Database Dynamics

F. Alfredo Rego

Adager

Apartado 248
Antigua
Guatemala

The disciplines of Database Dynamics deal with the normalized design, maintenance and orchestration of databases which perform well under heavy-duty use.

In this essay, we see specific examples based on IMAGE, the award-winning database management system built by Hewlett-Packard for the family of HP3000 computers. These examples illustrate fundamental principles which are simple, timely, timeless and, above all, powerful.

Database buzzwords

A database models the dynamic behavior of entities and their relationships by means of *entries*. An entry consists of a key (which uniquely identifies the entity or relationship) and a collection of attributes (which give quality and color to the entity or relationship).

Entities and relationships don't just sit there. They interact with one another and with their environments: Transactions happen which affect (and are affected by) entities and relationships. Such transactions include changes in database structure as well as changes in the meaning and value of the information maintained by the structure.

We cannot store a real entity or a real relationship in a database, just as we cannot store a *real* orchestra in a stereo cassette. At best, we can hope to store 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 characteristics which, in the case of databases, we call *fields*.

A *dataset* is a homogeneous collection of entries. There are different kinds of datasets, each optimized for a specific access technique. In IMAGE, we like to use *master* datasets to keep entities and *detail* datasets to keep relationships. (IMAGE master datasets come in two flavors: manual masters and automatic masters. Please see the IMAGE reference manual for specific details). Naturally, we may use all kinds of conglomerates of physical datasets (masters and/or details) to represent logical master *or* detail datasets. It all depends on our choices of specialized indexing techniques.

To make an IMAGE database functional, we access its entries in a variety of ways, ranging from serial scans of entire datasets (*the only way to go* in the good old days of batch machines) to hashing and chaining (quite convenient for online applications). Hashing and chaining are techniques based on direct access to specific addresses so that we may jump directly into the entry or entries which interest us without having to wade through millions of irrelevant entries. Please see the IMAGE reference manual for a detailed discussion of hashing and chaining.

The database challenge

Fundamentally, we are interested in two database operations: the addition of new entries and the finding of existing entries (so that we may relate them, report them, update them, or delete them). A Database Management System (DBMS) attempts to help us in the pursuit of these worthy objectives.

Structure vis-a-vis interface

For *efficiency's sake*, a DBMS has some type of internal structure to find and assemble entries. For *convenience's sake*, a DBMS has some type of user interface to create, maintain, and relate entries to produce, somehow, information on a timely basis. The resulting entries which the user "sees" through the interface may be real (if they exist physically in the database) or virtual (if they are the result of relational operations on real or virtual entries).

The user interface serves as an ambassador between the raw bits-and-bytes computer stuff and the human-like specifications of the end-user. A poor interface imposes the restrictions of the structure upon suffering users. A good interface shields users from the structure's shenanigans, while still being able to take full advantage of the structure's properties. A good user interface is as efficient as possible without being obnoxious. An interface knows the internals of the database structures as well as the externals of the user desires, and spends its existence translating back and forth between bits and thoughts. This may very well be the fastest kind of shuttle diplomacy!

Complexity and normalization

Ideally, things should be simple. Unfortunately, though, things *are* complex. But we should avoid *unnecessary* complexity. This is the objective of normalization, in the mathematical sense.

Normalization is the breakdown of seemingly-complex operations into simpler processes. The challenge, at the beginning, is to place the appropriate resources (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:

- entry
- dataset
- database
- computer
- node
- network

A normalized structure is open-ended. We can add more elements at any layer 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 communications and coordination costs. Normalization is neither good nor bad. It is simply a method which allows us the freedom to choose our favorite spot in a spectrum (or rainbow), 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 synchronization. This is analogous to a one-man band who must play all kinds of disparate instruments in a (more or less) coordinated fashion.

Usually, efficiency in terms of maintenance implies simplicity of roles and a multiplicity of role-players. If we want to change one role, we only have to change one player. But it can be a drag to keep track of thousands of players. This is analogous to those fascinating groups of musicians who play bells, one bell per musician. Each person is a specialist who can only play one note. In terms of maintenance, we can see how difficult it would be to tune a complex instrument during a performance and how easy it would be, on the other hand, to simply exchange a bell which is out of tune.

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, like a one-man band.

