On Some Metrics for Databases or What is a Very Large Database?

Rob Gerritsen, Howard Morgan, and Michael Zisman

Introduction

"What is a very large database?" was a question heard often at the first International Conference on Very Large Data Bases(VLDBs). It is clear that the question is of interest to a large number of users and researchers in the database field[8]. After some study, we have concluded that there has been little written at all on the question of how one can measure databases. Many articles have appeared that analyze or measure the performance of a specific algorithm or data organization method, but none seem to have focused on intrinsic measures of the database itself. This paper proposes a set of measures that can be determined by examining the database itself, i.e. its structure and the particular set of data that comprise it at any point in time. We describe the rationale behind these measures, how they can be computed, and provide some examples and insights into how they might be used. Finally, we show one such use by defining the term "very large database."

Just as the chemists have static, structural measures for compounds (i.e. chemical formulae) and dynamic measures (e.g. concentrations or volumes), we also propose both static measures which take into account the data structures that can be stored,

---

Research supported in part by the Office of Naval Research under Contract N00014-75-C-0440 (NR049-272).

and dynamic measures which describe the actual amounts of data of each type that are in the database. These measures are then combined in various ways to obtain some interesting and useful results. Some of the measures are multidimensional, and others are unidimensional. There is a tradeoff between the increased precision of a multidimensional measure, and the difficulty that people have in determining the isoplanes for such measures which allow comparisons, and for building our own intuition about each component of these measures.

Throughout the paper, we will make use of terminology from both the network and relational views of databases. Each offers different insights into the measurable characteristics of databases.

It is also important to note that we will be concerned with measures which are intrinsic to the database. As most of us realize, there are methods for trading off the complexity or running time of programs against the storage of information in the database (as an extreme example, one could maintain sorted sets in the database, or sort for each retrieval request).

Also, we will mainly be measuring things at a "syntactic" rather than "semantic" level. That is, our measures will not make use of semantic information carried by links, pointers, or domains of relations. For example, if a field in some record is used as a hash key to some other record, building an implicit link, our measures would not recognize this link. Hence, some care will have to be taken to precisely define what we mean by a database for purposes of measuring them.

We will try to point out those instances where more semantic information can be used. Such information would be contained in a higher level schema (relational or entity-set, for example). Some of our future work will be directed to these questions of how to incorporate such semantic information, without defeating the purpose of our measures as intrinsic measures of databases.

What is a Database?

It is clearly impossible to define what is a very large database without first defining what is a database. A survey by the Society for Management Information Systems once came up with 500 published definitions for MIS[9], and we have no doubt that a similar survey in the database field would have equally disastrous results. One of the problems with getting general agreement on a definition of database is that there are so many levels at which people view databases. Each level (conceptual, logical, physical) leads to a different set of characteristics that are focused on in the definition. For example, consider the following plausible definitions of database:

1. A database is a collection of all of the data necessary to describe an organization at a point in time.
2. A database is all of the files used for an application.
3. A database is all of the files that are used at a computer installation.

All of these pose problems in the sense that we can think up examples where things we might not "naturally" call databases

fit, or vice versa. Many of these examples arise from the problem of defining the boundaries of the term "organization." Under definition 1, the database for the Decision Sciences Department is but a piece of the database for the University of Pennsylvania. Which of the two is to be called database?

What we are seeking is a definition that will be natural, yet have some operational value in terms of the metrics that we shall propose. In order to talk about something we can "weigh", we need a cohesive database. This leads us to use the notions of connectedness, in both structure and time, as the basis for our definition.

> DEFINITION: A <u>database</u> is a set of data whose structure can be described by a connected graph, and whose instances can be accessed simultaneously. (Implicit, program dependent links such as those referred to in the introduction should be treated as explicit links for purposes of forming the connected graph.)

By the above we mean to include mainly databases that can be stored on-line. We admit that this excludes some interesting collections of data that some would like to call databases. However, the main thrust of the field over the past few years has been to worry about how to handle complex interconnections among data instances. The reason for this is to provide rapid access to these instances, and this indicates on-line data.

