IMS Full Function Database Design Guidelines

Rich Lewis
IMS Advanced Technical Support
IBM Americas

February 2006

IMS Full Function Database Design Guidelines

Table of Contents

Introduction

What this Document Includes

This document is intended to assist users in designing and tuning IMS full function databases. It provides advice on choosing parameters in DBDGEN. It explains the considerations one should understand in making these choices.

This document covers the following database types:

- PHDAM (Partitioned Hierarchical Direct Access Method)
- PHIDAM (Partitioned Hierarchical Indexed Direct Access Method)
- PSINDEX (Partitioned Secondary Index)
- HISAM (Hierarchical Indexed Sequential Access Method)
- SHISAM (Simple Hierarchical Indexed Sequential Access Method)
- HDAM (Hierarchical Direct Access Method)
- HIDAM (Hierarchical Indexed Direct Access Method)
- INDEX

What this Document Does Not Include

This document does not provide advice on the logical structure of a database, such as the hierarchical structure of the database, the placement of fields within segments, and the keys that should be used. It assumes that these are determined by the application data and the requirements of the application design.

Logical databases and logical relationships are not covered in this document.

HSAM (Hierarchical Sequential Access Method) and SHSAM (Simple Hierarchical Sequential Access Method) are not covered. These are special purpose databases which cannot be updated. They can only be loaded and read.

Using this Document

This document contains recommendations and "rules of thumb." Use them as guidance for specifying parameters. You may have valid reasons for making other choices.

The DBDGEN Macro Cross Reference on page 8 includes tables for the DBDGEN macros. Each table includes references in this document for the parameters used with the macro. You may use it when coding DBDGEN statements to help you choose appropriate values.

You should use this document in conjunction with the standard IMS manuals, such as *Administration Guide: Database* and the *Utilities Reference: System*.

Acknowledgements

Bill Stillwell, Diane Goff, and Steve Nathan have provided valuable advice on the contents of this document. Their reviews and suggestions have greatly improved its contents and readability.

Terminology

The following terms are used in this document. No attempt has been made to include definitions of all of the terms used. For further reference, see the *IMS Glossary* at *http://www.ibm.com/software/data/ims/shelf/glossary.htm*.

CA: A VSAM control area. This is a set of control intervals (CI). Typically, a CA is equivalent to cylinder.

CI: A VSAM control interval. This is the unit of information that VSAM transmits to and from direct access storage. It is analogous to an OSAM block.

Database record: A database record is composed of a root segment and all of its dependent segments.

IDCAMS: IDCAMS is the program name for the Access Method Services utility. This utility is used to define VSAM data sets.

RAP: Root anchor point. Roots in PHDAM and HDAM databases are chained from RAPs. Each block in the root addressable area contains a fixed number of RAPs.

DBDGEN Macro Cross Reference

The following tables contain references to guidance for specifying DBDGEN parameters. Some parameters do not have references in this document. Typically, they either do not affect the database design or they are used with logical relationships. These include the specifications of names, logical relationships, and some exit routines. These exit routines do not affect database design. These parameters are indicated by "Not covered" in the Reference column.

Macro	Parameter	Reference
DBD	ACCESS=(x,)	HISAM vs. HD Access Methods, p. 12; (P)HDAM vs. (P)HIDAM, p. 17
	ACCESS=(,x)	OSAM vs. VSAM, p. 20
	ACCESS=(,,x)	Secondary index user integrity option; Not covered
	ACCESS=(,,,x)	INDEX DOSCOMP option; Not covered
	NAME	Database name; Not covered
	PASSWD	VSAM password option; Not covered
	EXIT	Data Capture exit; Not covered
	VERSION	User version specification; Not covered
	DATXEXIT	Data Conversion User exit; Not covered
	RMNAME	Randomization Parameters, p. 31
	PSNAME	Defining Partition Selection, p. 23

Table 1. DBD Macro

Macro	Parameter	Reference
DATASET		Multiple Data Set Groups, p. 43
	BLOCK	OSAM Block Size Specifications, p.25; VSAM ESDS CI Size Specifications, p. 25
	SIZE	OSAM Block Size Specifications, p.25; VSAM ESDS CI Size Specifications, p. 25
	RECORD	Index CI Sizes and Record Sizes, p. 24
	FRSPC	Free Space, p. 28
	SEARCHA	HD Space Search Algorithm, p. 29
	SCAN	SCAN= Parameter on the DATASET Statement, p. 42
	DD1	DD name for primary database data set; Not covered
	OVFLW	INDEX and HISAM overflow data set DD name; Not covered

Table 2. DATASET Macro

Macro	Parameter	Reference
SEGM	BYTES	Fixed Length vs. Variable Length Segments, p. 34
	POINTER or PTR	Pointer Options, p. 37
	RULES	Pointer Options, p. 37
	COMPRTN	Compression, p. 45
	DSGROUP	Multiple Data Set Groups, p. 43
	FREQ	FREQ Parameter on the SEGM Statement, p. 26
	NAME	Segment name; Not covered
	PARENT=((,x))	Defining Hierarchical, Physical Twin, and Physical Child Pointers, p. 40
	EXIT	Data Capture exit specifications; Not covered

Table 3. SEGM Macro

Macro	Parameter	Reference
LCHILD	POINTER or PTR	Direct vs. Symbolic Pointers, p. 50
	INDEX	Name of indexed field; Used only in INDEX and PSINDEX DBDs; Not covered
	NAME	Segment and database names for the associated segment: Not covered
	PAIR	Paired logical segment name; Used only for logical relationships; Not covered
	RULES	Rules for sequencing of logical children with nonunique sequence fields or with no sequence fields; Used only for logical relationships; Not covered
	RKSIZE	Root key size of target segment database record; Used only with PSINDEX DBDs; Not covered

Table 4. LCHILD Macro

Macro	Parameter	Reference
FIELD	NAME=(x,)	Name of field; Not covered
	NAME=(,x)	Sequence field specification; Not covered
	NAME=(,,x)	Unsequenced Dependent Segments, p. 39
	NAME=(systrelfldname)	Unique Keys vs. Non-unique Keys, p. 49
	BYTES	Size of field; Not covered
	START	Beginning location of field in segment; Not covered
	TYPE	Data type of field; Not covered

Table 5. FIELD Macro

Macro	Parameter	Reference
XDFLD	NAME	Name given to search field for a secondary index; Not covered
	SEGMENT	Index source segment name; Not covered
	CONST	Shared Secondary Indexes, p. 51
	DDATA	Duplicate Data, p. 51
	NULLVAL	Sparse Indexing, p. 52
	EXTRTN	Sparse Indexing, p. 52
	SUBSEQ	Unique Keys vs. Non-unique Keys, p. 49
	SRCH	Secondary Index Keys, p. 49

Table 6. XDFLD Macro

HALDB

High Availability Large Database (HALDB) was introduced in IMS Version 7. It added three additional types of full function databases. These are PHDAM, PHIDAM, and PSINDEX. Application programs that use non-HALDB databases work without change when the databases are migrated to HALDB. HALDB provides three major advantages over non-HALDB full function databases:

1. Larger database capacity

   HALDB databases can be spread across 1 to 1001 partitions. Each partition may have 1 to 10 data sets. Each data set may be up to 4 gigabytes. This means that HALDB databases have a maximum capacity of over 40 terabytes.

2. Shorter off-line reorganization times

   HALDB partitions may be reorganized independently and in parallel. When you reorganize a partition or database, you do not have to rebuild or update the secondary indexes that point to it. Similarly, you do not have to update any logically related databases. IMS automatically updates the secondary index pointers and logical relationship pointers when they are first used following the reorganization. These updates use the HALDB "self-healing pointer" process. The combination of reorganizing partitions in parallel and the self-healing pointer process can greatly reduce the elapsed time required for reorganizing HALDB databases with off-line processes.

3. Online reorganization

   HALDB online reorganization allows an IMS online system to reorganize HALDB partitions while applications read and update the partitions. The applications can run in the same IMS online system or in data sharing subsystems. HALDB online reorganization was introduced in IMS Version 9.

The Redbook, *The Complete IMS HALDB Guide*, provides information on migrating databases to HALDB, defining their partitions, and maintaining them. It is available at http://www.redbooks.ibm.com/abstracts/sg246945.html.

HALDB Recommendation

Installations which have database requirements answered by HALDB should convert those databases to HALDB. You do not need to convert existing databases which do not require greater capacities and do not need shorter reorganization outages. IMS intends to support non-HALDB full function database types for the foreseeable future. On the other

hand, enhancements for full function databases will be concentrated in HALDB. For this reason, you should create new IMS full function databases as HALDB types.

HISAM vs. HD Access Methods

IMS full function databases may use either the Hierarchical Indexed Sequential Access Method (HISAM) or one of the Hierarchical Direct (HD) access methods.

The HISAM database types are:

- Hierarchical Indexed Sequential Access Method (HISAM)
- Simple Hierarchical Indexed Sequential Access Method (SHISAM)

The HD access method database types are:

- Partitioned Hierarchical Direct Access Method (PHDAM)
- Partitioned Hierarchical Indexed Direct Access Method (PHIDAM)
- Hierarchical Direct Access Method (HDAM)
- Hierarchical Indexed Direct Access Method (HIDAM)

HISAM and the HD access methods differ primarily in the way that they store segments and use space in their data sets.

HISAM

HISAM stores segments in two data sets. These are the primary data set and overflow data set. The primary data set is a KSDS. The overflow is an ESDS. For each database record, HISAM places the root and some dependents in a logical record in the primary data set. The number of dependents that HISAM stores in the primary data set depends on the size of the logical record that you define. HISAM places dependent segments which do not fit in the index in the overflow data set. The dependents may require multiple logical records in overflow.

