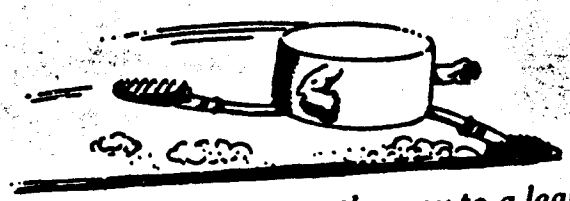
The world of database design will be explored, step by step, in this 14-part series, Process-Driven Data Design.

LESSON ONE: DURABLE, DOABLE DATABASES

14 STEPS TO A FIRMER DATABASE



Frank Sweet will show you the way to a leaner, meaner, more flexible database.

by Frank Sweet

Have you ever noticed that there are just two kinds of data entities in the real world? Imagine, say, a 10,000-record vendor file where, on average, 100 new vendors are added each month and 100 inactive ones purged. Its monthly volatility (turnover divided by population) is 1%. Now consider a file of 1,000 purchase orders where 1,000 orders are added each month and the same number closed. Its volatility would be 100%.

Compute the volatility of the permanent files in your dp shop and you'll uncover something strange; there are two populations out there—two classes of records. A histogram shows two peaks: one around 1% like our vendor example, the other near 100% like the purchase orders. Volatility is binary, it seems, and almost every record falls into one group or the other. Why is this? Does it apply to every database? Does it reveal an important underlying truth about data?

The answer to the last two questions is yes. But to find the why, we will journey into the world of database design. We will see that objectively good designs exist, and we'll learn to recognize their traits. We'll review equations in data flow dynamics like the above, and mnemonics like $K = DU^3$. We'll manipulate the boxes and arrows of Bachman diagrams to derive, split, and merge entities. We'll examine six useful design patterns we can apply to many applications. And we'll return with a checklist that, applied to a design, tells us if it's ready to be built.

In all, we'll cover 14 topics in the next few months:

- Durable, Doable Databases—the marks of a sound design
- Data Flow Dynamics—the mathematics of data movement
- Objects and Events—the two populations of records
- Keyfield Design—dataless, unique, unambiguous, unchanging

The secret to making a database design doable is to keep it simple, and this implies modularity.

- The Two-Headed Arrow—many-to-many relationships
- The Headless Arrow—one-to-one relationships
- Optional Arrows—splitting entities
- Hard Sets, Soft Sets, and Summary Fields—integrity vs. ease of use
- The Identifier and Its Name—the most-used design pattern
- The Past Event—when audit trail is mandatory
- The Future Event—productive skepticism
- The Template—cookie-cutter records
- The Self-Related Record—bill-of-material structure
- Numbers, Keyfields, and Real Things—a completion checklist

The hallmarks of a sound database design are that it is durable and doable. A durable design survives changes in the business environment. Managers, policies, products, and styles all come and go. But a database represents millions of dollars' worth of painstakingly collected data about our firm and we'd rather it didn't come and go with them. A doable design is one that's easy to implement. It capitalizes on the one gift we all share (that we get better at anything each time we repeat it) and doesn't penalize our common flaw (we never get something right the first time).

Durable databases can survive the very applications that created them. I recall a shop that has been through two payroll systems, a now-defunct workmen's comp system, and two security systems in the past six years. Yet their employee database, which those applications used, has been chugging away uninterrupted since it was first installed. At the other extreme lies the painful case of a design so dependent on transitory management style that it was unknowingly wrecked by the stroke of an executive pen mere weeks after implementation. Understand, durability is no accident. We deliberately build it into designs by modeling underlying business reality and by making sure the result is shared among applications and can be expanded with new data.

REFLECT REAL EVENTS

Modeling reality means that our data structures reflect real events, real activities. They must not simply mimic records in other filing systems such as forms and documents. Forms can disappear without a trace, while products, vendors, customers, and employees cannot. We model reality by letting our designs be determined by the business processes they will support. We call this process-driven data design, and we will return to how it is done in a moment.

Sharing it among applications means that different people use the same database for different purposes. Avoiding redundancy, DBAs call this, and there are three reasons why it's important. First, redundant data implies redundant data updators, and it's pointless to have two people doing the same job. Also, the more users, the greater the incentive to keep it accurate, which means the data are more timely and reliable for everyone. Finally, the more widely a database is shared, the more durable it is. Its users hold it steady despite our yearly reorganizational tempests.

Making it expandable so new data can be added means developing the skills and tools to add new fields to existing records, new records to existing files, or new files to the database without shutting down or retrofitting existing applications. Expandable databases are more durable because each new application brings new data needs. Application developers will use the central database if their needs are easily accommodated. Otherwise, each group could go its own way, and the centrally shared data pool would be stillborn.

The most doable database designs are those that can be brought up faster and with less effort than comparable flat files. The worst are disasters where the application quickly reaches 90% complete and stays there for months. There are two secrets to making it doable: include only what we need, and keep it simple.

Data design conceals a treacherous twist to the dilemma: do we want it right or do we want it Friday? It's the urge to include too much. Designing a vendor file for a payables system, we conclude that we need vendor ID number, name, address, and amount owed. We should stop, but temptation draws us on. We're designing a database to be shared by future applications, we reason. Shouldn't we find out what they'll need and include it too? The bait looks tempting—doing a thorough job— and the risk looks slight—a few days' extra work. But, to do it, we'd have to identify every data element and every vendor attribute that any user could ever want for any conceivable purpose. In short, it becomes an endless undertaking.

The power of database management systems is that they enable records to be stretched to include new fields as new applications arise, and do so without making us track down and recompile existing programs. Consequently, the first way we make a database design doable is to include only what we need at the time we design it.

The second secret of doability is to keep it simple, and this implies modularity. There are really only a handful of basic database design patterns. We use them like building blocks, in one application after another. We build designs out of them rather than deriving each application from scratch. We'll introduce these patterns in the ninth installment. We'll start, though, next time with dataflow dynamics. ●

Frank Sweet is corporate manager of data administration for the Charter Co., Jacksonville, Fla.

We continue our exploration of database design in part two of this 14-part series, Process-Driven Data Design.

DATA FLOW DYNAMICS

by Frank Sweet

Data flow through files like water in an irrigation system. Some pipes carry much volume, others less. In parts of a database, records race along in a rush while in other places they pool into reservoirs with little perceptible movement. Yet everywhere they follow the rule,

POOLTIME × FLOWRATE = POPULATION.

In other words, if we mentally isolate a section of a system, we'll find that records follow a simple, steady-state law: the length of time that records spend in any section, multiplied by the rate at which records flow into the section, is numerically equal to the number of records currently in the section.

Database designers must know how to derive and use the consequences of this law. It can be applied in all steps of our job, from initial user interviews to physical file design. In a moment, we'll illustrate this with a sample problem, but first let's examine what we mean by steady state.

Steady state means that the flowrate of records entering a section is equal, over the long pull, to the rate at which they leave. Say we have a file where 1,000 new records are added each month. The file is in a steady state if, over a long period (a year, for instance), an average of 1,000 records is also removed from the file each month. Some records might be removed soon after they're added, while others could stay on file forever. Nevertheless, if 12,000 are added each year and approximately 12,000 removed, the file fits our definition. With few exceptions, every system we'll study is in a steady state. Look at it this way: if more records were removed each month than were added, the file would soon disappear entirely, thus reaching a steady state with a population of zero. If the reverse were true, it would quickly reach the capacity limit of the medium on which it's stored.

Another aspect of steady state is that the section we study contains no unidentified data sinks or data sources. Consider a credit-checking procedure where credit applications are funneled into a department for review. If we know that their total backlog is more or less constant from year to year, we conclude that, on average, just as many emerge each month as are sent in. The only way it could be otherwise would be if somewhere in the department applications were being destroyed, never to be seen again (data sink), or if someone in the department were producing new applications internally (data source). Both are unlikely.

If the number of records-out always equals the number of records-in, why did the formula use flowrate into a section, rather than out of it? Either is valid, but records-in (new records added) are usually easier to measure. A typical vendor file, say, receives new records whenever the firm first deals with a new supplier. Inactive vendors are commonly purged once a year. If we're researching the system, it's often easier to estimate how many new vendors are added each month than to find the purge rate. And unless the firm is withering away or in a state of uncontrolled growth, we can be confident that, over the long haul, the number of inactive records purged each year will be close to the number of new records added.

VALUE OF THE FORMULA

The importance of the formula is that given any two of the terms, we can easily compute the third. Given all three, we can cross-check them against each other. But enough lecturing. You now know enough to solve the following problem. It was taken from actual experience and, though the application is disguised, the numbers are authentic. It can be solved by using the flowrate formula as an instrument for applying common sense. In part three of this series, I'll present the answer I actually encountered.

