Sorting Nonredundant Files—Techniques Used in the FACT Compiler*

John B. Glore
Minneapolis-Honeywell Regulator Company,† Wellesley Hills, Mass.

Some typical file structures, including some called "nonredundant," are examined, and the methods used in FACT to sort such files are discussed.

The nonredundant file structure discussed in this paper was designed for the FACT compiler jointly by Computer Sciences Corporation and Minneapolis-Honeywell. Techniques were also developed to sort such files. This work was revised and extended to an integrated and workable sorting system by several members of the Honeywell staff, including the writer.

1. Files and File Structures

Let us consider certain concepts associated with conventional computer files. Such files are collections of *items*. Each item is a well-defined collection of *fields*; an item is the smallest unit normally manipulated by a program's input-output system. We shall regard a field as an elementary unit of information. Thus, a fundamental relationship between a file and its items, and between an item and its fields is that of class inclusion. For this reason a file may be called an hierarchical information structure. Such structures contain information, explicitly or implicitly, at several *levels*. For a *simple file* (e.g. a file containing but one type of item) there are three information levels: (1) at the highest (file) level there is usually label data plus file boundary marks such as beginning-of-file and end-of-file indicators; (2) at the item level there may be item delimiters (if item size is not implicit in the programs that read or write the file); (3) at the lowest level are each item's fields.

While recognizing the hierarchical properties of the simplest files we shall restrict here the meaning of *hierarchical*. It will be applied to a different kind of file—one containing more than one type of item and in which items of one kind carry information common to one or more items of other kinds.

Label-checking, end-of-file checking and other operations on the file itself are customarily handled at a different level of logic than item processing. Also, manipulating an item's fields is quite different from obtaining and identifying the items themselves. For example, sorting is concerned with rearranging a file's items, never with changing their internal arrangement—this is an editing function. For these reasons we exclude the file and the field when counting levels; by this criterion a simple file has but one level.[1]

1.1 *Simple Files.* A simple file named F1 of items each named ITEM and comprising fields called AF, BF, CF and DF, would be described to FACT as follows:

```
F1
  *ITEM
    AF
    BF
    CF
    DF
```

The asterisk (*) preceding the item name is used to distinguish items from fields. Other information about fields, such as length and mode, required by FACT, is omitted here for clarity. To be consistent with FACT terminology "group" will be used hereafter in the same sense as "item", except that "group" will sometimes also mean the file itself. Because the file has but one instance, no asterisk precedes its name.[2] Similarly, each field may have but one instance in each group.[3] The asterisk indicates a group can occur an

* Presented at an ACM Sort Symposium November 29, 30, 1962.
† EDP Division.

[1] An item may itself have a complex internal hierarchical information structure. For example, a COBOL record may be described as containing nested sets of information at several levels. (A COBOL record is equivalent to an item here.) These subsets are called *primary groups*. The items themselves are called *secondary groups*. Primary groups must ultimately contain fields. For instance, a $6 \times 10 \times 4$ array contained in an item would be handled by FACT as a 3-level nested structure of primary groups, each ultimate group containing a field. Each of the 240 elements would be assessed by appropriate subscripting of the group names.

Because the types of problems for which FACT was designed involve little or no sorting of arrays and because FACT handles primary and secondary groups quite differently, the FACT sorts were designed to ignore an item's internal groupings. Logically, however, the techniques discussed in this paper are applicable to both the external and internal rearrangement of items.

[2] When a program needs different versions of a file, as an updating routine would read an input master file and write an output master file, the two versions must be described to FACT by two different file names.

[3] In FACT, fields or sets of fields that occur with fixed frequency (>1) within a type of item must be described as components of primary groups; fields or sets of fields that can occur with variable frequency must be described as components of one or more types of secondary groups (items) subordinate to the original group type. These rules must be applied repeatedly to the resulting groupings until all multiple occurrences disappear.

indefinite number of times. Identation indicates class inclusion. Thus the diagram represents any file named F1, containing groups of one type called ITEM, each of which has the four fields AF, BF, CF and DF. The order of these groups is not relevant to the file's structure.

FACT constructs a definite item format from the information normally supplied in the diagram. Thus, every group in a simple file has the same internal structure. FACT requires separate descriptions of groups having different structures, and assigns them different type-codes.[4] To FACT, two groups have the same structure if and only if: (1) both contain the same number of fields, and (2) every field of one group has exactly one counterpart in the other such that both members of each pair have the same intention (meaning), and (3) the location of every field in one group is the same as the location of its counterpart in the other.[5]

Other criteria are possible, of course, but these are convenient and fairly standard, and they serve to illustrate basic principles as well as any others.

