A Design for the Library History Data Base

by

Florian Bruckner

Dept. of Computer Science
Technical Report CS-80-48

University of Waterloo
Waterloo, Ontario, CANADA

November 1980

1 INTRODUCTION

1.1 Problem Statement

The Library of the University of Waterloo serves 30,000 patrons, providing a collection of more than 1.6 million items. On an average day activity at the various circulation desks totals more than 5000 transactions for the library's GEAC on-line circulation system [GEAC78]. For each transaction such as charge and discharge, the GEAC system creates a log record containing comprehensive information about the transaction. The log records are periodically appended to the history file, which resides on tape.

The efficient management of an enterprise of this size and activity obviously requires large amounts of preprocessed information about all its aspects. In this thesis, we are concerned with maintaining information about the library's usage as an aid to management decision making. Decision problems pertinent to this aspect of the enterprise require various types of statistical reports about short-term behaviour (weekly, biweekly, monthly), as well as long range planning aids (academic term, year, seven-year period).

TABLE OF CONTENTS

1 INTRODUCTION

1.1 Problem Statement

The Library of the University of Waterloo serves 30,000 patrons, providing a collection of more than 1.6 million items. On an average day activity at the various circulation desks totals more than 5000 transactions for the library's GEAC on-line circulation system [GEAC78]. For each transaction such as charge and discharge, the GEAC system creates a log record containing comprehensive information about the transaction. The log records are periodically appended to the history file, which resides on tape.

The efficient management of an enterprise of this size and activity obviously requires large amounts of preprocessed information about all its aspects. In this thesis, we are concerned with maintaining information about the library's usage as an aid to management decision making. Decision problems pertinent to this aspect of the enterprise require various types of statistical reports about short-term behaviour (weekly, biweekly, monthly), as well as long range planning aids (academic term, year, seven-year period).

The information about the behaviour of the collection, such as usage patterns, peaks in demands for certain items and aging of items, can, at least in principle, be extracted from the history file (in connection with the library master catalogue). This thesis provides a discussion of various methods to preformat the history file and their applicability with respect to particular constraints. The goal is to outline a suggestion as to how to rearrange the history file into a meaningful data base of manageable size to support efficient report generation.

1.2 The History File

The history file is a collection of variable length records of (at this time) four different types. Each record contains a field indicating its type as a numerical value as follows:

code	transaction described
01	charge
02	discharge
03	single copy request (RFP)
04	multi copy request (hold)

The record layout is the same for all four types of record; only the meaning of the various fields differs for each type. A detailed record definition is shown in tables 1.1 and 1.2. Note that some fields are undefined for certain record types, e.g. field 10 contains no valid infor-

FIELD #	LENGTH	TYPE	CONTENTS
0	2	binary	length of record
1	2	99	transaction type
2	3	999	terminal number
3	8	YY-MM-DD	date transaction was created
4	5	99999	time in 2 sec. from 00:00, transaction was created
5	9	999999999	item number (physical volume)
6	9	999999999	record number of patron on GEAC
7	3	999	status bits from transaction
8	3	999	charge: secondary status bits discharge: number of fine notices RFP: processing code hold: items linked via hold
9	3	999	location, valid only for transaction types RFP and hold
10	8	YY-MM-DD	charge: due date hold: date not required
11	4	9999	charge: due time hold: time not required
12	8	YY-MM-DD	discharge: date returned hold: date before which not required
13	4	9999	discharge: time returned hold: time before which not required
14	9	9999999.99	discharge: fine amount hold: pointer to call # record
15	3	999	charge: number of times renewed hold: copy # that satisfies hold
16	2	99	charge: number of overdue notices sent
17	10	9999999999	patron type description
18	3	999	department # of patron
19	9	999999999	status code for patron
20	3	999	loan privileges of patron
21	2	99	year code
22	1	9	privilege category
23	2	99	statistical classification
24	8	YY-MM-DD	date registered
25	8	YY-MM-DD	expiry date
26	3	999	default loan period of item
27	3	999	items's default location
28	3	999	items's status
29	3	999	status of the call number
30	8	99999999	accession number (link to library master catalogue)

Table 1.1 History file layout
(fixed section)

FIELD #	LENGTH	TYPE	CONTENTS
31	2	binary	length of call number
32	var	alphanum	call number text (max length 30)
33	2	binary	length of author field
34	var	alphanum	author text
35	2	binary	length of title text
36	var	alphanum	title text

Table 1.2 History file record layout
(variable length section)

mation in record type 02. However, all fields of the fixed length portion are present in records of all types, undefined fields being set to zero. The variable section of a record is only included in records describing a transaction that involves an item not listed in the library master catalogue. Its existence is indicated indirectly in the length field of the record.

Certain relations between the data contained in some fields are immediately obvious from the record definition. Denote the content of field i by (i); then (1) = 01 implies (9) = (12) = (13) = (14) = 0 ; (1) = 02 implies (9) = (15) = (16) = 0 ; etc.; Other relations follow from the semantics of the data. However, deducing such relations from data semantics has some potential pitfalls, as the semantics are not clearly defined. One might for instance suspect that, analogous to the above, (1) = 02 implies (3) = (12), but this relation does not hold.

Typically, the history file grows by more than 5000 records a day. Approximately one in four records has a variable portion, having an average length of 55 bytes when present. In one year the file thus grows by roughly 280 million bytes, or 10 tape volumes. The record types 01 and 02 each make up approximately 48% of all records; types 03 and 04 each account for approximately 2%.

1.3 Computing Facilities and Constraints

Report generation from the history file in connection with the library master catalogue is to be run as a batch application under VM on the University of Waterloo's Computing Centre's IBM 370/158 and 3031. The Computing Centre is currently replacing the older 2314 disk drives by the model 3350, and will replace the processor 3031 by a pair of 4341 [Watt80]. On the 3350 system, the disk packs are stationary, i.e. not mountable. This implies that the history data base cannot be kept on off-line disk packs to be mounted only during specific time slots available for library data processing. On the other hand, it is obvious that the data base cannot reside on one or more stationary 3350 disk drives, thus depriving the Computing Centre of these drives.

As a consequence of these constraints, the history data base must reside off-line on tape. This leaves the choice of two different general strategies in the design of the data base:

1) Design the data base on-line on one or more 3350 disk packs. After constructing the data base, dump the disks on tape. Note that the data base need never be updated, as it will only contain fixed facts about past library transaction activities. Future additions to the data base can be constructed as successive generations in the same way as the first one. When reports are to be produced from this data base, it is loaded from tape onto 3350 disks, and can be accessed directly. The disks are freed again after the reports are generated.

2) Design the data base as a collection of sequentially accessible files. In this approach, the data base need not be loaded onto disk prior to accessing it. The constraint of sequential access is not a severe one, since report generation requires all pertinent data over a given subject range to be examined, in a suitably ordered sequence. Different orderings, however, cannot be maintained by pointers, as on disk, but will require different versions of the same file, sorted in the respective orders.

Report generation usually involves only simple and very fast computations, and the time required to read the files from tape will be the governing factor in determining the elapsed run time for a given application. Thus by reading sequentially and processing "simultaneously", the second

approach will yield faster report generating facilities than the first; in fact, elapsed time for a typical report generator will be of the same order as that required merely to load the data base onto disk. Therefore in this thesis, we will concentrate on the latter strategy.

1.4 Thesis Outline

The goal of this thesis is to examine various methods to devise a data base which is suited to meet the library's demand for efficient report generation. The reports will deal with a wide range of aspects of the library's collection. They do not form a fixed set of reports; rather the very existence of a general reporting facility will create even more demand for information, and thus reports, whose nature and subject are highly unpredictable today.

Given the constraints outlined above, we will analyze different methods with respect to their applicability and usefulness for tape files. In chapter 2 we will discuss various techniques to compress files, i.e. to reduce their size while keeping all data available. In chapter 3, we look at record splitting methods to divide a file into several subfiles, each containing part of the original data. After considering advantages and trade-offs of different approaches, we will present an outline of a solution to this design problem in chapter 4. We conclude the thesis in chap-

ter 5 with final comments and details of the design to be investigated further.

2 COMPRESSION

2.1 Introductory Remarks

While there is no standardized definition of compression, we will adopt the definitions of Gottlieb et al. [Gottlieb75]:

> Compaction of data means any technique which reduces the size of the physical representation of the data while preserving a subset of the information deemed "relevant information".
>
> Compression of data is a compaction technique which is completely reversible (i.e. compression preserves all information contained in the data).
>
> Compression Factor is the size of the compressed file expressed as percentage of the original file (called Compression Ratio in [Gottlieb75]).

Compaction techniques which are not completely reversible include truncation and rounding of numerical values and rear end compaction of a sorted sequence of keys. Because of the unpredictability of future demands for reports to be extracted from the history file, we cannot afford any loss of information while reducing the file's size. Thus we are confined to compression techniques as defined above.

It should be noted that compression is not to be confused with storage efficient data design. The thoughtful data designer does, of course, take into consideration the aspect of space efficiency, and may apply techniques that are similar to those which are used in compression, e.g. decimal to binary conversion, data packing, or field differencing [Ruth72]. However, the result of his/her work is the input to a compression algorithm, and it is not in the scope of the deviser of a compression technique to redesign the data for optimal overall results. The potential merit of a good data design is conceivably greater than that of good compression of a weakly designed file, just as improving the abstract schema for a data base may have more impact than improving the internal/physical schema [Santoro80]. However, when looking for an efficient compression technique for the history file, we have chosen to accept the file as it was originally designed [Library Systems].

2.2 Overview of Compression Techniques

No mathematically precise overall concept of compression has yet been devised. Major work in this area was done in the business environment, based mostly on isolated practical cases. The apparent lack of interest in the scientific community might be based on one of the following two impressions: 1) Given the task of compressing a file with certain

characteristics, the best suited compression technique is obvious and can be determined by straightforward reasoning. 2) A compression problem is parametrically dependent on so many characteristics and peculiarities of the given file and on environmental constraints that there seems to be no methodical way to determine an optimal solution. Although for some files one can fairly easily find compression methods that work "quite well", we consider the first impression a delusion, for the general case. The latter impression, on the other hand, actually constitutes a challenge which has not yet been seriously accepted.

Compression techniques can be roughly divided into three major categories: statistical encoding, pattern substitution, and differencing. They are, however, not clearly delineated, and many actual compression techniques fall under two of these categories. A possible categorization of selected techniques is depicted in figure 2.1.

In this chapter, our task will be to find from this variety of techniques those which are specifically suited for the history file. Because of the size of the file, we will concentrate on very fast algorithms which yield good compression factors, rather than on those which promise optimal results with medium or slow speeds. Our criteria in comparing different techniques thus will be speed and compression factor; we will not be concerned with the conceptual com-

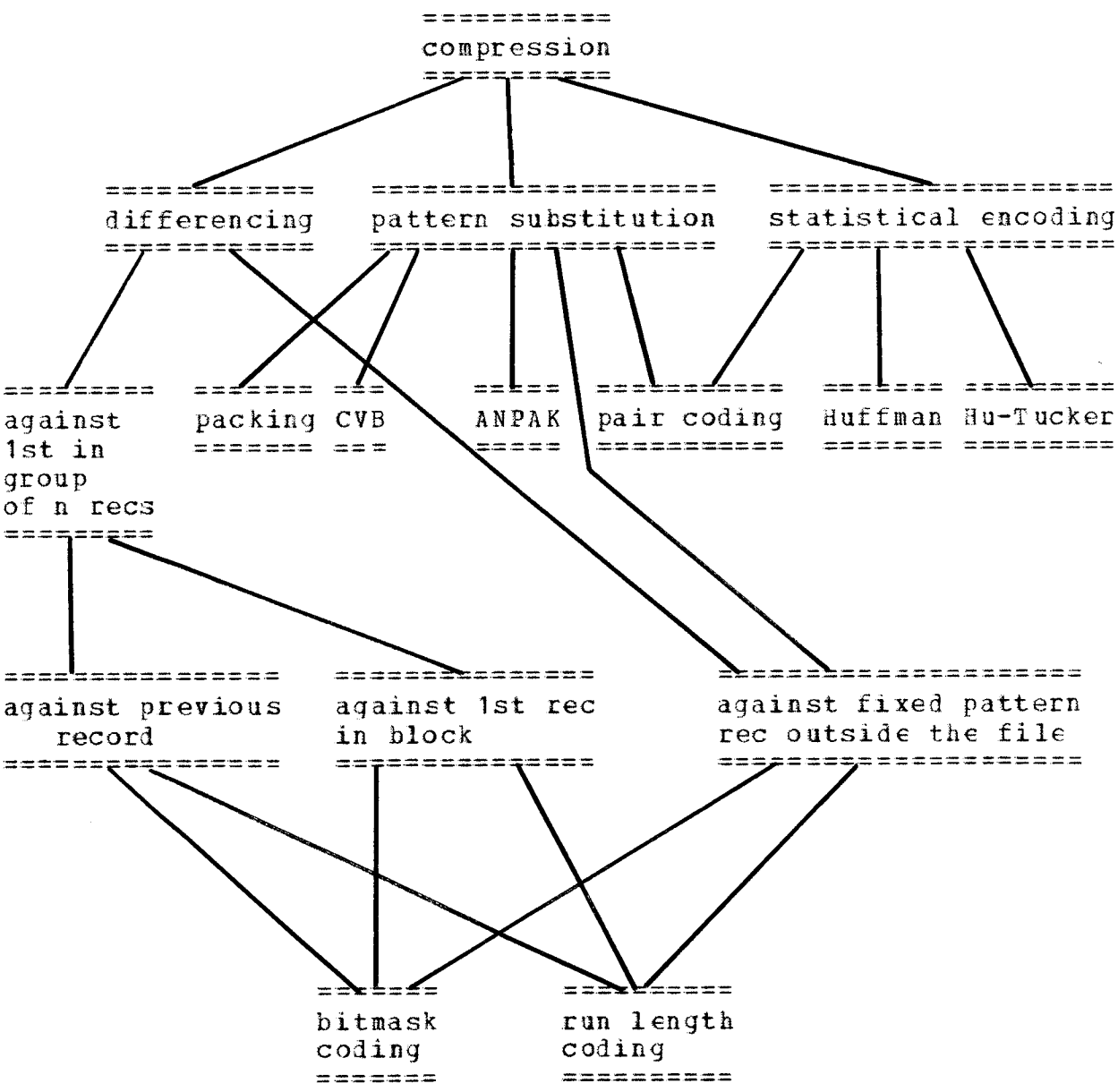

Figure 2.1 A possible categorization of some compression techniques

plexity or the space requirement for a program implementing a given technique. In the following three sections, we will discuss for each category its major typical representatives and assess their applicability to our problem. Finally, we will present results of test runs for those techniques deemed most promising.

2.3 Statistical Encoding

Data in a file are usually represented as strings of characters over a fixed alphabet. Different characters of this alphabet will normally occur in the file with different frequencies. Statistical encoding techniques try to exploit this fact by translating the original alphabet into a code of variable length codewords, such that the length of the codeword for a given character is inversely related to its frequency in the file.

An early practical example of statistical encoding is the Morse code for English language text. This code translates letters of the English alphabet into codewords over {".","_"," "}, while taking advantage of the different frequencies with which the letters occur in normal English text. The most frequent letter e is encoded ". ", while the much less frequently occuring q is encoded "__._ ". From results in information theory we know that if individual characters are encoded, the shortest possible binary repre-

sentation for a text of N characters over a finite alphabet is N * H, where H is the entropy of the distribution of characters in the text [Abramson63]. Note that the more uneven the distribution, the better the compression factor.

An encoding technique which comes closest to the lower bound of N * H is the Huffman encoding scheme [Huffman52]. This scheme has also the important prefix property, i.e. no codeword c(i) is the prefix of another codeword c(j). Thus retrieving the original text from the encoded one requires only one single pass over the input stream.

The information theoretic lower bound of N * H is based on the assumption that the probability of appearance of any character in the text is independent of the probability of appearance of any other character. This assumption, however, is in practice violated in virtually every file. Dependencies of probability of appearance occur in a variety of ways: in English text, for example, the occurrence of the letter "t" will drastically increase the probability of appearance for the letter "h", in many files zeros and blanks tend to appear in "clumps", etc.

The assumption of independent probabilities being violated, Huffman encoding of individual characters may not yield the best possible compression factor. But because it assumes independent probabilities and is, therefore, independent of the semantics of the data, the expected compres-

sion factor for a given file can be calculated from a simple statistical pass over the data, and without actually applying the compression routine.

An extension of Huffman encoding that does take data semantics into consideration could be viewed as successive application of two compression techniques. Usually the most frequent character in a file, such as the zero in the history file, appears in "clumps". Clumps of length greater than two can be replaced by a sequence "(flag) (count)" in the original text. This technique is a type of run-length encoding and is described in more detail below under pattern substitution. The flag character is then added to the original alphabet and Huffman encoding is applied to the text in which the clumps have been replaced.

In general, Huffman encoding can be combined with any other compression technique to yield better compression factors. Extensions of Huffman encoding like the one above can be found as indivisible encoding schemes in the literature [Ruth72], [Mulford71], [Gottlieb75]. We prefer the cleaner view of extended Huffman encoding as pure Huffman encoding, preceded by any other suitable compression technique.

