IBM Research

THE GAMMA-0 n-ARY RELATIONAL DATA BASE INTERFACE SPECIFICATIONS OF OBJECTS AND OPERATIONS

D. Bjørner/E. F. Codd/K. L. Deckert/
I. L. Traiger

April 11, 1973

RJ 1200

Yorktown Heights, New York

San Jose, California

Zurich, Switzerland

THE GAMMA-0 n-ARY RELATIONAL DATA BASE INTERFACE SPECIFICATIONS of OBJECTS and OPERATIONS

by

D. Bjørner
E.F. Codd
K.L. Deckert (*)
I.L. Traiger

IBM Research Laboratory
San Jose, California

ABSTRACT: GAMMA-0 is a low-level, single-user interface for the manipulation of collections of relations of assorted degrees. It is intended to be used as a base for implementing well-regulated sharing of data amongst many users, and higher-level interfaces such as the "relational algebra", the "data sublanguage ALPHA" and others.

RJ 1200 (#19229)
April 11, 1973
Computer Sciences

(*) Work done while Dr. Deckert was on assignment from IBM General Products Division

BLANK PAGE

SUMMARY:

This report presents the preliminary specifications for an n-ary relational data base interface system, referred to as GAMMA-0. This interface presents to the user a collection of relations of assorted degrees. Each relation consists of a control tuple followed by zero, one or more data tuples. The ordering of data tuples is determined by the users who populates the relation. Each tuple is composed of n items, one for each of the domains of the relation. A relation is categorized as either a regular- or a class relation, an inversion or the unique master relation. The master relation holds certain specifications for all relations; class relat ons are all unary relations; inversions are all binary relations providing low-level indexing to a domain of some parent relation. Inversions are maintained dynamically by the system. With the exception of class items, which are variable-length byte strings, all relation items are 32-bit quantities.

A number of system generated 32-bit identifiers are defined. Relation identifiers, rids, which are unique to every relation; tuple identifiers, tids,

which are unique to every tuple in a given relation; domain identifiers, dids, which identify each domain in a given relation; scan identifiers, sids, which denote the state of the sequential scan of relations. Fourteen commands are defined: for the creation and dropping of relations; for the insertion and deletion of tuples: for the generation and dropping of inversions; for the creation-, control and dropping of scans and the output of subtuples according to scan control conditions; for the update and reading of subtuples, and for the reordering of tuples and retrieval of tuple identifiers.

CONTENTS

REFERENCES:

<1> I. L. Traiger, D. Bjørner and E. F. Codd: "Locking Extension of GAMMA-0", December 18, 1972.

<2> E. F. Codd: "A Relational Model of Data for Large Shared Data Banks", Comm. ACM, vol. 13, No. 6, June 1970, pp. 377-387.

<3> ---: "Relational Completeness of Data Base Sublanguages", Courant Computer Science Symposia (no. 6): 'Data Base Systems', New York City, May 24-25, 1971, Ed. R. Rustin, Prentice-Hall, N.Y. 1972, pp. 65-98.

<4> ---: "Further Normalization of the Data Base Relational Model", ibid: pp. 33-64.

<5> ---: "A Data Base Sublanguage founded on the Relational Calculus", IBM Research Report RJ893, San Jose, California, July 26, 1971.

<6> ---: "Normalized Data Base Structure: A Brief Tutorial", IBM Research Report RJ935, San Jose, California, November 3, 1971.

1. LANGUAGE

GAMMA-0 is a 'partial' language which together with a (suitable) 'host' language, H, forms a 'complete' language, L.

****
* FIG.01 * See page 120.
****

GAMMA-0 is one of a series of relational data base interface 'systems'. It is, we believe, a lowest level interface which supports n-ary relations in an "implementation independent" manner, based on single user traffic, and avoids redundant machinery. We further believe that GAMMA-0 is a natural stepping stone to a (tlightly) higher level interface, GAMMA-1, which supports multiple concurrent users. Members of the GAMMA-series of interfaces will be designed so that they readily allow the realization of such languages as DSL-alpha, the RELATIONAL CALCULUS and the RELATIONAL ALGEBRA <Refs. 5,3,3>. The GAMMA-0 operators are intended to support the implementation, at some higher level, of catalog support, search optimization, authorization, interlocking, journaling, audit, version maintenance and accounting. We have, however, tried to avoid the

provision of objects and operators exclusively for these services.

The operators of GAMMA-0 are of a 'non-propagating' variety, i.e. execution of operator G does not trigger the execution of any other (GAMMA-0) operator, nor a second execution of G itself. The GAMMA-0 interface is intended to be implementable using already existing lower level systems, like e.g. the Hursley Relational Prototype and the Cambridge Relational Access Method. The GAMMA-0 system however makes no special concessions to either of these, nor other, implementation systems. The set of GAMMA-0 operators is "lean" but not minimal.

At each interface in the GAMMA family the operators provide the fullest detection of data errors and program errors that is feasible without adding any data objects purely for error detection purposes, and without incurring a first-order performance penalty (that is, an additional access to secondary storage). One may think of the error detection capability at a given interface as being consistent with the "consciousness" at that interface.

For security reasons tight control will be exercised over the use of objects and operators at (GAMMA) level "J" and below, by users at level "J+1". Not all of this control can reside in the authorization machinery implemented in terms of GAMMA-0. And on "integrity" this first statement of intention: at a high level interface in the GAMMA family, the user will indicate his needs regarding the integrity of the data he wishes to access. At a lower level these needs are translated into constraints that other users must comply with - if they are to access any of the same data concurrently.

In these preliminary specifications we shall only cover GAMMA-0. Essential aspects of GAMMA-1 has been considered and were influential in certain GAMMA-0 design choices. GAMMA-1 will (hopefully soon) be documented.

1.1 FORMAL LANGUAGE DEFINITION

The complete language L is definable in three parts: it has a universe of discourse, U, a set, T, of terms and a mapping, M, from terms, t, in T, into subsets, u, in U. One can view each of these three components as pairs of elements, one from each of the two complementing sublanguages: H and GAMMA-0:

L = <<Th,Tg>,<Mh,Mg>,<Uh,Ug>>

where Um and Ug may overlap. In the following we shall only define aspects of Lg:

Lg = <Tg,Mg,Ug>

****
*   *
* FIG.02 * See page 121.
*   *
****

Tg is the set of GAMMA-0 commands; in this document 14 of these will be described. A subsequent document, <Ref. 1>, will cover an additional 5 commands. Mg will primarily be defined only with respect to each individual command of GAMMA-0. In certain parts, however, we shall have to explain the effect, i.e. the meaning, of certain allowable compositions of GAMMA-0 commands. Ug is the set of objects of GAMMA-0. In this document only three categories of objects will be described: (1) data structures, (2) identifiers and (3) scans. Ref. 1 will then detail on the (4) scope- and (5) lock objects.

In the following we shall very briefly cover some of the aspects of GAMMA-0. Further and full details then follows in sections 2-9.

1.1.1. DATA STRUCTURES

The data structure part of the universe of discourse contains the following (basic) structures:

1. Relations
2. Tuples
3. Items

(1) The GAMMA-0 system supports 4 distinct sets of relations: (1) the solitary set of the 'catalog-of-relations' relation MR (also referred to as the 'master relation'); (2) the initially empty set, R, of 'regular relations' of assorted degrees; (3) the set, C, of 'classes', also initially empty, and (4) the set, I, of 'inversions', likewise initially, as far as the user is concerned, an empty set.

FIG.03 See page 122.

(2) The GAMMA-0 system similarly supports two distinct sets of tuples: (1) the set, I, of 'tuples' and (2) the set, B, of 'byte-string 1-tuples'. The set of 'tuples' has two subsets, (1.1) the set, D, of 'relation

descriptor 7-tuples', and (1.2) the set, N, of 'regular n-tuples'. A relation consists of zero, one or more, ordered tuples.That is: tuples are members of relations. A tuple consists of an ordered sequence of one or more items.

(3) Items either denote themselves or are identifiers. Distinct objects of the environment being modelled are represented as distinct data items or other objects of the data base. The set, It, of items properly contains the set, Id, of identifiers and the set of byte string 1-tuples, B.

1.1.2 IDENTIFIERS

Relations, tuples and scan are identifiable through 32 bit encoded identifiers. A relation identifier, rid, denotes a relation and all relations are uniquely identified. If a relation has been identified (i.e. selected) -- through its rid -- then a tuple identifier, tid, (known to that relation) denotes a tuple (of that relation). Tuple identifiers are unique only within a relation. Two tuples from distinct relations may have the same tid insofar as their encoded 32 bit value is concerned. A scan identifier, sid, denotes (the state of) a particular scan of a particular relation. Several scans

may exist on the same relation. All scans, whether on one particular relation, or across all relations currently being scannned or for which scans once existed, are unique. A given 32 bit encoding may either represent an rid, a tid or an sid. The system will know from the 'context' in which this 32 bit encoding is issued which unique interpretation is to be followed. For more on identifiers, see section 3..

1.1.3 SCANS

A scan of a relation is a certain kind of an ordered reading of some or all of its tuples, subject to certain contraints and user controllable ambiguities. Such scans are objects of the GAMMA-0 system in the sense that the scan can be partially controlled by the user.

1.1.4 GAMMA-0 COMMANDS

In the next paragraph we list the underlined names of all the GAMMA-0 commands as well as explaining briefly some of their functions.

The user specifies the (1) creation of regular- and class relations; the (2 and 14) dropping of all but the master

relation, and the (13) generation of inversions. Tuples can be (3) inserted into-, and (4) deleted from regular- and class relations. The user can further specify where among the ordered tuples already positioned in the relation a new tuple is to be inserted. Tuples of a relation can then be reordered through the use of the move tuple command. Already existing tuples of a relation can be (6) updated, but only partially -- the primary key portions cannot be updated. Tuples can also be partially or completely read, through a get command. Reading and updating assumes knowledge on the part of the user of the tuple identifier of the desired tuple. If this tid is not known then their value can be retrieved through the tuple id from subtuple command, where the subtuple may specify that only the tid of tuples having certain partial values are desired. Tuple identifiers can also be retrieved through a scan (9, 10, 11 and 12). A scan in addition reads a user contrlled part of the tuples scanned. A scan is created (9), controlled through (10) set scan and tuple identifiers and tuples read out through the use of next (11) commands. A scan is finally dropped (12).

1.2 PERFORMANCE ORIENTED FEATURES

There are roughly speaking four identifiable aspects to the manner in which we have considered the performance

oriented features of GAMMA-0. (1) Above the GAMMA-0 interface we envision that (1.1) the system provided, 32 bit, tuple identifiers, (1.2) the ordering of tuples and (1.3) the availability of user generated inversions will enhance effective use of the interface language. (2) Below the interface the system maintained scans should provide efficiency over user controlled scans. (3) The user will want to investigate only items properly within tuples, and the system will in many implementations deal individually with items of tuples, we have therefore made available to the interface user commands that deal directly with such components of tuples. Thus we deal with item level operations both above and below the interface. Finally (4) the space-conserving features provided by the compact, 32 bit, representation of item- and tuple identifiers should also be taken as a performance oriented feature. This latter feature enables the user to deal with fixed formatted relation whose items do not denote themselves. These items need not be of variable field length, but identify such variable field length items in addition to identifying tuples of other relations.

1.3 REPRESENTATIONS and DENOTATIONS

ENCODED VALUES versus DENOTED VALUES

GAMMA-0 identifiers are represented as 32 bit words. By their encoded value we mean the particular 32 bit bit-pattern; and by their denoted value: "that which they denote" -- which may be either a relation (rid), a tuple (tid) of a relation (whose rid is also known, and assumed), or a scan (sid). GAMMA-0 items may either be of the fixed 32 bit word format variety -- in which case they are either identifiers or denote themselves, i.e. their meaning is known only outside the GAMMA-0 data base -- or the items are of the variable field length byte string variety, in which case they always denote themselves.

2 RELATIONS and TUPLES

For a general motivation of a data base founded on the relational view we refer to <Refs. 2-3-4-5-6>. In this section we shall first cover the various categories of relations and then (within relations, their) tuples. Given the motivation we still have left to fully motivate our particular choice/selection of the four (and no more) categories of relations and tuples, and their imposed properties.

The two main categories of relations are, insofar as most users are concerned, the category of regular- and the category of class- relations.

2.1 RELATIONS

A relation, R, is (like) a table. R always has a fixed number, nR, of domains (columns); nR is referred to as the degree of R. Domains of any relation are identified by unsigned integer valued domain identifiers, did, starting with 1 and ending with n. The number, cardinality(R), of tuples of (any) R is variable. Each relation has associated with it a unique relation

identifier, rid; these identifiers are generated by the system upon creation of each new relation.

FIG.04 See page 123.

2.1.1 MASTER RELATION

The system comes equipped with a master relation, MR, of degree 7 and (initial) cardinality 1. mrid is the corresponding master relation identifier. have the GAMMA-0 imposed defined meaning: 'type', 'degree', 'primary key mask', 'control tuple identifier', 'inversion parent relation identifier', 'inverted domain did' and 'relation identifier'. The primary key of the relation, MR, itself is domain did=7.

FIG.05 See page 124.

The master relation contains tuples describing all relations, including itself. The relations are described within MR as to their (1) type, (2) degree, (3) primary key(s), (4) control tuple tid, (5) inversion parent rid, (6) inverted domain did and (7) relation identifier

(rrid, crid or irid). Only a subset of these apply to any given relation. Thus you may think of the master relation as the (level 0) 'root' of a tree of height 1, all of whose leaves (at level 1) are the remaining relations of the data base.

```
**************
*            *
*   FIG.06   *    See page 125.
*            *
**************
```

2.1.2 REGULAR RELATIONS

The degree of a regular relation may range anywhere from 1 to some implementation defined (finite) limit (say: 2**32-1). With each relation is associated a fixed primary key which is a non-empty subset of the first 32 domains of that relation. If the relation has fewer than 32 domains then primary key is of course within those (fewer) domains. The specification of the primary key is found in MR; domain 3.

```
**************
*            *
*   FIG.07   *    See Page 126.
*            *
**************
```

Regular relations are mainly used to 'compactly' store (structured) data base information; i.e. the 'actual' data 'values' are seldomly contained in regular relations

but usually in class- relations; instead the items of tuples of regular relations may, and will typically, refer to tuples of regular- and to items, i.e. tuples, of class relations. Thus the user can specify for each domain of a regular relation whether its items refer to values (contained elsewhere) or denote values themselves. This specification is found in a control tuple permanently associated with each relation. The value of the control tuple of regular relations is defined by the user. Items of tuples of a regular relation are all of fixed 'size', we propose a 32 bit word.

2.1.3 CLASS RELATIONS

The degree of a class (relation) is always 1. Its items (=tuples) always denote themselves; in fact they are permitted to be variable field length byte strings. The primary key is, of course, domain 1.

```
**************
*            *
*   FIG.08   *    See Page 127.
*            *
**************
```

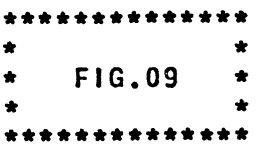
2.1.4 INVERSIONS

Inversions are always binary relations. Items of their

first domain are items of some domain of some parent-, either: master-, regular-, or class-, -relation. Items of their second domain are the corresponding tuple identifiers of the parent relation -- such that domain did=1 items correspond in value to the domain item values of the denoted tuple. Thus the inversion of a relation is with respect to one domain only -- but the same relation may have one inversion for each of its domains. Tuples of the inversion of course have their own tuple identifiers, not to be confused with those of its domain 2 items. The primary key of an inversion is, of course, domain did=2.

2.2 TUPLES

A relation, R, has cardinality(R) tuples. If this value is 0 we say that the relation is empty. The control tuple of a relation does not count in the computation of the cardinality. Every relation is described by a tuple in MR. Each distinct tuple of any relation has associated with it (in a manner defined and maintained by the system) a unique tuple-identifier, tid. Each tuple has exactly nR components, i.e. items.



See Page 128.

The system provides the user with tuple identifiers, i.e. the user can retrieve this identifier, but cannot update it.

2.2.1 REGULAR N-TUPLES

Each item of an regular tuple is user-defined with respect to its 'meaning', but is always a 32 bit 'word'. An item of a tuple of any regular relation (other than MR) either (1) denotes, i.e. refer to, some user-defined 'item' bit-string of length 32, or (2) denotes some item-, tuple- or relation- 'identifier' or (3) denotes some 'rubbish' (i.e. the user interpretation is algorithmically erroneous; and of course, the GAMMA-0 system cannot catch such errors).

2.2.2 CONTROL TUPLES

Each relation has -- from its creation, by the user -- associated with it a control tuple. The control tuple identifier, ctidR, for relation R, can be found in the corresponding relation descriptor 7-tuple (domain did=4).

The user specifies the control tuple item values when the relation is created. The idea behind the presence of this control tuple is that it serve as a 'codifier' for the corresponding domains, i.e. the domain items (in tuples), of the given relation. All class relations has a 1-tuple control, coded:<null>. The Master relation, MR, has the control 7-tuple: <null, null, null, null, null, null, null>. The control tuple cannot be updated; but it can be read.

See FIG.07 Page 126.

Thus the intended denotation of tuple items may be indicated by the corresponding control tuple items for the given relation. If some control tuple item has value: null, then the regular tuple items are said to denote themselves, i.e. they are user defined; otherwise the value of the control tuple item is taken to denote some relation identifier, rid. In that case, the corresponding tuple items are regular relation tuple- (tid) or class relation item- (iid) identifiers, denoting respectively some regular n-tuple, or some classified N-byte string 1-tuple in the relation denoted by the corresponding control tuple item. We presently take the view that if stored item values actually represent relation- or scan identifiers then the corresponding control tuple item is coded null. An example of this will be given in the very

next section.

2.2.3 RELATION DESCRIPTOR 7-TUPLES

The items of the relation descriptor 7-tuples are the only items that the system (1) checks for validity, and (2) let determine various system actions. The 'meaning' of the tuple items are, however, GAMMA-0 system imposed as follows: (See FIG.05 Page 124.)

d1: 'type':

(1) master relation, only the tuple designating MR has this type. The system checks that no other relation descriptor tuple entered by the user specifies this type. Domain 5 and 6 both specifies null.

(2) regular relation, the relation described has the degree specified by d2 and primary key mask as specified by d3; each domain item of the described relation is a 32 bit word (; the GAMMA-0 system does not inspect the 'values' of items of the designated domains). Domains 5 and 6 both specifies null.

(3) class relation, the relation is a unary relation whose items (tuples) 'stores' n-byte bit strings, d3 specifies a '1' in position 1, all other positions specifying '0's; i.e. domain 1 of the specified class is, of course, the primary key.

(4) inversion, the relation is a binary relation, whose parent relation is specified by d5; d6 the domain on which the parent relation is inverted. All domains of inversion descriptor tuples apply, i.e. are coded with descriptive information.

d2: 'degree'

(1) if d1=master relation, then this field is 7.

(2) if d1=regular relation, then this field specifies the degree of the described relation.

(3) if d1=class relation, then this field specifies 1, class relations always being unary.

(4) if d1=inversion, then this field specifies 2.

d3: 'primary key mask'

(1) for d1=master relation, this 32 bit mask (presently) specifies a '1' in position 7, all other bits are '0's.

(2) for d1=regular relation, any, except the all-zero, bit mask is allowed; its 32 bit restriction therefore imposes the system restriction that the primary keys of any such relation must be within the first 32 domains. If d2 specifies some n less than 32, then '1's in the d3 field beyond bit n can never occur|

No component of a primary key of a tuple in any relation is permitted to have value null.

(3) for d1=class relation, this field has a '1' in position 1 and all other bits '0'; the primary key of the class relation is always domain 1.

(4) for d1=inversion, this field has a '1' in position 2, '0' elsewhere.

d4: 'control tuple identifier'

(1-4) For all relations this field contains the tuple identifier, ctidR, of the control tuple in the relation, R, defined by this relation

descriptor 7-tuple, i.e. as denoted by d7.

d5: 'parent relation identifier'

(4) Applies to inversions only. This item denotes the relation identifier, rrid, of a (parent) relation of which this (described) relation is an inversion; the domain id on which the inversion took place is found in d6.

d6: 'inverted domain'

(4) (applies only if d1 specifies inversion). This field contains the did of the domain of the d5 specified parent relation on which this (described) inversion was inverted.

d7: 'relation identifier'

(1) specifies mrid.

(2) specifies some rrid, the relation identifier of the described regular relation.

(3) specifies some crid, the relation identifier of the described class relation.

(4) specifies some irid, the relation identifier of the described inversion.

2.2.4 N-BYTE-STRING 1-TUPLES

Tuples of classes are just one component tuples. They may often be referred to as either: 'items' or 'strings'. These strings are user-defined, and of bit length 0mod8 (i.e. they are byte-strings). 'BOSTON', 'Washington' and '01234567' are examples of items. Items always belong to classes, thus e.g. 'BOSTON' may belong to class Rcities, 'Washington' maybe then to classes: Rnames, Rstates and Rcities. Each item is made known to the system by entering the item into a class selected by the user. In return the system computes a unique item identifier, iid.