The Case of the Unreserved Work Orders. As part of a materials management project (MRP, shop loading, etc.), we were designing a work order reservation system. New manufacturing jobs, or work orders, had their component requirements checked against available inventory before being issued to the shop. The idea was to avoid starting a job for which parts were missing. Instead, the job would be held while needed parts were expedited. The manual system was working well and our task was simply to automate it by checking if each order's parts were in stock, releasing the job to the shop if they were, otherwise printing an expedite list.

It was obvious we'd need a file of pending work orders—those held awaiting arrival of needed parts—and to avoid the order-number wraparound we needed to know just how long, on average, we could expect pending orders to stay on file. Investigating the manual system, we found 200 work orders scattered around in expediting, awaiting parts arrivals. Although individual orders came and went, the total pending backlog had been constant at about 200 for as long as users could recall. Further, we were told that about 40 new work orders were issued to the floor each month and that, on average, each waited for about 10 days for parts to arrive before being released.

How long will orders wait on file, on average, before expediting scares up the parts for each one?

In part three we'll offer the answer we found and explore the world of "Objects and Events." ◉

Frank Sweet is corporate manager of data administration for the Charter Co., Jacksonville, Fla.

We continue our exploration of database design in part three of this 14-part series, Process-Driven Data Design.

OBJECTS AND EVENTS

by Frank Sweet

I left us with a problem last time, "The Case of the Unreserved Work Orders." We were offered values for all three terms in the equation: POOLTIME × FLOWRATE = POPULATION. We observed 200 orders awaiting parts and were informed in interviews that 40 new ones were issued each month and that their average wait time was 10 days. The question was, "What is the average pooltime?" The answer is five months, no less.

Our challenge, to estimate the average wait time, involved deciding whether to accept the numbers as given or to cross-check them against one another. The interview-given pooltime, in months (0.3), times the flowrate (40) is by no means equal to the observed population (200). The three figures we collected (flowrate, pooltime, population) are contradictory. We must decide which one is wrong.

We observed the 200 orders sitting around—that number cannot be challenged. The 40 new orders per month would be easy to verify, thus unlikely to be misstated. Besides, what motive would there be for understating it? It's more likely to be overstated ("See how hard we work"). By elimination, we come to the 10-day wait time. The number is suspect for two reasons. First, our verifying it would seem impossible. Second, it's the sort of thing higher management puts into goals or management objectives ("Orders must be processed within 10 days"). When this happens, many will unconsciously adopt a convenient fiction rather than admit an unpleasant truth. Discarding the fishy 10-day datum, we compute average pooltime to be population (200) divided by flowrate (40/month), or five months.

Incidentally, in the real case, our curiosity was so aroused by the five-month delay in getting jobs onto the floor that we launched another study just to find the cause. This eventually led to a vendor quality/performance history system.

An even more important number in design is a record's volatility, the reciprocal of pooltime. It's defined as: VOLATILITY = FLOWRATE ÷ POPULATION. Earlier, we pointed out that volatility is binary and nearly every type of record falls into one of two monthly-volatility groups: 1% or 100%. The underlying reason is that we computerize data about two classes of real-world phenomena: objects and events.

Objects are tangible things that exist independent of time. Vendors, customers, products, employees, warehouses, and ocean-going freighters are all objects. The volatility of their records is low—around 1%. Often called "base data" or "master files," involatile records store reference information such as address, location, and name or description. Objects, in other words, just sit there and don't do much.

Events are happenings; each occurs at a specific instant. Freight deliveries, shipments, labor charges, receipts, and disbursements are events. Their volatility is high—around 100%. Called "transaction data" or "permanent work files," they keep track of what's going on out there. They hold fields like date (when did it happen), responsibility (who did it), cause (why), and the like. Events, in other words, keep happening.

It's vital that we recognize which boxes (records) in our Bachman diagrams represent involatile objects and which reflect volatile events. The distinction is so important, in fact, that I once proposed using different data-diagram symbols for the two—rectangular boxes for objects, and hexagonal ones for events. Sadly, I cannot draw a nice-looking hexagon, and the technique never caught on.

But distinguishing between classes of entities is not difficult. The computation is straightforward and, after doing it a few times, we can recognize them at a glance. We notice, for instance, that the fields—shipment-date, order-date, and patient-admission-date—make sense in that the words convey clear images. Employee-date, freighter-date, and vendor-date, on the other hand, are somehow unsatisfactory and incomplete. This is because the essence of an event is that it occurs at a specific instant while an object is not so transient.

SEPERATE OBJECTS, EVENTS

There are three reasons to distinguish between objects and events. First, we can derive events from objects but not vice versa. This means that in constructing a long-range plan—an overall database architecture—for a pool of shared data, we begin with the easily identified object records and then derive events by Bachman manipulation.

Second, shared files imply standard data names and, most especially, standard keyfield formats. We sell both ideas more easily if we first apply them to involatile reference data, rather than to less widely familiar events.

Third, how we physically implement a design into our database management system depends on each record's volatility in many ways. Multiple access paths, backup/recovery procedures, and migration techniques, all vary with volatility.

Two warnings: First, don't confuse records in another system with objects. Objects have an external reality while records model either objects or events. We would err, for example, in considering a purchase order record involatile on the grounds that it models a physical thing—a document. That document is simply another form of record which, in turn, models an event—an agreement to buy something. When in doubt, compute the volatility. Second, don't be overly strict in interpreting "tangible" reality. Nations, sales districts, and colors aren't strictly tangible, yet they would be involatile object entities in a database. Again, compute volatility when in doubt.

Next time, we'll investigate keyfield design. ◉

Frank Sweet is corporate manager of data administration for the Charter Co., Jacksonville, Fla.

We continue our exploration of database design in part four of this 14-part series, Process-Driven Data Design.

KEYFIELD DESIGN

by Frank Sweet

Database design consists of modeling the business world. Each vendor record should reflect the characteristics of real vendors, and there's one data element that every vendor record has: a keyfield or vendor ID number. After all, that's the way you tell them apart. A moment's thought reveals that keyfields should be dataless, unchanging, unambiguous, and unique. A handy way of remembering this is $K = DU^3$.

Keyfields should be dataless. "Vendor number?" asks our user. "Well, it should contain region code, industry type, size code, purchasing-agent code...." Why do some users insist on embedding clusters of data into ID numbers? Few design issues degenerate so swiftly into ultimata and recriminations between designers and users. There are explanations as well as solutions.

But first, examine the phenomenon itself. It leads to two major problems: the first is nature's 90-10 rule, the second is that data do change.

Nature's 90-10 rule says that 90% of whatever occurrences you're measuring are produced by 10% of the population. If you insert classification schemes into keyfields, you'll run afoul of the 90-10 rule. For example, consider the Case of the Cargo Tracking System:

The application kept track of ocean freighters carrying food products around the world. Each record represented a shipment with such data items as ship name, cargo quantity (in tons), and value (in dollars). The record's four-part keyfield was designed as "CF126523," where the first two bytes indicate the type of cargo from a table of a few dozen types. ("CF," by the way, means "coffee beans.") The next two bytes show the port of origin from a table of seaports, and "12" means Santos, Brazil.

The third slice of keyfield shows the shipment's destination port, from the same table of seaports; "65" means Jacksonville, Fla. Finally, there's a two-byte sequence number. In the example, it means that shipment "CF126523" is the twenty-third shipment on file carrying coffee beans from Santos to Jacksonville.

The 90-10 rule simply says that most coffee is shipped from Brazil (or somewhere in South America) to Jacksonville (or somewhere where there's a roasting plant). Packaged foods, on the other hand, leave processing plants outbound for distribution sites.

This yields two consequences. First, many possible numbers will never be used. Just as you wouldn't bring lobsters to Maine, no one would send coffee beans to Brazil. Second, and more important, is that in the normal course of business, hundreds of shipments carry them from Santos to Jacksonville.

In other words, though there were unused gaps in the numbering scheme, the heavily used sequences (e.g., "CF1265...") soon exhausted all possible numbers. The system had not been up two years before there were 99 shipment records, most of only historical interest, from Santos to Jacksonville. How did we code the hundredth one?

There is no clean solution. The two answers usually proposed for this problem are to make the sequence-number portion of the keyfield longer or to add duplicate entries to one of the tables. The former idea soon foundered on the grim reality of changing the length of a master file's keyfield. It simply could not be done without retrofitting the entire application and its associated administrative procedures. In short, it would have taken too long. If the system were to survive, that hundredth shipment had to be recorded immediately.

We adopted the latter idea. We added code "CO" to the cargo-type table with the same meaning as "CF." This postponed the day of reckoning for another two years. It also transformed the system into a maintenance nightmare. Think of all those summaries by cargo type that could no longer be produced by sorting the records.

