IMAGE/SQL Database Foundations

F. Alfredo Rego

Adager Corporation
Sun Valley, Idaho 83353-3000 • *USA*

www.adager.com

Business-critical online transaction processing

IMAGE/SQL databases (which are an exclusive competitive advantage of Hewlett-Packard's HP e3000 Business Servers) are built for ***online transaction processing*** applications (OLTP). For convenience, we say "IMAGE" instead of "IMAGE/SQL."

These specialized high-performance databases are the life-blood of business-critical processes that must be "up" all the time, because every second of "down" time is very expensive.

Hewlett-Packard wrote in an advertisement that appeared in *Computerworld* and *Information Week:*

> *If your company's survival depends on system availability, the HP e3000 is the one to rely on. The HP e3000 delivers* 99.9% *uptime. The Datapro User Survey of midrange systems ranked the HP e3000 #* 1 *in reliability.*

Award-winning IMAGE gets the job done well, reliably, and within reasonable economic constraints. This is a rarity in this era of hype and it is something worthy of celebration.

We are very pleased to work with IMAGE, but we are never satisfied. Since it is human nature to try to squeeze even more from the good things that we already have, we would like to study how we can get as much as possible out of our IMAGE databases, now and in the future.

Where do we begin?

A database (even a very sophisticated database) is just a *crude* model of the reality of an organization. We can't store reality in a database, just as we can't keep an actual orchestra in a CD.

At best, we can hope to maintain a half-decent description or representation which, through the magic of electronics, will play back a reasonably useful likeness.

The representation, due to limitations of technology and economics, will consist of a group of values for a relatively small collection of "features" (or "peculiarities" or "attributes" or "characteristics") that we want "to keep track of" because we consider them important for the functioning of our specific enterprise. For example, consider a personnel database and the typical attributes it maintains for employees: name, data of birth, salary, and so on.

Whether the attributes that we sustain in our databases *are* important or not is a crucial matter, because there is no sense in carrying around an overwhelming load of useless material.

Learning to say "no" (emphatically but respectfully) may be our most valuable design skill, after all.

A differential approach to database design

The database design (and implementation) process is a jungle that looks impenetrable. A good first step is to sort things out into a few significant categories that are relatively easy for us to determine.

An interesting approach came to me as I was discussing differential calculus with my kids. We talked about *change* (and about various *rates* of change as well as about *rates of rates* of change, and so on). Everything changes and items that appear to stay the same do so only because they change very slowly according to our perception.

With this idea in mind, we took a piece of paper and drew a *change line* that had "things that change a lot" at one end and "things that tend to stay the same" at the other end. We then had a lot of fun filling in the range in between.

We began with our human cycles of hunger, thirst, and sleepiness throughout a 24-hour period. We examined everybody's growth chart (as recorded on one of our kitchen's posts). We looked at family pictures, dating from the times of our grandparents to the most recent prints, including wedding pictures and baby pictures. We looked at photos of our house as it went through several remodeling efforts.

We looked at World history. We reviewed geology (as homework for a trip to the Galápagos islands). We went over astronomy and the amazing accomplishments of NASA (after having watched my TV conversation with Jim Lovell). We discussed cosmology and theology.

Back to Earth and to the here and now, it turns out that this differential approach with respect to time can be very useful for designing databases.

Segregate things according to their rates of change

We use *picoseconds* (trillionths of a second) to measure events which we think are super-fast. We use *eons* (billions of years) to measure events which we think are super-slow.

Somewhere in the middle of this wide spectrum we find the phenomena which occupy most of our attention. These are the events that are the most useful and interesting from a human perspective.

Most IMAGE databases keep track of resources (and their interrelationships) whose event-speed ranges from a "fast" which we can measure in seconds to a "slow" which we can measure in years. *Some* IMAGE databases, of course, also use real (floating-point) numbers in IEEE format to keep track of information that requires more precision than standard commercial transactions. The choice is yours and IMAGE is happy to provide the necessary database structures.

Big pie in the sky but tiny pie on the table

IMAGE gives you many options, but you must choose between *being complex* and *being complicated*.

Lousy designers take approaches that are convoluted, byzantine, perplexing, labyrinthine, fancy, sophisticated, involved, ornate, embellished, adorned, inconvenient, disadvantageous, ornery, oppressive, tyrannical, onerous, and taxing, even when dealing with simple situations. They offer a big pie in the sky but deliver a tiny pie on the table. Their motto is: *If you can make it complicated, why make it simple?*

Good designers can abstract—from a thicket of technical details and marketing extravaganzas—a couple of *essential* elements.

Resources and their relationships

There are two kinds of *fundamental types* or *categories* that are sufficient to create exceptional and powerful database models, even for apparently complex circumstances.

The two *essential* categories for designing databases are:

- ***Resources*** (*things, individuals, products, possessions, articles,* and so on). A resource has an independent, separate, or self-contained existence. We shall use ***entity*** as a synonym for *resource* in cases for which the term "resource" does not feel right (when we refer to people, for instance).
- ***Interrelationships involving resources*** (*interactions, collaborations, involvements, connections, cross references, entanglements, embroilments,* and so on). For convenience, we shall use ***relationships*** as a synonym for *interrelationships*,

even though there are subtle technical differences between the terms. ***It is important to note that relationships, in their own right, have important properties that we must represent in a well-designed database.***

This disarmingly simple—yet rigorous—approach to "dividing and conquering" gives us an edge over the complexity-oriented competition.

Kinds of resource and kinds of relationships

The ***kinds of resources*** in an enterprise and the ***kinds of relationships*** among such resources tend to remain reasonably stable for relatively long periods of time. For example, a company will always have *people* who work for it and *products* that it makes.

Specific resources and their specific relationships

Specific resources and ***their specific relationships*** tend to change more. For instance, regarding people, we may *hire* new employees, we may *fire* some, and we may *lose* some to the competition. Regarding products, we may *discontinue* old ones and we may *create* new ones.

The fact that we add (or delete) a *specific* resource or a specific relationship does not mean that we have to change the *kinds* of resources or relationships that are the vital parts of our organization.

In IMAGE, we use ***data entries*** to represent specific resources or specific relationships.

Datasets

In IMAGE, we use ***datasets*** to represent stable kinds of resources and their stable kinds of relationships. A ***dataset*** (master or detail) is a homogeneous collection of data entries. For instance, the dataset of all employees, or the dataset of all airplanes, or the dataset of all classrooms.

A dataset represents a *category* (a group whose members have a collection of attributes in common). Each individual member has a ***key***—a unique identifier that distinguishes a given object from all other objects within a system. The specific values of the ***non-key attributes*** *may be* different for different members of a dataset. The specific values of the ***key attributes*** *must be* different for different members of a dataset.

The only difference between *master* datasets and *detail* datasets is their performance-oriented method of access. Both masters and details provide the ability to access their data entries by means of *serial*—also called *sequential*—scans and by

means of *directed* access to any data entry via its absolute address (entry number). In addition:

- Master datasets are biased for *hashing* and *B-Tree indexed* access.
- Detail datasets are biased for *chained (linked)* access along pre-defined (but optional) performance-enhancing paths.

Datasets for entities

There are ***specific entities*** (for instance, the employee whose name is Janice López, or the company called Control Engineering, or the department called Sales). Each specific entity has its own ***specific entity attributes*** (such as Janice's employee number, or her date of birth). We want to keep track of these entity attributes in our database and we want to be able to access, as quickly as possible, a given entity and its attributes. IMAGE master datasets are optimized for this kind of access by means of hashing and B-Tree indexing.

Datasets for relationships

There are also specific relationships (for example: the assignment relationship between the entity called Janice López—an employee—and the entity called Sales—a department within the company). Each specific relationship has its own ***specific relationship attributes*** (such as the starting date of Janice's assignment to the Sales department, which may be different from the starting date of Janice's employment at Control Engineering). We want to keep track of these relationship attributes in our database and we want to be able to access, as quickly as possible, all of the relationships—and their respective attributes—for a given entity. IMAGE detail datasets are optimized for this kind of access by means of chaining.

