Batch processing is described from the application programmer point of view. Restart and recovery techniques are also discussed. Other parts of the series discuss objectives and architecture, data base structuring, data communication, and transaction processing facilities.

The information management system IMS/VS Part III: Batch processing facilities

by W. C. McGee

The batch processing facilities of IMS/VS permit the user to provide for the arbitrary processing of IMS/VS data bases. This is accomplished by user-written application programs that are stored in the IMS/VS application program library and are invoked during batch executions of the system. Such programs are called *batch processing programs*, to distinguish them from other types of application programs accommodated by IMS/VS. Typical uses for batch processing facilities are:

- *Data base maintenance* in which batches of change records are accumulated and processed periodically to add new records to and to update existing records in one or more data bases.
- *Report generation* in which records of a data base are retrieved sequentially to produce a report.

The facilities described here are those that are available in the Data Base system of IMS/VS. Additional batch processing facilities are provided in the Data Communication and Fast Path features, but since these facilities are closely related to the transaction processing facilities of IMS/VS, they are discussed in Part V.

The batch processing facilities of the IMS/VS Data Base system are presented by first describing the general structure and operation of batch processing programs. This is followed by a discussion of the data services available to batch processing programs. Next, the definition and cataloging of batch processing programs are described. Finally, the operation of batch executions is discussed. Since the batch processing program is the only type of application program to be discussed in this part, it will be referred to simply as "application program."

Application program structure and operation

Application programs to be run under IMS/VS may be written in System/370 Assembler Language, COBOL, or PL/I. An application program communicates with the system through a set of Program Communication Blocks (PCBs). The PCBs for a program are produced and stored in a library at the time the program is defined to IMS/VS, and are loaded into the batch region each time the program is scheduled for execution. PCBs contain user-declared program attributes as well as parameters that are passed between the program and system during execution.

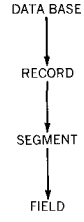
An application program is written as a subroutine that has a standard entry point name and accepts as parameters the addresses of (or pointers to) its associated PCBs. When the program is invoked by IMS/VS, these addresses or pointers are set to point to the program's PCBs, in the order of their appearance in the program definition.

Programs invoke system services through calls to a standard interface routine, specifying the function to be performed, the PCB to be used to communicate the parameters and results of the call, and additional parameters as appropriate to the function being invoked. As a result of the call, control goes to the interface routine and thence to various system modules to carry out the requested function. The system places feedback information in the designated PCB and returns control to the program. When the application program completes, it returns control to IMS/VS by executing a RETURN statement.

Data services

IMS/VS provides two distinct classes of data structures: (1) the Data Administrator (DA) class of structures perceived by the data administrator, and (2) the Application Programmer (AP) class of structures perceived by the application programmer. In this section we describe the AP class, the facilities provided for defining AP structures, and the facilities provided for manipulating AP structures from application programs.

Figure 1 Application programmer data structure types



The AP data structure class is a subset of the DA class as shown in Figure 1. In particular, AP records are structurally independent of one another, in contrast to DA records which may be interconnected through logical relationships. Structure types in the AP class generally have the same attributes and composition rules as the like-named structure types in the DA class.

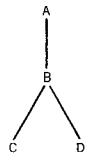
AP data bases are virtual, in that they do not physically exist on secondary storage. An AP data base is materialized, on a seg-

ment by segment basis, from either a physical data base or a logical data base (itself a virtual structure) in accordance with the following relation:

$$\text{AP data base} = \text{senseg} * \text{procseq} * \left\{\begin{matrix}\text{physical data base}\\ \text{logical data base}\end{matrix}\right\}$$

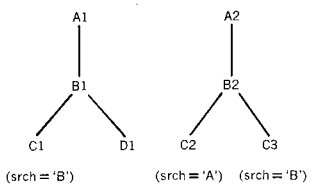
Here *senseg* and *procseq* are optional transformations that must be applied in the order indicated. If an AP data base is derived from a physical data base, the latter is treated as a strictly hierarchic structure, i.e., any logical relationships that are defined on the data base are ignored.

