THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 52, No. 10, December, 1973
Printed in U.S.A.

Information Management System:

MASTER LINKS—A Hierarchical Data System

By T. A. GIBSON and P. F. STOCKHAUSEN

(Manuscript received October 5, 1972)

MASTER LINKS is a software system used to build, administer, and access hierarchical data bases. It is designed to operate in a time-sharing environment, and, in particular, it allows multiple concurrent updates and retrievals on the same data base.

A BUILD module is used to specify the hierarchical configuration of a data base and an initial "storage mapping" of the elements of the hierarchy into a particular file layout. A set of administrative routines is provided for altering the mapping and other such maintenance purposes. The access routines have three levels of interface, from primitive and flexible to sophisticated and functional. The interfaces are all defined in terms of the hierarchical structure and independent of the storage mapping. Thus, an alteration of the storage mapping for a data base does not require changing any programs that access data using these interfaces.

The lowest-level interface enables the calling program to add to the data base, update a value, or retrieve a value, in terms of a hierarchy position. The second-level interface facilitates traversal of a hierarchy by enabling the calling program to specify portions of the hierarchy over which a process is to operate. Such a specification, called an "access tree," consists of data which can be generated at execution time by the calling routine. As in the first level, data are transferred one at a time. The third-level interface is a function evaluation mechanism which computes values from data base values and other computed values according to function definitions passed to it at execution time. Like an access tree, a function definition is itself data which can be constructed at execution time by the client process.

I. MASTER LINKS OBJECTIVES

The Master Links data system is a collection of software that accesses and manipulates data stored in a hierarchical structure on a computer's secondary storage devices. It services requests from "client" programs to store and retrieve data, and to create and release space in the data structure.

The Master Links project was designed with the following goals:

(*i*) Provide a basic "low level" set of access mechanisms to retrieve and store data items, and to create and delete branches of the hierarchy. Client programs using these mechanisms work entirely in terms of the hierarchical structure.
(*ii*) Provide "high level" access mechanisms that simplify the programming task for complex retrieval requests.
(*iii*) Support many concurrent users on a data base, doing both retrievals and updates.
(*iv*) Operate well in a time-sharing environment.
(*v*) Enhance portability of the system by basing its design on machine-independent concepts.

Other goals are presented in the text of this paper.

This report begins with a definition of the elements of hierarchical data structures, and a description of the basic access mechanisms, in Section II. Section III examines the requirements of typical client processes. Then high-level access mechanisms are described in Sections IV and V. Thus, these four sections describe the system as viewed by its users. Section VI delves into the system design and shows how the structures are arranged to provide these capabilities in portable form with high performance. The final section discusses the experience acquired with current implementations, and presents an outline of current and future developments of Master Links.

II. ELEMENTS OF HIERARCHICAL DATA STRUCTURE

The elements of a hierarchical data structure are entities, groups, and fields. Groups and fields are the permanent elements of a data base. They are established by a process called "building" the data base. Entities are the dynamic elements. They are added and deleted at any time by client programs using the basic access mechanisms of Master Links. Client programs also use the basic access mechanisms to transmit data for a field of an existing entity.

2.1 *Entities, Groups, and Fields*

A *field* is a set of data all identified by the same *field name.* There are several types of fields: numerical, character, logical, and date.

An *entity* is an element which holds one value for each of a given list of fields. We will draw an entity as a rectangle, with the field names to one side and the values inside, thus:

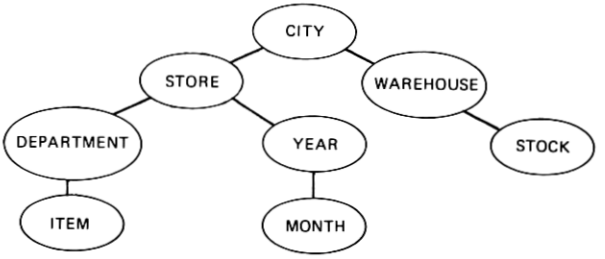
STORE NAME	PLAZA
EARNINGS	10325

A *group* is a set of entities with the same fields. A group has a name which indicates the nature of its entities. The name of a group will be written in an ellipse:

STORE

STORE NAME	PLAZA	MAIN ST	RT 46	PLAZA
EARNINGS	10325	69238	21420	96823

A data base is composed of a set of groups which are hierarchically related. One group is the top group. All others descend in a tree fashion:



This is a STORE and WAREHOUSE data base. It is subdivided by CITY at the top. Each city of the chain of stores is represented by one entity in the CITY group. There are several stores per city, several departments per store, and several items per department. In addition, certain data are kept on an annual and a monthly basis for each store. Each city also has zero, one, or more warehouses, and there are several items of STOCK per warehouse.

Although called a tree, the structure is always drawn "upside down." This is not in fact unusual. Corporation organization charts are frequently drawn this way, as are part lists, inventory lists, etc. It

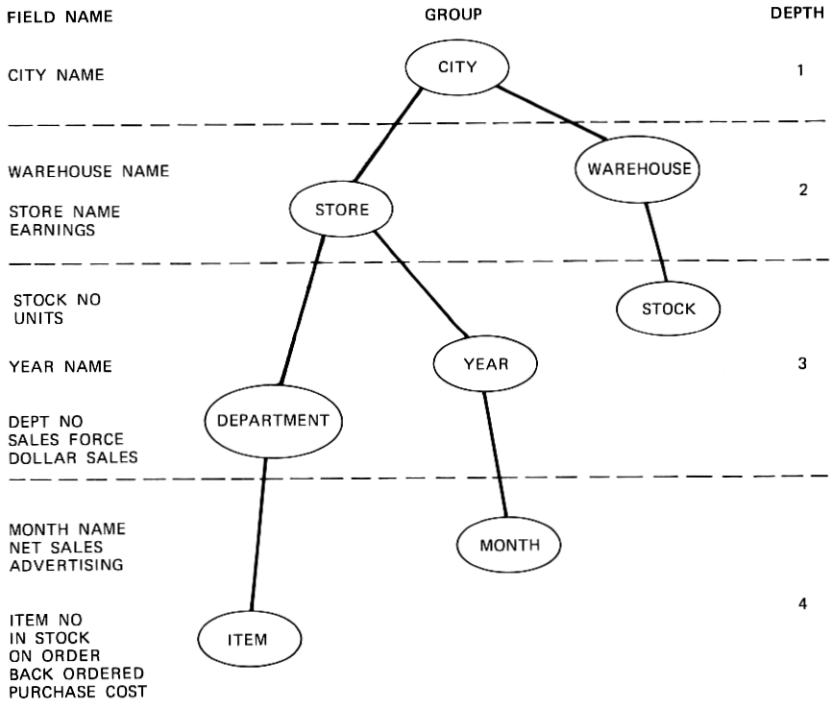
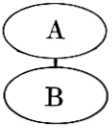


Fig. 1—A group tree with fields.

places the major components at the top and the detailed ones lower down.

Figure 1 shows this data base with fields assigned to each group. Figure 2 shows a blowup of the entities. The ellipsis (· · ·) in a group indicates several entities not shown.

The parent of a group is the group immediately above it. The top group has no parent. All other groups can have only one parent. The parent of an entity is the entity immediately above it. Entities in the top group have no parent and all others have one parent. The parental relation of entities must parallel those of the groups. Thus if group B has group A for its parent:



then all entities of B must have their parent entities in A.