The generality of relationships

Our goal is to add generality and to reduce complexity. Consequently, we do not want to restrict relationships unnecessarily. In fact, most relationships should allow a given entity to be related to zero, one, or more entities—of the same kind or of different kinds.

Under special circumstances, we may want to apply restrictions—but restrictions should *not* be the default. Let's take *marriage* as an example. Under strict Catholic rules, a priest or a nun should be married to *zero* people. Under "standard" law, a person can be married to *one* other person—of the opposite sex in most jurisdictions, but not necessarily so in others. Under polygamist rules, a man could be married to *several* women.

The *number* of entities involved in a relationship is just one dimension. The *kind* of entities involved in a relationship is another dimension. The involved entities may be members of different classes—for instance, an employee may be related to a department through an *assignment* relationship. Or the entities involved in a relationship may be members of the same class—for example, an employee may be related to another employee through a *management* relationship.

Regardless of your moral—or legal—restrictions regarding marriage or any other relationship, your DBMS should *not* force its restrictions on you.

Databases

An IMAGE ***database*** is a homogeneous collection of datasets and their supporting data structures.

Attributes

Resources and their relationships have *characteristics* or *attributes* (such as *name, date, salary,* etc.) that define specific dimensions of "color" or "character" to distinguish their current *state* or *status* from the state or status of other resources or relationships.

An attribute's *value* quantifies a specific *dimension* for a specific entity or for a specific relationship. Each attribute for a given entity or for a given relationship can have only one specific value whose choice is limited by the attribute's *domain*.

For instance, you may have an attribute called "day of the week" whose domain is "Monday, Tuesday, ..., Sunday."

In fact, the term "data validation" implies a verification of the specific values of attributes, to make sure that they fall within the attribute's domain (for instance, we should not have a database that has a value of "722" for the "day of the week" attribute—neither should we have a value of "1756" for the "year of birth" attribute for a current employee).

Keys

One of these attributes (or a combination of a few attributes that represent essential properties) is a ***key*** that unequivocally identifies a ***given resource*** or a ***given relationship.*** An example of a key for people with a given nationality is *passport number.*

The key is composed of one or more attributes that form a unique *handle*. A key denotes an object or a relationship among objects, while non-key attributes denote the *properties* of an object or a relationship.

A key ***identifies*** a given entity or a given relationship, setting it off from other entities or relationships of the same kind.

As an interesting example of entities with *several possible keys,* consider the periodic table of the elements. Each of these attributes is a valid key in its own right:

- The element's *symbol* (for instance, "C").
- The element's *name* (for instance, "Carbon").
- The element's *atomic number* (for instance, "6").

You could choose *one* of these candidate keys as ***the*** key, even though the table includes all of these keys for convenience's sake under a wide range of situations.

Functional dependencies

The ***functional dependencies*** among well-chosen keys and attributes will tend to show remarkable stability. For instance, the functional dependency between a personal identification number and a given person will probably hold for life.

Individual attributes of a specific resource or a specific relationship tend to change more than the ***types*** of resources and relationships. For instance, Jane Doe's *position* within the company may change, or her *home address* may change, or her *salary* may change, or her *family name* may change. But, in spite of these relatively minor changes, Jane Doe remains as a valuable member of the "people" type of entity for the company. This is an integral part of our differential approach to database design.

Intelligent attributes

We may certainly think of attributes as just pieces of data, but I prefer to think of attributes as a combination of *data* and *the intelligence required to process the data.* As an instance, let's study our treatment, in Adager, of the name attribute as we apply it to the names of individuals in our customer database. We have implemented this technology easily, within the rules of IMAGE. We keep the information for names thus:

```
<LastName><Separator&Title><FirstName>
```

LastName may have only one family name ("Smith," as in the U.S. tradition)—or many family names ("Montes Fernández de García Salas," as in the Guatemalan tradition).

Separator&Title is one of several codes (for instance, a colon ":" for "Mr." or a semicolon ";" for "Mrs."). *Separator&Title* serves a triple function:

- Separator between LastName and FirstName.
- Clue to the person's "title" or "greeting qualifier."
- Data compressor, because there is no need for an extra blank to separate last names from first names.

My name attribute, for instance, is encoded as "Rego:Francisco Alfredo" and decoded as "Mr. Francisco Alfredo Rego" (or decoded as "Mr. Rego" for a greeting). In addition to the information, we have encoded the intelligence required to process this information.

Fields

A ***field*** is the smallest meaningful component of information in an IMAGE database.

A data entry is composed of one or more fields which store the specific characteristics of an entry's keys and attributes. For instance, we may define a data entry for an *employee* with these fields: *name, birth date, salary.* We may also define a data entry for an *airplane* with these fields: *name, number of engines, price.*

Data items

For convenience, IMAGE defines fields by means of global entities called ***data items.***

We define a data item only *once*, and we then use a given data item to define fields in as many different datasets as we desire.

A field is a data item which is referenced in a dataset's data entry. The data item *name*, for instance, is referenced as an *employee name* (such as *Germaine Soffey*) and as an *airplane name* (such as *Boeing 747*).

IMAGE and QUERY standardized a useful syntax in the early 1970s, with a period between the dataset name and the field's data item name. So, we say "the *employee.name* field" and "the *airplane.name* field."

Performance-oriented access strategies

In a high-performance online database system, we need to get information about given entities and their relationships while somebody *waits over the counter* or *waits over the telephone.*

Too much waiting and that somebody will prefer to go to our competition. This means that we must find the entry (or group of entries) in question—among billions of entries—as efficiently as possible.

The challenge is to get such entries with the minimum of hassle in the fastest possible way while utilizing the least amount of disc space and the tiniest amount of effort during the original ("data capture") creation of the entries.

Obviously, we have too many conflicting requirements and, as a consequence, there is no ideal access method that will be perfect under all circumstances. We must be willing to pay some price and we must reach some kind of compromise.

IMAGE provides several kinds of access methods which we can use according to our needs under various conditions.

We should design (and periodically tune up) our databases to provide the fastest possible response time for the most important transactions and queries. We want to minimize the effort required to answer the most frequently asked questions.

This issue ***(answering the most frequently asked questions)*** is what must guide our design and maintenance choices regarding database access methods.

Serial or sequential

From the *current* entry, get the next (or the previous) available entry (regardless of the value of its search field). IMAGE does not care *how* you got to the current entry and you must be aware of your whereabouts to avoid trouble.

This method works fine for small datasets (or for month-end batch processing, which reviews every entry) but it may take "forever" and be unacceptable for online situations.

Directed

Get the entry (if it exists) at a specific data entry number or address. Caution:

- Master entries may change their location due to secondary migrations or repacking.
- Detail entries may change their location do to repacking.

Calculated or hashed

Get the master entry (if it exists) whose search field contains a given value, regardless of its location.

Hashing is a mechanism that converts a *value* to a *number* (within a well-defined number range). Master datasets use hashing to locate, very quickly, *one* entry of interest among millions of entries. (There are two kinds of hashing in IMAGE, depending on the data type of the search field.)

We provide the value of the search field and IMAGE calculates the appropriate primary address within the master dataset where this entry should reside.

It is possible that several master entries (with different search field values) may result in the same calculated primary address. IMAGE keeps all such entries linked together in a synonym chain, for performance reasons.

Chained

From the *current* entry, get the next (or the previous) entry with a congruent search-field value.

Chaining is a mechanism that links (logical) neighbors even when they may be (physically) millions of entries away from each other.

IMAGE does not care *how* you got to the current entry and you must be aware of your whereabouts to avoid trouble. Normally, you call *dbfind* to specify a detail's path (and a specific search-field value within that path) *before* you call *dbget* to get the next (or the previous) entry with a congruent search-field value. If you forget to call *dbfind* before calling *dbget*, IMAGE will use the primary path as the default current path and the current entry's search-field value as the default congruency criterion. If these defaults are not what you were expecting, you will get an incongruous surprise.