Implementing Huffman encoding on a byte-oriented machine will result in a relatively expensive compression routine, because variable length bit strings are not effectively handled by the machine's instruction set. Experiments con-

ducted by Gottlieb et al. have shown that Huffman encoding takes roughly twenty times the processor time used by a byte-oriented compression method based on differencing [Gottlieb75].

A statistical encoding technique which is related to the Huffman scheme was developed by Hu and Tucker [Hu71]. The Hu-Tucker algorithm has the additional property that it maintains the order of the original alphabet, i.e.

$$\text{if } d(i) < d(j), \text{ then } c(d(i)) < c(d(j)),$$

where $c(d(k))$ is the codeword encoding character $d(k)$. The advantage of this technique is that a file can be sorted without decompressing it first. Because of the constraint of ordering, the compression factor achieved will in general be worse than that of the Huffman encoding scheme, while the processor time used is essentially the same [Gottlieb75].

2.4 Pattern Substitution

While statistical techniques encode the source text character by character, methods belonging to this category work on aggregates of several characters at one time. These aggregates may be letter pairs, logical fields of a record, or patterns that exceed field boundaries. The input text is scanned for predetermined patterns, and when such a pattern is found it is replaced by a shorter string of characters which does not occur in the original text, or by the empty

string. Application of a pattern substituting compression technique requires exact knowledge of the underlying alphabet of the file as well as some insight into its semantics. Candidate patterns must be found and their length and frequency of occurrence be determined. Techniques in this category thus depend heavily on the properties of the individual file to which they are to be applied and are not as well suited for a general compression facility as are statistical methods.

Run time will generally increase (and the compression factor decrease) with increasing complexity of patterns to be recognized and replaced. A very fast compression technique for the history file is, for instance, one that finds the pattern "-" and replaces it by the empty string. This achieves a compression factor of roughly 93% and, in IBM Assembler language, takes two instructions (one MVC and one TR instruction) per record, in addition to input/output and loop control.

A very complicated pattern substitution technique is the ANPAK compressor described by Marron and De Maine [Marron67]. ANPAK is a component of the universal compression package COPAK for data bases. It is a recursive bit-pattern recognition technique which determines the patterns to look for automatically during the compression process. Because of its intended universality, its compression factor

varies greatly with the properties of the data on which it works. Results range from 100% (no compression) to 61%, as reported in [Marron67] for the text of a technical report. While the authors do not disclose the time used for compression, they do say that "...read-in time for compressed information plus time for decompression is significantly less than read-in time for the original information." For this technique, however, compression is much more expensive than decompression, because recursively finding repeating patterns and constructing the translation table requires several passes over the input stream and a greater amount of computation than decompression by simple table look-up.

A pattern substituting technique that is particularly suited to handle English language text was developed at the Scientific Exchange, Smithsonian Institute, on an IBM System 360 [Snyderman70]. This technique takes advantage of the fact that the text to be compressed contains only 88 of the 256 IBM EBCDIC 8-bit codewords (bytes). The 168 codewords which never occur in the original text are used to encode the most frequent pairs of characters. Taking into account upper and lower case letters, numbers, special symbols, and blank, it is not an easy task to determine the 168 character pairs most frequently used in English text. Also, it is not a trivial problem to reassign the 256 codewords to pairs and single characters in a fashion that supports a fast running implementation of this compression technique. After a series

of trial and error tests the authors arrived at a routine that achieved a compression factor of 65% and used 73 msec per 1000 characters on an IBM 360/40.

This pair encoding method is of very limited applicability to the history file, as only the variable portion of the extended records, and thus less than 10% of the file, contains English text. Pair encoding for the numerical values contained in the other 90% of the file can be achieved much faster and with less consideration, by transforming the data from the zoned to the packed format using the PACK instruction [IBM Principles of Operation]. Each field in the fixed part of the record can be packed without loss of information, provided the hyphens in the date fields are first removed (for example, by a procedure as outlined above). The reason, of course, is that we know from the data definition that all left half-bytes contain a hexadecimal "F", which can be reconstituted during decompression by the UNPK instruction.

This simple packing of numerical values can be viewed as a type of pattern substitution, for it repeatedly replaces a pattern "Fx" by the shorter "x", if "Fx" is not the last byte in a field, or else by "xF" (x being a hexadecimal digit from 0 to 9). Applied to the history file, this method compresses the fixed part of the records from 149 to 91 bytes, yielding a compression factor of 65% for the entire

file. It takes per record essentially one MVC and one TR (to remove the hyphens as above) and 30 PACK instructions, one for each field.

There are , however, two disadvantages in packing field by field, caused by peculiarities of the PACK instruction. The last byte packed by one instruction is not compressed, and packing a field of even length n results in (n/2 + 1) bytes, where the left half of the leftmost byte is wasted. This can be avoided by applying PACK instructions with maximum length, across field boundaries, and overlaying successive instructions by one byte. The fixed part can now be packed into 70 bytes using only 10 PACK instructions, and the compression factor achieved is 52% overall.

Note that the above compression method changes the fixed part to a 70 byte long packed signed decimal number. This number can be worked on by another pattern substituting technique, namely decimal to binary conversion. Since

2**472 < 10**138 < 2**480,

a binary number capable of representing any 70 byte long decimal number, and thus any fixed part of the record, fits into 480/8 = 60 bytes. This constitutes a compression factor of less than 46% for the history file. The implementation of this technique on an IBM machine, however, is not quite as straightforward as the preceding compression step, because the Assembler instruction CVB can only handle 9 decimal digits at a time.

The last method we will discuss in this category is probably the most widely used compression technique, null suppression. This pattern substituting technique takes advantage of the fact that files often contain a high percentage of null, or default, values, such as blank or zero, and that these values tend to appear in clumps.

There are two main strategies to represent runs of default values by shorter patterns, bit mapping and run-length encoding. In bit mapping, one bit is used to represent one unit in the record. Units can be bytes, machine words, fields, or arbitrary chunks of the record. The bits are collected in a bit map, which is appended to the front of the record. If a unit represented by a bit contains only default values, the bit is set to zero and the unit is omitted from the record (i.e. replaced by the empty string); otherwise the unit is kept and the bit is set to one. Alternatively, in run-length encoding, a run of default values of length greater than two is replaced by the pattern "(flag) (count)". The character serving as flag must not occur in the original data. If all characters are used, we can select as a flag the least frequently occurring character and double it when it appears in the data.

Detailed statistics of the object file are needed for both strategies to determine the unit sizes or the space for the run-length count that yield best compression factors.

Trial runs with a representative portion of the file are probably the easiest and fastest way to find a good solution. A generalization of these methods is to treat several characters in the same way as the default character. This can be done by including one bit map for each character or using a replacement pattern of the form "(flag) (character) (count)".

Because the history file contains a substantial percentage of zeros, null suppression appears to produce a good compression factor. However, because estimates cannot be given as easily as for packing and decimal to binary conversion, results of trial runs with null suppression are postponed to section 2.6.

2.5 Differencing

In contrast to the previously described methods, which achieve compression by changing the physical representation of the data, the emphasis of differencing techniques is on the information content of the data. Their goal is to reduce the overall amount of information recorded in the file by not repeating parts of the information that are already stored elsewhere, within or without the file. In general, an information unit to be compressed is compared to a reference unit, and only the difference between them is kept in the record. A special type of differencing uses one

field in a record as a reference unit for another, similar field in the same record. If, for example, a record contains the fields (date1),(date2), compression might be achieved by storing "(date1)(interval2)", where interval2 = (date2 - date1). This technique, however, belongs more to the realm of data design than to that of compression, because it applies directly to the definition of the data structure (record). In the following, we will therefore limit the discussion to differencing techniques with reference units chosen from outside the data structure currently being compressed.

Among these, we can distinguish between techniques that use fixed reference units (the reference record), and those which change the reference record dynamically as successive records are compressed. A very simple example of differencing with a fixed reference unit is the following: An integer field contains a year which falls in the range between 1900 and 1999 (this range might not be known at data design time). The field can be compressed by using 1900 as a reference unit and storing (actual year - 1900), which is in the range of 0 to 99, and fits into a smaller field. Note that the difference recorded is the result of arithmetic subtraction.

In cases where arithmetic subtraction cannot be used to construct a meaningful "difference", we can apply the log-

ical "exclusive or" operation bit by bit. The result of this operation is a 0 for equal bits, and a 1 otherwise, or, on a more practical byte level, we will obtain X'00' if and only if a character equals its counterpart in the reference unit. Characters which yield a X'00' can then be omitted from the record. Of course, we must somehow indicate where and how many characters were omitted. As a practical example, null suppression can be viewed as a differencing technique, where the reference record consists of a fixed string of zeros (X'F0'), and the difference is constructed by exclusive or. Bytes which produce X'00' results are omitted and their absence is indicated either by run-length or by bit map encoding. In general, null suppression is a good "second phase" for this type of exclusive-or-differencing.

Both arithmetic subtraction and exclusive-or-differencing can also be used with dynamically changing reference records. A widely used instance of this type of differencing is a front end compression technique for a sorted directory of keys: The n leading bits of a key that are identical to the corresponding bits of the preceding key are redundant (they do not help to distinguish the keys) and can be omitted. Note that the (n+1)st bit can be omitted as well, because if the keys are equal in exactly n leading bits, they must differ in the (n+1)st bit, which is thus uniquely determined by the preceding key [Gottlieb75]. The reference record used here consists of the preceding key, and the dif-

ference is obtained by applying the exclusive or operation. Run-length encoding is then used to compress the leading string of 0-bits, and the first 1-bit is omitted. Since $n \leq m$, where m is the length of the entire key, the run-length count can be stored in log(m) bits, which often constitutes a considerable saving in space.

As a generalization of the front compression method, we can compress a file sequentially by taking the preceding record as a reference record for differencing. This type of technique has a property which distiguishes it fundamentally from statistical and pattern substituting techniques, and even from differencing with fixed reference record: to decompress a given record, we will have to decompress all records preceding it in the file. While this is hardly a severe constraint if the file is only to be processed sequentially, it makes the application to a direct access file difficult. In this case practicality dictates that the first record of each block that is directly accessed should be left uncompressed (or compressed against some fixed reference), at the expense of the compression factor.

The compression factor achieved by differencing against the preceding record depends heavily on the order of the records in the file. The heuristic of sorting by the largest field [Gottlieb75] may yield acceptable results, but in general only a detailed analysis of the actual contents of

each field will give enough information about orderings that potentially yield an optimal compression factor. In practice, however, the ordering of a sequential file is often dictated by the applications that run against it. If this ordering allows only a moderate compression factor, differencing against the preceding record can prove too ineffective and must be abandoned in favor of other compression techniques. In most cases it is highly undesirable to keep a file compressed in the most suitable order for this purpose and to decompress and resort the entire file before it is accessed by an application program.

Another problem with differencing against the preceding record is reliability. If one record is physically damaged, all succeeding records in the file cannot be decompressed and are lost. In contrast, with pattern substituting techniques only the damaged record itself is not decompressable. The propagation of the damage can be checked to a certain extent by leaving the first record in a block of n records uncompressed, thus limiting the damage to the rest of that block. Depending on the blocking factor n, the compression factor can deteriorate to a point where other compression techniques begin to appear more favorable than differencing against the preceding record.

2.6 Test Runs

For test runs of several compression techniques a partial history file was available which contained all transaction records for the day 20/11/79. This sample file is very small in comparison to the actual history file, but it can be taken to be representative in most aspects. It has the advantage that it can be stored on a VM minidisk, and thus testing of different compression routines and making changes to them could be done quickly and conveniently on-line under VM CMS.

The file consisted of 5951 transaction records, of which 2162 were extended by a variable part, as opposed to 25% in the entire history file. The overrepresentation of the extended records results from the fact that they correspond mainly to reserved material, including previous assignments and final exams for various courses. With a final exam period approaching, these items were at the peak of their year-round demand. The larger than average portion of extended records does not influence results from compression techniques that work on the fixed part only (overall compression factor estimates will even be somewhat pessimistic), and it helps in testing compression techniques that apply to the variable parts of the records.

In selecting candidate compression techniques to be tested, we were guided by a compression factor of 46%, which

is achieved by converting to binary the 70-byte decimal number produced from the fixed part by packing, as outlined in section 2.4. Only fast compression techniques that are likely to do better on the history file were taken into consideration. The major representative of Statistical Encoding, the Huffman encoding scheme, was not implemented for testing, because of the predictably large amount of processor time it uses in comparison with other methods, and because compression factors reported for this technique in [Gottlieb75] are only slightly better than 46%. We first concentrated on compressing the fixed part of the records, as this makes up more than 90% of the data. All compression routines that were tried removed as a first step the ten hyphens contained in the date fields, transforming the fixed part into a numerical string of length 139.

The packing technique, as described in section 2.4, yields a compression factor of 52% for the entire history file. In the fixed parts of the sample file, we found more than 45% zeros occurring in strings of length greater than two. These zeros are still present in the packed version of the file, and null suppression suggests itself to compress the fixed part further. Because the packing routine represents two characters in one byte, a following null suppression algorithm would have to work on a half-byte level. In an actual implementation, we therefore reversed the order of the routines to avoid this difficulty. In summary, the

first candidate compression program, RUNPACK, consisted of the following steps:

1) Remove hyphens in date fields
2) Replace runs of zeros by combined flag/count byte, stored in two consecutive right half-bytes
3) Pack compressed variable length version of fixed part
4) Append variable part unchanged, if it exists.

Step 1) is implemented in a straightforward way. Since the hyphens have fixed offsets from the start of the record, a predetermined pattern is moved into a work area and according to this pattern a TR instruction collects all bytes of the current record into the work area except the ten hyphens. In step 2), a run of zeros is replaced only if its length exceeds two bytes, since only then can compression be achieved. It is sufficient to start looking for zero runs at the last byte of field 3, because before this, only two consecutive zeros can occur. In the sample file we found 44786 strings of zeros, only one of which was longer than 63 bytes. We therefore reserved six bits in the flag/count byte to indicate the length of a zero string, and set the two leftmost bits to one. A half-byte containing a hexadecimal digit from C to F is thus uniquely identified as the left half of a flag/count byte. This byte must initially be stored in two right half-bytes in order to "survive" the packing step. Zero runs longer than 63 bytes are encoded in

several successive flag/count bytes, as needed. Step 3) packs the run length encoded fixed part, which now has variable length and whose end is indicated by the contents of a register. The two halves of a flag/count byte will be adjacent after packing, although they might be separated by a byte boundary, which causes only insignificant inconvenience for decompression. Finally, step 4) appends the extended part, if it exists, to the compressed fixed part. A program listing of RUNPACK can be found in the appendix.

Test runs with this compression routine against the sample history file showed an overall compression factor of 38.14%, which is somewhat pessimistic for the entire history file because of the untypically large proportion of extended records in the sample file. Assuming, more realistically, that only every fourth record is extended, a compression factor of 35.76% would have been achieved. The fixed part of the records was compressed with a factor of 29.9%. A lower bound for this is 23.3%, under the assumption that zeros make up 50% of the fixed part and they all occur in maximum length runs.

The test runs were timed in terms of internal timer units, one unit being equal to 26.04166 microseconds. The processor time used to compress the sample file was roughly 211000 timer units, or 5.49 seconds. This time includes the time used to read the data from disk, which was timed sepa-

rately at approximately 93000 timer units, leaving for actual compression 118000 timer units, or 3.1 seconds. With this technique, we are thus likely to compress one year's volume of the history file in less than 20 minutes processor time by a factor of 35.76%.

As a next phase we tried a compression technique that would also take into account the variable length part of the extended records. The obvious choice for this was a type of differencing method, working on the file sorted by accession number (field 30). The selection of this ordering was made easy by a number of advantages with respect to both processing and compression: Most processing against the history file will be via accession number [Damon], which is facilitated by this ordering, and identical extended parts can be made to appear in consecutive records by a subsort on item number (field 5). To begin with, we tested a routine that implemented a rather crude exclusive-or-type differencing technique. Differencing was done against the preceding record on the field level, and absence of fields in the compressed record was indicated in a bit map. In detail, the routine NFDIFF implemented the following steps:

1) Remove hyphens in the date fields
2) Construct difference against preceding record by exclusive or
3) Locate fields that differ from zero and set corresponding bits in bit map to one

4) Concatenate bit map and fields corresponding to non-zeros to form output record.

Step 1) is identical to step 1) in RUNPACK. In step 2), the difference is calculated in a work area between the current record and the preceding one, which has been saved in uncompressed form in a special area. The length fields in the extended part were treated as part of the corresponding data fields, so that the extended part is viewed as a collection of only three fields. The first record in the file is processed against a dummy record containing all zeros (X'F0') in the fixed part, and blanks in the variable part. Step 3) looks in the work area for nonzero bytes (≠X'00') using a TRT instruction. When a nonzero byte is found, the position in the bit map of the corresponding bit for its field is looked up in tables, and the bit is set to one. In step 4), the start and length of each field are determined from tables and the entire field is appended to the current end of the output record. A program listing of NFDIFF is included in the appendix.