Your choices for the logical record sizes of the primary and overflow data sets can have a significant effect on database performance. Ideally, you want IMS to store a database record only in the primary data set. This eliminates I/Os to the overflow data set. If the database records are uniform in size and not too large, your choice is simplified. You can make the logical record size of the primary data set large enough to hold a database record without wasting a lot of space. If database record sizes vary greatly, a primary record size large enough to hold the larger database records would waste space for others.

Each logical record in the overflow data set contains segments from only one database record. Any free space in the logical record can only be used for inserts into the same database record. This can make the sizing of logical records difficult when database records vary in size. Large logical record sizes tend to waste space. Small sizes tend to spread database records over more logical records. This requires IMS to do more I/Os to process the database.

HISAM does not use pointers to navigate between segments in a database record. It always stores segments in hierarchical sequence. IMS accesses a segment by reading all of the segments from the root to the required segment. This can be disadvantageous with large database records. Large records may require IMS to read multiple logical records in the overflow data set.

HISAM does not reuse space that is occupied by deleted dependent segments. These segments remain in the data sets when they are deleted. IMS sets a bit in their prefix to indicate that they have been deleted. This has two effects. First, IMS may have to read through these segments to reach other segments. Second, later inserts cannot use space created by these deletions. Instead, they have to expand the database. You must reorganize the database to recover the space.

Inserts of roots into a HISAM database require a new logical record in the primary data set. If a logical record is not available in the CI, a CI split is required.

Inserts of dependent segments may require updates to multiple CIs. Since the segments are maintained in hierarchical sequence, the insert of a segment in the middle of a database record requires the movement of the following segments. If there is not room at the end of the last logical record used by the database record, new dependents require a new logical record. This logical record is at the end of the overflow data set. If there are high volumes of inserts, there tends to be a high volume of activity at the end of the data set. IMS must extend the data set frequently.

Replaces of variable length segments often require the movement of other segments. If the size of the variable length segment changes, the segments which follow it in the database record must be moved. Similar considerations apply to fixed length segments when they are compressed. Compressed fixed length segments are actually variable length when they are stored. An explanation of compression appears on page 45.

HISAM does not support multiple data set groups. You may define only two data sets, the primary (KSDS) and the overflow (ESDS), for a HISAM database. See page 43 for an explanation of multiple data set groups.

You select HISAM by specifying ACCESS=HISAM on the DBD statement.

Recommendations for Using HISAM

HISAM is best suited for databases with the following characteristics:

1. Database records are relatively small. They typically require only one logical record in the primary data set (KSDS). Occasionally, they might require one logical record in the overflow data set (ESDS).
2. Database records are relatively uniform is size. There is not a large range in size for most database records.

3. The database is stable. There are not high volumes of deletes and inserts in the database.
4. The database will not grow past the data set size limitations of 4 gigabytes per data set.

SHISAM

Simple HISAM (SHISAM) is a simplified version of HISAM. SHISAM databases have the following restrictions:

- They have no dependent segments. Only root segments are allowed.
- They cannot have secondary indexes.
- They cannot have logical relationships.
- All segments are fixed length. They do not support variable length segments.
- Compression is not allowed.

A SHISAM database is stored in a KSDS. The segments do not have an IMS prefix. Since they do not have an IMS prefix, you can process a SHISAM database as a KSDS without using DL/I calls. Conversely, you can process a KSDS with fixed length records as a SHISAM database using DL/I calls.

Since SHISAM segments have no prefix and SHISAM requires only one data set, SHISAM makes efficient use of space. It requires less space than a root-only HISAM, HIDAM, or PHIDAM database.

You select SHISAM by specifying ACCESS=SHISAM on the DBD statement.

Recommendations for Using SHISAM

If you have a database which meets the SHISAM restrictions, you should consider using SHISAM for it. Additionally, if you need to process a KSDS with fixed length records as an IMS database, you may define it as SHISAM.

HD Access Methods

HD access methods store segments in one or more data sets. They may use either OSAM or VSAM ESDS for these data sets. HIDAM and PHIDAM databases also include indexes. The indexes point to the root segments which are stored in the ESDS or OSAM data sets.

HD access methods and HISAM have different techniques for navigating between segments. HD uses pointers which are stored in the prefix of segments. A prefix may contain separate pointers to different segment types. This allows IMS to more directly

navigate to dependent segments. Figure 1 illustrates this. Segment A has pointers to the first B segment, the first C segment, and the first D segment. IMS does not have to access any of the B or C segments to reach the D segments.

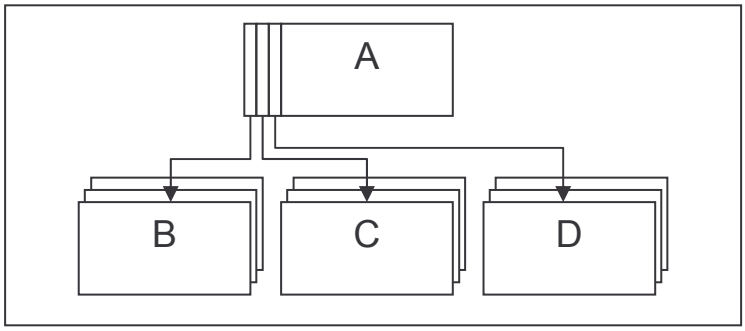


Figure 1. HD Pointers

The use of direct pointers usually makes HD the preferable access method for databases with large database records. It is easier for IMS to navigate to individual segments within the database.

HD is the preferable access method for highly volatile databases with many deletes and inserts. HD reuses the space occupied by deleted segments. When you delete segments, their space is freed. Inserts may place segments in this free space.

CI splits can occur when you insert roots into HISAM, HIDAM and PHIDAM databases. But there is a significant difference in the likelihood of splits with the different database types. The logical records for HISAM KSDSs are much larger than those for HIDAM and PHIDAM. The logical records for HISAM contain the root segment and, typically, several dependents. HIDAM and PHIDAM indexes contain only the key, a delete byte, and a four-byte pointer to the root segment. Since HISAM logical records are much larger, fewer will fit in a CI. This affects the probability of CI splits. The following example illustrates this point. Consider a HISAM database whose primary data set has 1K logical records and an 8K CI size. Twenty-five percent free space would provide room for only two inserts before a split would be required. Compare that with a HIDAM or PHIDAM database. If the key were 10 bytes, each logical record would be only 15 bytes. Twenty-five percent free space in an 8K CI would provide room for 136 inserts before a split would be required. HIDAM and PHIDAM are typically better choices for databases with many inserts of root segments.

Replaces of variable length segments do not require the movement of other segments in HD databases. If the segment grows so that it does not fit in its previous location, IMS stores the data portion of the segment elsewhere. The segment's prefix does not move. Pointers to this segment from other segments are not changed. A pointer to the data portion is added to the prefix. This technique minimizes the effect on the rest of the database for these types of replaces.

HD supports multiple data set groups. They give you greater flexibility in handling varying space requirements and access patterns for different segment types. See page 43 for an explanation of multiple data set groups.

HD includes HALDB PHDAM and PHIDAM databases. These databases can contain up to 1001 partitions and hold up to 40 terabytes of data.

You select HD by specifying HDAM, HIDAM, PHDAM, or PHIDAM on the ACCESS= parameter of the DBD statement.

Recommendations for Using HD Access Methods

HD is well suited for databases with the following characteristics:

1. Database records have great variations in their sizes.
2. Database records are large and contain many types of segments.
3. There are high volumes of deletes and inserts in the database.
4. The database may have significant growth.

(P)HDAM vs. (P)HIDAM

PHDAM, HDAM, PHIDAM, and HIDAM databases share some of the same characteristics. They use direct pointers to navigate between segments in a database record. When segments are deleted, the space they occupied is available for the insertion of other segments. The primary difference between the types is that PHIDAM and HIDAM have indexes. This allows you to easily process the databases in root key sequence. PHDAM and HDAM use randomizing modules to determine where root segments are stored. IMS does not store or retrieve their roots in key sequence.

Space Use

There are some important distinctions in how space is used by PHDAM and HDAM versus PHIDAM and HIDAM. PHIDAM and HIDAM tend to use less space. When they are reorganized, the database records are written in sequence leaving only the specified free space. When PHDAM and HDAM are reorganized, the database records are chained from their root anchor points (RAPs). These are spread across the root addressable area (RAA). If the database records vary in size, it may be difficult to create free space that is evenly spread across the RAA. More information and advice on free space appears under Free Space on page 28. Since PHIDAM and HIDAM tend to use less space, batch jobs which process the entire database typically perform better with these database types.

Sequential Processing

PHIDAM and HIDAM databases allow you to access database records in key sequence. If you require key-sequential access for a PHDAM or HDAM database, you may use a secondary index where the secondary index key is the root key. Applications which require sequential processing may use the secondary index when accessing the database. This allows IMS to use space according to the rules of PHDAM and HDAM databases while allowing applications to access the databases in key sequence. It is also possible to create a "sequential randomizer" which places roots in the PHDAM or HDAM database in root key sequence. You may use the IBM product, *IMS Sequential Randomizer Generator*, to create sequential randomizers.

I/Os

PHIDAM and HIDAM databases have indexes. When you request a root segment by its key, IMS reads the index to find the location of the key. This processing is not required with PHDAM and HDAM databases. For them, IMS invokes the randomizer to find the root anchor point (RAP) from which the root is chained. Typically, IMS does more processing to retrieve or insert PHIDAM and HIDAM roots. This tends to give PHDAM

and HDAM a performance advantage. Of course, this assumes that the PHDAM partition or HDAM database is reasonably well organized.

Reorganizations

Most databases have to be reorganized occasionally. The time between reorganizations usually depends on the type of insert and delete activities, the distribution of free space, and the database type. PHIDAM and HIDAM databases differ from PHDAM and HDAM databases in their reorganization characteristics. If a PHIDAM or HIDAM database has large amounts of inserts in a range of root keys without deletes in the same range, it is likely to need reorganization. This is not necessarily true with PHDAM and HDAM. Since randomizers spread the roots across the partitions or databases, activity for a range of keys tends to be spread across the entire PHDAM partition or HDAM database. PHDAM and HDAM tend to be better choices for these types of databases.