The 90-10 rule problem has nothing to do with the food industry or even with event tracking. It is most often found in part-numbering schemes for spare parts in heavy industries like steel or chemicals, or for components in manufacturing. It's incredible how many different kinds of nuts and bolts there are.

2 CHOICES, BOTH ARE WRONG

We see the second pitfall, the unchanging keyfield, by considering what happens when a shipment is diverted. Shipment CR126523 is on its way when dispatching decides its Florida-bound cargo is more urgently needed in Baltimore. But destination-port is part of the record's keyfield. When a shipment is redirected we're faced with two choices—both wrong.

If we leave the shipment's ID number unchanged, its destination-port code becomes inaccurate. Not that it is redundant, inconsistent, untimely, or anything less. It is simply flat-out wrong. Anyone who uses this information will be misled. Worse, it's one of the most important pieces of data in the record. Why else did our user want it in the keyfield in the first place?

If we change the ID number we lose the shipment's audit history. Understand, most data are not in computer files. Shipment numbers are in letters, telephone note pads, contracts, scrawled on the backs of envelopes, on countless 3-by-5 file cards, and in the skulls of users everywhere. If we

Ideally, a record's keyfield should be little more than a meaningless serial number.

change ID numbers on file, the lion's share of our records (the noncomputerized part) will be wrong.

Visualize the consequences by imagining a user who's been monitoring a ship's progress across the ocean. One day, inquiring about it with the old ID number, she's told that no such shipment exists. Put yourself in her shoes: she knows it was out there yesterday, and she knows it didn't arrive anywhere today. Have you ever wondered how that business about the Bermuda Triangle got started? Now you know.

Notice, by the way, that if we bring up this issue during design, the user will inevitably say, "Oh, but those data [whatever they are] will never change!" Now, if you believe this, get in touch with me—I have a Caribbean island I'd like to sell you.

The solution is to keep data out of the keyfield. Fields in database records come in two flavors: identification and description. ID numbers identify; data fields carry data. The keyfield identifies the entity that the record models (a shipment), while all the other fields describe its attributes. Put origin, destination, cargo, etc., in the body of the record. If users want output sorted by these fields, do it. If they want to retrieve on-line using them, let them. But keep them out of the keyfield. Ideally, a record's keyfield should be little more than a meaningless serial number.

THEY DON'T TRUST US

Why do some users insist on embedding data into keyfields? It's because they don't trust us. Remember your last payables system? The vendor master was up and used by hundreds of programs when the user wanted to add "vendor industry type" to it. Since the record's filler had been consumed years before, you realized you'd have to make the record longer. To do this, you'd have had to track down, modify, and recompile all those undocumented programs. So you told him that it would take six months and cost tens of thousands of dollars. He withdrew the service request, but he still needed the data, so he simply assigned blocks of ID numbers to the different industry types. As the years went by, he got into the habit of structuring ID numbers with data and has never stopped.

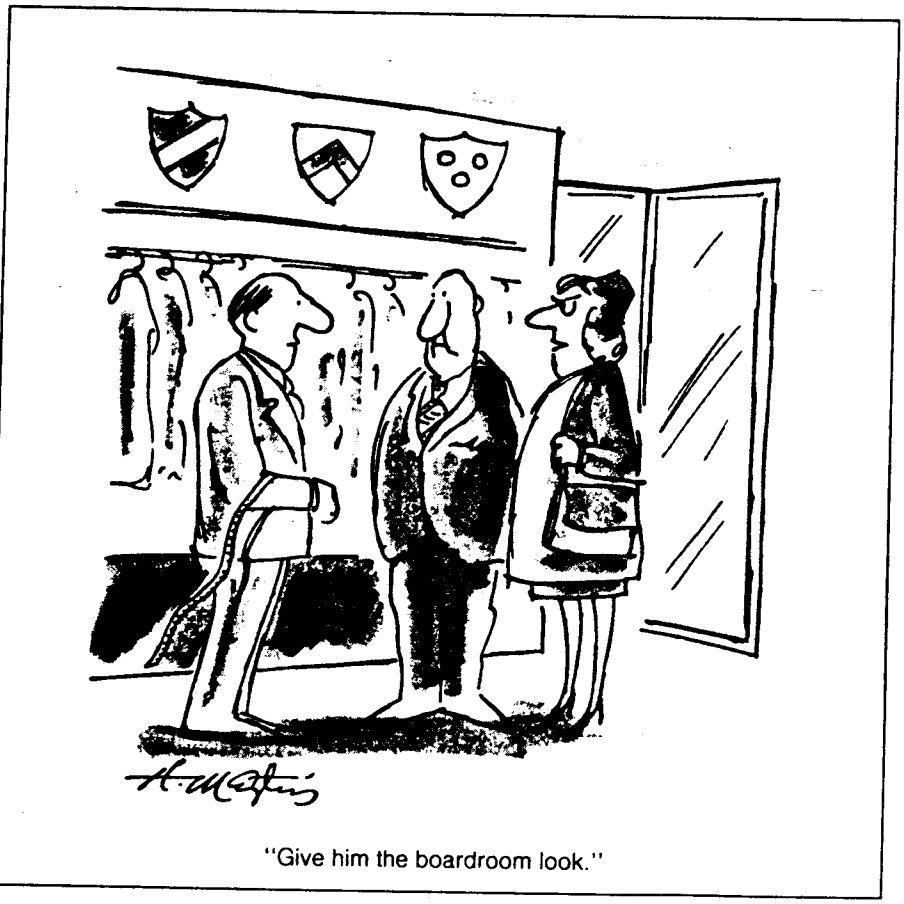
The solution centers on our ability to react quickly to the changing business environment, adding new items that can be used as secondary access keys as needed. With today's tools, we can provide this level of service. But modern tools have their own drawbacks. Each keyfield, besides being dataless and unchanging, should be unambiguous. In other words, you should not have different real world entities (two employees, for example) on file with the same ID number.

The problem emerges when we misapply technology. Older data access methods (ISAM, VSAM) made it difficult or impossible to write multiple direct-access records with the same key. Today's database packages allow it, but it's still unwise.

One way the problem arises is through shortsighted choice of ID number. Case history: the Personnel File. Everyone agreed to use social security number as employee ID number in the new system. The application was installed and turned over to the client before someone asked what to do about Brazilian employees (they're the ones who ship all that coffee). Brazil is a sovereign nation and its citizens aren't issued U.S. social security numbers. The proposed solution was to invent a number (999-99-9999) and use it for every Brazilian. The user's question was, "You mean the database won't let me have two employees with the same ID number?" Our too-truthful answer was that the package would, in fact, allow such an aberration. Needless to say, the file is now a bit tricky to update. It also produces interesting telephone conversations between personnel managers: "Not that 999-99-9999, Harry, the other 999-99-9999!" (accompanied by a great deal of arm waving).

Finally, keyfields should be unique. You shouldn't have two records (with different ID numbers) for the same vendor. The problem arises when each of two different user organizations wants total authority over the same file. They compromise by splitting the range of possible ID numbers between them. Each maintains the records within his or her range of ID numbers. But since nobody coordinates new vendor numbering with anyone else, common vendors get two numbers. The results are interesting but not very safe. Several years ago, my then-employer got stuck with a $150,000 bad debt from a deadbeat corporation. The offender's record was immediately tagged as "NO CREDIT-BAD RISK" by our credit department. Three weeks later, they stuck us again for $200,000. Same outfit, different corporate ID.

Next time, we'll examine "The Two-Headed Arrow." ◉



"Give him the boardroom look."

Frank Sweet is corporate manager of data administration for the Charter Co., Jacksonville, Fla.

We continue our exploration of database design in part five of this 14-part series, Process-Driven Data Design.

THE TWO-HEADED ARROW

by Frank Sweet

Bachman data-structure diagrams consist of boxes and arrows. The boxes represent data entities or types of records. The arrows depict relationships between records. Each arrowhead marks the "many" end of a relationship.

Bachman diagrams are useful to designers for two reasons. First, they pack so much information into a succinct, easily reproduced form. A few lines sketched on a blackboard or notepad replace tedious, easily misunderstood explanations. Also, we can manipulate the symbols, like the terms of an equation, to derive a detailed conceptual database design from a high-level summary. Diagram manipulations let us conclude things about our design. We can test it for consistency and examine alternatives before spending days and dollars physically building it in our shop's database management system.

The three basic diagram manipulations are these: two-headed arrows produce intersection entities, headless arrows will merge entities, and optional arrows will split entities. We'll cover these rules and others in the next few issues.