Test runs with NFDIFF against the sample file showed a slightly higher compression factor (39.77%) than that for RUNPACK. This results from the fact that the compressed fixed part of the records still contains a considerable amount of zeros, which is caused partly by the field level operation of NFDIFF, and partly by the fact that complete fields containing only zeros are kept when they are differ-

entiated against nonzero fields. The good compression result for the extended parts could not offset this disadvantage. Note, however, that the compression factor achieved by NFDIFF on the sample file is not as easily projected for a year's volume of the history file as the result for RUNPACK. In fact, it can only be taken as a very loose upper bound for the factor that can be achieved in actual production. Identical extended parts appear in the sample file in a maximum of eight consecutive records, the average being less than five, while in a sorted year's volume identical extended parts will occur consecutively in several hundred records. The elimination of all these except for the first occurrence will have a much greater impact than in the sample file and will sharply improve NFDIFF's compression factor. The processor time used by NFDIFF was (within the limits of error in measurement) equal to the time used by RUNPACK.

Because compression of the extended part of the record is accomplished far better by differencing than by any statistical or pattern substituting technique discussed above, we concentrated attempts to improve on NFDIFF's compression factor again on the fixed part. We note that fields in the fixed part are still in the zoned numerical format after NFDIFF's treatment. The obvious next phase thus was to append to NFDIFF's four steps a step

5) Pack surviving fields from the fixed part,

resulting in the routine DIFFPACK. Step 5) was implemented in a straightforward way, and inserted into NFDIFF at the logical end of the program. A listing of DIFFPACK is shown in the appendix. Processor time measured for DIFFPACK on the sample file was about 3.5 seconds, which projects to about 22 minutes processor time used to compress a year's volume of the history file. The compression factor, however, was significantly improved to 23.64%. This figure is subject to the same remarks as that for NFDIFF, and will improve when a larger portion of the history file is compressed.

Using the same consideration that lead us from the discussion of packing as a pattern substituting technique to the actual routine RUNPACK, we can precede step 5) of DIFFACK by a step

4a) Replace runs of zeros in the differentiated fixed part by a flag/count byte.

The resulting routine DIFRUNPK can be viewed as a successive application of NFDIFF and RUNPACK, or, in terms of sections 2.4 and 2.5, as a concatenation of pattern substitution, exclusive-or-differencing using bit map, null suppression with run length encoding, and pattern substitution (packing). A program listing of DIFRUNPK is found in the appendix. The processor time used to compress the sample file was increased to about 4.5 seconds, which means roughly 28 minutes to compress a year's volume of the history file. The compression factor achieved by DIFRUNPK was 20.15% for the

sample file, and will still likely be better for a year's volume.

Table 2.1 summarizes the yearly predictions for the several data compression techniques. The test runs show that the history file lends itself to considerable compression by relatively simple and fast routines. Because the history data base is to reside on tape, storage cost saved by compression is of minor importance and can be neglected. The advantage of compression lies in the fact that reading time is largely reduced at the low cost of decompression, which

technique/program	processor time	compression factor
packing	< 10 min	52%
convert to binary after packing	approx 20 min	46%
RUNPACK	20 min	35%
NFDIFF	20 min	< 39%
DIFFPACK	22 min	< 24%
DIFRUNPK	28 min	< 20%

Table 2.1 Predicted run time and compression factor for some compression techniques when applied to a year's volume of the history file

amounts to a net saving in the cost of running an application against the data base. Another advantage is that a larger time span can be recorded on a single tape, reducing the number of tapes to be mounted and thus the elapsed time for long range statistical reports. In the following chapter, we present further methods that address the problem of reducing the read time for applications from a totally different starting point than compression.

3 RECORD PARTITIONING VS REPORT PARTITIONING

3.1 Overview and Applicability of Record Splitting Methods

The records in the history file each contain information that describes a certain transaction in great detail, so that each specific report generator will use only a subset of the fields contained in the history records to produce specific statistics. For example, a report on the usage of a given class of items will not be interested in the number of the terminal where a transaction originally was entered (field 2). Even though compression reduces the overall amount of physical storage units to be transferred from tape into main memory, a reporting program still spends a considerable portion of its time to read-in data which are irrelevant to its purpose. To reduce this overhead, methods have been developed to regroup the fields of a record into several separate records, which are then collected in different files.

Three major representatives of these record splitting methods are the techniques of Benner [Benner67], Eisner and Severance [Eisner76], and Babad [Babad77]. In each model, the fields of the record to be split are distributed over a primary record and zero or more auxiliary, or secondary,

records. Such a system of the primary and secondary files is intended to reside on storage devices which are hierarchically organized. Access to the file system will always be through the primary file, which resides on the fastest, but most expensive, storage device. The general strategy then is to assign the various data fields to the primary and secondary records such that the total system cost is minimized.

For each model, the total system cost includes the storage cost for each data field and the cost for each application to access the data fields it needs to process. The storage cost for a data field depends on its length, its density (i.e., the expected percentage of records in the original file that contain non-default values in this field), and on the characteristics of the storage device to which it is assigned. The access cost for an application depends on the number of files it must access to extract the fields it needs, and on the position of these files within the storage hierarchy. Associated with each application is a weighting factor that takes into account the frequency of access to the file system as well as a notion of priority of the respective application. The following list summarizes the items and characteristics that the models assume to be

known in advance:

d(i) - data fields i = 1,...,n

q(i,s) - probability that data field d(i) is of size s

p(i) - density of field d(i), i.e. probability that field d(i) contains a non-default value

A(j) - application running against the file system j = 1,...,m

val(j) - relative importance of application A(j)

a(i,j) - frequency of access to field d(i) by application A(j)

ACC(S) - access cost per unit for storage device S S = 1,...,t

ST(S) - storage cost per unit for device S

While all three record splitting methods take these characteristics into account, they differ in the emphasis laid on them and in the way in which the characteristics are considered in the total cost formula. A detailed description of these methods can be found in a recent essay [Lau78]. For our purpose it is sufficient to note that data items that are expected to contain default values rarely and that are accessed often by applications with relatively high importance are stored in the primary file, while less dense and less frequently required items, as well as items that are accessed by less important applications only, are pushed off to secondary files.

Much of the gain achieved by these methods stems from the underlying assumption of a hierarchy of storage devices with increasing capacity and access time, but decreasing storage cost. Because, in the design of the history data base, we are restricted to only one storage medium, namely tape, all secondary files as proposed by the record splitting methods have the same access speed and storage cost, and the gain based on a storage hierarchy vanishes. Moreover, the implicit use of pointers, especially in Babad's model, suggests an implementation on direct access storage devices, and is rendered more difficult, or even impossible, on tape.

The accuracy of the results obtained when optimizing the design according to the overall cost formula of any of the record splitting methods depends heavily on the accuracy of the statistical input data. A relatively small error in estimating the required statistics may lead to an "optimal" design which is far from being optimal in actual production. While, in the case of the history file, we can easily determine the length for fixed fields or the statistical distribution for variable fields, and even a sufficiently exact probability for a field to contain a default value, we cannot even hope to approximate the total application activities of the data items. As pointed out earlier, the set of reports required by the library's management is not fixed, but rather will be growing to contain reports which are undefined as of today. On the other hand, reports considered useful today may later decline in their importance.

In the design of the history data base, we thus must view the set of users, or applications, as being highly dynamic and volatile. The assumption of a relatively static system being violated in our design problem, we are in no position to provide the most important statistics for the record splitting methods, namely a list of the relative importance of each of n application programs running against the data base, and a measure of total activity for each data item in the history record.

Another difficulty is illustrated by the following realistic case. Most queries against the history data base will be made using as selection criteria only fields from a very small subset of all fields. Although we cannot estimate the activity of these fields within any reasonable error margin, we can safely predict that their activity will be high relative to the activity of fields not used as selection criteria by most applications. Assume that such a field is assigned to the primary record. It is now quite conceivable that a single application of moderate relative importance uses this field to select the records it has to extract (thereby contributing to the activity of that field), and that the fields it is processing are stored in some secondary file. Because of the sequential nature of tape, processing time and cost may warrant that we repeat the selection criterion for this particular application in the secondary file. As a result, the primary file can be by-

passed and the application has only to read the secondary file, which contains its pertinent data. This duplicating of data in different files, however, is beyond the scope of the above record splitting methods.

In summary, the record splitting techniques proposed in the literature do not apply to the design of the history data base for two reasons: 1) With very cheap and conceptually simple techniques we can compress the history file to an extent that allows us to repeat every data item five times without exceeding the size of the original file, and 2) because all files of the data base will reside on tape, the notion of a primary file does not apply to our problem. These facts give us much more flexibility in choosing a design for the history data base than the previously discussed record splitting methods assume.

3.2 An Application Partitioning Model

Although the record splitting methods reviewed in the previous section do not seem to be applicable to the design of the history data base, we do want to reduce the overhead of reading irrelevant data. We therefore still want to organize the data base into several files, but under different assumptions (as discussed above) and aiming at a different goal. The difference in the design goal manifests itself in a fundamental difference in the notion of overall system

cost. We will largely disregard the storage cost of the various files, and concentrate on improving the access cost (speed) for individual applications. In addition, we must include in the total cost, the cost of a structuring program that creates the different files, because this program will have to be run regularly to incorporate newly generated portions of the raw history file into the data base.

Rather than partitioning the set of data fields into segments that appear in exactly one file and forcing the applications to access several files in order to extract their relevant data, we will take a reversed approach. We will partition the applications into user groups within which each needs to access one common file only, and we will allow the data fields to be replicated in all files in which they are needed. The partitioning must be such that applications that have many required data fields in common fall into the same group. Each single application will have a certain overhead in reading irrelevant data, but will share the cost of producing its input file with all other applications in its group. The total system cost formula thus comprises the cost of the structuring program that produces the different group files, and the total overhead cost for all applications.

Both components of the total system cost are strongly dependent on the granularity of the partition, or, equiva-

lently, on the number k of files to be generated (assuming that for each k we can find a minimal overhead partition). While the total application overhead cost Ov(k) decreases monotonically and smoothly with increasing k, the cost Cr(k) of creating the files increases with k and has discontinuous jumps at certain points. The explanation for this is as follows: the structuring program takes as input the raw history file and creates from it the group files. If we assume that t+1 tape drives are allocated to this process, all of which are served by the same channel, t group files can be produced simultaneously during one pass over the history file. The increase of Cr(k) in an interval [n*t+1,n*t+t] (n≥0) is thus very slight, as it reflects a relatively small increase in the amount of processing time plus the cost of an increasing number of channel programs to be executed. However, each time k crosses a boundary from n*t to n*t+1, the history file must be rewound and a complete new run of the structuring program must start. Indicative graphs of the functions Ov(k) and Cr(k) are depicted in figures 3.1 and 3.2, respectively.

In order to present our application splitting model more formally, we denote the set of all data fields by D={d(i) ; i=1...m}, where each data field d(i) is of size s(i) (i=1...m). The set of fields that application A(j) must access is denoted by the subset D(j) of D (j=1...n). We then define the directed labelled graph G=(V,E) as fol-

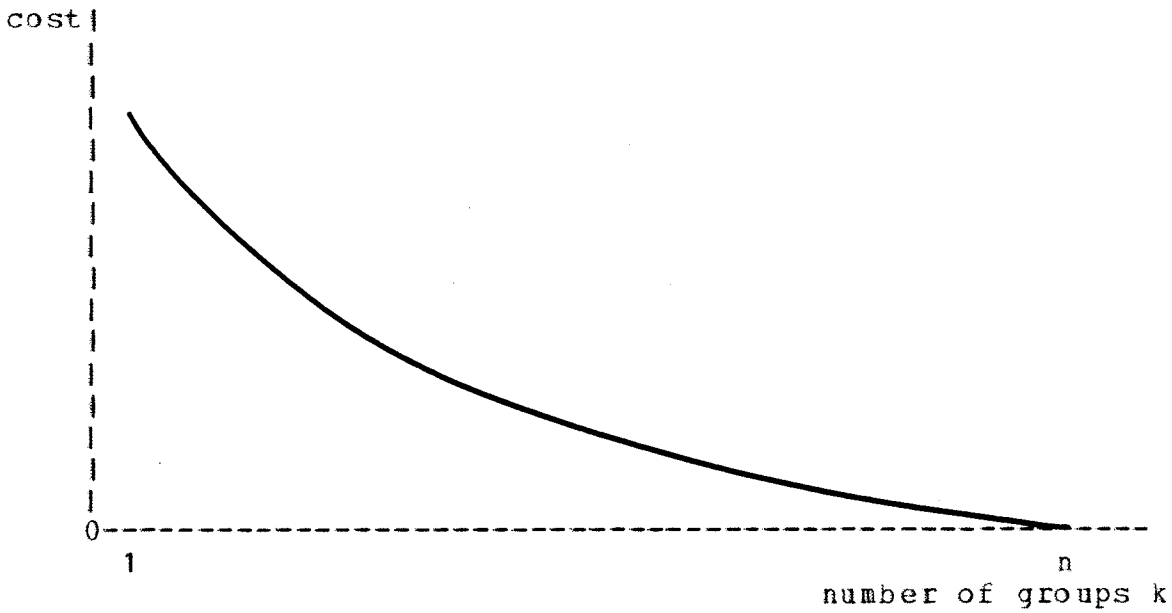

Figure 3.1 Cost of overhead in reading irrelevant data as a function of the granularity of the report partition

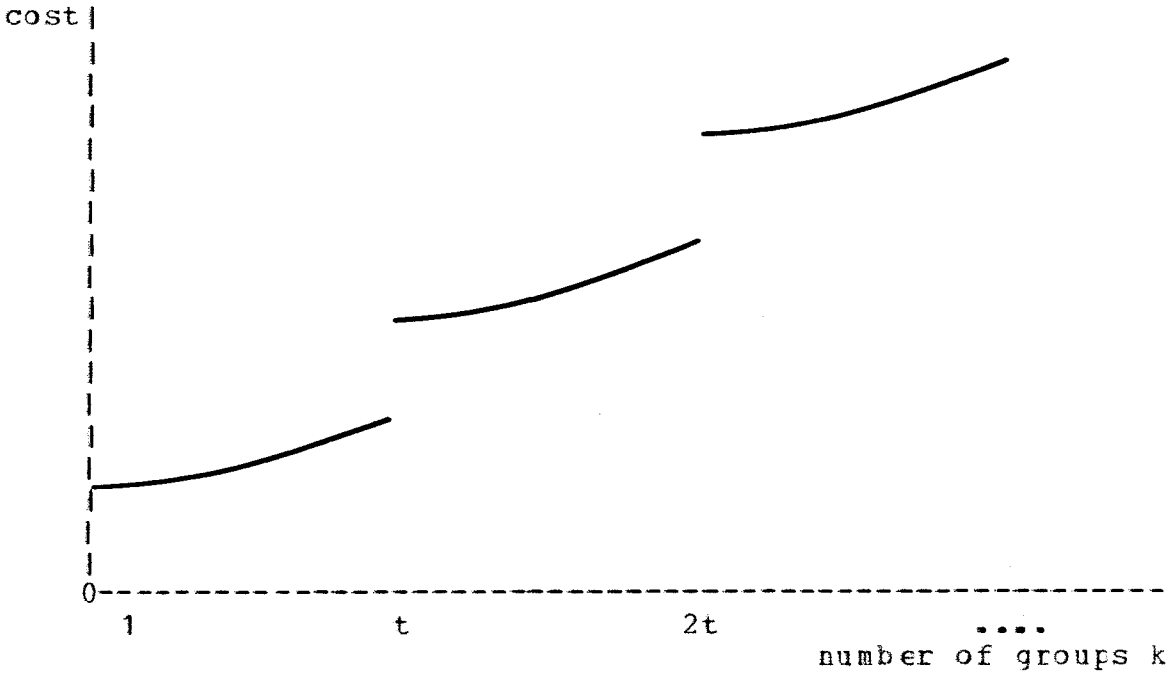

Figure 3.2 Cost of the structuring program as a function of the granularity of the report partition

lows: the set of vertices is the power set P(D) of D. Each vertex v has associated with it a weight w(v)=|{A(j) ; D(j)=v,j=1...n}|, where |X| denotes the cardinality of set X. Thus w(v) indicates the number of applications that must access exactly the data fields contained in v. (If desired, w(v) can be arbitrarily assigned to reflect some measure of relative importance of the applications by defining w(v) = SUM val(j), such that D(j) = v, j=1...n. If we set val(j) = 1 for all j, our simpler definition is equivalent to this one and is thus deemed sufficient to demonstrate the general idea.) An edge is directed from v(i) to v(j) if and only if v(i) is contained in v(j), and is labelled by e(v(i),v(j)) = SUM s(k), such that d(k) is in v(j) - v(i). Thus e(v(i),v(j)) indicates the total length of data fields that v(j) has in excess of v(i). For an application that needs to access the data fields in the set u, but has available only a file containing the larger set v (such that v includes u), the overhead then is equal to the label on the directed edge from u to v. We define a set of distinguished vertices as N = {v ; v=D(j), j=1...n}, i.e. the set of vertices that are equal to D(j) for some j. An example of this model is shown in Figure 3.3.

