The structuring of a data base and its implementation through the use of several access methods are presented. A number of implementation methods are described and compared. Also presented are retrieval, updating, reorganization, and recovery. Other parts of this five-part series on IMS/VS include objectives and architecture, batch processing, data communication, and transaction processing.

The informatiom management system IMS/VS Part II: Data base facilities

by W. C. McGee

IMS/VS provides facilities that assist the user in establishing and maintaining a collection of data to be shared among a number of applications. Such a collection is a *data base*. IMS/VS provides support for the following data base functions:

- Definition.
- Creation.
- Retrieval and updating.
- Reorganization and recovery.

The general approach taken for supporting these functions is to provide special languages and utility programs for performing the definition, reorganization, and recovery functions, and to provide a set of call-level functions for data base retrieval and updating so that these operations can be conveniently achieved through user-written application programs. Data base creation is accomplished by a combination of IMS/VS utility programs and user-written load programs.

IMS/VS provides two distinct classes of data structures: (1) the class of structures seen by the *data administrator* (which we call the *DA class*); and (2) the class of structures seen by the *application programmer* (which we call the *AP class*). The DA class has been designed for the efficient storage and retrieval of shared data. This is basically a hierarchic class, but with provision for the interconnecting of hierarchies into networks to avoid redundant storage. The AP class is a subset of the DA class. It is strictly hierarchic, and affords a simplified view of data appropriate for the development of application programs. Provision has been made for defining AP structures in terms of DA struc-

tures, and all application program operations on AP structures are automatically reflected in the underlying DA structures. The AP class and associated retrieval and updating facilities of IMS/VS are discussed more fully in Part III, on batch processing facilities.

In this part we describe the DA data structure class, the implementation of DA structures, and the facilities provided for the creation, reorganization, and recovery of DA structures.

The DA data structure class

The data structure types in the DA data structure class are identified in Figure 1. The figure indicates informally the way in which structures are composed of other structures. Thus the construct A → B means generally that structures of type A are composed of structures of type B.

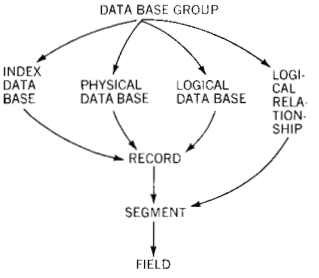
segments

The basic structure type in the DA class is the *segment*, which is a string of bytes. Segment types may be defined by the user. Attributes of a segment type include its name, length type (fixed or variable), and length (if fixed) or minimum length and maximum length (if variable). If a segment type has the fixed length attribute, every instance of that segment type has the same length. If it has the variable length attribute, instance lengths range between the defined minimum and maximum. The first two bytes of a variable-length segment contain the segment length.

One or more *fields* may be associated with a segment. A field is a sequence of bytes within a segment. Field type attributes include name, value type, (fixed) length, and starting position in the associated segment. The following value types are provided:

Type	*Allowed lengths (bytes)*
Hexadecimal	1–255
Character	1–255
Packed decimal	1–16

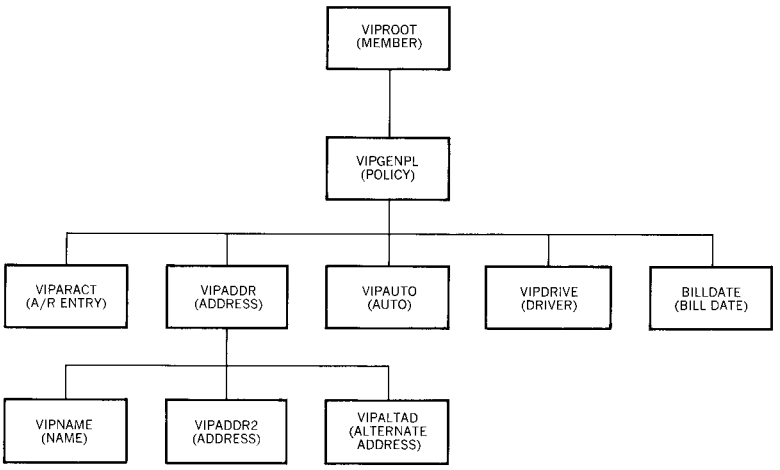
Figure 1 Data administrator data structure types



The segment is typically used to represent an application entity, such as an automobile or an insurance policy. The fields of a segment represent the values of the attributes of such entities. For example, within a segment that represents an automobile, fields may be used to represent the make, model, and year of the automobile.

One of the fields in a segment may be designated as the segment sequence field, or *key*. If a segment type has a key, segments of that type occur under their parent segment (as defined in the

Figure 2 Example record type diagram



following) in ascending key value sequence. A key may be of type *multiple* or *unique*, meaning that duplicate key values may or may not occur, respectively, under a given parent. Segment type and field type attributes are declared to IMS/VS using a special data description language. As an example, to declare a segment named VIPAUTO, the following statements can be used:

```
SEGM NAME=VIPAUTO, BYTES=119, . . .
FIELD NAME=(CARNMBR,SEQ,U), BYTES=2, START=5, TYPE=C
FIELD NAME=USECODE, BYTES=1, START=1, TYPE=C
FIELD NAME=EFFDTE, BYTES=3, START=2, TYPE=C
FIELD NAME=MODELYR, BYTES=2, START=7, TYPE=C
FIELD NAME=MAKE, BYTES=5, START=9, TYPE=C
etc.
```

records

Segments are aggregated into *records* in conformance with a record type, which is a single-rooted tree of distinct segment types. The root segment of a record type is typically used to represent a major application entity type, and the dependent segment types represent hierarchically subordinate entity types or collections of attributes that occur optionally or with variable frequency. Figure 2 illustrates the record type for the Automobile Club of Michigan auto insurance policy data base, and indicates the entity type or attribute collection represented by each segment type. A record is a single-rooted tree of segments that is produced from its record type according to the following rules:

1. The root segment type produces a single segment of that type.
2. Each dependent (child) segment type produces zero or more segments of that type (twins) under each instance of its

parent segment type. If the child segment type has a key, the twins are in sequence by key value; otherwise their sequence is determined by the manner in which they are placed in the record by the user's program.

Record types are defined by specifying a parent segment name in each segment type definition. The definition of the record type in Figure 2 has the following form:

```
SEGM NAME=VIPROOT, PARENT=0, . . .
SEGM NAME=VIPGENPL, PARENT=VIPROOT, . . .
SEGM NAME=VIPARACT, PARENT=VIPGENPL, . . .
SEGM NAME=VIPADDR, PARENT=VIPGENPL, . . .
SEGM NAME=VIPNAME, PARENT=VIPADDR, . . .
etc.
```

Segments with keys also have *concatenated keys*. The concatenated key of a segment is the concatenation of key values from the given segment and its antecedent segments, i.e., the segments on the path from the given segment to the root segment. Concatenated keys are used to uniquely identify segments within a data base record.

physical data base

A set of records of a single type is a *physical data base*. The sequence of records in a data base is determined by the method used to implement it. A data base is defined by the statement DBD NAME=data-base-name . . . followed by the SEGM and FIELD statements that define its record type.