Let us also note that backup file dumps are not normally included in our definition of database. Thus, if you have three generations of your database on tape, our set of metrics can be applied to any one of the generations, but not to all three taken

as a whole. Unless there are explicit structural links between the database generations, each generation is treated as a separate database. In other words, historical data can be considered to be part of the database only if it is explicitly contained, connected to other data (say the current values), and simultaneously available. We rule out copies maintained purely for back-up because we feel that these copies do not affect the inherent complexity of the database.

The strength of the connected graph that is representative of the database, and the ease (defined by some algorithm) with which it can become disconnected is probably related to the structure of the organization that is using the database [7]. We would expect the resulting pieces of the graph, once disconnected, to be representative of sub-databases for sub-organizations. We will be able to discuss this further once we have examined some static measures.

Static Measures of Database Size

Static measures are those which do not change as the database grows in size or number of occurrences of records, sets, or items. Our intent is to measure the inherent complexity of the schema or data structure being used. One would expect that the more complex the schema, the more choices there are to be made in navigating through the particular data structure, and it is this view that guides our choice of specific measures. We note here that we are addressing issues of logical data structure complexity and not the complexity of the conceptual data structure as seen by the user.

Consider a database structure (schema) to be represented by a directed graph where each node represents a record type and each arc represents a functional dependency or set relation with direction from the owner to the member record type. Some possible measures of data structure complexity are:

1. the number of record types (nodes)
2. the number of field (item) types at each node
3. the number of arcs (set types)
4. a measure compounding the complexity of the basic structures from which the schema is developed
5. graph density
6. connectivity (bridge analysis)

Items 1 and 3 address the diversity of the schema structure. As each of these measures increase, the complexity of the schema structure will increase, but less than linearly. An increase of the number of nodes relative to the number of arcs causes little net increase in complexity, but an increase of the number of arcs relative to the number of nodes increases the number of relationships or functional dependencies in the schema graph and hence increases its complexity. A fully connected graph could be an example of a complex schema graph. We note, however, that a fully connected graph does not represent the maximum number of relationships that can exist between nodes because any two nodes may have more than one arc connecting them representing more than one logical relationship.

Item 2 is similar to items 1 and 3 in that it addresses the potential amount of information that a user will need to successfully interact with the database. It is possible to view the schema graph which we have been discussing in more detail by exploding each node into a tree where each leaf of the tree corresponds to a data field type.

Item 4 specifies discrimination between the basic structures (hierarchies, confluencies, and recursive hierarchies) from which the schema is developed [5,6]. A confluency, or confluent hierarchy, is said to exist when a record type is a member in more than one set (i.e. it is owned by more than one record type). Confluencies add complexity to the schema graph because they create a potential for "cycles" in the schema graph (in testing for cycles we do not consider the direction of the arc since we can traverse it in either direction). Thus, confluencies make it possible to navigate from node A to node B via more than one path. If there are no confluencies in the schema, then there exists at most one path between any two nodes.

Just as a graph can be used to describe the schema, a graph can also be used to represent the data stored according to that schema (a data graph). If the schema graph is a tree, then the data graph will be a forest (a group of trees). Confluencies also create cycles in the data graph. We are in no way suggesting that confluencies should not be used because they are complex; we do wish to point out the potential schema complexity which can arise from their use. Normally this added complexity in the schema is offset by reduced execution time in the processing programs and reduced storage requirements.

In analyzing recursive hierarchies, we note immediately that it is impossible to detect recursive hierarchies by examining only the schema graph. A recursive hierarchy is implemented as two sets between the same owner and member records, or as two arcs connecting two nodes. However, in examining a schema graph and observing two arcs connecting two nodes we can only infer that there are two relationships between these two nodes. These two arcs can be the basis for a recursive hierarchy (e.g. bill of materials structure) or simply represent two separate relationships. To determine if a particular instance represents a recursive hierarchy requires a semantic interpretation of the data structure. The complexity of recursive hierarchies (if we wish to consider them complex) arises from the manner in which the program can elect to navigate through the database and not from the underlying data structure. Since we are only concerned here with the complexity forced on the program by the schema graph, we will associate no additional complexity with recursive hierarchies over and above that which we will associate with confluencies.