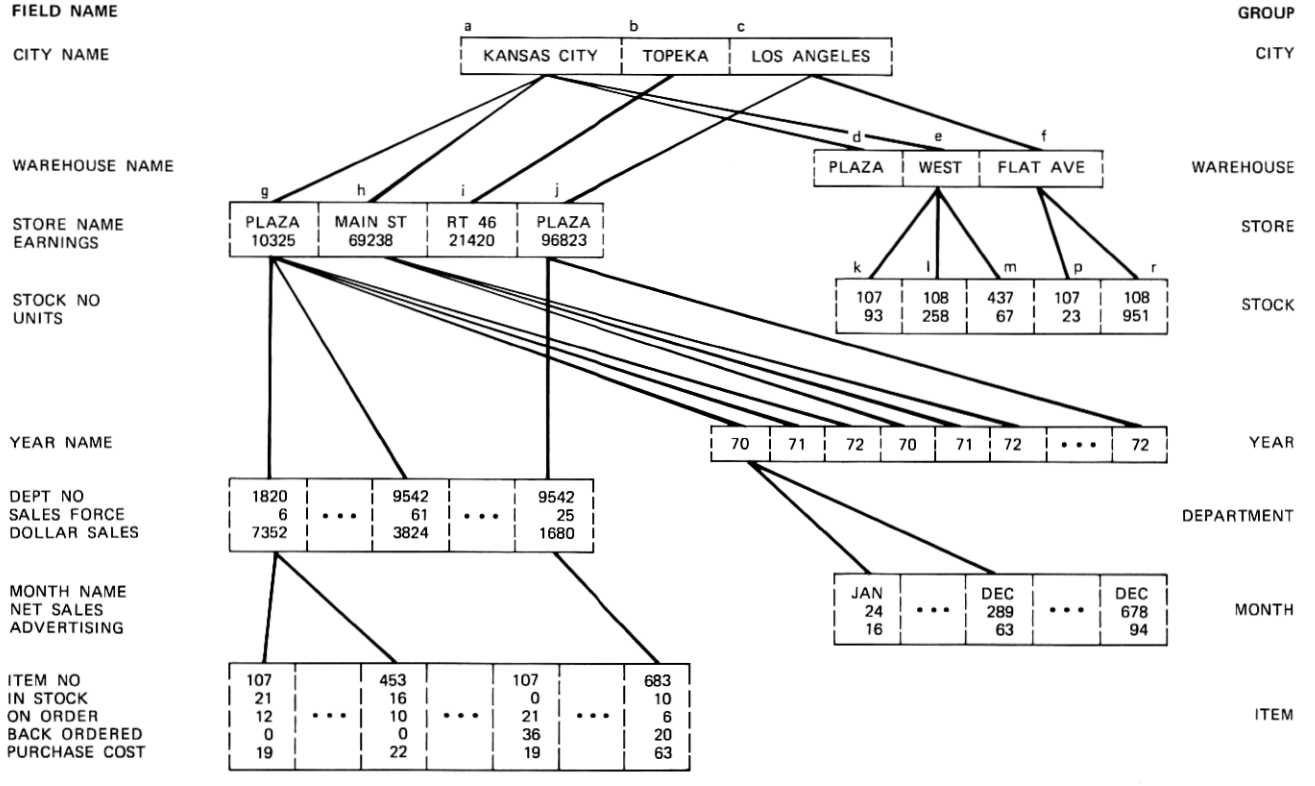


Fig. 2—A blowup of the entities.

In Fig. 2 the entities are labeled a, b, $\cdots$, r. These labels are not a part of the data base, but are used only as references in this paper.

We have made use of the terms "parent of an entity" and "parent of a group." This suggests the use of other genealogical terms. The kth ancestor of an entity is the entity k steps above it. (Hence the first ancestor is the parent.) The offspring of an entity are all the entities immediately (one step) below it. The descendants are all the entities below it.

For each entity, all its offspring in one group form *a family*. Entities a, b, and c are a family; d and e another family; g and h another family. Notice that entity a has two families under it, one in STORE and one in WAREHOUSE. If two entities are in the same family, such as d and e, they are *siblings* to another. If two entities have the same parent, but are in different families, such as d and g, they are *step-siblings*.

2.2 *Building the Data Base and Entity Dynamics*

A particular data base is established by defining the group tree and the fields of each group. This process is called *building* the data base. The language for describing the data base is called the *build language*. Using this language a data base designer describes the permanent attributes of his data base and submits the description to a utility program called BUILD. After BUILD has processed the description, the data base has no entities, and no data, but only a "skeleton" structure.

Entities are the dynamic components of a data base. They may be added or deleted online, even while other users are working on the data base. Thus the actual data base grows and acquires data, but always in accordance with the structure defined by BUILD.

2.3 *Basic Access Mechanisms*

There are five basic operations which programs can perform on the data structure:

(*i*) Select a top entity or an entity whose parent has been previously selected.
(*ii*) Add a new offspring to a selected entity or add a new entity to the top group.
(*iii*) Delete a selected entity.
(*iv*) Select a field.

(*v*) Transmit data to or from a selected entity for the selected field.

These five basic operations make possible any manipulation of the data structure except modification of the permanent attributes of the data base established by BUILD.

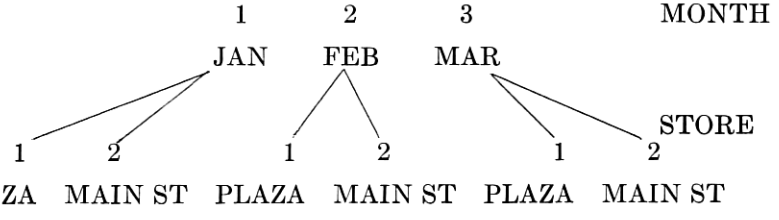
2.4 *Identifications*

A user (or a program) accessing the data base must be able to uniquely identify each element. Users identify elements by *names*, such as 'KANSAS CITY,' 'WAREHOUSE,' or 'EARNINGS.' Names are also called *external identifiers*, because they are used (by people) external to the software. The Master Links software uses *internal identifiers*, which are integers such as group 2, entity 7, field 13. The term *identifier* refers to both internal and external identifiers.

Fields and groups are given unique identifiers. Figure 2 shows group and field names. These names are selected at the time the data base is built and then do not change. Their internal identifiers are positive integers assigned by BUILD.

The identification of entities must be done somewhat differently, since they are not established by BUILD. The internal identifier of an entity is a positive integer called the *entity index*. The first entity of a family has index 1, the second has index 2, etc. Thus, the internal identifier is unique only within a family.

This method of identifying entities allows implicit associations to be established among the entities of a group. The most common use of this is to assign the same index to all the entities which have some attribute in common. In this case the external identifier names that attribute. For example, a data base with just MONTH and STORE groups is shown with internal and external identifiers.



All store entities with index 1 have the common attribute of storing data for the PLAZA store. One can request processing of all data for the PLAZA store, and this will cause all STORE entities with

index 1 to be processed. To uniquely identify a single entity in the store group requires specifying both a month and a store identifier.

Another implicit association possible is the ordering of entities. Again using months as an example, JAN through DEC can be assigned indexes 1 through 12 respectively.

III. THE CLIENT PROCESS

Processes that access data bases have a strong tendency to access either many fields from a few entities, or few fields from many entities. An example of the first type is

ENTER NEW SHIPMENT DATA FOR WAREHOUSE_____.

Values for many fields are to be put into one warehouse entity. For this type of request the basic access mechanisms are quite convenient. An example of the second type of request is

FOR ALL STORES IN CITIES ____, ____, AND ____, PRINT STORE NAME, AND 1972 NET SALES PER STORE DIVIDED BY EARNINGS.

Only a few fields (STORE NAME, NET SALES, and EARNINGS) are required, but a large number of specific city, store, year, and month entities must be accessed to fulfill this request. Further, the values of NET SALES and EARNINGS that are retrieved must be functionally combined into the values of "1972 NET SALES per store divided by EARNINGS." It is possible to do these tasks by using the basic access mechanisms, but the programming is tedious and lengthy. Master Links provides a set of higher-level access mechanisms that makes programming of the above PRINT request as simple and straightforward as this:

(*i*) Declare which entities are to be processed.
(*ii*) Step to each of these entities in turn, and retrieve and print a value for the requested function.