See FIG.08 Page 127.

3. IDENTIFIERS

All GAMMA-0 identifiers are encoded as 32-bit words. The system maintains three basic categories of 'identifiers': 'relation-', 'tuple-' and 'item-' and 'scan-identifiers'.

(1) mrid, rrid, crid, irid

(2) tid, ctid

(2) iid

(3) sid

FIG.10 See Page 129.

mrid denotes the (only) master relation identifier; rrid some regular relation identifier; crid some class relation identifier and irid some inversion identifier. tid denotes some tuple identifier; ctid some control tid; iid some item identifier, and sid some scan identifier.

The generic term: rid, is used as a sign for either of the four identifiers: mrid, rrid, crid and irid. Given a 32 bit word, i.e. its encoded value, and told that this is, i.e. represents, an rid, we have specified no direct way in which we can tell which of the four types it is.

(One can find out e.g. by looking up this rid in the MR and check domain 1 of the corresponding tuple).

The system in addition maintains a set of domain identifiers for each relation. Their denoted values range from 1 through n, where n is the degree of the relation in question. Items of this set will be denoted by did's.

In the lifetime of the nR/D/B -- relations will come and go, but two such relations, whether co-existing or existing in disjoint time intervals, shall never receive identically coded rids.

Similarly, in the lifetime of any relation within the nR/D/B -- tuples come and go, but two such tuples, whether co-existing or existing in disjoint time intervals, shall never receive identically coded tids. Uniqueness of these tids is only to within the relation, thus two tuples, belonging to distinct relations may have identically coded tids.

Finally, scans on relations come and go, but two scan identifiers, whether co-existing (on the same or distinct relations) or existing in disjoint time intervals, shall never receive identical bit encodings.

Two special identifier encoded values must be provided

for: null and '*'. For their use and meaning, see chapters 5 and 8.

The above statements amount to the definition of certain axiomatic properties of the identifier space.

4. THE GAMMA-0 DATA BASE INITIALIZATION

When an n-ary relational data base, such as the one whose properties we are now detailing, is initialized, i.e. made ready to receive e.g. the data base administrator's and other users' GAMMA-0 commands (as to its future structure and content) its:

1. relation space contains just one relation, MR, which itself has just two tuples: the all-null encoded control tuple, and the relation descriptor tuple for MR itself. MR has the relation identifier mrid. The initial tuple has the tuple identifier tid0. And the:

2. identifier space contains the five identifiers: mrid, tid0, ctidMR (the control tuple identifier of the master relation), null and '*'.

5. OPERATIONS - A MORE DETAILED SURVEY

5.1 GENERAL

Henceforth we shall use the following terminology: strings refer to N-byte string 1-tuples; items to the components of a regular relation n-tuple and tuple is used in order to refer to regular n-tuples.

Relations are 'created' and tuples 'inserted'. Tuples can be 'read', 'scanned', 'updated' and 'deleted'. Relations can be 'dropped'; scanning tuples is done by scanning a relation -- or parts thereof.

Strings and tuples are formed by operations in the host language and inserted into classes, respectively regular relations through the proper use of GAMMA-0 commands. It is the responsibility of the host language user to set up the tuple in the format expected by the GAMMA-0 interface and according to the degree of the receiving class, respectively regular relation. If it is a regular relation and its degree is n, then you may, e.g., think of the host language user setting up a named 'vector' or 'structure' of (at most) n 32-bit words. A proper insertion of this named vector (i.e. tuple) into a

specified relation causes the system to compute and return to the user a distinct tid. This tid is unique within the relation, and over the entire lifespan of the relation. In these specifications we shall always name the host language vector which is later to become a tuple: tuple.

5.2 THE CREATION OF CLASSES and THE INSERTION OF BYTE STRING ITEMS

The user specifies the establishment of new classes, and the system formats such classes, identifies them and inserts a class relation descriptor tuple in MR. The user specifies such an insertion by providing the system with a partial 'descriptor' tuple for the class, C, to be created. The system finalizes the class relation descriptor tuple it is inserting in MR with the relation identifier, crid, for the class relation created; in addition the system sets up the proper template for a new class, i.e. the class identified by the crid is initially empty; its control tuple must always have the user specified encoded value: <null>.

Strings are named, uniquely classified and identified by the user insertion of a 1-tuple in some class C. The user provides the system with the string itself, some variable

field length byte string, and a class identifier. The system responds by inserting the string in the designated class, computing a unique iid for this string, the iid is then returned to the user. Uniqueness of the iid is to within the class, and over the entire lifespan of the class. Let such a string (e.g. in the HOST language) be denoted as: 'Washington'. The user can then insert this same string in several classes, one at a time, each time providing the same string, but a new (distinct) class identifier. In return, the user will receive as many (as far as the system, and thus also the user, is concerned) distinct iids, one per class. The user shall view this multiple insertion as effecting multiple copies of the string in the nR/D/B -- although, of course, at most one per class (relation). It may be that some of the encoded iid values for the multiple instances of e.g. 'Washington' are identical, then that is purely coincidental.

5.3 THE CREATION OF REGULAR RELATIONS

The user requests the establishment of any regular relation. The request specifies part of the regular relation descriptor tuple, and all of the control tuple, for the new relation. The input descriptor contains the specification of the 'degree', nR, of the relation, R, to

be created , and the 32-bit so-called 'primary key mask'. The system honors the request by (1) setting up a new, initially empty regular relation of the desired degree; (2) inserting the relation descriptor tuple in MR; (3) computing a unique tid for that tuple in MR; (4) computing a unique rid (=rrid) for the newly created regular relation, (5) inserting this rrid in the MR regular relation descriptor tuple, domain 7; and (6) returning this rrid to the user. The system (7) also set relation descriptor tuple items of domains 5 and 6 to null. Finally (8) the system accepts the user defined control tuple, and inserts this in the created relation as its control tuple; returning (9) the control tuple identifier, ctid, to the user and (10) also inserting this identifier in the relation descriptor tuple, domain 4.

5.4 TUPLE OPERATIONS AND THEIR RAMIFICATIONS

Thus tuples can be inserted. We can also delete entire tuples, and scan all or parts of tuples and update parts of tuples. We can finally get (=read) subtuples, and get the tuple id (only) of tuples, given the value of subtuples of these.

Insertion of any tuple is always specified to be

immediately after some other (already inserted) tuple. This latter tuple is specified through its known tid. Thus tuples of relations are ordered with respect to their so-called insertion order, see section 5.6.

For the system to delete a tuple in MR corresponds to the dropping of a whole, possibly, non-empty relation -- i.e. is the "dual" of creating an initially empty relation as accomplished through the system insertion of a tuple in MR.

Rewriting a tuple in the master relation is not permitted.

If the tid of a tuple, including the control tuple, in some given relation rid is known, then parts or all of the associated tuple can be read through a get.

No component of a primary key may have the encoded null value. Thus insertion of such a tuple is prevented, and update of parts or all of the primary key is always prevented.

Updating, at the GAMMA-0 level, is "in place', regardless of the manner in which update is implemented at a lower level. (Here the reader should take note of the

distinction between the "Hursley-" and the "Cambridge-" implementations of respectively the HP (binary relational data base-) and the RAM systems).

All inversions associated with a relation are automatically updated to reflect insertions, deletions and possible updates.

5.5 SCANS

There are two kinds of scans possible, one aided by the system through the user issued special scan commands, the other 'simulated' and maintained by the user through a sequence of alternating commands.

5.5.1 SYSTEM MONITORED SCANS

If the tids of tuples in some known relation rid are not known (but exist), then one can scan the relation. A scan can also be used to read all tuples in a given relation as of a given span of time. A scan is a sequence of next commands enclosed in a pair of opening create- and set-scan commands and a closing drop scan command. A next operation is much like a get, one difference being that the system provides a system defined, but to the user

apparently unknown, tid. A sequence of nexts causes the system to supply tuples having distinct tids. The scan sequence of tuples follows that of their insertion order, see sec. 5.6. A scan can be specified to commence after any tuple, provided this tuple's tid is known. Any create scan request causes the return to the user of the control tuple identifier; this allow the setting of a scan to commence with the top tuple of the relation and is useful also if no tuple identifiers are known. The set scan command further allows the user to specify that only tuples, portions of which satisfy certain item encoded 'filter' values, are retrived; for more on this, see below: Filtered Search. Once the last tuple of a relation have been scanned the system instead of returning some tuple returns an appropriate end-of-scan completion code.

****
*   *
* FIG.11 * See page 130.
*   *
****

Several scans can co-exist on the same relation. The order in which matching create- and drop-scans are issued is at the user's discretion, they need not be 'nested', but may partially overlap. Assuming single user traffic, then two scans on the same relation -- beginning with the same tuple and with no other commands inserting, updating or deleting tuples therein -- will produce exactly the same sequence of tuples from that relation.

See Page 131.

5.5.2 USER MONITORED SCANS

Unknown tuple identifiers of partially known tuples can be retrieved through the use of a command, tuple id from subtuple, which specifies the 'filter' (i.e. the subtuple) item values. The tid returned is that of the first tuple whose domain values equals those of the input subtuple. The system 'search' for this tuple (id) commences with the tuple following one whose tid the user can specify. If the user specifies the control tuple tid the search commences with the top tuple. If the user specifies tid value null, then likewise. The search ends in either of two ways, either with the first tuple having a subtuple satisfying the equality criterion, or after the last, the bottom, tuple if no tuples were found which could satisfy equality.

Given the tid, the user can then get the suple denoted. And following this, the user can get the tid of the next tuple after this, satisfying the same (or changed) filter values. Thus, by alternating between (first) 'get tuple id from subtuple' and 'get subtuple (from tid)', with the

user properly furnishing the values of successive tids, the user can set up the so-called user-controlled scan.

Whenever in this document we refer to a scan, we shall mean a system monitored scan.

5.5.3 FILTERED SEARCH

The 'get tuple id from subtuple' and the 'set scan' commands both permit the user to specify that only certain tuples be retrived, namely those for which portions (i.e. subtuples) 'satisfy' the input subtuple (i.e. the 'filter') item values. This filter is provided by these commands and effect the outcome of the 'tuple id from subtuple' command directly, and the 'next' commands controlled by the 'set scan' command. The 'satisfaction' criteria are as follows: ((0) If the input subtuple, the 'filter', in some domain specifies the encoded item value: null, then the tuple to be selected must have the very same item value: null in this domain.) (1) If the filter specifies '*' for some domain, then the selected tuple may have any value (including null and '*'_ in this domain. (2) if the filter specifies other than (null or) '*', the selected tuple must have an encoded item value in the corresponding domain equalling exactly taht of the filter. (Criterion (0) is properly included in criterion

(2)).

Specifying a filter with only '*' encoded item values will bring out the tuple (id) immediately following that of the user specified starting tid. Specifying nothing but '*' encoded values may thus be redundant, in that just one suffices, and that one may be associated with any domain.

5.6 TUPLE ORDERING

Tuples within any relation are ordered. Ordering of regular- and class- relations is accomplished, i.e. is (partially) defined, through the use of the 'insert tuple' command, and the meaning of ordering is (then completely) defined with respect to the scan of two or more tuples of a given relation. The next two sections will then deal with this insertion- and scan- order. Thereafter we shall cover the ordering of tuples of the master relation, and of all inversions.

5.6.1 INSERT and MOVE ORDER

Let T_{i1}, T_{i2}, ..., T_{iq} be all the tuples of some relation, R_i, as of a given instance of time. The control tuple is not included in this list. Let these tuples be

ordered in the sequence just listed. Let the tuple identifier of T$\underline{i}$ be tid$\underline{i}$, where we make no assumption as to the 'ordering' of the tuple identifiers when viewed as 32-bit unsigned integers. Now a 'new' tuple, T$\underline{k}$, may be inserted in either of three 'places' in relation R$\underline{i}$; (1) 'in front' of T$\underline{i1}$, also called 'at the, or on top' of R$\underline{i}$; (2) 'at the bottom' of R$\underline{i}$, i.e. 'after' the 'last' tuple (T$\underline{iq}$); or (3) 'in between' adjacent tuples, say T$\underline{ii}$ and T$\underline{i(i+1)}$. To insert a tuple in front of T$\underline{i1}$ will be viewed as inserting it 'after' the control tuple associated with R$\underline{i}$; this control tuple has the tuple identifier: ctid$\underline{i}$. The three cases of insertion can therefore be accomplished if the insertion command always specifies which tuple the new tuple is to be inserted after, and this already existing tuple can be referred to by its tuple identifier, tid'; which for the three cases above becomes: ctid$\underline{i}$, tid$\underline{ii}$ and tid$\underline{iq}$ (the latter also by: $\underline{null}$, (default)).

If a tuple, T$\underline{k}$, was inserted after T$\underline{ii}$, and T$\underline{ii}$ is subsequently deleted, before either of T$\underline{i(i-1)}$ and T$\underline{k}$ are deleted, then T$\underline{k}$ will now 'follow' T$\underline{i(i-1)}$, i.e. had T$\underline{ii}$ been deleted before T$\underline{k}$ was inserted, inserting T$\underline{k}$ after T$\underline{i(i-1)}$ would have left the same effect -- provided 'everything else' is unchanged! If j=1 then T$\underline{i(i-1)}$ is the control tuple.

The updating of a tuple, T$\underline{i}$i, does not change its tid and shall not change its insert order with respect to (previously) adjacent tuples, i.e. 'update' is always "in place".

A string of insertions of tuples: T$\underline{k1}$, T$\underline{k2}$, ..., T$\underline{km}$ -- and in that order, all directed to be after the same, already inserted, tuple, T$\underline{i}$i, results in the following (reversed) ordering of T$\underline{kn}$, for n = 1,2,...,m: ..., T$\underline{i}$i, T$\underline{km}$, T$\underline{k(m-1)}$, ..., T$\underline{k2}$, T$\underline{k1}$, ...

If the direct order of T$\underline{kn}$ is desired, then one should direct insertion after tuple with identifier: tid$\underline{i(n-1)}$ which for n=1 is tid$\underline{i}$i, and for all other n up to m is the tid returned from the immediately preceeding insertion:

COMMAND INPUT:	OUTPUT:
1: insert T$\underline{k1}$ after tid$\underline{i}$i	tid$\underline{k1}$
2: insert T$\underline{k2}$ after tid$\underline{k1}$	tid$\underline{k2}$
3: insert ...	
.	
.	
.	
m-1: insert T$\underline{k(m-1)}$ after tid$\underline{k(m-2)}$	tid$\underline{k(m-1)}$
m: insert T$\underline{km}$ after tid$\underline{k(m-1)}$	tid$\underline{km}$

MOVE:

Through the use of the move tuple command, a tuple can be moved to become positioned immediately after some other tuple, specified through its tuple id. Thus if before we had the order: Tii, Ti(i+1), ..., Ti(k-1), Tik, Ti(k+1), and if we specify the move of Tik to now follow Tii, then we will hereafter have the order: Tii, Tik, Ti(i+1), ..., Ti(k-1), Ti(k+1), ...

5.6.2 SCAN ORDER

Let Ti1, Ti2, ..., Tiq (q=0,1,...,Q) be all the tuples of relation Ri, not including the Ri control tuple, cTi. Let the tuples of Ri be ordered as listed above, Ti1 being the 'first' tuple (on 'top') of Ri. A scan of Ri can be specified to 'start after' any tuple, including (after) the control tuple. Let tidij be the tuple identifier for Tij, and ctidi the control tuple identifier.

Then beginning a scan after tidii will 'in some sense' initially set the associated scan identifier (sid) to 'denote' Ti(i+1), where if tidij = ctidi, this tuple, Ti(i+1) is Ti1, and if j=q, then there is (presently) no such tuple, i.e. Ti(i+1) is undefined. Now let the

'create scan' command be followed by a sequence of next commands on (that relation and) that scan identifier; such that no other GAMMA-0 commands intervene with the next sequence of commands. The system will then produce a sequence of tuples which starts with T$i(j+1)$ and in which, if Tik is scanned, i.e. returned to the user, then T$i(k-1)$ was the immediately preceding scanned tuple (for $k = j+2, j+3, \ldots, q$).

We now lift the restriction that only next commands may appear between a pair of enclosing ('sid'-corresponding) create- and drop-scan commands. We do so in order to point out some properties of the GAMMA-0 system.

If, while a scan identifier denotes Tik, i.e. Tik is the tuple to be returned in response to a next command on this sid, a tuple Tij is deleted, then three possibilities exist.

(A) If Tij precedes Tik, the subsequent next commands will not be affected, they will produce the same sequence of tuples as if Tij had not been deleted.

(B) If Tij properly succeeds Tik, then a subsequent series of next commands bringing the scan all the way past T$i(j+1)$, will not produce Tij, as it otherwise would have done had not Tij been deleted.

(C) If Tij is Tik, then the immediately subsequent next command will not produce Tik (=Tij) but instead, provided Tik was not the 'last' tuple of Ri, Ti(k+1) will be produced. In the case Tik was the last tuple, no tuple will be produced, but an end-of-scan completion code will be set.

It is thus we see that the scan identifier is immediately effected only in the latter case (C). It is also for this reason that the end-of-scan completion code is set (at time t") only when (at that time) the first next is issued after a previous next (at time, e.g. t') produced the bottom tuple.

Similar case-studies can be made with respect to the insertion and updating of tuples. (D) An insertion or the updating of a tuple, Tij, properly preceeding Tik, will not affect the order, nor the value of subsequently scanned tuples. (E) The insertion or the updating of a tuple properly succeeding Tik will affect the scan, but (provided everything else is unchanged) Tik will still be the immediately next scanned tuple. (F) If we update Tik before it is scanned, the scan will yield the updated tuple. (G) If we insert a tuple, Tij, 'after' Ti(k-1), then a subsequent scan shall still produce Tik, not Tij!

The purpose of these example-experiments is to lead up to the statement that "a scan is only as good as the context in which it is being pursued"; and hence: "if one desires a scan' of the tuples existing as of the very time instance the 'create scan command was issued, then one had better prevent (certain) insertions and deletions in the relation to be scanned". If one further wishes not just to get the tuples as of a given time, but also with the values they had at that time, then one had better also prevent certain updates. The adjective 'certain' should indicate that as long as these commands take place with respect to tuples already scanned, or more precisely: properly preceeding the one tuple to be scanned next, then the ongoing and subsequently continuing scan will not be affected.

The 'preventions' alluded to above, will in a multi-thread-user system, amount to some kind of scope control of- and lock on parts or all of the given relation(s).

5.6.3 MASTER RELATION ORDERING

Tuples of the master relation are insert ordered with respect only to the time ordered sequence in which the corresponding relations (regular-, class- and inversions)

were created. Through the use of the move command these relation descriptor tuples can then be reordered.

5.6.4 INVERSION ORDERING

Tuples of inversions are ordered with respect to their immediate 32-bit item encodings -- with ordering on domain one as major, domain two as minor.

5.7 TUPLE DOMAIN SELECTION

We may up till now have given the reader the impression that when tuples were scanned, read or their tid searched for, that all items, i.e. the entire tuple, had to be involved. We now repair this impression by describing the concept of the partial selection of tuple items.

The insertion, update, reading (through get) and the scanning of tuples are operations that need not involve the entire n-tuple. The update tuple need not involve all of the non-primary key domains. In order therefore to indicate to the GAMMA-0 system the domains of interest, the GAMMA-0 command is qualified by a '*domain identifier selection list*', *dlist*.

Now it is possible to update only subparts of the non-primary key portion of a tuple, leaving unspecified parts unchanged (with respect to their past 'values') -- as long as we are not violating usual update rules. And it is likewise that we are not required to receive a tuple involving all n items as a result of get's or next's, but can narrow down our desire by properly selecting the domains of interest.

5.8 STATES

We shall informally define three separate notions of 'states' of the GAMMA-0 system:

(1) DATA STRUCTURE STATE SPACE

At any one time the system supports a number of relations, and within these a number of tuples. The collection of all these relations, e.g. as described in the master relation, and the values of all their tuples, together form the composite 'data structure state' of the GAMMA-0 universe of discourse.

Whenever a new relation is created, or an old dropped, or whenever tuples are inserted, updated or deleted, this state changes, and only then.

(2) IDENTIFIER STATE SPACE

(See chapter 3.) Similarly: at any one time the system maintains a space of identifiers. This identifier space can be viewed two ways (1): the denotation of an identifier is that for which it was generated. Thus an rid denotes the whole, content-wise dynamically changing, relation of which it is an identifier, a tid (in conjunction with a suitable rid) some n-tuple, an iid (together with a suitable crid) a byte string item, and an sid the scan of a relation. (2): The coding of an identifier is a 32-bit word, its value does not change once generated, but it will cease to exist with respect to the GAMMA-0 system identifier space once its denotation has been deleted, and as a 32-bit word of some encoding it shall never again be re-generated. When we speak of the state of the identifier space we mean the structured collection of (1) all the presently GAMMA-0 generated 32-bit quantities that codes identifiers; together with (2) now defunct, but once existing, identifiers.