The segments of a data base have a *hierarchic sequence*, which is defined as the sequence obtained by applying the following (recursive) procedure to the root segment of each record of the data base in sequence:

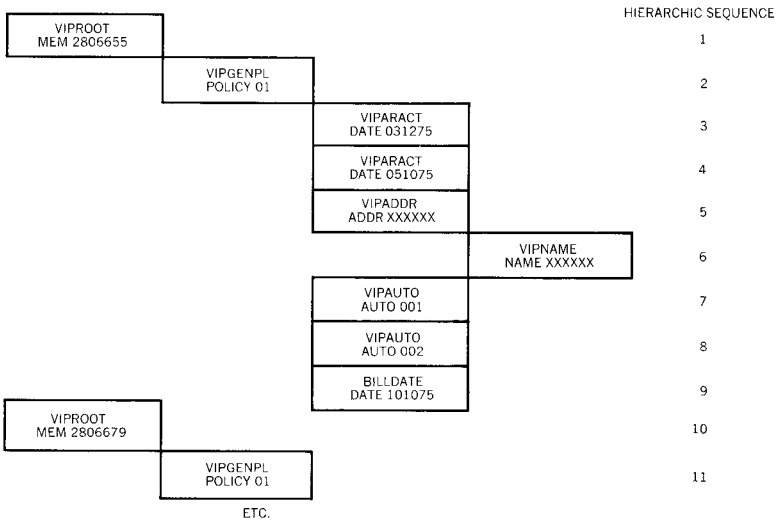
```
traverse x:
  get x;
  for each child y of x, traverse y;
```

Twins are taken in sequence, and sets of twins under a given parent are taken in the order of definition of the corresponding segment types.

Figure 3 illustrates two instances of the record type of Figure 2, with the hierarchic sequence as indicated.

The structure types described so far are sufficient to represent many different information situations. It often happens, however, that the use of these structures alone results in redundant data being carried in two or more physical data bases. To obviate this, IMS/VS provides the *logical relationship* structure type

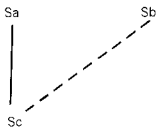
Figure 3 Example data base records



that, in effect, permits data stored in one data base to be accessed from another. Logical relationships are also convenient for representing many-to-many binary relations between application entities.

unidirectional

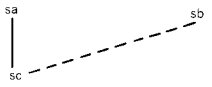
Two types of logical relationships are provided: unidirectional and bidirectional. A *unidirectional logical relationship* consists of three segment types interconnected as follows:



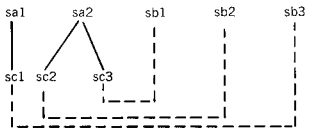
Sa and Sc must stand in a parent-child relationship in some record type. In the logical relationship, Sa is called the *physical parent* of Sc, and Sc is called the *physical child* of Sa. Sb may be in the same record type as Sa, or in some other record type, and is called the *logical parent* of Sc. The *logical child* of Sb is Sc.

Sb may in fact coincide with Sa, thereby giving the following structure:

An instance of a unidirectional logical relationship consists of one or more constructs of the following form:



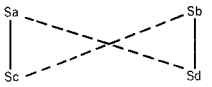
where sa, sb, and sc are instances, respectively, of Sa, Sb, and Sc. Any (sa,sb) pair may be connected once in this manner, and every sc must appear in one and only one such connection, as in the example:



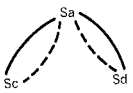
This construct is called "unidirectional" because in its implementation there exists only one path between sa and sb; namely, the path (sa,sc,sb). In order to implement this path, each sc must contain in its initial byte positions the concatenated key of its associated sb.

bidirectional

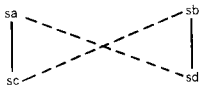
A *bidirectional logical relationship* consists of four segment types interconnected as follows:



Sa and Sc must be parent and child in some record type, as must Sb and Sd (in the same or different record type). Sa and Sb may also coincide as follows:

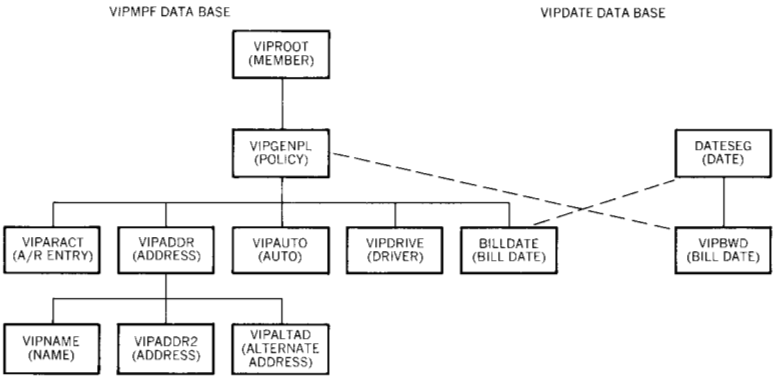


An instance of a bidirectional logical relationship consists of one or more constructions of the form:



where sa, sb, sc, and sd are instances, respectively, of Sa, Sb, Sc, and Sd. Any pair (sa,sb) may be connected once in this manner, and every sc and every sd must appear in one and only one such connection. The construct is called "bidirectional" since two paths now connect sa and sb, i.e., (sa,sc,sb) and (sb,sd,sa). Segments sc and sd are said to be paired. Both segments must contain the concatenated key of their respective logical parents, and the remainder of both segments must be identi-

Figure 4 Bidirectional logical relationship



cal. When one of these segments is modified, the system automatically modifies the other in identical manner.

If Sa and Sb represent two (or possibly the same) entity types, the logical relationship represents a binary relation between entities of the two types. The connection segment sc may be used for "intersection" data that describes each pair in the relation. Figure 4 illustrates a bidirectional logical relationship that represents a binary relation between a set of policies and a set of dates. The path from VIPGENPL to DATESEG can be used to determine the (presumably one) billing date for a given policy, and the reverse path can be used to determine the policies that have a given billing date.

The logical relationship of Figure 4 is defined with the following statements:

```
DBD NAME=VIPMPF . . .                  DBD NAME=VIPDATE . . .
. . .                                  . . .
SEGM NAME=VIPGENPL . . .               SEGM NAME=DATESEG . . .
LCHILD NAME=(VIPBWD,VIPDATE),          LCHILD NAME=(BILLDATE,VIPMPF),
       PAIR=BILLDATE . . .                    PAIR=VIPBWD . . .
. . .                                  . . .
SEGM NAME=BILLDATE,                    SEGM NAME=VIPBWD,
      PARENT=((VIPGENPL),                    PARENT=((DATESEG,
      (DATESEG,P,VIPDATE)) . . .             (VIPGENPL,P,VIPMPF)) . . .
```

The physical data base and logical relationship constructs just presented permit the construction of data structures of considerable generality and complexity. Experience indicates that, although such structures are necessary for efficient storage and retrieval, they are rarely needed for the expression of data processing procedures; rather, they tend to complicate the development and maintenance of such procedures. IMS/VS has, there-

fore, been designed to limit the application programmer's view to hierarchically structured records, and to provide facilities that allow the data administrator to define virtual hierarchic structures in terms of physical data bases and logical relationships. These virtual structures are *logical data bases.*

logical data base

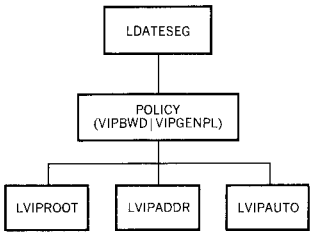
A logical data base is, like a physical data base, composed of one or more records, with each record being a single-rooted tree of segments patterned after a single tree-structured record type. A logical data base record type may be constructed by selecting any physical data base record type, and for each segment type therein performing one of the following actions:

- Include the segment type as it is (possibly with a new name).
- Omit the segment type (and its dependent segment types).
- If the segment type is a logical child, replace it with a new segment type that is the concatenation of the logical child with its logical parent.