The entities to be processed are declared with an *access tree*. The access tree provides directions to the *generator* which steps to each of the entities in turn. Finally, the retrieval is performed by the *function evaluator* which does all the work of evaluating functions of data stored in a data base. These tools for client programmers are described in the next two sections.

IV. ACCESS TREES AND THE GENERATOR

An access tree describes a subtree of the entities of the data base. Thus any entity on the access tree has all its ancestors on the access tree. It can also be visualized as a "pruned" entity tree: when an entity is removed, so are all its descendants. Several concepts underlie the mechanism for building an access tree:

(*i*) The generated group
(*ii*) The refined inclusion of an entity
(*iii*) The refined set of entities
(*iv*) Independently refined sets
(*v*) Whole-family inclusion.

These are described in turn. The data base of Fig. 2 is used for all examples.

4.1 *The Generated Group*

Some groups of the data base will contain data needed by the process, and some will not. Those that contain needed data, and all their ancestors, are the generated groups. The rest of the groups have no entities on the access tree and therefore will not be generated.

The client process may specify what groups have needed data. It therefore specifies by implication the generated groups and the groups to be pruned from the access tree. The generated groups all have entities on the access tree. They are there either by refined inclusion or by whole inclusion.

4.2 *Refined Inclusion*

Refined inclusion means an entity has been put on the access tree by explicitly giving its group identity and entity identity within its family. In Fig. 3, KANSAS CITY has been explicitly named to the access tree, and therefore is a case of refined inclusion. In writing programs, the internal identities are used: the group number and the entity index within its family. In our examples in this section, we will use external identities, as has been done in Fig. 1. It is confusing to wade through a lot of numerical codes in examples when trying to learn about concepts.

The year entity whose name is 70 is not unique. There are several such entities, one for each STORE entity. They have the same external identity, 70, the same internal identity, index 1, and therefore

REFINEMENTS

GROUPS	REFINE LIST 1	REFINE LIST 2	REFINE LIST 3	REFINE LIST 4
CITY	KANSAS CITY	KANSAS CITY	TOPEKA	KANSAS CITY
STORE	PLAZA	MAIN ST	RT 46	
WAREHOUSE				WEST

INDEPENDENT REFINEMENTS

GROUPS	REFINE LIST 1	REFINE LIST 2
YEAR	71	72
MONTH	DEC	JAN

GENERATED GROUPS

CITY, STORE, YEAR, MONTH, WAREHOUSE, STOCK

RESULTING ACCESS TREE

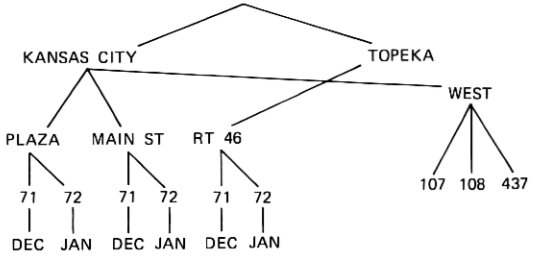


Fig. 3—Building an access tree.

have an association from family to family by identity. Wherever this condition exists, a single refinement can describe many entities in the data base. This is called *multiple refinement.* A refinement to

GROUP	*REFINE LIST*
YEAR	70

denotes every 70 entity of Fig. 2.

Refinements can depend on specific ancestors. This happens when a refine list has two or more entries. Thus:

GROUP	*REFINE LIST*
CITY	KANSAS CITY
STORE	PLAZA

identifies the PLAZA store only in KANSAS CITY, not the one in LOS ANGELES. PLAZA is called a *dependent refinement.*

A refinement can be both multiple and dependent, hence is called a *multiple-dependent refinement.* An example is

GROUP	*REFINE LIST*
YEAR	70
MONTH	JAN

which specifies a set of 70 entities, and the JAN entities under those 70 entities.

A refinement is not restricted to immediately adjacent levels of the data base. The following refinement is acceptable:

GROUP	*REFINE LIST*
CITY	KANSAS CITY
MONTH	JAN

The groups of a refine list must proceed down the data base from ancestors to descendant. However, groups may be skipped in the list.

4.3 *The Refine Set*

A *refine set* is a set of refine lists on particular groups. The groups of the refine set may be any groups, but the first group must be an ancestor of all the others. Figure 3 shows a refine set on the groups CITY, STORE, and WAREHOUSE, and another refine set on YEAR and MONTH.

A group can only be involved in one refine set. Every refine list of a set must start with an entity from the set's first group. Hence, to be a legal refine list, it must proceed to give entities ancestor to descendant down one path of the group tree, and from groups in the refine set.

A refine set allows a multiplicity of dependent refinements. The English clause

> FOR PLAZA AND MAIN ST STORES AND WEST WAREHOUSE IN KANSAS CITY AND RT 46 STORE IN TOPEKA

is easily represented as a refine set. In fact, Fig. 3 gives the refine lists to do this.

4.4 *Independent Refine Sets*

Refine sets alone cannot be used to represent independent refinements. The clause

FOR CITIES ——, ——, ——, ——, and ——, IN THE YEARS ——, ——, ——, ——, and ——

represents five cities, and five years in each city. The list of cities is a refine set on one group, CITY. The list of years is a refine set on another group, YEARS. What the entire clause specifies is the independent combination of these two refine sets. This leads to the concept of independent refine sets.

Refine sets are independent if their groups are mutually exclusive, i.e., any group can be in only one of the refine sets. Figure 3 shows a pair of independent refine sets and the resulting members of the access tree. Any number of independent refinements can be put on an access tree.

4.5 *Whole Inclusion*

Figure 3 has all stock items of the WEST warehouse on the access tree. Any generated group not in a refine set is included on the access tree on a *whole inclusion* basis. This means that whole families of the group are either included or excluded from the access tree, depending on whether their parent is included or excluded respectively.

Using Fig. 2, other examples of whole inclusion are:

(*i*) CITY group refined to KANSAS CITY, STORE group whole, all other groups pruned from the access tree. This puts entities a, g, and h on the access tree.

(*ii*) CITY and STORE groups whole. All other groups pruned. This puts all cities and all stores on the access tree.

(*iii*) CITY refined to KANSAS CITY and TOPEKA. YEAR independently refined to 71 and 72. STORE and MONTH whole. All other groups pruned. This puts entities a, b, g, h, and i; all 71 and 72 entities under g, h, and i; and all month entities under those year entities onto the access tree.

4.6 *Generating Entities on the Access Tree*

The generator accepts an access tree as an input. It generates only entities on the access tree. For brevity in this section we will say "offspring," "sibling," etc., but always mean "offspring on the access tree," "sibling on the access tree," etc.

On each call the generator takes one of the following actions:

(A) Takes a step to the "next" entity of the access tree, opening that entity for data accesses. The entity reached is said to be generated. The client program is informed of the group of the entity and the entity's identity among its siblings.
(B) Notifies the client process that the previous entity generated was the last of a family on the access tree. A new entity is not generated on this call. This action gives the client process an opportunity to perform summary processing on families.
(C) Notifies the client process that the previous entity generated was the last on the access tree. This action gives the client process an opportunity to perform final summary processing, and to exit from the processing loop.

On the first call, the entity generated is the leftmost entity of the top group. This becomes the current entity. On subsequent calls, the generator tries to step from the current entity to another entity in the following order:

1. Leftmost offspring of the current entity.
2. Sibling of the current entity.
3. Step-sibling of the current entity.

The first of these that succeeds becomes the new current entity. If all fail, the current entity is redefined as the parent of the current entity, and the above process resumed at step 2. The effect is to continue the list with

4. Sibling of the first ancestor of the current entity.
5. Step-sibling of the first ancestor of the current entity.
6. Sibling of the second ancestor of the current entity.