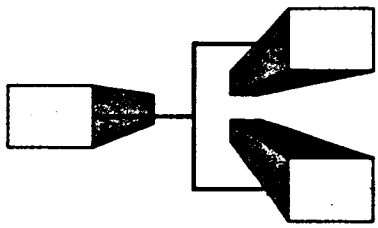
First, though, consider the symbols themselves. Boxes model data entities, the things about which we'll store data. They are important because once implemented, they become different types of records in the database. Arrows model interrecord relationships. They are important for three reasons.

First, they embody referential data integrity. In other words, a "vendor" box pointing to a "purchase order" box means that we must not store a new purchase order unless it is associated with a valid vendor. Neither should we erase a vendor as long as it has purchase orders associated with it.

Second, arrows show important access paths (predefined JOINS, in relational terms) that the finished application will use (for example, given a vendor, find its purchase orders). Finally, many database management systems use disk address pointer chains or arrays to relate records to one another. Conceptual relationships are a starting point for defining these physical relationships.

Here are six sample Bachman diagrams. The first three are not valid in a finished conceptual database design.

No arrowhead. Which of the two is the "many" end? We'll talk about that next time.



Conceptually meaningless, although this can be implemented in some database management systems.

This one is conceptually meaningful but theoretically impossible. This situation does not occur in real life. We will discuss it in a moment.

The next three, though bizarre, do make sense.

Quite common actually, a bill-of-material structure.

Weird but legitimate. I've seen only one like this. It belongs to a large-city rescue squad's mapping database.

Also common in real life—each organizational unit reports to one and only one other unit, but each may have several units reporting to it. It is not directly implementable in most database packages.

Let's examine the first diagram manipulation: the two-headed arrow and the missing intersection. A two-headed arrow means an entity is missing from our design. There's a record out there that we must identify and include before translating conceptual into physical design.

The above figure tells us that the relationship between the two records is not simply one of header detail. Since an arrowhead is the many end of a one-to-many relationship, twin arrowheads don't tell us which is the many end. Say we wanted a part-number catalog master file as well as a file of purchase orders. Obviously, purchased parts and their purchase orders are related in some way, but where does the arrowhead go? One PO can include many different parts, putting the arrowhead on the right. But wait, any one part can be included in many different purchase orders. This means that the arrowhead goes on the left.

Our problem is caused by lack of an entity. The many-to-many situation indicates that we are missing a record. There is a box, a thing about which we need to keep data, that we have not yet identified.

Think about it. The PURCHORD record carries data about a purchase order (independent of what individual items are in the PO). The PARTNUM record holds a part's catalog data (regardless of any POS that exist). Where does "quantity ordered" go? Not in PARTNUM, because any one part could be ordered in different quantities on many POS, but not in PURCHORD either: a PO can include different quantities of many

Conceptual relationships are a starting point for defining physical relationships.

parts. We are missing the record for a purchase order's line item. Therefore, the diagram would look something like this:

Each purchase order may include many line items, but each line item is on only one PO. Similarly, a part number can be on many line items (in different POs), but each line line item is for only one part number.

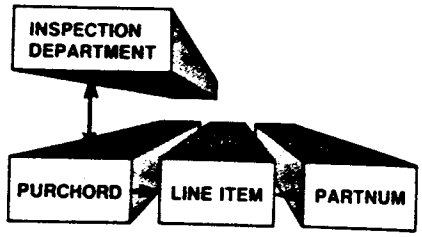
In short, a two-headed arrow is meaningful because it indicates a many-to-many relationship, but it is a symptom that there's an entity missing. A hidden record exits which, when found, forms the intersection of the two records already identified. When you find a many-to-many situation in file design, feel free to tag it with arrows at each end. Remember, though, it means we are not finished. Find the missing intersection record, break the arrow in two, insert the intersection, reverse the broken pieces of arrow, and the two-headed relationship will vanish.

One last thought: whenever a two-headed arrow is replaced with a new intersection entity, review the other relationships in which the original parent entities were involved. Look at each arrow pointing toward the parent boxes and consider its meaning. You may find it is more appropriately drawn pointing to the newly formed intersection box than to the original parent box. The relationship still carries the same meaning, but moving it to point to the new box makes the diagram more precise.

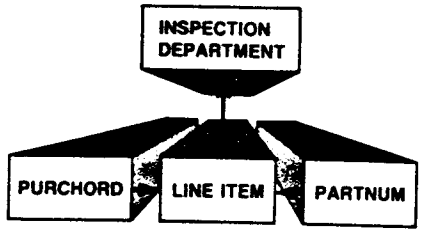
Each inspection department can be involved with many purchase orders. The fact that their involvement ("inspection date," "quantity rejected," etc.) is at the level of individual line items can be revealed in three steps.

Start by identifying the many-to-many relationship.

Identify and draw the intersection entity.



Reconsider the relationship between inspection department and purchase order.



Next time, we'll review the second basic manipulation, merging entities with the headless arrow. ◉

Frank Sweet is corporate manager of data administration for the Charter Co., Jacksonville, Fla.

We continue our exploration of database design in parts six and seven of this 14-part series, Process-Driven Data Design.

HEADLESS AND OPTIONAL ARROWS

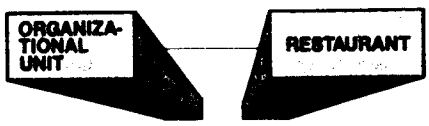
by Frank Sweet

Bachman diagrams show relationships (arrows) among information entities or records (boxes) in conceptual database design. Now, symmetry seems to demand that real-world relationships should come in three flavors:

The idea is so appealing, in fact, that to say only the one-to-many flavor really exists seems to imply an irrational universe. Yet this is the case. Douglas Adams, author of *A Hitchhiker's Guide to the Galaxy,* theorizes that if anyone should discover the purpose of the universe (what it's good for), it will instantly vanish and be replaced by something even more bizarre and inexplicable. I believe this has already happened. In conceptual database design, only one-to-many relationships actually exist for long. The other two are simply intermediate design steps that must be resolved into one-to-manys.

We looked at the many-to-many relationship last time. We showed that such a two-headed arrow tells us we are not finished. It points out that there's an intersection record missing which, when identified, resolves the two-headed arrow into two one-to-many relationships.

The one-to-one headless arrow tells us that there are too many boxes present. Consider the following design modeling a restaurant chain:



Every restaurant is also an organizational unit of the firm. Moreover, each one is only one such unit. Not all units are restaurants, of course; there are also offices, warehouses, districts, and so on. But for every unit that is a restaurant, it is only one restaurant. Some units, like districts or regions, may have several restaurants reporting to them but these aren't the same thing as the restaurants themselves. We first model the situation with a one-to-one relationship, and then consider: are retail stores and organizational units really different entities? No. The terms used are merely context-dependent names for the same class of real-world tangible objects.

The general rule is that when we find a one-to-one relationship between two boxes, we replace them with just one box. The headless arrow means that we are really dealing with one entity. The final criterion, based on normalization, is that if every data element in both boxes can be uniquely and unambiguously determine[1] by the key-field of either box, then you really just have one record type.

For example, the four boxes, "CICS User," "Insurance Claimant," "Credcard Holder," and "Computer Programmer" are all just different views of "Employee." "Inbound Shipment" and "Outbound Shipment" just mean "Shipment."

In physical database design, we do sometimes implement one-to-one relationships in order to conserve core, disk-space, I/O, or cpu cycles. Consider an employee record with fields for executive stock options and bonuses. Since these data apply to only a few employees, the record would have much empty space for most personnel. Wasted disk space at $2 per megabyte per month (3350 rental) is not as costly as it once was, but it's still irritating. We could compress the employee record, but that costs cpu cycles and makes restructuring more difficult. A cleaner solution is to hang a smaller record off it, to hold the fields that apply only to executives. Only an employee record that needs those fields would own such a subordinate record and, at most, it would own just one.

As another example, Ken Thorn, of Giant Food Inc. in Washington, D.C. asks, "What's the relationship between 'states of the union' and 'governors of states'? Clearly, they bear a one-to-one relationship with each other yet they are not the same entity (one is more organic than the other)."

Part of the problem yields to better definition: if it's an American history database, for example, where we store political biographies, a one-to-many relationship is revealed. Each state, since its entry into the Union, has had many governors. If, on the other hand, we only mean to capture data about current officeholders, then normalization tells us that they are truly the same information entity, despite intuition.

But now we're sailing the treacherous shoals between theory and reality. For, even if our users earnestly promise that all they'll ever want is current data, experienced database designers would still make them separate physical records. History-keeping (audit trail) is eventually required by almost every application and it would be easier to add it if the volatile portion (elections in this case) were separate. That way, you could add an effective-date field to the governor record and keep historical occurrences alongside the current one.

But these one-to-ones are made to fit the conceptual design into the limitations of our software, hardware, or planning ability. In a theoretically perfect conceptual design, neither the headless arrow nor the two-headed arrow would exist.