For a fixed k, we can now find the total overhead Ov(k) by the following steps:

1) select k vertices from the graph G, such that there exists a directed edge from each vertex in N to at least one selected vertex

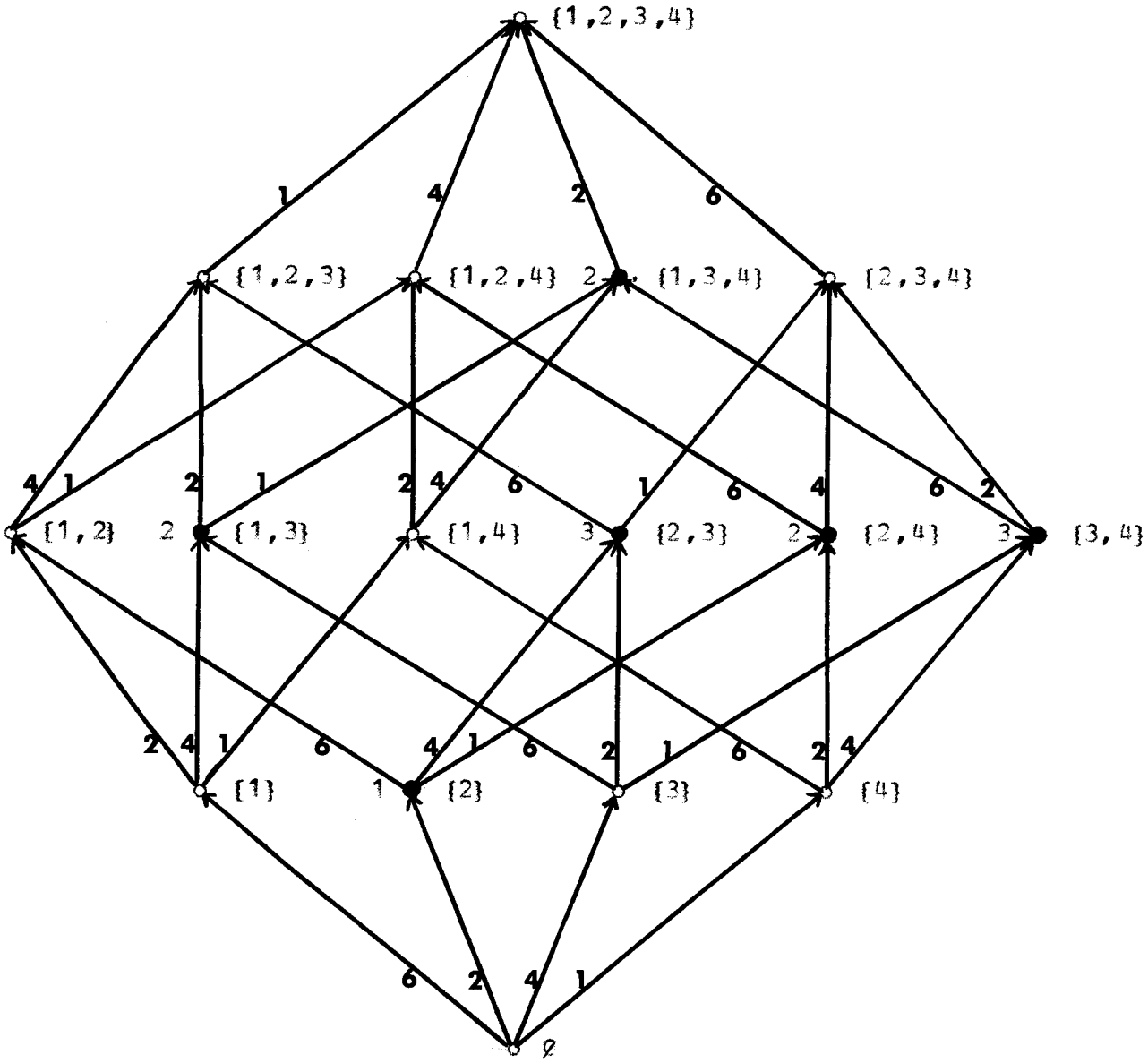

Figure 3.3 Example of the Application Partitioning Model

D = {1,2,3,4} ; N = {{2},{1,3},{2,3},{2,4},{3,4},{1,3,4}}; s(1) = 6, s(2) = 2, s(3) = 4, s(4) = 1; Numbers on the left of vertices indicate nonzero weights. Vertices c have weights zero. Edges that can be deduced from transitivity are omitted for clarity. For k=3 the selected vertices are {1,3,4}, {2,3,4}, and {2,4}, the total overhead is Ov(3) = 1*1 + 2*1 + 3*1 + 2*0 + 3*2 + 2*0 = 12.

2) calculate the sum of o(v)=e(v)*w(v) for all vertices v in N, where e(v) is the minimum of the labels on edges from v to the selected vertices
3) repeat step 2) for all selections of k vertices from graph G that satisfy the condition in 1)
4) the minimal sum encountered in step 2) is the total overhead.

Note that this model assumes that a fixed set of applications and their required data fields are defined before the algorithm can be executed. However, we can still achieve a high degree of flexibility by incorporating a procedure that implements this algorithm into the structuring program that produces the group files. A description of the set of applications thus needs to be specified only before each run of the structuring program, and a change in report requirements is reflected immediately in the following data base generation. (How to cope in practice with application partitions that differ for successive generations will be discussed in the following chapter.)

The design problem then is to find a partitioning of the set of applications into k groups such that the total cost Cr(k) + Ov(k) is minimized. Cr(k) can best be determined by measuring trial runs of the structuring program for k=1,...,t. The function values can then be stored for k=1,...,n, and remain valid until the number t+1 of tape

drives allocated to the structuring process is changed. The function Ov(k) must be calculated for each run of the structuring program, for which the report requirements have changed. For each k in the interval [1,n], Ov(k) must then be determined by the above algorithm in order to minimize the total cost.

Thus minimizing the total cost function may itself prove quite costly, as the algorithm to determine Ov(k) will consume a considerable amount of processor time. In addition, it must be run n times, because a solution for k vertices cannot be determined based on a solution previously found for less than k vertices. In the following section we will present a fast heuristic method to determine a partition of the set of applications and its cost, which can be used as an approximation for Ov(k).

3.3 A Heuristic Algorithm For Report Partitioning

A graph resulting from our application partitioning model has the structure of the relational diagram of the partially ordered set $(P(D),\leq)$, where the partial order $\leq$ denotes the usual relation of set inclusion. The integer labels on the edges from $\emptyset$ (the empty set) to the atoms d(i) (i=1...m) uniquely determine the labels on all edges in the entire graph. The label e(u,v) on an edge from vertex u to vertex v is simply the sum of the labels on all edges from $\emptyset$ to atoms

d, such that d is in v - u. It follows for vertices u,v,w,z that 1) $u \leq v \leq w$ implies $e(u,w) = e(u,v) + e(v,w)$, and 2) $u \leq v$, $u \leq w$, $v \leq z$, $w \leq z$ implies $e(u,v) = e(w,z)$ and $e(u,w) = e(v,z)$. Each vertex v has associated with it a weight $w(v) \geq 0$. The set of distinguished vertices is $N = \{v \; ; \; w(v) > 0\}$.

A problem instance thus is described by a tuple $P = (D, \{e(\emptyset, d(i)) \; ; \; i=1...m\}, \{(v, w(v)) \; , \; w(v)>0\}, k)$. A solution to the problem P is a pair (S,c), where S is a subset of P(D), the set of selected vertices, $|S| = k$, and

$$c = \sum_{u \text{ in } N} \left(\min_{v \text{ in } S} e(u,v) \right).$$

An optimal solution (S,c) for problem P is a solution with minimal c over all solutions for P.

At this time, no inherent ordering of the solution space $\{(S,c)\}$ has been discovered that could direct the search for an optimal solution. We therefore are not able to present an algorithm that produces an optimal solution in less time than a brute force exhaustive search approach as outlined in section 3.2. In the following, we outline a heuristic method to determine a feasible solution (S,c) for a problem P.

We start with a list SLIST of all vertices in N. Two vertices u and v are repeatedly replaced in SLIST by their "father" $\langle u,v \rangle := u+v$ (where + denotes set union), the vertices being chosen such that $\langle u,v \rangle$ adds the least cost of all possible fathers of vertices in SLIST. We stop when

|SLIST|≤k. The vertices contained in SLIST are the selected vertices: S = SLIST.

A selected vertex r thus can be described recursively as

$$r = v \quad \text{or} \quad r = \langle s,t \rangle,$$

where v is a vertex in N, s and t are selected vertices. Note that this notation incorporates which vertices were successively replaced by a selected vertex. The cost c of a solution (S,c) is the sum of the added costs c(<s,t>) that are generated when two vertices s,t are replaced by their father <s,t> during the run of the algorithm. Replacing s and t by <s,t> contributes the cost

$$c(\langle s,t \rangle) := cf(s) * e(s,\langle s,t \rangle) + cf(t) * e(t,\langle s,t \rangle),$$

where the cost factor cf(r) is defined as

$$cf(v) := w(v)$$
$$cf(\langle s,t \rangle) := cf(s) + cf(t),$$

and indicates the number of users (or their total importance) currently requiring access to r.

The algorithm consists of the following steps:

1) SLIST := N ; COST := 0 ;
2) if |SLIST| ≤ k
   then return (SLIST,COST) ;
3) CLIST := { (<u,v>,c(<u,v>)) ; u,v in N } ;
4) loop;
5) select from CLIST a tuple (<s,t>,c(<s,t>)) with minimal c(<s,t>) ;
6) delete from SLIST the vertices s and t ;
7) add to SLIST the vertex <s,t> ;

```
 8)      COST := COST + c(<s,t>) ;
 9)      if |SLIST| = k
            then return (SLIST,COST) ;
10)      delete from CLIST all tuples of the form
         (<s,x>,c(<s,x>)) , (<x,s>,c(<x,s>)),
         (<t,x>,c(<t,x>)), or (<x,t>,c(<x,t>)),
         where x is any vertex;
11)      CLIST := CLIST union
         { (<u,<s,t>>,c(<u,<s,t>>)) ; u in SLIST } ;
12)   repeat ;
```

Each application A(j) is now assigned a file containing the set of data items r, which is uniquely determined by 1) r is in SLIST; 2) the proposition SON(D(j),r) holds, where SON is recusively defined as

```
SON(D(j),r)       <==>  D(j) = r
SON(D(j),<s,t>)   <==>  SON(D(j),s)  or  SON(D(j),t).
```

The solution (S,c) produced by the above algorithm is, in general, not an optimal one (except for the trivial cases k=1, k≥|N|, and k=|N|-1.) However, the algorithm provides a fast means to determine a feasible solution; moreover, by omitting step 2) and changing step 8) to

```
8m)  print (SLIST,COST) ; if |SLIST| = 1 then exit ;
```

solutions for all k=1 ...|N|-1 can be produced in one run.

A solution produced by this algorithm could be used as the starting point for a hillclimbing method in order to try to find a cheaper solution. However, this may prove too

ineffective, because very many choices must "blindly" be explored, which results in an expensive search process without the guarantee of success. Further research of the properties of the problem graph and the solution space is needed in order to devise an algorithm of acceptable cost that produces an optimal sclutich.

4 OUTLINE OF PROPOSED DESIGN

4.1 The Management System

Many aspects of the future environment of the history data base are still unclear at this time. After examining various methods to design and improve the data base, we therefore cannot present a final design in concrete form, nor actual programs that would implement such a design. Neither can we give complete functional specifications in the form "input specification - output specification", for which a programmer has to devise the black box that produces the required output from the input. We rather will outline a possible approach to the design problem in view of insights gained in chapters two and three, as well as other considerations not mentioned before, as a guideline for the systems analyst who will have to define the overall system layout.

We propose a logical organization of the history data base whose central component is a management system which comprises all functions that are involved in converting the raw history file covering some time period (henceforth denoted a "generation") into a set of input tapes for the different application groups. The structural arrangement of the functional units is depicted in figure 4.1. This

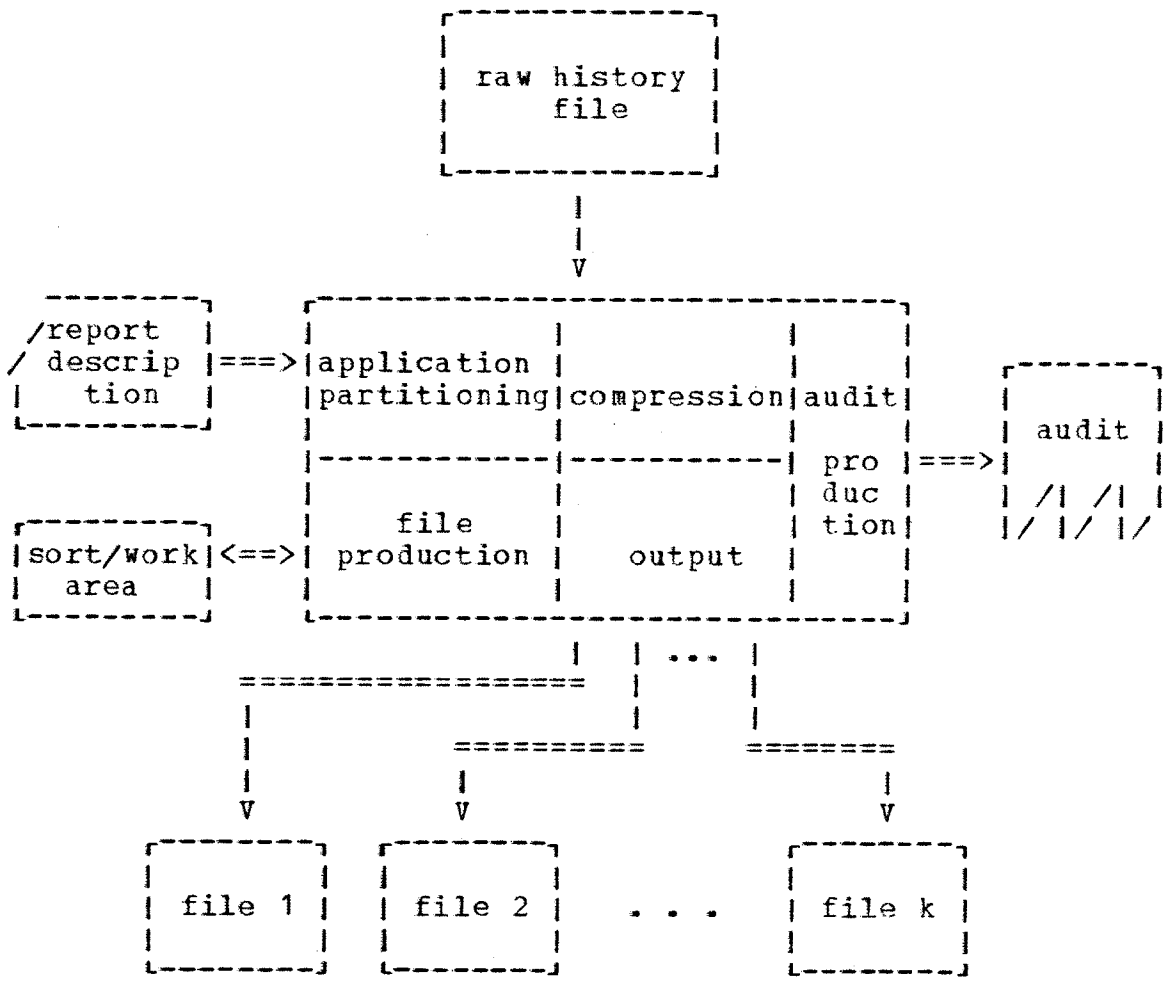

Figure 4.1 Proposed structure of the management system for the history data base.

arrangement guarantees a maximum of flexibility to react to changes in report requirements as well as in the raw history file. In the following we will outline some suggestions and considerations as to how these units should be designed.

Application Partitioning: This component takes as input a description of applications A(j) that are to be against the data base, and the sets of fields D(j) they must access (j=1...n). It is responsible for constructing a partition of the set of applications into groups which use the same input file, as indicated in section 3.2. The audit production component is then informed about the chosen partition and the file production component is instructed as to how many files must be constructed and which data fields they are to contain.

The input description of A(j) and D(j) can be conveniently represented in the form of a matrix (u(i,j)), where the entry u(i,j) indicates the usage of data field i by application j (i=1...33; j=1...n). In the simplest case, u(i,j)=1 if application j must access field i, otherwise u(i,j)=0. Allowing any rational value for u(i,j), on the other hand, enables us to indicate a measure of relative importance for the different applications as well as a notion of priority among the data fields, if this appears desirable.

The cost formula presented in section 3.2 does not reflect a measure of continuity for successive generations in the history data base: Minimizing according to this formula may result in significant changes of partitions as well as changes in the record layout for a given file from generation to generation, thus causing confusion as to which application takes what input file for what generation, and where to find its pertinent data fields in the records of this file. Two approaches to relieve the problem are to construct a data dictionary system which describes the layouts for each generation and to convert all previous generations whenever the layout in revised. A solution will be elaborated in section 4.2.

We conjecture that the application partitioning routine will have the most crucial impact on the overall system performance and cost among the components of the management system.

File Production: This component takes as input a list of file specifications prepared by the application partitioning routine. According to these specifications it extracts data fields from the raw history file and collects them into records, which it then forwards to the compression component. While file production is a minor component of the management system in terms of its logical complexity, it contributes much to the overall system cost, as a result of

the predictably large amount of sorting that has to be done during the preparation of the output files.