⋮

etc.

This process defines the meaning of "next" entity for action A.

Actions B and C allow the client program many opportunities to perform processing on individual entities, summaries after families of entities are generated, and a summary at the end of the tree. Process loops are generally organized with the generator at the top of the loop. Following this is a section of code that tests which action was effected by the generator, and at what group. If an entity is generated in a group where retrievals are to be made, control is passed to a section of code that makes the retrievals from that entity and processes the data.

If a "done with family" action is signaled on a group for which family summaries are being made, control is passed to a section of code that effects the summaries for that group. After each of these code sections is complete, control is returned to the generator to take the next step. Eventually the "done with tree" action is signaled. The process then exits from the processing loop and executes terminal processing.

Only the generator looks at the access tree data structure, and it confines the process to entities on the tree. Entities not on the access tree simply do not exist, as far as the process loop is concerned.

The process can direct the generator to break from its normal sequence of "next" entity steps. Thus further screening by data dependent "match" conditions of the entities to be processed can be done. When an entity of a particular group is reached, the client process can retrieve data from it and test for a match condition. If the match condition is satisfied, other data are retrieved from the entity and entered into the process. Then the generator is told to generate the next entity, usually an offspring of the current entity.

But if the match condition is not satisfied, the client program goes directly to the generator, calling it with a skip option that causes all descendants of the current entity to be skipped. Normally the next entity generated in this case is a sibling of the current entity. Other skip options are available. Thus the process has final control over the entities entering the process, within the confines of the access tree.

4.7 *Summary of Access Trees and Generators*

The generator and access trees provide a mechanism for efficiently accessing in a subtree of a data base those entities which may supply the data needed to process a particular request. Access trees have a natural derivation from English clauses that delimit the scope of a request. The generator can directly access the entities specified by an access tree. Thus, together, they constitute a very significant bridge between natural-language query and efficient retrieval algorithms.

V. FUNCTION EVALUATOR

An application program interacts with a data base at each entity generated. For retrieval processes, the values to be displayed are often combinations or functions of the stored data. The function "NET SALES per store divided by EARNINGS" has one value per entity of the store group. The values for this function could have been given a name at build time, and established as a field of the data base. In

theory, any possible function that has one value per entity of some group could be a stored field of that group. In practice, only those of sufficiently high usage are stored, and the others are computed on request. Thus there must be some mechanism which can deliver upon request the value of a stored field or of a function of stored fields. This mechanism is called the *function evaluator.*

This section presents a definition of several classes of fields whose values are not stored but can be derived from the stored data and from the hierarchical structure itself.

5.1 *Summarizing a Field*

A hierarchy provides a structure for efficiently summarizing data. For example, a user of the sample data base may require the total DOLLAR SALES for each store. To obtain such a total for a given store, the values of "DOLLAR SALES" must be summed over each department in that store. Repeating this summation for each store produces a set of values for the derived field "total DOLLAR SALES per store," defined at the store group. This type of function is called a *level raise* because it raises the level of definition of a field from one group to a higher group.

The set of entities used to evaluate a level-raise function for the store group consists of one entity of the store group and a collection of descendants of that entity. A *subtree under group G* is defined as an access tree containing at most one entity of group G. Hence, in the descendants of G a subtree under group G may branch out, but from G up to the root there is only one entity path. Let G be an ancestor of G′ and f′ a stored or derived field of G′. A level raise produces a value for a field f of group G by summarizing a field f′ of group G′ across the G′ entities in a subtree under group G. Values entering into the level raise are those of f′ for entities of the subtree. The set of values it produces for all entities of G defines a field f of group G.

In the level-raise function "total DOLLAR SALES per store," "total" is an instance of a level-raise operation, "store" is G, and "DOLLAR SALES" is f′, where G′ is the department group. In order to construct an efficient computation algorithm, level-raise operations are restricted to those which can operate sequentially on a set of values for the field f′ to produce a single value of f. Examples of level-raise operations are total, average, minimum, maximum, and standard deviation for numeric-valued fields; any, all, and none for logic-valued fields; and concatenation for character-string fields.

5.2 *Retrieval Within an Entity*

A derived field measuring average sales might be defined as "DOLLAR SALES divided by SALES FORCE." Since both component fields are defined for each department entity, their quotient is also defined for each department entity, and hence describes a derived field of the department group. Any function of fields is called a field function. The operands of a field function may be level-raised, as in "maximum DOLLAR SALES per store divided by EARNINGS." This is a function of two store fields, "maximum DOLLAR SALES per store" and "EARNINGS." A field function can be defined in terms of fields of different groups, as in "DOLLAR SALES divided by EARNINGS." The numerator is a department field, the denominator a store field. The expression has one value for each department, and hence defines a field of the department group.

A field function for group G is defined as any function of constants, fields of G, and fields of ancestors of G. These fields may be stored or derived. A field function produces a new field of G. It is applied at a single entity and produces a value defined for that entity. The class of field functions contains such operations as the standard arithmetic, Boolean, and trigonometric operations; logarithms; and IF–THEN–ELSE assignments.

Arbitrary nesting of level-raise and field functions is well defined since a function of either class generates a field. An example of such nesting is "maximum per store of (DOLLAR SALES minus total per department of (PURCHASE COST times the sum of ON ORDER and BACK ORDERED))." This expression is equivalent to the following statements:

x = PURCHASE COST times the sum of ON ORDER and BACK ORDERED
y = total x per department
z = DOLLAR SALES minus y
f = maximum z per store.

x, y, z, and f are derived fields. x is an item field, y and z are department fields, f is a store field.

5.3 *Entity Specification and Qualification*

The functions considered thus far generate new fields. The next discussion treats functions which modify the set of entities over which a field is evaluated. An *entity-specification* function describes a process

which, given a subtree under group G, selects another subtree under group G using only the intrinsic order of the entities in each group or a constant entity designation. In the sample data base, the years and months are ordered within their respective families. Therefore, the request "ADVERTISING divided by previous year ADVERTISING" is defined for each month. Given a particular month the numerator is obtained directly, whereas the denominator is retrieved for an entity whose location in the tree structure is determined relative to the given month by the operation "previous year." This is called a *relative entity specification.*

Constant entity specification denotes a fixed subtree under group G which overrides the given subtree. The ratio of NET SALES to January NET SALES describes a constant entity specification.

Hierarchical structures make entity specification an efficient process for selecting the entities over which to evaluate an expression. A more general but less efficient selection is that of entity qualification, as in the "with" phrase of "average per year of (BACK ORDERED with PURCHASE COST greater than 500)." Entity qualification is independent of the order of entities in a group. All entities must be examined according to a criterion, such as "PURCHASE COST greater than 500." Each entity is assigned the value "accept" or "reject." When an entity is rejected, all of its descendants are rejected as well. The descendants of an accepted entity are likewise accepted as far as that criterion is concerned. The qualification process is inefficient because data must be retrieved for all candidate entities; in entity specification no test data are retrieved from any entities. Hence, the earlier example with a January specification might be equivalently phrased "NET SALES divided by total NET SALES with MONTH NAME = JAN." In the denominator, each month entity must be examined to determine whether or not its name is January. Although constant entity specification can be contorted into entity qualification if entity identifiers are stored values, the relative entity-specification functions, such as "previous," cannot be expressed at all with entity-qualification, unless the family-order relations are also stored as data values.