Whenever a new relation is created or generated, or an old dropped, or whenever tuples are inserted or deleted, this state changes. The state also changes

whenever a new scan is created, or whenever an existing scan is dropped; it does not change as a result of ongoing scans, due to the scans themselves.

(3) SCAN STATE RELATION -- RELATION USAGE STATE

At any one time a relation has associated with it, zero, one or more scans. With each scan is associated a scan identifier. Each scan identifier, sid, denotes the scan; and its denotation involves, amongst other matters, which tuple (tid) is 'next', which selection (dlist) of domains of this tuple is to be affected, and according to which 'filter' the scan is to be pursued. The state of the usage of the relation is a composition of all sids associated with that relation and their denotation. We shall conceptually model this concept of a state as a relation, the so-called 'scan state relation'. The actual content of the scan state relation then specifies which present state the aggregate scan is now in. The actual content of the scanned relation is not part of this notion of state. The relation usage state is always affected by the create scan, the drop scan, the next, set scan and the delete relation commands. It is never affected by the create relation, generate inversion, drop inversion, tuple id from first tuple, tuple from tuple id and update tuple commands. It may be

affected by the delete tuple and the insert tuple commands.

In this and the next paragraph we assume familiarity with the 'create-' and 'set scan' commands, see sections 8.9 and 8.10. With each scannable relation we can therefore associate an initially empty 'scan state relation', ssr. If the degree of the scanned relation is 'n' then the degree of the associated scan state relation can be set to e.g. 2*n+1. Each tuple of the ssr will denote one of the scans currently enacted on the associated relation. Domain 1 of the ssr denotes tids; domain 2 through n+1 either null's or dids; and domain n+2 through 2*n+1 32-bit encoded filter values. The tid is that of either the set scan command, or that of a tuple just read in response to a next. The did values of the tuple correspond to the did of the domain to which they belong. If null they are ignored.

The create command inserts a tuple in the scan state relation. The create scan command initializes values of domains 2 through n+1 and prepares those domains of the range n+2 through 2*n+1 which are to receive (other than *) filter values. The set scan command then finalizes the values of domains n+2 through 2*n+1 and domain 1. The command is thus an update

type command with respect to the scan state relation. The next command only updates the domain 1 value, and only if the tid does not denote the bottom tuple of the scanned relation. The drop scan command corresponds to the deletion of a tuple in the scan state relation.

Notice that the ssr is not a 'true and honest' GAMMA-0 relation. It has e.g. no primary key, and it may contain several identically coded tuples, however with distinct sids.

6. SINGLE-USER/MULTIPLE-THREAD-USER GAMMA INTERFACES

The present document explicates only those features of the GAMMA series of n-ary relational data base interfaces which are independent of whether a single-user (only) or multiple-thread-user implementation is sought; and independent of 'special' services being provided by the GAMMA system for the user other than a rudimentary set of operations.

This document has, however, already been checked against our plans for multiple-thread-user support facilities; and we rest somewhat assured that whenever such facilities are introduced, say: as extensions within the GAMMA-0 system, then they will not interfere with the present GAMMA-0 notions of data structures, identifiers and commands.

We shall therefore assume that the system below the GAMMA-0 interface is (1) logically a mono-processing system, insofar as the GAMMA-0 user is concerned, and (2) that if this system below GAMMA-0 is employing multi-tasking assists, then any such (invoked) multi-processing will not alter the semantics, Mg, of the GAMMA-0 interface.

7. ON GAMMA-0 COMMANDS

7.1 GENERAL

The GAMMA-0 set of commands can be partitioned into three sets:

(CAT1) Commands which create and drop relations,

(CAT2) commands which are provided for the insertion, reading, scanning and updating of- and search for tuple identifier of- tuples, and:

(CAT3) commands which are provided in order to enhance that elusive notion: system performance (with respect either to a set of related commands, as issued, say, by one user; or to all commands as seen from the system side of the interface). These latter commands have to do with inversions.

7.2 COMMAND FORMAT

Each command definition in GAMMA-0 takes the form of a

procedure. The operation denoted by an GAMMA-0 command is therefore invoked by a call to such a procedure. Each procedure call (1) accepts a list of input parameters, i.e. input values, (2) effects certain operations on the GAMMA-0 universe of discourse and (3) returns both (3.1) a set of output result values and (3.2) one of several possible completion codes.

The explication therefore of the next many commands will be templated according to the following format:

```
COMMAND NAME
SYNTACTIC FORM
    BASIC COMMAND DESCRIPTION
        INPUT PARAMETER/VALUE SET
        OUTPUT PARAMETER/VALUE SET
        EFFECT OF COMMAND INVOCATION
            GENERAL
            DATA STRUCTURE STATE CHANGE
            IDENTIFIER SPACE STATE CHANGE
        COMPLETION CODES
    REMARKS
```

The 'remarks' section comments on the possible spectrum of uses of the given command.

7.3 PARAMETER PASSING

In calling procedures, basically three rules of 'passing parameters' are possible: these are classically referred to as:

(1) call by value

(2) call by reference

(3) call by name

In section 8. we shall not assume any particular scheme whereby parameters are passed. The subject will however be dealt with in depth in a subsequent report.

7.4 LIST OF GAMMA-0 COMMANDS

Before proceeding into the detailing of individual commands we now present the list of CAT1, CAT2 and CAT3 commands:

GENERAL ABSTRACT SYNTACTIC COMMAND FORMAT:

OP <inputs> <outputs> <completion code>

where the latter will be understood, thus always omitted.

Command Names:	Inputs:	Outputs:
CAT1:		
1. create relation	<d-tuple,c-tuple>	<rid,ctid>
2. drop relation	<rid>	<>
CAT2:		
3. insert tuple	<rid,tid1,tuple>	<tid2>
4. delete tuple	<rid,tid>	<>
5. move tuple	<rid,tid1,tid2>	<>
6. update subtuple	<rid,tid,dlist,subtuple>	<>
7. tuple id from next tuple	<rid,tid1,dlist,subtuple>	<tid2>
8. subtuple from tuple id	<rid,tid,dlist>	<subtuple>
9. create scan	<rid,dlist1,dlist2>	<sid,ctid>
10. set scan	<sid,tid,subtuple>	<>
11. next subtuple	<sid>	<tid,subtuple>
12. drop scan	<sid>	<>
CAT3:		
13. generate inversion	<rid1,did>	<rid2,ctid>
14. drop inversion	<rid>	<>

7.5 COMMAND CASE STUDIES

As seen from the above list of commands there are the following input parameters:

rid, rid1, rid2
tid, tid1, tid2
tuple, subtuple
c-tuple
d-tuple
dlist1, dlist2
sid
did

For each command a subset of these are required. Their values must conform to what the command procedure expects to receive -- but because of the late 'binding' of these procedure calls, the called procedures are set up to check themselves that proper formatting of the individual parameters is the case. If -- what the GAMMA-0 user believes to be -- a GAMMA-0 command does not even meet the syntactical formatting of the commands, they are rejected before even passed on to the individual procedures; this rejection corresponds to the completion code "syntax check". Once they have entered the appropriate procedure it will check the actual formatting of the individual parameters and in many cases also their

values. It expects to find these within a narrow set of possibilities, but must be prepared for all kind of 'rubbish'. In our exposition of the individual commands we shall therefore have to detail all these cases, correct as well as erroneous. We hope that our 'case' numbering will be found somewhat consistently applied throughout.

8. GAMMA-0 COMMAND SPECIFICATIONS

8.1 CREATE RELATION

CR <d-tuple,c-tuple> <rid,ctid> <ccc>

This command is used to establish either regular- or class- relations. The type, degree and primary key mask of this relation is specified in d-tuple, and all of the control tuple of the new relation is specified by c-tuple; both of which are furnished by the user. The system now establishes the desired relation conditionally upon the validity of d-tuple and c-tuple. In return the user is provided with the relation identifier, rid, of the new relation and the control tuple identifier, ctid, of its control tuple.

INPUTS:

d-tuple

Must be of degree 3, otherwise error case (6) arises. Item 1 denote the type of the relation to be scannned; the user specifies either: "REGULAR" (2) or "CLASS" (3), otherwise error

case (7) arises. Item 2 the degree, an integer valued positive number, of the relation; if the type specifies 'class', then degree must specify degree=1, otherwise error case (8) arises. For 'regular' relations, degree must be at least 1, otherwise the same error case (8) is in effect. Item 3 denotes the primary key mask for the relation to be created. If the degree of the relation is N, and N is less than 32, then there can be no '1's beyond bit position N, otherwise error case (9) arises; regardless of relation type, the primary key mask must specify at least one '1' bit in some valid position, otherwsie error case (10) arises; if the relation is of type 'class', then the primary key mask must specify a '1' only in position one, all other positions specifying '0's, otherwise error case (11) occurs.

c-tuple

Must be of degree N, as specified by the second d-tuple item, otherwise error case (12) occurs. Its item values must all be specified by the user, and must be either a relation identifier, athe relation type code RID, or be null. Only these types of entries are allowed in the

c-tuple of any regular relation; otherwise error case (13) occurs. If the relation to be formed is a class relation, then the only c-tuple item must have the encoded value: null; otherwise error case (13') arises.

OUTPUTS:

rid

If cases (2) or (3) still prevail, then this output parameter will come to specify the rid of the newly created relation, otherwise its encoded value will be null.

ctid

If cases (2) or (3) still prevail, then this output parameter will specify the system generated control tuple identifier for the c-tuple now part of the newly established relation, otherwise its encoded value will be null.

EFFECTS:

The system establishes an initially empty relation

of the desired degree and class, and with the specified primary key mask -- only if cases (2) or (3) still prevail; in all other cases only the completion code is a meaningfull return value. The system inserts the specified control tuple in connection with the relation and generates a ctid for this tuple, as well as generating an rid for the (entire, empty) relation. Both these two latter identifiers are entered together with the three items of the d-tuple in respective domains as a relation descriptor 7-tuple in the master relation, and by the system.

FIG.13 See Page 132.

COMPLETION CODES:

(a) degree of d-tuple not 3, (6)

(b) type not 'class' or 'regular', (7)

(c) degree of class relation not 1, (8)

(d) degree of regular relation must be larger than 0, (8')

(e) '1' bits beyond bit position N, degree of relation less than 32, (9)

(f) all zero primary key mask not permitted, (10)

(g) class relation must specify a '1' bit in position one only, (11)

(h) degree of c-tuple incompatible with degree of relation, (12)

(i) item values of control tuple not of correct type, (13)

(j) item value of control tuple must be <u>null</u> for class relations, (13')

8.2 DROP RELATION

DR <rid> <> <ccc>

This command is used to drop a regular- or a class-relation, rid. If this relation is the parent of a number of inversions, then all these are also dropped.

INPUTS:

rid

rid must specify the identifier of either a class (3) or a regular (2) relation, otherwise error case (5) arises; this command cannot be used to drop inversions, nor can anyone drop the master relation.

OUTPUTS:

N/A

EFFECTS:

If cases (2) or (3) prevail, then the system will

unconditionally drop the entire relation, delete its descriptor tuple in the master relation -- and if this dropped relation has any inversions on it, then all these will also be dropped together with their descriptor tuples in the master relation. Any inversions on the master relation will be updated to reflect the deleted tuples of the master relation.

****
*   *
* FIG.14 * See Page 133.
*   *
****

COMPLETION CODES:

(a) No dropping of other than regular- or class-relations, (5)

8.3 INSERT TUPLE

IT <rid,tid1,tuple> <tid2> <ccc>

This command is used to populate a relation with tuples. rid denote such a relation; tid1 the tuple after which the insertion is to take place, tuple denote all of the eventually inserted tuple; tid2 is the system generated identifier of the inserted tuple. If rid denotes a class relation, then tid2 is an iid. Insertions are only permitted into class- and regular-relations.

INPUTS:

rid

There are five cases: (1) rid = mrid, (2) rid (=rrid) denotes a regular relation, (3) rid (=crid) denotes a class relation, (4) rid (=irid) denotes an inversion and (5) rid is not known to the system at all. No tuples can be inserted into the master-, nor any inversions; thus cases (1), (4) and (5) are all erroneous.

tid

Two cases are possible. Either tid is known by the system as a tuple identifier for the relation denoted by rid, or it is not, in which case situation (6) arises. All tuples of a relation, including its control tuple, have unique tuple identifiers which are (said to be) known to the system.

tuple

For cases (2) and (3) the degree of tuple must conform to the degree of the receiving relation, i.e. must be n respectively 1, where n is the degree of the receiving regular- or class-relation; if not we have case (7), otherwise the above cases prevail.

OUTPUTS:

tid2

For the successful completion of cases (2) tid=tid2 and (3) iid=tid2 will be returned with a meaningful value, otherwise it is set to null. In the satisfactory cases it will either (2) denote the tuple identifier of the tuple

inserted in the regular relation rrid; or (3) the item identifier of the item inserted in the class relation crid.

EFFECTS:

(2) The systems now examines the tuple. If the receiving regular relation already has a tuple whose primary key values match those of the input tuple, then no insertion takes place, tid' denotes the tuple identifier of the already existing tuple, a suitable completion code is set and case (8) is in effect; otherwise the tuple is inserted and the system computes a new tid (=tid2) which is returned. The system also checks that no component of the supplied primary key has the null value, if there is a null value, case (9) arises, and no insertion takes place. If the receiving relation is the parent of some inversions, then these are updated according to the input tuple values.

(3) The tuple is unconditionally taken as a non-null N-byte string 1-tuple. If it is not already a member of the receiving class it is inserted, else we get case (8'), no insertion then takes place, but the iid returned is that of the already entered item. In the former case, (3) still prevailing, a new iid is computed and returned, and the item is properly inserted.

Both the data structure- and the identifier- states are affected, i.e. changed, for proper insertions in cases (2) and (3); in all other cases no state changes take place.

FIG.15.1	See Page 134.
FIG.15.2	See Page 135.
FIG.15.3	See Page 136.

COMPLETION CODES:

(a) no insertions in master relation, (1)

(b) no insertions in inversions, (4)

(c) rid not known, (5)

(d) tid1 not known, (6)

(e) degree of tuple incompatible with that of receiving relation, (7)

(f) ordinary tuple already in relation, (8)

(g) byte string item already in class, (8')

(h) primary key component value is null, no insertion, (9)

8.4 DELETE TUPLE

DT <rid,tid> <> <ccc>

This command serves the purpose of deleting a tuple from a relation. rid denotes the relation in question, tid some tuple in that relation. No deletion of tuples in either the master- or any inversion is allowed.

INPUTS:

rid

There are five cases: (1) rid=mrid, the master relation; (2) rid (=rrid) denotes a known regular relation; (3) rid (=crid) denotes a known class, (4) rid (=irid) denotes an inversion, or (5) rid is not known to the system as an rid.

tid

There are two subcases: either tid is a tuple identifier known to the system as belonging to the indicated relation for cases (2) and (3), or (6) it is not.

OUTPUTS:

N/A

EFFECTS:

Two cases will affect the state of the GAMMA-0 universe of discourse:

(2) and (3): The specified tuple, which is either an ordinary tuple, or a byte string 1-tuple, is unconditionally deleted from (both) the specified (rid) relation and from possible inversions associated with this rid relation.

Both the states of the data structure- and the identifier- part of the GAMMA-0 universe of discourse will be affected -- but only in the above two cases. The changes amount to some 'trimming' of the number of relations and/or tuples and similar 'trimming' of the affected identifier spaces.

In all other cases the system effects no actions on these states, but return with an appropriate completion code.

8.6 UPDATE SUBTUPLE

US <rrid,tid,dlist,subtuple> <> <ccc>

The idea behind this command is to update non-primary key components of an already existing tuple, tid, in a given regular relation, rrid. Such an update preserves the tid -- which may be referred to in many places (relations), thus ruling out the replacement of this command with the pair: delete and then insert, for which new tid's are always generated.

INPUTS:

rrid

rrid must (2) specify the rid of a regular relation, known to the system, otherwise we have case (1,3,4,5).

tid

tid must specify a tuple identifier of a tuple in the relation denotes by rrid, otherwise case (6) arises.

FIG.16.1	See Page 137.
FIG.16.2	See Page 138.

COMPLETION CODES:

(a) no deletions in MR, (1)

(b) no deletions in inversions, (4)

(c) rid not known to system, (5)

(d) tid not known to specified relation, (6)

subtuple

for case (2) above, the system inspects the tuple; otherwise it is rejected. The degree of tuple must in case (2) be less than or equal to n, the degree of the receiving regular relation. If not, case (7) arises.

dlist

The dlist is inspected only in case (2): it must have the same degree as subtuple, or have degree 0 in which case dlist is taken to denote the ordered list of domain identifiers from 1 to n inclusive -- and in which case subtuple must have degree n also. Otherwise case (8) arises. The dlist, when not empty, but otherwise of correct degree, has as items domain identifiers; their value must range in the interval 1 through n inclusive, if not, then we get case (9). Also, there must be no multiple occurrences of the same domain identifier, if so, case (10) comes about. None of the primary key domains must be updated, thus they must not be referred to in dlist, otherwise case (11) is in effect.

OUTPUTS:

N/A

EFFECTS:

Only in case (2) may the system perform actions on the GAMMA-0 data structures, thereby changing the associated state. In no case does it alter the 'state' of the identifier space.

The system will now replace components selected by dlist and of the tid specified tuple of rrid. The new values will be taken from the input tuple. Notice, that in no case will the tid be changed. The updating, when successful, will be automatically propagated to any appropriate inversion(s) on the specified parent relation -- i.e. to those inversions which were inverted on the domains selected by dlist.

****
*   *
* FIG.18 * See Page 140.
*   *
****

COMPLETION CODES:

(a) no updating of tuples in MR , class relations

or inversions, (1), (3) and (4)

(b) rid not an rrid known to the system, (5)

(c) tid not known in regular relation rrid, (6)

(d) incorrect subtuple degree, (7)

(e) dlist versus tuple degree incompatibility, (8)

(f) did values outside range, (9)

(g) duplicate did values, (10)

(h) dids of the primary key, (11)

8.9 CREATE SCAN

CS <rid,dlist1,dlist2> <sid,ctid> <ccc>

This command prepares for a reading of tuples of a relation. The command does not itself deliver any tuple(s) to the user. Thus a proper scan is actually the composition of this command with at least one 'set scan' and one or more 'next' commands (following it) and a terminating 'drop-scan' command. The GAMMA-0 system does not, however, impose this syntactic context restriction. A given instance of this command sets up a specific instance of a scan. Several scans may co-exist on the same relation, the system therefore returns an sid which the user uses to indicate back to the system which scan is involved in subsequent 'set scan', 'next' and 'drop-scan' commands. dlist1 is a list of domain identifiers, its purpose is -- during the scan -- to select only a subset of the components of scanned tuples for delivery to the user. dlist2 is also a list of domain identifiers; its purpose is to select that subset of the relation tuple domains to which any subsequent 'set scan' "filter" subtuple is to apply.

INPUTS:

rid

rid is (1) either the rid for MR, i.e. mrid; (2) or rid (=rrid) denotes a regular relation; or (3) rid (=crid) denotes a class relation; or (4) rid (=irid) denotes an inversion, or (5) rid is not known to the system as some rid.

dlist1 and dlist2

If either or both of the dlists are empty, then they denote the ordered list of domain identifiers 1-n, where n is the degree of the relation denoted by rid, cases (1), (2), (3) and (4). If a dlist is not the empty list, then its length must fall in the interval 1-n, inclusive; otherwise case (7) arises. If cases (1), (2), (3) or (4) prevail, then the domain identifiers of dlist are checked for their value: there must be no duplicates and all values must themselves fall in the closed interval 1-n; otherwise cases (8), respectively (9) comes about. For cases (1) and (2) the dlist may be other than the entire set of domain identifiers for the referenced relation, for case (3) it follows that dlist is just the 1-member list whose value must be

did=1, for case (4) it may be either null, did=1, did=2, did=(1,2) or did=(2,1).

For the meaning of dlist1 and dlist2 see EFFECTS below and the 'set scan' and 'next' commands. Briefly: dlist1 shall denote those components of the scanned tuples that the user wishes to be returned to him in response to a 'next' command. dlist2 denotes those domains of the scanned relation on which the search for matching subtuples shall be done; the actual 'filter' values shall be provided by the user through a 'set scan' command which thus must precede any 'next' command on this sid.

OUTPUTS:

sid

sid is a scan-identifier, it is provided by the system with then encoded value: NIL if a non-satisfactory command completion code was set, otherwise it is set to null. The sid is a 32-bit word whose bit encoding does not change from the time it is issued by the system in response to this command till the time its existence is terminated by the user issuing the

'drop-scan' command.

ctid

ctid is the control tuple identifier of the relation selected by rid.