When a concatenated segment type is produced, the antecedents and dependents of the logical parent in question can be copied as dependents of the concatenated segment.

Thus, segment types of any number of physical data base record types may be incorporated into a single tree-structured logical data base record type. To illustrate, Figure 5 shows a logical data base record type that has been constructed from the VIPMPF and VIPDATE data bases shown in Figure 4. This record type would be appropriate for processing all the policies that have a given billing date.

Figure 5 Logical data base record type



Since logical data bases are virtual data bases, they do not have a separate existence in secondary storage. Instead, segments of logical data bases are materialized as required from the underlying physical data bases and are passed to the application program. Similarly, logical data base segments are accepted from the application program, and the underlying structures are inserted, replaced, or deleted as appropriate.

Although logical data bases are defined as an aid to application programmers, it is convenient to regard them as part of the DA class, since they are usually defined at the same time as physical data bases and logical relationships, and use the same data definition language. A logical data base definition has the same general form as a physical data base definition, except that no fields are defined. (The physical data base field definitions are used.) Also every segment type has one or two source segment types defined for it. To illustrate, the definition of the logical data base record type in Figure 5 has the following form:

```
DBD   NAME=LDATE . . .
SEGM NAME=LDATESEG,PARENT=0,SOURCE=(DATESEG,VIPDATE)
SEGM NAME=POLICY,PARENT=LDATESEG,
      SOURCE=((VIPBWD,,VIPDATE),(VIPGENPL,,VIPMPF))
SEGM NAME=LVIPROOT,PARENT=POLICY,
      SOURCE=((VIPROOT,,VIPMPF))
SEGM NAME=LVIPADDR,PARENT=POLICY,
      SOURCE=((VIPADDR,,VIPMPF))
SEGM NAME=LVIPAUTO,PARENT=POLICY,
      SOURCE=((VIPAUTO,,VIPMPF))
```

index data base

To expedite direct reference to physical data bases by way of their data content, IMS/VS provides an *index data base* construct. An index data base is a degenerate case of a physical data base, in which the record type consists of a single (root) segment type. The following types of index data bases are provided:

Primary index data bases provide a means for directly accessing the records of a physical data base (i.e., the indexed data base) by way of the key values in the record root segments. The index data base contains one record for each record in the indexed data base. Each record of the index data base contains the key value from the root segment of the corresponding record in the indexed data base. (Indexed data base keys must be unique.) Each index data base record also contains a pointer to the root segment of the corresponding record in the indexed data base, in the manner described in the next section. The key values also serve as key fields for the index data base records, i.e., index data base records are in sequence and are uniquely identified by the key values.

Secondary index data bases provide means for directly accessing a segment within a physical or logical data base by means of data within the segment or within some dependent of the segment. The segment to be accessed is called the *target segment*, and the segment that contains the fields on which access is predicated (collectively called the *search field*) is called the *source segment*. A secondary index data base contains one record for each occurrence of the source segment type in the indexed data base.

Each index data base record contains a search field value which is the concatenation of the values of up to five fields in the associated source segment, and a pointer to the target segment that is associated with the source segment. The index data base record may also contain a *subsequence field* derived from up to five source segment fields, up to five *duplicate data* fields taken from the source segment, and arbitrary user-maintained data.

The search field and subsequence field together serve as the secondary index data base record key, which may be either a unique or a multiple key. The purpose of the subsequence field is to permit the generation of unique keys within the index data base, even in the presence of duplicate search fields, and thus make the accessing of the index data base more efficient.

Index data bases are automatically created and maintained by the system. In addition, they may be accessed by application programs in the same manner as physical data bases. The provision for carrying duplicate data in secondary indexes is useful for rapidly accessing frequently used data.

Index data bases are defined in a manner similar to physical data bases. An LCHILD statement is used to designate the target segment type, and the target segment type definition specifies, through an XDFLD statement, the source segment type and search field.

At most, one primary index data base and any number of secondary index data bases may be associated with a given physical data base. All the physical data bases connected by logical relationships, together with their associated index data bases, constitute a *data base group*. The data base group is a significant construct from an operational point of view, since in general all members of a group must be present to process any one of them.

Implementation of DA data structures

To implement the data structures just described, IMS/VS uses the following operating system access methods:

- Sequential Access Method (SAM).
- Indexed Sequential Access Method (ISAM).
- Virtual Storage Access Method (VSAM).

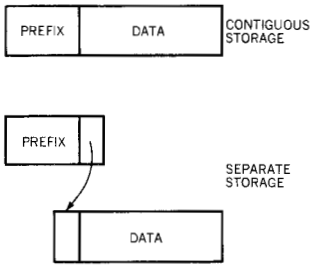
IMS/VS also uses the specially developed Overflow Sequential Access Method (OSAM) to supplement ISAM. OSAM provides for the storage and retrieval of fixed-length unblocked or blocked records on direct access storage devices. Records may be accessed either sequentially or directly by relative byte number.

IMS/VS data structure implementations make extensive use of physical pointers. A physical pointer contains a four-byte number that is the relative number of a byte within an access method data set. A physical pointer points to a data set record or to a byte sequence therein by specifying the relative byte number of the first byte of the record or sequence.

For implementing physical data bases, IMS/VS provides the following implementation methods:

- Hierarchic Sequential Access Method (HSAM).
- Hierarchic Indexed Sequential Access Method (HISAM).
- Hierarchic Indexed Direct Access Method (HIDAM).
- Hierarchic Direct Access Method (HDAM).
- Generalized Sequential Access Method (GSAM).
- Data Entry Data Base (DEDB).
- Main Storage Data Base (MSDB).

Figure 6 Variable-length segment implementation



Any one of these methods may be selected by the user for implementing a given physical data base. Selection is made in the DBD statement for the data base as in the following example:

```
DBD NAME=VIPMPF, ACCESS=(HDAM,OSAM)
```

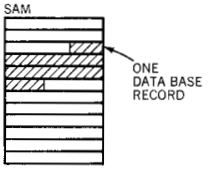
Each access method has different storage and performance characteristics, and the user is, therefore, able to select a method that closely matches the requirements of the particular data base being implemented. The methods also differ in the degree to which they accommodate the various data constructs described so far. Thus only HISAM, HIDAM, and HDAM data bases may participate in logical relationships and have associated index data bases. Other limitations are noted later in this paper.

In all implementations, a segment is implemented in two parts: (1) a *prefix part* that contains a segment type code and other implementation related information; and (2) a *data part* that contains the DA segment byte sequence as shown in Figure 6. The prefix and data parts of a segment are stored contiguously unless the length of a (variable-length) segment increases beyond the space originally allocated. In that event, the data part is stored separately, and connected to the prefix by a physical pointer. For brevity, the term "segment" is now assumed to denote the implementation of a DA segment.