Items 5 and 6 are similar to item 4 in that they address the complexity of navigating through the database. This complexity arises when there exists more than one path between two nodes. If there is only one path between two nodes then there is no decision required about how to navigate from one of these nodes to the other. However, if there exists more than one path a decison making procedure is necessary when one wishes to navigate between these nodes. This component of complexity can be measured either by measuring the "density" of the graph (the

number of cycles and multiple arcs), or by analyzing the bridges in the graph.

A bridge is an arc which, when removed from a connected graph, will result in two disconnected graphs. For example, every arc in a tree is a bridge. Clearly, if there is more than one path between two nodes, then no arc on any of these paths is a bridge. As the number of non-bridges in the graph decreases, we expect the complexity of the graph to decrease. As an aside, we suggest that a useful database design technique might be to perform such a bridge analysis. By observing how an originally fully connected graph can be decomposed into subgraphs by removing bridges, the designer may get a better understanding of the functional dependencies in the resultant cluster of subgraphs. Such subgraphs may also reveal the structure of major user views (sub-schemas). The same sort of analysis can be performed with cutpoints. Cutpoints can be split into two nodes with a connecting arc. This arc is a bridge and subject to bridge analysis, as above.

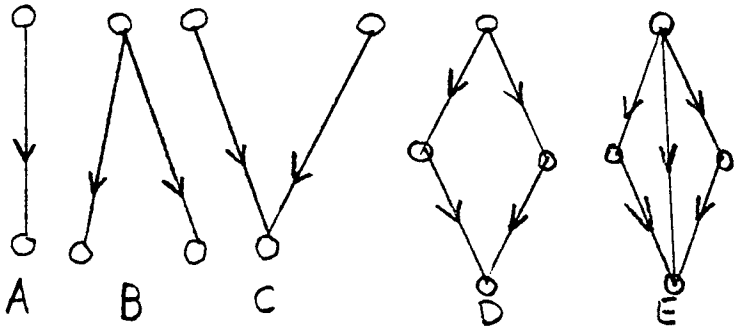
We suggest the following three dimensional scale for measuring schema complexity:

1. the log base 2 of the product of the number of nodes (record types) and the number of arcs (number of set types)

2. The sum of the log base 2 of the number of arcs directed into each node. That is, for each node, count the number of arcs directed into that node, take the log, and sum over all nodes.

3. The average number of field types (domains) per record (tuple).

Before trying to rationalize these measures, consider the following examples:



	nodes	arcs	complexity
A	2	1	1,0,5
B	3	2	2+,0,5
C	3	2	2+,1,5
D	4	4	4,1,5
E	4	5	4+,1+,5

The first measure is representative of graph size and density. As the number of nodes increase and the number of relationships increase, this measure will increase. In one sense, this is a measure of how much information is needed to navigate through the database.

The second measure represents the complexity of the primitive data structures from which the schema graph is formed. A tree structure (hierarchy) is a baseline of zero complexity in this dimension. The number of confluencies and degrees of confluency will determine the magnitude of this measure. In both cases, logarithms are used only to reduce the explosive nature of the measures.

The third measure relates to the complexity required to maintain the database. As the number of data items in the database increases, it is reasonable to expect an increase in the processing requirements to maintain the database in a timely and accurate manner.

One useful measure which can be derived from these complexity measures is what we shall call the schema complexity ratio. In designing a database structure, one must specify the functional dependencies which are to be incorporated into the schema. It is possible to derive a minimum schema structure automatically by examining the queries which are to be answered with the database [6,1]. By deriving the functional dependencies from the questions to be asked of the database, we can construct a schema with the minimum number of sets which will answer all of the queries and incorporate no data redundancy into the database.

Since the number of sets is minimized, this schema will have minimum complexity in the first two components of our measure. After this minimum schema is derived, the database designer will normally add set relationships to simplify sub-schemas and to increase processing efficiency. In so doing, the database designer is making trade-offs between schema complexity and processing efficiency. The first two components of our complexity measure will be increased by adding set types.

A schema complexity ratio can be calculated as follows:

1. Calculate the complexity measures for the minimum schema and the actual schema to be used.

2. Treat the first two components of these two measures as points in a two-space and calculate vector lengths.

3. Compute the ratio of the vector length of the actual schema to the vector length of the minimum schema. This will be the schema complexity ratio.

This ratio will measure the extent to which complexity has been added to the schema to increase the efficiency of the processing programs. With this ratio we can begin to quantify the trade-off between schema complexity and program efficiency.