"Congruent search-field value" means different things for masters and for details:

- In *master* datasets, all data entries whose ***search field values hash to the same primary address*** are linked together by means of *synonym chains.*
- In *detail datasets with paths,* all data entries whose ***search fields have the same value*** are linked together by means of *path chains.*

To implement high-performance chains, IMAGE uses specialized *list* data structures to maintain *link* pointers (forward and backward) as well as *head* and *foot* pointers (to be able to locate the beginning and the end of the chain).

These data structures allow IMAGE to navigate through masses of data and to access, in a hurry, those entries which are congruent to each other by means of their search field values. For instance, all 25 checks written by customer number 7702 out of a possible 86,042,600 checks can be found in a fraction of a second.

The paths through a detail can be *sorted*. Regardless of the time of the month when a given customer's checks were cashed, we can have a neat chain for each customer which has logically linked together, by check number, all of the customer's checks.

Due to entry recycling (IMAGE reuses the locations of old entries that have been deleted), check number 205 may be in entry number 2 and check number 10 may be in entry number 1045 (i.e., check number 205 is physically located *before* check number 10) but, when we retrieve the checks via a sorted path, we will get check number 10 before check number 205.

We pay two kinds of prices for these conveniences. There is *extra work* when we add the entry to the dataset (chain point-

ers must be updated) and there is *extra space* in the entry to store such pointers.

Paths as optional performance boosters

For performance reasons, you may use paths to hard-wire some *obvious relationships* as "hot" in the database's structure. But you do not want to be stuck for life, since some hot relationships may cool off and some sleepers may wake up unexpectedly.

Fortunately, IMAGE paths are nothing more than performance-boosting options for *retrieval time.* You may or may not choose to include paths in your design, at your discretion. You can always add or delete paths at any time (using Adager, for instance) during the database's life, even after having added millions of entries. You can also sort (and unsort) paths at any time.

Paths have nothing to do with database structures and they *are a performance option and not a requirement* (i.e., you can design an IMAGE database that consists entirely of stand-alone datasets).

Pointers: essential & non-essential

IMAGE's strategy—in terms of optimizing access—includes the use of pointers. But there are various types of pointers and, before expressing unsubstantial opinions regarding IMAGE, one must understand the specific kind of pointers that IMAGE uses.

Standard hierarchical and network database management systems use *essential pointers.* Essential pointers convey information about your information. If you delete the pointers, you lose information. You need these pointers, desperately, when you are dealing with a hierarchical DBMS or with a network DBMS.

IMAGE, on the other hand, is neither a hierarchical nor a network DBMS. IMAGE uses *non-essential pointers* which do *not* convey information about your information. IMAGE's pointers are intended for indexing, to improve the performance of accessing given data entries within a database. Non-essential pointers are redundant and you can always eliminate them—at any time—without losing any information (the only loss you suffer is a loss in performance). By the same token, you can always add non-essential pointers. In fact, Adager was created in 1978 *because of* this fundamental idea.

If you decide to exchange your perception of your *entities* (modeled via *master data entries,* for instance) and your *relationships* (modeled via *detail data entries,* for example), you can

convert IMAGE master datasets to detail datasets and vice versa. Whether or not you will ever choose to take advantage of the tremendous flexibility that IMAGE offers you, it is nice to know that the flexibility is there.

B-Tree indexing for masters

IMAGE provides B-Tree indices for master datasets so that you may quickly access groups of master data entries whose ***search field values are within a given range*** (even when they are physically scattered all over the dataset).

Third-Party Indexing (TPI)

In addition to sorted access (to both master and detail entries) according to the values of *any* fields, TPI offers other advanced features (such as keyword retrieval).

Bushwhacking or highway driving?

IMAGE allows us the freedom to go explorer-like, making our way through thick woods by cutting away bushes and branches with sequential and direct access methods.

IMAGE also allows us the convenience of traveling through "pre-established hubs" by means of techniques such as ***hashing***, ***paths***, ***B-Tree indexing***, and ***TPI***.

We do not *have to* access anything in a predetermined way, but it is nice to know that we may choose to do so, if we know that a given "well-trodden route" will get us more quickly to our desired destination.

Why wade through swamps if we can use a bridge? Why swim across the Atlantic if we can take the Concorde?

Critical Item Update (CIU)

DBUTIL allows you to configure ***critical-item update*** (CIU) to allow (or to forbid) IMAGE's native *dbupdate* intrinsic to update *detail sort fields* and *detail search fields*.

This is a wonderful capability in terms of performance, because we can do (with a single call to *dbupdate*, which will modify only the affected chains) what would otherwise require a call to *dbdelete* (which would unlink all chains for this data entry, whether affected or not) followed by a call to *dbput* (which would relink all chains for this data entry).

We should be careful and precise regarding terminology. For instance, the loose usage of technical terms such as *search item* or *sort item* may confuse the issue of *items* as global database entities vs. *fields* as local dataset objects.

A given *global* data item can be *specifically* used as a *regular field* in dataset A, as a *search field* in dataset B, as a *sort field* in

dataset C, and as a *search field* for one path and as a *sort field* for another path in dataset D.

*Critical **Field** Update (CFU)* should have been the appropriate term, but it is too late now. Sigh!

Keys vs. search fields

A key is simply a ***field*** (or a ***group*** of fields) which uniquely identifies an entity or a relationship.

A key *does not have to be* an IMAGE search field (although a key *may be* an IMAGE search field). An IMAGE search field is defined only for performance's sake, to be able to take advantage of *hashing* and *B-Tree indexing* (for master datasets) and *chaining* (for detail datasets).

Generally, a resource's key is a *simple key* while a relationship's key is a *concatenated key,* made up by the keys of all the related entities.

A teaser

If the same entities engage in the same kinds of relationships but in different ways or under different circumstances, each relationship's key must include some additional attribute(s) as discriminants among the various relationships.

What on Earth, you say? Don't panic. With the help of a couple of famous movie stars who did some outrageous things a few decades ago, we may be able to see the light.

As an example of *entities* we may consider people (such as Elizabeth Taylor and Richard Burton) and as an example of *relationships* we may consider marriages.

Usually, most people think that there is *one* relationship between two entities, but the Burton-Taylor example shows that it is possible to have multiple relationships between two entities (I forgot how many times they were married to—and divorced from—each other, not to mention their own marriages to *other* people).

How can we model these various relationships between Liz and Richard (ignoring, for now, their own marriages to *other* people)? By assigning each marriage a unique *key* which is an unequivocal combination of some minimal set of *attributes* such as these:

- key for spouse # 1
- key for spouse # 2
- marriage date
- marriage place
- marriage authority (name of judge)
- divorce date
- divorce place

For most people, "key for spouse # 1" and "key for spouse # 2" should be sufficient (even for people with multiple marriages, because most people with several marriages marry different spouses). For Taylor and Burton, obviously, we need at least one more attribute to uniquely identify a given marriage, in addition to the keys for both spouses.

A combination of these three attributes may still not be sufficient, if they used the same place for more than one ceremony:

- key for spouse # 1
- key for spouse # 2
- marriage place

A combination of these three attributes may still not be sufficient, if they used the same judge for more than one ceremony:

- key for spouse # 1
- key for spouse # 2
- marriage authority (name of judge)

A combination of these three attributes *might* be sufficient, because it is very unlikely that these two stars managed to get a complete "marriage-divorce-marriage" cycle in one day:

- key for spouse # 1
- key for spouse # 2
- marriage date

Some modeling strategies are OK for reflecting *simple* relationships but have tremendous difficulties dealing with situations such as this that require the ability to model *zero, one,* or *many* relationships among *various* kinds of resources (or even within *the same* kind of entities, such as "people").

You should take full advantage of IMAGE's modeling power for these kinds of subtle challenges, which crop up in the most unexpected corner cases and boundary conditions in business-critical situations.

Transactions

Resources and their relationships don't just sit there. They interact with one another and with their environments by means of ***transactions*** which affect (and are affected by) such resources and their relationships.