HSAM

In HSAM, a physical data base is implemented as a single SAM data set with fixed-length unblocked records, as shown in Figure 7. The segments of each data base record are stored in hierarchic sequence, in one or more consecutive data set records. Each data set record holds an integral number of segments, and any residual space in the record is unused. The last segment of one data base record is followed immediately (in the same or next data set record) by the root segment of the next data base record. When the data base is created, data base records are stored in the sequence in which they are presented by the user's program. If the root segments have keys, records must be presented (and stored) in ascending key value.

Figure 7 Example of HSAM implementation



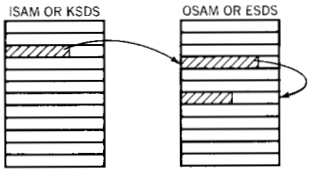
A simple variation of HSAM is provided for the common case of root-only data bases. This Simple Hierarchic Sequential Access Method (SHSAM) is the same as HSAM, except that segments have no prefixes. HSAM is intended for data for which the predominant mode of access is sequential. Direct access is possible but not often used, since it requires a linear search of the data base, and updating requires the writing of a new version of the data base.

HISAM

In HISAM, a physical data base may be implemented on one of two access method bases, ISAM/OSAM or VSAM. With the ISAM/OSAM base, a physical data base is implemented as two data sets, an ISAM data set and an OSAM data set. With the VSAM base, two data sets are also used, a Key Sequenced Data Set (KSDS) and an Entry Sequenced Data Set (ESDS). The two implementations are similar, with the ISAM and OSAM data sets in the first playing the roles of the KSDS and ESDS, respectively, in the second. For brevity, we describe only the VSAM implementation, and note differences for ISAM/OSAM as appropriate.

As in HSAM, a data base record in HISAM is implemented as a physically contiguous sequence of segments taken in hierarchic order. Instead of overlaying the resulting sequence on SAM records, however, the sequence is divided into one or more subsequences, with each subsequence consisting of a whole number of segments. The first subsequence is placed in a KSDS record, and the remaining subsequences, if any, are placed in one or more ESDS records that are chained by physical pointers to the associated KSDS record, as shown in Figure 8. Each record holds as many segments as it can, with any residual space being available for segment inserts.

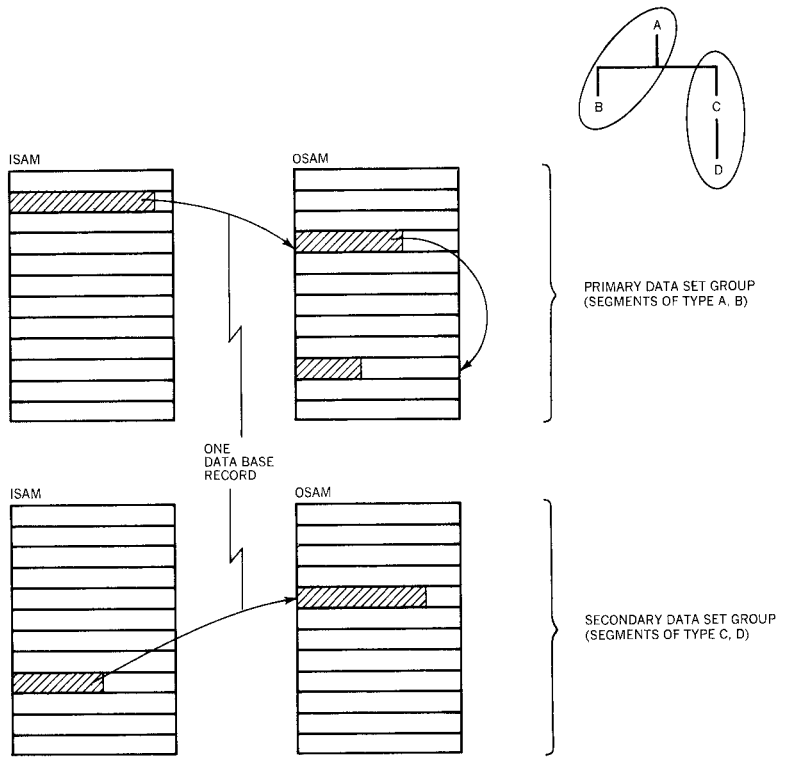
Figure 8 Example of HISAM implementation



The root segment type of a HISAM data base must have a unique key. The records of the data base are in ascending sequence on the values of this key. The corresponding KSDS records are also sequenced and are uniquely identified by this key. The ESDS records used for a given data base record may occur in any sequence.

One implementation difference between ISAM/OSAM and VSAM occurs in the insertion of records after the data base has been created. In ISAM/OSAM, the first subsequence of such a record is placed in the OSAM data set, and is chained to the ISAM record with the next higher key value. In VSAM, a new VSAM record is added to the KSDS, and the first subsequence is placed in that record. Another implementation difference lies in the provision in ISAM/OSAM for partitioning the data base record type into subtrees of segment types, and storing the instances of each subtree in a separate *data set group*. Such a group consists of an ISAM and an OSAM data set as shown in Figure 9. The *primary*

Figure 9 Example of HISAM implementation with multiple data set groups



data set group holds subtrees headed by the root segment type, and each *secondary data set group* holds subtrees headed by a different second-level segment type.

By analogy with SHSAM, a Simple Hierarchic Indexed Sequential Access Method (SHISAM) is provided for root-only data bases, for which only a VSAM implementation is provided. Each data base record is wholly contained in the KSDS, and no ESDS is required. Prefixes and pointers are removed from both the root segment and the containing KSDS record, so that the data base becomes physically identical to a VSAM Key Sequenced Data Set.

The parameters of a HISAM implementation are declared in one or more DATASET statements, one statement for each data set group. Parameters include the identity of Job Control Language statements that contain data set allocation parameters, secondary storage device type, and data set record length and blocking factor.

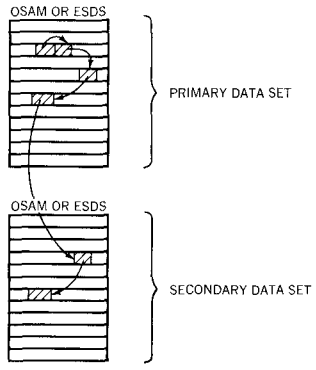
HISAM is useful for data bases to which both sequential and direct access by root segment key is required, and which do not

require extensive segment insertion and deletion. The use of a second data set (i.e., OSAM or ESDS) to hold overflow segments permits efficient keyed access to data base records having a wide variation in record lengths, and also permits the physical separation of high-usage and low-usage data within a record.

HIDAM

In the Hierarchic Indexed Direct Access Method (HIDAM), a physical data base is implemented as one or more OSAM or ESDS data sets, wherein each data set holds all occurrences of a given set of segment types (without regard to record structure). As shown in Figure 10, the data set that holds root segment occurrences is called the *primary data set*, and the remainder (which are optional) are called *secondary data sets*.

Figure 10 Example of HIDAM implementation.



In contrast to HSAM and HISAM, segments of a HIDAM data base record may be stored at arbitrary locations relative to one another. Record structure is preserved through physical pointers contained in the segment prefixes. Data set records are used as "containers" for holding segments. IMS/VS manages the space within the records, and allocates and reclaims space as segments are inserted and deleted. In allocating space, an attempt is made to keep adjoining segments in the hierarchic sequence as close together in storage as possible.