Several output files can be extracted in one pass over the raw history file, the limit being dictated by the amount of sorting work space available and the number of tape drives allocated to the output files. The file production component thus has to make a decision as to which specifications best facilitate parallel creation of the respective files. For example, for files which require the same ordering, the union of their sets of required data items can be extracted from the raw history file into the sorting work area (space permitting), and after sorting the respective output files can be extracted directly from there. We regard the implementation of the file production component as being rather straightforward.

Compression: After the input files for the various applications are prepared in the sorting work area by the file production component, they are passed to the compression component for further reduction of their physical size (in case a file need not be sorted, it is forwarded record by record by the file production during a pass over the raw history file). This component will have at its disposal several routines which implement different compression techniques and apply to different types of records. As pointed out in chapter two, we suggest that emphasis be put on very

fast compression techniques with good results rather than on techniques that achieve an optimal compression factor with moderate speed. We conjecture that the general relation between compression factor and processor time used to achieve it is the one depicted in figure 4.2. Note that

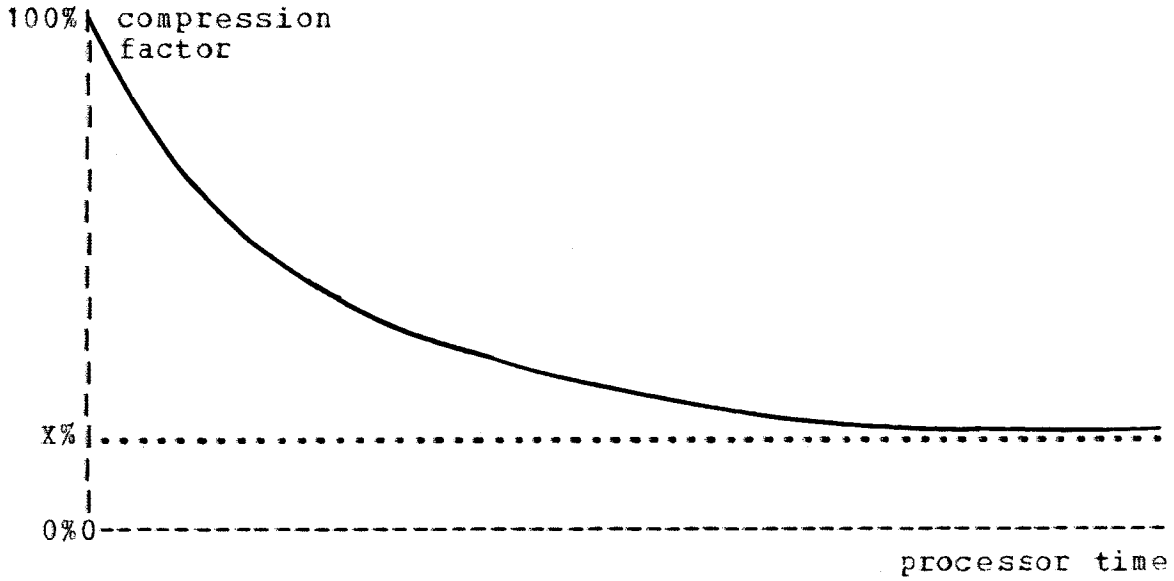

Figure 4.2 Conjectured general relation between processor time used and ccmpression ratio achieved.

while Huffman encoding is optimal under limited assumptions (see section 2.3), no method is known to determine the optimal compression factor X for a given file when those assumptions are violated.

We propose that the initial design of the compression component contain two routines which resemble the programs RUNPACK and DIFRUNPK, and a routine that uses only packing. For reasons of reliability (see section 2.5) we suggest that techniques that employ differencing only be used with files in which the variable fields 31, 32, or 33 make up a major portion. For all other files we propose using a RUNPACK-like procedure, except for files that are known to contain only few default values, where the cheaper and faster packing routine will achieve a compression factor comparable to that of RUNPACK.

Further, more elaborate, compression routines can be added to this component throughout the lifetime of the history data base, as special needs arise or particular properties of files are discovered that can be exploited advantageously, and as available processor time allows.

Output: This is the simplest component of the management system. It merely opens the output files, distributes the results from the compression component to the pertinent data sets, and closes the files after a new generation of the history data base is produced. In fact, we present it as a separate component of the management system only for reasons of clarity. In a practical implementation it is likely to be incorporated as a last step into the compression component.

Audit Production: This component of the management system keeps track of all activity which the implementor deems worthwile tracing. It prints the results on hardcopy at the end of each run of the management system, or writes diagnostic messages during a run, which is essential especially during the testing period for the entire data base. A permanent task of the audit production is to list the partitioning of the applications into groups, as decided by the application partitioning component, along with information about compression routines used and other resulting characteristics of the files generated.

4.2 Transparency

We have repeatedly pointed out that a fundamental property of the future history data base is its dynamic structure. The most important consequence of this dynamic structure is its impact on the life-time cost of the total system. Contrary to a static system, the life-time cost of the history data base cannot be described as a function of merely storage cost and processor time. Besides set-up, testing, and maintenance cost for the system, it must incorporate the cost of the development of application programs to generate the required reports. The development cost for a given report generator in terms of programmer hours used to define the report layout, to write the program, and to set

up the job control, may well exceed the cost of running this application in terms of processor time and access cost for the data. In the following, we will demonstrate how a considerate design of the system can help to keep down the cost of future program development.

The files produced from the raw history file by the management system contain data in compressed form for specific groups of applications. As explained in section 4.1, the compression technique applied to a given file depends on the contents of its data fields, which in turn may vary over different generations. Instead of burdening every application, and thus its programmer, with the routine task of determining the compression technique used for a specific file and then decompressing it accordingly, this routine can be designed once and for all as an interface between the file system and the application programs. It can then be invoked via a special GET macro, with a file name as parameter, and return appropriately decompressed records ready for processing. Besides the advantage of the saving in programming effort, this interface makes the compression of files transparent to the applications, thereby providing a higher degree of independence for both these applications and the file system. This independence facilitates fast and easy adding and changing of routines in the compression component, because the corresponding changes to the decompression routines must only be made in the interface, while no application program needs to be aware of them.

Arguments similar to those used above hold for the transparancy of the application partition for a given data base generation. If we extend the interface by a component that has access to a list of descriptions for all files in the system (the data dictionary), it will be sufficient for an application program to state the data fields it has to process. The interface can then decide automatically which file is best suited to the application's needs and provide for its allocation. The data dictionary can be kept in a file that is created and updated by the audit production component of the management system, which is easily extended accordingly.

The advantage of such an arrangement is obvious: the application partitioning component of the management system can be allowed a free hand without causing the confusion mentioned above. Moreover, applications defined later on which still have to access files from previous generations can be accomodated automatically with the pertinent files that cause the least overhead. The alternative approach to achieve this advantage, namely reformatting all previous generations (see section 4.1), is much more expensive in terms of processing time required for the management system, as it amounts to reprocessing all previous raw history files. In addition, all existing applications would have to be partly rewritten to adapt to changing file layouts. We therefore propose to include in the data base design an

interface between the file system and the applications as outlined above. A structural diagram of the complete history data base system is shown in figure 4.3.

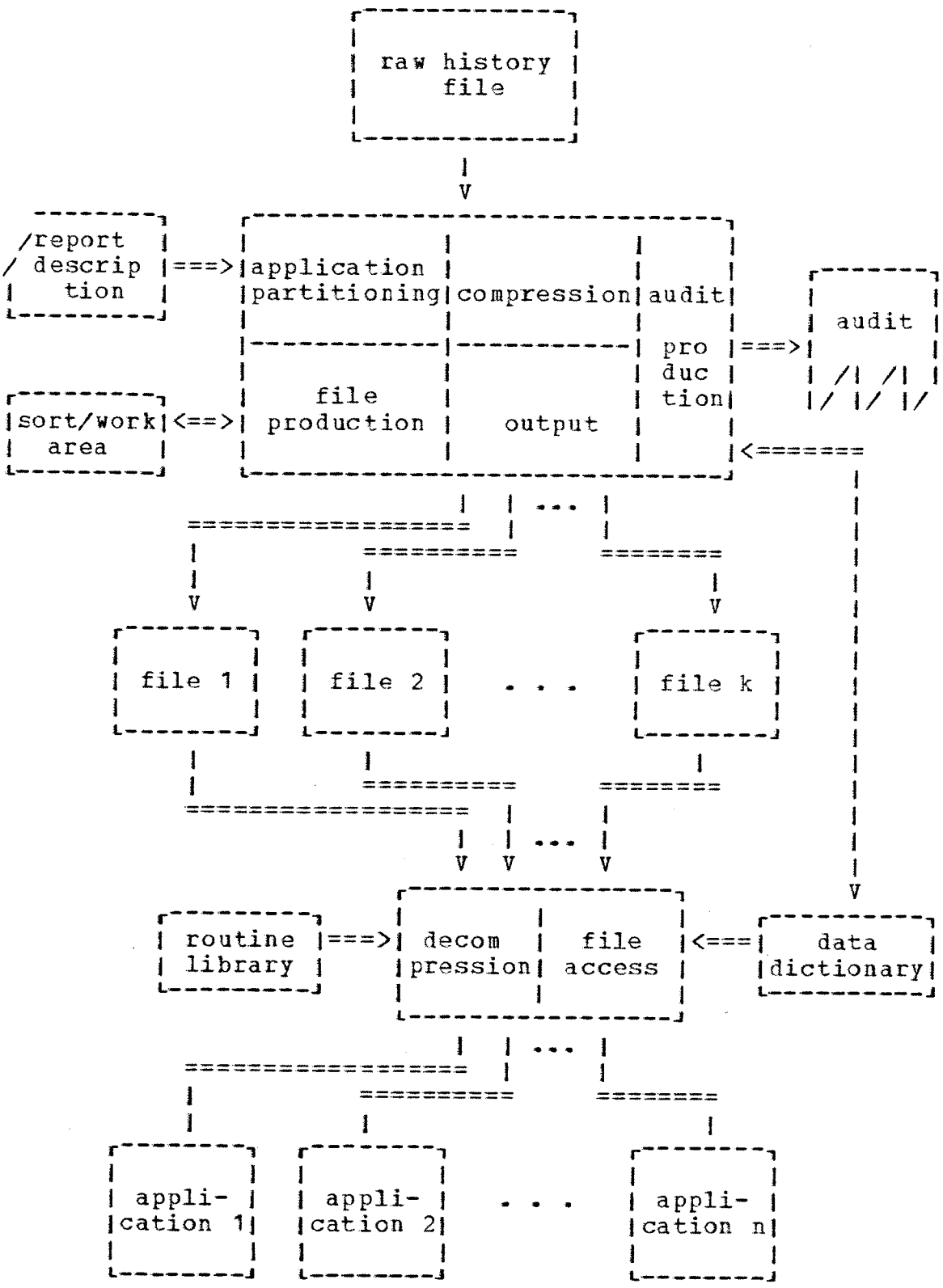

Figure 4.3 Proposed structure of the history data base system.

5 CONCLUSION

The design of the history data base had to take into account two major constraints: the data base is to reside on tape, and the applications running against it are yet to be defined and will vary with time. In addition, the amount of data will grow during the entire lifetime of the system. The confinement to tape led us to disregard in our considerations the storage cost of the data base and to concentrate on improving the access speed for the applications. This was achieved by devising an application partitioning model and by compression. While the application partitioning model gives us the means to reduce the amount of irrelevant data each report generator has to read, compression increases the density of the stored information, thus further reducing the time to transfer data into main memory.

The problems presented by the dynamic growth and the changing report requirements are solved in the design of the management system. It can be run at fixed or variable time intervals (which must be empirically determined after the system is implemented) to incorporate a new generation into the history data base. At each run the applications that are currently of interest can be specified to produce an application partition with minimal overhead. An interface between

file system and applications provides transparancy of compression and application partitions, which makes the applications independent of file layout and data representation and thus saves program development and maintenance cost.

An implementation of the proposed design can be developed in a straightforward manner and can become operational after a short testing period. The dynamic structure of the system together with the transparancy of compression and application partitioning facilitates changes and improvements in all system components throughout the lifetime of the history data base.

REFERENCES

[Abramson63] -

Abramson, N., Information Theory and Coding. McGraw-Hill, New York, 1963

[Babad77] -

Babad, J.M., A Record and File Partitioning Model. CACM 20, 1 (Jan. 1977), pp. 22-31

[Benner67] -

Benner, F.J., On Designing Generalized File Records for Management Information Systems. Proc. AFIPS 1967 FJCC, AFIPS Press, Montvale, N.J., pp. 291-303

[Damon] -

Damon, Gene, Assistant Librarian for Systems, Library, University of Waterloo. private communications, 1979-1980

[Eisner76] -

Eisner, M.J. and D.G. Severance, Mathematical Techniques for Efficient Record Segmentation in Large Shared Data Bases. JACM 23, 4 (Oct. 1976), pp. 619-635

[GEAC78] -

GEAC Computer Corporation, GEAC Library System. Toronto, Feb. 1978

[Gottlieb75] -

Gottlieb, D. and S.A Hagerth, P.G.H. Lehot, and H.S. Rabinowitz, A Classification of Compression Methods and Their Usefulness for a Large Data Processing Center. AFIPS 44, pp. 453-458

[Hu71] -

Hu, T.C. and A.C. Tucker, Optimal Computer Search Trees and Variable Length Alphabetical Codes. S.I.A.M. Journal of Applied Mathematics, 21, 514 (1971)

[Huffman52] -

Huffman, D.A., A Method for Construction of Minimal Redundancy Codes. Proc. I.R.I.E., 51, pp.1098-1101 Sept. 1952

[IBM Principles of Operation] -

Principles of Operation (GA 22-7000-4), IBM Systems Products Division, Product Publications, Dept. B98 Poughkeepsie, N.Y., 1974

[Lau78] -

Lau, C.K., Record Partitioning for Data Base Management Systems, Master's Essay, University of Waterloo, Dec. 1978

[Library Systems] -

System Development Dept., The Library, University of Waterloo, Tape Lay-out for the History File. Internal Document

[Marron67] -

Marron,B.A. and P.A.D. De Maine, Automatic Data Compression. CACM 10, 11 (Nov. 1967), pp. 711-715

[Mulford71] -

Mulford, J.B. and R.K. Ridall, Data Compression Techniques for Economic Processing of Large Commercial Files. ACM Symposium on Information Storage and Retrieval, 1971, pp. 207-215

[Ruth72] -

Ruth, S.S. and P.J. Kreutzer, Data Compression for Large Business Files. Datamation, Sept. 1972, pp. 62-66

[Santoro80] -

Santoro, N., Efficient Abstract Implementations for Relational Data Structures. Research report CS-80-21, University of Waterloo, April 1980

[Snyderman70] -

Snyderman, M. and B. Hunt, The Myriad Virtues of Text Compaction, Datamation, Dec. 1 1970, pp. 36-40

[Watt80] -

Watt, R., Memorandum to Users of OSBATCH, Dept. of Computing Services, University of Waterloo, Apr. 12 1980

APPENDIX

In the following, we present the source codes of the compression programs RUNPACK, NFDIFF, DIFFPACK, and DIFRUNPK, as described in section 2.6. All programs were run as independent mainlines. Instructions pertinent to statistical and timing measurements are not shown in the listings, which otherwise present the programs as they were used in the test runs. We do not claim that these programs are of maximal efficiency in terms of storage or processing time requirements.