While we have implicitly performed this analysis on a network database, we believe that it is also useful for measuring the complexity of a relational database. The difficulty here comes from the fact that in a network database all of the relationships are stated a priori and in a relational system they are constructed dynamically. However, for a given application or end-user view (equivalent in scope to the network sub-schema) we can state all of the required relations and hence the operations which must be performed on the base relations to create the desired relations. From this we can construct the data structure graph and analyze it in the manner described in this paper. In so doing, we are analyzing the complexity of the underlying data structure and not the user's view of the data. It may be possible to extend this type of analysis to measure the complexity of the user view as well.

Dynamic Measures of Database Size

The above measures deal with a database without any data actually loaded into it. We now wish to consider some "dynamic" measures, which take into account which parts of the schema actually have a lot of instances, and which do not. The obvious dynamic measure of a database is its physical size, usually measured in bits, reels, packs or tons [8]. Although independent measurement in each dimension is possible, we will introduce a single measure that has both dynamic and static components.

Some easily obtainable components of dynamic measures are record sizes and the number of record occurrences. If S is a size function in bits, and F is a frequency function, then the data size in bits of a database containing n record types, R1, R2, ..., Rn, is

$$\sum_{i=1}^{n} S(Ri) * F(Ri) \qquad (1)$$

Instead of focusing on records, as in (1), a similar result can be obtained based on item types.

Other important dynamic measures are the volatility of a database, e.g., the rapidity with which values change, records are removed, and new records are inserted; and the access rates, e.g., the relative frequencies of access of various components of the database.

A third measure, derived from these two is simply their ratio:

access rate / volatility .

This measure is very important to the Data base Administrator.

In general, higher values in this ratio imply the need for more structure. This is so because structure tends to improve retrieval efficiency at the expense of maintenance.

A Single Measure

A comparison of databases based solely on dynamic measures can be misleading because it will ignore the structure and the complexity of the database, and the interactions between physical size, structure, and complexity.

Since schema complexity tends to reduce the physical size of the database by removing data redundancy, it seemed to us that one good way of comparing two databases of varying size and complexity was to compare the size of the storage that would be required for each if all complexity were removed. This permits the derivation of a one dimensional measure incorporating traditional dynamic measures, a schema complexity measure and a measure of their interaction. An interesting side benefit of the approach used is that it gets us around the problem of measuring complexity itself, although complexity is directly incorporated in the measure! Since a database with no structure can said to be flat, we call our measurement the flat size of a database.

In terms of the relational model [3,4], a flattened database is a single relation in first normal form. Since first normal form is the only normal form that incorporates all functional dependencies within a single relation, data must go through first normal form representation to restructure a database if a change in the functional dependencies is part of the reason for restructuring. Because first normal form is thus essential to

restructuring, flat size is also useful as a predictor of the time that is required for restructuring.

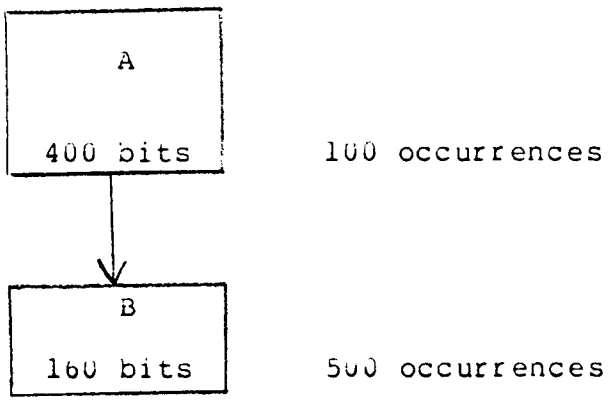


Figure 1. Simple hierarchical database.

Figure 1 illustrates a simple hierarchy. The data size of this database given by (1) is:

400 x 100 + 160 x 500 = 120,000 bits.

Any discussion on measurement, especially a pragmatic one, cannot avoid the effects of implementation. There are of course several ways to implement a structure such as is illustrated in Figure 1. One way is to pull the keys of A into appropriate occurrences of B and store the database in third normal form. Two other methods were proposed by the Data Base Task Group (DBTG) [2]: as a chain or as a pointer array. Note that implementation in third normal form implies searching through the relation B to find those tuples related to a specific tuple in A. In a large database environment searching is generally to be avoided or minimized, so we will focus on the implementations that use pointers. As an aside we note that implementors of

relational systems are now turning to pointer implementations very similar to those proposed by the DBTG.