Creeping Keys

The root keys in some applications increase in value over time. An example is a key based on the time of the insertion of the root. Such keys are called “creeping keys.”

For HDAM databases this does not have special significance. HDAM spreads the roots randomly across the root addressable area. Deleted segments will typically make room for new segments. Space use tends to be randomly distributed across the root addressable area.

For PHDAM databases using key range partitioning, space use tends to be randomly distributed across the root addressable areas of the partitions. New roots with creeping keys are placed in the last partition. You may have to create new partitions for new time periods.

You may want to avoid the requirement to add new partitions with PHDAM. If the total amount of data in the database does not increase, but the key values increase, you may want to use a partition selection exit routine instead of using key range partitioning. The exit routine can assign keys to partitions by using only a low order subset of the key. For example, if the key were 8 numeric digits and you wanted ten partitions, you could use the low order digit to assign the key to a partition. Key nnnnnnn1 could be assigned to the first partition; key nnnnnnn2 could be assigned to the second partition; and so on with key nnnnnnn0 assigned to partition 10. Assuming that the values of the low order digit were evenly distributed from 0 to 9, the keys would be evenly distributed across the partitions. Since the database is PHDAM, deletes would create usable space for future inserts.

For HIDAM databases IMS places new roots with creeping keys near the last ones inserted. This is at the end of the used space in the data set. When you delete the oldest database records, you make space near the beginning of the data set. IMS typically cannot use this space for new insertions. In this sense, the space used in the data set tends to creep from the front to the end of the data set. The database will likely require reorganizations. HIDAM with creeping keys has a second disadvantage. The inserts of new keys are into the last CI in the primary index. This causes CI/CA splits and disorganization of the index. This is a second reason why these databases likely require frequent reorganizations.

PHIDAM is similar to HIDAM. With key range partitioning, all inserts for creeping keys are made into the last partition. Space used in the partition data set tends to creep from the front to the end. You may need to add new partitions occasionally. This starts the process again for the new partition. With a partition selection exit routine, new roots might be placed in any of the partitions. This does not eliminate the problem of the creeping use of the data sets. New roots in a partition will always go into the end of the partition. The creeping occurs in parallel across all the partitions.

PHDAM or HDAM is often a better choice than PHIDAM or HIDAM for databases whose root key values increase over time. They eliminate the problem of creeping space use. They may substantially reduce the frequency with which you must do reorganizations. Of course, PHDAM or HDAM do not maintain roots in key sequence. If you must process a database in key sequence, you may be able to use PHDAM or HDAM with a secondary index. The secondary index can provide the sequencing you need. The secondary index might suffer the effects of creeping keys, but the space use in the database would not creep.

Recommendations Summary for (P)HDAM vs. (P)HIDAM

PHIDAM and HIDAM have the following advantages over PHDAM and HDAM:

1. They tend to use less space. This provides a performance advantage for batch jobs which sequentially process the entire database.
2. They allow you easily to process a database in root key sequence.

PHDAM and HDAM have the following advantages over PHDAM and HDAM:

1. They tend to require fewer I/Os to retrieve and insert root segments.
2. They tend to require fewer organizations when update activity is concentrated in a range of keys.
3. They tend to handle space management with creeping root keys better.

OSAM vs. VSAM

Some IMS full function database data sets may be either OSAM or VSAM ESDSs. These are HDAM data sets, PHDAM database data sets, HIDAM database data sets, and PHIDAM database data sets. You cannot use OSAM for HIDAM indexes, PHIDAM indexes, secondary indexes, or HALDB ILDSs. These data sets must be VSAM KSDSs.

OSAM is the preferred access method whenever it may be used. OSAM has several advantages.

- OSAM has a superior way of writing multiple blocks. When an application sync point occurs, IMS sorts the altered OSAM blocks by physical location within volumes. IMS writes blocks which are on the same volume with chained writes. This reduces the processor time required on the z/OS system. IMS does the chained writes for different volumes in parallel. This reduces the elapsed time for the writes.

- OSAM has a shorter processor instruction path length for most of its processing. This is especially significant in online systems. This means that OSAM typically uses less processor time than VSAM.

- OSAM has OSAM Sequential Buffering (OSB). VSAM does not have a capability similar to OSB. When OSB is invoked, IMS does anticipatory reads. For sequential processes, OSB reads blocks into buffers before the application requires them. This eliminates the wait time for reads and can significantly reduce the elapsed time for sequential processing.

- For HDAM and HIDAM, OSAM data sets may be up to 8 gigabytes in size. HDAM and HIDAM ESDSs are limited to 4 gigabytes.

- You may reuse OSAM data sets. When you reorganize a database, you do not have to delete and redefine OSAM data sets. When you reorganize HDAM and HIDAM databases, you must delete and redefine VSAM data sets. This VSAM requirement does not apply to PHDAM and PHIDAM databases.

There is a warning about the reuse of OSAM data sets. You should not reuse multivolume OSAM data sets. If you do not scratch and reallocate a multivolume OSAM data set before reusing it, an invalid end-of-file mark may be left on the last volume of the data set. This can lead to lost data. You can use HALDB to avoid database data sets that are so large that they must span multiple volumes.

You select OSAM or VSAM by specifying OSAM or VSAM on the second subparameter of the ACCESS= parameter on the DBD statement. VSAM is the default.

Recommendations Summary for OSAM vs. VSAM

Use OSAM data sets, not VSAM ESDSs, with PHDAM, PHIDAM, HDAM, and HIDAM databases.

HALDB Partition Selection

HALDB databases have one to 1001 partitions. IMS assigns database records to partitions based on the key of the root segment. This assignment is done as part of partition selection. You specify how partition selection is done. You tell IMS to use either key ranges or a partition selection exit routine. Partition selection also determines the order in which partitions are processed by sequential processing.

Key Range Partition Selection

Most HALDB databases use key range partition selection. In this method, you assign a high key to each partition. IMS assigns roots within a key range to each partition. The partitions are ordered in the order of the high keys. With PSINDEX and PHIDAM databases, sequential processing retrieves all the roots in key sequence. With PHDAM, you assign a range of keys to a partition, but the randomizer determines the order of the roots within a partition.

Partition Selection Exit Routine

You can write a partition selection exit routine for any type of HALDB database. Your exit routine assigns root keys to partitions. It also determines the order of the partitions. IMS passes the key of the root segment and information about all of the partitions to the exit routine. The routine uses the key to assign the root to a partition.

These exit routines are typically used to assign database records to partitions based on a subset of the key. For example, part of the key may be a country code, department code, or similar bit of information. You could use this to group data from each country or department into its own partition. Application processing requirements usually determine if you should use this type of partitioning. For example, you might want to process only the data for a particular country or department with certain jobs. It would be more efficient if only the database records for that country or department resided in a partition. Then your job would need to process only one of the partitions in the database.

Another use of these exit routines is to spread activity across multiple partitions. A typical example is a database where each new root has a higher key than the existing roots in the database. Key range partitioning would place these new roots in the last partition. You could use a partition selection exit routine to spread the insert activity across all of the partitions. This is further explained under Creeping Keys on page 18.

You should be careful about choosing a partition selection exit routine for PSINDEX and PHIDAM databases. The exit routine typically will not assign roots to partitions in key order. If you have chosen PHIDAM so that you can process the database in key sequence

order, you cannot use an exit routine which assigns keys by other than key range. Similarly, if you have created the secondary index for secondary index key sequence processing, you cannot use an exit routine which assigns secondary index keys by other than key range.

Defining Partition Selection

A partition selection exit routine is defined with the PSNAME parameter on the DBD statement. The parameter specifies the name of the exit routine. If the PSNAME parameter is not included, key range partitioning is implied. The specification on the DBD may be changed during partition definition or with a DBRC command. The HALDB Partition Definition utility includes a "Part. Selection Routine" field in the "Master Database values" section. If you specify a name, it will be used as the partition selection exit routine name. If you make the field blanks or nulls, key range partitioning is used. The INIT.DB and CHANGE.DB DBRC commands include PARTSEL and HIKEY parameters. PARTSEL specifies the name of a partition selection exit routine. HIKEY specifies that key range partitioning will be used. These parameters override the specification on the DBD.

Recommendations Summary for HALDB Partition Selection

The recommendations for partition selection are:

1. Key range partitioning is appropriate for most databases
2. You may use a partition selection exit routine to group data into partitions by a subset of the root key.
3. You may use a partition selection exit routine to spread activity across all of the partitions in the database.
4. Do not use a partition selection exit routine for a PHIDAM database which you must process in key sequence order.
5. Do not use a partition selection exit routine for a secondary index which you will use for processing a database in secondary key order.

Block Sizes, CI Sizes, and Record Sizes

You specify the block sizes of OSAM data sets, CI sizes of VSAM data sets, and record sizes of KSDSs when you create databases. This section explains the requirements and recommendations for these sizes.

Index CI Sizes and Record Sizes

PHIDAM primary indexes, HIDAM primary indexes, secondary indexes, and HISAM databases use VSAM KSDSs. This section discusses record sizes and CI sizes for these KSDSs.

You do not need to specify the RECORD parameter on the DATASET statement for HIDAM primary indexes and secondary indexes. IMS automatically calculates the record sizes for these data sets during DBDGEN. They are reported in the output listing of DBDGEN. You should use these sizes for the RECORDSIZE parameter on the IDCAMS DEFINE statements when you define the data sets. If you specify a DATASET macro RECORD parameter smaller than the calculated size, it will not hold the index record and the index cannot be loaded. If you specify a RECORD parameter larger than the required size, you waste space in the data set.

The appropriate record sizes for PSINDEX data sets are reported in the DBDGEN output for these secondary indexes. You should use these sizes for the RECORDSIZE parameter on the IDCAMS DEFINE statements when you create these data sets.