Even though it is paradoxical, our experience shows that normalized databases may actually occupy less total disc space than unnormalized databases. Particularly if the keys are short, which, fortunately, seems to be the case most of the time. For instance, your identity number is probably shorter than your job description. Naturally, we can go to ridiculous extremes and normalize a database to death. We could conceivably chop up the information about an employee in many entries, each containing a single attribute such as name, birth date, salary, and so on. But this would really be splitting hairs! Common sense should prevent us from committing such atrocities, and this is the motivation behind the rules for the fifth normal form: An entry is in fifth normal form when there is nothing significant left to normalize!

Access strategies

In an online database system, we want to get information about given entities and their relationships while somebody waits over the counter or over the telephone. This means that we want to find the entry (or group of entries) of interest to us, among millions of entries, as efficiently as possible. We should design (and periodically tune) our database systems to provide the fastest possible response time for the most important transactions and queries.

Some people have spent endless amounts of time and talent on a fascinating problem: How to minimize the effort required to answer the most infrequently-asked (and most arcane) questions. Other people have invested their time and talent on another problem: How to minimize the effort required to answer the most frequently-asked (arcane or not) questions, while still preserving a reasonably efficient environment for those who must toil, on a daily basis, with the thankless task of feeding and baby-sitting the database.

IMAGE provides two access methods which are optimized for efficiency: hashing and chaining. We may access entities (in master datasets) by means of hashing and we may access relationships (in detail datasets) by means of chains which IMAGE maintains for us as we add or delete detail entries. These are *contents-oriented* access modes (as opposed to *address-oriented* access modes, such as serial or directed).

IMAGE allows us the freedom to go "explorer-like" with sequential and direct access methods. It also allows us the convenience of traveling through "pre-established hubs" by means of techniques such as hashing and paths. We do not have to access anything in a predetermined way. But it is nice to know that we may do so, if we know that a given "routine-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?

Naturally, we may have valid reasons (usually having to do with convenience, performance, or both) that motivate us to use our own combinations of physical master and detail datasets, with or without physical paths, to model a given collection of entities and/or relationships. Usually, these valid reasons are dictated by our choices of customized indexing techniques which we build on top of IMAGE's intrinsic access modes. (IMAGE itself does not have indexing. IMAGE provides pre-fabricated access methods which allow us to implement all kinds of indexing strategies, according to our pleasure).

Entities, relationships, and keys

In terms of space, an entity may be related to zero, one, or more entities (of its own class or of different classes). In terms of time, these relationships may happen all at once or they may happen one after another, in a strictly sequential fashion. To make things more interesting, some virtuoso relationships may come all at once in an unending sequence of complexities!

A relationship *is* an entity. It all depends on our viewpoint. For example, a marriage is a *relationship* between two people, and a marriage is also the subject of attention of a marriage counselor who treats it as an *entity*. By the same token, an entity *is* a relationship. For example, an individual is an *entity*, and an individual is a *relationship* formed by internal organs, genes, environment, and so on. It is a matter of *convenience* to designate some "thing" as an entity or as a relationship.

Usually, a relationship's key is a concatenated key, composed of a collection of the keys of the related entities. If we can relate the same entities in different ways or under different circumstances (thereby giving rise to several detail entries to represent the different relationships), then each relationship's key must include some additional attribute(s) which define the differences. For example, consider *discretionary* pricing (or *discriminatory*, or whatever you may want to call it). In this case, the price of a product for a customer may depend on the part's supplier, on the customer's rating within the company, on the order date and/or on the ship date, and so on. In other words, the price is an attribute of the relationship among all these entities; the price is not an attribute of the product alone.

IMAGE's implementation highlights

An IMAGE entry (master or detail) models an entity or a relationship with equal ease. The only difference between a master entry and a detail entry is the method of access: master datasets are biased for hashing while detail datasets are biased for chaining.

An IMAGE dataset (master or detail) is a homogeneous collection of entries and an IMAGE database is a homogeneous collection of datasets. Since the fundamental atomic unit is the entry, let's review its main features. An IMAGE entry has:

- A unique identifier (key) for the represented entity or relationship;
- Attributes (if any) which further qualify the characteristics of the entity or relationship.

Please note that a key is simply a field (or a collection of fields) which uniquely identifies an entry. A key does *not* have to be an IMAGE search field. IMAGE search fields are defined only for performance's sake, to allow paths between master and detail datasets. Paths are particularly attractive for online access to fashionable relationships, since paths tell IMAGE to maintain appropriate physical linkages when adding or deleting entries.