Pointer array implementations require two pointers for each member record occurrence (one pointer to the member record occurrence and one pointer in the member record occurrence pointing to the owner). Chain implementations require one pointer for each member record occurrence and one pointer for each owner record occurrence. The DBTG proposed that chains can optionally be doubly linked, doubling the number of pointers, and that pointers to owners can also be optionally included, adding one pointer for each member record occurrence. In general, since a pointer array implementation reduces search the most at the lowest cost in pointers, we will assume such an implementation in our examples.

The implementation overhead for the structure in Figure 1 is therefore 1000 pointers. Since most implementations use one computer word for a pointer, a pointer uses the equivalent space of 32 (usually) bits. Assuming 32, the implementation overhead is 32000 bits.

Define <u>implementation penalty</u> as:

(data size + implementation overhead) / data size .

Then the implementation penalty for the database of Figure 1 is 1.27 .

Flattening this structure simply requires replication of 400 bits from the owning record in each member record. Hence the flat size is given by:

$$F(B) * [S(A) + S(B)] \qquad (2)$$

or 280,000 bits in our example. Note that the flat size will be slightly larger (up to 319,600 bits) if there are any empty hierarchies (sets without members, in DBTG terminology).

We define efficiency as:

data size / flat size.

and implemented efficiency as:

efficiency * implementation penalty.

Hence, the efficiency of the database in Figure 1 is .428 and it has an implemented efficiency of .544 .

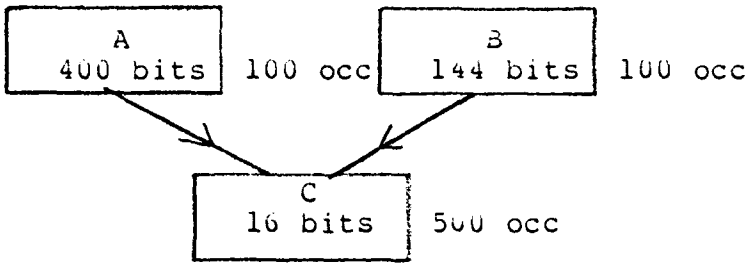


Figure 2. Database consisting of a Confluent Hierarchy.

Figure 2 illustrates a database with the same flat size as the database in Figure 1. The data size of this database is 62,400 bits, the implementation overhead is 64,000 bits; hence the efficiency is .223 , the implementation penalty is 2.03 , and the implemented efficiency is .453 . The implementation penalty associated with this example serves to illustrate why implementational considerations cannot be ignored!

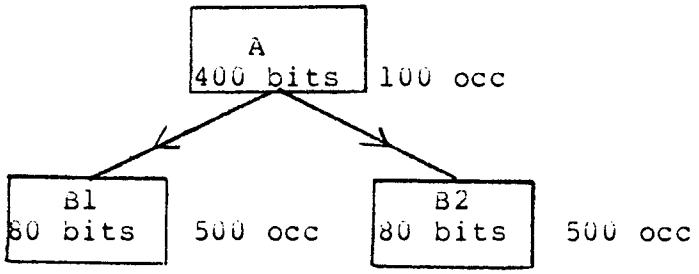


Figure 3. A tree structure.

Figure 3 illustrates a structure that has the same data size as the database of Figure 1. The implementation overhead of this database is 64,000 bits. Flattening it requires a natural join (Cartesian product) of all occurrences of B1 and B2 owned by the same occurrence of A. We have considered the question of whether such a join is required, or if it would be sufficient to replicate A in occurrences of B1 and B2 independently.

A strong argument for the join is that it permits the derivation of a single relation, rather than the multiple relations that would result if the join was not used. Conversely, the semantics of a particular data structure often indicate that most of the tuples resulting from a join are meaningless. We have elected to use the join on a structure such as the one in Figure 3 because it seems to be more complete (one can always discard meaningless tuples), realizing that we do not yet have a satisfactory conclusion. Computation of the number of tuples in a relation following a join is simply the product of the number of tuples in each input relation. For the database in Figure 3, we need to sum up 100 such multiplications. This

requires that we know the exact number of occurrences of B1 and B2 associated with each of the 100 occurrences of A.