You must specify the RECORD parameter on the DATASET statement for HISAM databases. As mentioned under HISAM on page 12, ideally a record in the primary data set should be large enough to contain an entire database record. In some cases this could waste a lot of space. This is true when database records vary significantly in size. Small records will use only a small amount of the VSAM record. In this case, the primary data set record size should be large enough to hold the majority of database records. The overflow data set record size should be large enough to hold the remainder of most other database records. Record sizes are specified on the RECORD parameter of the DATASET macro.

The CI size for the data component of index data sets (KSDSs) determines how many logical records are stored in a CI. Large CI sizes tend to be good for sequential processing. Multiple records are read or written with one I/O. Large CI sizes may not be a benefit for random processing. Typically, random processing accesses only one record per CI. For HISAM databases and indexes where sequential processing is most important, CI sizes of at least 8K should typically be used.

The size of the KSDS index component CI can be either explicitly specified in your IDCAMS DEFINE statement or allowed to default. The default causes VSAM to calculate the index CI size. If you use z/OS 1.3 or a later release, you should use the default calculation. It optimizes the index component CI size.

OSAM Block Sizes and VSAM ESDS CI Sizes

OSAM block sizes and VSAM ESDS CI sizes affect the number of I/Os required to process a database and the sizes of buffers in IMS systems. Typically, block sizes and CI sizes should be large enough to hold entire database records. When database record sizes are very large or vary greatly, this may not be possible. Then you should attempt to make the block or CI size large enough to hold the frequently accessed segments in the database records. For sequential processing, large sizes are typically good. They reduce the number of I/Os that are required to process the database. Block or CI sizes of at least 8K should typically be used for databases which will be used with sequential processing. For random processing, large sizes are not as beneficial. They increase the overhead of I/O processing and the space required for database buffers. If a complete database record is stored in a small block or CI, there may be no benefit in using a larger size.

OSAM Block Size Specifications

For PHDAM and PHIDAM databases you specify OSAM block sizes during partition definition. You do this either with the "Block Size" field in the HALDB Partition Definition utility or with the BLOCKSZE parameter on the DBRC INIT.PART or CHANGE.PART commands. For HDAM and HIDAM databases you specify block sizes with the SIZE parameter on the DATASET statement of DBDGEN. Alternatively, DBDGEN will calculate these block sizes from the specification of the BLOCK parameter on the DATASET statement. The value specified on the BLOCK parameter includes only space that is used for segments. DBDGEN adds space for a FSEAP and HDAM RAPs to calculate the block size from the BLOCK specification. Most users specify SIZE rather than BLOCK since SIZE is the actual block size.

VSAM ESDS CI Size Specifications

For VSAM database data sets IMS always uses the CI size specified in the RECORDSIZE parameter of the IDCAMS DEFINE statement. It does not use the CI size specified by the SIZE or BLOCK parameters on the DATASET statement of DBDGEN. For documentation reasons it is recommended that you specify a SIZE parameter which matches the CI size that you will use. This could eliminate confusion which might occur if the size in the DBDGEN differed from that actually used.

If you want to change the CI size for a VSAM data set, it is not necessary to do a DBDGEN. You only have to change the CI size in the IDCAMS DEFINE before reloading the database.

HALDB partition definition does not include a capability to specify VSAM CI sizes. They are specified only with IDCAMS DEFINE statements.

FREQ Parameter on the SEGM Statement

The SEGM statement for HISAM, SHISAM, INDEX, and PSINDEX databases includes the FREQ parameter. This parameter is optional. You do not need to specify it. It is ignored for PSINDEX, INDEX, and SHISAM databases. The FREQ parameter specifies the frequency of a segment in a HISAM database. This is the average number of these segments per its parent. For root segments, the documentation states that FREQ is the number of roots in the database; however, specifying FREQ for roots does not affect the output of DBDGEN.

For HISAM, DBDGEN uses the FREQ specifications on dependent segments to calculate suggested logical record sizes and CI sizes. These are listed in the output of DBDGEN. DBDGEN always produces the same suggestions for the prime and overflow data sets. If you know the distribution of your database record sizes, you can choose logical record and CI sizes which are better for your databases. If you do not know them, you are unlikely to know the frequency of segment occurrences. For these reasons, you do not need to specify this parameter.

You never need to specify the FREQ parameter. IMS ignores it for PSINDEX, INDEX, and SHISAM databases. It is not valid for PHDAM, PHIDAM, HDAM, and HIDAM databases.

Recommendations Summary for Block Sizes, CI Sizes, and Record Sizes

The recommendations for block sizes, CI sizes, and record sizes are:

1. Do not specify the RECORD parameter on the DATASET statement for INDEX databases (secondary indexes and HIDAM primary indexes).
2. Always use the output listing of DBDGEN to determine the RECORDSIZE parameters to use for the IDCAMS DEFINE for KSDSs. These include data sets for PSINDEX, INDEX, PHIDAM, HIDAM, and HISAM databases.
3. If you use z/OS 1.3 or a later release, you should specify the data component CI size for KSDSs. Do this with an IDCAMS DEFINE statement. Do not specify the index component CI size. Let the system calculate it.

4. The CI size for VSAM ESDS and KSDS data sets is not controlled by DBDGEN parameters. It is determined by the RECORDSIZE parameter on the IDCAMS DEFINE statement. To avoid confusion, specify the SIZE parameter on the DATASET statement for VSAM data sets to match the CI size specified with IDCAMS.
5. For HDAM and HIDAM OSAM data sets specify the SIZE parameter on the DATASET statement, not the BLOCK parameter.
6. Do not specify the FREQ parameter on SEGM statements.

Free Space

The Effects of Free Space

Free space in a database affects the processing done by IMS to retrieve and insert segments. Free space in a database increases the likelihood that an insert of a segment will go in the same block as the segment from which it is chained. This tends to reduce the number of I/Os required for inserting and retrieving segments. On the other hand, adding free space increases the number of blocks that must be read for sequential processing since it spreads data across more blocks.

Free space is created when a database is initially loaded, when it is reorganized, and when segments are deleted.

Specifying Free Space

Free space in HIDAM databases is specified with the FRSPC parameter on the DATASET statement. The first subparameter specifies the 'free block frequency factor' (fbff). A value of 'n' specifies that every nth block in the data set is left as free space by a load or reorganization. The second subparameter specifies the 'free space percentage factor' (fspf). A value of 'n' specifies that n percent of each block is left as free space by a load or reorganization. Different data sets in the same database may have different free space specifications.

Free space in PHIDAM databases is specified in the partition definitions. These parameters are the 'free block frequency factor' and the 'free space percentage factor' that are described in the previous paragraph. Each data set in a partition uses the same free space parameters. The values are specified either in the HALDB Partition Definition Utility or with the DBRC INIT.PART command FBFF and FSPF parameters.

Free space in HDAM and PHDAM databases may be specified by using the same parameters used with HIDAM and PHIDAM. You should not do this. Instead, you should use other parameters to control how space is used in HDAM and PHDAM databases. If you specify a free block frequency factor, a reorganization cannot place any database records in the free blocks even though some records would be chained from the RAPs in these blocks. If you specify a free space percentage factor, each block in the RAA would have space that could not be used by a reorganization. Some root segments as well as their dependents could be forced to the overflow area. A better way to specify the free space characteristics for these databases is through the use of randomization parameters. Randomization parameters are explained below.

Free space in PHIDAM primary indexes, HIDAM primary indexes, and secondary indexes is specified with the FREESPACE parameter on the IDCAMS DEFINE

statements. It is not defined in DBDs or HALDB partition definitions. The FREESPACE parameter has two subparameters. The first specifies the percentage of free space in each CI. The second specifies the percentage of free space in each CA. You should specify free space when you expect to insert new root segments in PHIDAM and HIDAM databases or when you expect to insert new secondary index entries. The recommended amount of free space depends on the amount of inserts. For example, if you expect to insert 5% more root segments in a HALDB partition between reorganizations of the index, you should specify at least 5% free space in the CIs. You should also specify free space in the CAs. This creates empty CIs in the CA which are available for splitting CIs without forcing the split of the CA. Free space is created uniformly across the data set. If the inserts will be concentrated in a small part of the key range, free space may not be beneficial. In this case, most of the free space will not be used. Instead, VSAM will have to continually split CIs and CAs to create room for the new entries. Unfortunately, you cannot do much to reduce these splits.

HD Space Search Algorithm

IMS has an HD space search algorithm. IMS uses this algorithm when inserting segments in PHDAM, PHIDAM, HDAM, and HIDAM databases. The algorithm is documented in the *IMS Administration Guide: Database* under "How the HD Space Search Algorithm Works." In general, IMS attempts to place a new segment in the same block or CI with the segment or RAP from which it is chained. If there is not space in this block or CI, IMS may attempt to place the segment in the second most desirable block. This is the nearest block or CI that was left free when the database or partition was loaded or reorganized. The search does not use the second most desirable block in two cases. First, if free block frequency is not specified in the database or partition definition, there is no second most desirable block. Second, HDAM and HIDAM DBD definitions can override the use of the second most desirable block in the space search. This is done by specifying SEARCHA=1 on the DATASET macro. Since the purpose of specifying the free block frequency factor is to create free blocks or CIs for later inserts, SEARCHA=1 should not be specified. This parameter is included to provide compatibility with old IMS releases which did not have the second most desirable block step in the HD space search algorithm.

There is no option equivalent to SEARCHA=1 for PHDAM and PHIDAM databases. They always include the search of the second most desirable block when the free space percentage factor is specified.

Recommendations Summary for Free Space

The recommendations for free space are:

1. Use the ‘free block frequency factor and/or the ‘free space percentage factor’ to create free space in HIDAM and PHIDAM databases.