Transactions define the ***allowed behaviors*** of data entries, which must be as efficient and as effective as possible:

- *Adding* new data entries (via IMAGE's native *dbput* intrinsic or via SQL's *insert* statement).

- *Finding* existing data entries (via IMAGE's native *dbfind* and/or *dbget* intrinsics or via SQL's *select* statement) so that we may relate them, report them, update them, or delete them.
- *Modifying* individual attributes of existing data entries (via IMAGE's native *dbupdate* intrinsic or via SQL's *update* statement).
- *Deleting* existing data entries (via IMAGE's native *dbdelete* intrinsic or via SQL's *delete* statement).

Locking

IMAGE database transactions may be launched by a single online user, or by a single batch process, or by many (possibly thousands) of concurrent online *and* batch processes.

To avoid chaos, each individual transaction needs to be undisturbed by other concurrent transactions. IMAGE offers several choices of granularity regarding *locking strategies* to make sure that we can achieve a fair compromise between high performance, throughput, exclusivity (for each transaction thread), and sharing (among various transaction threads).

IMAGE provides native *dblock* and *dbunlock* intrinsics, as well as other methods that allow us to control the flow of concurrent transaction threads, including back-out processing.

Worthwhile choices

A database can be as simple as the organized collection of an individual's e-mail messages or as complex as the organized collection of an airline's complete structure (airplanes, crews, airport gates, flight schedules, baggage routing, cargo, maintenance records, maintenance schedules, reservations, travel agents, frequent-flier programs, personnel, finance, training, investor relations, and so on).

It is possible to design a complex and cumbersome database for an individual's e-mail messages and it is possible to design an elegant database for an airline that is simple, economical, easy to use, and easy to maintain under heavy-duty use in harsh environments. It all depends on the designer's ability (or lack thereof) to abstract the essential qualities of the "reality" that the database is supposed to model.

There is a big difference between *knowing the syntax of some DDL* (data definition language) and *being able to design a good database.* There is a big difference between *knowing the syntax of some language* (such as English) and *being able to write a good poem.* There is a big difference between *knowing SQL (or the high-performance native IMAGE intrinsics)* and *being able to use an IMAGE database to our greatest advantage.*

We are confronted with virtually infinite choices regarding the number and variety of real-life "situations" that we *want* to model as well as the database "solutions" that we *can* create.

The challenge that every designer faces is to choose a small subset of ***worthwhile*** "situations" and an ***effective*** database model ("solution") that will do the trick as economically and as efficiently as possible, with the minimum of daily fiddling once the database is up and running under real-life non-academic conditions.

Complexity and normalization

Ideally, things should be simple. Unfortunately, things are complex. But we should avoid unnecessary complexity. This is the objective of ***normalization***. I have developed a wry working definition of normalization:

> *Keep together those things that belong together and separate those things that do not belong together.*

Deciding which things belong together is, obviously, a matter of taste. Nevertheless, an expensive taste may bankrupt us and matters of taste really *do matter.*

Normalization is the breakdown of seemingly complex operations into simpler processes. The challenge, at the beginning, is to place the appropriate elements (no more and no less) where they belong, at the appropriate level, at the appropriate place, at the appropriate time. Then, the challenge continues, since we must be able to reallocate resources quickly and effectively to balance the load, at any time, all the time. Normalizing is an ongoing, dynamic activity.

Normalization applies at every level in the global computer hierarchy, even though people generally associate normalization with data entries and with datasets (which fields should we include in this particular data entry and which fields should we exclude from it, placing them on another dataset?)

A normalized structure is open-ended. We can *add* more elements to any layer of abstraction without affecting existing systems. We can *delete* elements from any layer without affecting existing systems which do not access such elements.

Efficiency and normalization

Do we want to favor ***efficiency in terms of access*** or do we want to favor ***efficiency in terms of maintenance***?

In general, the higher the degree of normalization (i.e., the finer the splitting into chunks), the higher the costs of communication and coordination. Normalization is neither good nor

bad. It is simply a method which allows us the freedom to choose our favorite spot in a range which has highly unnormalized databases at one end and highly normalized databases at the other.

Usually, efficiency in terms of access implies ***redundancy***. But redundancy, in itself, is not bad. It is just more difficult to maintain a bunch of redundant things in perfect synchrony.

A super-normalized database contains a large number of small entries, with many instances of key fields distributed over many datasets. Even simple queries may require that we assemble the information from many sources, but we may have a better chance that each of these sources is correct. It is simpler to maintain a "specialist" source up to date than it is to maintain a complex source which tries to keep track of everything at the same time.

Normal Forms

The rules for the First Normal Form specify that data entries of the same type (i.e., belonging to the same dataset) must be uniquely keyed and must not have repeating groups.

The rules for the Second and Third Normal Forms specify that every field must be either part of the key or must provide a single-valued fact about *exactly* the whole key and *nothing else*. In addition, a relationship between data entries in different (master) datasets is always represented by a linking-detail data entry that contains, as search fields, the values of the keys of the involved masters.

We could get carried away and go to ridiculous extremes to normalize a database to death. We could conceivably slice the information about an employee into many entries, each containing—in addition to the key—a single attribute such as name, birth date, salary, and so on.

Common sense should prevent us from committing such atrocities and this is the motivation for the Fifth Normal Form *("there is nothing significant left to normalize").*

Components and compounds

There are two complementary ideas that we can use while structuring an organization's database model:

- *Disassociation*
- *Association*

Disassociation involves the conceptual separation of a compound into simple and meaningful *components*.

Association involves the orderly assembly of components into a meaningful *compound*.

Normalizing and relating

A database is based on two complementary ideas: *normalizing* and *relating*.

Normalizing is analogous to disassociation and involves the conceptual separation of information into simple and meaningful data items.

Relating is analogous to association and involves the orderly assembly of *data* into meaningful *information* which must be available as quickly—and as accurately—as possible, at any time.

A database management system relies on components and assemblies. A normalized database consists of fundamental, linearly independent, atomic components which, through relational operations, become *useful information.*

Hierarchies

A tree-like structure is, perhaps, the most common ordering method in the Universe, as corroborated by rivers, nerves, arteries, veins, organizational charts, pedigrees, and so on.

Such a hierarchy is very useful to view information from one direction (from the root looking towards the leaves, or from the leaves looking towards the root, or modest combinations of both directions without wandering too far in a sideways direction). But what happens when you want to view your data from more interesting angles? This usually involves heavy performance penalties, because you traverse your information outside of the common channels.

As an illustration, suppose that your information has to do with customers and products. You may be interested in seeing ***all products bought by a given customer*** and you may also be interested in writing to ***all the customers that bought a given product.*** You might also need to review ***all the customers and products handled by a given sales person.***

A hierarchy of abstractions

We'll see some examples that show how convenient it is to use IMAGE to model these challenging cases. Meanwhile, let's use the concept of *hierarchy* in a slightly different context.

Specification

At the highest abstraction level, guiding all other issues, we must consider the fundamental *business logic* that drives our enterprise. This fundamental logic dictates the *specifications* of our design, regardless of the implementations that we choose.

Implementation

At the lowest abstraction level, we deal with the challenges that are specific to the *implementations* of our specifications. We must be sure that we don't get bogged down by housekeeping duties, such as providing privacy and security for concurrent transactions, allowing for diverse networking response times and protocols, and so on.

Housekeeping

Fortunately, in the HP e3000 computing platform, IMAGE (the database management system) and MPE/iX (the operating system) take care of many of these low-level housekeeping chores, but we must still coordinate their performance.

Divide and conquer

It is important to avoid mixing top-level strategic concerns with low-level housekeeping chores which, in substandard systems, can easily vary from implementation to implementation—and even from version to version within one particular implementation.

Just ask your less fortunate friends how many times they have been forced to recompile and/or to relink all of their applications after having "upgraded" a run-of-the-mill database management system (or an operating system, or their "iron").

Daily maintenance

Some of the most vital continuous tasks include setting up dynamic dataset expansion parameters, repacking datasets, managing dataset capacities, properly backing up jumbo datasets and other structures that use a combination of MPE ***and*** Posix files (to avoid orphan files on the backups), etc.