Each of the tuples generated from the database in Figure 3 contains 560 bits. If we assume that there are no empty sets, then the number of tuples generated will be at least 900, but no more than 160,900. The minimum 900 tuples will result if 98 occurrences of A own exactly 1 occurrence of B1 and 1 occurrence of B2; the 99th occurrence of A owns 1 occurrence of B1 and 401 occurrences of B2; and the 100th occurrence of A owns 401 occurrences of B1 and 1 occurrence of B2. The maximum 160,900 tuples result if 99 occurrences of A are paired with 99 occurrences of B1 and B2, and the 100th occurrence owns the remaining 401 occurrences of B1 and B2.

The expected number of tuples can be statistically calculated as the value of a vector product where the elements of each vector are positive random variables constrained to sum to a constant. Assuming independence, the elements of the two vectors have expected values of F(B1) / F(A) and F(B2) / F(A) respectively, or 5 in this case. Hence, the expected number of tuples is 2500, and the expected flat size is 1,400,000 bits. The implementation penalty is 1.53, the efficiency is .066, and the implemented efficiency is .131 .

A fourth interesting basic structure is the recursive structure shown in Figure 4 (usually known as the bill of material (BOM) structure). This database has a data size of 120,000 bits and a flat size of 480,000. The implementation

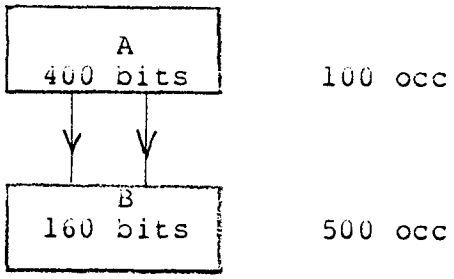


Figure 4. A database with a recursive structure.

penalty is 1.53, the efficiency is .25, and the implemented efficiency is .362 . Although the structure in Figure 4 is used to represent (possibly) recursive structures, flattening this structure does not require recursion. We note that this structure can also have a non-recursive usage, for example if A is a PROFESSOR record and B is a STUDENT record, and the two sets represent ADVISOR and SUPERVISOR relationships.

Although we have chosen to flatten databases into single relations, we recognize that with schema graphs that include trees such as in Figure 3, there are good reasons for permitting multiple relations in the flat database. If a join is not taken (to end up with a single relation), then the number of resulting relations is exactly equal to the number of record types that are not owners of any sets, and the number of tuples of each such relation is the number of occurrences of the corresponding record type (with no members). Under this second method of flattening, the efficiency of the Figure 3 database is .25 and the implemented efficiency is .362 .

Applications

Metrics in general are useful for comparisons among several similar items, and for prediction of characteristics of similar items. In our case, this means being able to answer the question "what is a very large database?", which is done in the next

section, and to predict the performance of various algorithms on databases of varying size and complexity.

It was originally our intention to create a set of "unit" databases, which could be used to benchmark algorithms. The performance of these algorithms on these "unit" databases could then be scaled by our size measures to predict performance on the real databases. Since our measures make no use of database semantics, however, this proves quite difficult.

One example where this does work is the following: Consider an algorithm which is to print one detail report for each leaf record in a database, containing all information which is linked to that record. In essence, this involves flattening the database and printing out one detail report for each tuple in the resulting first normal form relation.

The time this type of algorithm takes is proportional to the vector measures of static and dynamic complexity, and to the flattened size of the database. Hence it could be scaled by taking a vector which consists of the derivatives of time to the corresponding measure, e.g., dt/d(number of sets), dt/d(number of occurrences of each set), etc.

We have not yet had the time to explore the use of these "unit" databases, but plan to do so in the near future. However, we have found a use for some of our measures already.

What is a Very Large Database?

We take a very pragmatic view in attempting to separate very large databases from the merely large ones. That is that one of

the most complex operations that one performs on a database is restructuring of the database. For some types of restructuring operations, the entire database must be dumped and then reloaded. It is our feeling that the time taken for this operation can be used as the basis for a definition.