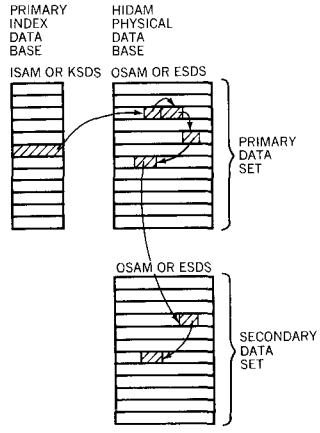
One or both of the following two types of pointers may be used to implement a record for HIDAM:

- *Hierarchic pointers* connect the segments of a record in their hierarchic sequence. Both forward and backward pointers may be used.
- *Physical child-physical twin pointers* connect a parent segment and each of its sets of twins in a chain, with the twins being in sequence by key value (if any). A separate chain can be provided for each child segment type, and backward as well as forward pointers can be used.

Hierarchic pointers are used to traverse a record in hierarchic sequence, while child-twin pointers permit rapid access to any segment within a record.

As in HISAM, the root segment type of a HIDAM data base must have a unique key, and the records of the data base are in ascending sequence on values of this key. Record sequencing is implemented through an associated primary index data base, and may be implemented optionally through forward and backward physical twin pointers in the root segments. The latter implementation obviates access to the index data base, when processing records sequentially.

Figure 11 Example of primary index implementation



A primary index data base, which may be associated only with a HIDAM physical data base, is implemented as an ISAM/OSAM data set pair or a single VSAM Key Sequential Data Set (KSDS) as shown in Figure 11. Each index data base record is placed in a separate ISAM or KSDS record, with the latter in sequence on the index data base record key. In ISAM/OSAM, records added after the initial creation of the index are placed in the OSAM data set and chained, as in HISAM.

The prefix of each root segment in a primary index data base record contains a physical pointer to the root segment of the associated physical data base record. Direct access to a record with key value K is accomplished in the following two steps:

1. Using the indexing mechanism of ISAM or VSAM, locate the index data base record containing key value K.
2. Using the pointer in the record found in step 1, locate the physical data base root segment that contains key value K.

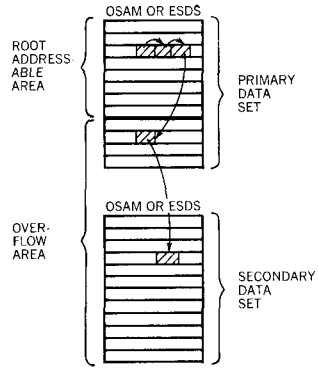
The primary index data base is automatically maintained by the system. Insertion or deletion of a record in the indexed data base causes the insertion or deletion of the corresponding index data base record.

HIDAM is useful when both sequential and direct access by root key are required, and when many segment insertions and deletions can be expected. The provision of multiple data sets for a HIDAM data base permits the parameters of each data set to be set to best accommodate the contained segment types.

HDAM

The Hierarchic Direct Access Method (HDAM) is similar to HIDAM in that segments are stored in one primary data set and in one or more secondary data sets, with the segments of a record interconnected through physical pointers. Unlike HIDAM, an HDAM data base has no associated primary index data base. Instead, direct access is provided through a randomizing routine supplied by the user. The primary data set is divided into two areas: (1) a *root addressable area* that consists of the first n records of the data set, where n is user-selectable; and (2) an *overflow area* that contains the remaining records of the data set. The secondary data sets are extensions of the over-flow area as shown in Figure 12.

Figure 12 Example of HDAM implementation



The root addressable area is used to store root segments and a limited number of dependent segments, the limit being set by the user. When a data base record is to be added, the user's randomizing algorithm converts the root segment key value (which may be unique or multiple) into a record number in the range $(1,n)$. If space is available in the record thus selected, the root segment and dependent segments up to the user-specified limit are placed

in this record, and the remaining segments are placed in one or more overflow area records. If space is not available in the record randomized to, an attempt is made to store the root and its dependents in records that are physically close to the record randomized to, or, failing this, in the over-flow area. In any case, the root segment is added to a *synonym* chain that is anchored in the record randomized to and that is used to connect all root segment synonyms, i.e., root segments that randomize to the same root addressable area record. A root segment is retrieved by randomizing a given key value, locating the root addressable area record thus selected, and following the synonym chain for that record to the root with the given key value.

To minimize the length of synonym chains, provision is made to establish up to ten synonym chains in each root addressable area record. The randomizing algorithm must produce both a record number and a chain number that specify the record and the chain within that record that hold the root with a given key.

The root segments (hence records) of an HDAM data base are in sequence by key value within synonym chain within data set record.

Parameters declared for an HDAM implementation include a randomizing module name, the size of the root addressable area, the maximum number of bytes of root and dependent segments to be stored at one time in the root addressable area, and the number of synonym chains to be included in each root addressable area record.

HDAM is useful when direct access to a data base is required, and the key set for the data base is relatively stable and capable of being randomized. As in HIDAM, HDAM can tolerate a large amount of segment insertion and deletion.

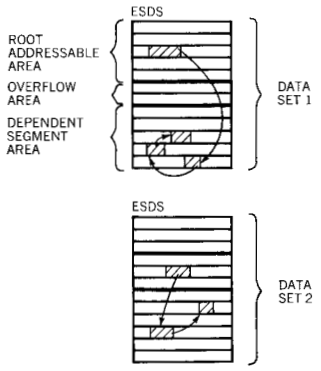
GSAM

The Generalized Sequential Access Method (GSAM) may be used to implement root-only data bases with fixed-length root segments. A GSAM data base may be implemented as a SAM data set or a VSAM ESDS, and is intended primarily for data exchange between IMS/VS application programs and other user programs that access SAM or VSAM data sets.

DEDB

The Data Entry Data Base (DEDB) implementation method is intended for applications in which a large number of key-driven terminals enter data for later processing by batch programs. This method is available only with the Fast Path feature. In a DEDB, a data base record type is restricted to a single root segment type and a single dependent segment type. A root segment is typically used to represent a terminal from which data are being entered, and dependent segments are used to hold the individual entries from that terminal.

Figure 13 Example of DEDB implementation



A DEDB is implemented as one or more VSAM ESDSs, as shown in Figure 13. Multiple data sets may be used to partition the data base into sets of data base records, i.e., all segments of a given data base record are stored in the same data set. In this way, data bases that exceed the capacity of a single data set may be accommodated.

The placement of root segments in a DEDB is governed by a user-supplied randomizing algorithm. Dependent segments are stored in a separate dependent segment area at the end of the data set. In the interest of rapid storage, dependent segments are stored in time-of-entry sequence without regard to keeping related segments physically close. Dependent segments are chained to their root in last-in-first-out manner.

MSDB

The Main Storage Data Base (MSDB) implementation method is intended for data bases that have very high access rates and that can be conveniently held in (virtual) main storage. This method is available only with the Fast Path feature.

A MSDB record type is restricted to a single root segment type of fixed length. Two root segment keying methods are provided for a given MSDB. In the first, a key value is carried in each root segment, as in other implementation methods. In the second, a one-to-one correspondence is set up between root segments and logical terminals, and the name of a logical terminal is taken to be the key of the associated root segment. The first keying method is appropriate for data bases in which many terminals must access the same—usually small—set of root segments, such as ledgers. The second is suited for data bases in which each terminal requires a dedicated storage area, such as teller records.