The processing sequence transformation procseq is applicable only to DA data bases for which one or more secondary indexes exist. Its purpose is to permit the indexed data base to be searched and processed in a sequence that is defined by a secondary index. Each entry in the index gives rise to a separate AP data base record whose root segment is the target segment pointed to by the index entry, and whose dependent segments are the antecedents and dependents of the target segment. For example, if the structure

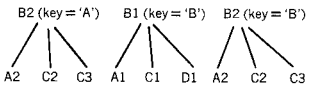


represents a DA record type in which B is the target segment type and C is the source segment type for a secondary index, the processing sequence transformation for that secondary index will produce AP records having the following structure:

The search field value in the index entry becomes the key of the corresponding AP root segment. Since duplicate search field values can occur in the index, duplicate root segment keys can occur in the AP data base. To illustrate, the following DA data base:



gives rise to the following AP data base:



The sensitive segment transformation senseg permits the application program to see only a subset of the segment types in the DA record type, or in a DA record type that is subjected to the processing sequence transformation. The sensitive segment transformation is achieved by omitting one or more segment types (except the root) from the record type. When a segment type is omitted, all its dependent segment types must also be omitted. For example the following record type:

can be subjected to a sensitive segment transformation to produce the following record type:

Instead of dropping a segment type (and its dependents) entirely, it is possible to declare *key sensitivity* for individual segment types. Instances of such segments appear to be null byte strings to the program, but their dependent segments are retained in the AP record structure. This facility is sometimes useful in protecting the confidentiality of data in a given segment type, while allowing access to its dependents.

AP data bases are defined as part of the definition of the program that accesses them. For each AP data base to be accessed by a program, a PCB statement and zero or more SENSEG statements are included in the program definition, as follows:

PCB TYPE=DB, DBDNAME={physical-data-base-name / logical-data-base-name},
PROCSEQ=secondary-index-data-base-name . . .
SENSEG . . .
SENSEG . . .
. . .

A SENSEG statement is included for each segment type to be retained in the AP record type. The definition of an AP data base results in generation of a *type DB PCB* that is subsequently referred to by the program in requesting services related to this AP data base.

The manipulation of AP data structures is accomplished through call statements in the application program. A single call statement accesses a single AP segment or a single path of AP segments, where a path consists of a given segment (the path-defining segment) and all its antecedents. A call statement has the form (in PL/I programs):

```
CALL PLITDLI (parmcount,function,pcbptr,workarea,
              ssa1,ssa2,. . .)
```

where

- *function* designates a character string variable that holds the name of the function to be performed.
- *pcbptr* designates a pointer variable that points to a DB PCB (in effect pcbptr designates the AP data base to be operated on).
- *ssa1*, *ssa2*, . . . designate character string variables that hold Segment Search Arguments (SSAs) that collectively designate the segment or segment path to be accessed.
- *workarea* designates an area in the program where AP segments and segment paths are deposited and picked up by the system.

A segment is in general designated by specifying an SSA for each level in the record hierarchy, down to and including the level of the segment in question. The root-level SSA designates a particular root segment, the second-level SSA designates a particular segment under the root, and so on. Each SSA has the following form:

```
segment-type-name *command-codes (condition)
```

and designates the first segment under the designated parent that is of the specified type and meets the specified condition. Conditions consist of one or more logical predicates that are separated by AND and OR operators, where a predicate has the following form:

```
field-name relational-operator value
```

Field name refers to a field in the segments that are being tested, and must have been declared in the definition of the underlying DA segment type(s). The command codes in an SSA are used to modify the designation process and the call function.

The system responds to a data manipulation call by performing the function called for and by placing feedback information in the PCB specified in the call, including the following:

- *Status code* to indicate that the function has been performed successfully, or that it has not been performed, for a reason indicated in the code.
- *Level, type*, and *concatenated key* of the segment accessed, or of the segment that defines the path accessed.

The *direct retrieval* of a particular segment or path from an AP data base is accomplished with the Get Unique (GU) function. The call specifies an SSA for each level down to and including the desired segment. The system responds by finding the first path in the data base that satisfies the specified SSAs. If such a path can be found, the system moves into the program work area the path-defining segment and every higher-level segment for which a command code of 'D' is specified in the corresponding SSA. Thus, to retrieve only the path-defining segment, no 'D' command codes are used; to retrieve the entire path, all higher-level SSAs contain the 'D' command code. If the designated path cannot be found, a "not found" status code is returned.

The SSAs for a GU call typically take the following form:

segment-type (key-field-name = value)

so that the call becomes a call for a unique path or segment in the data base.

single positioning

For the *sequential retrieval* of segments in hierarchic sequence, the Get Next (GN) function is provided. The system maintains a current position pointer that can point to any segment in the data base. When a GN call is issued, the current position pointer is advanced to point to the next segment in hierarchic sequence, and that segment is copied into the program work area. When the last segment in the data base has been retrieved in this manner, the next GN call returns a status code that indicates that the end of the data base has been reached.