In summary, level-raise and field functions can be computed for all entities of a group. A function of either type produces one value for each entity of a group, and hence defines a nonstored field of the group. Entity specification and qualification functions produce a subtree at each entity of a group. A function evaluator enables the user of an interactive data system to dynamically define and redefine derived

fields and retrieve values for these fields, all in an interactive communication in real time. A field is either a data-base field or a function of fields, such that it has one value for each entity of some group. A function evaluator enables a client program to retrieve values of a field without having to distinguish whether the field is stored or derived. To accomplish this, a function evaluator must be able to accept function definitions during the dialogue, rather than have them compiled into machine-executable code. In addition, it must be capable of evaluating arbitrarily nested functions if the user can truly ignore distinctions between stored and derived fields. Otherwise the user would be constrained to use only specific types of fields in each class of functions.

5.4 *The Retrieval Process*

This section presents an algorithm for an evaluator capable of computing values for derived fields over a hierarchical data base. The algorithm is a recursive procedure.

Input: A four-tuple: (f, G, t, S).

Arguments:

f: A field defined at group G.
G: A group.
t: An entity-selection function for group G.
S: A subtree under group G.

The argument f can be a stored field, v; a field function, p; or a level-raise function, l.

p: A field function whose nth argument is the triple (f_n, G_n, t_n):
  f_n: A field for group G_n.
  G_n: A group, either G or an ancestor of G.
  t_n: An entity-selection function for group G_n.
l: A level-raise function with three arguments:
  f′: A field for group G′.
  G′: A descendant of G.
  t′: An entity-selection function for group G′.

The argument t can be an entity-specification function, s, or an entity-qualification function, m.

s: An entity-specification function with one argument:
  t_1: An entity specification for group G.
m: An entity-qualification function with three arguments, having the same definition as a triple in p, above.

The algorithm for this evaluation is as follows.

1. If t has the form $s(t_1)$, perform the specification function s on S at G to produce a new subtree S_1; then evaluate (f, G, t_1, S_1). Return.
2. If t has the form $m(f_1, G_1, t_1)$, set t′ to null, evaluate (f_1, G_1, t', S), and perform the qualification function m on the result to produce S_1. If S_1 contains an entity of G evaluate (f, G, t_1, S_1) and return; otherwise return a null value.
3. If f is a stored field, v, retrieve its value for the entity of group G on S. Return.
4. If f has the form $p\{(f_1, G_1, t_1), (f_2, G_2, t_2), \cdots\}$, do step 4a for each component (f_n, G_n, t_n); apply p to the resulting values and return.
   4a. If $G_n = G$, evaluate (f_n, G, t_n, S); otherwise construct S_n as the subtree under group G_n containing the entity of G_n present on S and containing all of that entity's descendants, and evaluate (f_n, G_n, t_n, S_n).
5. If f has the form $l(f', G', t')$, select S′ as a subtree of S under group G′, containing the first G′ entity of S. Evaluate (f', G', t', S'). For each succeeding entity of G′ on S, do step 5a. Return.
   5a. Construct a subtree S′ for group G′ using the G′ entity specified in step 5, and evaluate (f', G', t', S'); then apply l to the previous result and the new value.

This algorithm is summarized by the following production.

$$
\begin{aligned}
(f, G, t, S) \rightarrow\ & (f, G, s(t_1), S) \mid \\
& (f, G, m(f_1, G_1, t_1), S) \mid \\
& (v, G, t, S) \mid \\
& (p\{(f_1, G_1, t_1), (f_2, G_2, t_2), \cdots\}, t, S) \mid \\
& (l(f', G', t'), G, t, S)
\end{aligned}
$$

5.5 *Unavailable Data*

Some of the fields in a data base may not have values at some time. For example, a new stock item, X, may be ordered although its selling price has not yet been determined. Now someone designing a new product using parts X, Y, and Z needs to determine the total selling price of the components, that is, a summation level raise restricting items to X, Y, and Z. Clearly, if the value of "selling price" is unavailable for X, then the value of the sum is also unavailable. Should an unavailable unit of data be assigned the value zero, the level raise would produce 0 plus Y plus Z as the material cost of the product—an

alarming situation at best. Similarly, the field function "selling price times IN STOCK" must yield a result of unavailable if the value of either operand is unavailable. Notice that this situation is not one in which the user has entered a value of null, but rather one in which the data are not available or have not been entered. NA (*not a*vailable) indicates that no significant data are present.

Unavailable values occur as well in logical-valued fields, particularly when level-raising with operators of "any" and "all." "Any X" has the value "true" if the field X has the value "true" for any descendant of the current entity. "All X" is true only if X is true for all descendants. If X has a value of NA (not available) it could be true or false, but we do not know which. Representing TRUE, FALSE, and NA numerically such that TRUE < NA < FALSE, an investigation of each possible situation will verify the following:

$$\begin{aligned} \text{any X} &= \min(\text{X}) \\ \text{all X} &= \max(\text{X}) \\ \text{any not X} &= \text{not all X} \\ \text{all not X} &= \text{not any X} \end{aligned}$$

where: not NA = NA,
not TRUE = FALSE,
not FALSE = TRUE.

In criteria evaluation, such as testing if X is less than Y, the result must be NA if the value of either X or Y is NA. If either value is unknown, the criterion may or may not be satisfied; the result is unavailable.

NA is a value which describes the absence of a value. Entity-qualification functions produce an accept or reject status. "Reject" describes the absence of an entity. In "average (ON ORDER with PURCHASE COST greater than 500) per department" the qualifier rejects entities in the averaging. "Reject" is needed as a value of stored and derived fields as well. It enables IF–THEN–ELSE statements to express entity qualification. Moreover, suppose that before April of a certain year the entire sales of a department was recorded in the NET SALES field, while after that time the sales were broken down into NET SALES and SALES TAX. Now if SALES TAX is given the value zero in the first months, the expression "average tax per year" for any department will produce a peculiar result because of the zeroes averaged in. NA is unacceptable as the value of SALES TAX in the early months since it would cause a field such as "average (NET SALES plus SALES TAX) per year" to return a value of NA,

although the true value is well defined. Instead the stored value "reject" is used. Operationally "reject" is the identity for PLUS, MINUS, AND, and OR. For other operations, if any operand has the value "reject" the result is also "reject." Hence, when adding 5 to "reject" the result is 5, and when testing whether or not 5 is less than "reject" the result is "reject."

VI. POINTER AND DATA STRUCTURE

The previous sections have defined Master Links from the user's point of view. To implement the features described, and to achieve the other stated goals (high performance in a time-sharing environment, portability, and multiple concurrent users), require a new approach to the layout of the data base elements onto the host systems files. In the classical approach to data-base design, records are used for many purposes. One purpose is to associate data values; another is retrieval efficiency: data values used together are stored together in a record. Update interlocking is a third use: exclusive control of a record or set of records is granted to a process so that it may make a series of changes to their contents without interference from other processes.

Master Links provides three distinct tools to achieve these three results, without having to rely on physical-storage records:

(*i*) Association of data items is accomplished by the pointer structure described in Section 6.2.1.
(*ii*) Retrieval efficiency is achieved by a parametrized layout of the data values into a data block, Section 6.2.2.
(*iii*) Multiple concurrent updates by many users is made possible by the concept of a lock unit, described in Section 6.1.

With Master Links, programs (and people) work with the logical structure of a data base, unhampered by its physical layout on the direct-access files. The details of record and file boundaries are invisible at the logical level. The basic concepts of Master Links, as well as all or most of the detail logic that implements the concepts, are independent of any machine or host system.

The mechanism used to achieve this freedom is the stream.

6.1 *Streams*

A *word* is an arbitrary unit of storage, the meaning of which is determined by the host system. A *stream* is a series of words. A particu-

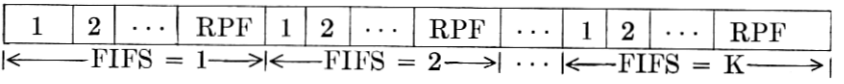
lar word in a stream is identified by its position in the series. This ordinal number is called WIS (*word in stream*). The size of a stream in words may be increased or decreased to accomodate changes in data-base size. A data base is built from several streams. A stream therefore needs an identifier, which is designated S. A particular word of a data base is completely determined by S and WIS. A stream is made up of a series of records, where a record is defined as a set of words transmitted between primary and secondary storage by the host system as a single unit. A pointer into a stream is always in terms of WIS, never in terms of record number, or word in record, or any other host-system concept.