2. Use randomization parameters to create free space in HDAM and PHDAM databases. Do not specify fbff and fspf for them.
3. Use the IDCAMS DEFINE FREESPACE parameter to create free space in PHIDAM primary indexes, HIDAM primary indexes, and secondary indexes.
4. Specify SEARCHA=0 on the DATASET macros for HDAM databases. This is the default.

Randomization Parameters

Randomizers and their parameters are specified in the DBD for HDAM. The DBD for PHDAM only specifies the default values. The definition of the PHDAM partition specifies the values used for the partition. Different partitions in a PHDAM database may have different randomization parameter values.

The randomization parameters on the DBD statement have the following syntax:

RMNAME=(mod, anch, rbn, bytes)

- *mod* is the randomizer module name. It must be specified.
- *anch* is the number of root anchor points per block. It defaults to 1. The maximum value is 255.
- *rbn* is the number of blocks in the root addressable area. It must be specified.
- *bytes* specifies the maximum number of bytes of a database record that may be stored in the root addressable area in a series of inserts without a call to another database record. The default is no maximum.

For PHDAM these values may be defined with the HALDB Partition Definition Utility (PDU) or the DBRC INIT.PART or CHANGE.PART commands. The DBRC command parameters are:

- RANDOMZR(name) is the randomizer module name.
- ANCHOR(value) corresponds to the *anch* parameter in the DBD.
- HIBLOCK(value) corresponds to the *rbn* parameter in the DBD.
- BYTES(value) corresponds to the *bytes* parameter in the DBD.

Randomizer

IMS supplies several randomizers. DFSHDC40 is appropriate for most databases and HALDB partitions. Its use is recommended unless you have specific knowledge of your key distribution and you DO NOT want the keys randomly spread across the RAPs. DFSHDC40 spreads roots randomly across the RAPs in a HDAM database or a PHDAM partition. You do not need to do any analysis of key distributions when you use DFSHDC40.

DFSHDC40 converts the key into a randomly chosen four-byte binary number. This number is chosen independently of the number of RAPs in the database or partition. Then DFSHDC40 multiples this number by the number of RAPs in the database or partition and divides the result by the maximum value. This assigns keys to RAPs in an order that is unaffected by the number of RAPs. So if you unload a database or partition and change the number of RAPs, the reload will still be a sequential process.

In some cases you may want to use a different randomizer. This should only occur when you have knowledge of the key distribution and you do not want a random distribution of the keys across the database or partition. For example, you might want to assign the keys to RAPs in key sequence. The IBM product, *IMS Sequential Randomizer Generator*, can create randomizers that do this.

Number or RAPs

The number of RAPs for a HDAM database or PHDAM partition is the product of the number of RAPs per block times the number of blocks in the root addressable area. When more RAPs are used, the probability of a long chain of roots from a RAP is diminished. A rule of thumb is that the number of RAPs in a database or partition should be at least twice the number of roots. When DFSHDC40 is used, the roots are distributed across the RAPs in a Poisson distribution. When the number of RAPs is twice the number of roots, the distribution has the following characteristics:

- 79% of roots will be the first root on their RAP chain.
- 18% of roots will be the second root on their RAP chain.
- 3% of roots will be the third root on their RAP chain.
- 0.38% of roots will be past the third root on their RAP chain.

When the number of RAPs is three times the number of roots, the distribution has the following characteristics:

- 85% of roots will be the first root on their RAP chain.
- 13% of roots will be the second root on their RAP chain.
- 1.5% of roots will be the third root on their RAP chain.
- 0.13% of roots will be past the third root on their RAP chain.

Since a RAP is only four bytes, the space used by RAPs is rarely significant.

Size of Root Addressable Area

The root addressable area (RAA) is the set of OSAM blocks or VSAM CIs which hold RAPs. Other blocks in the data set contain the overflow area. When the RAA is too small, the segments in a database record are often stored in overflow. Reading them requires reading the RAP block and reading blocks in the overflow area. When the RAA is too large, there is unnecessary unused space in the RAA. A sequential read of the database or partition has to read more blocks. A rule of thumb is that the RAA should be 35% larger than the size of all of the segments in the HDAM database or PHDAM partition. This creates approximately 25% free space. Larger sizes may be desirable because they may help avoid the need for reorganizations. They increase the probability

that IMS will place segments in the block containing the RAP from which they are chained.

The BYTES parameter

The *bytes* parameter specifies the maximum number of bytes of a database record that may be stored in the root addressable area in a series of inserts without a call to another database record. This includes inserts that are done by reorganizations. When you reorganize a database or partition any segment which causes the database record to exceed this parameter is placed in overflow. This reserves space for segments of other database records. It helps avoid situations where exceptionally large database records force other database records into overflow. A rule of thumb is that the *bytes* parameter should be twice the average database record size. This assumes that all of the segments in the database are in the first data set group. This rule of thumb may not apply to some databases. If you are aware of your database record size distributions, you may want to use other values for *bytes*. The *bytes* parameter does not affect any segments which are in other data set groups.

Specifying Randomization Parameters for PHDAM

The randomization parameters used for PHDAM partitions are defined in the partition definitions. The DBD for PHDAM only specifies the default values. If you change a PHDAM DBD, the randomization values used by its partitions are not changed. To modify the parameters for a partition, you must make changes to the partition definition which is stored in the RECONs. You can do this with either the HALDB Partition Definition Utility (PDU) or a DBRC CHANGE.PART command.

Recommendations Summary for Randomization Parameters

The general recommendations for randomization parameters follow. You may be aware of reasons for using different values.

1. Use DFSHDC40 for the randomization module.
2. The number of RAPs in a database or partition should be at least twice the number of roots.
3. The root addressable area (RAA) size should be 35% larger than the size of all of the segments in the first data set of an HDAM database or PHDAM partition.
4. The *bytes* parameter should be twice the average database record size.

Fixed Length vs. Variable Length Segments

You may define segments as either fixed length or variable length. You define a fixed length segment by specifying one value for the BYTES= parameter on the SEGM statement for the segment. You define a variable length segment by specifying two values for the BYTES= parameter. A specification of BYTES=(2000,100) defines a variable length segment with a maximum size of 2000 bytes and a minimum size of 100 bytes. The number of bytes specifies the size of the data portion of the segment. It does not include the size of the segment prefix.

The maximum size for a segment is limited by the record size of the data set which contains the segment type. The minimum size of variable length segments is the minimum size stored in the data set. If the segment is sequenced, the minimum size must be large enough to include the key field with one exception. If the segment is compressed and key compression is used, the minimum size may be as small as 4 bytes.

Variable length segments begin with a two-byte length field. Programs use this field to determine the size of the segment. The size specified by an application program may be smaller than the minimum size. For example, a program may create a segment with "20" in the length field even when the minimum size is 30 bytes. In this case, IMS will store an extra 10 bytes in the data set. Later retrievals of the segment will return a 20 byte segment to application programs. The minimum size that a program specifies must be large enough to include the key field for sequenced segments.

If you have a variable amount of data that could be placed in one segment, you have two basic choices in your database design. You could use a variable length segment or you could use a fixed length segment large enough to hold the maximum amount of data. The following explains the advantages and disadvantages of these choices.

Variable length segment

A variable length segment is the natural way to store a variable amount of data. It uses the minimal amount of space but could require extra I/Os. If a replace call increases the size of a segment, the larger segment may not fit in the same space that the smaller segment occupied. For HISAM, this could cause the segment to move to another logical record in the overflow data set. It could also force IMS to move other segments in the database record. This was discussed under HISAM on page 12. This makes variable length segments less attractive for HISAM databases. For HD access methods IMS does not move the prefix when you replace a segment. If the larger segment does not fit in the previous location, it uses another technique. It splits the segment by moving the data portion of the segment elsewhere and adding a pointer in the prefix to the data portion. This was discussed under HD Access Methods on page 14. Splitting the segment can

cause extra I/Os because the prefix and the data portions of a segment may be in different blocks.

If you do not change the length of the segment after you insert it in the database, you should define a variable length segment. This results in optimal space usage. Since you do not change the length, you avoid the overhead caused by movement in HISAM or by split segments with HD access methods. If you change the length of the segment, your choice is not obvious. A variable length segment causes extra overhead for moving the segment, but it optimizes the use of space. A good practice with variable length segments is to specify a minimum size which avoids the splitting of segments by most replace calls. For example, you could make the minimum size large enough to hold 80 or 90 percent of the segments. Only the very large ones would be subject to being split.

Fixed length segment

A fixed length segment must be large enough to hold the maximum amount of data for a segment. The importance of this disadvantage depends on the range for the segment. If the maximum amount of data is much larger than the typical amount, this can have a significant effect. It could make database records span multiple blocks unnecessarily. Fixed length segments are easily handled by replace calls. If a replace call adds more information to the segment, it does not grow in size. So the replaced segment can always fit in the previously occupied space. This avoids the HISAM movement or HD split segment considerations.

If you update the segment frequently, you may want to use a fixed length segment. This will depend on the trade off between the extra space required for the segment versus the potential I/O savings.

Fixed length segments which are compressed have the space usage characteristics of variable length segments. Replacements of these segments may force movement of them in HISAM or split segments with HD access methods. See Compression on page 45 for a further explanation of compression.

Recommendations Summary for Fixed Length vs. Variable Length Segments

The recommendations for choosing between fixed length and variable length segments are:

1. If segments are rarely replaced and contain variable amounts of data, you should define them as variable length.
2. If segments are frequently replaced and the amount of data stored in the segments varies within a small range, you should define the segments as fixed length.

3. If segments are frequently replaced and the amount of data stored in the segment varies over a large range, your choice between fixed length and variable length segments is a trade off between space use and I/O requirements. You can use a minimum size to lessen the number of split segments.
4. If a variable length segment is frequently replaced, consider specifying a minimum size that accommodates most segment occurrences.
5. If a HISAM segment is frequently replaced with a different size segment and other segments follow it in the hierarchy, do not define it as variable length. Either use an HD access method or define the segment as fixed length large enough to hold the maximum size.