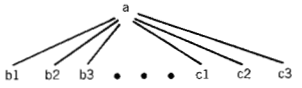
GN calls may be qualified so that only segments that meet a specified condition are retrieved. If one or more SSAs are specified, the system retrieves one or more segments on the first path following the current position that satisfies the SSAs.

The scope of the GN call is the entire data base. In particular, if a GN call is qualified, the system searches the entire data base for the qualifying path. The scope of sequential retrieval can be limited to the dependents of a particular segment, by using the Get Next within Parent (GNP) function. The first GNP call following a GU or GN call establishes the current segment as the parent for sequential retrieval, and retrieves the first dependent of the parent. Subsequent GNP calls retrieve successive dependents in hierarchic sequence. When the last dependent has been

retrieved, the system indicates "not found." GNP calls may be qualified in the same manner as GN calls.

multiple positioning

The sequential retrieval process that has been described up to this point uses *single positioning*, by which a single current position pointer is maintained (in the PCB) for the entire data base. To maintain multiple positions, a program may elect to use two or more DB PCBs, each of which maintains a different position within the same data base. Alternatively, *multiple positioning* may be selected when defining the PCB. With this option, the system maintains in the PCB a current position pointer for each segment type in the data base, in such a way that if S1,S2, . . . , S*n* is a path of segment types, the position pointers for S1,S2, . . . ,S*n* always point to a valid path of segments of the corresponding types. Qualified retrieval calls generally affect only the pointers for the segment types that are involved in the call, and paths defined by the remaining pointers are undisturbed. Multiple positioning is useful for processing children of different types in parallel. For example, in the following record:



the retrieval sequence b1, c1, b2, c2, . . . can be achieved easily with multiple positioning.

GU calls to AP data bases derived from physical data bases are implemented by first locating the desired root segment, and then searching—via pointers or physical adjacency—for the desired dependent segments. If the physical data base is HISAM, HIDAM, or HDAM and the root segment SSA specifies a key value, the root segment is found directly by using the indexing or randomizing technique for the implementation. In other cases, a scan of the data base is required. GN calls are implemented by searching—again via pointers or physical adjacency—for segments that satisfy the call. When the AP data base is derived from a logical data base, the retrieval of segments that have direct counterparts in a physical data base is accomplished as just described. The retrieval of a concatenated segment in general requires the retrieval of both physical data base segments that make up the concatenated segment.

When the AP data base is obtained by a processing sequence transformation, retrieval is as has just been described, except that a secondary rather than primary index is used to locate the root segment. The sensitive segment transformation is effected by skipping over segments to which the program is not sensitive, and for segments to which key-sensitivity only has been specified, returning only the segment's key to the key feedback field in the PCB.

Segments are added to an AP data base with the Insert (ISRT) function. The segment to be added is first assembled in the program work area, and an ISRT call is then issued that contains an SSA for each level down to and including the segment to be inserted. The system uses the SSAs to select, as in GU, a path to the new segment's parent, and then extends the path by adding the new segment. If the new segment has a key, the segment is inserted into its twins in key value sequence. If the key is unique and the key value is already present, the call is rejected. If the new segment does not have a key, the segment is inserted as specified in the definition of the segment type: first in the twin set, last in the twin set, or immediately ahead of the current twin.

The ISRT function may also be used to extend a path with multiple hierarchically related segments. The segments to be added are assembled in the work area before the call is issued, and the command code 'D' is specified in the SSA for the highest-level segment to be inserted. The system responds by taking each segment in sequence from the work area and inserting it as in single-segment insertion.

When an AP data base is derived directly from a physical data base, insert operations on the former are reflected directly in the latter. ISRT calls to HSAM data bases are not allowed (except in load programs). With HISAM data bases, the insertion of a dependent segment generally requires the displacement of data within an access method record to make room for the new segment. If data are displaced out of the record, the displacement process continues into the overflow record(s). The insertion of a root segment implies either the creation of a new overflow record (ISAM/OSAM) or the creation of a new KSDS record (VSAM). In HIDAM and HDAM data bases, inserted dependent segments are placed as close as possible to their hierarchic predecessors, on the assumption that the most common access sequence is the hierarchic sequence. HDAM root segments are placed in or close to the root addressable area record to which the segment key value randomizes.