Exclusive use of a segment of a stream, called a lock unit, is required for updates. In fact, the lock unit may be a record, a set of records, a file, etc., depending on the capabilities of the host file-management system. The interlocking of multiple concurrent updates anywhere in the data base occurs correctly regardless of the boundaries of the lock units. A lock unit is, from the traffic point of view, a resource. It is important that as few lock units as possible be locked for the shortest time possible in servicing an update, and that a lock unit cover the smallest possible area.

One can plan efficient use of streams in terms of S and WIS alone. The probability that two words, WIS and WIS + K, of the same stream are in the same record is 1 for $K = 0$ and linearly decreases to zero with the magnitude of K. That is, words close together in the stream are likely to be in the same record. The same is true of two words and a lock unit. Thus, by adopting a probabilistic viewpoint, efficient use of streams can be planned without a detailed knowledge of file and record boundaries.

Streams are implemented in the Master Links software using direct-access files. Catalogued, direct-access files with a fixed number of unformatted, fixed-length records are used because such files are generally available and operate efficiently on existing time-sharing systems. This is the simplest and most commonly available type of direct-access file available today. A *file set* is a (possibly null) series of direct access files. Each file of the series has the same dimensions: RPF records per file, and WPR words per record. A file of a file set is identified by its ordinal position in the series; this number is called FIFS (*file in file set*). A file set forms a series of computer words when the files are viewed as logically concatenated in the order of their FIFS numbers, with the records of each file being logically concatenated in the order of their record numbers. Thus, graphically, a file set

can be pictured as follows:



Each box represents a record; its record number is shown inside the box. This construction does not imply that the files, or even the records of a file, be physically concatenated on secondary storage. The actual allocation of files upon direct-access devices is a responsibility of the host file-management system. A file set can grow, and the unit of growth is a file. A new file is assigned the next ordinal number available for the file set to which it is assigned. A file set is, therefore, a finite but extensible series of computer words, and hence is an implementation of a stream. Several file sets are used to implement the several streams of a data base.

To access word WIS in stream S, the file set for S is determined from S. Then the dimension RPF and WPR are determined. By integer division and modulo arithmetic on WIS, the FIFS, the record number, and the word in record are calculated. Thus the word is described in terms of files and records, and can be accessed.

6.2 *Pointer Structure*

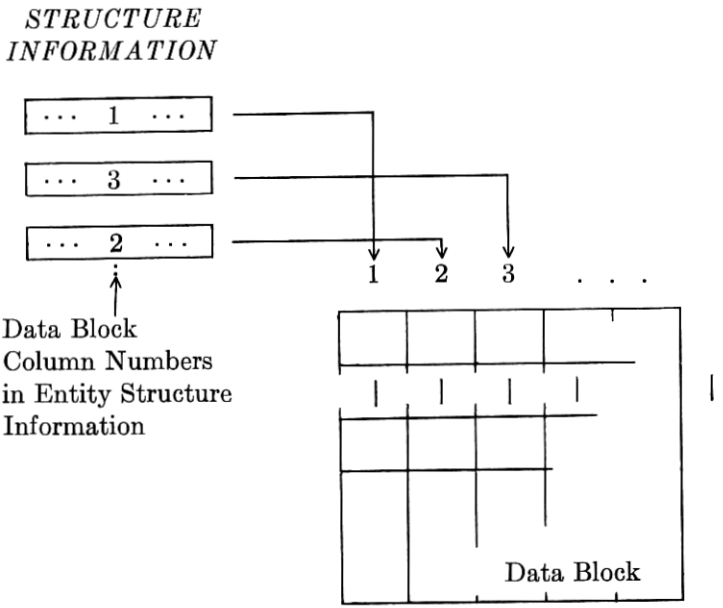
This section describes the pointer structure of Master Links. The design derives from the following goals:

(*i*) The data structure must be designed for auxiliary storage.
(*ii*) Data may be updated and elements added to and deleted from the hierarchy by simple, efficient algorithms.
(*iii*) These operations serve multiple concurrent users.
(*iv*) The integrity of the data structure must be maintained in the event of a machine failure.
(*v*) A single set of algorithms must access any hierarchical data base.
(*vi*) The storage of the hierarchy must provide efficient hierarchical traversal; that is, at any position in a hierarchy, the accessing routines must be able to directly address any subordinate or sibling.

6.2.1 *Development of the Pointer Structure*

In Section II, entities were described as having data and structure. Structure connects an entity to its relatives. In order to attain efficient traversal from an entity to any of its siblings or offspring, regardless

of the number of offspring, the structure part of the entities in a family must be stored contiguously. Otherwise, a sequence of reads would be needed to follow the chain of sibling pointers through auxiliary storage. For families to grow in real time and still have their members contiguous, the growth process requires copying the old family description to some available space and then appending new entities. If data of an entity were stored with the structure, this copy process would become expensive and would leave large amounts of space vacant. Rather, the data are stored separately in a matrix, or data block. The columns of the matrix correspond to entities, the rows to fields. Therefore, the structure information for an entity must include a reference to the data-block column number assigned to that entity.



Column numbers, rather than absolute storage locations, are used to reference the data block, allowing separation of structure from data.

The hierarchical structure is completely divorced from the data storage structure. Whichever way the matrix is stored—by row, by column, by submatrix—has no bearing on the hierarchical structure information. The Master Links data-block storage arrangement is discussed in Section 6.3.

The offspring of an entity are linked to the entity by means of pointers which specify their storage locations in a stream. Every

entity has one offspring pointer for each family of offspring. The collection of the data-block column number and the offspring pointers for one entity is called an *entity pointer set.*

data-block column number (one word)	pointer to an offspring family (one word)	pointer to another offspring family (one word)	...

An entity pointer set (EPS) contains the structure information for an entity in a contiguous stream of words.

Since each entity in a group has the same number of offspring families, the entity pointer sets for all entities in one group are the same size. Therefore, since siblings are adjacent, a pointer to the beginning of a family provides direct access to each member of the family. If the siblings were chained together, a null pointer in the chain would indicate the end of a family. However, with the siblings contiguous, the size of each family must be stored instead. It is most convenient to store the family size just before its first entity pointer set.

Family Size (n)	EPS_1	EPS_2	...	EPS_n

A family of n entities is described by a one-word family size, n, followed by n entity pointer sets (EPS), one for each entity in the family.

The entity index is the ordinal position of the entity pointer set in the physical pointer structure for one family. In deleting an entity from a family, it is important not to change the entity index of other entities in the family. Therefore, a special flag is encoded in the entity pointer set to show that the entity is deleted. Entities which are physically present in a stream but which have been flagged as deleted have *reserved* status. They are not considered present in the hierarchy. If the delete flag of a reserved entity is later turned off, the entity becomes *active* and is then treated as part of the hierarchy.

All the family pointer sets of a group are stored in a stream. This stream is called the *master link* of the group.

In summary, the description of the entities in a group is stored in a stream of words. A group is made up of families. A family is described by a family size, followed by a set of pointers for each entity in the family. An entity pointer set consists of a data pointer and a collection of offspring pointers. The data pointer is a column number of the data block for the group. The offspring pointers specify word positions of the appropriate streams. Entities allocated in the pointer

structure are either active or reserved; the latter do not appear in the logical structure of the data base.