1.2. *Complex Single-Level Files.* Any difference in group structure requires definition of different *group-types.* For example, suppose a file named F2 contains some groups with the above structure, others containing fields called AF, EF, and FF, and still others containing the fields AF, EF, GF and HF. Three group-types must be described, as follows:

```
F2
  *ITEM1
     AF
     BF
     CF
     DF
  *ITEM2
     AF
     EF
     FF
  *ITEM3
     AF
     EF
     GF
     HF
```

ITEM1, ITEM2, and ITEM3 are assumed to be the three group-type names. Each group-type's field names must be stated in full, even though AF is common to all the group-types and EF is common to the last two. The equal indentation of the three group-type names specifies to FACT that there is no hierarchical relationship among any groups in the file. We may say that all have a common *father,* the file, and hence all may be called *brothers.* The different group-types are *brother group-types.*

Unlike F1, F2 cannot be called simple, as it contains three group-types. It remains a *single-level* file, however, as all its groups (except the file itself) are brothers. The file description implies no restriction on the order or relative frequency of brother groups. In particular, it does *not* specify that all groups of ITEM1 precede all groups of ITEM2, nor that all groups of ITEM2 precede all groups of ITEM3. Because there is no implied connection among brother groups, all groups of a single-level file may be rearranged freely without distorting the file's information content. Logically, a single-level file may contain any definite number of group-types, provided the object program can distinguish them unambiguously.

1.3. *Simple Hierarchical Files.* A different kind of file structure exists when information in groups of different type has an implied connection. Suppose that each group of F1, above, were broken into two parts, the first containing its AF and BF fields, and the second its CF and DF fields. Let the two group types be called PART1 and PART2 respectively. Procedures that used F1 would now require groups of both these types. Further, they would need a specific PART2 for each PART1. We would normally regard each pair as an inclusive set containing a PART1 and a PART2 as subsets on the same level ("peers"). Alternatively, we could consider the PART1 as the more inclusive set containing the PART2 as a subset. Equally well, the PART2 could be considered more inclusive, and the PART1 its subset. The last two structures would be described to FACT as follows:

```
F3              F4
  *PART1          *PART2
     AF              CF
     BF              DF
    *PART2          *PART1
       CF              AF
       DF              BF
```

As before, indentation represents inclusiveness. Both structures are deemed *hierarchical* because they depict different levels of group. In such a file each *nonterminating group* (i.e. PART1 of F3) is associated with one *or more* lowest level groups (i.e. PART2 of F3). These are called *terminating groups.* All groups in a single-level file are terminating; F2 contains three types of terminating groups.

The fields of each nonterminating group are implicitly associated with each and every terminating group belonging to the containing nonterminating group. Thus, every terminating group of an hierarchical file, together with its father, contains information equivalent to a single-level file item that states explicitly both the nonterminating and the terminating group fields. These fields together comprise

[4] A unique type code is created for each group-type of the file. The appropriate code is inserted into a small (8-bit) common field of every group issued. When the object program obtains a group from an input file its type code is used to enter tables containing information pertinent to its hierarchical position and internal structure.

[5] These definitions presuppose that a field can always be located without ambiguity: (a) at a fixed distance in bits from the group base (first bit), or (b) at a possibly variable distance from the group base, by counting a fixed number of fields from the group base, or (c) by a combination of methods (a) and (b). Methods (b) and (c) in turn require that the object program be able to determine a variable field's length with reference to a character count or by recognizing a delimiter. The criteria are broad enough so that two groups with different variable field lengths are considered to have the same structure if they satisfy the other definitions stated in the text.

a terminating group's single level item equivalent (abbreviated *SLIE*). There is one SLIE for each terminating group in the file.

For the situation supposed above, neither F3 nor F4 appears to have an advantage over the "peer pair" structure and all three seem inferior to the simple file structure of F1 because they would entail extra housekeeping. If, however, F1 were sorted into groupings of items with identical AF-BF value, the following procedure would transform F1 to F3: (1) for each grouping, issue a single PART1 containing the grouping's AF-BF; (2) for every group in this grouping, issue a PART 2 belonging to this PART1; repeat step (2) for every grouping.

The resulting file, F3, would contain one PART1 for every different AF-BF (i.e. AF *and* BF) value plus one PART2 for every F1 item. Neglecting possible waste in packing fields into words, the new file would require less space than the old—exactly the amount needed in F1 to store repeated instances of identical AF-BF values. In this sense an hierarchical file may be called *nonredundant*. Similarly, if F1 were sorted into groupings of identical CF-DF value, a nonredundant file of structure F4 could be constructed. Whether F3 or F4 would be more compact would depend entirely on the distribution of data in the original file—that is, on whether there were more different AF-BF values or more different CF-DF values.