During an on-line execution, MSDBs are held in main storage with root segments in ascending key value. When the on-line execution is shut down, all MSDBs are dumped to disk data sets. These data sets are used to reload the data bases into main storage when the next on-line execution is started.

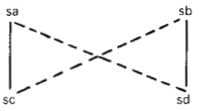
logical relationships

Logical relationships among segments of physical data bases are implemented by means of pointers of the following types: (1) physical pointers of the type used to implement the physical data bases themselves; and (2) symbolic pointers, which are concatenated keys. Physical pointers cannot be used to point to segments in HISAM data bases because these segments are subject to relocation during normal updating operations.

The following unidirectional logical relationship construct:

is implemented by placing a *logical parent pointer* in sc, i.e., a pointer that points to the logical parent (sb) of sc. This pointer enables a traversal from sa to sb via the physical parent-child path between sa and sc, and thence via the pointer in sc to sb. If sb is in a HISAM data base, the pointer in sc is a symbolic one, and is the concatenated key of sb, which must be present in the DA view of sc. If sb is in an HDAM or HIDAM data base, the pointer may be symbolic, or it may be a physical pointer carried in the prefix of sc.

The following bidirectional logical relationship construct



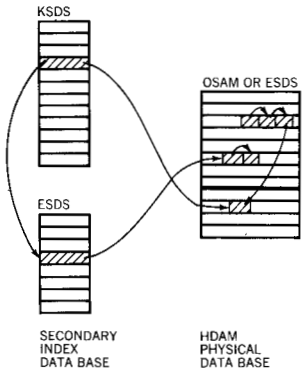
may be implemented in two ways. In *physical pairing*, logical parent pointers are placed in each of sc and sd. Two paths are thus created, one from sa to sb, and the other from sb to sa, each in the manner of the unidirectional implementation.

In *virtual pairing*, one of the two segments sc or sd does not physically exist (it exists only in the DA view of the construct). Assume sd does not exist. The path from sa to sb is implemented as in the unidirectional implementation. The reverse path is implemented by chaining sb and all associated sc's with *logical child-logical twin* pointers and placing a physical parent pointer in each sc as follows:

The traversal from sb to sa is then accomplished by following the logical child-logical twin pointers to the appropriate sc, and thence by physical parent pointer to sa. Logical child-logical twin chains can be bidirectional, and the twins on such a chain can be sequenced as though they were physical children of the logical parent. When virtual pairing is used, the real logical child of the logical child pair cannot occur in a HISAM data base, since HISAM provides no twin pointers of any kind.

Secondary index data bases are implemented by VSAM data sets. If the key of the secondary index record is unique, the data base is implemented as a single KSDS, with each data base record being placed in a separate data set record. If the key is multiple, the data base is implemented as a KSDS-ESDS pair, with the KSDS holding the first record with a given key value and the ESDS holding the remaining records with the same key value. The records with the same key value are chained together with

Figure 14 Example of secondary index implementation



physical pointers. In both cases, records of the KSDS are in ascending sequence on record key value as shown in Figure 14.

Each secondary index record contains a pointer to a target segment in a physical data base. If the physical data base is HISAM, the pointer is symbolic; i.e., it is the concatenated key of the target segment, and if it is not present in the DA view of the index record, it is concatenated to the end of the record. If the physical data base is HIDAM or HDAM, the pointer may be symbolic, or it may a physical pointer in the index record prefix.

The retrieval of target segments having a common associated search field value v is accomplished as follows:

1. The first secondary index record whose search field has the value v is located, using the generic key retrieval facility of VSAM. (The index record key is the search field plus the subsequence field.).
2. The pointer in the record that is located in (1) is followed to locate the target segment.
3. The duplicate key chain, if any, from the record located in (1) is followed into the ESDS, and the pointer in each record on the chain is followed to locate a target segment.
4. Steps (2) and (3) are repeated for each subsequent record in the KSDS that has the same search field value v.

Secondary index data bases are maintained by the system. When source segments are inserted or deleted, corresponding records are inserted or deleted in the secondary index data base. When any source field value changes, the corresponding index record is repositioned in the index, if necessary, to reflect the new value.

Definition and creation of DA structures

Each physical data base, logical data base, and index data base to be created must first be defined to IMS/VS. This definition is accomplished with a utility program called DBDGEN that runs in a batch region. Input to DBDGEN is a set of control statements for one data base, having the following general form:

```
DBD NAME=data-base-name
(DATASET,SEGM,LCHILD,FIELD,XDFLD statements)
DBDGEN
FINISH
END
```

Output is a Data Base Description (DBD) control block that is stored in the IMS/VS data base description library. The DBD

control block is used by the system during data base creation, recovery, and reorganization operations, and in handling data base requests from application programs.

A DBD may be modified by deleting the old version with an OS utility, and rerunning DBDGEN with the new version. When a DBD is modified, it is also generally necessary to recreate or reorganize the data base described by the DBD.

Because of the variety of data sources and procedures that may be used to create a data base, IMS/VS has been designed so that data base creation is accomplished with a user-written program called a load program (as opposed to, say, a generalized creation program). Such a program builds one or more physical data bases by issuing calls to IMS/VS. Each call results in the addition of a single new segment or path or segments to one of the data bases that is being created. Except for the implementation methods that use randomizing algorithms to store root segments, data base records must be added in ascending root segment key sequence. In all implementation methods, segments within each data base record must be added in hierarchic sequence.

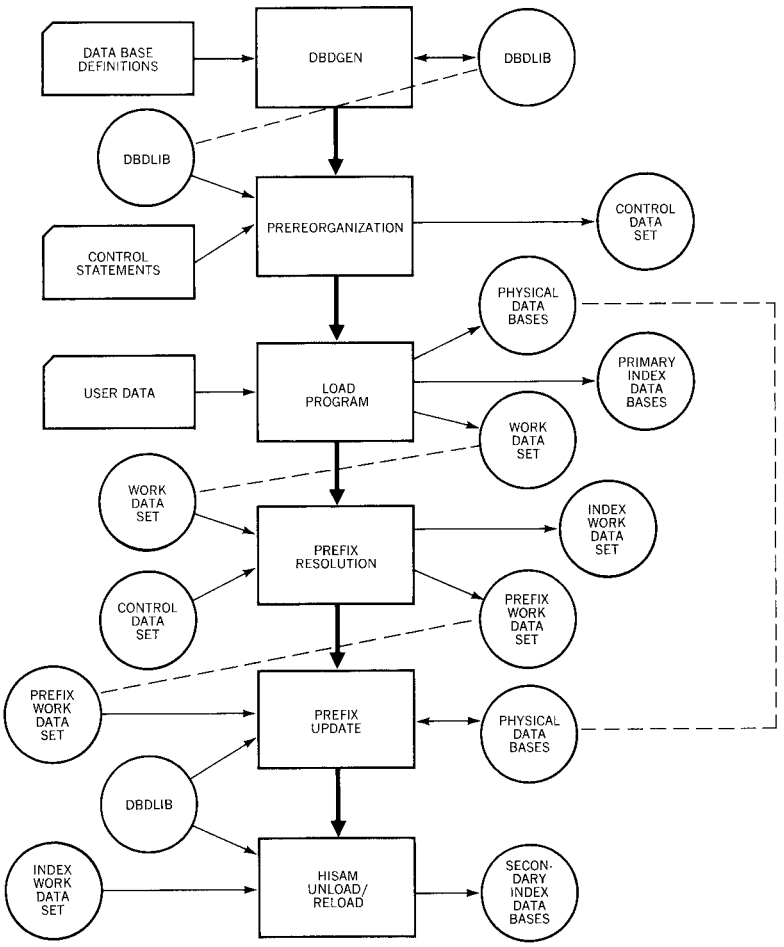
Logical data bases gain their existence through the creation of the underlying physical data bases. Index data bases are created by the system as a byproduct of the creation of the associated indexed data bases.