EFFECTS:

In cases (1), (2), (3) or (4) a new sid is generated.

With the selected relation the GAMMA-0 system then associates a triplet: the newly established sid, the input dlist1 and the input dlist2. Any subsequent 'set scan' and 'next' command referring to this sid, will automatically also refer (implicitly, that is) to the (previously input) dlists.

There are no change of state of the GAMMA-0 data structure universe. The state of the GAMMA-0 identifier state is changed by the introduction of a new, system-wide unique, sid.

****
*   *
* FIG.21 * See Page 143.
*   *
****

COMPLETION CODES:

(a) rid not known to system, (5)

(c,d,e) dlist1 or dlist2 or both of degree incompatible with identified relation, (7.1 or 7.2 or 7.3)

(f,g,h) dlist1, or dlist2 or both, have did values outside proper range, (8.1, 8.2 or 8.3)

(i,j,k) duplicate dlist did values, (9.1, 9.2 or 9.3)

REMARKS:

At any one time several 'create scan' commands may be outstanding on any given relation, and originating with any given user. For each instance of the command being executed a new and distinctly encoded sid is returned. It may also be distinct with respect to its denotation, i.e. whatever tuple it points to. The denotation of an sid is independent of any other sids,

whether on the same or other relations. At a given point in time several such sids on the same relation may locate the same tuple due to the sequence of 'next' commands. These 'next' commands must therefore specify with which scan sid the 'next' is to be associated.

8.10 SET SCAN

SS <sid,tid,subtuple> <> <ccc>

With this command the user can specify 'after' which tuple, denoted by its tid, the sid scan for next subtuples is to commence; and the user can also specify that the search is to be performed on tuples whose components, as selected by the dlist2 of the opening 'create scan', matches those of the input "filter" subtuple.

INPUTS:

sid

sid must be the scan identifier for some relation currently being scanned (1), (2), (3) or (4); otherwise case (5) arises.

tid

tid must the tuple identifier of some tuple, including the control tuple, of the relation implied by the sid, or (6) tid may be encoded as null -- in which case it denotes the tid of the

bottom tuple of the relation --, otherwise case (7) arises.

subtuple

the degree of subtuple must match that of the degree of the dlist2 issued by that opening 'create scan' command which returned this sid; otherwise case (8) arises. The values of the components of the subtuple may be any, including null and '*'. The system will now 'update' its scan triplet: <sid,dlist1,dlist2> replacing the last parameter list with the pair: <dlist2,subtuple>.

OUTPUTS:

N/A

EFFECTS:

If cases (1), (2), (3) or (4) still prevails, then the system will properly update the denotation of the input sid, see section 5.6(3). All subsequent 'next subtuple' commands will be based on returning subtuples of tuples whose components in the domains selected by dlist2 matches those of the paired

input subtuple of this command.

There are no state changes to the data structure- nor to the identifier- spaces; the relation usage state, i.e. the scan state relation for the relation implied by sid is properly updated. This update may be in either of two ways. The previous state may have specified that the part associated with this sid had not yet been properly initialized, i.e. that only a 'create scan', but no 'set scan' had yet been executed -- or the state may have specified that, for this sid, there had been a previous 'set scan' successfully completed -- and therefore, in this latter case, that some partial scan had already, probably, taken place, but is being 're-directed' by this new 'set scan' command.

FIG.22.1
FIG.22.2

See Page 144.

See Page 145.

COMPLETION CODES:

(a) sid not known to system, (5)

(b) tid not known to relation denoted by sid, (7)

(c) degree of subtuple does not match that of the dlist2 associated with this sid.

(d) tid specifies <u>null</u>, no scan possible, (6)

8.11 NEXT SUBTUPLE

NS <sid> <tid,subtuple> <ccc>

This command presumes (sid) that some 'create scan' command has earlier been executed, and successfully completed on the relation implied by the sid. It also presumes that the denotation of this sid has been properly 'set' by a preceeding 'set scan' command. The command uses the sid returned by that 'create scan' command execution to indicate which instance of a scan on a given relation is desired. The effect of a successful execution of this command is that the user is given some subtuple and its identifier tid back. A sequence of 'next' commands on the same scan will then give distinct tuple as answers, until the relation has been completely scanned -- or until the user himself terminates the scan, whichever comes first. The sequence of tuples returned is that of the insertion ordering, see section 5.5. The actual subtuple delivered will have components from the tuple denoted by tid only from those domains selected by the dlist1 of the matching (sid) 'create scan' command. And the tuples denoted by the tid will have values in the domains selected by the matching 'create scan' command dlist2 equal to those of the most recently issued, and

successfully completed, 'set scan' subtuple input.

INPUTS:

sid

sid either specifies a valid, i.e. by the GAMMA-0 system currently maintained, scan identifier for the designated relation (1), (2), (3) or (4), or the specified sid is invalid with respect to the designated relation, (6).

OUTPUTS:

tid

If any of the cases (1), (2), (3) or (4) prevails, and if the sid does denote some tuple in the specified relation then tid is the identifier of this tuple. If instead cases (5), (6), (7) or (8) are in effect, then the returned tid is meaningless (it may then be set to NULL). (For cases (7) and (8), see below).

subtuple

As for tid above, in cases: (1), (2), (3) or

(4), the tuple returned is a sub-tuple of the tuple located by the scan. The system forms this subtuple as directed by the dlist1 system-associated with each sid, and user specified through the 'create scan' command which generated this sid.

EFFECTS:

If (7) the relation denoted by the input sid is empty, or (8) if the sid points to the 'next-after-last' tuple, then an appropriate completion code: end-of-scan is set, and no action is performed on neither the denotation of the sid nor the data structure, i.e. no meaningful tid nor subtuple is returned. If instead the sid does point to a tuple of the designated relation, then the tid and the proper sub-tuple of the tuple denoted by tid is returned to the user. The sub-tuple is assembled by the system according to the dlist, i.e. only those domains identified in this list will have items in the returned tuple.

There are no state changes. The relation usage state, or the (aggregate) scan state relation, see section 5.6 (3), is changed: The denotation of the sid shall now reflect the fact that the search for

a 'next subtuple' shall commence with the tuple presently following the one just returned, see section 5.5. This 'next tuple may be the next-after-the-last' tuple of the relation, in which case a subsequent 'next' command will set the 'end-of-scan' completion code and return no tuple.

```
**************
*            *
*   FIG.23   *    See Page 146.
*            *
**************
```

COMPLETION CODES:

(a) no such sid, (5), (6)

(b) end-of-scan, 'last' tuple already read, (8)

(c) empty relation, (7)

8.12 DROP SCAN

DS <sid> <> <ccc>

With this command a user can specify the proper termination of a scan, either as a result of all tuples having now been scanned, or (prematurely) before all such tuples have been scanned. This command is usually issued only in connection with an earlier execution of the matching opening 'create scan' command corresponding to the user input sid. Thus the command -- with the intentional usage as here described -- actually is part of a 'structured' command:

```
create scan <rid,dlist1,dlist2> <sid,ctid>
   set scan <sid,tid,subtuple> <>
   next subtuple <sid> <tid,subtuple>
   next subtuple <sid> <tid,subtuple>
    ...
    ...
   next subtuple <sid> <tid,subtuple>
drop scan <sid> <>
```

Note how the sid output parameter of the create scan command becomes the input parameter of all the

subsequent commands.

INPUTS:

sid

sid either is a scan identifier for some relation, (1), (2), (3) or (4), or (6) it is not.

OUTPUTS:

N/A

EFFECTS:

Only cases (1), (2), (3) and (4) are of interest: the effect of the command is now to foreclose on this sid, i.e. cease to make it available for scanning purposes. Thus we may say that the scan identifier space is decreased by exactly that member. Note that we were here speaking axiomatically, about the properties of sid's. In all other cases no state changes are effected.

****
*   *
* FIG.24 * See Page 147.
*   *
****

COMPLETION CODES:

(a) sid not known to the system, (6)

8.7 TUPLE ID FROM NEXT SUBTUPLE

TIFNS <rid,tid1,dlist,subtuple> <tid2> <ccc>

Given the value of a (dlist) subtuple of some tuple (believed to be) in some relation, rid, this command can be used to get the corresponding identifier, tid2, of the first (ful) tuple in the relation which matches the supllied subtuple on the domains selected by the domain identifier selection list, dlist. Knowing this tid2 can then be used to 'get' the entire tuple -- but in a separate operation. tid1 specifies the tuple 'after' which the system is to 'search' for the next subtuple. Through the judicious use of this command the user can effect his own scan of a relation; where in the system provided 'scan' commands, the system maintains the 'scan-pointer' the user here has to do this himself -- suitably by letting the subsequent TIFNS command input as tid1 the tid2 gotten from the previous TIFNS command.

INPUTS:

rid

rid either (1) is the mrid, or (2) an rrid, or

(3) a crid, or (4) an irid, or (5) is not known to the system as an rid.

subtuple

tuple must be of degree m, where m ranges between 1 and n, with n being the degree of the relation denoted by rid; otherwise case (6) arises. In case (3) tuple is a byte string item. Components of the input tuple are said to 'belong' to the domains of the rid as selected by dlist. Component values may be null in which case the search must look for a similar value in the 'next' tuple, or may be: /*/, in which case the search shall accept any value in the corresponding domain.

dlist

dlist is a list of domain identifiers. Its degree must match that of tuple, otherwise case (7) arises. There must be no duplicate values of did's in the list, otherwise case (8) arises. And all values of the did's must fall in the interval: 1-n, otherwise case (9) arises.

tidl

tid1 must be the identifier of a tuple known to the relation, this includes the ctid of the control tuple; otherwise case (10) arises.

OUTPUTS:

tid2

In the case of successfull command completion, i.e. (1), (2), (3) or (4) still prevailing, tid2 is the tuple identifier of the 'next' tuple after the one denoted by tid1 in relation rid, such that this tuple have values in the domains denoted by dlist matching those of the input subtuple.

EFFECTS:

The system now 'searches' the relation denoted by rid. The search starts with the 'next' tuple identified by tid1. The search is for the next tuple whose component values in the domains denoted by dlist matches those of the input subtuplele. These values are the 32-bit word codings. (i.e. they are not of what the components denote). If an input tuple component value is null then the searched tuple must have the corresponding searched

tuple component value must also be null. If instead it is: '*', then any corresponding item value of that domain in the searched tuples is permitted, null included. The search is sequenced according to the insertion ordering, i.e. behaves internally as would a scan of the relation. The search may have either of two outcomes, i.e. may end in either of two ways. Either a tuple is found satisfying the search criteria, or none is found, in the latter case the search stops when the last tuple of the relation has been 'compared'.

No state changes will occur as a result of this command being executed.

FIG.19 See Page 141.

COMPLETION CODES:

(a) rid not known, (5)

(b) incorrect degree of subtuple, (6)

(c) incompatible tuple/dlist degree, (7)

(d) did values of dlist outside range, (8)

(e) duplicate did values in dlist, (9)

(f) no such tuple found, (10)

8.8 SUBTUPLE FROM TUPLE IDENTIFIER

SFTI <rid,tid,dlist> <subtuple> <ccc>

With this command the user can retrieve a subtuple from a relation provided the system is supplied with the proper relation and specific tuple identifier, rid and tid. dlist selects components of the tuple denoted by tid, and only these are returned to the user.

INPUTS:

rid

rid either (1) is the mrid, or (2) is an rrid, or (3) is a crid, or (4) is an irid, or (5) is not known to the system as an rid.

tid

tid is either known as a tuple identifier of a tuple, including the control tuple, of the relation denoted by rid, or (6) it is not.

dlist

dlist is either an empty list in which case it denotes the ordered list of domain identifiers 1-n, where n is the degree of the identified relation. Or dlist is a list whose length falls in the interval 1-n, else case (7) arises. If dlist is of the proper length, then its values are checked: they must all fall in the same interval and there must be no duplicates, otherwise cases (8), respectively (9) arises.

OUTPUTS:

subtuple

In cases (1), (2), (3) or (4) the system returns a sub-tuple of the tuple identified by the rid,tid pair. The items of the sub-tuple are those selected from the (complete) n-tuple as specified by dlist. In all other cases the system returns no meaningfull tuple but sets an appropriate completion code.

EFFECTS:

In cases (1), (2), (3) or (4) the system does return a meaningful subtuple -- as directed by the input dlist -- and no state changes are made to the

relation- and tuple universe nor to the space of identifiers. In all other cases only a suitable completion code is returned.

FIG.20	See Page 142.

COMPLETION CODES:

(a) rid not known, (5)

(b) tid not known in relation, (6)

(c) degree of dlist is incorrect, (7)

(d) duplicate dlist values, (9)

(e) dlist values outside proper range, (8)

8.13 GENERATE INVERSION

GI <rid,did> <irid,ctid> <ccc>

The idea behind this command is, from a relation rid, to let the system generate and automatically maintain an inversion, irid; where the inversion takes place with respects to one domain only. Thus the user provides the identifier, rid, of the 'parent' relation to be inverted, and the domain identifier, did, for the domain with respect to which inversion is to take place; in return the user is provided with the relation identifier, irid, for the new relation.

INPUTS:

rid

rid either (1) is the mrid, or (2) is an rrid, or (3) is a crid, or (4) is an irid, or (5) is not known to the system as an rid.

did

Two, orthogonal, cases: did is within the range of domain identifiers for the specified

relation, or (6) it is not.

OUTPUTS:

irid

Only in cases (1), (2) and (3) may irid make sense. It then specifies the identifier for the inversion. It may either have been generated in direct response to this command instance, or have been already generated by some earlier such command, or the relation it now denotes may have been generated (internally) by the system (for reasons of its own optimization needs) and now made known to the (community of) user(s). Whatever the case, irid will be returned, whether old or new.

EFFECTS:

Cases (1), (2) and (3) may now go through in either of two ways: If the system already is maintaining an inversion on the given relation and domain, then we get cases (1.1), (2.1) and (3.1) respectively. If instead no such inversions exists then we get cases (1.2), (2.2) and (3.2) respectively. In both cases (.1) and (.2) the system returns the irid of

either (.1) the 'old', already existing inversion, or (.2) the newly established inversion. A suitable completion code will distinguish the two possibilities. In all other cases no inversion will be created and the returned irid is meaningless; say it's value is set to null.

Inverted relations are all binary. Domain 1 (did=1) consist of all the items of that domain (did) of the parent relation on which the inversion took place. There will be one entry for each tuple in the parent relation, thus there may be duplicate (valued) entries in this domain. Domain 2 (did=2) of the corresponding tuple identifiers.

The ordering of the tuple of the inversion is with respect to the 32-bit word encoding of first (major) domain 1 entries, then (minor) domain 2 entries.

Three possibilities exist, (A), (B) and (C). Either (A) the system maintains an inversion on this parent relation and the specified domain and this inversion has not earlier been requested created; in this case the system inserts an inversion descriptor tuple in MR for the previously privately maintained inversion, but now publicly known

inversion. Thus, if the system does maintain inversions, but these have not (yet) been made known, then they are not described by any tuple in MR. Or (B) the system does not, privately or publicly, maintain such an inversion; in this case the appropriate inversion is set up, filled with tuples extracted properly and inserted in the above described order, whereupon the corresponding inversion descriptor tuple is inserted in MR. Or (C) the system already publicly maintains such an inversion; in this case the corresponding inversion descriptor tuple already exists in MR.

In the first two cases, (A) and (B), the irid returned is new -- the identifier space as well as the data structure space is changed. In the last case, (C), the irid returned is that of the already existing relation. In case (A) the data structure space is augmented by the entire inversion previously privately maintained by the system, and is augmented by the inversion descriptor tuple inserted in MR. The identifier space is likewise augmented.

The future effect of a relation having possibly many inversions as its children is the following: First any insert, delete or update command must

work on the 'parent' relation, i.e. specify its rid, rather than some irid. Then if any of these commands are executed on rid, their effect on that relation is unconditionally 'propagated' to all currently maintained inversions for the given parent relation. Propagation effectively preserves ordering.

****
*   *
* FIG.25 * See Page 148.
*   *
****

COMPLETION CODES:

(a) no such rid, (5)

(b) no inversion on inversions, (4)

(c) did oustide range of denoted relation, (6)

8.14 DROP INVERSION

DI <irid> <> <ccc>

irid denotes the inversion to be dropped.

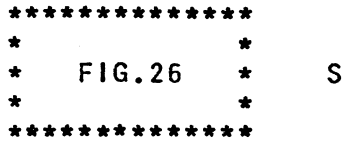
INPUTS:

irid

irid either (1) is the mrid, or (2) is an rrid, or (3) is a crid, or (4) is an irid, or is not known to the system as an rid. Only case (4) will lead to successfull actions.

EFFECTS:

If the system recognizes the irid as the rid of some inversion, then this inversion will be dropped, its entry in the MR will be deleted. The system may however decide itself to maintain (privately) this inversion -- for reasons of its 'own' (?) optimization. To the user, however, it will henceforth logically appear as if there was no inversion of the kind just dropped.



See Page 149.

COMPLETION CODES:

(a) irid not known as an irid, (1), (2), (3) and (5)

8.5 MOVE TUPLE

MV <rid,tid1,tid2> <> <ccc>

With this command the order of tuples of a relation, rid, can be re-arranged without interfering with the tuple value, nor its identifier, tid1, encoding. The future order will have the tuple identified by tid1 immediately following that of the tuple identified by tid2.

INPUTS:

rid

rid must denote either (1) the master-, (2) a regular- or (3) a class relation; no reordering can occur on tuples of inversions, (4). If rid is not known to the system as an rid, then error case (5) arises.

tid1 and tid2

Both these two identifiers must denote tuples known to the relation identified by rid, otherwise case (6) arises; case (5) takes

precedence over case (6). If tid1 and tid2 are identical, i.e. have the same encoded values, then no action is performed; it is considered an error case (7) in that no tuple can follow itself. If the tuple denoted by tid1 immediately follows that denoted by tid2, prior to the execution of this command, then again no 'cahnge' occurs, but the command is considered correct.

tid1

This identifier must specify a tuple other than the control tuple of the relation denoted by rid, otherwise case (8) arises. If the encoded value of tid1 is null its denotation is that of the bottom tuple of the relation.

tid2

This identifier may specify any tuple, including the control tuple of the relation denoted by rid. If the encoded value is null, its denotation is (again) that of the bottom tuple.

OUTPUTS:

N/A

EFFECTS:

If cases (1), (2) or (3) prevails, then the tuple denoted by tid1 is 'moved' so that its future order immediately follows that of the tuple denoted by tid2.

****
*   *
* FIG.17 * See Page 139.
*   *
****

COMPLETION CODES:

(a) no move of tuples of inversions, (4)

(b) rid not known to system, (5)

9c) tid1 (tid2) not known to relation, (6) ((6'))

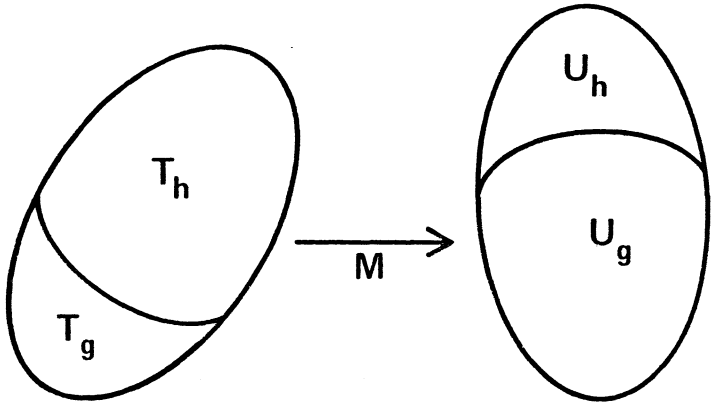
(d) tid1 may not be the ctid, (8)

(e) tid1 and tid2 are identical, (7)

9. CONCLUSION

We have explained and defined the simple set of GAMMA-0 data structure- and identifier objects and the lean, but not minimal set of GAMMA-0 operations defined on these. In a follow-on report we shall investigate some of the properties of desirable parameter passing schemes as well as proposing a GAMMA-0 identifier validation scheme. So far only the single-user aspects have been reported; subsequent reports will deal with the multi-user aspects of the GAMMA family of interfaces, in particular with GAMMA-1. We shall then report on schemes for the orderly, highly user-controllable sharing of data as well as aspects on temporary mutually-exclusive locking.