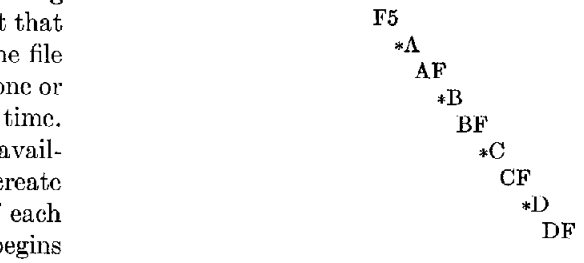
So far we have assumed random access to all groups of a file and have not stated how one can discern the connection between a specific nonterminating group and its terminating groups. As keying would eliminate most or all space saving, a positional relationship is probably implied even for random stores. In serial-access storage media such as FACT's magnetic tape files, each nonterminating group is issued immediately before the string comprising its included terminating groups. For this reason nonterminating groups are commonly called *headers* and terminating groups *trailers*. Their relative position is the only fact that establishes which header is any trailer's father. As the file is read forward, the first header is stored in core. Its one or more trailers succeed it. These are read one at a time. Because its header is stored, each trailer's SLIE is available when the trailer is read. These may be used to create the equivalent single-level file, if desired. The end of each string except the last is marked by the header that begins the next string. This header is stored, etc., until the end-of-file signals the last string's end.

Each header must precede its trailers for another reason: each string's length is variable, depending as it does on change in header field value. A string may be quite long—in extreme cases comprising all groups in the file. Thus, only if it occurs before its trailers can we guarantee the proper header's availability to all of its trailers. For this reason the file must be read forward, unless special provision is made.[6]

Creating a serial hierarchical file may yield a somewhat different result than the "grouping" operation discussed above. A header, and then a trailer, are issued from the first simple file item. The header information is also stored. The next simple file item is obtained. If (any of) its header fields differ from those stored another header and a trailer are issued and the new header information is stored. Otherwise only a trailer is issued; the previously stored header information is left unchanged.

This procedure creates an hierarchical file consisting of a header followed by one or more trailers, followed by another header, etc., until the original file ends, as before. But the number of headers equals the number of *changes* (+1) in header field value rather than the number of different header field values.[7] The former will equal the later only if the simple file was in order by header field values; otherwise the number of changes will usually exceed greatly the number of different values.[7] Such results could be expected if F3 were created from F1 sorted by CF-DF or if F4 were created from F1 sorted by AF-BF. So hierarchical files need not be nonredundant; they may be just as redundant as single-level files in extreme cases. As a rule of thumb, the more file structure and file order correspond, the more nonredundant an hierarchical file can be. This principle has important implications for file designers. It also affects the design of FACT sort programs, in that the sorts must allow for usually unpredictable changes in a file's volume as it is rearranged.

1.4. *Hierarchical Files of More than Two Levels.* Hierarchical files of more levels than two can be created reversibly from a simple file by rather simple extension of the preceding rules. In general, the number of levels can be extended up to the number of fields in the equivalent single-level file group. For example, consider the file F5 diagrammed below:



F5 contains four types of group at four levels. Their assumed group names are A, B, C and D, respectively. Each F1 item field has been assigned to one of the four group-

[6] An hierarchical file can be made *reversible* (readable either forward or backward) if an *ender* is issued at the end of every string of trailers. Each ender must contain the same data as the string's header.

Programs that read reversible files forward must store headers and ignore enders. Programs that read reversible files backward must store enders and ignore headers. The presence of enders doubles the amount of tape storage devoted to headers. Consequently, a reversible file's value should be weighed carefully against its cost. Sometimes an equivalent single-level file would require less tape space.

[7] To illustrate: the sequence 1,1,2,1,2,1, comprises four changes in value but only two different values.

types in F5. In F5, each A "contains" one or more B's; each B "contains" one or more C's; each C "contains" one or more D's, where "contains" means "represents a logical set inclusive of." Therefore each D participates in the CF of the C containing it, in the BF of the B containing it, and in the AF of the A containing it. These three fields, plus the D's DF, comprise its SLIE—all are essential.

Like F3, F5 has one type of terminating group or trailer, i.e. D. But now each trailer has three headers, a C, a B and an A. Let us call any group that "contains" another its *ancestor*. A group has exactly one ancestor at every level above it. A group's immediate ancestor is its father. In F5, every D's father is a specific C; every D has three ancestors (not counting the file itself) while every B has one ancestor.

The groups "contained in" a specific group are its *descendents*. Let a group's immediate descendents be called its *sons*. A trailer may have no son. Every header must have at least one son, and may have many sons. Sons of the same father are termed *brothers*. All brothers are at the same level in the file. (For file F2, above, and for certain files to be introduced later, some brothers may be of different group-type.)

