on the primary key alone. In other words, eliminate data redundancy. "Customer Address" should only be in the Customer file. It should not also appear in the Invoice file just because Invoices are sent to Customers. The Customer must be related to the Invoice file in some manner, but that relationship should be via the primary key of the Customer record. The goal is minimum redundancy.

Third Normal Form requires Second Normal Form, plus the elimination all transitive dependencies. Simply stated again, this means don't put vital information where all references to it might be eliminated by normal processing. For example, if all Invoice records are deleted from your system after they're paid, make sure that the Invoice file is not the ONLY place you store your customer's addresses. If a customer pays his bills promptly, you certainly don't want to lose track of him!

There is also a Fourth and Fifth Normal Form in relational theory. In practice, however, Third Normal Form is the generally accepted standard for most data processing shops. Third Normal Form is what I refer to as a "normalized" database in this paper. If you want a more thorough discussion of the normalization theory or process, there are many papers available. Three I have referenced in the writing of this paper are:

A SIMPLE GUIDE TO FIVE NORMAL FORMS IN RELATIONAL DATABASE THEORY
William Kent
Communications of the ACM
February 1983

and

SIX STEPS TO A NORMALIZED DATABASE
Paul Bass
Supergroup Magazine
May/June 1985

and

HOW TO DESIGN FOR THE FOURTH GENERATION
Leigh Solland
Baltimore/Washington RUG - Interex 1985
Paper 3013

There are literally hundreds of others available. Any that explain the process clearly are worth reading.

Why aren't most data bases normalized?

Let us assume that the average programmer or analyst can understand enough relational theory to describe a normalized data structure. Why then, aren't more data bases normalized?

The primary reason is, there is no perceived advantage to normalization. Most traditional programming languages do not readily lend themselves to normalized data structures. A Third Normal Form database does not necessarily make for more work for the average programmer, but certainly does not make for less!

Normalized data bases tend to have more files with each file having shorter records. Those files all must have separate open, read, check for end of file, and close routines. It is simply easier for the average programmer to have fewer files. Fewer files mean a lot less code in most programming languages.

It is also easier to build on familiar techniques. Even if those techniques aren't necessarily as relevant for present day computers. For example, many systems have been designed to minimize reads and writes at all costs, even though almost all systems can utilize Multi-record reads and disc caching with little or no programmer intervention. This doesn't mean that disc I/O should not be minimized. But it is not the hobgoblin it once was.

It should be noted however, that normalization does tend to penalize record retrieval. Data that might be on one file in a traditional structure might have to be read from two or more files in a normalized structure.

Why should data bases be normalized?

If it is so much trouble to use normalized data bases, then why on earth should we even bother? The answer is simple; normalization yields data structures that are stable, accessible, and flexible. The design minimizes data redundancy and maximizes consistency. If the database is designed properly, any subsequent application should be able to access and manipulate the data as well as the first.

This is the major problem with most "unfriendly" data bases out there today. Most were designed around a single application. Their structure is build to get the most out of that application. Unfortunately, most designers don't realize that the fundamental data entities and relationships for a company change very little over time. The systems that process that data, however, and the specific procedures involved, change more frequently.

This point must be clearly understood, before the advantages of normalization can be realized. A normalized database is inherently independent of its applications.

The "Danger Zones"

If you are in the process of designing a database or building an application, you may want to run down a mental check list here. There are several "warning signs" of a non-normalized database. If these signs are present, some "rules" have been violated. That does not mean the database design is "bad". It is simply not perfectly normalized. As we'll see later, there are sometimes perfectly valid reasons for breaking the rules.

Arrays

Anyone who ever took a computer programming class in school learned to manipulate arrays. They are perfect for teaching looping and control break processing without actually getting into messy concepts like files and data management systems. They are also easy to understand and control, so most programmers quickly become quite dexterous with them. Programmers are people, and people like to stick with what they know, so arrays tend to crop up in a lot of data bases.

There is nothing wrong with arrays. In fact, they are very handy for certain things. Arrays are great for "summary" type records. If I have an individual sales record for each of my 100 products for each week of the year; at the end of the year, it would be nice to archive that data on 100 records. Each record would have a 52 occurrence array, and each occurrence would have the sales for that product in that week. That seems a sensible idea, so why wait until the end of the year? Why not use the array records for daily processing?

The most obvious problem is actually the most minor; the problem of unused occurrences. If we used our example array for daily processing, over half the array would be wasted space until the middle of each year. However, unless each occurrence was quite large, the actual space "wasted" would not significant.

The major difficulties with arrays arise when we change our processing requirements. What if I decide to record daily sales figures for the 10 most active products, but only want monthly sales of the bottom 50? My array made assumptions about my processing that had nothing to do with the basic data. The data includes product, total sales, and the period of time involved. The array forced my period of time to be 7 days. Would your boss allow you design a database that would permit only 10 products or only 50 invoices? The principle is the same.

Arrays force processing assumptions on data structures that inevitably must be revised, and arrays are notoriously resistant to revision. They also tend to be exceptionally tenacious. Programmers seem willing to go to almost any length to keep their arrays (eg. "Well, when the array overflows, we add another record with the same key, except we set a flag on the second record indicating it's a duplicate. Then we increment a counter on the first record showing how many duplicate records we have ...).

A normalized data structure may force more records to be read for a given report, but it is relatively easy to restructure, and it is very processing independent.

The "Everything" Record

The next "warning sign" is what I call the "Everything Record". It is revealed by the presence of very large records in relatively few files. The general idea seems to suggest that if you put all your data into a single record, you can save a lot of time and effort opening, closing, and reading files. The chief benefits are simpler file handling routines and fewer records to read. You don't have to go elsewhere to get any information. Normalized databases tend to have more files, but each file has shorter records.