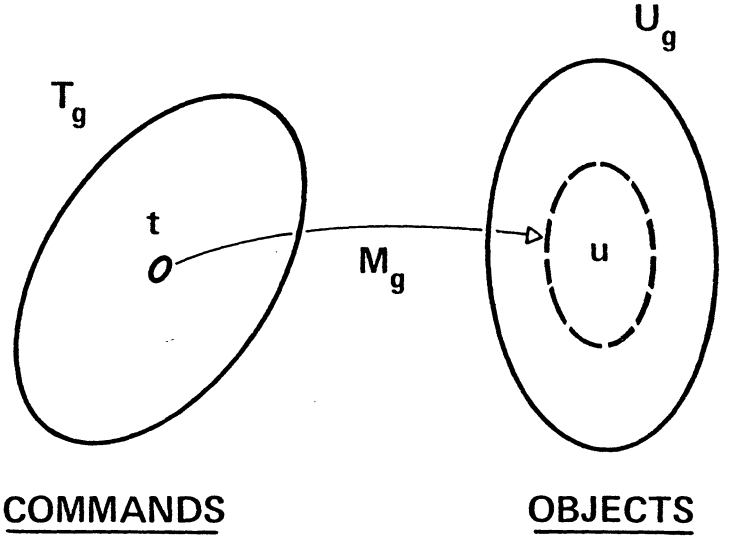
GAMMA-0



T : PROGRAMS U : UNIVERSE of DISCOURSE

$L = < T, M, U >$

FIG. 1



$$M_g\,(t) = u$$

FIG. 2

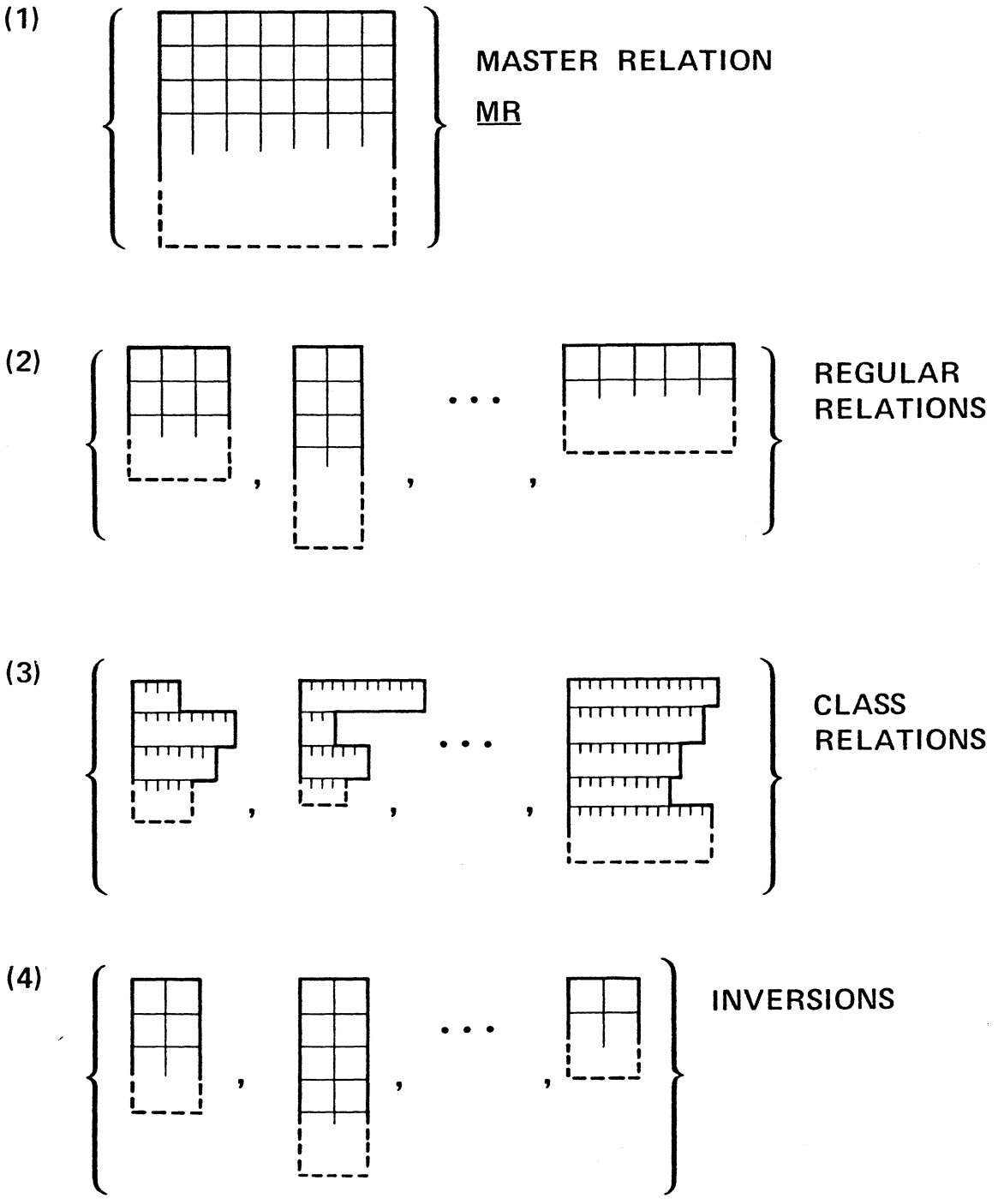
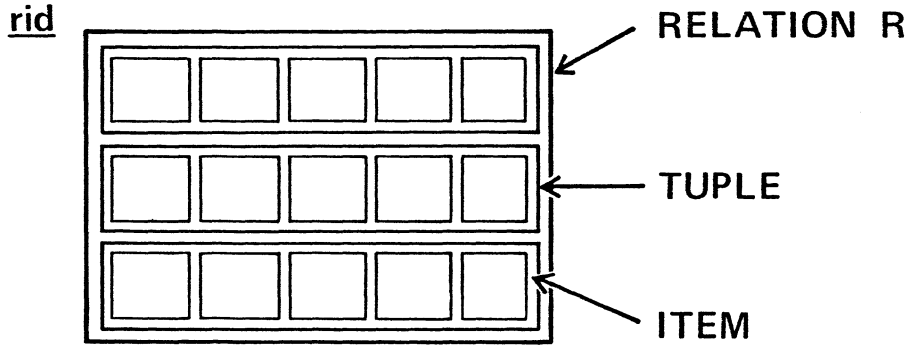


FIG. 3 RELATIONS



WILL HENCEFORTH BE DIAGRAMMED THUS:

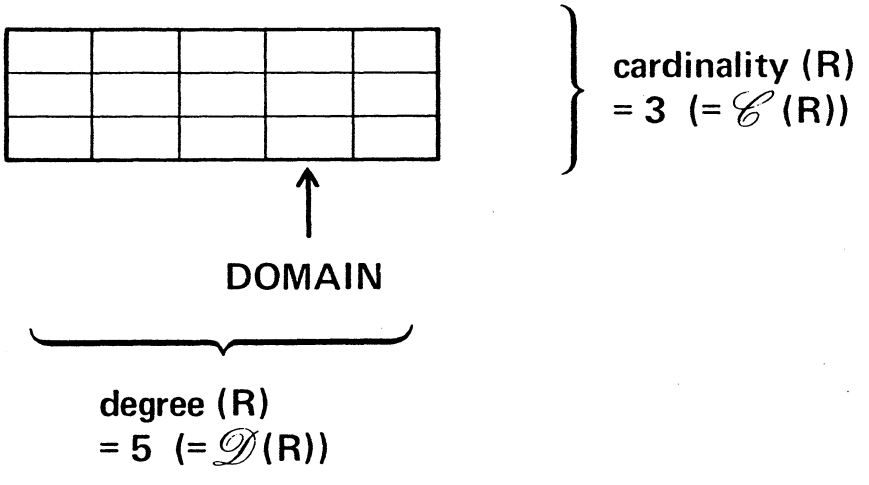
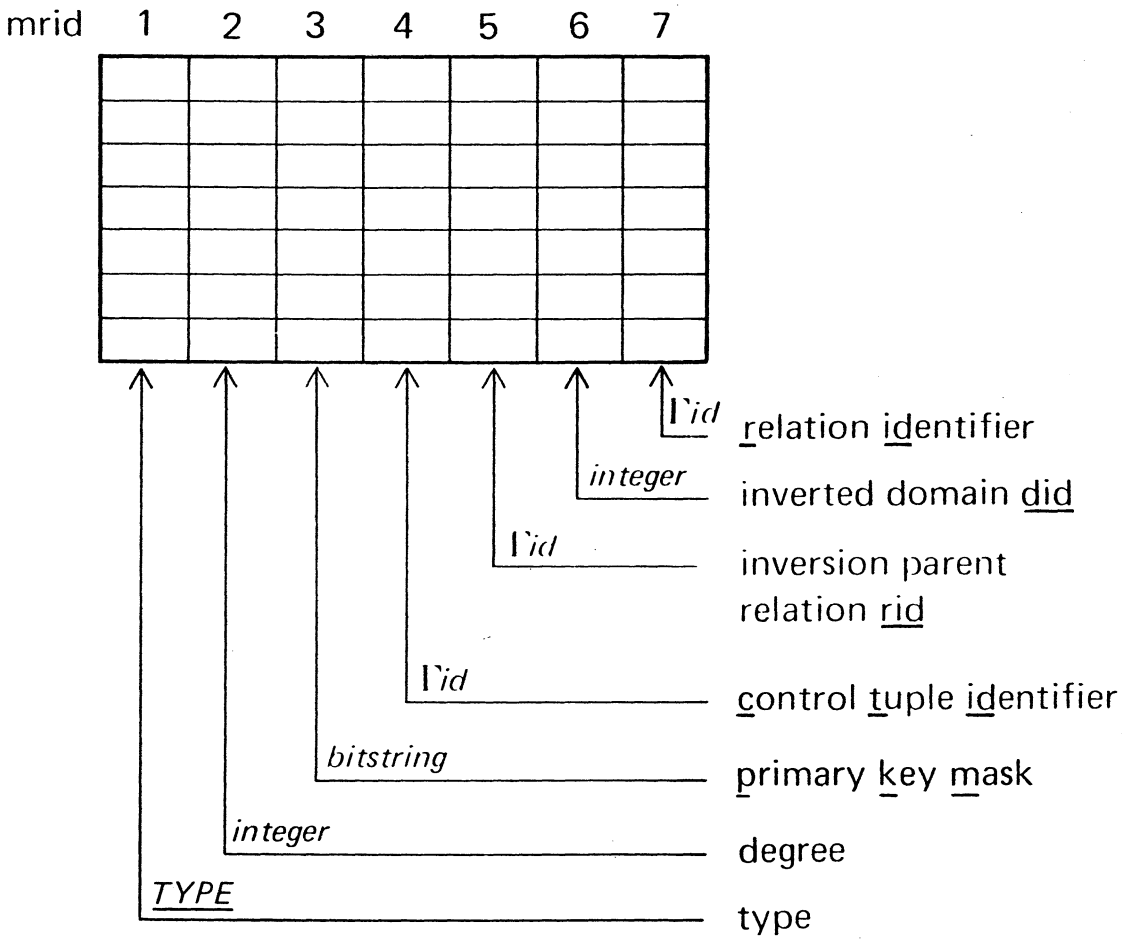


FIG. 4 EXAMPLE RELATION



TYPE = { MASTER, REGULAR, CLASS, INVERSION }

Γ ≡ *GAMMA-0*

FIG. 5 MASTER RELATION, MR

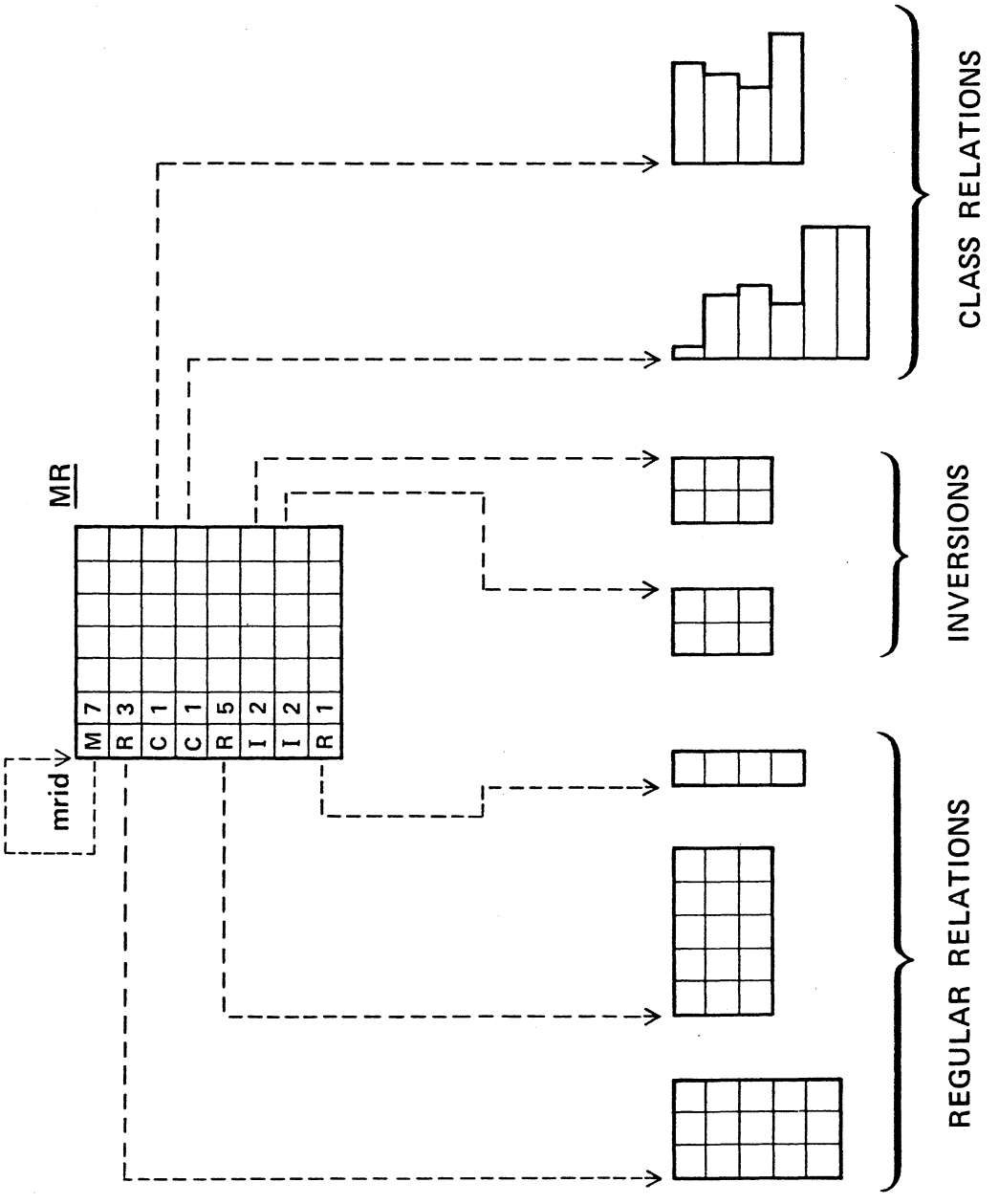
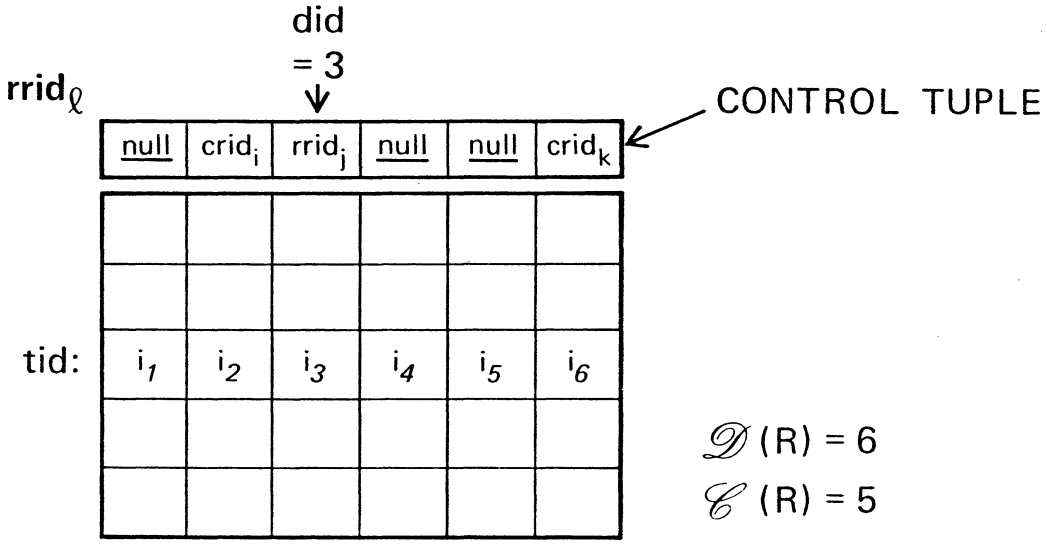

MR
M 7
R 3
C 1
C 1
R 5
I 2
I 2
R 1
mrid
CLASS RELATIONS
INVERSIONS
REGULAR RELATIONS

FIG. 6



---

i_1, i_4, i_5 : denote: themselves,

i_2 : items of CLASS RELATION crid$_i$

i_3 : tuples of REGULAR RELATION rrid$_j$

i_6 : items of CLASS RELATION crid$_k$

---

i_2, i_3, i_6 : are all GAMMA-O generated (system) identifiers (Γ_{id})

i_1, i_4, i_5 : are all USER generated

FIG. 7 REGULAR RELATION

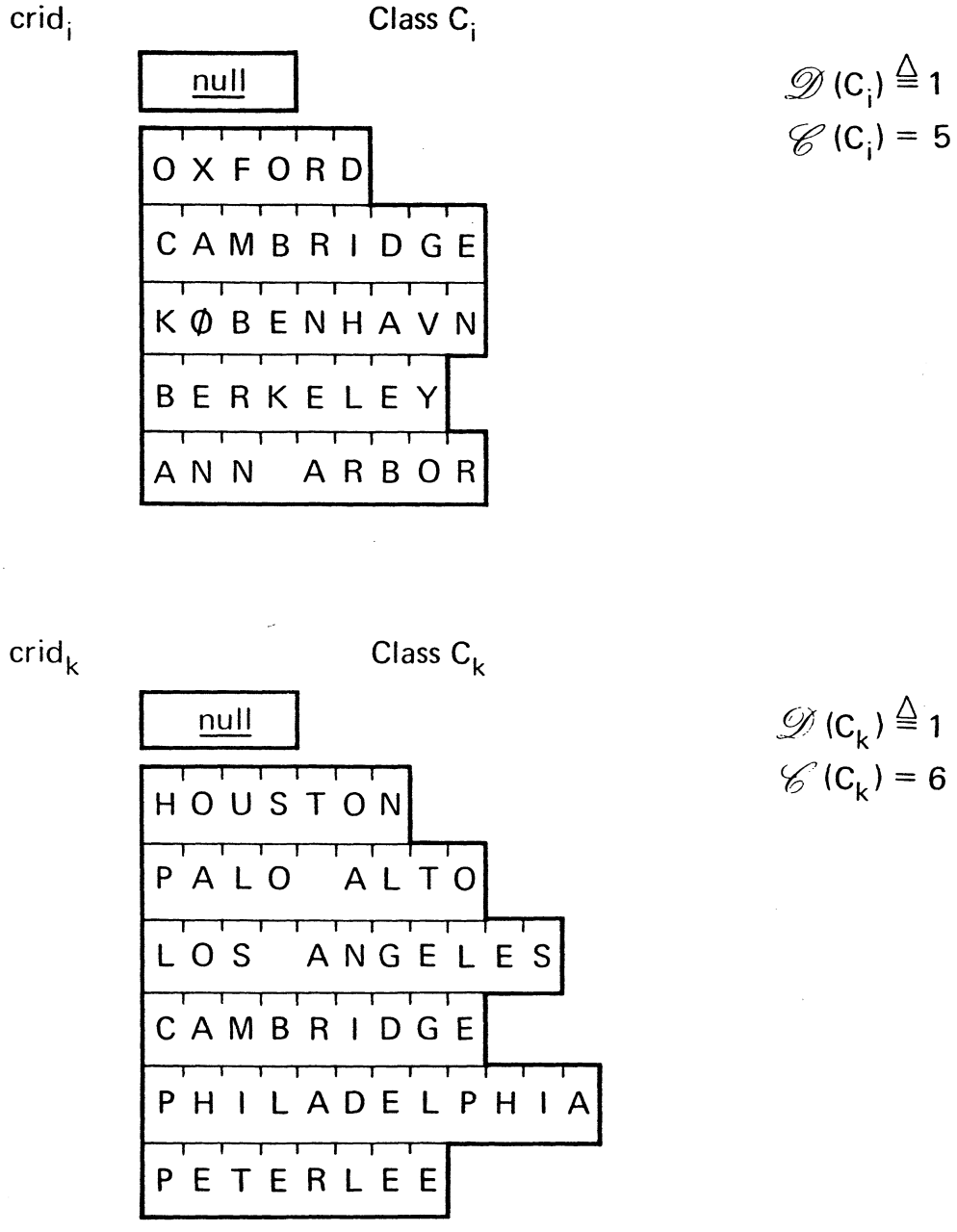
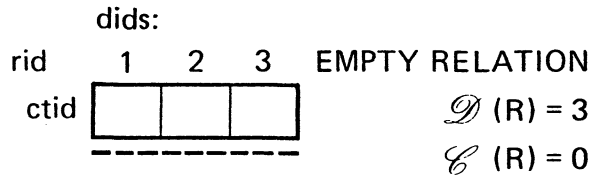