We have noted that a flatttened database provides, in a first normal form relation, all of the data that might be needed to reload the database into a new structure. Let us assume, then, that for purposes of deciding whether or not a database is very large, we shall use this flat size.

Next we assume that such reorganizations must take place when they will not disrupt the functioning of the organization for which they are maintained. In most organizations, this means that a maximum down time for the database is 5 PM Friday to 9 AM Monday, or a total of 64 hours. Normally, the act of restructuring takes place in two steps: a flat dump of the data from its old structure followed by reformatting the flattened database into its new structure. This means that the flat file must be processed twice.

Hence, any database whose flat size would take more than 32 hours to output is "very large." With today's technology, we may wish to assume a transfer rate of 1 megabyte/second. At this rate a total of 1.152 x 10**11 bytes, or just about 10**12 bits of exploded size, would constitute a very large database.

This figure of 10**12 bits seems to agree with other intuitive definitions of what is a very large database, relative to smaller databases. Clearly, as transfer rates are improved,

the size break will go up. However, for the near future, it is unlikely that we will see more than two or three orders of magnitude growth in transfer rate.

An interesting question to ask is what is the smallest database that is a VLDB. The answer will of course vary according to the structure that is used. Assuming a simple hierarchy of records A and B, with A owning, and functions F and S as defined above, then the flat size is given by:

$$F(B) * [S(A) + S(B)]$$

and the data size, X, is:

$$X = F(A) * S(A) + F(B) * S(B)$$

The problem can be stated as:

MIN X

subject to

$$F(B) * [S(A) + S(B)] \geq 10 ** 12$$

$$S(A) + S(B) \geq LOG2\ F(B)$$

$$S(A) \geq LOG2\ F(A)$$

$$S(B) \geq LOG2\ F(A) - LOG2\ F(B)$$

$S(A), S(B) > 0$ and integer.

The three constraints (we state all three, although the last two are sufficient) based on the logs of the frequencies are derived from the requirement that tuples must be unique. For example, since F(B) tuples will occur in the flat database, each tuple must contain at least LOG2 F(B) bits.

The optimal solution (found by an heuristic search) appears to be:

F(A) = 1

S(A) = 1,000,005

F(B) = 999,976

S(B) = 20

X = 20,999,525

Although this is a rather strange database, with one very large record and many tiny records, it is interesting to note that a hypothetical VLDB (of minimal structure) exists, containing only 2.099953 x 10 ** 7 bits.

Summary

We have attempted to present a set of metrics that can be used to determine the static and dynamic size and complexity of a database, independent of the processing that is performed on the database. These measures seem to fit with our intuition of what constitutes a complex vs. a simple database, and how size relates to complexity.

We hope in a future paper to present some examples of the use of these metrics to aid in predicting the performance of various types of algorithms for processing databases. It is our hope that this first step in trying to measure inherent characteristics of databases will lead to some fruitful discussion in the database community on the general measurement question.

References

1. Clough, L., W. D. Haseman, and Y. H. So, "Methodology of Optimal Data Structures," Draft, Carnegie-Mellon University, December 1975.

2. CODASYL CODASYL Data Base Task Group April 71 Report, available from ACM, New York City.

3. Codd, E. F. "A Relational Model of Data for Large Shared Data Banks," CACM 13, 6, June 1970, pp377-387.

4. Codd, E. F. "Further Normalization of the Database Relational Model," in Rustin, R. (Ed), Data Base Systems, Prenctice Hall, 1972.

5. Gerritsen, Rob "Understanding Data Structures," PhD Thesis, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1975, available from NTIS, Springfield, VA 22151, accession number ADA008937.

6. Gerritsen, Rob "A Preliminary System for the Design of DBTG Data Structures," Comm. ACM 18, 10, October 1975.

7. Hackathorn, R. D. "Analyzing Transaction Activity to a Large Database: An Empicirical Study," Proc. International Conference on Very Large Data Bases, Framingham, Massachusetts, September 1975, pp502-504.

8. Morgan, Howard L. "The Very Large Very Large Database Conference," Comm. ACM November 1975.

9. Society for Management Information Systems "Definitions of MIS," Special Report No. 1, 1970.

Department of Decision Sciences
The Wharton School
University of Pennsylvania
Philadelphia, PA 19174