Very large data records are rarely a conscious decision on the part of the database designer. Most "everything" records just seem to grow as

applications evolve. It is a lot easier to add a new field to an existing record than to consider the implications of a new file in the overall design.

"Everything records" often suffer from a "merging" of data entities. If several fields have been added to a record to indicate a number of different client statuses, for example; perhaps a new "client-status" file should be created. That way new status codes could be created and old ones deleted without a structural change to the primary file. The resulting design would also be easier to comprehend.

It is difficult for programmers to understand very large record structures. It is hard to grasp the individual meaning and relative importance of 300 different data items in a single record. Normalization clarifies data relationships and anything that clarifies the structure, simplifies development and maintenance.

The Multi-Record Type File

The next "warning sign" is another example of "merged" entities. That is the multi-record type file. (eg. Header Records, Detail Records, Trailer records). There are few advantages to multi-record type files. The only one I can think of is a reduction in file opens. There are a lot of disadvantages!

Multi-record type files complicate programs. The programmer must check for file type, save record locations for later updates, check for different data types in overlapping fields, and a number of other irritations. Any record structure change is major problem. You can't just add a field, you must check every other record type in the file and make adjustments. Even minor changes cause major ripples.

Terrible performance problems can also result. At one site I visited, there was one Image data set, containing over 1 million records, that had 17 different record types described for it. Three record types comprised over

99 percent of file, two types had less than 100 records each, one had 5. To report those 5 records, over 1 million had to be read! The user base had little idea which record type was which. They were forced to decide if they should scrap an fairly satisfactory system or live with it, as it was, forever.

Multi-record type files perpetuate old, batch processing concepts. They come from an era when sequential files were the cutting edge of data management technology. There is little advantage in retaining this technique.

Calculated Items

Calculated fields are also considered a "warning sign" in data design. Although I have found no specific prohibition against stored, calculated values in relational theory, they are generally rejected in normalized designs. If calculated values are stored in the database, their accuracy is always in question. This restriction is usually rather loose, however. If accuracy of data is vitally important and processing time is relatively unimportant, calculated fields should be avoided. If processing time is of paramount importance, calculated fields are well worth considering.

All the "warning signs" listed above fall into a broad category I call "language specific designs". They result from applying application programming techniques to data structure design. When programmers learn how to "code around" unusual data designs, they typically add those techniques to their "bag of tricks". Unfortunately, they also tend to add the unusual data designs to that mental bag too. This perpetuates "unfriendly" designs. It's like the old saying: "When the only tool you have is a hammer, all your problems start to look like nails." Don't repeat a questionable design just because you know how to program around it.

File Specific Application Designs

Another broad category of "warning signs" is "file specific application designs". This is the opposite side of the design coin. In this case it is the application that suffers because the developer overuses certain file features. For example, if an entire application is built around the generic retrieval capability of KSAM, future systems may suffer.

A more common but less obvious example of a "file specific application design" involves item level locking in Image. Item level locking has widely been described as an important component in "strong" application locking. While Item level locking may be useful in a single application, it is an invitation to misfortune in the long term. Applications that rely on item level locking to achieve acceptable performance, are extremely fragile. If a subsequent application does not follow exactly the same locking rules as the first, Image resolves the conflict by using the broadest locking level requested. This could easily cause an existing application to suddenly become unacceptably slow.

For a more complete discussion of locking strategies and pitfalls, I would suggest reading Chapter 15 of the Image/3000 Handbook, "Picking the Lock". A general rule can be applied; don't design your database around the way you code, and don't code around the "neat tricks" in your file management system.

When should you "break the rules"?

As I mentioned earlier, the "warning signs" indicate some "rules" have been violated. But, there are valid reasons for breaking those rules. If performance concerns become overwhelming or a change will greatly simplify the overall processing, then bend the rules just enough to get by. Don't assume that all normalization should be thrown out the window just because one report runs slow. If possible, wait until the user base has a chance to exercise the system. They will show real bottlenecks, as opposed to predicted ones.

I know of at least three examples, from personal experience, where violations of the normalization rules dramatically enhanced the existing system, without compromising the overall design.

One site I visited did not have any calculated or summary fields, even though they had hundreds of thousands of transactions. Even the simplest of calculated reports took hours. Since day old data was accurate enough for management reports, several summary files were constructed. In each summary file, thousands of records were compressed into a few summary records each night. The reports then ran almost instantly against these shorter files the next day.

Another site had a system that was almost totally table driven. They had over 30 different types of "lookup" files in which the records all had the same format; "code-value", followed by "description". They were all put into a single multi-record type file to create a "table of tables". This kept the overall structure cleaner and made adding new table types much simpler.

The third site (a well known software vendor, very familiar to me) had a database in which each Customer was related to one or more CPU records. The number of CPU records per Customer was not originally keep on the Customer record. It was soon discovered that every Sales Rep wanted to instantly see the number of CPUs each Customer had. Rather than force that number to be counted on each inquiry, it proved to be more effective to count CPUs when they were added, store that count on the Customer record, then have a batch process re-count periodically for "insurance".

In all three instances, the data designs were "de-normalized" to a limited degree to aid in processing. But, none of the sites had to corrupt their basically sound designs; and all of the sites have added new systems on top of their original databases with little or no problems.

Conclusion

In the final analysis, subsequent systems are the measure of a database design. If the second application runs as well against your database as the first, then you have a strong, stable, flexible design.

Data (the database) and processing (the applications that use that database) are and should be separate. Processing is based on the needs of the user at the moment. Data reflects the information needs of a company over time. With accessible data and flexible data structures, MIS can plan for the needs of systems yet to come and keep the users satisfied. And that is, after all, our job.