6.2.2 *Pointer Structure Algorithms*

The algorithms which modify the data-base structure must be safe over abnormal termination of the process. A process can be abnormally terminated in many ways, such as by a user interrupt, a hardware or software failure in the host environment, or an intentional stop by the host system for exceeding some resource allocation. The key to making a transaction safe over unexpected terminations is to first allocate any new space needed, then fill out the new space, and finally link the new space with the old by a single pointer. If the write of that pointer succeeds, the new information is secure. If it fails, the area remains disconnected and wasted, but the data structure remains intact.

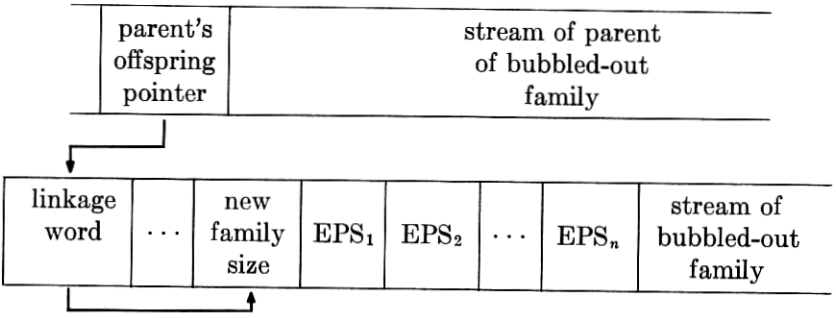
The algorithms must also work correctly when several concurrent users are trying to execute them. This is assured by locking a word to be updated (the lock unit must cover the entire physical record containing that word), reading the record to obtain a fresh copy, updating the value and any other values in the same record, re-writing the record, and then relinquishing the lock.

There are three functions which modify the pointer structure. An entity's status can be reversed (from active to reserved or back again); new entities can be added to an existing family; and a family can be created and linked to its parent.

To reverse an entity's status, the stream location of the first word of the entity pointer set is computed. The process then requests of the host environment exclusive control of the lock unit containing this word. When exclusive control is authorized, the record is read, the required word is updated, the record is written back to auxiliary storage, and the exclusive control is relinquished. If the process is terminated before the write, it can be re-executed because nothing has been altered. If it is terminated after the write, a restart procedure can read the record to determine that the update was successful, and skip re-doing it.

Extra entities are added to a family by first locking the record which contains the word pointed to by the parent's offspring pointer. This word, called the *linkage word* of the family, is the single word to be made a pointer to the new space. The first time that this algorithm is applied to a given family the linkage word contains the family size. Next, a sequence of contiguous words in the stream is allocated and

locked. The existing entity pointer sets of the family are copied into the new space; new entity pointer sets are constructed by generating parent pointers, new data column numbers, and null offspring pointers; any of the new entities that are to be reserved for future assignment are marked reserved; and, finally, the new family size is placed into the first word of the new space. After this new area has been written and unlocked, the linkage word is updated to point to the new area. The record containing the linkage word is then written and unlocked. This update of a single word links the new family pointer set to the existing hierarchy and disconnects the old family description from it. The whole process is called a *bubble-out*; the wasted space containing the old family description is called a *bubble.*



The linkage word is a single word connecting the new family description to the previously existing structure.

Notice that the linkage word had to be locked from the start to the finish to keep other users from adding to the same family at the same time. Such interference could cause the family update to be lost entirely.

Creating a new family adds one complication to the bubble-out algorithm. Here, there is no linkage word, so the parent's offspring pointer must be locked and re-read instead. If the parent's offspring pointer to this group is still null after locking and re-reading, it is updated to point to the new family. If non-null, another user has in the meantime attached a family to the parent, so either the status-change or the bubble-out algorithm is entered.

6.2.3 *Further Considerations*

There are several considerations which affect the performance of the bubble-out algorithm. Among these are the handling of available space, the disposition of bubbles, and the use of an entity reservation

factor for assigning sets of entities at one time. These considerations involve important balances between execution speed and auxiliary-storage space.

For each master link, the WIS for the next available word of the stream is stored as a header to the master link. The bubble-out algorithm first locks the header, then the linkage word. The new family size is calculated and the header is updated, written, and unlocked. Then each record to be written is locked, written, and unlocked before the next one is locked. Since the linkage word is already in use it must lie somewhere between the header and the available space. Hence, the locations locked are within one stream, and in numerically increasing order of WIS. This precludes any chance of a deadlock since a stream is stored in an ordered set of records. As for the bubbles, a utility can be run in the background from time to time to remove them. The bubble-out algorithm assigns column numbers to the new entities, so it must update an available-data-block-column word at the same time it updates the stream available-space word. Hence, the appropriate place to store this available-column number is the master link header.

A parameter of the bubble-out process allows the reservation of extra (inactive) entities. Assigning the extra entities causes only one bubble-out for all of the entities created, releasing only one bubble. The status-change algorithm is considerably more efficient than the bubble-out algorithm, so the average entity creation cost is reduced. The bubble-out algorithm assigns data-column numbers to the new entities, so data for the entities of a family are stored in sets of contiguous columns. As explained in the next section, this usually makes data accessing more efficient than if the columns of a family were scattered throughout the block. The cost of reservations is the cost of carrying the extra entities in storage before they are activated. The reservation factor can specify a constant increase or a growth factor as a percentage of the current family size.

6.3 *Data Blocks*

Data-base processes have a strong tendency to access either values of many fields from a few entities, or of a few fields from many entities. In the latter case the entities tend to be requested in an order determined by the hierarchy of the data base. A given data base will have a mix of these two types. If the first type predominates, it is efficient to order the values column-wise. If the second is more common, efficiency is gained by arranging the values row-wise, and by assuring

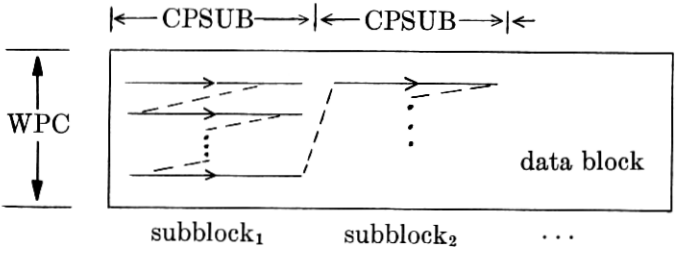
that entities processed together occupy, with high probability, adjacent columns.

For Master Links, the mechanism for storing values is the *data block*. A data block is a matrix of values with one column for each entity, and one row for each field. An element of this matrix is one value, which takes up one or more words. The number of words needed to store one value is a parameter of the field, called its *size*. Thus all the values of one row are of the same size, but the values in two different rows may have different sizes. For each group of a data base there is one data block. The arrangement of a block into records is controlled by several block parameters which are attributes of the corresponding group. These parameters provide a variety of possible structures, of which the column-wise and row-wise layouts are special cases. Using the block parameters as inputs, a single algorithm can access any block arrangement.

6.3.1 *Layout of a Data Block*

A data block is stored in one stream. The block has two parameters, words per column, WPC, and columns per subblock, CPSUB, which are used to divide the block into subblocks. WPC is an integer equal to the sum of the sizes of the fields. This is the vertical dimension, in words, of the block, and also of the subblocks. The parameter CPSUB defines the horizontal dimension of a subblock. The first subblock consists of columns 1, ···, CPSUB; the second is columns CPSUB + 1, ···, 2·CPSUB; etc. A block is then a horizontal concatenation of subblocks.

The ordering of words in a block is established by keeping the words of a multiword value together, and arranging the values in row-wise order within a subblock, and then concatenating the subblocks from left to right. This is the ordering used to store a block in a stream. The order of values in a block is illustrated below. Each solid arrow indicates contiguous storage of CPSUB values.



It should be noted that the subblock plays no further role except to establish an ordering. It does not correspond to a file, a record, or any other host-system concept. In fact, the mapping of block words onto stream words is performed without concern for record or file boundaries.