The slavery of the urgent

We should not fall into short-term oriented traps (such as the Year-2000 source of difficulty) that offer an apparent benefit (such as "saving space" by not storing century information) but then extort a burdensome payment when the day of reckoning arrives.

Our design choices always boil down to the dictum: ***Pay me now or pay me later (usually with a steep interest penalty).***

Technical break

The careful (or careless) selection (or default assignment) of design criteria may affect, sometimes dramatically, the performance of our databases.

We have a high investment in hardware, software, staff, and user goodwill. We certainly do not want "minor technical details" to undermine our efforts.

Let's take a break from high-level material and let's get our hands dirty with an implementation-dependent example that will give you the internal flavor of IMAGE's high-performance hashing technology (and its concomitant price when things get out of whack).

Search field hashing (or non-hashing) type

To access a master entry, IMAGE employs two distinct methods of calculating primary addresses.

The first method applies to master datasets with search fields of type I, J, K, R, or E (binary-oriented fields). The low order (right-most) 31 bits of the search field value, or the 16 bits of a half-word search field value, are used to form a 32-bit value. This value is then decremented by one, reduced modulo the dataset capacity and incremented by one to form a primary address.

This method is a more-or-less direct mapping from the value of the search field into an entry number within the constraints of the dataset's capacity. By allowing a sufficient capacity and by assigning search-field values which do not exceed such capacity, you may, in effect, implement your own hashing scheme.

It is perfectly legal to have a search field of type K30 with the first 28 half-words reserved for "alphanumeric data of your own choosing" and the last 2 half-words reserved for the record number which your own hashing algorithm came up with. QUERY, of course, would not like to access such a dataset. But your program will not have any problem with it, provided you have the appropriate scaffolding.

This method tends to produce a relatively high incidence of synonyms, in general, unless you make sure that the distribution of values does not fold back into itself. An example of a bad distribution would be a 32-bit search field with values for events in a given year, with the given year as the most significant digits: 19770016, 19820024, 19920030, 19990053, etc.

The second method of primary address calculation applies to master datasets with search fields of type U, X, Z, or P (character-oriented fields). In this case, the *entire* search field value, regardless of its length, is used to obtain a positive 32-bit value. This value is reduced modulo the dataset's capacity and then incremented by one to form a primary address. The algorithm which is used to obtain the 32-bit intermediate value attempts to approximate a uniform distribution of primary addresses in

the master dataset, regardless of the bias of the master dataset search field values.

The second method tends to produce a lesser incidence of synonyms if compared with the first method, for certain value distributions.

The intent of the two primary address algorithms is to spread master entries as uniformly as possible throughout the address space of the dataset. This uniform spread should reduce the number of synonyms.

Generally (although not always), master datasets with character-oriented search fields have fewer synonyms with prime capacities than with capacities which have many factors.

Migrating secondaries

You may have been puzzled by the apparently irrational behavior of certain *master* datasets. Try this: read, serially, all the entries of a master dataset, doing a *dbdelete* for every entry you read. You expect to have purged *all* the entries, but, to your surprise and chagrin, you may discover that you have a few left over. The same may happen when you delete only entries that meet certain criteria. After you finish, you may find that you have, indeed, some entries left over which should have been deleted.

The solution to this mystery lies in understanding *migrating secondaries*. (Many performance puzzles in masters have to do with understanding *secondaries*, migrating or not, because long synonym chains may leads to serious losses of throughput.)

What are migrating secondaries? In some cases, secondary entries of master datasets are automatically moved to storage locations other than the one originally assigned. This most often occurs when a new master data entry is assigned a primary address which has been previously occupied by a secondary entry. By definition, the secondary entry is a synonym to some other primary entry resident at their common primary address. Thus, the new entry represents the beginning of a new synonym chain. To accommodate this new chain the secondary entry is moved to an alternate secondary address and the new entry is added to the dataset as a new primary entry. This move and the necessary linkage and chain-head maintenance is done automatically by IMAGE but may take a significant amount of time in certain cases.

A secondary migration can also occur when the primary entry (of a synonym chain having one or more secondary entries) is deleted. Since retrieval of each entry occurs through a synonym chain, each synonym chain must have a primary entry residing at the chain's primary address. To maintain the

integrity of a synonym chain, IMAGE always moves the first secondary entry to the primary address of the deleted primary entry. The former first secondary entry is now the primary entry for the chain and the record formerly containing the secondary entry is now empty.

All of these gymnastics happen under the covers. It is good to be aware of these facts whenever you design and maintain high-performance IMAGE databases.

Visualization

Back to high-level design topics. Graphics are great for classifying resources and their relationships. I like to use:

- ***Rectangles*** to represent types of resources (entities tend to be somewhat stable and rectangles convey a feeling of steadiness).
- ***Ovals with outstretched lines reaching out to touch the rectangles*** to represent types of relationships among resources. The *Prolog* programming language uses *circles* for objects and *connecting lines* for relationships between objects. You may consider my *ovals with outstretched lines* as "lines that happen to have a *lump* in the middle" (just for the convenience of being able to write the *name* of the relationship in the lump/oval).

Regardless of the graphics you use to guide your classification, your entities and your relationships will conveniently fall into categories which are obvious to you and to people who are versed in your business.

Since we have all been exposed to standardized samples in the database literature, I would like to treat you to a refreshing new taste.

The "before" look

Here is an example that deals with two kinds of entities—*employees* and *departments*—and with two kinds of relationships between them—*assignment* and *management*. In many books on database management systems, we see this classic example treated along these lines:

Employee#	EmployeeName	Department	Salary	Manager
123	Janice López	Sales	55	Jane Smith
235	Chris Fox	Marketing	90	Sue Plus
813	Max Minim	Design	25	Fritz Peters

This standard treatment is fine from a performance viewpoint, because a given data entry—or row—has everything you want to know about a given employee.

But I believe this approach crams too much into the *Employee* table, thereby obscuring the model. Performance is, most certainly, a worthy goal. Modeling power is another worthy goal. Sometimes, unfortunately, these two praiseworthy objectives are at odds and we must make thoughtful tradeoffs.

In this example, I want to emphasize that a good DBMS should allow us to model anything we want, without forcing a "standard" framework on us. Because most of the database literature explains the highly-unnormalized approach illustrated in the previous table, I would like—for balance's sake—to mention the other end of the normalization spectrum. Any point along this wide range of normalization choices is perfectly acceptable, as long as we know *why* we are selecting it.

The proposed look

We can normalize this table by *separating* its attributes into *four* distinct tables:

1. A table that deals with the attributes of ***employee entities.***
2. A table that deals with the attributes of ***department entities***.
3. A table that deals with the attributes of ***assignment relationships between employees and departments***.
4. A table that deals with the attributes of ***management relationships between managers and the departments they manage***.

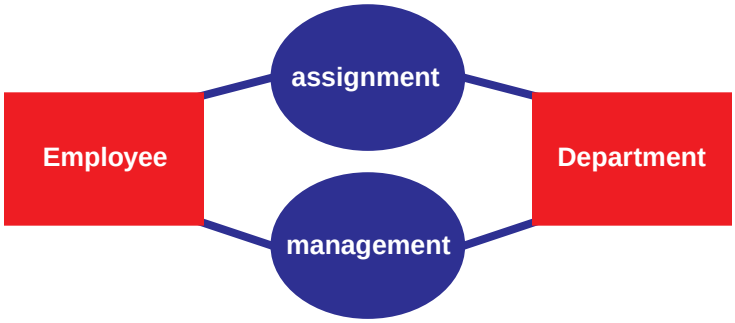
The losses

We use four tables instead of just one table. and we may cause more disc accesses for ***join*** operations—thereby lowering the performance of our database accesses.