RUNPACK

```
RUNPACK  START

*** HOUSEKEEPING,BASEREG,OPEN FILES
************************************

         STM    14,12,12(13)            HOUSEKEEPING
         BALR   3,0                     SET UP BASE REGISTERS
         USING  *,3                     .
         ST     13,SAVE13               HOUSEKEEPING
R1       EQU    1
RW1      EQU    4
RWEND    EQU    5
RW2      EQU    6
RWPTR    EQU    7
ROUTPTR  EQU    8
RC       EQU    11
         FSOPEN FSCB=LIB                OPEN HISTORY FILE
         FSERASE FSCB=OUT               CREATE FRESH
         FSOPEN FSCB=OUT                .OUTPUT FILE

*** INITIALIZATION AND INPUT
*****************************
GET      DS     0H
         FSREAD FSCB=LIB,ERROR=EOF      GET RECORD; EOFADDR=EOF
         MVC    WAREA(139),PATTERN      PREPARE EDITING
         TR     WAREA(139),INBUF        REMOVE HYPHENS

*** RUNLENGTH ENCODING FOR ZEROS
*********************************

         XR     RWEND,RWEND             CLEAR END POINTER
         XR     RWPTR,RWPTR             CLEAR WORK AREA POINTER
         XR     RW1,RW1                 CLEAR WORK REGISTER
         LA     RWEND,WAREA+138         INITIALIZE END OF WAREA
         LA     RWPTR,WAREA+10          POINT TO START OF STUFF
ZEROFIND DS     0H
         CR     RWPTR,RWEND             END OF WORK AREA REACHED ?
         BNL    PCKSTART                .YES, START PACKING
```

```
*    FIND START OF ZERO STRING

          XR     R1,R1                      CLEAR REGISTER USED IN TRT
          TRT    0(127,RWPTR),ZEROTAB       FIND ZERO
          BZ     PCKSTART                   NO MORE ZEROS: DONE
          CR     R1,RWEND                   IS ZERO BEYOND OUR SCOPE ?
          BH     PCKSTART                   .YES, DONE

*    FIND END OF ZERO STRING

          LR     RWPTR,R1                   POINT TO ZERO FOUND
          TRT    1(127,RWPTR),NZEROTAB      FIND NEXT NONZERO

*    COMPUTE LENGTH OF ZERO STRING

          LR     RW1,R1                     GET END OF STRING
          SR     RW1,RWPTR                  SUBTRACT START OF STRING
          CH     RW1,MINSIZE                IS STRING TOO SHORT ?
          BH     NOTSHORT                   .NO
          LA     RWPTR,1(R1)                BUMP POINTER TO STUFF
          B      ZEROFIND                   TRY AGAIN
NOTSHORT  DS     0H
          CH     RW1,MAXSIZE                IS STRING TOO LONG ?
          BNH    RUNLENTH                   .NO, GO ENCODE IT
          LH     RW1,MAXSIZE                GET MAX LENGTH ALLOWED
          LA     R1,0(RWPTR,RW1)            ADJUST END OF STRING

*    INSERT RUNLENGTH INTO 2 RIGHT HALF-BYTES

RUNLENTH  DS     0H
          LR     RW2,RW1                    SAVE LENGTH
          STC    RW1,1(RWPTR)               INSERT LENGTH AT START OF STRING
          SRL    RW1,4                      .
          STC    RW1,0(RWPTR)               .
          OI     0(RWPTR),X'0C'             MARK BYTE AS RUNLENGTH INDICATOR

*    MOVE STUFF TO FRONT AND CALC NEW END OF FIXED PART

          LA     RWPTR,2(RWPTR)             GET NEXT FREE BYTE
          LR     RW1,RWEND                  GET CURRENT END OF RECORD
          SR     RW1,R1                     CALC LENGTH OF MOVE
          EX     RW1,MVC
          SR     RWEND,RW2                  CALC NEW END: SUBTRACT LENGTH,
          LA     RWEND,2(RWEND)             .ADD ZERO COUNT
          B      ZEROFIND                   GO AGAIN
MVC       MVC    0(1,RWPTR),0(R1)
```

```
*** PACKING ROUTINE
********************

PCKSTART DS    0H
         LA    RWPTR,WAREA              POINT TO STUFF TO PACK
         LA    ROUTPTR,OUTBUF+2         POINT TO WHERE TO PACK IT
PACK     DS    0H
         PACK  0(8,ROUTPTR),0(15,RWPTR) PACK STUFF
         LA    ROUTPTR,7(ROUTPTR)       BUMP OUTPUT POINTER
         LA    RWPTR,14(RWPTR)          BUMP WORK AREA POINTER
         CR    RWPTR,RWEND              END OF WORK AREA REACHED ?
         BNH   PACK                     .NO, DO IT AGAIN
         LA    RW1,WAREA-1              CALCULATE LENGTH OF
         SR    RWEND,RW1                .COMPRESSED FIXED PART
         LA    RWEND,1(RWEND)           ADD ONE FOR DIVISION
         SRL   RWEND,1                  CALC LENGTH OF PACKED PART
         LH    RW1,INBUF                PICK UP LENGTH OF RECORD
         SH    RW1,H150                 SUBTRACT LENGTH OF FIXED PART
         BL    OUTPUT                   NO VARIABLE PART: DONE
         LA    RW2,OUTBUF+2(RWEND)      POINT TO FREE BYTE AFTER FIX PT
         EX    RW1,VMOVE                MOVE VARIABLE PART UNCHANGED
         LA    RWEND,1(RW1,RWEND)       CALC LENGTH OF OUTPUT RECORD

*** OUTPUT, RETURN
********************

OUTPUT   DS    0H
         STH   RWEND,OUTBUF             PUT LENGTH INTO RECORD
         LA    RWEND,2(RWEND)           ADD LENGTH OF LENGTH FIELD
         FSWRITE FSCB=OUT,BUFFER=OUTBUF,BSIZE=(RWEND)  PUT RECORD
         B     GET
EOF      DS    0H
         FSCLOSE FSCB=LIB               CLOSE HISTORY FILE
         FSCLOSE FSCB=OUT               CLOSE OUTPUT FILE
         L     13,SAVE13                HOUSEKEEPING
         LM    14,12,12(13)             .
         BR    14                       RETURN TO CONTROL PROGRAM
```

```
*** STORAGE DEFINITION
**********************

VMOVE     MVC   0(1,RW2),INBUF+151
LIB       FSCB  'COMP DATA B',RECFM=V,BUFFER=INBUF
OUT       FSCB  'DIFF DATA B',RECFM=V,BUFFER=OUTBUF
H150      DC    H'150'
MINSIZE   DC    H'2'
MAXSIZE   DC    H'63'
SAVE13    DS    F
OUTBUF    DC    3XL100'00'
          DS    0H
INBUF     DC    4CL100' '
WAREA     DC    X'020304050607080A0B0D0E0F101112131415161718191A'
          DC    X'1B1C1D1E1F202122232425262728292A2B2C2D2E2F303233'
          DC    X'353637383939A3B3C3E3F414243444546474849A4B4C'
          DC    X'4D4E4F505152535455565758595A5B5C5D5E5F606162636465'
          DC    X'666768696A6B6C6D6E6F707172737475777977A7B7C7E'
          DC    X'7F818283848586878889A8B8C8D8E8F909192939495 96'
PATTERN   DC    X'020304050607080A0B0D0E0F101112131415161718191A'
          DC    X'1B1C1D1E1F202122232425262728292A2B2C2D2E2F303233'
          DC    X'353637383939A3B3C3E3F414243444546474849A4B4C'
          DC    X'4D4E4F505152535455565758595A5B5C5D5E5F606162636465'
          DC    X'666768696A6B6C6D6E6F707172737475777977A7B7C7E'
          DC    X'7F818283848586878889A8B8C8D8E8F909192939495 96'
ZEROTAB   DC    240X'00',X'FF',15X'00'
NZEROTAB  DC    240X'FF',X'00',15X'FF'
          END
```

NFDIFF

```
NFDIFF   START

*** HOUSEKEEPING,BASEREG,OPEN FILES
************************************

         STM    14,12,12(13)          HOUSEKEEPING
         BALR   3,0                   SET UP BASE REGISTERS
         USING  *,3                   .
         ST     13,SAVE13             HOUSEKEEPING
R1       EQU    1
RW1      EQU    4
RWEND    EQU    5
RW2      EQU    6
RWPTR    EQU    7
ROUTPTR  EQU    8
RW3      EQU    10
RC       EQU    11
         FSOPEN FSCB=LIB              OPEN HISTORY FILE
         FSERASE FSCB=OUT             CREATE FRESH
         FSOPEN FSCB=OUT              .OUTPUT FILE

*** INITIALIZATION AND INPUT
*****************************

         XR     RW3,RW3               CLEAR WORK REGISTER
GET      DS     0H
         FSREAD FSCB=LIB,ERROR=EOF    GET RECORD; EOFADDR=EOF
         XC     BITMASK(5),BITMASK    CLEAR BITMASK

*    REMOVE HYPHENS

         MVC    WAREA(117),PATTERN    PREPARE FOR TRANSLATION
         TR     WAREA(117),INBUF      TRANSL TILL BEFORE LAST HYPHEN
         MVC    INBUF+2(117),WAREA    SAVE FOR LATER USE
         LH     RW1,INBUF             GET LENGTH OF RECORD
         SH     RW1,H128              CALC (LENGTH OF REST) -1
         EX     RW1,XMOVE             MOVE REST UNCHANGED
         EX     RW1,SMOVE             SAVE ALSO FOR LATER USE

*    CALCULATE DIFFERENCE TO PREVIOUS RECORD

         AH     RW1,H117              GET (ENTIRE LENGTH) -1
         EX     RW1,XOR               DIFFERENTIATE AGAINST PREV REC
         ST     RW1,SL                SAVE LENGTH
         B      DIFFER                GO FIND DIFFERENCE
```

```
*** SET UP VARIABLE PART OF CONTROL TABLES
*******************************************

VSETUP   DS    0H
         MVI   FORK,X'00'                    DISABLE BRANCH AT FORK
         LA    RW1,INBUF+141                 POINT TO VARIABLE PART
         LA    RC,3                          SET COUNTER
FILL     DS    0H
         LH    RW2,0(RW1)                    GET LENGTH OF CALLNO FIELD
         LA    RW2,1(RW2)                    INCLUDE (LENGTH FIELD) -1
         STC   RW2,LENGTH-(INBUF+2)(RW1)     STORE INTO CONTROL TABLE
         LH    RW2,0(RW1)                    GET (CALLNO + LENGTH) - 2
         LA    RW3,MASKTAB-(INBUF+2)(RW1)    START OF FIELD IN CNTL TAB
         EX    RW2,VMOVE                     FILL CNTL FIELD WITH BIT
         LA    RW3,BYTETAB-(INBUF+2)(RW1)
         EX    RW2,VMOVE                     FILL IN OFFSET IN BITMASK
         LA    RW3,LENGTH-(INBUF+2)(RW1)
         EX    RW2,VMOVE                     FILL IN LENGTH OF FIELD
         LA    RW3,OFFSET-(INBUF+2)(RW1)
         EX    RW2,VMOVE                     . OFFSET OF FIELD IN REC
         BCT   RC,ON                         DO IT FOR 3 VARIABLE FIELDS
         B     ADJUST                        WHEN FINISHED, GO AHEAD

VMOVE    MVC   1(1,RW3),0(RW3)

ON       DS    0H
         LA    RW1,2(RW2,RW1)                GET LENGTH OF NEXT FIELD
         AH    RW2,0(RW3)                    OFFSET+LENGTH OF PREV FIELD
         LA    RW2,2(RW2)                    CALC OFFSET OF NEXT FIELD
         STC   RW2,OFFSET-(INBUF+2)(RW1)     STORE INTO CONTROL TABLE
         IC    RW3,M(RC)                     PICK UP BIT FOR THIS FIELD
         STC   RW3,MASKTAB-(INBUF+2)(RW1)    STORE IT INTO CONTROL TABLE
         IC    RW3,B(RC)                     PICK UP OFFSET IN BITMASK
         STC   RW3,BYTETAB-(INBUF+2)(RW1)    STORE INTO CONTROL TABLE
         B     FILL                          GO FILL ENTIRE CNTL FIELD

*** FIND FIELDS DIFFERENT FROM PREVIOUS RECORD
***********************************************

DIFFER   DS    0H
         MVI   FORK,X'F0'                    ENABLE BRANCH AT FORK
         LA    RW3,WAREA+139                 POINT TO END OF FIXED PART
         XR    RWPTR,RWPTR                   CLEAR WORK AREA POINTER
         XR    R1,R1                         CLEAR REGISTER USED IN TRT
         LA    ROUTPTR,PAREA                 POINT TO OUTPUT AREA
         L     RWEND,SL                      PICK UP LENGTH OF RECORD
         LA    RWEND,WAREA+1(RWEND)          INITIALIZE END OF WAREA
         LA    RWPTR,WAREA                   POINT TO START OF STUFF
```

```
*   FIELDS DIFFERENT FROM PREV RECORD HAVE A BYTE ¬= X'00'

FINDDIFF DS    0H
         TRT   0(256,RWPTR),DIFFTAB    FIND DIFFERENT POSITION
         BZ    OUTPUT                  NO MORE DIFFERENCES: DONE
         CR    R1,RW3                  IS DIFF IN VARIABLE PART ?
         BL    ADJUST                  .NO, GO AHEAD
         CR    R1,RWEND                IS DIFF BEYOND CURRENT END ?
         BNL   OUTPUT                  .YES, DONE
FORK     EQU   *+1                     CONDITION MASK FOR BRANCH INSTR
         B     VSETUP                  MUST SET VARIABLE CONTROL TABLES

*   ADJUST BITMASK

ADJUST   DS    0H
         XR    RW1,RW1
         IC    RW1,BYTETAB-WAREA(R1)   PICK UP OFFSET FOR MASK BYTE
         LA    RW1,BITMASK(RW1)        POINT INTO BITMASK
         LA    RW2,MASKTAB-WAREA(R1)   POINT TO BYTE TO BE ORED
         OC    0(1,RW1),0(RW2)         SWITCH BIT FOR THIS FIELD ON

*   MOVE DIFFERENT FIELD TC OUTPUT

         XR    RW1,RW1
         IC    RW1,OFFSET-WAREA(R1)    PICK UP OFFSET OF THIS FIELD
         LA    RWPTR,INBUF+2(RW1)      POINT TO THIS FIELD IN INBUF
         IC    RW1,LENGTH-WAREA(R1)    PICK UP (LENGTH-1) FOR FIELD
         EX    RW1,MOVE                MOVE DIFFERENT FIELD TO OUTPUT

*   BUMP POINTERS

         LA    ROUTPTR,1(RW1,ROUTPTR)  POINT TO NEXT FREE BYTE IN OUTPU
         LA    RWPTR,WAREA-(INBUF+1)(RW1,RWPTR) NEXT FIELD IN WAREA
         CR    RWPTR,RWEND             BEYOND CURRENT END ?
         BL    FINDDIFF                .NO, DO IT AGAIN
         B     OUTPUT                  DONE

*** OUTPUT, RETURN
******************

OUTPUT   DS    0H
         LA    RW1,OUTBUF+2            GET START OF OUTPUT AREA
         SR    ROUTPTR,RW1             CALC LENGTH OF RECORD
         STH   ROUTPTR,RECL            STORE INTO LENGTH FIELD
         LA    ROUTPTR,2(ROUTPTR)      ADD LENGTH OF LENGTH FIELD
         FSWRITE FSCB=OUT,BUFFER=OUTBUF,BSIZE=(ROUTPTR) PUT RECORD
         L     RW1,SL                  PICK UP ORIGINAL LENGTH
         EX    RW1,MOVEOLD             SAVE PROCESSED REC FOR DIFF'CING
         B     GET                     GO PROCESS NEXT RECORD

MOVEOLD  MVC   XOREA(1),INBUF+2
```

```
EOF       DS    0H
          FSCLOSE FSCB=LIB              CLOSE HISTORY FILE
          FSCLOSE FSCB=OUT              CLOSE OUTPUT FILE
          L     13,SAVE13               HOUSEKEEPING
          LM    14,12,12(13)            .
          BR    14                      RETURN TO CONTROL PROGRAM

*** STORAGE DEFINITION
**********************

XOR       XC    WAREA(1),XOREA
MOVE      MVC   0(1,ROUTPTR),0(RWPTR)
XMOVE     MVC   WAREA+117(1),INBUF+129
SMOVE     MVC   INBUF+119(1),INBUF+129

*   FILE CONTROL BLOCKS

LIB       FSCB  'COMP DATA B',RECFM=V,BUFFER=INBUF
OUT       FSCB  'DIFF DATA B',RECFM=V,BUFFER=OUTBUF

*   BITS AND BYTES FOR FILLING VARIABLE PART OF CONTROL TABS

M         EQU   *-1
          DC    X'8001'
B         EQU   *-1
          DC    X'0403'
H117      DC    H'117'
H128      DC    H'128'
H138      DC    H'138'
SAVE13    DS    F
SL        DS    F
PACKEND   DS    F

*   OUTPUT BUFFER

OUTBUF    DS    0H
RECL      DS    H
BITMASK   DC    5X'00'
PAREA     DC    3XL100'00'
          DS    0H

*   INPUT BUFFER

INBUF     DC    4CL100' '

*   WORK AREAS

WAREA     DS    139C,176C
XOREA     DC    139C'0',176C' '
```

```
*   PATTERN FOR REMOVAL OF HYPHENS USING TR INSTRUCTION

PATTERN  DC    X'0203040506070 80A0B0D0E0F101112131415161718191A'
         DC    X'1B1C1D1E1F202122232425262728292A2B2C2D2E2F303233'
         DC    X'353637383 93A3B3C3E3F414243444546474849 4A4B4C'
         DC    X'4D4E4F505152535455565758595A5B5C5D5E5F606162636465'
         DC    X'666768696A6B6C6D6E6F707172737475767779 7A7B7C7E'
         DC    X'7F818283848586878889 8A8B8C8D8E8F909192939495 96'

*   TABLE FOR TRT

DIFFTAB  DC    X'00'                          (255 NONZERO BYTES MUST FOLLOW !

*   CONTROL TABLE CONTAINS BITS FOR CORRESPONDING FIELDS

MASKTAB  DC    X'8080',3X'40',6X'20',5X'10',9X'08',9X'04',3X'02'
         DC    3X'01',3X'80',6X'40',4X'20',6X'10',4X'08',9X'04'
         DC    3X'02',2X'01',10X'80',3X'40',9X'20',3X'10',X'0808'
         DC    X'04',X'0202',6X'01',6X'80',3X'40',3X'20',3X'10'
         DC    3X'08',8X'04'
VMASKTAB DC    176X'02'

*   CONTROL TABLE CONTAINS OFFSET FOR CORRESPONDING FIELDS

OFFSET   DC    X'0000',3X'02',6X'05',5X'0B',9X'10',9X'19',3X'22'
         DC    3X'25',3X'28',6X'2B',4X'31',6X'35',4X'3B',9X'3F'
         DC    3X'48',2X'4B',10X'4D',3X'57',9X'5A',3X'63',X'6666'
         DC    X'68',X'6969',6X'6B',6X'71',3X'77',3X'7A',3X'7D'
         DC    3X'80',8X'83'
VOFFSET  DC    176X'8B'

*   CONTROL TABLE CONTAINS LENGTH OF CORRESPONDING FIELDS

LENGTH   DC    X'0101',3X'02',6X'05',5X'04',9X'08',9X'08',3X'02'
         DC    3X'02',3X'02',6X'05',4X'03',6X'05',4X'03',9X'08'
         DC    3X'02',2X'01',10X'09',3X'02',9X'08',3X'02',X'0101'
         DC    X'00',X'0101',6X'05',6X'05',3X'02',3X'02',3X'02'
         DC    3X'02',8X'07'
VLENGTH  DC    176X'00'

*   CONTROL TABLE CONTAINS BITMASK-BYTE FOR CORRESPONDING FIELDS

BYTETAB  DC    40X'00',37X'01',36X'02',26X'03'
VBYTETAB DC    176X'03'
         END
```