When the AP data base is derived from a logical data base, the insertion of an AP segment that has a single physical segment counterpart is implemented as has just been described. The insertion of a concatenated segment is carried out in conformance with the *insert rule* specified in the definition of the logical parent type in question. The *physical insert* rule requires the logical parent to be present. If it is, the logical child part of the segment is inserted and chained to the logical parent. With the *logical insert* rule, the logical parent may be absent, and the system derives both a logical child and a logical parent segment from the concatenated segment.

Segments and paths of AP data bases may be replaced with the Replace (REPL) function. Before a REPL call is issued, the segment or path to be replaced must be retrieved with a Get Hold Unique (GHU), Get Hold Next (GHN), or Get Hold Next within Parent (GHNP) call. These calls perform the same function as GU, GN, and GNP, respectively, and in addition advise the system that the retrieved data may be replaced by a subsequent REPL call. The retrieved data are modified as required in the work area, and a REPL call is issued to cause the modified data to be written back to the data base.

The values of key fields cannot be changed through the REPL function. Before copying the work area back to the data base, the system determines whether any key field value has changed, and if so it rejects the call. Field values used for secondary indexing may be changed, and if so the system automatically adjusts the secondary indexes involved.

In HISAM, the replacement of a variable-length segment may cause data displacement as in ISRT. In HIDAM and HDAM data bases, the replacement of a variable-length segment may cause the prefix and data parts of a segment to be separated if the length increases, or reunited if the length decreases. REPL calls to HSAM data bases are not allowed.

The replacement of a concatenated segment in a logical data base is done in conformance with the *replace rule* specified in the definition of the logical parent. A *physical replace* rule permits changes only to the logical child part of the segment, and rejects the call if changes to the logical parent part are attempted. A *logical replace* rule is the same, except that changes to the logical parent part are simply ignored. A *virtual replace* rule permits changes to both parts of the concatenated segment.

Segments and paths of AP data bases may be deleted with the Delete (DLET) function. The segment or path to be deleted is first retrieved with a GHU, GHN, or GHNP call, and a DLET call is then issued to accomplish the deletion. The deletion of a given segment implies the deletion of all its dependent segments.

The deletion of an AP segment generally results in the deletion of the underlying physical data base segment. An exception occurs in deleting segments that participate in logical relationships. Such segments are actually deleted only when they are no longer required to maintain logical relationships.

When a segment is deleted from a HIDAM or HDAM physical data base, the space occupied by the segment is reclaimed, i.e., the space is made available for new segments. In HISAM data bases, the segment is simply tagged in its prefix as being deleted;

the space occupied by the segment can be reclaimed only through reorganization of the data base. The system automatically adjusts any indexes affected by a segment deletion. DLET calls to HSAM data bases are not allowed.

In using IMS/VS data services, application programs must observe *processing options* declared for a program in its definition. A processing option may be specified for each AP data base or segment type that the program intends to access, and may be any combination of G (retrieval), I (insert), R (replace), and D (delete), together with an intent (P) to use path access. Any call that violates the declared processing option is rejected.

Application program definition and cataloging

Application programs are defined to IMS/VS through the use of the Program Specification Block Generation (PSBGEN) utility program. That utility accepts declarations of such program attributes as name and language, the definitions of the AP data bases to be accessed by the program, and the type of access to be made. The PSBGEN generates a Program Specification Block (PSB) that contains a PCB for each AP data base defined. The PSB is stored in the IMS/VS PSB library.

In addition to being defined, an application program must be compiled and stored in the IMS/VS application program library. These functions are provided by the appropriate language compiler and operating system utility programs.

Batch executions

Batch processing is accomplished by starting a batch execution of the system. A batch execution is a normal operating system job that is initiated by the operator through a job request that specifies the IMS/VS *region controller* module as the program to be executed, and allocates to the job the data sets for the physical data bases to be accessed. The job request also specifies, as a parameter to be passed to the region controller, the name of the user-written application program to be run in this system execution.

When an operating system region is available, the job is scheduled for execution. The region controller is fetched from the IMS/VS load module library and given control. The region controller performs the following operations:

1. Loads from IMS/VS control block libraries the PSB and the DBDs that define the application program and the data bases that it is to access.

2. Loads from the IMS/VS load module library the action modules that are required in this execution of the system.
3. Loads the application program from the IMS/VS application program library.
4. Transfers control to the application program.

When the program completes, it returns control to the region controller, which in turn relinquishes control to the operating system, thereby ending the job.