6.3.2 *Example of Block to Stream Mapping*

Consider a block with three rows and five columns, where SIZE (words per value) is set to 1, 2, and 1 for rows 1, 2, and 3, respectively. Suppose a numeric value takes one word, and character value takes one word for every four characters. Then a sample of the block looks like:

1.00	2.00	3.00
AAAAAAAA	BBBBBBBB	CCCCCCCC
10	20	30

4.00	5.00
DDDDDDDD	EEEEEEEE
40	50

Words per column (WPC) is four. The mapping of this block onto a stream is shown below for three different values of CPSUB.

CPSUB = 1:

1.00	AAAA	AAAA	10	2.00	BBBB	BBBB	20	3.00
CCCC	CCCC	30	4.00	DDDD	DDDD	40	5.00	
EEEE	EEEE	50						

CPSUB = 5:

1.00	2.00	3.00	4.00	5.00	AAAA	AAAA	BBBB	BBBB
CCCC	CCCC	DDDD	DDDD	EEEE	EEEE	10		
20	30	40	50					

CPSUB = 3:

1.00	2.00	3.00	AAAA	AAAA	BBBB	BBBB	CCCC
CCCC	10	20	30	4.00	5.00		DDDD	DDDD
EEEE	EEEE			40	50			

6.3.3 *Accessing a Value from a Data Block*

For the nth row of a block, SIZE_n is the number of words for one value in the row, and DIC_n is the *d*isplacement *i*n *c*olumn of the row which is defined as one plus the sum of the SIZE's of the first $n - 1$ rows. Thus for the example in Section 6.3.2.:

n	*SIZE* (words)	*DIC* (words)
1	1	1
2	2	2 $(= 1 + \text{SIZE}_1)$
3	1	4 $(= 1 + \text{SIZE}_1 + \text{SIZE}_2)$

The inputs to the algorithm for accessing a value of a data block are the identifier of a field, and a column number, COLNO. DIC, SIZE, and the group of the field can be determined since they are attributes of the field. WPC, CPSUB, and the stream identifier, S, are then determined since they are attributes of the group. From these the following calculations are made using integer arithmetic:

$$\begin{aligned}
\text{SUBBLKS} &= \text{COLNO}/\text{CPSUB} \\
\text{WPSUB} &= \text{WPC} \cdot \text{CPSUB} \\
\text{WORDSABOVE} &= \text{CPSUB} \cdot (\text{DIC-1}) \\
\text{WORDSLEFT} &= ((\text{COLNO-1}) \text{modulo CPSUB}) \cdot \text{SIZE} \\
\text{WIS} &= 1 + \text{SUBBLKS} \cdot \text{WPSUB} + \text{WORDSABOVE} \\
&\quad + \text{WORDSLEFT}
\end{aligned}$$

SUBBLKS is the number of subblocks previous to the subblock containing the sought value. WPSUB is the words per subblock. WORDSABOVE is the number of words above the sought value in its subblock. WORDSLEFT is the number of words to the left of the sought value in its row of the subblock. WIS is the word in stream of the first word of the sought value. Hence S and WIS and SIZE are known. These are the inputs needed to access a stream, as described in Section 6.1.

6.3.4 *Row-wise, Column-wise, and Intermediate Layout*

Note that when CPSUB equals 1 the order of storage is column-wise. When CPSUB equals the words per record, storage is row-wise. An intermediate setting of CPSUB between 1 and WPR will for certain usage patterns achieve performance superior to either column-wise or row-wise organizations. This is illustrated in the following example. Suppose that a block has 100 rows and 100 columns. Suppose that process R uses all the data in one row, and that process C uses all the

data in one column, and that these processes are run equally often. Suppose also that WPR = 100. Then if CPSUB = 1, C must read one record, R must read 100 records. The average records per process run is 50.5. If CPSUB = 100, C must read 100 records whereas R reads only one, for the same average. If CPSUB = 10 the block is divided into a 10 × 10 checkerboard of records. Each process must read 10 records for an average of 10 records per process. This is the optimum CPSUB for this example.

A utility called CONVERT can be used to change a block from one value of CPSUB to another. Modifying CPSUB adjusts the data base to reflect a changed or unpredicted pattern of usage. It also makes possible periodic changing of the data layout to conform to a cyclic pattern of usage. All programs accessing a data block do so in terms of column numbers and fields. The assignment of a value to a block, row, or column is unchanged by CONVERT, and hence no program is invalidated.

6.4 *Review of the Advantages of Streams*

The process of design involves constructing transformations to achieve a desired structure using available structures as media. The desired structures for Master Links are a hierarchy and data blocks. The transformation is carried out in two steps, from direct-access files into streams and from streams into blocks and pointer sets. The structures and their attributes are summarized in this table:

STRUCTURE	*ATTRIBUTES*
Catalogued Direct-Access Files	Internal Identity (FSNO, FIFS) NAME Records Per File (RPF) Words Per Record (WPR)
Streams	Stream Identity (S) Word In Stream (WIS)
Blocks	Block Identity (B) Words Per Column (WPC) Columns Per Subblock (CPSUB) ROW *n* SIZE for ROW *n* Displacement In Column (DIC) For ROW *n* Column Number (COLNO)

It is no accident that streams, the intermediate structure, are so simple. They amount to an idealized direct access media. The advantage of using this intermediate structure is that it crystalizes the separation of the Master Links structures from the physical-storage media. The programs that implement the desired structure are coded independent of the actual direct-access media. In particular, the parametrized layout of a block would be very cumbersome to implement directly in terms of files, records, and word in record. It is very straightforward in terms of word in stream.

Since the Master Links structure is separated from the physical media, media management utilities such as CONVERT can be run without altering any Master Links programs. The separation of structure from media also makes possible the implementation of alternative media. Streams might be implemented as arrays in primary storage for small data bases, or implemented in an entirely different manner upon direct access files, such as with all streams in one extensible file. Finally, this separation enhances the portability of Master Links allowing most of the logic of the system to be based on a machine-independent direct-access structure.

VII. EXPERIENCE AND FUTURE EXTENSIONS

An experimental version of Master Links was operational in 1970. It was based on the concepts and supported all the features reported in this article, except portability and certain utilities. A production version was completed in May 1972. It supports all features, including portability, all utilities, two different stream implementations, plus improved performance. These versions have been used for a variety of different types of projects: inventory, financial, budget and resource allocation, and construction program administration data bases. Together with the Natural Dialogue System[1] it forms the basis of the Off-The-Shelf-System.[2]

Several efforts are under way to extend and improve the system:

(*i*) Networks—allowing a group to have more than one parent.

(*ii*) Field length data—allowing strings of data, such as a time series of values or a paragraph of text, to be stored efficiently as a single value.

(*iii*) Function evaluation—computing in parallel all requested level raises that are defined over a common subtree. Hence, in "total IN STOCK divided by total ON ORDER," the numerator and denominator totals will be taken simultaneously in a single pass over the item entities of a department.

(*iv*) Access tree generator—allowing execution-time determination of the hierarchy. Suppose, for instance, that a new item field, item class, describes the level of supervision required to approve acquisition of the item. Then "total IN STOCK by item class" is a meaningful function, but the hierarchy formed by partitioning item entities according to their values of item class must be computed at execution time, if the "by" field is allowed to be arbitrarily specified by the user.

(*v*) Report generator—accepting a description of the content and layout of a report and on request producing an instance of the report.

REFERENCES

1. Puerling, B. W., and Roberto, J. T., "The Natural Dialogue System," B.S.T.J., this issue, pp. 1725–1741.
2. Heindel, L. E., and Roberto, J. T., "The Off-The-Shelf-System—A Packaged Information Management System," B.S.T.J., this issue, pp. 1743–1763.