DIFFPACK

```
DIFFPACK START

*** HOUSEKEEPING,BASEREG,OPEN FILES
************************************
         STM   14,12,12(13)          HOUSEKEEPING
         BALR  3,0                   SET UP BASE REGISTERS
         USING *,3                   .
         ST    13,SAVE13             HOUSEKEEPING
R1       EQU   1
RW1      EQU   4
RWEND    EQU   5
RW2      EQU   6
RWPTR    EQU   7
ROUTPTR  EQU   8
RW3      EQU   10
RC       EQU   11
         FSOPEN FSCB=LIB             OPEN HISTORY FILE
         FSERASE FSCB=OUT            CREATE FRESH
         FSOPEN FSCB=OUT             .OUTPUT FILE

*** INITIALIZATION AND INPUT
*****************************
         XR    RW3,RW3               CLEAR WORK REGISTER
GET      DS    0H
         FSREAD FSCB=LIB,ERROR=EOF   GET RECORD; EOFADDR=EOF
         XC    BITMASK(5),BITMASK    CLEAR BITMASK

*   REMOVE HYPHENS

         MVC   WAREA(117),PATTERN    PREPARE FOR TRANSLATION
         TR    WAREA(117),INBUF      TRANSL TILL BEFORE LAST HYPHEN
         MVC   INBUF+2(117),WAREA    SAVE FOR LATER USE
         LH    RW1,INBUF             GET LENGTH OF RECORD
         SH    RW1,H128              CALC (LENGTH OF REST) -1
         EX    RW1,XMOVE             MOVE REST UNCHANGED
         EX    RW1,SMOVE             SAVE ALSO FOR LATER USE

*   CALCULATE DIFFERENCE TO PREVIOUS RECORD

         AH    RW1,H117              GET (ENTIRE LENGTH) -1
         EX    RW1,XOR               DIFFERENTIATE AGAINST PREV REC
         ST    RW1,SL                SAVE LENGTH
         B     DIFFER                GO FIND DIFFERENCE
```

```
*** SET UP VARIABLE PART OF CONTROL TABLES
*****************************************

VSETUP   DS    0H
         ST    ROUTPTR,FIXEDEND              SAVE END OF COMPR FIXED PT
         MVI   FORK,X'00'                    DISABLE BRANCH AT FORK
         LA    RW1,INBUF+141                 POINT TO VARIABLE PART
         LA    RC,3                          SET COUNTER
FILL     DS    0H
         LH    RW2,0(RW1)                    GET LENGTH OF CALLNO FIELD
         LA    RW2,1(RW2)                    INCLUDE (LENGTH FIELD) -1
         STC   RW2,LENGTH-(INBUF+2)(RW1)     STORE INTO CONTROL TABLE
         LH    RW2,0(RW1)                    GET (CALLNO + LENGTH) - 2
         LA    RW3,MASKTAB-(INBUF+2)(RW1)    START OF FIELD IN CNTL TAB
         EX    RW2,VMOVE                     FILL CNTL FIELD WITH BIT
         LA    RW3,BYTETAB-(INBUF+2)(RW1)
         EX    RW2,VMOVE                     FILL IN OFFSET IN BITMASK
         LA    RW3,LENGTH-(INBUF+2)(RW1)
         EX    RW2,VMOVE                     FILL IN LENGTH OF FIELD
         LA    RW3,OFFSET-(INBUF+2)(RW1)
         EX    RW2,VMOVE                     . OFFSET OF FIELD IN REC
         BCT   RC,ON                         DO IT FOR 3 VARIABLE FIELDS
         B     ADJUST                        WHEN FINISHED, GO AHEAD

VMOVE    MVC   1(1,RW3),0(RW3)

ON       DS    0H
         LA    RW1,2(RW2,RW1)                GET LENGTH OF NEXT FIELD
         AH    RW2,0(RW3)                    OFFSET+LENGTH OF PREV FIELD
         LA    RW2,2(RW2)                    CALC OFFSET OF NEXT FIELD
         STC   RW2,OFFSET-(INBUF+2)(RW1)     STORE INTO CONTROL TABLE
         IC    RW3,M(RC)                     PICK UP BIT FOR THIS FIELD
         STC   RW3,MASKTAB-(INBUF+2)(RW1)    STORE IT INTO CONTROL TABLE
         IC    RW3,B(RC)                     PICK UP OFFSET IN BITMASK
         STC   RW3,BYTETAB-(INBUF+2)(RW1)    STORE INTO CONTROL TABLE
         B     FILL                          GO FILL ENTIRE CNTL FIELD

*** FIND FIELDS DIFFERENT FROM PREVIOUS RECORD
***********************************************

DIFFER   DS    0H
         MVI   FORK,X'F0'                    ENABLE BRANCH AT FORK
         XR    RW3,RW3                       CLEAR WORK REGISTER
         ST    RW3,FIXEDEND                  DEFAULT FOR PACKING ROUTINE
         LA    RW3,WAREA+139                 POINT TO END OF FIXED PART
         XR    RWPTR,RWPTR                   CLEAR WORK AREA POINTER
         XR    R1,R1                         CLEAR REGISTER USED IN TRT
         LA    ROUTPTR,WAREA                 POINT TO START OF FIXED PART
         L     RWEND,SL                      PICK UP LENGTH OF RECORD
         LA    RWEND,WAREA+1(RWEND)          INITIALIZE END OF WAREA
         LA    RWPTR,WAREA                   POINT TO START OF STUFF
```

```
*   FIELDS DIFFERENT FROM PREV RECORD HAVE A BYTE ¬= X'00'

FINDDIFF DS    0H
         TRT   0(256,RWPTR),DIFFTAB    FIND DIFFERENT POSITION
         BZ    PCKSTART                NO MORE DIFFERENCES: DONE
         CR    R1,RW3                  IS DIFF IN VARIABLE PART ?
         BL    ADJUST                  .NO, GO AHEAD
         CR    R1,RWEND                IS DIFF BEYOND CURRENT END ?
         BNL   PCKSTART                .YES, DONE
FORK     EQU   *+1                     CONDITION MASK FOR BRANCH INSTR
         B     VSETUP                  MUST SET VARIABLE CONTROL TABLES

*   ADJUST BITMASK

ADJUST   DS    0H
         XR    RW1,RW1
         IC    RW1,BYTETAB-WAREA(R1)   PICK UP OFFSET FOR MASK BYTE
         LA    RW1,BITMASK(RW1)        POINT INTO BITMASK
         LA    RW2,MASKTAB-WAREA(R1)   POINT TO BYTE TO BE ORED
         OC    0(1,RW1),0(RW2)         SWITCH BIT FOR THIS FIELD ON

*   MOVE DIFFERENT FIELD TO FRONT

         XR    RW1,RW1
         IC    RW1,OFFSET-WAREA(R1)    PICK UP OFFSET OF THIS FIELD
         LA    RWPTR,INBUF+2(RW1)      POINT TO THIS FIELD IN INBUF
         IC    RW1,LENGTH-WAREA(R1)    PICK UP (LENGTH-1) FOR FIELD
         EX    RW1,MOVE                MOVE DIFFERENT FIELD TO FRONT

*   BUMP POINTERS

         LA    ROUTPTR,1(RW1,ROUTPTR)  POINT TO NEXT FREE BYTE IN OUTPU
         LA    RWPTR,WAREA-(INBUF+1)(RW1,RWPTR) NEXT FIELD IN WAREA
         CR    RWPTR,RWEND             BEYOND CURRENT END ?
         BL    FINDDIFF                .NO, DO IT AGAIN
         B     PCKSTART                DONE

*** PACK COMPRESSED FIXED PART OF RECORD
****************************************

PCKSTART DS    0H
         L     RWEND,FIXEDEND          POINT TO END OF FIXED PART
         LTR   RWEND,RWEND             WAS IT REALLY SET ?
         BNZ   OK                      .YES, RECORD HAS VARIABLE PART
         LR    RWEND,ROUTPTR           ELSE CURRENT END = FIXED END
OK       DS    0H
         LR    RW2,ROUTPTR             PICK UP END OF COMPRESSED RECORD
         LA    RWPTR,WAREA             POINT TO STUFF TO PACK
         LA    ROUTPTR,PAREA           POINT TO WHERE TO PACK IT
```

```
*   PACK COMPRESSED FIXED PART FROM WAREA INTO OUTPUT BUFFER

PACK     DS    0H
         PACK  0(8,ROUTPTR),0(15,RWPTR)  PACK STUFF
         LA    ROUTPTR,7(ROUTPTR)        BUMP OUTPUT POINTER
         LA    RWPTR,14(RWPTR)           BUMP WORK AREA POINTER
         CR    RWPTR,RWEND               END OF WORK AREA REACHED ?
         BL    PACK                      .NO, DO IT AGAIN

*   COMPUTE LENGTH OF PACKED PART

         SR    RW2,RWEND                 CALC LENGTH OF VARIABLE PART
         LA    RW1,WAREA                 CALCULATE LENGTH OF
         SR    RWEND,RW1                 .COMPRESSED FIXED PART
         LA    RWEND,1(RWEND)            ADD ONE FOR DIVISICN
         SRL   RWEND,1                   CALC LENGTH OF PACKED PART

*   IF THERE IS A VARIABLE PART, MOVE IT TO OUTPUT BUFFER

         LTR   RW2,RW2                   IS THERE A VARIABLE PART ?
         BZ    OUTPUT                    .NO, DONE
         BCTR  RW2,0                     SUBTRACT ONE FOR MOVE
         L     RW1,FIXEDEND              POINT TO VARIABLE PART IN WAREA
         LA    ROUTPTR,PAREA(RWEND)      POINT AFTER PACKED OUTPUT
         EX    RW2,YMOVE                 MOVE VARIABLE PART UNCHANGED
         LA    RWEND,1(RW2,RWEND)        CALC LENGTH OF OUTPUT RECORD
         B     OUTPUT

YMOVE    MVC   0(1,ROUTPTR),0(RW1)

*** OUTPUT, RETURN
******************

OUTPUT   DS    0H
         LA    RWEND,5(RWEND)            ADD LENGTH OF BITMASK
         STH   RWEND,RECL                STORE INTO LENGTH FIELD
         LA    RWEND,2(RWEND)            ADD LENGTH OF LENGTH FIELD
         FSWRITE FSCB=OUT,BUFFER=OUTBUF,BSIZE=(RWEND)  PUT RECORD
         L     RW1,SL                    PICK UP ORIGINAL LENGTH
         EX    RW1,MOVEOLD               SAVE PROCESSED REC FOR DIFF'CING
         B     GET                       GO PROCESS NEXT RECORD

MOVEOLD  MVC   XOREA(1),INBUF+2

EOF      DS    0H
         FSCLOSE FSCB=LIB                CLOSE HISTORY FILE
         FSCLOSE FSCB=OUT                CLOSE OUTPUT FILE
         L     13,SAVE13                 HOUSEKEEPING
         LM    14,12,12(13)              .
         BR    14                        RETURN TO CONTROL PROGRAM
```

```
*** STORAGE DEFINITION
**********************

XOR       XC    WAREA(1),XOREA
MOVE      MVC   0(1,ROUTPTR),0(RWPTR)
XMOVE     MVC   WAREA+117(1),INBUF+129
SMOVE     MVC   INBUF+119(1),INBUF+129

*   FILE CONTROL BLOCKS

LIB       FSCB  'COMP DATA B',RECFM=V,BUFFER=INBUF
OUT       FSCB  'DIFF DATA B',RECFM=V,BUFFER=OUTBUF

*   BITS AND BYTES FOR FILLING VARIABLE PART OF CONTROL TABS

M         EQU   *-1
          DC    X'8001'
B         EQU   *-1
          DC    X'0403'
H117      DC    H'117'
H128      DC    H'128'
H138      DC    H'138'
SAVE13    DS    F
SL        DS    F
FIXEDEND  DS    F

*   OUTPUT BUFFER

OUTBUF    DS    0H
RECL      DS    H
BITMASK   DC    5X'00'
PAREA     DC    3XL100'00'
          DS    0H

*   INPUT BUFFER

INBUF     DC    4CL100' '

*   WORK AREAS

WAREA     DS    139C,176C
XOREA     DC    139C'0',176C' '

*   PATTERN FOR REMOVAL OF HYPHENS USING TR INSTRUCTION

PATTERN   DC    X'020304050607080A0B0D0E0F101112131415161718191A'
          DC    X'1B1C1D1E1F202122232425262728292A2B2C2D2E2F303233'
          DC    X'353637383939A3B3C3E3F414243444546474849 4A4B4C'
          DC    X'4D4E4F505152535455565758595A5B5C5D5E5F606162636465'
          DC    X'666768696A6B6C6D6E6F707172737477677797A7B7C7E'
          DC    X'7F818283848586878889 8A8B8C8D8E8F90919293949596'
```

```
*   TABLE FOR TRT

DIFFTAB  DC    X'00'                          (255 NONZERO BYTES MUST FOLLOW !

*   CONTROL TABLE CONTAINS BITS FOR CORRESPONDING FIELDS

MASKTAB  DC    X'8080',3X'40',6X'20',5X'10',9X'08',9X'04',3X'02'
         DC    3X'01',3X'80',6X'40',4X'20',6X'10',4X'08',9X'04'
         DC    3X'02',2X'01',10X'80',3X'40',9X'20',3X'10',X'0808'
         DC    X'04',X'0202',6X'01',6X'80',3X'40',3X'20',3X'10'
         DC    3X'08',8X'04'
VMASKTAB DC    176X'02'

*   CONTROL TABLE CONTAINS OFFSET FOR CORRESPONDING FIELDS

OFFSET   DC    X'0000',3X'02',6X'05',5X'0B',9X'10',9X'19',3X'22'
         DC    3X'25',3X'28',6X'2B',4X'31',6X'35',4X'3B',9X'3F'
         DC    3X'48',2X'4B',10X'4D',3X'57',9X'5A',3X'63',X'6666'
         DC    X'68',X'6969',6X'6B',6X'71',3X'77',3X'7A',3X'7D'
         DC    3X'80',8X'83'
VOFFSET  DC    176X'8B'

*   CONTROL TABLE CONTAINS LENGTH OF CORRESPONDING FIELDS

LENGTH   DC    X'0101',3X'02',6X'05',5X'04',9X'08',9X'08',3X'02'
         DC    3X'02',3X'02',6X'05',4X'03',6X'05',4X'03',9X'08'
         DC    3X'02',2X'01',10X'09',3X'02',9X'08',3X'02',X'0101'
         DC    X'00',X'0101',6X'05',6X'05',3X'02',3X'02',3X'02'
         DC    3X'02',8X'07'
VLENGTH  DC    176X'00'

*   CONTROL TABLE CONTAINS BITMASK-BYTE FOR CORRESPONDING FIELDS

BYTETAB  DC    40X'00',37X'01',36X'02',26X'03'
VBYTETAB DC    176X'03'
         END
```