A batch execution may include the use of a system log to record changes that the application program makes to the data bases. The log is useful for restoring data that have been erroneously changed or deleted by an incorrect program, and for restarting the program after system failure.

checkpoint/restart

To protect data bases against the effects of incorrect application programs, IMS/VS provides a Data Base Backout utility program. The backout program searches the system log in the backward direction for data base change entries that have been attributed to the program in question, and uses the "before" image from each such entry to replace the corresponding data in the data bases. The utility ends when it encounters the scheduling entry for the program.

To preclude the rerunning of an entire job in the event of system failure, IMS/VS provides a *program checkpoint/restart* facility that permits programs to make periodic copies of selected program and system variables (*checkpoints*), and to be restarted from such checkpoints in the event of system failure. Program checkpoints are taken by a Checkpoint (CHKP) call that specifies the program variables to be saved, and a checkpoint identification for subsequent reference to the checkpoint. The system responds to the call by flushing the data base buffers to direct access storage and by creating a checkpoint entry in the system log that contains the user-specified checkpoint identification, the keys of the last data base records to be processed, and the user-specified program variables. The system also writes a checkpoint message to the operator, giving the checkpoint identification.

Before restarting a program, following a system failure, the Data Base Backout utility must be run to backout the changes made by the program since a specified checkpoint. The utility ends when it finds the specified checkpoint (or the program scheduling entry, if that occurs first) in the system log. If the system log was not closed at the time of failure, it may be closed by running a Log Tape Terminator utility program. This program uses a main storage dump to locate and flush log buffers, and then closes the log data set.

A program is restarted through the regular batch execution initiation procedure, with specification of the identification of the checkpoint from which restart is to occur. The application program must issue, as part of its initialization processing, an Extended Restart (XRST) call that specifies the program variables to be restored. The system responds by searching the system log for the specified checkpoint. The system then restores the specified program variables and position in GSAM data bases. The program must separately restore position in other data bases and in non-IMS/VS data sets.

Summary and concluding remarks

The batch processing facilities of the IMS/VS Data Base System permit users to write application programs in COBOL, PL/I, or System/370 assembler language; to define these programs to the system; and to execute them through batch executions of the system. Application programs may issue calls to the system to retrieve, modify, add, and delete data base data. For programming simplicity, application programs view data bases as sets of hierarchic records, and transformations between this view and the underlying data are carried out automatically. A system log and a set of utility programs are provided to facilitate the recovery of data bases from the effects of incorrect programs, and to facilitate the restarting of long-running programs, following a system outage.

IMS/VS was one of the first generalized data base management systems to provide hierarchic data structures. Other early systems providing such structures were the FACT[1] system for the Honeywell H800, the Generalized Information System (GIS),[2] and to a limited extent the formatted file systems.[3] The original motive for this type of structure was to eliminate redundancy inherent in flat files, when the entity being modeled has variably-occurring attributes or subordinate entities. (Engineering documents, a common entity in the early IMS milieu, have this property.) Since then, the hierarchic data structure has become a proper logical view or model of data, i.e., it is a natural way of looking at certain collections of data. IMS is frequently cited as the exemplar of the hierarchic model. The lesson from this experience and other similar experiences seems to be that if a data model is to be successful, it must simultaneously solve the data view problem and the implementation-performance problem.

CITED REFERENCES

1. R. F. Clippinger, "FACT: a business compiler," *Annual Review in Automatic Programming* 2, 231–292, Pergamon Press, Elmsford, New York, 1961.
2. J. H. Bryant and P. Semple, "GIS and file management," *Proceedings of the ACM 1966 National Conference*, 97–107, New York (1966).

3. J. P. Fry and E. H. Sibley, "Evolution of data-base management systems," *ACM Computing Surveys* **8**, 1, 7–42 (March 1976).

GENERAL REFERENCES

1. IBM Corporation, *IMS/VS Version 1 System/Application Design Guide*, Document SH20-9025, IBM Corporation, Data Processing Division, White Plains, New York 10604.

2. IBM Corporation, *IMS/VS Version 1 Application Programming Reference Manual*, Document SH20-9026, IBM Corporation, Data Processing Division, White Plains, New York 10604.

3. IBM Corporation, *IMS/VS Version 1 Utilities Reference Manual*, Document SH20-9029, IBM Corporation, Data Processing Division, White Plains, New York 10604.

Reprint Order No. G321-5048.