Pointer Options

IMS allows you to specify the types of pointers that are used between segments in HDAM, HIDAM, PHDAM, and PHIDAM databases. This section provides advice on choosing the pointers for certain types of segments.

Hierarchic versus Child and Twin Pointers

IMS provides two types of pointer schemes. These are hierarchic pointers versus child and twin pointers.

Hierarchic pointers may be specified as either forward or both forward and backward. A hierarchic forward pointer points to the next segment in the database record hierarchy. Hierarchic backward pointers point from a dependent segment to the previous dependent segment in the database record. Figure 2 shows an example of hierarchic pointing with forward pointers. Each dependent segment has only one pointer. It points to the next segment in the hierarchy. To get from the root segment A to its first D child segment, IMS must traverse all of the B and C segments which are children of A. Root segments have a hierarchic pointer to their first dependent segment and another pointer to the next root. Hierarchic backward pointers point from a dependent segment to a previous dependent segment.

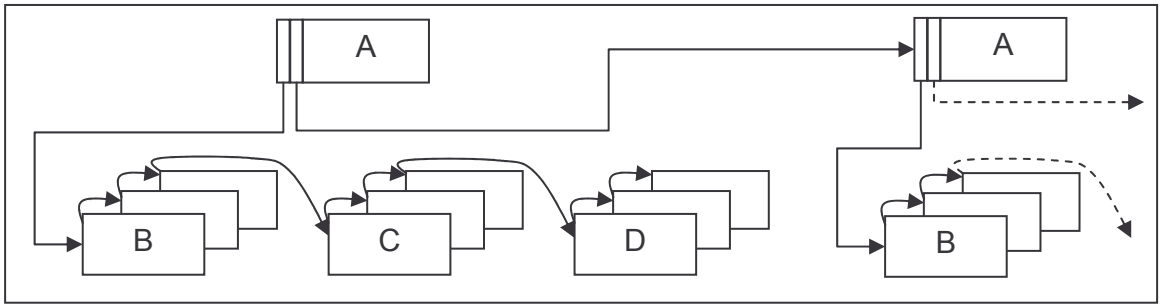


Figure 2. Hierarchic pointers

Child and twin pointers are an alternative to hierarchic pointers. Child pointers may be either child first or child first and last. A child first pointer in a segment points to the first child of a segment type. A child last pointer points to the last child of a segment type. If a segment has multiple children types, it has multiple child first pointers and, possibly, multiple child last pointers. Twin pointers may be either forward or both forward and backward. They point to other segments of the same type which are under the same parent. Figure 3 illustrates the use of child first and twin forward pointers. Segment A has three child first pointers. The first points to the first B segment under A. The second points to the first C segment. The third points to the first D segment. Twin pointers point

to the next segment of the same type. There are twin pointers between root segments and between dependent segments.

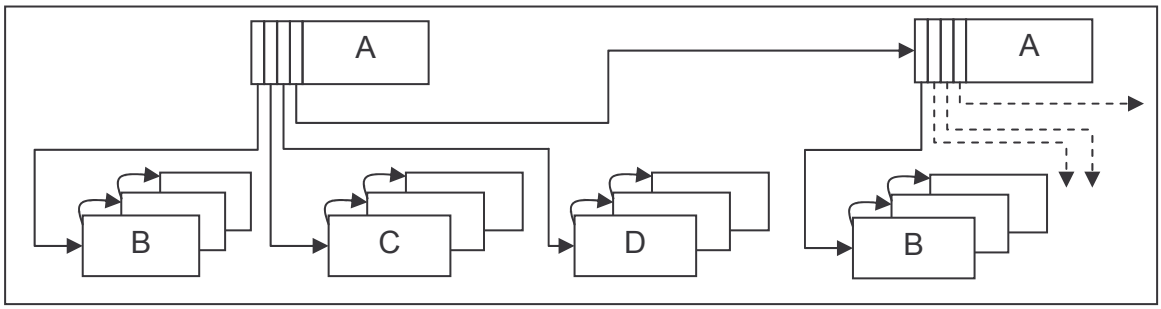


Figure 3. Child and twin pointers

For most IMS databases child and twin pointers are preferable to hierarchic pointers. Hierarchic pointers take less space since each dependent segment has only one pointer and each root segment has only two pointers. They have the disadvantage of usually requiring more accesses to find the desired segment. This is especially true when there are a large number of segments in a database record.

Hierarchic pointers are not available with PHDAM and PHIDAM databases.

Forward Only versus Forward and Backward Pointers

Hierarchic and twin pointers may be forward only (one forward pointer) or both forward and backward (a forward pointer and a backward pointer). Backward pointers are useful when segments are deleted from a chain and the previous segment in the chain has not been accessed. This can occur when a segment is entered via a logical relationship. If the backward pointer does not exist, IMS must find the previous segment in the twin chain or hierarchic chain. It does this by following the parent pointer to the segment's parent and following either the hierarchic or child and twin pointers until in reaches the previous segment in the chain. It recognizes the previous segment because its pointer points to the segment being deleted. The inclusion of backward pointers requires more space and more processing for inserts and deletes. Both the previous and next segments in the chain must have their pointers adjusted when a segment is inserted or deleted. Since the only need for backward pointers is for deletion by a logical path, backward pointers should not be used unless the database satisfies all of the following conditions:

- The database participates in a logical relationship.
- Segments are deleted via the logical relationship path.
- There are typically many segments in the chain which contains the segment to be deleted.

HIDAM and PHIDAM Root Segments

For HIDAM root segments you may specify no twin pointers, forward only twin pointers, or both forward and backward twin pointers. When you specify no twin pointers, roots are accessed via the index. When you specify both forward and backward twin pointers, roots are accessed by the twin forward pointer when proceeding from a previous root segment with get next (GN) processing. This eliminates the need for accessing the index. When you specify forward only twin pointers, they are not used for accessing segments. IMS maintains them, but does not use them. This creates overhead with no benefits; therefore, you should never define forward only pointers for HIDAM root segments. PHIDAM does not allow the specification of forward only twin pointers on root segments. It only allows either no twin or both forward and backward twin pointers.

When you specify both forward and backward twin pointers for HIDAM or PHIDAM roots, IMS must maintain the twin chain when roots are inserted and deleted. It must adjust the pointers in the previous and following roots. This overhead may be significant when there are frequent insertions and deletions of roots.

The overhead of using the index to access the next root in a HIDAM database or PHIDAM partition may not be significant. An index CI typically holds many index entries. This means that accessing the next root typically does not require reading another index CI. For this reason, you should not specify twin forward and backward pointers for most HIDAM or PHIDAM roots segments.

There is a reason why some installations define twin forward and backward pointers on HIDAM and PHIDAM roots. If the index is damaged, the use of twin forward and backward pointers allows you to unload the database or partition if the first pointer in the index is good. Unload uses the twin forward pointers to find successive roots in the database. You may use the unload file to reload the database. This recreates the index. Without these pointers, roots must be found with the index. This would prevent the successful unloading of the database with a damaged index. In this case, you would need to recover the index from an image copy and log records or to rebuild the index by using a tool, such as the IBM IMS Index Builder.

Unsequenced Dependent Segments

When you define a dependent segment, you have the option of specifying a sequence field. IMS maintains segments with sequence fields in key sequence. IMS orders segments without sequence fields by a combination of the rules defined for the segments and the order in which application programs insert the segments.

Some applications rely on segments being returned in key sequence. Others do not depend on this sequencing. If applications do not require that segments be in key

sequence, it is best not to specify a sequence field. Then IMS does not necessarily have to follow the twin chain when inserting the segment.

For segments without sequence fields the location of the inserted segment depends on the last subparameter of the RULES= parameter on the SEGM statement for the segment. The possible values are FIRST, LAST, and HERE. LAST is the default. HERE causes IMS to insert the segment at the current location. This is the least overhead. FIRST causes IMS to insert the segment at the beginning of the twin chain. IMS finds this location with minimal overhead since there is a pointer in the parent to the beginning of the twin chain. LAST can cause significant overhead with long twin chains. You can avoid this by specifying a Physical Child Last pointer from the parent to this segment type. Then IMS can go directly to the last segment in the twin chain. So if you specify LAST or use it as the default, you should define a physical child last pointer from the parent when there may be a long twin chain. This advice only applies to segments without sequence fields.

Defining Hierarchical, Physical Twin, and Physical Child Pointers

You define hierarchic pointers by specifying PTR=H or PTR=HB on SEGM statements. Use PTR=HB to specify both forward and backward pointers.

You define twin pointers by specifying PTR=T or PTR=TB on SEGM statements. Use PTR=TB to specify both forward and backward pointers.

You define physical child pointers in the SEGM statement for the child. The second subparameter of the PARENT parameter is either SNGL or DBLE. SNGL causes IMS to place a physical child first pointer in the parent segment. DBLE causes IMS to place both a physical child first and a physical child last pointer in the parent segment. SNGL is the default. Figure 3 shows an example of a SEGM statement for a child segment whose parent will contain both physical child first and physical child last pointers to it.

```
SEGM NAME=EXPR,BYTES=20,PTR=T,PARENT=((NAME,DBLE))
```

Figure 3. Definition of physical child first and physical child last pointers

Recommendations Summary for Pointer Options

The recommendations for pointer options are:

1. Use child and twin pointers instead of hierarchic pointers.

2. Do not specify twin backward pointers for dependent segments unless you satisfy the criteria for deletes with logical relationships as explained above.
3. Never specify twin forward only pointers for HIDAM roots.
4. Specify no twin pointers for HIDAM and PHIDAM roots.
5. If you specify RULES=(,LAST) or use last as the default for segments without sequence fields, you should define a physical child last pointer from the parent if there may be a long twin chain.