PART 7: SPLITTING A BOX

Now, as we begin part seven, let's look into how we go about splitting a box in two when an optional arrow appears.

We've been manipulating data structure, and operating conceptually on Bachman diagrams while designing a database. In part five we saw how to derive a new data entity (a box) with the two-headed arrow rule. Just above, in the headless arrow discussion, we showed how to eliminate a box by merging two entities into one. Both rules are reliably objective; like arithmetic, they always work.

Next, we'll inspect two more manipulations: splitting entities with optional arrows and merging those with similar relationships. These rules are more subjective than the prior two. In other words, they

We consider splitting entities with optional arrows and merging those with similar relationships.

indicate solutions that are likely, but not absolutely certain. They warn us to investigate further.

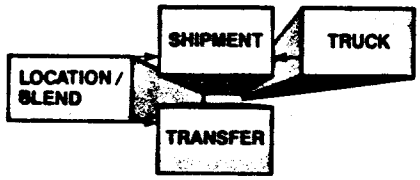
Consider the following database design for a pipe-tobacco wholesaler. Cut tobacco is stored in warehouses and distributed in trucks.

The location/blend record tells how many pounds of one tobacco blend there are at a specific warehouse. If there are 20 warehouses and 50 blends in all, we could have up to 1,000 occurrences of this record (perhaps not every blend will be stored at each location). Any blend may be shipped or received many times. Similarly, each truck may be involved in many movements. But, by definition, each movement represents an occasion where a single location's inventory of a blend was either incremented because tobacco arrived or decremented because some was sent off. Also, each movement involves only one vehicle.

The situation seems straightforward. The fields in the location/blend record include location number, blend number, inventory balance, and the like. Truck record holds vehicle number, cargo capacity, miles-since-maintenance, etc. And movement contains date, quantity shipped, and an in-or-out flag. But doesn't the truck-to-movement arrow mean each movement must involve a truck?

Think about transferring tobacco within a location. Many pipe tobaccos are produced by blending others—that's why they're called blends. The process is one of simply shoveling a measured amount from one hopper to another and, though it certainly affects the inventory balances, no vehicle is involved. The truck-movement relationship is then optional in some sense. Only one truck is involved in any shipments that go by truck, but some shipments do not involve trucks at all.

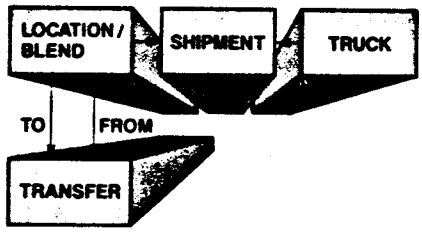
The optional arrow is a warning that the movement box, in an information modeling sense, could represent two fundamentally different entities. Redrawing it as two boxes, we have:



Shipment and transfer tell about different real-world events. As we refine the design further, their record layouts continue to diverge until, eventually, we find the only data common to both are date and quantity.

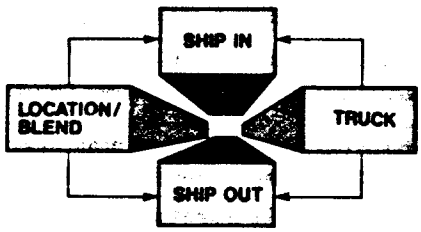
Our first indication of their duality was that optional arrow.

Now look at the arrow between location/blend and transfer. We can't say each transfer is associated with only one location/blend because the transfer event affects two different balances. It decrements the inventory balances and increments the location/blend balance it goes to.



The diagram does not mean that each transfer is related twice to the same location/blend, by the way. On the contrary, each transfer has two different relationships with two different location/blend record occurrences—a "from" relationship with one and a "to" with the other.

Merging records with similar relationships is basically the reverse of the process just described. For example:



Both intersection records, inbound shipment and outbound shipment, hold the same data elements and are related to everything around them in precisely the same way. This situation warns us to look at them more closely and see if they're not really modeling the same class of real-world event.

Next time, we'll look at hard sets, soft sets, and summary fields. ◉

Frank Sweet is corporate manager of data administation for the Charter Co., Jacksonville, Fla.

We continue our exploration of database design in parts eight and nine of this 14-part series, Process-Driven Data Design.

DATA INTEGRITY AND THE IDENTIFIER

by Frank Sweet

Database design is full of compromises. Data integrity and ease of use are fundamentally at cross-purposes. We cannot avoid trading one for the other. The problem isn't a software limitation; its roots lie in the very reasons we collect and store information. We want it to be right. We want it to be on time. We can never have both. Today we'll look at two examples of compromise, the difference between mandatory and optional interrecord relationships—hard sets vs. soft sets—and the desirability of summary fields.

Consider the following Bachman data structure diagram where we may have any number of invoices on file for a vendor, but each invoice must be related to only one vendor.

There are two ways to physically implement such a relationship between records (called a "set" in Codasyl-COBOL) in most databases. One is where each vendor record is the starting point of a chain of disk address pointers. (The vendor record points to the first of its invoices, which points to the second, etc.) In the other, each invoice simply contains its vendor number as a data field and pointer chains are not used.

Both approaches enable a program to retrieve the vendor record, given an invoice. With the first, it would read the invoice record and then use the database management system's obtain-owner or get-parent command. Under the second, the program would still read the invoice first, but then it would move vendor number from the invoice record to the vendor record's direct-access key and read it directly.

Both approaches let us extract all the invoices for a given vendor, although the pointer chains do it more efficiently. Here, the program would read the vendor file, then use its chain-following command (get next within parent or obtain next within set). Without pointer chains, the program would simply read every invoice in the file and selectively process only those that contain the given vendor number.

Where the efficiency of given-a-vendor-get-the-invoices activities is unimportant, the major difference between the approaches is in the strictness of data validation that the database management system can provide. "Hard set" is where we use the DBMS itself to guarantee data integrity—to ensure that each invoice is related to a valid vendor. "Soft set" is where we build integrity checking into the application. Pointer chains enable hard sets.

With a hard set, each invoice must be connected to a valid vendor. The rule is simple and strict. Look at four sample situations:

- Initial oversight. Imagine that a programmer fails to follow specifications or the specs don't say vendor number must be checked before storing a new invoice record. The DBMS would refuse the command and return an error code unless a valid vendor had been accessed first.
- Subsequent maintenance. Say postimplementation maintenance requires that users have the ability to change the vendor with which an invoice is associated. Again, if the changed program neglects to check the new vendor's validity, the DBMS would disallow the operation.
- Owner deletion. Let's look at it from the other angle. If a user or program attempts to erase a vendor record from the file, the DBMS won't allow the operation if any invoices are attached to the vendor.
- User override. Perhaps the user needs the ability to put invoices into the system before assigning them to a vendor (maybe identifying the vendor takes several days). The DBMS will not allow it.

Don't misunderstand. We aren't saying that any particular database package compels this degree of strictness; none does. But they enable it, and we're describing what happens when we take advantage of the capability.

DANGER OF ORPHANED INVOICES

With a soft set, the application program—not the DBMS—controls data integrity. Look again at the sample situations. In the case of initial oversight, a new invoice could be stored without a valid vendor number or with no vendor number at all. In subsequent maintenance, if the update program moves a nonvalid vendor number into the invoice and modifies it, it will remain inaccurate. Without hard-set integrity, a vendor could be deleted, leaving dozens of invoices orphaned and no longer meaningful. And, if the user needs to put invoices into the system before assigning them vendor numbers, there's nothing to prevent it.

Which is best? There is no answer. The trade-off, as we warned, is between data integrity and ease of use. Look at it this way: the power of the hard set is that it enforces strict referential integrity whether analysts, programmers, or users want it or not. That is also the hard set's weakness.

Summary fields are another area of compromise. Consider a hierarchical database. Not a hierarchical DBMS—that's just another name for IMS/DL1. I mean really hierarchical. You know, with records for parishes, dioceses, archdioceses, and so forth, right on up to the Pope. Each parish record contains a field, number-of-faithful, telling how many members it has. Each bishop's diocesan record holds the same data element, enumerating total parishioners in the diocese, and so on. These are evidently redundant and are called "summary fields" because each contains the sum of the same field in subordinate records. Summary fields are sometimes useful. They are always risky.

Summary fields are often useful because they enable one program to do the tedious summarization work for many others. For example, say many programs need

Summary fields are useful because they enable one program to do the summarization work for many others.

the diocesan totals. If each were to compute the sum on its own, it would have to read all the parishes in every diocese to get the numbers it needs. Each would be more complex, thereby taking longer to write, compile, debug, test, and maintain. Each would run longer.

By using summary fields, only one program (the one that maintains the lowest-level record—the parish) does the time-consuming work. Other programs obtain the data right out of the higher-level records. This makes them simpler and faster. It also requires a bit more disk space, but 3350 space rents for about $2 per megabyte per month and isn't a serious factor.