Since entities and relationships are equivalent, IMAGE uses the same construct ("entry") to represent either an entity or a relationship. For convenience (and performance) you may want to use master datasets as repositories of entities and detail datasets as repositories of relationships, since masters are biased for hashed access while details are optimized for chained access. This would allow you to pick out the entry that interests you *right now* (by means of hashing into the master) and would display its relationships *right now* (by following chain links in detail datasets, controlled by chain heads in the master entry).

The order of keys and/or attributes in an entity (or in a relationship) is arbitrary. Therefore, the sequence of fields in an IMAGE 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 the user'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 the affected fields. This gives us a high degree of data independence.

Database Dynamics

The concepts of *Database Dynamics* deal with the orchestration of the transactions which affect (and are affected by) databases. A transaction is something that adds, deletes, or modifies an entry (or a collection of entries, in the case of a complex transaction).

We use *picoseconds* (trillionths of a second) to measure events which we think are super-fast. We use *aeons* (billions of years) to measure events which we think are super-slow. Somewhere in the middle of this wide spectrum we find the events which occupy most of our attention in our daily concerns. By definition, these are the events which are the most useful and interesting. Most IMAGE databases, for instance, keep track of entities and relationships whose event-speed ranges from a "fast" which we can measure in hours to a "slow" which we can measure in years.

The functional dependencies among keys and attributes will tend to show a remarkable stability, particularly if you cluster things around entities and relationships which are obvious to you. For instance, the functional dependency between a personal identification number and the name of a person will probably hold for life. Nevertheless, the particular manners in which people access, combine, manipulate, present, and otherwise massage the data contained in the database will tend to change according to the inevitable shifts in the organization's political winds.

Given these facts of life, it might make more sense to focus our limited energies and resources on the analysis of the most permanent things: entities and their relationships. As a bonus, we find that this entity-relationship approach automatically and conveniently requires very simple interfaces to maintain (and obtain) information using the database.

Naturally, stability should not imply inflexibility. The challenge is to be as stable as possible while still being sufficiently flexible and adaptable to changing environmental conditions. But there should be some back-bone to the whole thing!

A practical database methodology

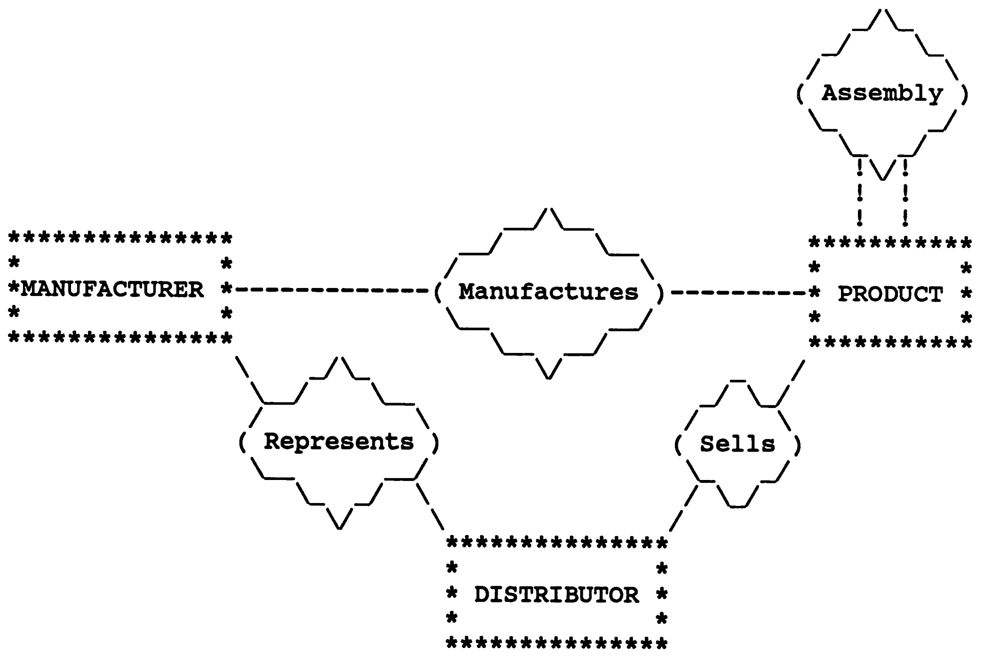
All this nice database theory is certainly a lot of intellectual fun. But you *also* have to address the practicalities! Specifically, you have to remember that your ultimate responsibility is to develop an application system which uses databases only as a means to an end.