SCAN= Parameter on the DATASET Statement

The SCAN parameter on the DATASET statement affects searches for free space. It is the number of cylinders that IMS scans when searching for space to insert a segment in a HDAM or HIDAM database. When a segment is inserted, IMS tries to place it in the "most desirable block". Generally, this is the block which contains the segment or RAP from which the inserted segment will be chained. Exceptions are explained in the "How the HD Space Search Algorithm Works" section of the *IMS Administration Guide: Database Manager* manual. If IMS does not find space in this block, it uses the HD space search algorithm to find space in another block. In general, the algorithm tries to find space in a nearby block on the same cylinder. If it does not find available space there, the SCAN parameter limits the number of adjacent cylinders that IMS examines when it looks for space. If IMS does not find space within this limit, it inserts the segment at the end of the data set.

SCAN=0 causes IMS to search for blocks in the buffer pool which are on the cylinder containing the most desirable block. SCAN=1 causes IMS to scan the pool for blocks on the two adjacent cylinders also. SCAN=2 causes IMS to scan the pool for blocks on the two cylinders which are on either side of the cylinder containing the most desirable block. IMS must scan its buffer pool once for each cylinder within the scan range. So with SCAN=3 IMS might scan the buffer pool seven times, once for each cylinder within the limit.

You should always specify SCAN=0. The default is SCAN=3. The default was set many years ago when cylinders were much smaller and buffers pools contained fewer buffers. Today, a 3390 cylinder holds over 0.5 megabytes and many buffer pools have thousands of buffers. Multiple scans of these pools can be costly. In any case, there is no particular advantage in putting the segment two or three cylinders away. It can just as easily be handled when placed at the end of the data set.

You cannot specify SCAN for HALDB databases. The space search algorithm for them always operates as if SCAN=0 were specified.

Recommendation Summary for the SCAN parameter

Always specify SCAN=0 on DATASET statements for HDAM and HIDAM databases.

Multiple Data Set Groups

PHDAM, PHIDAM, HDAM, and HIDAM databases may have multiple data set groups. You may use multiple data set groups to store different segment types in different data sets. You may store multiple segment types in the same data set, but all instances of the same segment type are stored in the same data set. Figure 4 is an illustration of the use of multiple data set groups.

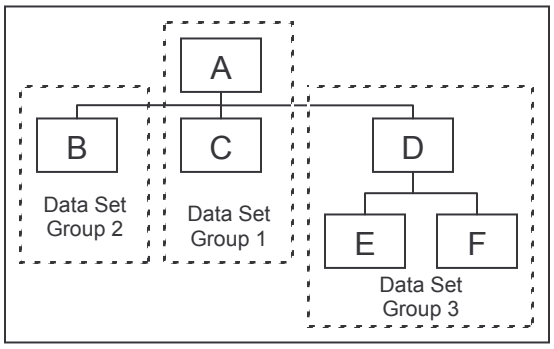


Figure 4. Multiple data set groups

In this illustration, segment types A and C are stored in the first data set group. Segment type B is stored in data set group 2. Segment types D, E, and F are stored in data set group 3.

You may use data set groups for various purposes.

First, you may use them to move infrequently used segments to their own data sets. Then the other segments in the database are stored in fewer blocks and may require fewer I/Os to access. This could be especially important in sequential jobs which read all of some segment types but do not read other segment types. In Figure 4, data set group 3 might have been created for this purpose.

Second, you may create data set groups to reduce the number of blocks which are read to access a segment. In Figure 4, data set group 2 might have been created for this purpose. If there are many instances of segment B, moving segment B to its own data set group increases the probability that the C segments will be in the same block with the root segment. This is useful if C segments are frequently accessed but B segments are rarely accessed.

Third, you may use data set groups for space management reasons. With HD access methods, IMS keeps a bit map indicating which blocks have room for the largest segment allowed in the data set. The space search algorithm uses this bit map when looking for space for inserts. Segment types with very large maximum sizes can make the bit map

misleading. The bit map may indicate that there is not space in a block for the largest segment in the data set, but there may be plenty of free space for smaller segments. Some installations isolate extremely large segment types in their own data set. This is beneficial for space search in other data sets. The bit maps in the other data sets are not skewed by the large segment types. This technique is especially beneficial when there are lots of inserts of the smaller segment types.

Fourth, you may use data set groups to provide more capacity for HDAM or HIDAM databases. You cannot partition these databases. If you use only one data set, the database would be limited to 8 gigabytes with OSAM or 4 gigabytes with VSAM. Since you may define up to 1001 partitions for PHDAM and PHIDAM databases, you do not need multiple data set groups to provide database capacity for them. Usually, it is preferable to convert a HDAM database to PHDAM or a HIDAM database to PHIDAM when additional capacity is needed.

You need to know the access patterns that applications will use before you can make the best decisions about the use of multiple data set groups. Multiple data set groups spread database records across blocks in different data sets. If you use them unnecessarily, you may cause IMS to do additional I/Os. You should define multiple data groups only when you have a reason to do so.

For HDAM and HIDAM databases you use DATASET statements to specify multiple data set groups. IMS assigns segments to the data set specified on the preceding DATASET statement.

For PHDAM and PHIDAM databases you assign a segment to a data set by specifying the DSGROUP= parameter on the SEGM statement.

Recommendations Summary for Multiple Data Set Groups

The recommendations for the use of multiple data set groups are:

1. Define multiple data set groups only when you have a reason to do so. You should use them primarily to reduce I/Os for databases with certain characteristics.
2. You may use multiple data set groups to provide additional capacity for HDAM and HIDAM databases, but conversion to PHDAM or PHIDAM is preferable.
3. Do not use multiple data set groups with PHDAM or PHIDAM databases to provide capacity. Instead, create more partitions for these databases. Multiple data set groups may be appropriate for PHDAM or PHIDAM databases to reduce I/Os.

Compression

You may compress segments by using Segment Edit/Compression routines. This reduces the amount of space that is required on DASD for the segments. It does not affect the view of the segments by application programs. You may use compression with PHDAM, PHIDAM, HISAM, HDAM, and HIDAM databases. You cannot use compression with PSINDEX, SHISAM, or INDEX databases.

You specify compression with the COMPRTN= parameter on the SEGM statement. You may use different specifications for different segment types. You may compress some segment types in a database while not compressing others. You may use different compression routines for different segment types.

IMS supplies a sample compression routine, DFSCMPX0, which implements run length encoding. This reduces all strings of four or more repeating bytes to three bytes. It can give excellent results for segments which contain many blanks, zeroes, or other values which appear in consecutive bytes. IMS also supplies a facility for creating compression routines which use special system hardware for compressing data. This is the Hardware Data Compression facility. You give sample data to the facility and it produces a compression dictionary which is optimized for the sample data. You may also purchase products which supply compression routines. Typically, these products do not provide better compression than that which is done by the facilities supplied with IMS. Instead, they provide ease of use in maintaining and administering compression. The IMS Hardware Data Compression Extended for z/OS tool provides these capabilities.

Compression can significantly reduce the amount of space that is required for storing a database. This can substantially reduce the number of reads and writes that are required for processing the database. The reduction can be significant when the entire database, area, or partition is processed. On the other hand, compression can also substantially increase the CPU time that is required to process a database. When segments are compressed, IMS must invoke an exit routine each time a segment is inserted, replaced, or read.

The use of compression is a trade-off between storage, I/Os, and CPU processing.

A more complete explanation of the use and implementation of compression with IMS is given in *A Guide to IMS Hardware Data Compression.* This guide is available at http://www.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/WP100416.

Key Compression vs. Data Compression

If you implement compression, you have the option of specifying whether you want key compression or only data compression. With key compression the entire segment past

the prefix is compressed. With data compression only the data following the key is compressed. Key compression cannot be used with HISAM root segments. They must use data compression.

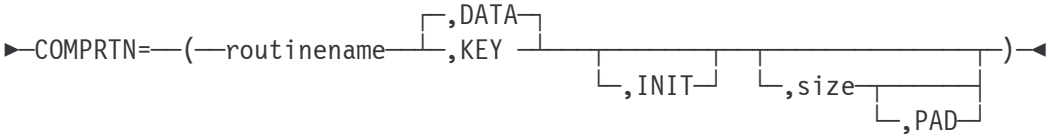
Obviously, key compression produces greater space savings. But there is a cost. When a database call needs to examine the key of a segment, IMS must invoke the exit routine when the key is compressed. The exit routine does not have to expand the entire segment, but it must expand the data through the key. This is not required when only the data is compressed. An example is a get call which is qualified on the key where the segment is in a long twin chain. IMS may have to look at many segments before it finds one that satisfies the call. With key compression this would require the expansion of many segments. With data compression only the segment which satisfies the call would be expanded.

The compression option you choose should depend on the amount of potential savings from key compression versus the extra processing it requires. This depends on the size of the key, the location of the key in the segment, and the type of processing done against the segment.

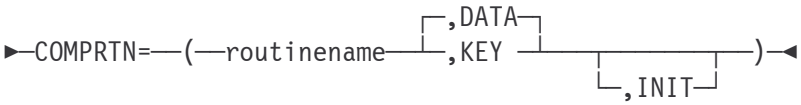
COMPRTN= Parameter

The COMPRTN= parameter on the SEGM statement is used to specify a segment edit/compression routine. The syntax of the COMPRTN= parameter is the following.

Full function fixed length segments



Full function variable length segments



There are five positional subparameters.

routinename

The first subparameter, `routinename`, identifies the name of the routine.

DATA and KEY

The second subparameter indicates if all of the segment, including the key, will be compressed or if only the data past the key will be compressed. DATA is the default. It indicates that only the data past the key will be compressed. KEY indicates that the key and all of the data will be compressed. KEY cannot be specified for HISAM root segments.