FIG. 8 CLASS RELATIONS



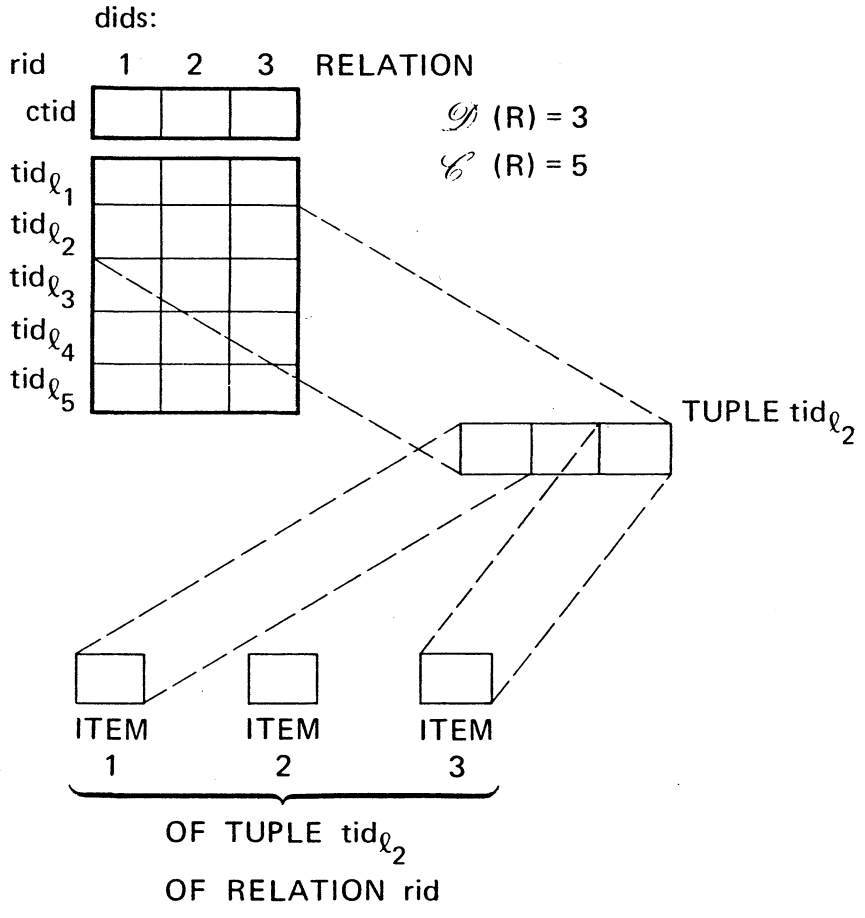
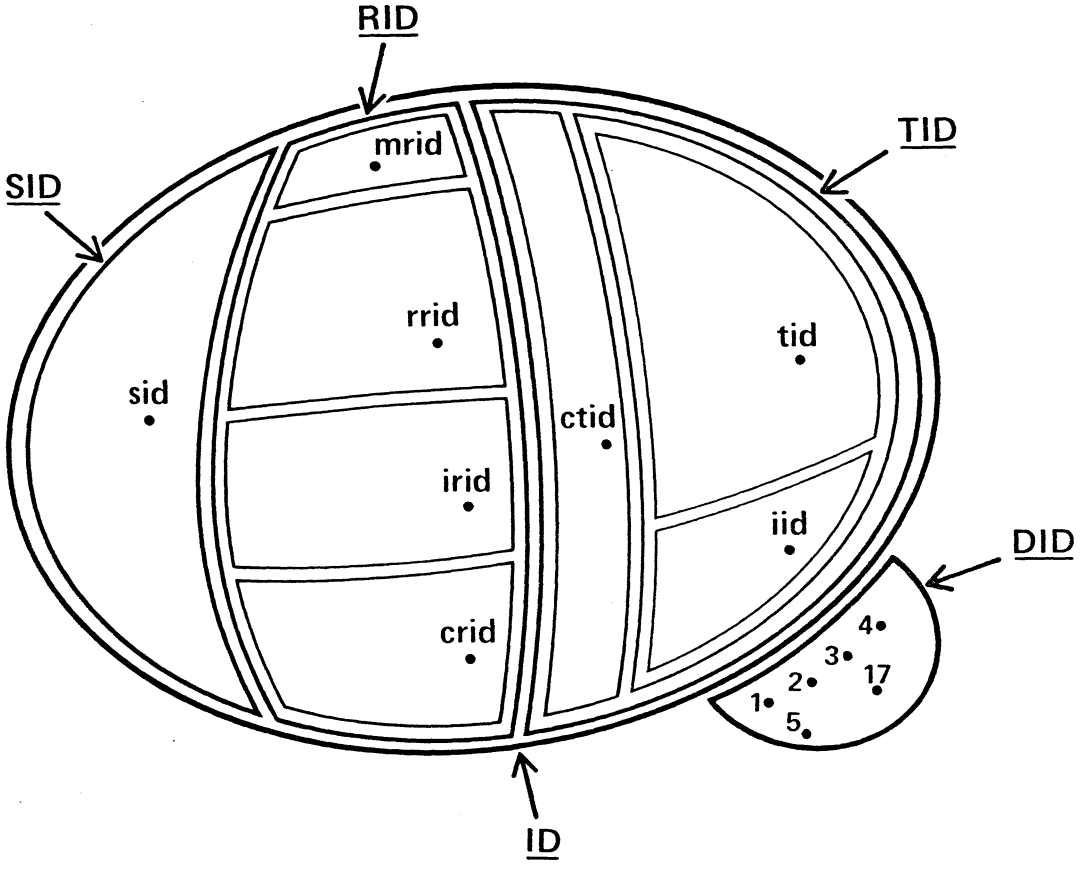


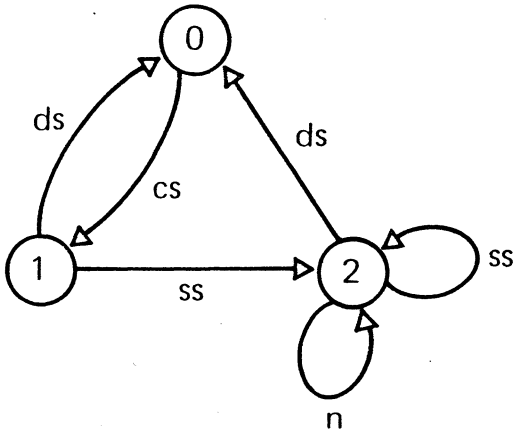
FIG. 9 REGULAR RELATION TUPLES



GAMMA-0 SYSTEM
(generated) IDENTIFIERS

FIG. 10

SINGLE SCAN

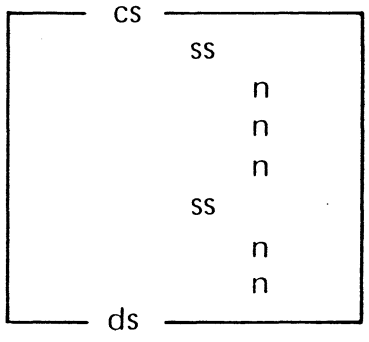


cs : create scan [rid, dlist1, dlist2] [sid, ctid]

ss : set scan [sid, tid, subtuple]

n : next [sid] [tid, subtuple]

ds : drop scan [sid]



example: SINGLE SCAN

FIG. 11 SCAN STATES

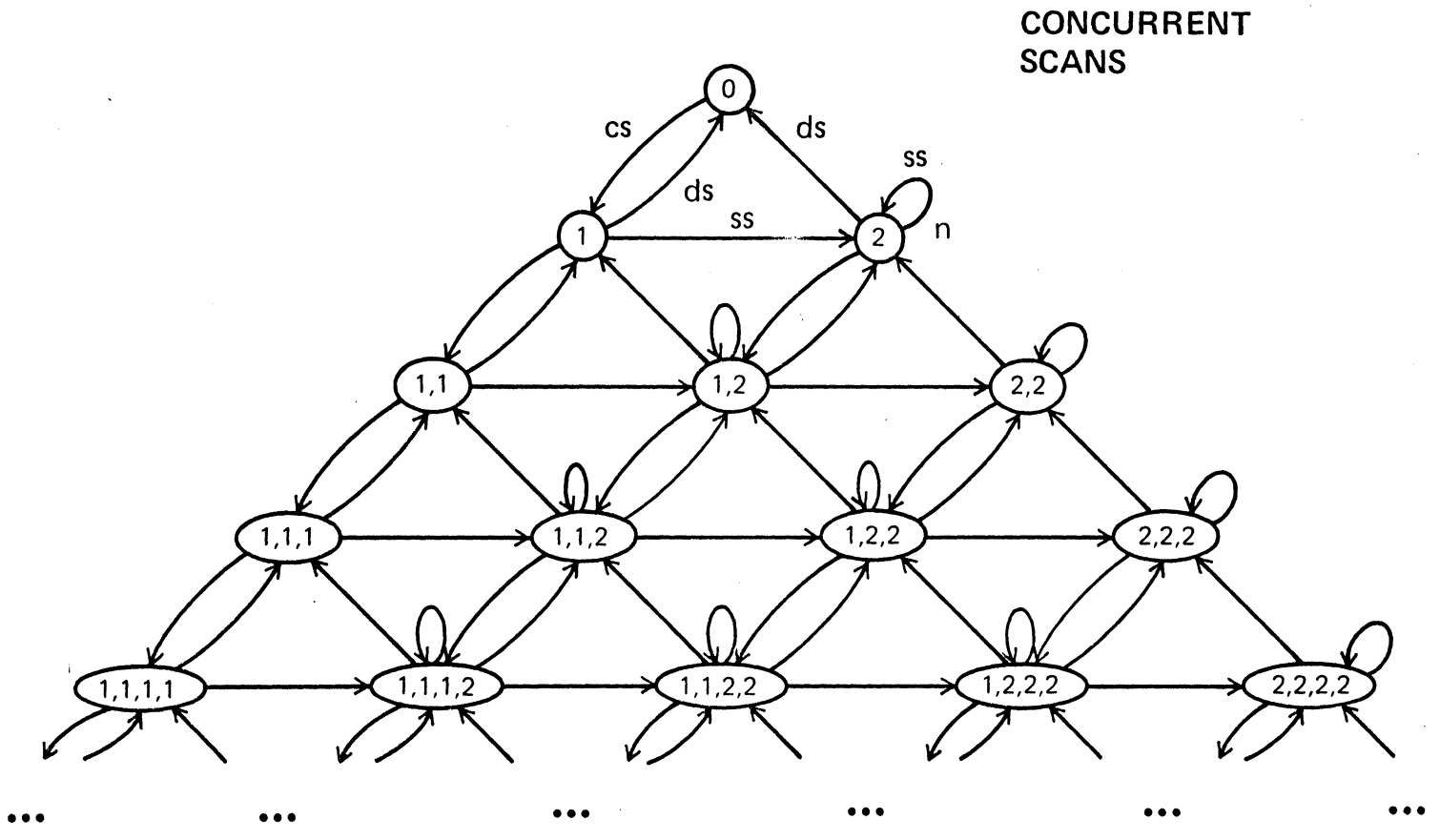


FIG. 12 SCAN STATES

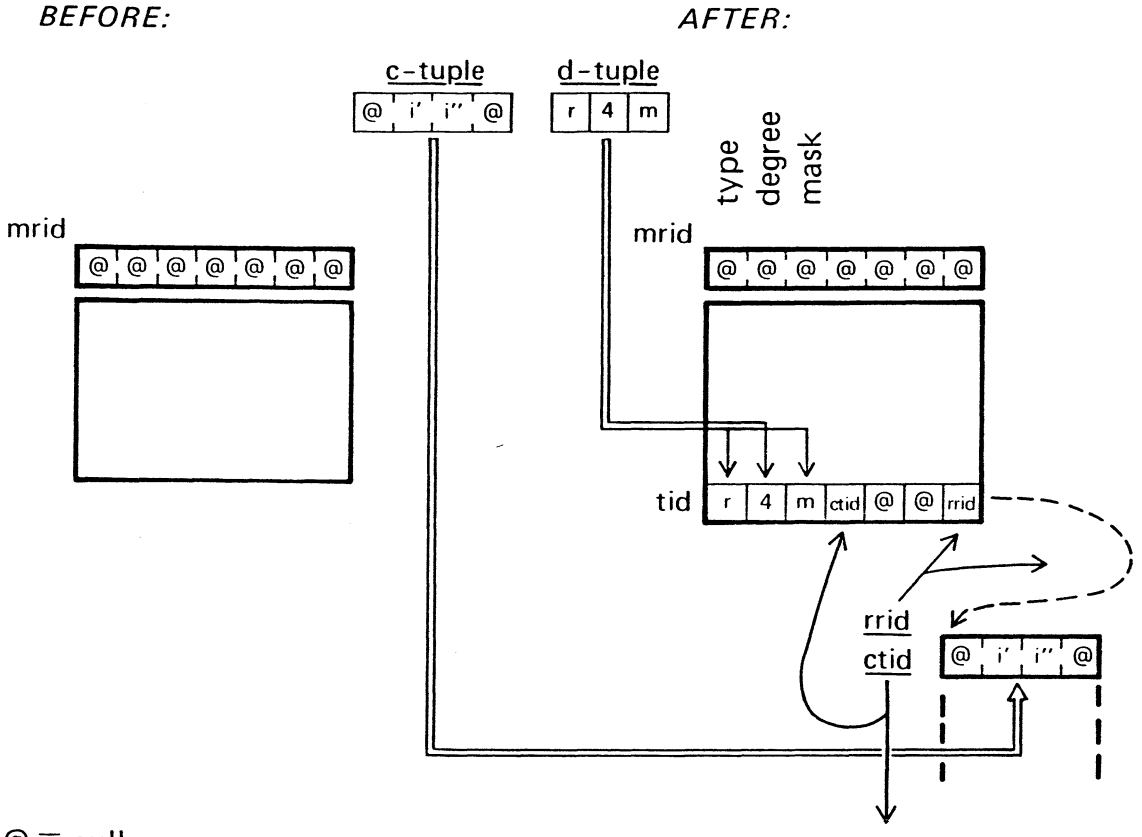


FIG. 13 CREATE RELATION

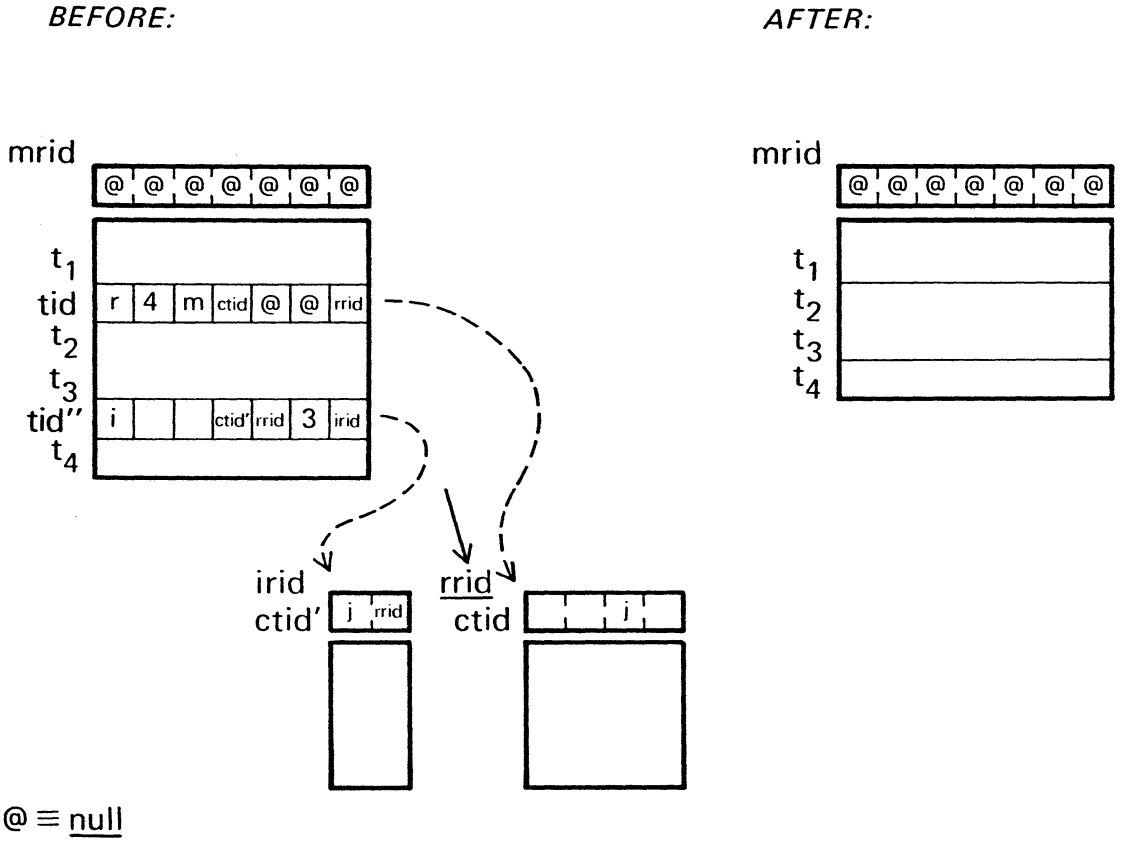


FIG. 14 DROP RELATION (example shows relation with one inversion)

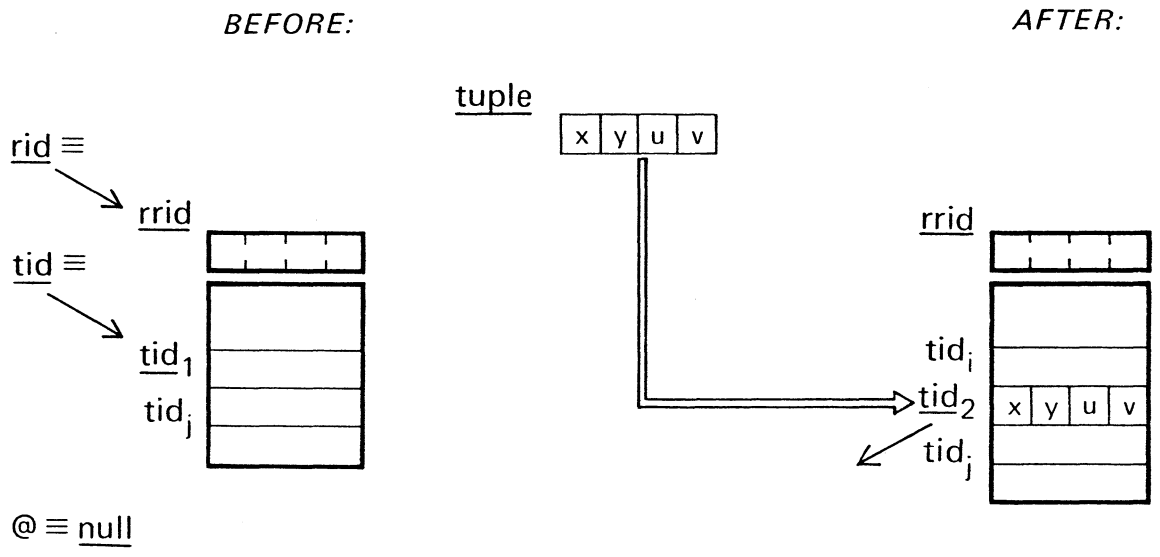


FIG. 15.1 INSERT TUPLE (no propagation/no inversions)