Creation of HISAM, HIDAM and HDAM data bases is normally done on a data base group basis, i.e., all physical data bases and index data bases within a group are created at the same time. (Reorganization facilities, which are described in the following section, permit data bases to be added to an existing group.) The creation of a group is done with a combination of IMS/VS utility programs and user load programs, all of which run in a batch region.

The procedure for the creation of a group is illustrated in Figure 15. The following programs are used in the procedure:

1. *Load program* issues calls to insert segments in one or more physical data bases. In responding to these calls, the system places segments in storage and resolves hierarchic, physical child, physical twin, and physical parent pointers based on placement in storage. Resolution of logical pointers is deferred until all data bases in the group have been loaded. In loading HIDAM and HDAM data bases, the user may specify that distributed free space be left in the data sets, so that segments that are inserted later will have a better chance of being stored close to their hierarchic neighbors. Free space is specified in the DATASET statement. For ex-

Figure 15 Data base group creation



ample, DATASET. . . .FRSPC=(5,50) . . . specifies that every fifth data set record is to be left vacant, and that no more than fifty percent of each data set record is to be used at load time. During load program execution, the system produces a primary index data base for each HIDAM physical data base created and a work data set that contains records of two kinds. One is a prefix work record for each inserted segment that is a logical parent or a logical child. Each record contains the segment's physical location and its concatenated key (if a logical parent) or its logical parent's concatenated key (if a logical child). The other kind of work data set record is an index work record for each inserted secondary index source segment, containing all the information needed to construct a secondary index entry. A single load program can be used to create all physical data bases in a group, or data bases may be created individually or in combinations with multiple load programs.

2. *Prefix resolution* is an IMS/VS utility program that sorts the work data set(s) produced in step (1) so that records for logical parents and for their associated logical children are physically contiguous. This enables the resolution of logical pointers. Work records are then distributed to a prefix work data set that contains prefix work records sorted by physical location within data set; and to an index work data set that contains index work records sorted on search field-subsequence field within secondary index.
3. *Prefix update* is an IMS/VS utility program that accepts the prefix work data set produced in step (2), and (for each record therein) locates the corresponding segment in a physical data base and completes its prefix.
4. *HISAM unload* and *HISAM reload* are IMS/VS utilities that are run in tandem to accept the index work data set produced in step (2), and create one or more secondary index data bases.

For MSDBs, data base creation is accomplished with the help of the MSDB maintenance utility program. This program accepts an old set of MSDBs in sequential data set form and a set of user-prepared changes. When no old version is provided as input, the program in effect creates a new set of MSDBs.

The Automobile Club of Michigan has created some sixty-four physical data bases for production work, using the HSAM, HISAM, HIDAM, and HDAM implementation methods. About half the data bases are HIDAM. In addition, some thirty physical data bases have been created for program testing and operator training. Among the larger data bases are the automobile insurance policy data bases. To simplify reorganization and recovery, policy information is held in five separate data bases that have identical record structure (Figure 2). The last digit of a member's number determines the data base in which his policy information is kept. Each data base contains about one-hundred sixty-two thousand records, with an average segment count of twenty-six and an average length of thirteen hundred twenty bytes. The five data bases together occupy eight 3330 disk packs.

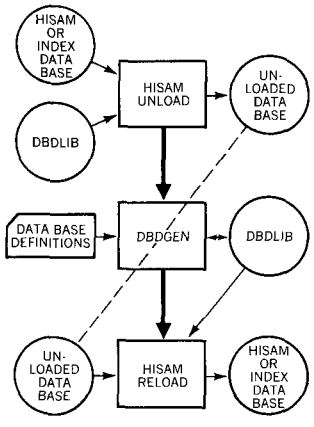
Reorganization of DA structures

Data bases that are subject to segment insertions, deletions, or variable-length segment replacements tend to become disorganized with use, i.e., segments normally stored together become separated, and space becomes fragmented into pieces too smalll to be used for segment insertions. In addition, deleted segments accumulate in HISAM data bases, since space is not reclaimed upon deletion. The effect of this disorganization is to increase both the number of secondary storage accesses and the amount of CPU processing required to respond to program calls for data services.

The IMS/VS data base reorganization facilities permit the user to restore the physical storage structure of an existing data base to the condition it would have if it had just been loaded. In general this is achieved by unloading the data base onto a sequential data set, with segments in hierarchic sequence, and subsequently reloading the data base, i.e., creating a new version of the data base, using the sequential data set created in the unload process. Deleted segments are dropped in the unloading process.

For the storage reorganization of a HISAM data base, the procedure of Figure 16 may be used. The procedure uses the HISAM unload utility program and the HISAM reload utility program. The procedure may be used to reorganize storage in both HISAM physical data bases and index data bases (whose storage organization is the same as HISAM data bases).