DIFRUNPK

```
DIFRUNPK START

*** HOUSEKEEPING,BASEREG,OPEN FILES
************************************

         STM   14,12,12(13)            HOUSEKEEPING
         BALR  3,0                     SET UP BASE REGISTERS
         USING *,3                     .
         ST    13,SAVE13               HOUSEKEEPING
R1       EQU   1
RW1      EQU   4
RWEND    EQU   5
RW2      EQU   6
RWPTR    EQU   7
ROUTPTR  EQU   8
RW3      EQU   10
RC       EQU   11
         FSOPEN FSCB=LIB               OPEN HISTORY FILE
         FSERASE FSCB=OUT              CREATE FRESH
         FSOPEN FSCB=OUT               .OUTPUT FILE

*** INITIALIZATION AND INPUT
*****************************

         XR    RW3,RW3                 CLEAR WORK REGISTER
GET      DS    0H
         FSREAD FSCB=LIB,ERROR=EOF     GET RECORD; EOFADDR=EOF
         XC    BITMASK(5),BITMASK      CLEAR BITMASK

*   REMOVE HYPHENS

         MVC   WAREA(117),PATTERN      PREPARE FOR TRANSLATION
         TR    WAREA(117),INBUF        TRANSL TILL BEFORE LAST HYPHEN
         MVC   INBUF+2(117),WAREA      SAVE FOR LATER USE
         LH    RW1,INBUF               GET LENGTH OF RECORD
         SH    RW1,H128                CALC (LENGTH OF REST) -1
         EX    RW1,XMOVE               MOVE REST UNCHANGED
         EX    RW1,SMOVE               SAVE ALSO FOR LATER USE

*   CALCULATE DIFFERENCE TO PREVIOUS RECORD

         AH    RW1,H117                GET (ENTIRE LENGTH) -1
         EX    RW1,XOR                 DIFFERENTIATE AGAINST PREV REC
         ST    RW1,SL                  SAVE LENGTH
         B     DIFFER                  GO FIND DIFFERENCE
```

```
*** SET UP VARIABLE PART OF CONTROL TABLES
*****************************************

VSETUP   DS    0H
         ST    ROUTPTR,FIXEDEND              SAVE END OF COMPR FIXED PT
         MVI   FORK,X'00'                    DISABLE BRANCH AT FORK
         LA    RW1,INBUF+141                 POINT TO VARIABLE PART
         LA    RC,3                          SET COUNTER
FILL     DS    0H
         LH    RW2,0(RW1)                    GET LENGTH OF CALLNO FIELD
         LA    RW2,1(RW2)                    INCLUDE (LENGTH FIELD) -1
         STC   RW2,LENGTH-(INBUF+2)(RW1)     STORE INTO CONTROL TABLE
         LH    RW2,0(RW1)                    GET (CALLNO + LENGTH) - 2
         LA    RW3,MASKTAB-(INBUF+2)(RW1)    START OF FIELD IN CNTL TAB
         EX    RW2,VMOVE                     FILL CNTL FIELD WITH BIT
         LA    RW3,BYTETAB-(INBUF+2)(RW1)
         EX    RW2,VMOVE                     FILL IN OFFSET IN BITMASK
         LA    RW3,LENGTH-(INBUF+2)(RW1)
         EX    RW2,VMOVE                     FILL IN LENGTH OF FIELD
         LA    RW3,OFFSET-(INBUF+2)(RW1)
         EX    RW2,VMOVE                     . OFFSET OF FIELD IN REC
         BCT   RC,ON                         DO IT FOR 3 VARIABLE FIELDS
         B     ADJUST                        WHEN FINISHED, GO AHEAD

VMOVE    MVC   1(1,RW3),0(RW3)

ON       DS    0H
         LA    RW1,2(RW2,RW1)                GET LENGTH OF NEXT FIELD
         AH    RW2,0(RW3)                    OFFSET+LENGTH OF PREV FIELD
         LA    RW2,2(RW2)                    CALC OFFSET OF NEXT FIELD
         STC   RW2,OFFSET-(INBUF+2)(RW1)     STORE INTO CONTROL TABLE
         IC    RW3,M(RC)                     PICK UP BIT FOR THIS FIELD
         STC   RW3,MASKTAB-(INBUF+2)(RW1)    STORE IT INTO CONTROL TABLE
         IC    RW3,B(RC)                     PICK UP OFFSET IN BITMASK
         STC   RW3,BYTETAB-(INBUF+2)(RW1)    STORE INTO CONTROL TABLE
         B     FILL                          GO FILL ENTIRE CNTL FIELD

*** FIND FIELDS DIFFERENT FROM PREVIOUS RECORD
***********************************************

DIFFER   DS    0H
         MVI   FORK,X'F0'                    ENABLE BRANCH AT FORK
         XR    RW3,RW3                       CLEAR WORK REGISTER
         ST    RW3,FIXEDEND                  DEFAULT FOR PACKING ROUTINE
         LA    RW3,WAREA+139                 POINT TO END OF FIXED PART
         XR    RWPTR,RWPTR                   CLEAR WORK AREA POINTER
         XR    R1,R1                         CLEAR REGISTER USED IN TRT
         LA    ROUTPTR,WAREA                 POINT TO START OF FIXED PART
         L     RWEND,SL                      PICK UP LENGTH OF RECORD
         LA    RWEND,WAREA+1(RWEND)          INITIALIZE END OF WAREA
         LA    RWPTR,WAREA                   POINT TO START OF STUFF
```

```
*   FIELDS DIFFERENT FROM PREV RECORD HAVE A BYTE ¬= X'00'

FINDDIFF DS    0H
         TRT   0(256,RWPTR),DIFFTAB    FIND DIFFERENT POSITION
         BZ    RUN                     NO MORE DIFFERENCES: DONE
         CR    R1,RW3                  IS DIFF IN VARIABLE PART ?
         BL    ADJUST                  .NO, GO AHEAD
         CR    R1,RWEND                IS DIFF BEYOND CURRENT END ?
         BNL   RUN                     .YES, DONE
FORK     EQU   *+1                     CONDITION MASK FOR BRANCH INSTR
         B     VSETUP                  MUST SET VARIABLE CONTROL TABLES

*   ADJUST BITMASK

ADJUST   DS    0H
         XR    RW1,RW1
         IC    RW1,BYTETAB-WAREA(R1)   PICK UP OFFSET FOR MASK BYTE
         LA    RW1,BITMASK(RW1)        POINT INTO BITMASK
         LA    RW2,MASKTAB-WAREA(R1)   POINT TO BYTE TO BE ORED
         OC    0(1,RW1),0(RW2)         SWITCH BIT FOR THIS FIELD ON

*   MOVE DIFFERENT FIELD TC FRONT

         XR    RW1,RW1
         IC    RW1,OFFSET-WAREA(R1)    PICK UP OFFSET OF THIS FIELD
         LA    RWPTR,INBUF+2(RW1)      POINT TO THIS FIELD IN INBUF
         IC    RW1,LENGTH-WAREA(R1)    PICK UP (LENGTH-1) FOR FIELD
         EX    RW1,MOVE                MOVE DIFFERENT FIELD TO FRONT

*   BUMP POINTERS

         LA    ROUTPTR,1(RW1,ROUTPTR)  POINT TO NEXT FREE BYTE IN OUTPU
         LA    RWPTR,WAREA-(INBUF+1)(RW1,RWPTR) NEXT FIELD IN WAREA
         CR    RWPTR,RWEND             BEYOND CURRENT END ?
         BL    FINDDIFF                .NO, DO IT AGAIN
         B     RUN                     DONE

*** RUNLENGTH ENCODING FOR ZEROS
***    LAST BYTE OF FIXED PART:  (FIXEDEND) - 1 FOR LONG RECS
***                              (ROUTPTR) - 1  ELSE (FIXEDEND = 0 )
*************************************************************************

RUN      DS    0H
         L     RWEND,FIXEDEND          POINT TO END OF FIXED PART
         LTR   RWEND,RWEND             WAS IT REALLY SET ?
         BNZ   OK                      .YES, RECORD HAS VARIABLE PART
         LR    RWEND,ROUTPTR           ELSE CURRENT END = FIXED END
OK       DS    0H
         LA    RWPTR,WAREA             POINT TO START OF STUFF
ZEROFIND DS    0H
         CR    RWPTR,RWEND             END OF WORK AREA REACHED ?
         BNL   PCKSTART                .YES, START PACKING
```

```
*    FIND START OF ZERO STRING

          XR     R1,R1                      CLEAR REGISTER USED IN TRT
          TRT    0(139,RWPTR),ZEROTAB       FIND ZERO
          BZ     PCKSTART                   NO MORE ZEROS: DONE
          CR     R1,RWEND                   IS ZERO BEYOND OUR SCOPE ?
          BNL    PCKSTART                   .YES, DONE

*    FIND END OF ZERO STRING

          LR     RWPTR,R1                   POINT TO ZERO FOUND
          TRT    1(127,RWPTR),NZEROTAB      FIND NEXT NONZERO

*    COMPUTE LENGTH OF ZERO STRING

          LR     RW1,R1                     GET END OF STRING
          SR     RW1,RWPTR                  SUBTRACT START OF STRING
          CH     RW1,MINSIZE                IS STRING TOO SHORT ?
          BH     NOTSHORT                   .NO
          LA     RWPTR,1(R1)                BUMP POINTER TO STUFF
          B      ZEROFIND                   TRY AGAIN
NOTSHORT  DS     0H
          CH     RW1,MAXSIZE                IS STRING TOO LONG ?
          BNH    RUNLENTH                   .NO, GO ENCODE IT
          LH     RW1,MAXSIZE                GET MAX LENGTH ALLOWED
          LA     R1,0(RWPRT,RW1)            ADJUST END OF STRING

*    INSERT RUNLENGTH INTO 2 RIGHT HALF-BYTES

RUNLENTH  DS     0H
          LR     RW2,RW1                    SAVE LENGTH
          STC    RW1,1(RWPTR)               INSERT LENGTH AT START OF STRING
          SRL    RW1,4                      .
          STC    RW1,0(RWPTR)               .
          OI     0(RWPTR),X'0C'             MARK BYTE AS RUNLENGTH INDICATOR

*    MOVE STUFF TO FRONT AND CALC NEW END OF FIXED PART

          LA     RWPTR,2(RWPTR)             GET NEXT FREE BYTE
          LR     RW1,RWEND                  GET CURRENT END OF RECORD
          SR     RW1,R1                     CALC LENGTH OF MOVE
          EX     RW1,MVC
          SR     RWEND,RW2                  CALC NEW END: SUBTRACT LENGTH,
          LA     RWEND,2(RWEND)             .ADD ZERO COUNT
          B      ZEROFIND                   GO AGAIN
MVC       MVC    0(1,RWPTR),0(R1)
```

```
*** PACK COMPRESSED FIXED PART OF RECORD
***    LAST BYTE OF COMPRESSED FIXED PART: (RWEND) - 1
***    START OF VARIABLE PART, IF EX:      (FIXEDEND)   (=0 IF ¬EX)
***    END OF VARIABLE PART:               (ROUTPTR) - 1
*****************************************************************

PCKSTART DS    0H
         LA    RWPTR,WAREA              POINT TO STUFF TO PACK
         LA    RW1,PAREA                POINT TO WHERE TO PACK IT

*   PACK COMPRESSED FIXED PART FROM WAREA INTO OUTPUT BUFFER

PACK     DS    0H
         PACK  0(8,RW1),0(15,RWPTR)     PACK STUFF
         LA    RW1,7(RW1)               BUMP OUTPUT POINTER
         LA    RWPTR,14(RWPTR)          BUMP WORK AREA POINTER
         CR    RWPTR,RWEND              END OF WORK AREA REACHED ?
         BL    PACK                     .NO, DO IT AGAIN

*   COMPUTE LENGTH OF PACKED PART

         LA    RW1,WAREA                CALCULATE LENGTH OF
         SR    RWEND,RW1                .COMPRESSED FIXED PART
         LA    RWEND,1(RWEND)           ADD ONE FOR DIVISION
         SRL   RWEND,1                  CALC LENGTH OF PACKED PART

*   IF THERE IS A VARIABLE PART, MOVE IT TO OUTPUT BUFFER

         L     RW1,FIXEDEND             POINT TO VARIABLE PART
         LTR   RW1,RW1                  IS THERE A VARIABLE PART ?
         BZ    OUTPUT                   .NO, DONE
         SR    ROUTPTR,RW1              CALC LENGTH OF VARIABLE PART
         BCTR  ROUTPTR,0                SUBTRACT ONE FOR MOVE
         LA    RW2,PAREA(RWEND)         POINT AFTER PACKED OUTPUT
         EX    ROUTPTR,YMOVE            MOVE VARIABLE PART UNCHANGED
         LA    RWEND,1(ROUTPTR,RWEND)   CALC LENGTH OF OUTPUT RECORD
         B     OUTPUT

YMOVE    MVC   0(1,RW2),0(RW1)

*** OUTPUT, RETURN
******************

OUTPUT   DS    0H
         LA    RWEND,5(RWEND)           ADD LENGTH OF BITMASK
         STH   RWEND,RECL               STORE INTO LENGTH FIELD
         LA    RWEND,2(RWEND)           ADD LENGTH OF LENGTH FIELD
         FSWRITE FSCB=OUT,BUFFER=OUTBUF,BSIZE=(RWEND) PUT RECORD
         L     RW1,SL                   PICK UP ORIGINAL LENGTH
         EX    RW1,MOVEOLD              SAVE PROCESSED REC FOR DIFF'CING
         B     GET                      GO PROCESS NEXT RECORD

MOVEOLD  MVC   XOREA(1),INBUF+2
```

```
EOF       DS    0H
          FSCLOSE FSCB=LIB                CLOSE HISTORY FILE
          FSCLOSE FSCB=OUT                CLOSE OUTPUT FILE
          L     13,SAVE13                 HOUSEKEEPING
          LM    14,12,12(13)              .
          BR    14                        RETURN TO CONTROL PROGRAM

*** STORAGE DEFINITION
**********************

XOR       XC    WAREA(1),XOREA
MOVE      MVC   0(1,ROUTPTR),0(RWPTR)
XMOVE     MVC   WAREA+117(1),INBUF+129
SMOVE     MVC   INBUF+119(1),INBUF+129

*   FILE CONTROL BLOCKS

LIB       FSCB  'COMP DATA B',RECFM=V,BUFFER=INBUF
OUT       FSCB  'DIFF DATA B',RECFM=V,BUFFER=OUTBUF

*   BITS AND BYTES FOR FILLING VARIABLE PART OF CONTROL TABS

M         EQU   *-1
          DC    X'8001'
B         EQU   *-1
          DC    X'0403'
MINSIZE   DC    H'2'
MAXSIZE   DC    H'63'
H117      DC    H'117'
H128      DC    H'128'
H138      DC    H'138'
SAVE13    DS    F
SL        DS    F
FIXEDEND  DS    F

*   OUTPUT BUFFER

OUTBUF    DS    0H
RECL      DS    H
BITMASK   DC    5X'00'
PAREA     DC    3XL100'00'
          DS    0H

*   INPUT BUFFER

INBUF     DC    4CL100' '

*   WORK AREAS

WAREA     DS    139C,176C
XOREA     DC    139C'0',176C' '
```

```
*   PATTERN FOR REMOVAL OF HYPHENS USING TR INSTRUCTION

PATTERN  DC    X'02030405060708 0A0B0D0E0F10111213141516171819 1A'
         DC    X'1B1C1D1E1F20212223242526272829 2A2B2C2D2E2F303233'
         DC    X'35363738393A3B3C3E3F41424344454647484 94A4B4C'
         DC    X'4D4E4F505152535455565758595A5B5C5D5E5F606162636465'
         DC    X'666768696A6B6C6D6E6F707172737476777 97A7B7C7E'
         DC    X'7F818283848586878889 8A8B8C8D8E8F90919293949596'

*   TABLE FOR TRT

DIFFTAB  DC    X'00'                        (255 NONZERO BYTES MUST FOLLOW !

*   CONTROL TABLE CONTAINS BITS FOR CORRESPONDING FIELDS

MASKTAB  DC    X'8080',3X'40',6X'20',5X'10',9X'08',9X'04',3X'02'
         DC    3X'01',3X'80',6X'40',4X'20',6X'10',4X'08',9X'04'
         DC    3X'02',2X'01',10X'80',3X'40',9X'20',3X'10',X'0808'
         DC    X'04',X'0202',6X'01',6X'80',3X'40',3X'20',3X'10'
         DC    3X'08',8X'04'
VMASKTAB DC    176X'02'

*   CONTROL TABLE CONTAINS OFFSET FOR CORRESPONDING FIELDS

OFFSET   DC    X'0000',3X'02',6X'05',5X'0B',9X'10',9X'19',3X'22'
         DC    3X'25',3X'28',6X'2B',4X'31',6X'35',4X'3B',9X'3F'
         DC    3X'48',2X'4B',10X'4D',3X'57',9X'5A',3X'63',X'6666'
         DC    X'68',X'6969',6X'6B',6X'71',3X'77',3X'7A',3X'7D'
         DC    3X'80',8X'83'
VOFFSET  DC    176X'8B'

*   TRANSLATION TABLE FOR ZERO RUNLENGTH ENCODING

NZEROTAB EQU   *-240
         DC   X'00' (240 PREVIOUS AND 15 FOLLOWING BYTES MUST ¬= X'00')

*   CONTROL TABLE CONTAINS LENGTH OF CORRESPONDING FIELDS

LENGTH   DC    X'0101',3X'02',6X'05',5X'04',9X'08',9X'08',3X'02'
         DC    3X'02',3X'02',6X'05',4X'03',6X'05',4X'03',9X'08'
         DC    3X'02',2X'01',10X'09',3X'02',9X'08',3X'02',X'0101'
         DC    X'00',X'0101',6X'05',6X'05',3X'02',3X'02',3X'02'
         DC    3X'02',8X'07'
VLENGTH  DC    176X'00'

*   TRANSLATION TABLE FOR ZERO RUNLENGTH ENCODING

ZEROTAB  DC    240X'00',X'FF'  (15 FOLLOWING BYTES MUST = X'00')

*   CONTROL TABLE CONTAINS BITMASK-BYTE FOR CORRESPONDING FIELDS

BYTETAB  DC    40X'00',37X'01',36X'02',26X'03'
VBYTETAB DC    176X'03'
         END
```