The gains

1. We have a more flexible model of reality *and* we do not need to introduce the concept of **nulls** at all. Nulls are a cumbersome idea that some people have proposed to deal with information that is missing from the database. Some of the missing information may be applicable and some may be inapplicable. I have found that, by simply separating entity attributes from relationship attributes, the concept of nulls becomes unnecessary
2. We allow employees to be assigned to zero, one, or more departments—with different salaries for each assignment, including multiple assignments to the same department.
3. We allow departments to have zero, one, or more managers.
4. We allow a given employee to be a subordinate in some department(s) and a manager in some—presumably *other*—department(s), and so on.

Here is a diagram:



The *management* relationship class is really just a specialized kind of the *assignment* relationship class. I diagram it separately for convenience.

This is a table for the *Employee* entity class:

Employee#	EmployeeName	BirthDate	BirthPlace
123	Janice López	19450926	USA
235	Chris Fox	19601203	UK
813	Max Minim	19540514	Perú

This is a table for the *Department* entity class:

Department	Budget	DateOfCharter
Sales	1000	19880213
Mktg	50000	19900203
Design	370	19770212

This is a table for the *Assignment* relationship class:

Employee#	Department	Salary	StartDate
123	Sales	55	19920514
235	Mktg	90	19900405
813	Design	25	19910506

This is a table for the *Management* relationship class. *Employee#* refers to an employee who happens to manage the given department.

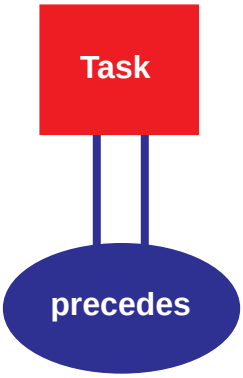
Notice that a given employee may manage zero, one, or more departments—and a given department may have zero, one, or more managers:

Employee#	Department
4528	Sales
4321	R&D
7704	Support

A task-precedence example

Critical-path task management is a particularly clean example that is a delight to model with IMAGE.

In this case, our entities are *tasks* and the relationships among tasks are their *precedence* specifications. Some tasks must be performed before others or we end up wasting valuable resources such as time and money. In construction projects, for example, plumbing must be done before tiling (although everyone has seen plumbers—or other specialists, if trade unions are strong—chipping away at beautiful tiles because some plumbing tasks were not completed properly before the tilers came along).



We have only one type of resources *(tasks)* and only one type of relationships among these resources *(preceding)*.

There is one master data entry for each individual task (and all tasks are consolidated in the *task* master dataset).

You can express relationships in the *active* voice (task A *precedes* task B) and you can also express relationships in the *passive* voice (task B *is preceded by* task A).

There are ***two*** linking detail data entries ("precedes" and "is preceded by") for each relevant relationship among tasks (with all such relationships consolidated in the *precedes* detail dataset). This conveniently allows for the assignment of:

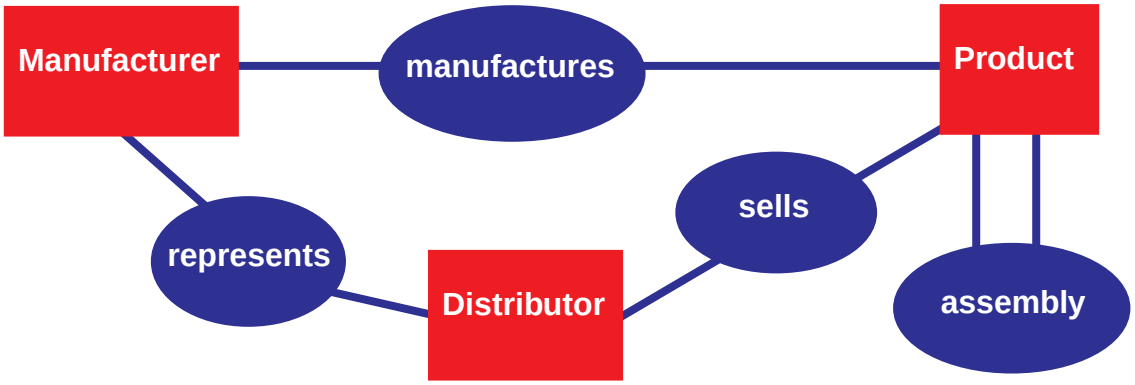
- Zero, one, or more tasks as predecessors for a given task.
- A given task as a successor for zero, one, or more tasks.

This example, modeled with a minimum of database elements, allows us to quickly answer either of these questions—online—with equal ease and performance:

- Which other tasks *must I complete before* I can begin this task?
- Which other tasks *can I start after* I complete this task?

Another example, if you are a manufacturer or a distributor, could proceed along these lines:

A manufacturing example



In this example, we have three types of resources *(manufacturers, products, and distributors)* and several types of relationships among these resources *(manufacturing, representing, selling, assembling)*.

Interestingly, resources are ***nouns*** *(manufacturer, product, distributor)* and relationships among resources are ***verbs*** *(manufacture, represent, sell, assemble)*.

Regarding resources:

- There is one master data entry for each individual manufacturer (and all manufacturers are consolidated in the *manufacturer* master dataset).
- There is one master data entry for each individual product (and all products are consolidated in the *product* master dataset).
- There is one master data entry for each individual distributor (and all distributors are consolidated in the *distributor* master dataset).

You can express relationships in the *active* voice:

- Manufacturer *manufactures* product.
- Distributor *represents* manufacturer.
- Distributor *sells* product.

You can also express relationships in the *passive* voice:

- Product *is manufactured by* manufacturer.
- Manufacturer *is represented by* distributor.
- Product is *sold by* distributor.

There is one linking detail data entry for each relationship between a manufacturer and a product (and all such relationships are consolidated in the *manufactures* detail dataset). This conveniently allows for the assignment of:

- Zero, one, or more products for a given manufacturer.
- Zero, one, or more manufacturers for a given product.

There is one linking detail data entry for each relationship between a manufacturer and a distributor (and all such relationships are consolidated in the *represents* detail dataset). This conveniently allows for the assignment of:

- Zero, one, or more distributors for a given manufacturer.
- Zero, one, or more manufacturers for a given distributor.

There is one linking detail data entry for each relationship between a distributor and a product (and all such relationships are consolidated in the *sells* detail dataset). This conveniently allows for the assignment of:

- Zero, one, or more products for a given distributor.
- zero, one, or more distributors for a given product.

There are ***two*** linking detail data entries ("contains" and "is contained by") for each relevant relationship among products (and all such relationships are consolidated in the *assembly* detail dataset). This conveniently allows for the assignment of:

- Zero, one, or more products as components for a given product.
- A given product as a component for zero, one, or more products.

This example of a *bill of materials,* modeled with a minimum of database elements, allows us to quickly answer either of these questions—online—with equal ease and performance:

- Which products ***do I need*** to assemble this product?
- Which products ***can I assemble*** with this product?

Turning your design graphics into an IMAGE schema

Please take these general guidelines with a grain of technological salt. For performance reasons, it may be reasonable to use other types (or combinations) of data structures and indices, but you can certainly begin, as a first cut, with these thoughts:

- Rectangles ***(collections of entities)*** can be represented by master datasets, which are optimized for hashed access

(i.e., you can find *a given data entry* very quickly according to the value of its search field).

- Ovals ***(collections of relationships)*** can be represented by detail datasets, which are optimized for chained access (i.e., you can find *a given group of data entries* very quickly according to the value of their search fields).

Obviously "hot" relationships can be made to perform like champions by means of paths (which use IMAGE's chaining shortcuts to find, with high online performance, the entities and their relationships that we want at any time, regardless of their physical location).

"So-so" relationships are, by definition, not worthy of paths. These lukewarm relationships will rarely pop up in daily online database usage. If they surface every now and again, they will become the subject of serial scans (which are not so bad if we do them in batch mode only once a month in the middle of the night). If we notice an alarming trend in the rate of serial scans, then we simply add a path to minimize waiting time.

Indexing: The Key to Performance

Typically, we are interested in accessing a group of entries from a database (for instance, "all the outstanding orders from customer XYZ").

One approach is to scan the database serially, beginning with the first entry and ending with the last entry, "running into" the desired entries along the way. If we have millions of entries, with only a few that meet our selection criteria, we may not be able to afford to use this approach for on-line applications.