Figure 16 HISAM and index storage reorganization

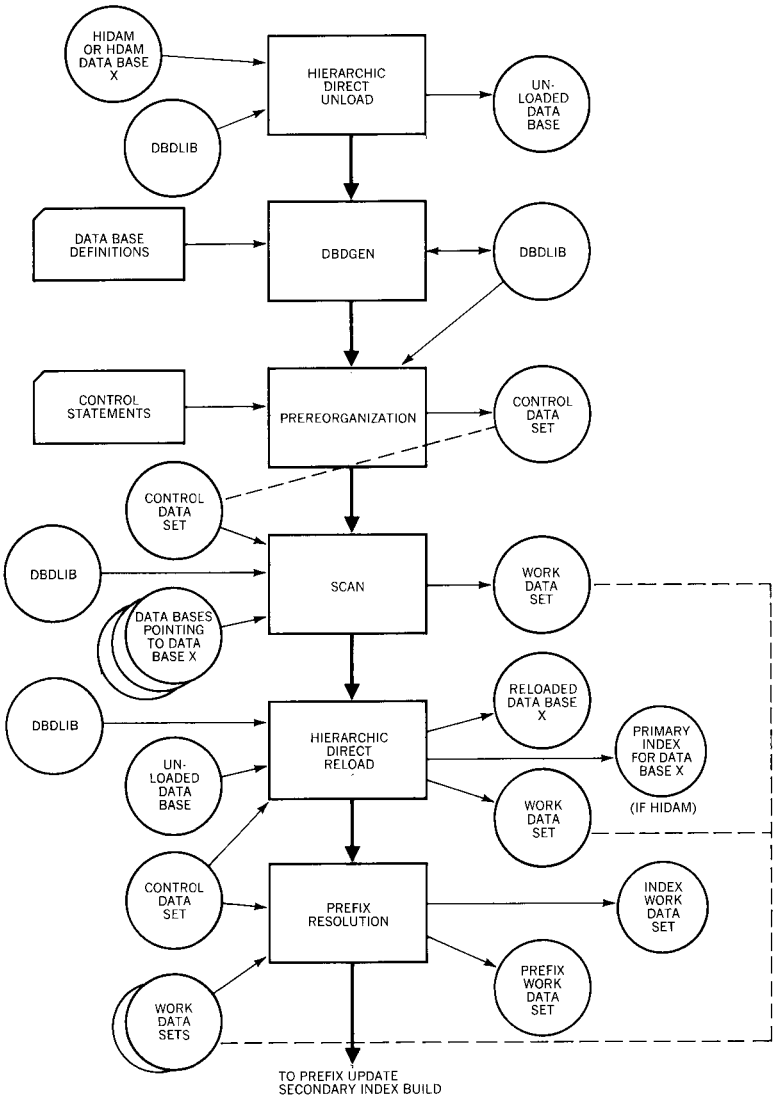


The storage reorganization of HIDAM and HDAM data bases uses the procedure of Figure 17, which makes use of the hierarchic direct unload utility program and the hierarchic direct reload utility program. This procedure differs significantly from the HISAM procedure in that in reorganizing a HIDAM or HDAM data base, it is necessary to move segments from their original insert position. As a consequence, pointers to these segments in other data bases must be adjusted. In reorganizing a HISAM data base, segments are also moved, but this has no effect on other data bases, since all pointers into HISAM are symbolic.

The process of resolving pointers in storage reorganization is similar to that used when a data base group is created. In particular, the reload program acts as a user load program. In addition to its primary function of loading segments and resolving physical parent-child pointers in the new data base, the reload utility creates a work data set with prefix work records and index work records as in initial data base load. A separate utility program, the data base scan utility, is used to create a similar work data set of logical parents and logical children in data bases that are not being reorganized, but which are related to the data base that is being reorganized. The two work data sets are combined by the prefix resolution utility that resolves logical parent-child pointers and generates the output for prefix update and secondary index creation, as in initial data base load. The procedure creates new indexes for the data base that is being reorganized, rather than attempting to adjust existing indexes.

The IMS/VS reorganization facilities may also be used for structure reorganization, i.e., for reflecting changes to DBD parameters into the stored data without requiring a special user-written program. Changes that are confined to a single data base are achieved with a variation of the storage reorganization procedure in which the DBD of the data base being restructured is re-

Figure 17 Hierarchic direct storage reorganization



placed with a new DBD that will govern the reloading operation. In the HISAM procedure, changes are limited to data set record and block length changes, and to changes in the operating system access method. In the HD procedure (which can accept and produce HISAM as well as HIDAM and HDAM data bases), additional changes are possible, including the following:

- Data base implementation method.
- Pointer options.

- Record structure. (Segment types may be deleted from and added to the end of record type paths.)
- Segment attributes (length, nonkey fields).
- Indexing parameters.

Structure reorganization can be carried out at the same time as storage reorganization.

The IMS/VS reorganization facilities also permit limited changes to be made to the composition of a data base group. These changes are the addition of a new data base to a group (i.e., establishing a new logical relationship), and the addition of a new secondary index data base.

The addition of a new data base generally requires the structure reorganization of the data base(s) to which it will be related. Similarly, the addition of a new secondary index data base requires the reorganization of the data base that is being indexed. These changes can be effected with variants of the procedures shown earlier, in which DBDGEN, unload, and reload or initial load are performed for each affected data base.

The reorganization of DEDB data bases is accomplished with a root segment reorganization utility that reorganizes the root addressable and overflow areas of a data base. In addition, Fast Path provides a dependent scan utility program that copies a set of physically contiguous dependent segments to a sequential data set, and a dependent delete utility program that recovers the space occupied by such a set of segments.

Data base recovery

A data base may be damaged in a variety of ways, including read/write errors, physical damage to a volume, inadvertent erasing by an operator, and by an application program error. The effect of such loss on the user's installation can be minimized through the use of data base recovery facilities.

The basic approach to data base recovery in IMS/VS is to make periodic copies of the data sets that underlie the data base and record data base changes on the system log. In the event of failure in a data set, the latest copy can be updated with changes logged since the copy was made, thus restoring the data set to its condition at the point of failure.

A data base change is recorded in the system log in the form of two segment images: the segment as it appeared before the change, and the segment as it appeared after the change. Additional information recorded includes the identity of the program

that made the change, the date and time of entry, and the identity of the data base, data set, and record being modified.

The copying of data bases is done with a data base image copy utility program. That program copies one data set at a time, thereby creating an image copy of the data set on disk or tape. Data bases are normally copied just after the data base has been initially loaded (to obviate reloading in the case of failure), and immediately after reorganization. (Copies made before a reorganization cannot be used in recovery.) Copies may also be made at intermediate points, as determined by the update activity against the data base.

When data base damage is discovered, the affected data sets may be recovered by running a data base recovery utility program. For each data set to be recovered, the utility performs the following actions:

1. It allocates space for a new version of the data set and loads the latest image copy into this space.
2. It reads the system log in the forward (time ascending) sequence looking for changes that have been made to the data set after the image copy. For each such log entry, the "after" image is used to replace the corresponding data in the data set.

Since the log contains more information than is needed to recover data bases, provision is made for extracting and consolidating data base change entries from the log, in the interests of minimizing recovery time and the number of log tapes that must be retained. This function is provided by a data base change accumulation utility program. The program maintains a change accumulation data set that contains data base change entries in ascending chronological sequence within data set within data base. Input to the utility consists of the current change accumulation data set and one or more log tapes to be consolidated. Output is a new change accumulation data set in which changes to the same segment are consolidated, and all changes prior to a specified purge date (e.g., date of image copy) are deleted.

When recovery is required, the change accumulation data set may be input to the recovery utility. The change accumulation data set records are processed first, and then any unconsolidated log tapes are processed as before. Recovery of an MSDB is provided through an MSDB dump/recovery utility program that reconstructs a data base from the appropriate data base dump and the system log.

Summary and concluding remarks

IMS/VS provides the following two data structure classes: structures perceived by the installation data administrator (DA), and structures perceived by the application programmer (AP). The DA class is basically hierarchically organized but provides a logical relationship facility that avoids redundant data storage and permits the representation of many-to-many relations. The DA class also includes primary and secondary index structures. For implementing DA structures, a variety of implementation methods are provided, so that the user may select the methods giving the best performance for particular data bases. Implementation methods range from sequential storage to storage by randomizing algorithms, and even include the implementing of data bases in main storage.

Facilities are provided for the definition and creation of data bases. Creation is governed primarily by user-written load programs, with utilities provided for the integration of data bases into groups. Reorganization and recovery facilities are also provided through a number of utility programs.

GENERAL REFERENCES

1. IBM Corporation, *IMS/VS Version 1 System/Application Design Guide*, Document SH20-9025, IBM Corporation, Data Processing Division, White Plains, New York 10604.
2. IBM Corporation, *IMS/VS Version 1 Fast Path Feature General Information Manual*, Document GH20-9069, IBM Corporation, Data Processing Division, White Plains, New York 10604.
3. IBM Corporation, *IMS/VS Version 1 Utilities Reference Manual*, Document SH20-9029, IBM Corporation, Data Processing Division, White Plains, New York 10604.

Reprint Order No. G321-5047.