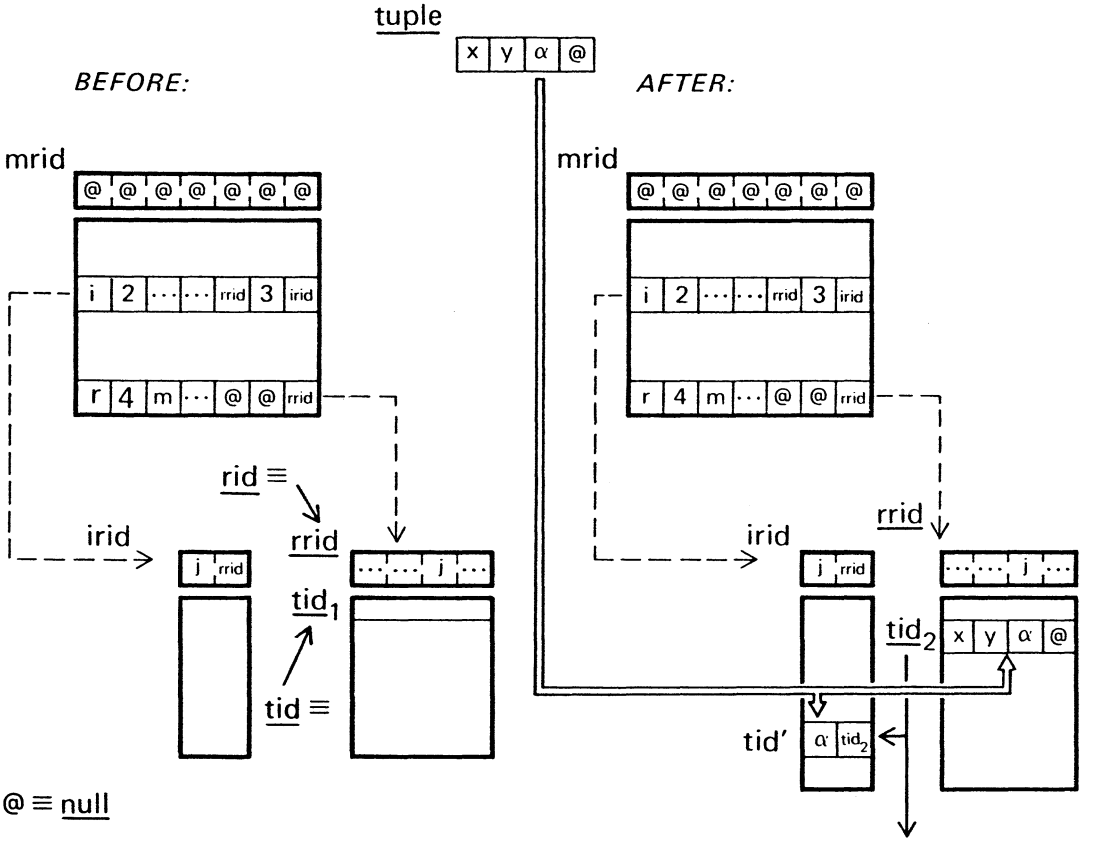


FIG. 15.2 INSERT TUPLE (propagation/inversion(s))

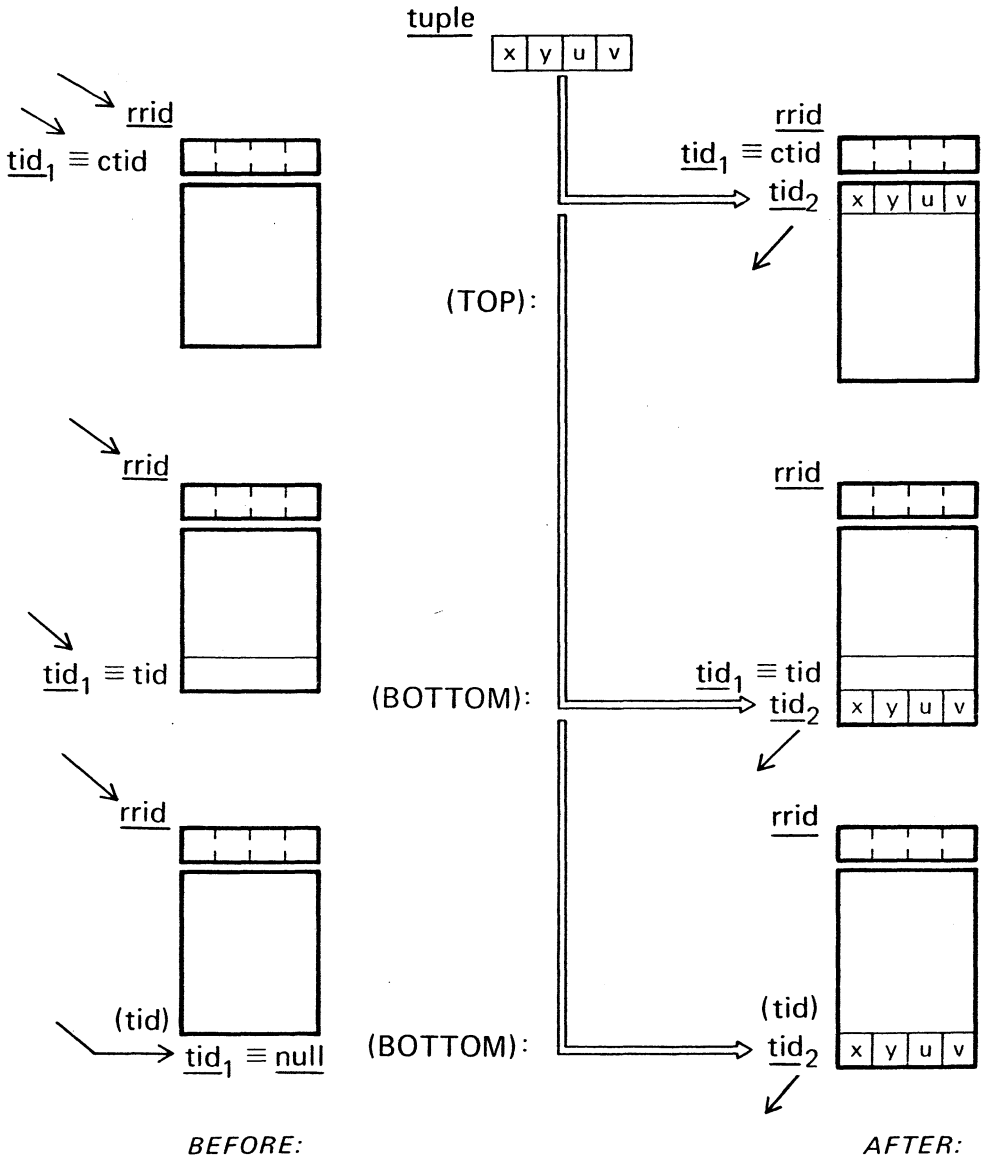


FIG. 15.3 INSERT TUPLE (variations in insert order)

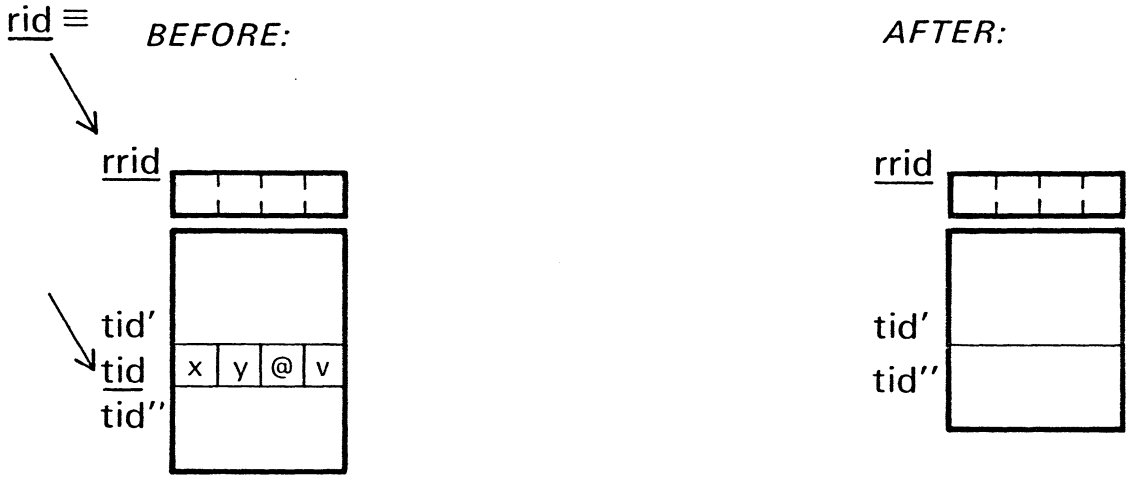


FIG. 16.1 DELETE TUPLE (no inversion/no propagation)

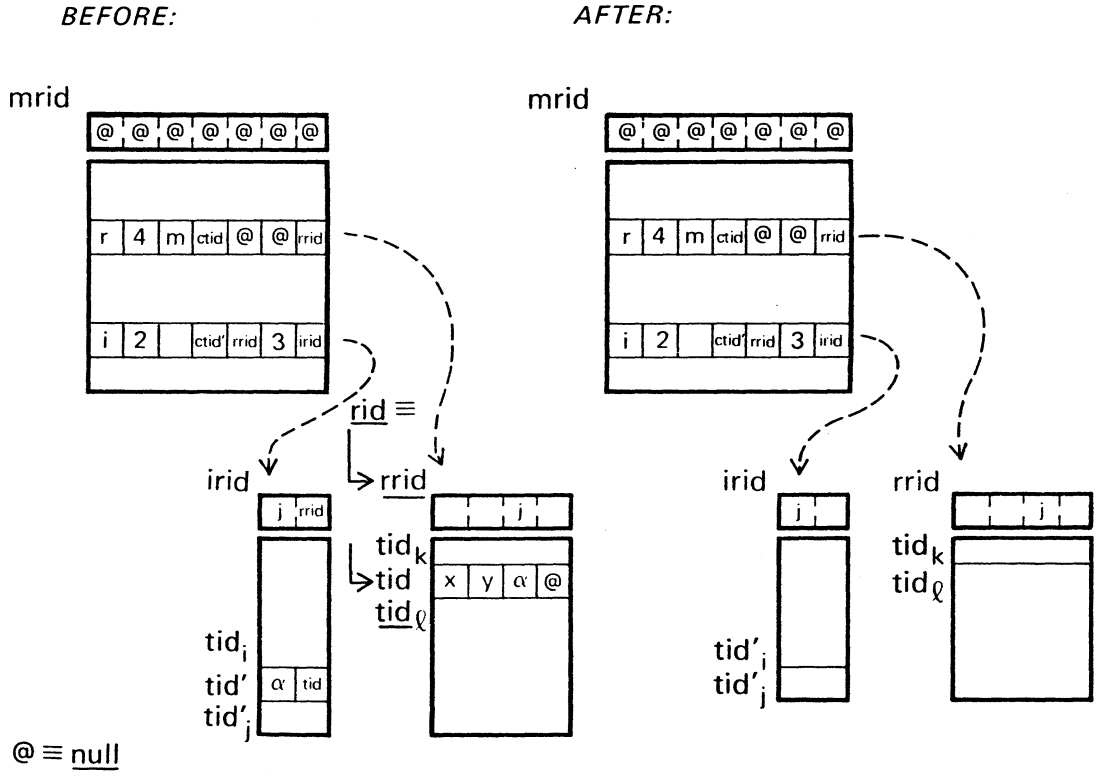


FIG. 16.2 DELETE TUPLE (inversion ⊃ : propagation)

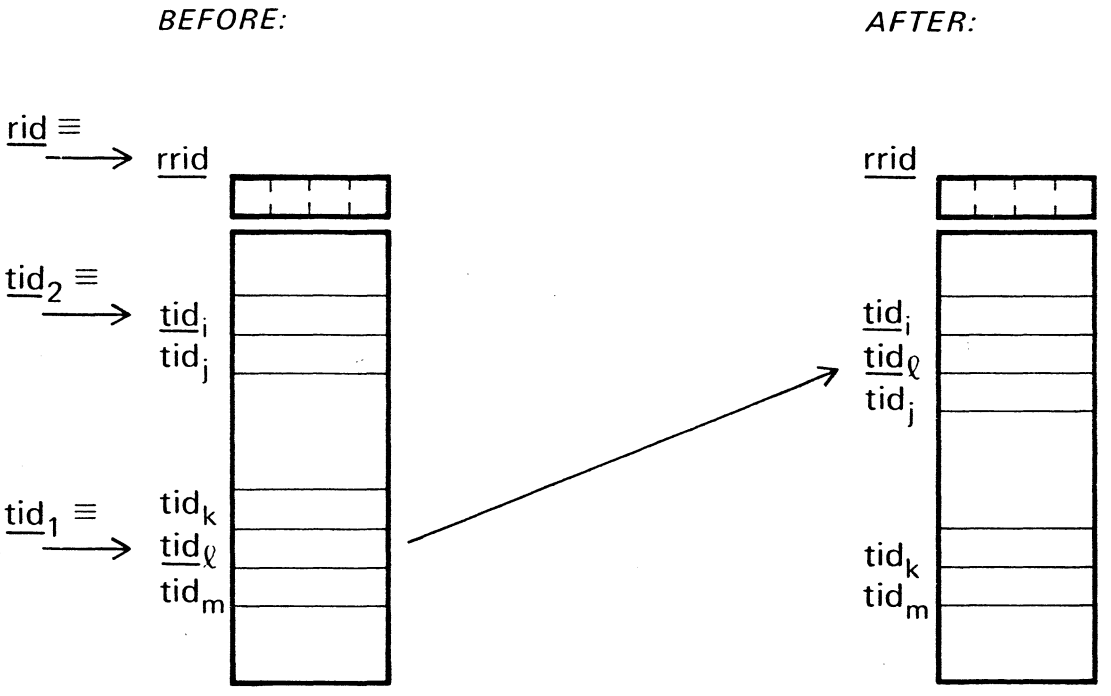


FIG. 17 MOVE TUPLE

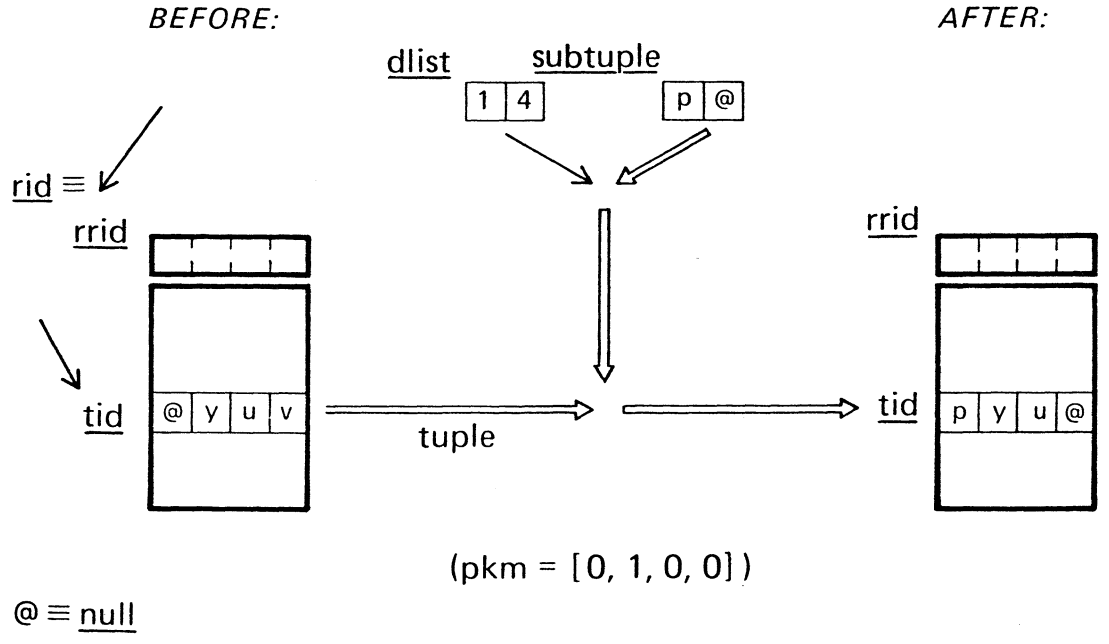


FIG. 18 UPDATE SUBTUPLE (no inversion ⊃: no propagation)

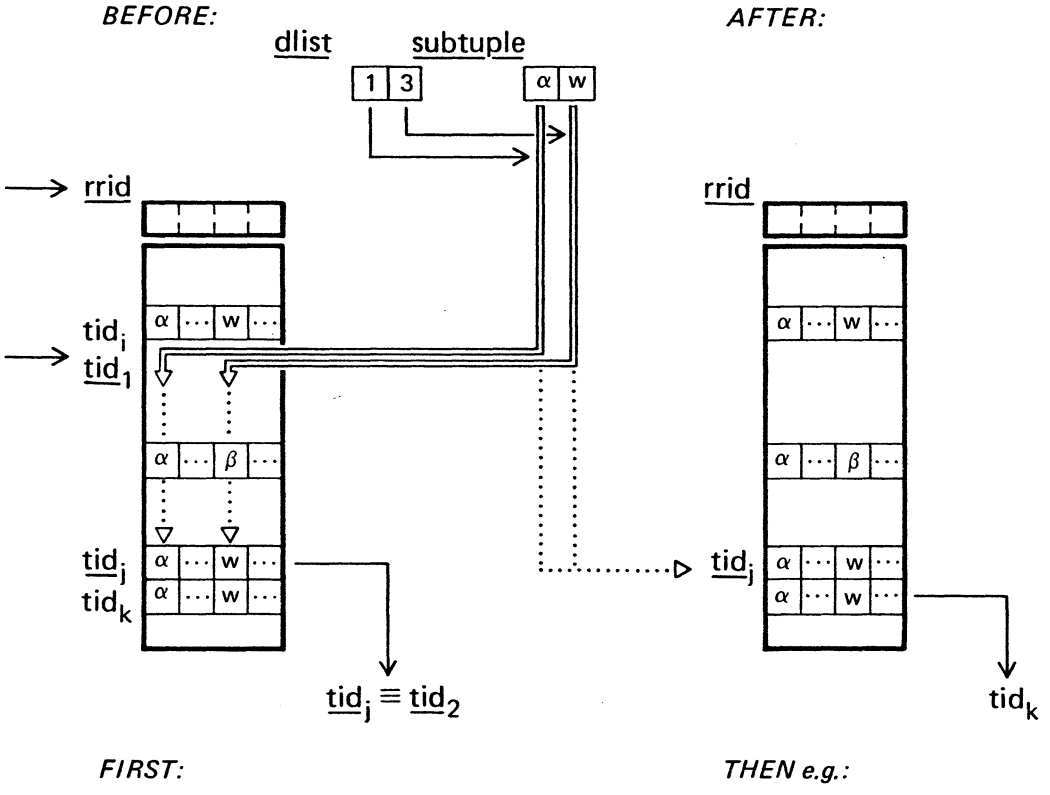


FIG. 19 TUPLE ID FROM NEXT SUBTUPLE
(filter on $<\alpha, w>$ in domains $<1, 3>$)