Another approach is to use indexing methods that allow us to jump directly into the entry or entries which interest us without having to wade through millions of irrelevant entries.

If performance is not a problem, we can always keep our information in simple tabular form. But if performance is an issue (particularly if you have millions of data entries) you may want to take advantage of smart database structures, such as those offered—but not forced upon you—by IMAGE (i.e., you may want to keep your information in "sophisticated tabular form").

Breaking free from indexing traps

There are several types of indexing methods, with various advantages and disadvantages, just as there are many kinds of database management systems. But let's not be confused by this apparent variety. Deep down inside, all databases are nothing more, or less, than bunches of bits. All indexing schemes are, by the same token, attempts to shortcut the route that leads us into certain desired bunches of bits within a database. The only purpose of an indexing system is to serve as a performance booster and we should be able to add, maintain and delete indices quickly and conveniently.

As long as we keep these fundamental concepts straight, we will be able to take advantage of indices when they exist, without having a nervous collapse when they are gone. Let's take one paragraph from Hewlett-Packard as an exercise in going back to basics. A while back, in an issue of HP's I*nformation Systems & Manufacturing News,* Terrie Murphy wrote in an article on ALLBASE:

> *HPSQL's simple tabular-data structure, with no predefined data-access paths, significantly increases database-administrator (DBA) and programmer productivity. DBAs have great freedom in structuring the database, since it is not necessary to predict all future access paths at design [time]. If the data is available in the database, it is immediately accessible at any future time. In non-relational models, all access paths need to be known when the database is designed. This adds significantly to overall program-development time. In addition, with no predefined data-access paths, the data structure can be modified in many ways without affecting existing programs; thus greatly simplifying application maintenance.*

The issue is "predefined access paths", as viewed from an ALLBASE perspective. We can easily rewrite the same paragraph from an IMAGE viewpoint:

> *IMAGE's simple tabular-data structure, with (or without) predefined data-access paths, significantly increases database-administrator (DBA) and programmer productivity. DBAs have great freedom in structuring the database, since it is not necessary to predict all future access paths at design [time]. If the data is available in the database, it is immediately accessible at any future time. In IMAGE, all access paths need not be known when the database is designed. This saves significant overall program-development time. In addition, with (or without) predefined data-access paths,*

the data structure can be modified in many ways without affecting existing programs; thus greatly simplifying application maintenance.

Without too much effort, we can also rewrite this paragraph so that predefined access paths appear as tyrants or as liberators. It's all a matter of political "spin" and marketing hype.

IMAGE has undeservedly gotten bad press regarding indexing and predefined access paths. In fact, IMAGE allows you independence from predefined access paths (and from many structural modifications), provided you follow some sensible guidelines.

As a prerequisite, you should be aware of several IMAGE design criteria that people tend to ignore:

1. An IMAGE dataset is a simple tabular data structure. The widespread belief that IMAGE is a "pointer-based network DBMS" is not true. You can build an IMAGE database that does not have any pointers whatsoever.
2. The IMAGE intrinsics that allow you to add, access and update entries (*dbput, dbget, dbupdate*) have an important parameter: the list of those specific fields that interest you.
3. The IMAGE *dbinfo* intrinsic gives you a wealth of information at run time.

Access lists

The order of keys and/or attributes in an entity (or in a relationship) is arbitrary. Therefore, the sequence of fields in an IMAGE data entry is also arbitrary.

To allow for stability within this flexibility, IMAGE provides the ***list*** construct to map any subsets and permutations of key(s) and/or attribute(s) to/from a program's buffers. This permits us to add, delete, or reshuffle fields without the need to recompile all the programs which access the affected dataset(s). We must recompile only those programs which explicitly access any fields that we have added or deleted.

This gives us a high degree of data independence, if we use late-binding techniques at run time (as opposed to hard-wiring everything into our programs at compilation time).

Late binding

Knowing these (and other) IMAGE design criteria is necessary but not sufficient. As another prerequisite, you should use high programming standards (this, naturally, applies to any kind of computer work that you do). A very important programming standard is that you should postpone binding as much as possible. This means that you should not burden your programs, at compilation time, with hard-wired stuff. You should wait until run time to adjust to the prevailing conditions of the day.

In the case of predefined access paths, if any, you should not even think about including (or excluding) them in the strategy of your programs. You should find out, at run time, whether a given field in a given dataset is an IMAGE search field or not (using *dbinfo*). If you are not dealing with a search field, you might have to do a serial scan of the whole dataset (using *dbget* mode 2 or 3) to find those entries, if any, whose field values you want. (You are certainly free to develop non-IMAGE indexing schemes to avoid such serial scans.) If you are dealing with an IMAGE search field, you can be much more efficient. For a master dataset, use hashing (*dbget* mode 7). For a detail dataset, use an IMAGE-provided combination of hashing and chaining (an initial *dbfind* followed by *dbget* mode 5 or 6).

If you follow these reasonable guidelines, your applications will be totally immune to changes in access paths. You will be able to add or delete paths at will, to suit the performance needs of your users. And, as a fun bonus, since the only difference between masters and details is *access method,* you will also be able to change masters to details or details to masters without impacting any of your application programs.

What do you think now about Terrie's assertion that "In non-relational models, all access paths need to be known when the database is designed"? I am sure Terrie meant to qualify this statement by adding, "if your programming standards are so low that you hard-code everything."

This hard-coding issue applies equally well to SQL, of course. If you hard-code in SQL, nothing will save you from getting into deep trouble. Let's illustrate this observation.

Avoid the "everything" default

In the case of adding, accessing or updating IMAGE entries, you should not even think of using "@" to specify the list of fields that interest you.

The "@" list asks IMAGE to deal with *all the current fields in the dataset* (for which your security class is authorized). If you add, delete or shuffle the fields of a dataset (or if you change your *dbopen* security class), you must edit and recompile all the programs that access that dataset.

If you add new fields, you risk dangerous buffer overruns (bounds violations) which may or may not be detected automatically at run time.

Absolutely the same is true in SQL if you use "*" instead of a specific list of columns.

Since this prospect does not attract me, I follow a strict methodology with IMAGE field lists. Even though it may take a little more effort up front, I always build a list with the ***names*** of those specific fields that the program needs to access (I prefer to look at names—rather than numbers—in my source code).

The first time I invoke an access intrinsic (*dbput, dbget* or *dbupdate*) on a given dataset, I pass it this list. Afterwards, when I invoke an access intrinsic (on the same dataset) that depends on the same list, I pass it IMAGE's asterisk list ("*"), which tells IMAGE "don't bother to assemble and check my list—simply reuse the previous list."

(The asterisk "*" means different things to different people and it is important to remember that SQL interprets the asterisk to mean "give me everything.")

For more than two decades now, I have been able to add, delete and shuffle fields in my IMAGE datasets. Even though this fact, in itself, is significant, it is even more impressive because I have not been forced to edit or recompile those programs that don't use such fields.

What do you think now about Terrie's opinion that "[with SQL] the data structure can be modified in many ways without affecting existing programs"? Of course, Terrie meant to qualify this opinion by adding, "provided you don't use the SQL asterisk ('*') instead of a specific list of columns in your SQL statements."

Performance and maintenance benefits of explicit lists

There is more to access lists than just flexibility. There is performance!

When you request the transfer of a data entry, IMAGE needs a list of the fields that you wish to transfer. This list is implemented as an array containing an ordered collection of data item identifiers, either names or numbers. (Any search/sort fields defined for the entry *must* be included in the list.) IMAGE will transfer from/to your own program's buffers only those fields specified in the list, in the order specified in the list, regardless of the number of fields in the data entry and regardless of the positions of these fields in the data entry.

Since IMAGE must enforce security regulations, it must make sure that you are authorized to access the fields you specify in your list. IMAGE must also make sure that your list contains valid field designators for the dataset in question.

All these checks take time. As a matter of fact, the IMAGE manual states that list processing is a relatively high overhead operation. Therefore, you should pass to IMAGE an explicit list only *once*, at the very beginning of your repetitive accesses to the same dataset. Thereafter, you should pass an asterisk as a request to IMAGE that it should use the previous list (which it has saved as the "current list" in its internal tables).