Since 1974, I have kept copious notes of theoretical and practical issues which have influenced my failures as well as my successes. The integration of these notes has led me to a practical database methodology whose ideas and steps I consider simple, timely, timeless and, above all, powerful. Here is the outline:

First of all, classify your Entities and your Relationships

Graphics are great for the process of classifying *and* for displaying the resulting classification! I like to use *rectangles* to represent collections of entities, *circles* to represent collections of relationships, and *lines* to make relationships explicit. Since entities and relationships are equivalent, this is a valid choice of geometric figures: After all, a rectangle and a circle are topologically equivalent!

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. For instance, if you are a manufacturer or a distributor, you could choose something along these lines:



Notice, with pleasure, that this fundamental step of classifying your entities and relationships has all kinds of bonuses. First of all, you will get a clearer picture, in your own mind, of your own system. Later on, you will also see that the resulting IMAGE database(s) will automatically have a clean and elegant design and will be in a very respectable state of normalization.

As an interesting example of a bill-of-materials, modeled with a minimum of database elements, please see the "Assembly" relationship which relates "products" to "products" (or, if you prefer, you may use "parts", or "components", or whatever, instead of "products") so that we may quickly answer either of these questions with equal ease: "Which products do I need to assemble this product?" and "Which products can I assemble with this product?".

Let's see now an example which stresses the importance of the attributes associated with *relationships*. Even though the related *entities* are important in themselves, we will see that shuffling things around a little bit to place the spotlight on the relationships may reward us with pleasant surprises. In this example, we will take attributes which are commonly assigned to entities and we shall assign them to relationships. This way, the entities are free to "wear" different attributes, depending on their relationships, without being stuck with them for life. This is the essence of dynamism, after all! Just for fun, let's study a database model of presidential administrations, on a world-wide scale. Please stretch your mind beyond any parochial limitations:

```
                         ********
                         *      *
                         * YEAR *
                         *      *
                         ********
                           !  !
                           !  !
                           !  !
                          _!__!_
***********           ___/      \___           *************
*         *          /              \          *           *
* COUNTRY *         / Presidential   \         * PRESIDENT *
*         *-------( Administration    )-------*           *
***********         \                /         *************
                     \___        ___/
                         \______/
                            !
                            !
                            !
                            !
                         *********
                         *       *
                         * PARTY *
                         *       *
                         *********
```

Notice that this design does not include restrictions such as *citizenship* and *uniqueness*. The same person could be the president of more than one country at the same time and many persons could be simultaneous presidents of the same country (have you heard recently of "presidents in exile"?). You can quickly find an administration by beginning year or by ending year, as well as all current administrations regardless of their beginning. Without any radical changes, this same design could apply to directors of corporations (and would be very useful to trace interlocking boards!)

Time for a little computerese!

Translate your nice graphics to IMAGE's database definition language. Create an IMAGE *schema* which the schema processor (DBSCHEMA) will understand. Rectangles ("collections of entities") translate immediately to master datasets (either manual masters or automatic masters, depending on your orchestration style; don't lose too much sleep on this). Circles ("collections of relationships") translate immediately to detail datasets. Lines which represent obviously "hot" relationships translate immediately into paths (so that we may use IMAGE's hashing and chaining shortcuts to find the entities and relationships which we want at any time). Lines which represent "so-so" relationships are, by definition, not worthy of paths. These lukewarm relationships will rarely pop up in our daily database usage. If they surface every now and again, they will become the subject of serial scans (which are not so bad if you do them 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. No big deal!

Refine your indexing for performance

IMAGE's *search fields* just happen to be convenient for the sake of IMAGE's hashing algorithm (which converts a data value to an address) and IMAGE's chaining algorithm (which links logical neighbors even when they are millions of entries away from each other). But you can design *any* mathematical mapping of your choice that will convert any data value into a reference to whatever keys you may have defined for IMAGE. Don't stop at the obvious. Let your creativity soar to new heights. IMAGE will be delighted to cooperate with you. If you need some inspiration, simply ask a fellow HP3000 user. You will be amazed at the unbelievable variety of IMAGE indexing techniques in existence today. It is *fun* to do fantastic things with IMAGE.