INIT

The third subparameter, INIT, is optional. It indicates that the routine will be driven at initialization and termination. This allows the routine to do some special processing, such as loading a table at initialization and deleting it at termination. Your exit routine requirements determine if this parameter must be specified.

size and PAD

The fourth and fifth subparameters are valid only for fixed length segments. The fourth subparameter has two possible meanings. It is either an "increment" size or a "PAD" size. The fifth subparameter determines the meaning of the fourth subparameter. If the fifth subparameter is omitted, the fourth subparameter is an increment size. If PAD is specified for the fifth subparameter, the fourth subparameter is a PAD size.

An increment size should not be specified. The following is an explanation of its meaning and why you should not specify it. An increment size is the number of bytes that the routine may add to a fixed length segment's size. This capability is provided because a compression routine algorithm could increase the size of a segment instead of decreasing it. Increment values of 1 to 32,767 may be specified. If an increment size less than 10 is specified or if no value is specified, an increment size of 10 is used. When PAD is specified, an increment size of 10 is used. The increment size is not required for any compression routines supplied by IMS or IBM products, such as IMS Hardware Data Compression Extended. They never add more than one byte to a segment. Specification of increment sizes is rarely required for any routines since most compression routines never increase a segment's size by more than 10 bytes.

A PAD size is used to specify a minimum size that the segment will be when written to DASD. When compression reduces the segment's length to less than the PAD size, IMS pads the segment to the PAD size before writing it to DASD. PAD sizes are never required, but may be recommended for performance reasons.

The use of PAD is more completely explained in *A Guide to IMS Hardware Data Compression.* It is available at http://www.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/WP100416.

Recommendations Summary for Compression

The recommendations for the use of compression are:

1. Evaluate the use of compression. The space saved may not be worth the extra processing required to compress and expand segments.
2. Evaluate the use of key compression versus data compression. Key compression may not be advisable for segments which have long twin chains and which are frequently processed by calls qualified on key fields.
3. It may be wise to specify a PAD value for full function fixed length segments which are compressed. This can reduce the likelihood that a segment's prefix and its data are placed in separate blocks.
4. Never specify the fourth subparameter (size) for COMPRTN= if you do not also specify the fifth subparameter (PAD).

Secondary Indexes

You may define secondary indexes for PHDAM, PHIDAM, HDAM, HIDAM, and HISAM databases. They are used primarily to provide an alternate key for retrieving records in a database. You may also use them to provide an alternate sequence for sequential processing.

You define secondary indexes for PHDAM and PHIDAM databases with ACCESS=PSINDEX on the DBD statement. Define secondary indexes for HDAM, HIDAM, and HISAM databases with ACCESS=INDEX on the DBD statement.
The indexed databases must include LCHILD and XDFLD statements for their secondary indexes.

Secondary Index Keys

The key for a secondary index is built by using one to five fields in the index source segment. These fields may be non-contiguous. They are specified with the SRCH= parameter on the XDFLD statement in the indexed database.

Unique Keys vs. Non-unique Keys

Secondary indexes for HALDB (PHDAM and PHIDAM) databases must have unique keys. Unique keys are not required for non-HALDB databases, but they are highly recommended. You define unique keys by specifying U in the third subparameter of the NAME parameter in the FIELD statement for the index key field. You define non-unique keys by specifying M. U is the default.

You can create unique keys by defining subsequence fields. Subsequence fields are added to the sequence fields so that a unique key may be created. These keys are the keys of the secondary index KSDS. Application programs do not use subsequence fields when they define a search using a secondary index. They use only the secondary index sequence fields. Subsequence fields are defined with the SUBSEQ= parameter on the XDFLD statement in DBDGEN.

For example, if you have an employee database, you may want to create a secondary index based on employee last name. Of course, there might be duplicate last names among the employees. Adding a subsequence field, such as employee serial number, would create unique keys. In some cases, there might not be a field in the source segment which provides uniqueness. IMS supplies system-related fields which you may use to create uniqueness. The system-related fields are the concatenated key field and the "/SX" field.

A concatenated key field contains the concatenated key of the target segment. The size of this field depends on the size of the concatenated key. A "/SX" field is a unique value generated by IMS. For non-HALDB databases it is the 4-byte relative byte address (RBA) of the target segment. For HALDB databases it is the 8-byte Indirect List Key (ILK) of the target segment. In most cases, the "/SX" field will require less space than the concatenated key field. System-related fields are defined with special names in the NAME= parameter of the FIELD statement. A concatenated key field is defined with a field name beginning with /CK. A "/SX" field is defined with a field name beginning with /SX.

Non-unique keys have two disadvantages. First, they require a second data set for the secondary index. The second data set is an ESDS which contains the duplicate keys. Second, each index entry requires an extra four bytes for the pointers which chain possible duplicate entries. Since a /SX field requires only four bytes in non-HALDB databases, it requires no more space than the duplicates pointer.

Application programs never see subsequence fields unless they process the secondary index as a database.

Direct vs. Symbolic Pointers

Secondary indexes use three types of pointers. Non-HALDB secondary indexes use either direct or symbolic pointers. HALDB secondary indexes use an extended pointer set (EPS).

You may use direct pointers with HDAM and HIDAM databases. They are four byte RBAs. They point at the target segment. You may use symbolic pointers with HDAM, HIDAM, and HISAM databases. Symbolic pointers are the concatenated key of the target segment. They may be up to 255 bytes, depending on the size of the concatenated key. When using a secondary index with symbolic pointers IMS must traverse the target database record from the root to the target segment. This might require accessing multiple OSAM blocks or VSAM CIs. For this reason, direct pointers tend to be more efficient when accessing target segments. This is especially true when the target is several levels below the root in the database structure. For indexes into HDAM databases, symbolic pointers cause IMS also to traverse from the RAP to the root. This makes direct pointers even more attractive.

Symbolic pointers have a significant advantage over direct pointers when reorganizing the indexed database. If you use direct pointers for a secondary index, you must rebuild it when you reorganize the indexed database. This is required because the old direct pointers are no longer accurate. On the other hand, if you use symbolic pointers, the secondary index remains good when you reorganize the indexed database. You do not have to rebuild the secondary index.

So the use of direct vs. symbolic pointers for HDAM and HIDAM databases is a trade-off. Direct pointers have two advantages. They are smaller and they provide more efficient access to target segments. Symbolic pointers have the advantage of not requiring rebuilds after reorganizations of the indexed database.

HALDB secondary indexes always use an extended pointer set (EPS). They do not use the direct or symbolic pointers which non-HALDB secondary indexes use. The EPS contains an internal direct pointer to the target. The self-healing pointer process updates this pointer after reorganizations of the indexed database. You do not have to rebuild a HALDB secondary index when you reorganize the indexed database. The EPS tends to give HALDB secondary indexes the advantages of both symbolic pointers and direct pointers. Access is direct but you do not have to rebuild the secondary index when you reorganize the indexed database.

You select symbolic pointers by specifying PTR=SYMB on the LCHILD statement in the indexed database. You select direct pointers by specifying PTR=INDX on the LCHILD statement. You must specify PTR=INDX on the LCHILD statement for HALDB.

Shared Secondary Indexes

Multiple secondary indexes for non-HALDB databases may reside in the same data set. This is called shared secondary indexes. You should never use shared secondary indexes. When multiple indexes share the same data set, a rebuild of any of the indexes requires a rebuild of all of them. Shared secondary indexes are used when a CONST= parameter is defined on the XDFLD statement in a DBDGEN.

Duplicate Data

You may include duplicate data fields in secondary index entries. These are fields in the secondary index which contain data from the source segment in the indexed database. Duplicate data fields are defined with the DDATA= parameter on the XDFLD statement in DBDGEN. Duplicate data fields are not part of the key of the secondary index. Duplicate data is only available to application programs which process the secondary index as a database, not as an index. That is, the DBDNAME= parameter in the PCB of the PSB references the secondary index, not the indexed database. Duplicate data fields make the secondary index larger. You should define them only when applications are designed to process the secondary indexes as databases. On the other hand, applications which take advantage of duplicate data can be more efficient. This occurs because they do not have to access the indexed database to have access to the data in the duplicate data fields. An application program never sees duplicate data unless it is processing the secondary index as a database.

User Data

You may include user data fields in secondary index entries. These fields are not explicitly defined in the DBD. They are created when the size of the secondary index entry (BYTES= parameter on the SEGM statement) is larger than that required to hold the explicitly defined fields. IMS does not maintain user data fields. User programs which process the secondary index as a database maintain these fields. User data fields are rarely used. If you rebuild the secondary index, IMS does not rebuild the user data fields. They are lost. For this reason, non-HALDB secondary indexes which use direct pointers should never have user data fields. HALDB secondary indexes and non-HALDB secondary indexes which use symbolic pointers do not need to be rebuilt after reorganizations; therefore, they might be able to use user data fields. An application program never sees user data fields unless it is processing the secondary index as a database.

Sparse Indexing

Sparse indexing allows you to build secondary indexes which do not have entries for every source segment in the indexed database. This can be useful with some applications. They may want a secondary index with entries for only certain segments. A sparse index can simplify the application and reduce the size of the secondary index. You can create a sparse index with either of two techniques. First, you may specify the NULLVAL= parameter on the XDFLD statement in the DBDGEN. If the indexed field or fields contain this value in every byte, IMS does not create an index entry for this source segment. Second, you may specify the EXTRTN= parameter on the XDFLD statement. IMS calls the specified exit routine for every insert, delete, or replace of the source segment. The exit routine may inspect the source segment and decide whether or not an index entry should be created for it. You may combine both techniques. If you specify both the NULLVAL= parameter and the EXTRTN= parameter, IMS creates the secondary index entry only if neither technique suppresses the entry.

Recommendations Summary for Secondary Indexes

The recommendations for secondary indexes are:

1. Use unique keys for secondary indexes. You may use /SX fields to create uniqueness.
2. Never use shared secondary indexes.
3. Specify duplicate data fields only when applications are designed to use them.
4. Define space for user data fields only when applications are designed to use them and when rebuilds of the secondary index are not required.

Index

P

R

S

T

V

X