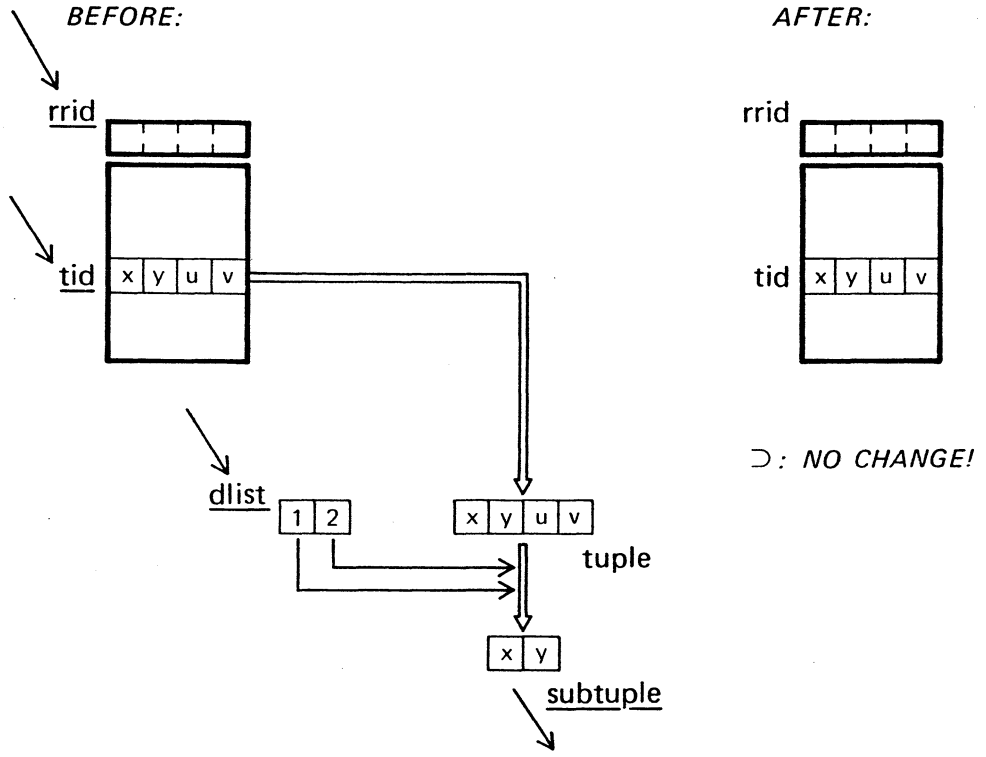


FIG. 20 SUBTUPLE FROM TUPLE ID

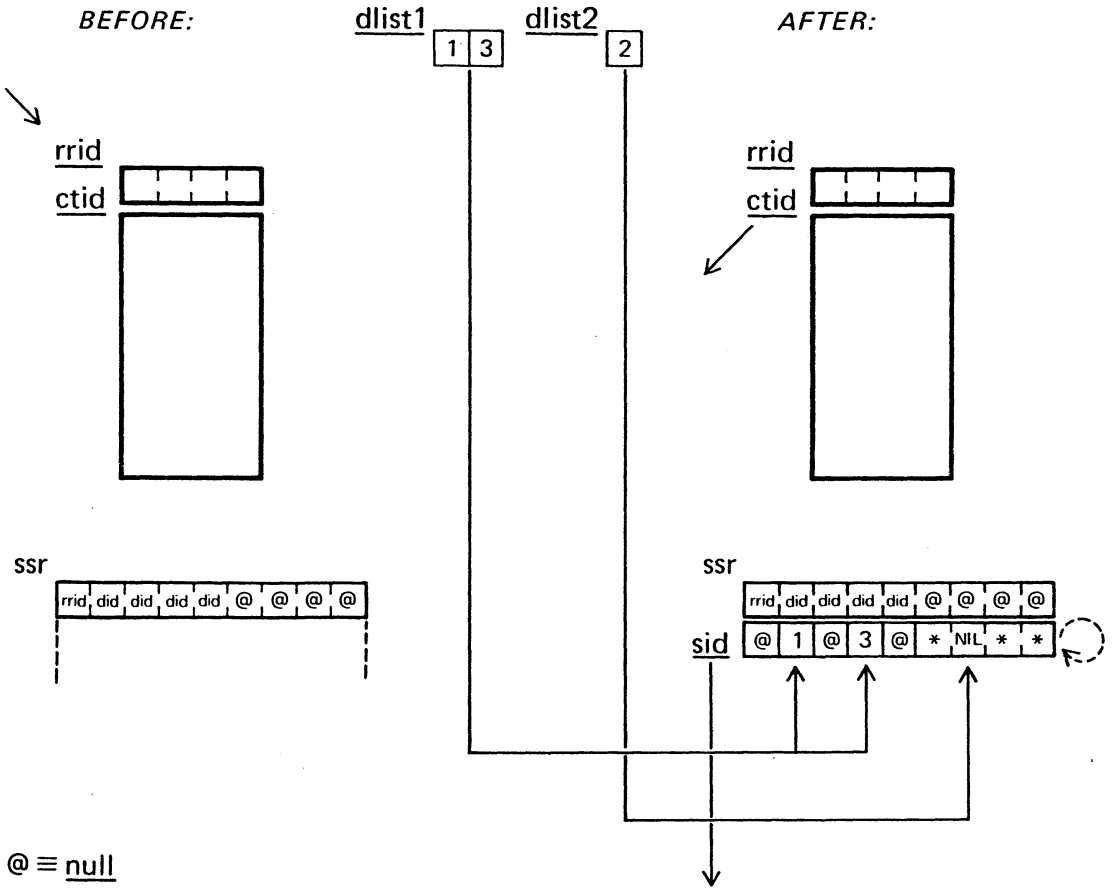


FIG. 21 CREATE SCAN (previously no scan on this relation)

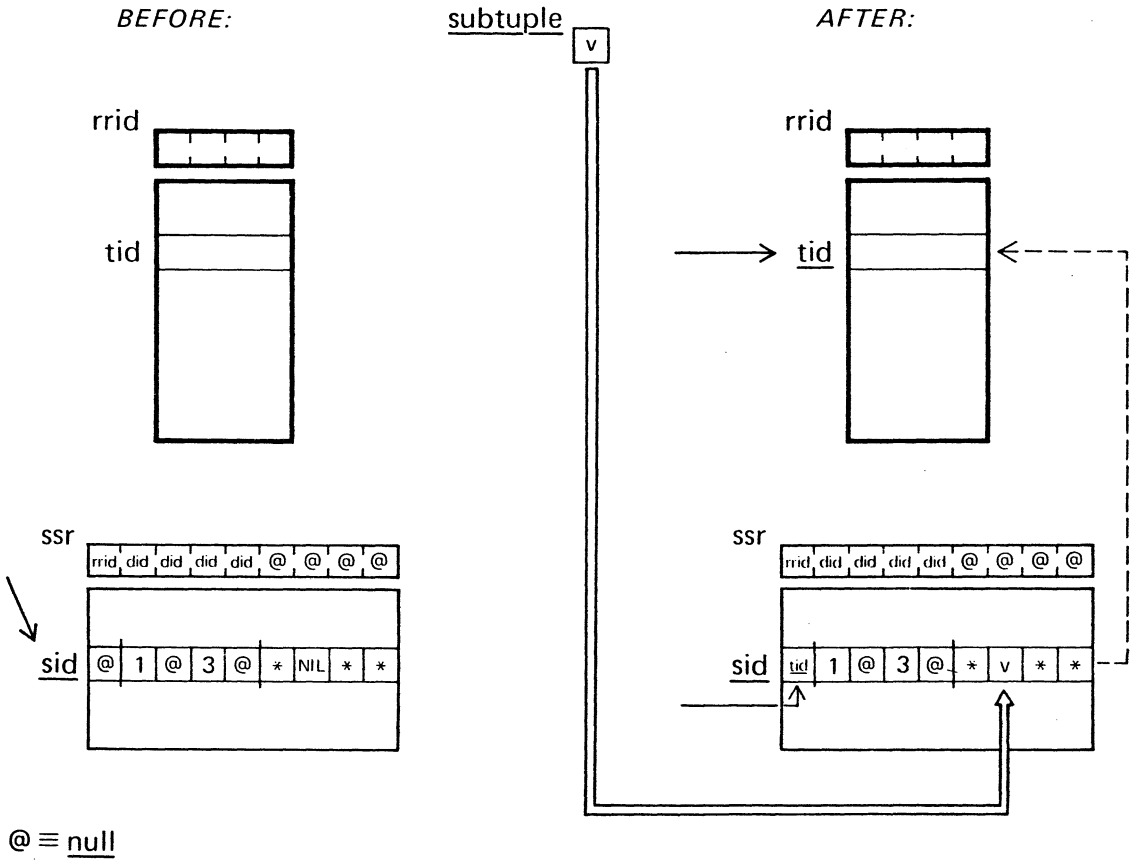


FIG. 22.1 SET SCAN (first "set scan" on this scan instance)

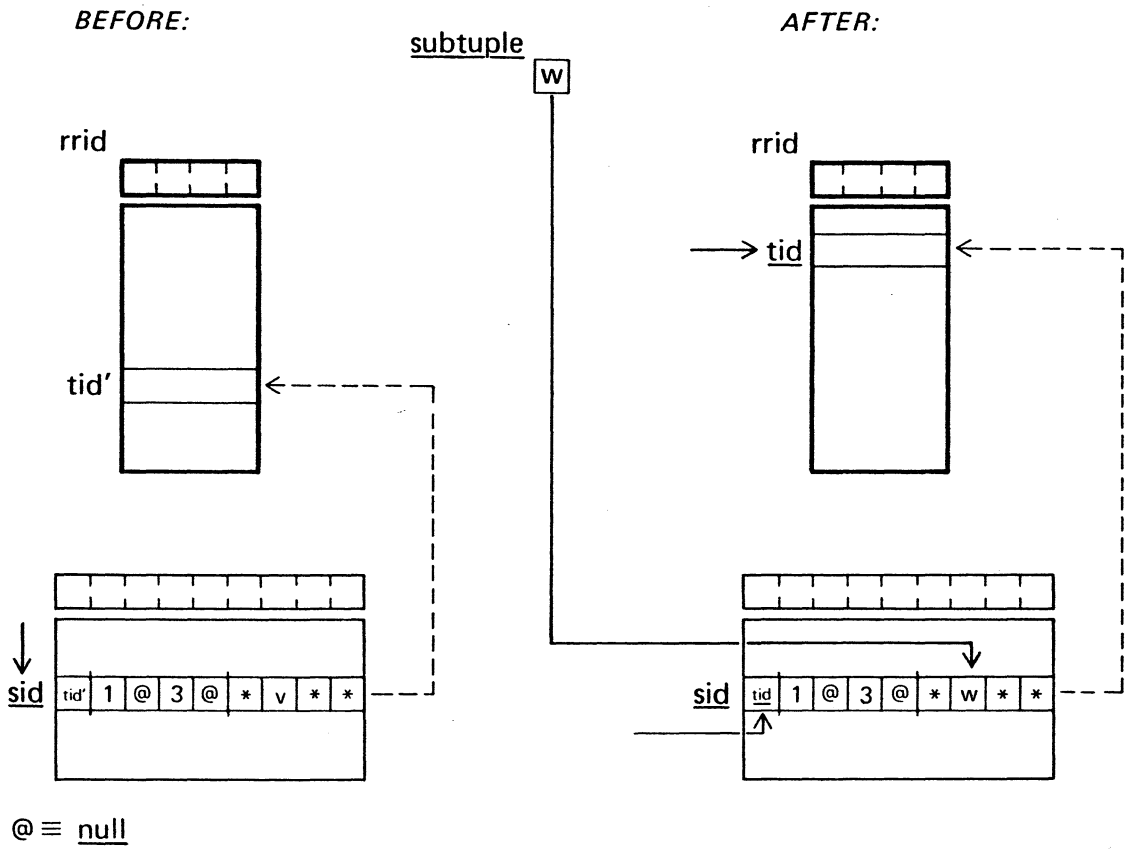


FIG. 22.2 SET SCAN (a subsequent "set scan" on this scan instance)

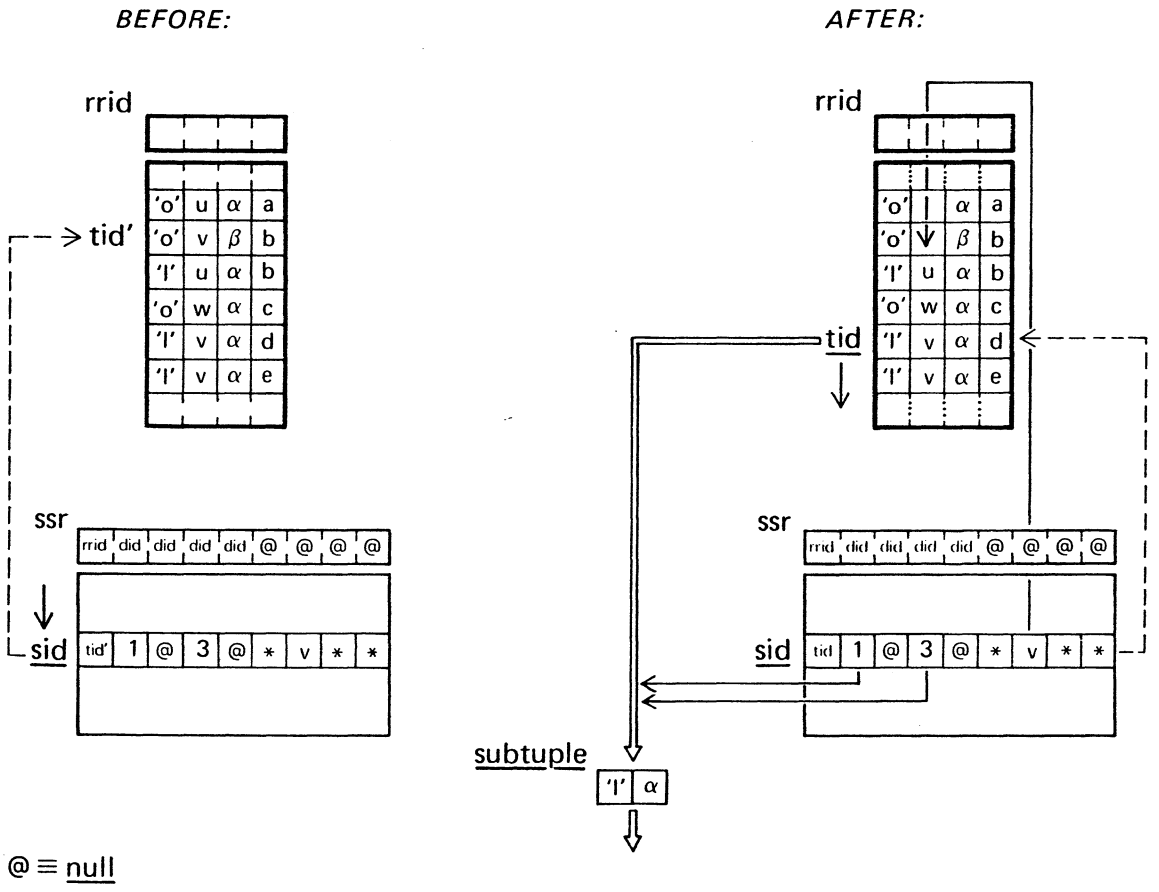


FIG. 23 NEXT SUBTUPLE

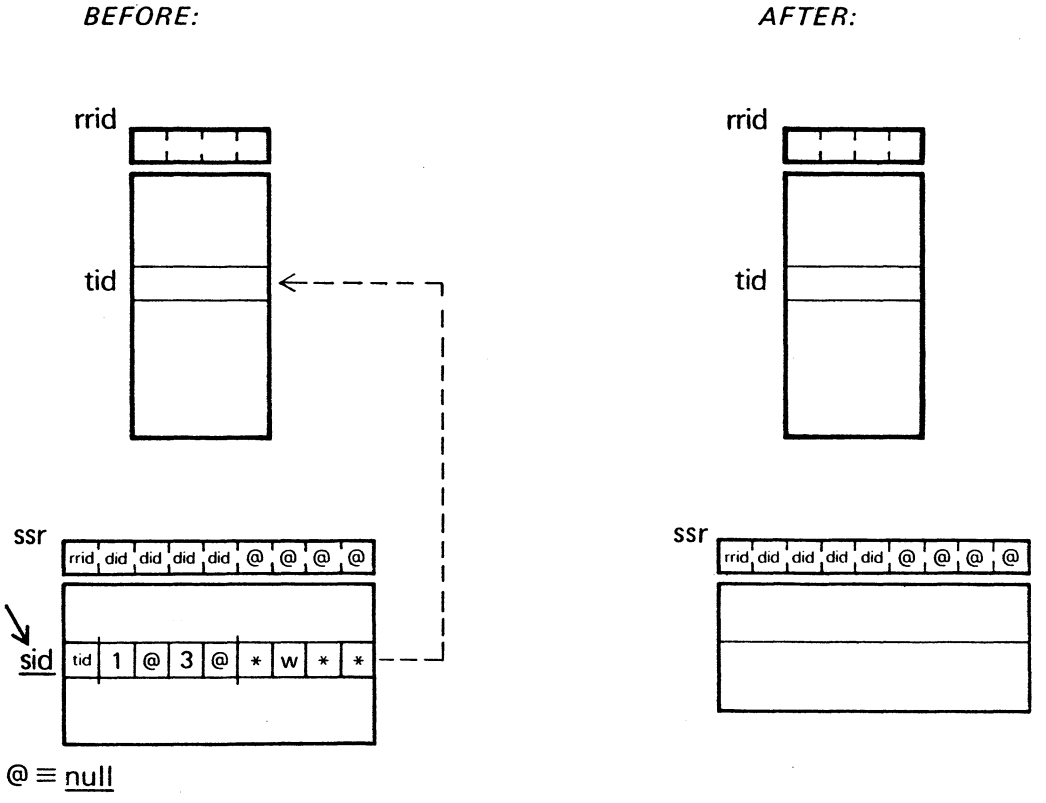
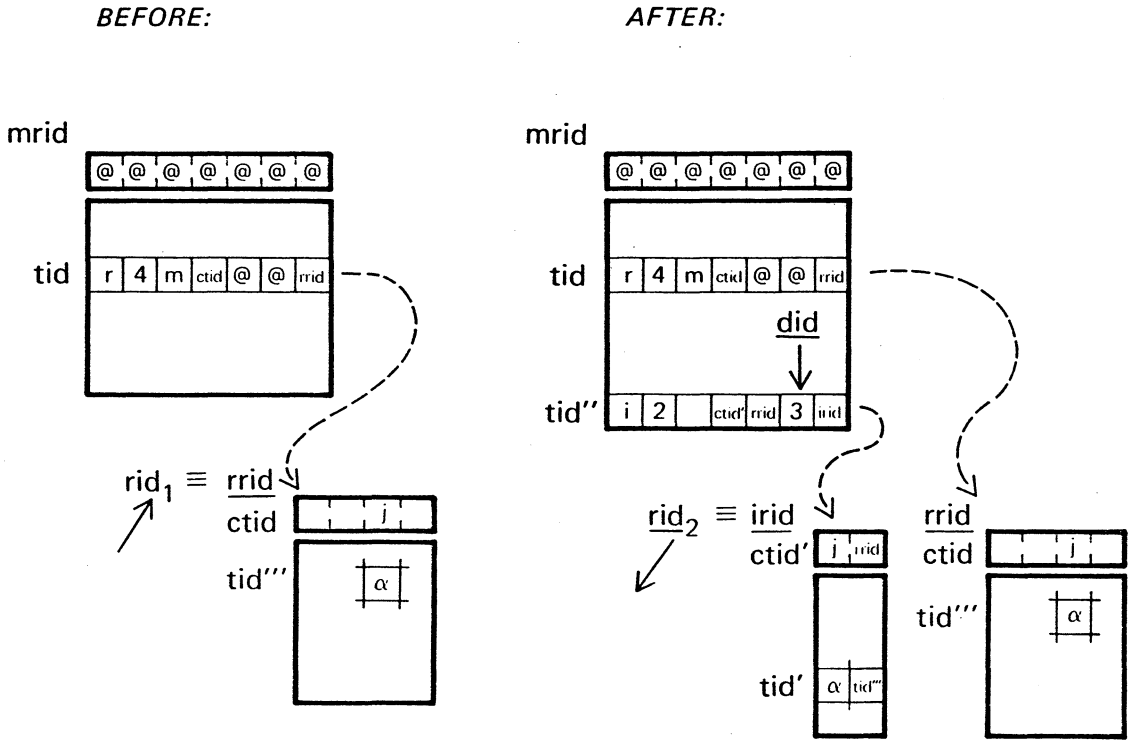


FIG. 24 DROP SCAN



@ ≡ null

FIG. 25 GENERATE INVERSION

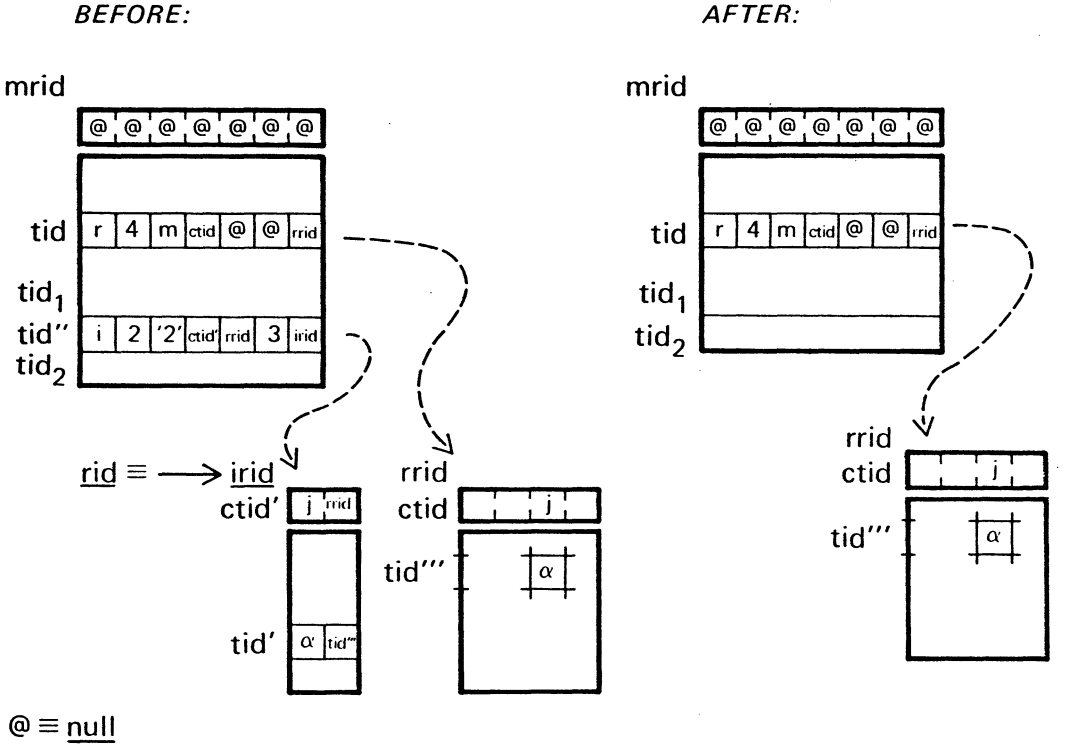


FIG. 26 DROP INVERSION