For instance, you may get sophisticated and decide to use *clusters* of masters and details to index selected entries which reside in specific masters or in specific details. If your indexing incorporates trees and other structures which facilitate keyed sequential access to entries, you may consider stand-alone detail datasets as ideally suited for the storage of such specialized structures. There is no reason to have indexing structures residing *outside* of IMAGE if we can have them as full IMAGE citizens. A stand-alone detail dataset is equivalent to a standard MPE file and offers many additional advantages, such as IMAGE's buffering, backup and protection mechanisms, locking, and remote access.

Choreograph your Transactions

This is the dymanic part! Specify the transactions that will allow you to add, modify, delete, and report these entities and their relationships. Decide whether or not some of these transactions need to be undisturbed by other concurrent transactions. Take advantage of IMAGE's locking to make sure that you achieve a fair compromise between *privacy* and *sharing*.

Perform your Transactions

At your convenience, add, delete, find, modify, relate, and report entries. Do it solo or invite your friends and fellow workers, from the next desk, from the next building, from the next country, or from anywhere in the network. IMAGE is a multi-tasking multi-computer database management system, after all!

A good performance implies the orchestration of a myriad of simple technical details into an impressive, overwhelming presentation. The presentation *is* your application. The technical details are the result of your normalization. Since you carefully allocated the appropriate resources (no more and no less) where they belong, at the appropriate level, at the appropriate place, at the appropriate time, you have a balanced performance.

Tune up your Performance

Balance, though, by its very nature, is a dynamic concept. You cannot just relax and assume that you will never lose it!

As you specify your masters, your details, and your paths, keep in mind that the important question is: "Can you define, redefine or cancel these entities and their relationships at any time during the life of the database?" For performance reasons, you may want to wire some *obvious* relationships *hot* in the database's structure by means of paths. But you do not want to be stuck for life, since some hot relationships may cool off and some sleepers may wake up unexpectedly!

The same questions apply to every component of your database and the same advice applies to every database administrator: Fine tune things in such a way that you reach a reasonable compromise between the *response time* for any of these functions and the *global throughput* for the whole transaction load.

Bravo! You are now a database master, thanks to IMAGE.

Hewlett-Packard's IMAGE database management system has unique mathematical properties which are natural consequences of its original design criteria. These IMAGE properties allow you to model your entities and their relationships in a nicely normalized fashion, without any unnecessary and inconvenient convolutions.

Take advantage of IMAGE's properties. They are sound and they are classic. But they do not have a life of their own. They need you!

Like any fine instrument, IMAGE is there, dormant, waiting for you to wake it up with *your* dynamism. Play it well, with soul. At the beginning, you may want to join a group of fellow enthusiasts, to improve your technique while you develop your style. Eventually, you may want to solo. In any case, happy crescendo!