Summary fields are risky because they are redundant. It's rather like keeping old balance, debits, credits, and new balance on the same record (any three are enough to do the job, all four is overkill). Being redundant, they can (and most likely will) become inconsistent. After a year, it is a safe bet that 10% of the records on file will carry summary totals that do not match the sum of their details. The causes are legion:

- User procedural error. A procedure says once summaries are computed, users must not modify details. But no procedure is followed perfectly by everyone all the time.
- Program bug. A program updates three parish records, goes to update the diocese record, and blows away with a data exception. Depending on how the files are managed, the summary totals may no longer match the details.
- Program maintenance. A programmer enhances an on-line inquiry program, enabling the user to update diocese records. The specs don't warn that its summary fields must be protected. They're put on the screen unprotected, and users promptly start changing them. (Specs? What specs?)
- System crash. An on-line program updates a parish record, gets ready to read the diocese record so it can increment the summary fields, and CICS crashes.

But all the causes are irrelevant. The point is that when anything goes wrong, the diocese records will be bad and the fact may never be detected.

Again, there is no absolute answer. If detail records are added or modified on-line but summary totals are only needed at the end of the month, we don't need the redundant summary fields at all. Why force every update program to read higher-level records, increment their totals, and rewrite them whenever a detail changes?

But if detail records are keyed in batches, while summaries are inquired on-line, why force every inquiry to read all the details and add them up on its own when the batch update could have done it once and for all?

Our best bet is to decide where our situation lies, somewhere between these two extremes. If we feel we need summary fields, we include them in our design. We are aware of the risk though, and make sure our users understand that inconsistencies will arise and they must find some way of detecting and correcting them.

SOME RECURRING PATTERNS

Now, in part nine ("The Identifier and Its Name") let's look into recurring patterns that appear in many database designs.

Designing databases can be curiously repetitive work. It's not only that after we've built a dozen inventory systems the thirteenth seems somehow familiar. Even working on an application where we have no prior experience we may feel a sense of déjà vu. Designs seem to take on their own lives. Useful ones appear again and again, heedless of application. Think of examples we've met: that neat way of handling a partial-name alphabetic search for a client's personnel file, reincarnated months later to handle another's vendor file. The audit trails we sketched to track money in an account are remarkably like those we later used to keep track of goods in a warehouse. Some techniques, indeed, are so reliable and widely useful that we can call them patterns in database design.

For the next couple of sections, let me show you five of my favorite patterns. They range in complexity from the simple identifier and its name examined today to the strange, labyrinthine, self-related record we'll meet in part 13 of this series.

Some are so widely applicable to information storage problems that they apply even to paper files. Others are limited to database management systems. What they have in common is that we've encountered them so often, in so many guises, they now seem like old friends. We can rely on them to do the job. Often, they will warn us when we miss something vital. More important, they help us listen to our user with productive skepticism. But more about this later.

The five patterns are

- the identifier and its name,
- the past event,
- the future event,
- the template, and
- the self-related record.

To make them tangible, we'll use them to design a sample application. Starting with a statement of system scope, we shall apply each, in turn, to derive a finished conceptual database design.

Our application is an imaginary maintenance equipment spare parts inventory system. The first order of business is to come up with an unpronounceable acronym, and our system is no exception. We'll call it MESPIS, and its threefold scope is to

- keep track of the on-hand stock quantity of each item in our firm's spare parts warehouse;
- produce a report when it is time to re-order a spare part, based on its stock quantity being too low; and
- identify, for each part, those pieces of equipment in which it is used.

Our first pattern is the identifier and its name. Every entity (everything about which we'll store business data) carries both identification fields and data fields. ID fields identify, data fields describe. ID fields tell our client's staff as well as our programs which specific occurrence of the entity is at hand. Data fields describe attributes or characteristics of the entity.

As we saw in "Objects and Events," (see part three of the series, Sept. 15, p. 152) entities come in two groups, volatile events and involatile objects. The pattern simply tells us that objects should always carry at least one field of each type: an entity identifier (an ID-number or keyfield) and a name (or description). Without these two fields, we wouldn't have a viable entity at all.

Since our system deals with objects (spare parts), we begin by drawing a box labeled "part" and know that the record must have an identification number of some sort and a name.

Of course, since the scope statement spoke of stock quantity, we include this field as well.

Now and again we'll run across a client who affirms that a proposed involatile entity needs no name or description. We include one nonetheless (productive skepticism). If we neglect to do so, we'd probably come back in six months and find them using address or location or some other fields to store the information. To keep the other field usable, we would then retrofit the record and add description after all. So, we might as well do it now and get it over with.

Next time, we'll continue designing the MESPIS database by examining past and future events. ◉

Frank Sweet is corporate manager of data administration for the Charter Co., Jacksonville, Fla.

We continue our exploration of database design in parts 10 and 11 of this 14-part series, Process-Driven Data Design.

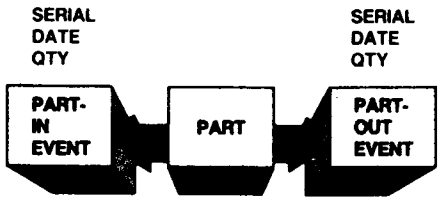
PAST AND FUTURE EVENTS

by Frank Sweet

To refresh your memory, our sample application is an imaginary Maintenance Equipment Spare Parts Inventory System. MESPIS's first scope goal is to "Keep track of the on-hand stock quantity of each item in our firm's spare parts warehouse." So far, our database design contains only one record.

Applications that record the past need past-event records. In the context of meeting our first goal, what is the single most vital data field (as opposed to ID field) in the part-item record? On-hand-stock-qty, of course. The field is actually named in the scope statement. With it, we can begin to address the other goals. Without it, we haven't a prayer.

The past-event pattern tells us that when a field is so important that the whole application hinges on it, an update audit trail isn't a luxury; it's a bare necessity. We must create a volatile entity (an event) to record every occasion when the on-hand-stock-qty was updated.



Some users might deny that audit-trail records are needed. The decisive criterion is simple: is the accuracy of the field in question part of the goal? If it is, we must include the event record. Failure to do so could result in the following six-months-later scenario: the field is vital, one or two cases are suspected to be wrong, the application is challenged, and we are asked to demonstrate how the questionable values were arrived at.

The point is not that past-event records make our demonstration easier. They avoid the challenge altogether. Consider your own experience. How often do you phone your bank because of a questionable account balance? How often would you call if its monthly statement showed only your current balance and did not list every check and deposit?

What fields should the past-event records carry? Data fields and ID fields, naturally. Data fields include quantity, date, and description. Quantity (in or out, as the case may be) is essential since the records' purpose is to justify the balance carried in the part-item record. Date is also needed for the same reason.

Description is not crucial since we're now dealing with events, not objects. It's not forbidden, of course, just not mandatory. The user might want to write somewhere: "This is when Harry replaced the steam trap on unit 12 because he backed the truck into it." If so, include a description. (Alternatively, since the above sentence tells us the individual withdrawing the part, the cost center to be charged, and the reason for the withdrawal, each of these elements could be codified and made separate fields.)

An identifying serial number of some sort is also needed. Without such a number, there is no way to tell a program (or a person) which specific event we are referring to in any particular case. If some law of nature decreed that there be a maximum of one "in" and one "out" for a given part on any given date, then the date itself would seem capable of doubling as the record's identifier. There are two reasons why this would be unwise. First, it's unlikely that such a law exists. Second, mixing meaningful data into identifiers leads to unresolvable confusion if those data must themselves be modified (Harry didn't take the part on Thursday after all; it was Friday). Ideally, record keyfields should be unique, unambiguous, unchanging, and dataless.

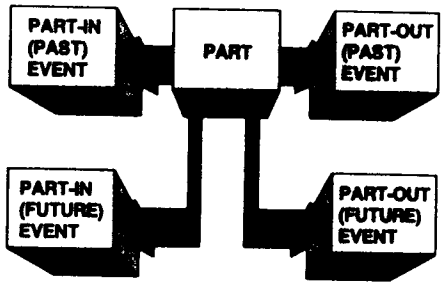
A final thought on our past events: two different boxes are shown—part-in and part-out. They might be merged into just one type of record in physical database design. The decision will pivot on the similarity of their fields and their relationships to other records. For now, it's best to keep them separate. After all, part-comes-in and part-goes-out are fundamentally different happenings in the real world.

PART 11: FUTURE EVENTS

Now, in part 11, let's investigate a close relative of the past event, the future event. Systems that are meant to manipulate the future (to make something happen) need future-event records. Unfortunately, dp historically grew out of accounting, so most older systems simply record the past. So far, MESPIS looks to the past, and our design reflects this.