In general, names for fields are easier for program maintenance purposes. Data item numbers are more efficient and compact, but tend to be more obscure and dangerous, since data item numbers will change if you delete unreferenced data items from the middle of the item table (or if you add new data items in the middle of the item table).

There is a problem if your data entry has many fields and the names of these fields all have 16 characters. Limitations on buffer size for character-type variables may not allow you to hold such a gigantic name-oriented list. In this case, an integer array to hold a numeric list is better. But you can still do everything in a name-oriented way!

An elegant solution is to call *dbinfo* with item *names* in order to find out their corresponding item *numbers*, which you then use in your numeric list. This gives you the best of both worlds.

Structural freedom

By binding as late as possible, we gain two kinds of freedom: the freedom from predefined access paths and the freedom from rigid data structures.

We are able to add, maintain and delete indices quickly and conveniently. We can use the indices that are "bound" with the official DBMS (such as hashing & chaining and B-Trees in IMAGE) and we can use our own (or third-party) indices to complement the official indices.

Indices are only one aspect of the general database structure. We are also able to add, maintain, reshuffle, and delete any other database objects, such as datasets, data items, fields, paths, sort features, security classes, and so on, with minimal impact (if any) to our current applications.

High performance, high availability, economy

We want to design, maintain, and orchestrate databases which perform well under heavy-duty use. There are all kinds of database management systems, in all price ranges and with different requirements. Some have ravenous appetites regarding hardware resources and human attention while others, such as Hewlett-Packard's IMAGE/SQL, are lean and mean.

Scott Hirsh once said that using the HP e3000 computer and its IMAGE database management system was ***spectacularly uneventful***. This is obviously boring, but would you rather get some unwelcome excitement into your business-critical computing environment? Reserve such treats for your competition!

IMAGE provides proven solutions to key issues for the elite business-critical market:

1. *High availability* (reliability, robustness, resilience, few demands on your time and attention). If you are tired of spending your life fiddling with fragile and temperamental systems, IMAGE is for you.
2. *High performance* (heavy-duty transaction throughput, very short on-line response time per transaction). If your users are tired of spending their lives waiting for "the system" to respond (or to come up after a crash), IMAGE is for you.
3. *High concurrence* (support for hundreds or thousands of simultaneous clients via Macintoshes, Unix workstations, PCs under Windows or DOS, as well as simple terminals, either locally or remotely, through direct serial connections or through your company's Ethernet or Intranet, as well as via the worldwide Internet.). If your current system breaks down after it exceeds a light workload, the IMAGE workhorse is for you.

Interoperability

There are two camps in the standards battlefield. Some people want to ***standardize components*** while other people, more realistically, want to ***standardize interfaces***.

Sometimes, "the most popular" component—according to a given set of someone else's criteria—is not necessarily the most appropriate for your circumstances. In this case, you should wisely choose an *elite* element (such as IMAGE) for specialized needs (such as business-critical computing).

The key is interoperability: there should be different computing platforms, operating systems, and database management systems, each suited for a particular purpose, and they should all be able to exchange information easily, reliably and economically.

In a truly open environment, the question is one of appropriateness, since you want the freedom to use the best available tool for each specialized purpose. It would be foolish to say that IMAGE is better than non-IMAGE for all cases, and it is foolish to say that non-IMAGE is always better than IMAGE. You should be able to choose an open mix of IMAGE and non-IMAGE according to your needs (and resources). And all of your applications should be able to intercommunicate.

IMAGE allows this flexibility, because you can write custom programs in a variety of computer languages and you can use middleware that takes advantage of SQL, HTML, Java, ODBC, JDBC, ADBC, and so on.

The fact that you choose IMAGE for high-performance and high-availability elite applications does not preclude you from *also* using other widespread computer platforms and database management systems in your enterprise's bag of tricks.

You can use IMAGE as a database server in a worldwide Internet client/server environment, even if the environment is severely limited to using only SQL as its common database language. Naturally, if you want to enjoy the legendary speed of IMAGE, you can always communicate with it via its native high-performance intrinsics. It is your choice. You can enjoy the best of both worlds.

Multitasking and multiprocessing

MPE/iX, the operating system for the HP e3000 computer, has been a ***multitasking*** platform from day one, with ***multi*** built into its name from the very beginning (the "M" in MPE), not tacked on as an afterthought. MPE is ***multi***-user as well as being a ***multi***tude of other useful things.

Posix

MPE/iX is Posix compliant (with the "iX" qualifier, we have a *Multi Programming Executive with integrated Posix*).

Posix is an operating-system interface definition with deep roots into the Unix operating system and into the C programming language.

Here is a bit of good-natured technological trivia. There is a lot of hype behind Unix, but it contains ancient commands such as "isatty" (you may be forgiven for not remembering that "tty" stands for the ten-characters-per-second teletypewriter of the 1960s, the pre-legacy days of computing, when Unix began as an operating system for ***one*** user—hence ***uni***—as a contrast to Multics, a multiuser operating system). For a Multics chronology, visit:

http://www.lilli.com/chrono.html

MPE/iX supports all standard Internet protocols, including HTTP for web serving directly from your IMAGE databases.

Internet protocols

The HP e3000 has a Java platform that enables convenient (and protected) access to your valuable IMAGE databases from any device that uses Java (such as your Mac, your PC, your Unix workstation, your cellular phone, your pager, and so on).

Java

Let's examine just one example that comes to my mind in Sun Valley, a few minutes before hopping into my car to drive to the airport for a flight to California to visit Hewlett-Packard.

Beware of hype

Since the 1980s (before it became fashionable), I have lived in a farm surrounded by national forest at six thousand feet up in the mountains. The farm's converted barn is home to all kinds of computer equipment, including Adager's Internet setup. As a healthy contrast, I can see elk and deer while I write this essay.

Every now and then I drive twenty minutes to the airport without encountering any traffic worth mentioning. Then, I land in California's San Francisco Bay Area and feel transported to a different world, so vividly described by James Burke in his book *Connections:*

> *The prime examples of man's love-hate relationship with technology: the motor car, which makes mobility possible, and the traffic jam, which makes it impossible.*

Unlimited mobility, as promised by car salespeople, is the hype. A ***traffic jam*** (conveniently ignored by car salespeople in their spiels) drives home the reality of life-after-hype.

Beware of hype: What some people call ***instant credit*** is really ***instant debt***.

You want to liberate yourself, not to become a slave to the trend of the day.

Are you going to go for hype or for suitability? The choice is yours at any time, including the opportunities that you have ***right here*** and ***right now***. Choose carefully.

The bottom line

IMAGE is remarkably *economical* to own and operate in terms of hardware resources and human attention. It works reliably and frugally and does not require an army of expensive prima donnas to keep it up and running. There are many sites that support—with a couple of normal individuals—many concurrent users linked to a single HP e3000 IMAGE database server.

But IMAGE, by itself, is not sufficient. It is just a tool and you, as s human being, must still use it wisely.

You can reap the benefits of IMAGE only if you do a good job designing your databases and maintaining them in a healthy state. Common sense will be your best ally (provided, of course, that you know how to read the appropriate technical manuals).

Having something is totally different from *being able to use that something when the need arises.*

Analogously, *knowing what to do* is not the same as *doing what you know.* And, even if you know what to do and do what you know, perhaps that is *not* what you ***should*** do.

After all, there is such a thing as ***appropriateness***. A British story drives this point home:

> *The young executive leaving at 6 PM found the president of the company standing, looking puzzled, with a piece of paper in front of the shredder.*
>
> *"This document is very important, and my secretary has left," said the president. "Can you make this bloody thing work?"*
>
> *"Certainly, sir," said the young executive, who turned on the machine, inserted the paper, and pressed the start button.*
>
> *"Excellent, excellent," said the president as his paper disappeared inside the shredder. "You can go home now, as I just had to make one photocopy as a backup."*