Appendix A

IMAGE schema for the *Distributor* database mentioned in the *Practical Methodology* section.

```
Begin database DISTR;

<<

NOTES:

 Your imagination and convenience should decide how many (and
 which) attributes to include at the "...".

 The paths are NOT necessary at all, but we include them as
 examples of performance boosters for relationships which
 seem to be "hot and heavy".  You can always take ALL paths
 away, or add OTHER paths at will.  IMAGE does not care: You
 are free to play with the tradeoffs involving time, space
 and the bias on efficiency.

 Capacities can go from one entry to several million entries.
 The help of intelligently-defined paths becomes more obvious
 when you deal with millions of entries.  Toy-like academic
 examples, of course, do not require any overhead in terms of
 structure.

>>

Passwords:

  10  SeeAll;
  <<...>>


Items:

  Manufacturer#,    x6 ;
  Distributor#,     x4 ;
  Product#,         x10;
  assembly,         x10;
  component,        x10;
  name,             x40;
  address1,         x40;
  address2,         x40;
  city,             x30;
  department,       x30;  <<Or "State" or "Province">>
  country,          x30;
  amount,           r4 ;
  production,       j2 ;
  supervision,      j2 ;
  responsibility,   j2 ;
  <<...>>
```

```
Sets:

Name:  MANUFACTURER, manual;

Entry:
  Manufacturer# (2),  <<2 paths, to "Manufactures"
                        and "Represents">>
  name,
  address1,
  address2,
  city,
  department,
  country;
  <<...>>
 Capacity: 2000;

Name:  PRODUCT, manual;

Entry:
  Product# (4),  <<4 paths, to "Manufactures", to "Sells",
                   to "Assembly" as a part of another
                   product and to "Assembly" as a product
                   which, itself, has components>>
  name;
  <<...>>
 Capacity: 80000;

Name:  DISTRIBUTOR, manual;

Entry:
  Distributor# (2),  <<2 paths, to "Represents" and "Sells">>
  name,
  address1,
  address2,
  city,
  department,
  country;
  <<...>>
 Capacity: 20000;

Name:  ASSEMBLY, detail;  <<Relates products which are, in turn,
                            parts of other products>>

Entry:
  assembly  (PRODUCT),  <<This search field allows us to find all
                          the products which we need to assemble
                          a given product>>
  component (PRODUCT),  <<This search field allows us to find all
                          the products which we can assemble with
                          a given product>>
  amount,
  Production,           <<Person in charge, for instance>>
  Supervision,          <<Person in charge, for instance>>
  Responsibility;       <<Person in charge, for instance>>
  <<...>>
 Capacity: 150000;

Name:  MANUFACTURES, detail;  <<Relates manufacturers
                                to products>> Entry:
  Manufacturer# (MANUFACTURER),
  Product#      (PRODUCT);
  <<...>>
 Capacity: 2000000;

Name:  REPRESENTS, detail;  <<Relates manufacturers
                              to distributors>> Entry:
  Manufacturer# (MANUFACTURER),
  Distributor#  (DISTRIBUTOR);
  <<...>>
 Capacity: 5000;

Name:  SELLS, detail;  <<Relates distributors to products>>

Entry:
  Distributor#  (DISTRIBUTOR),
  Product#      (PRODUCT);
  <<...>>
 Capacity: 5000;
 End.
```

Appendix B

IMAGE schema for the *Presidential* database mentioned in the *Practical Methodology* section.

```
Begin database CHIEF;

<<

NOTES:

 Your imagination and convenience should decide how many (and
 which) attributes to include at the "...".

 The paths are NOT necessary at all, but we include them as
 examples of performance boosters for relationships which
 seem to be "hot and heavy".  You can always take ALL paths
 away, or add OTHER paths at will.  IMAGE does not care: You
 are free to play with the tradeoffs involving time, space
 and the bias on efficiency.

 Capacities can go from one entry to several million entries.
 The help of intelligently-defined paths becomes more obvious
 when you deal with millions of entries.  Toy-like academic
 examples, of course, do not require any overhead in terms of
 structure.

>>

Passwords:

  35  GuessWho;
  <<...>>


Items:

  Country#,       i  ;
  President#,     i2 ;
  Name,           x50;
  Year,           i2 ;
  Year-In,        i2 ;
  Year-Out,       i2 ;
  party,          x20;
  <<...>>


Sets:

Name:  COUNTRY, manual;

Entry:
  Country# (1),  <<path to "Administration">>
  name;
  <<...>>
 Capacity: 300;

Name:  YEAR, manual;

Entry:
  Year (2);  <<2 paths, to "Administration">>
  <<...>>
 Capacity: 4000;  <<You might want to go back to the Etruscans!>>

Name:  PRESIDENT, manual;

Entry:
  President# (1),  <<path to "Administration">>
  name;
  <<...>>
 Capacity: 20000;

Name:  PARTY, manual;

Entry:
  Party (1);  <<path to "Administration">>
  <<...>>
 Capacity: 1000;

Name:  ADMINISTRATION, detail;  <<Relates countries to presidents
                                  to In and Out years>>

Entry:
  Year-In  (YEAR),  <<This search field allows us to find all
                      the administrations, anywhere in the world,
                      which began in a given year>>

  Year-Out (YEAR),  <<This search field allows us to find all the
                      administrations, anywhere in the world,
                      which ended in a given year.  We can treat
                      Year-out=0 as a current administration>>

  Country# (COUNTRY),

  President# (PRESIDENT),
  Party (Party); <<Who knows?  Presidents might have different
                 parties from one administration to the other.
                 Party is NOT an attribute of PRESIDENTS,
                 as it might appear.  Party IS an attribute of
                 ADMINISTRATIONS!>>

  <<...>>
 Capacity: 150000;

 End.
```