But MESPIS has three goals. Is one of them intended to make something happen? Consider the second goal: "Produce a report when it is time to reorder a spare part, based on its stock quantity being too low." Evidently, the report is meant to get the part resupplied; its ultimate goal is to avoid running out. This is important because it hints that our system looks to the future as well as the past. Hence, we turn our attention to the third database design pattern, the future event. It tells us that, to manipulate the future, we need future-event entities.

Adding these to our design results in the following diagram. The picture now shows one involatile object (part) and four volatile events (in and out, past and future).



The fields in the two new records will be the same as those in the past events: date, serial number, quantity, and (optionally) description, since the same rationale

Past-event records don't make our demonstration easier; they avoid the challenge altogether.

applies. But why do you look troubled? Ah, you want to know how on earth we came up with two more boxes despite having no idea what we're going to do with them. Well, I'm reluctant to admit it, but I haven't the foggiest notion either. I threw them in because that's what the future-event pattern told us to do and, as I mentioned, it is an old and trusted friend. Now that we have them, what say we press on and figure out their use?

Consider the resupply report. "Based on its stock quantity being too low," the goal said. Evidently, we need a field, qty-that's-too-low, in part-item. The daily resupply report will list all items whose on-hand-stock-qty is less than its qty-that's-too-low.

But a moment's thought reveals that once listed, an item will continue to be listed every day until resupply arrives and on-hand-stock-qty gets back up to where it should be. Surely, that can't be what was wanted. After it's printed, the report will undoubtedly go to someone in purchasing who buys parts to resupply the stockroom. Assume she immediately orders everything on the first day's report. How will she feel about being harassed every day about those same items until they arrive? When that happens, we know it will be futile to plead that "We met the goals," for the time-hallowed reply is, "What we really wanted was . . . "—and that path leads to madness.

Should we then suppress an item's appearance on the resupply report if it is already on order? Well, not exactly. If on-hand-stock-qty is 13, and qty-that's-too-low is 350, and she ordered 12, she deserves harassing. She should have ordered at least 337. The solution is to report only those items where on-hand-stock-qty plus the sum of all the already-ordered-qtys is less than qty-that's-too-low.

This leads us to seek a place to put the quantity and expected arrival date of every outstanding purchase order. In other words, we need a record for each expected part-in event. The future-event pattern happens to have provided just that.

As a second example, consider our supply of framis bearings. This vital component's on-hand-stock-qty is 20. Its qty-that's-too-low is 15. And its sum of already-ordered-qty is 10. Since 20 plus 10 is not less than 15, we don't want the thing to show up on report. If it does appear we'll soon have more framis bearings (whatever they are) than we need.

But investigation reveals that Maintenance plans to tear down the paraxylene dehydration tower's regeneration loop and replace 50 framis bearings as routine preventive maintenance. Consequently, they'll need 50 framis bearings next month, and if we fail to include the item in the report, they might not be ordered in time.

CASE IS NOT AN EXCEPTION

Don't think this is an unusual case to be handled as an exception. Major preventive maintenance is costly in labor, materials, and (especially) downtime. If we fail to use this information in selecting items for resupply, our system will consistently fail to keep in stock the very parts that are most needed. Listen! Did you hear those voices? They sounded like, "We met the goals," followed by, "What we really wanted was . . . ."

Taking preventive maintenance into account, an item should go on report if on-hand-stock-qty plus already-ordered-qty minus planned-maint-qty is less than qty-that's-too-low. In other words, we need a record for expected part-out events. We happen to have one handy.

Call the two scenarios "resupply suppression based on ordered quantity" and "resupply triggered by planned use." There's something thought provoking about them. Notice three things: they represent significant requirements, they would not have been noticed, and intuitive solutions would have led to blind alleys. They represent significant requirements because failure to handle either would have made MESPIS unacceptable, generating emergency change requests within days after installation. They would not have been noticed because the craft of design centers around asking what output the user wants. Yet neither case affects output format or content in the slightest. Intuitive solutions would have led to cul-de-sacs because, in both cases, the solutions would have buckled under postinstallation pressures. Suppressing every item once it's been listed leads to insufficient reorder, and treating routine preventive maintenance as an exception ignores the heaviest spare parts use of all.

Yet the future-event pattern provided the solution to both, even before the problems became visible. ◉

Frank Sweet is corporate manager of data administration for the Charter Co., Jacksonville, Fla.

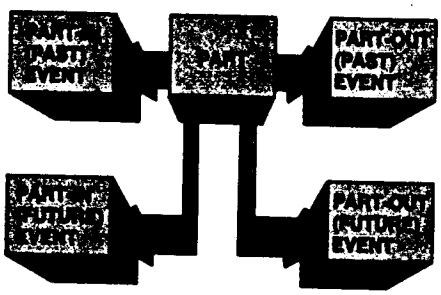
The final three installments of our 14-part series, Process-Driven Data Design.

THE TEMPLATE

by Frank Sweet

Future-event records tend to be quite volatile. Their short lives and high flowrate often call for a cookie-cutter entity to help stamp them out in assembly-line fashion.

We've been designing a maintenance equipment spare parts inventory system called MESPIS. Its goal is to report when we must reorder spare parts, based on their stock balance being too low. So far, our database looks like this:



Part, the hub of our design, contains ID-number, description, and on-hand-stock-qty for each spare part. The two past-event records document every occasion when the part's on-hand balance was updated. The two future events predict arrivals of ordered parts and planned usage. Now, we'll examine the template, or cookie-cutter, pattern.

Compute the volatility of our part-out future event. Say 10 maintenance work orders are done each month. They are planned 30 days in advance, and each requires about 100 different kinds of spare parts. Once the parts are actually consumed, their future-event records vanish, replaced by past events. Hence, some 1,000 new future part-out records are born every month, each with a life span of about one month. This 1,000-record population has a volatility of 100% per month. Work it out, and you'll see that if maintenance work orders were scheduled three months in advance, the volatility would be only 33% per month. In other words, future-event volatility is inversely proportional to the user's planning horizon; the less farseeing the forecast, the higher the volatility.

Contrast this with past events. They appear when parts are consumed and last for however long we need their audit trail. For example, keeping records in a system for six months results in a 17% monthly volatility. Past-event volatility is inversely proportional to the user's need for history; the more history needed, the lower the volatility.

Since hindsight is sharper than prophecy, any application's future is less certain than its past. Consequently, future-event records are, by far, the most volatile entities in a database. They are produced in a steady stream, live out their short lives, and vanish.

Precisely because they emerge in a steady stream, loading them with data can be tiresome. A model or template record helps. This pattern tells us that when we have a highly volatile entity, we should plan how we'll produce its fields. One way is to find a record wherein we can house standard default values for the fields.

Consider lead time. We saw last issue that our future part-in record carries a date telling when the event (part's arrival) is anticipated. Notice that this information has value even beyond the scope of our system. With it, for instance, we could produce a report comparing the expected arrival dates of those framis bearings we spoke of with their planned consumption dates. True, such an expediter's report is beyond the scope of our development contract, but it's nice to know we could respond quickly to such a request if called upon.

The problem is, where does the information come from? We could ask the user to enter it manually each time. This is more work for our unfortunate friend in purchasing. In addition to bombarding her with reorder warnings, we ask her to guess the date each part will arrive. How would she go about it? Framis bearings take four weeks, fernst gaskets take six, and light bulbs come in overnight. Knowing the nature of the part, she would add its typical lead time to today's date, giving a likely arrival date. Such lead time characterizes the part itself. It is a template datum because it helps compute a field (arrival date) in the volatile part-in future event. Hence, we should add estimated-lead time to the record layout of part-item. Similarly, we should inspect all fields in future-event records: how will each be produced? Would a template help?

Notice that we design the template to help the user, not replace him. The computed date is simply a first guess, offered as a suggestion. Final responsibility for accuracy remains with the user and we must enable him to manually overlay the computer's estimate with his own.

A productive-skepticism warning: users do not always see a template's usefulness as clearly as they do its threat. A template's goal is to handle the routine that makes up 80% of any activity, enabling users to override exceptions. But some confuse importance with ease of automation. With data such as lead time or price, users have been known to refuse template defaults and insist on having it done by hand: "Delivery-date is too important to be entrusted to the computer." Only time and

A productive-skepticism warning: users do not always see a template's usefulness as clearly as they do its threat.

familiarity can alleviate fear. In this situation, our wisest course is to include templates in the design but temporarily leave them out of the processing. Thus, a typical sequence of change requests reads:

January (system newly installed): "Delivery dates must be entered by hand. They are too important to be left to the computer."

April: "We need a list of standard lead times we can refer to (while manually computing and entering all those delivery dates)."

July: "The resupply report should automatically compute each order's estimated delivery date. But this must not go directly into the database. Instead, we shall transcribe the dates from report to database, correcting each as needed."

October: "Computed delivery dates should go automatically into the database. A separate Executive Review Report must be provided, however. It should list each day's computed delivery dates so we can carefully review them and correct those in error."

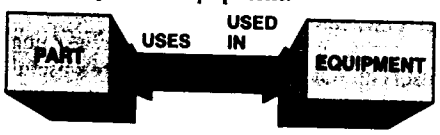
January: "Executive Review Report? Never heard of it. Oh yes, now I remember—that's the one Joe binds and files. Nobody knows what it's for."

THE SELF-RELATED RECORD

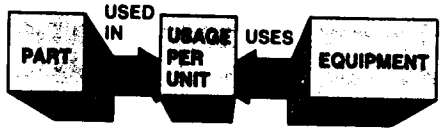
In previous issues we addressed two of the three goals in the scope statement for a maintenance equipment spare parts inventory system. We've determined that the hub of our database is a part-item record with one occurrence for each different kind of part we stock. The record holds the part's ID number, description, on-hand quantity, reorder point, and delivery lead time.

Now, in part 13, we examine the third goal, that is, to "identify, for each part, those pieces of equipment in which it is used." The statement also implies the converse: to identify, for each piece of equipment, the spare parts it uses.

One approach is to add an equipment box, that is, a file containing a record for each piece of equipment.



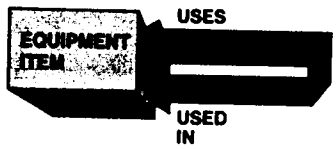
The arrow is two-headed because, while any part (e.g., lubricant) could be used in many different kinds of equipment, a given piece of equipment could require many different types of parts. Recall that a two-headed arrow warns of a missing intersection entity. Replacing it with the intersection, we have:



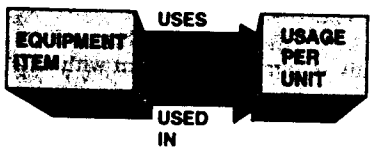
This approach is satisfactory and might do the job. Its main flaw lies in the need to discriminate between a spare part and a piece of equipment. Terminology, it turns out, often depends on context, and something that's called a spare part one moment might be termed a piece of equipment the next.

Looking at an automobile, we might consider the entire engine assembly a spare part. But if we consider the engine as equipment, its ignition cabling group (i.e., coil, distributor, wires) might be termed a spare. And, while a faulty ignition system (equipment) is being repaired, a single spark plug cable (spare part) could be replaced.

If such context-dependent terminology were the case in our application, then treating part-item and equipment as two different entities would be a mistake. A less redundant solution is to have just one entity and call it equipment/item as a compromise.



The situation regarding this two-headed arrow is precisely the same as in the prior one. It means we need an intersection entity. What makes it confusing is that the same record lies at both ends of the arrow. But, though the relationships are harder to visualize, the same rules apply. The resulting pattern is so widely used that it has a name: bill of material.



NUMBERS, KEYFIELDS, ETC.

It is dangerously easy to think a conceptual database is designed when, in fact, major issues are still unresolved. The topics keyfields, numbers, and real things comprise a form of checklist we find useful in deciding whether we are really finished. They are not new. We have mentioned all three before and in part 14 we review our prior discussions.

Keyfields—unique, unambiguous, unchanging, and dataless. Every entity should have a keyfield—an identifier that will tell a person or program which occurrence is at hand. Keyfields should be:

- Unique. Each real-world object or event should be represented by only one occurrence of its entity. Don't have two item records for the same part.
- Unambiguous. Each occurrence of an entity should model only one real-world object or event. Don't mix light bulbs and gaskets in a single time record.
- Unchanging. Once they are assigned, an entity occurrence's keyfields should remain unchanged.
- Dataless. Keyfields identify. Data fields describe entity attributes. Don't mix the two functions.

Numbers—population and volatility. An easily avoided error in database design is to neglect numerical analysis. To an intern, it may seem that the experienced surgeon takes risks. To an apprentice, the seasoned engineer may appear to guess at pressure vessel stress. Similarly, to novice database designers, veterans can seem to shortcut numerical analysis of the data. In all three cases, appearances are deceptive.

A database designer working on his or her tenth materials management system might give the illusion of being unaware that the bill of material is a complete template for supply requisitions, or that maintenance work-order volatility is between 50% and 100%. But, like the swan's effortless glide, it is an illusion that conceals frantic paddling beneath the surface. Before signing off a conceptual design, we must know every entity's population and volatility. Until we do, our design is unfinished.

Real things—objects and events. Application systems analysis usually begins by studying the existing system, automated or not. This is the easiest way to find out what it's all about. But our database design would be less than professional if it simply modeled the existing system. Our goal, after all, is to model underlying physical reality.

Every box in our design should represent an identifiable entity in the real world. Nonvolatile boxes simulate objects (or intersection data about pairs of objects). Volatile boxes model events or happenings that actually take place. No box should simply model a record in another data processing system, automated or manual.

Concluding a conceptual database design, we ask ourselves three questions:

Does every box have a unique, unambiguous, unchanging dataless keyfield?

Can we produce reasonable population and volatility estimates for every box?

Does every box represent either a real-world object or a real-world event?

Yes? Then we're done. ◉

Should vendors buy their software-rich oems?

THE DEC STATUS QUO

All of these moves by its competitors to beef up their oem programs may not be a threat to the number one minicomputer supplier, Digital Equipment Corp., a company official says. In fact, Brian Cranston, DEC's distributor program manager, claims it is reassuring to hear about all the new marketing support programs other vendors are offering, for DEC has been making similar offers to select oems since it pioneered the authorized distributor concept five years ago. To stay above the competition, DEC has just introduced a new program, in cooperation with a Madison Avenue advertising agency, to provide oems with assistance in devising advertising and sales promotion plans.

"This is something our competitors are going to have a hard time copying," says Cranston. "It's very easy to copy a discount schedule and even go one better—you just change the number from 30% to 35%. It's a lot harder to address the kinds of needs that our promotional planning program will address."

Harry Beisswenger, president of Compute-R-Systems, a Plymouth Meeting, Pa., software house specializing in legal packages, finds advantages to participating in the DEC distributor program, in part because it is mature. An oem/vendor relationship, he says, "is like a marriage; it takes a while to work out. It takes time for manufacturers to learn to deal with oems, and DEC has a lot of experience there."

Peter Lowber of the Yankee Group believes that DEC has a good track record in providing support to its authorized distributors, but adds that "they are expensive."

Cranston believes DEC's discounts are "pretty competitive." The company now offers anywhere from 15% to 37% off on the PDP-11 (with an average discount of about 30%), and up to 26% off on the VAX (averaging about 19%).

"Come in. Have a snack. Have a drink. Talk to some people. Leave."

CARTOON BY SIDNEY HARRIS

Says Lowber: "I think one problem that DEC has had is that it has pricing strategies that conflict with its own sales force and its oems. Some of DEC's oems can't sell the VAX because the customer base is going to buy it from the direct sales force. Then what happens is that the direct sales force might discover that its customer needs application XYZ that has been developed by oem ABC. So it'll go to the oem and offer it the software sale. The oem is not going to decline a sale, even though it's not selling the VAX, so it ends up being more of a software supplier."

Vendor interest in converting their oems into software houses, rather than full system packagers, is moving into a second phase that alarms some industry officials while it is welcomed by others. Hewlett-Packard, Lowber points out, recently bought its second largest oem. "It was probably a good decision in this case," he says, "because what HP got out of it was a bunch of applications software experts that understood not only how to develop software but also how to train a sales force in supporting that software. So the company didn't just bring in a bunch of vertical applications software but also the ability to train a whole field staff to offer specialized support."

Will other oems be acquired? "It's an interesting proposition," says Horne at Prime, "but I think it's a double-edged sword. Most of the software firms that have been acquired lately have gone at an extremely high price-to-earnings ratio. The only time it may be worthwhile is in the startup phase, but then you have a real problem of how to keep the principal engineers interested in the product once they are owned by a Fortune 500 company."

Many oems are equally skeptical. "I could see it happening," says Ken Tratar at Systems Management, "and it would foul everything up. They're buying oems to buy their vertical marketplace . . . but they don't know the business. They look at an oem with, say, a particular piece of hospital software, and they say, 'Boy, we'll just buy that oem and make the product available to all of our people across the country to sell.' And that's a fatal flaw, because the average computer salesman doesn't know anything about hospitals. On the other hand, the oem people probably came from a hospital environment or worked very closely with a hospital to design their package and have the product knowledge to sell on a limited basis." ✱

Carol Fletcher is a Chicago-based free-lance writer with a special interest in technology and business topics. She formerly was an editor and reporter for several computer-related magazines.