The following rules are invoked to create an hierarchical file and are used to determine a group's ancestors as the file is read: (1) every father must precede (i.e. be issued before) its sons, and (2) brothers can occur in any order within the string begun by their father, and (3) a string of father and sons is ended when any ancestor of a son is issued, or by the end of file.

A procedure to create F5 from F1 will illustrate the process. Assume that the first F1 item has been obtained. Then:

(1) Issue an A, a B, a C, and then a D containing the current ITEM'S AF, BF, CF, and DF fields respectively; store the current AF, BF, and CF; to to (2).
(2) Obtain the next ITEM; go to (3).
(3) If the current ITEM's AF differs from the stored AF, go to (1); otherwise go to (4).
(4) If the current ITEM's BF differs from the stored BF, issue a B, a C, and then a D, containing respectively the current item's BF, CF, and DF; preserve the stored AF; store the new BF and CF, erasing the previous stored values; go to (2). If the current BF and stored BF are equal, however, go to (5).
(5) If the current ITEM's CF differs from the stored CF, issue a C and then a D containing, respectively, the current item's CF and DF fields; preserve the stored AF and BF; store the new CF, erasing its previous value; go to (2). If the current and stored CFs are equal, however, go to (6).
(6) Issue a D containing the current item's DF; to to (2).

Continue until the end of F1 is detected.

As a result of this procedure an initial A, B, C and D are issued. Thereafter a new A is issued for each change[7] in AF, a new B for each change in BF *or* AF, a new C for each change in CF *or* BF *or* AF, and a new D for every ITEM. Extension of the procedure to 3-level, 5-level and higher-level files is obvious.

When F5 is read[8] the fields of each A are put into an A storage, erasing its previous contents. The fields of each B and of each C are similarly stored. Thus, when each terminating (D) group is obtained a complete and correct SLIE is available. If desired, each SLIE can be issued, recreating F1.

Figure 1 illustrates these transformations for a hypothetical data set. Column 1 shows a series of F5 groups; column 2 depicts their SLIEs. Colume 3 may be thought of as a series of F1 items; column 4 represents the F5 file produced from them.

Just as F3 and F4 are different hierarchical arrangements of the group-types PART1 and PART2, so alternate hierarchical arrangements of the group-types A, B, C, and D are possible. For example, F6 below could be created from F1.

```
F6
  *C
    CF
      *A
        AF
          *D
            DF
              *B
                BF
```

Many three-level arrangements are also possible, if two of the group-types are collapsed into one. The best design is usually the most nonredundant for the file's usual order.

1.5. *Complex Hierarchical Files.* Complex hierarchical files can be created from complex single-level files according to the rules stated above. For example, F7, below can be created from F2, above.

```
F7
  *A
    AF
      *B
        BF
          *C
            CF
              *D
                DF
      *E
        EF
          *F
            FF
          *G
            GF
              *H
                HF
```

In F7 there is one type of group, A, at the highest level. An A may contain either B's or E's or both as sons—in

[8] Again, the file must be read forward unless reversible. Hierarchical files of more than two levels must, if reversible, have an ender corresponding to every header. These must be distinguishable by type. Each ender must contain the same information as the corresponding header. They are issued in order opposite to the corresponding headers. For instance, for F5, if a new C is to be issued, an ender (C′) containing the last CF must be issued just before the new C. If a new B is to be issued, a C′ and a B′ must first be issued, etc. As before, programs that read reversible files forward ignore all enders while programs reading them backward must ignore all headers.

any order and with any relative frequency—B and E are brother group-types, as the indentation shows. Similarly, F and G are brother group-types—an E may contain F's or G's or both as sons. C is *not* a brother group-type of F and G, although they are at the same level, since they are sons of different fathers. It would be illegal to move a C into the immediate neighborhood of an F or a G. F7 has three types of terminating group: D, F and H. Note that D and H are at the same level but that F is at a higher level. This is legitimate in FACT files. Each terminating group-type and its ancestor group-types are said to comprise an *hierarchy*. F7 contains the following hierarchies: A, B, C, D; A, E, F; and A, E, G, H. Note that A is common to all three hierarchies and that E is common to the last two. The hierarchies of other complex hierarchical files may or may not have such common groups. If, however, a group-type is common to more than one hierarchy, it must be at the same level in every hierarchy containing it. (Levels are counted beginning with the most inclusive.) Thus the structure:

```
F8
  *A
    AF
      *B
        BF
  *C
    CF
      *A
        AF
```

is illegal in a FACT file because A is at different levels in two hierarchies. Also, a group-type may not be the son of different fathers in different hierarchies. Within the limits of these restrictions the group-types of each hierarchy can be arranged independently of one another. Similarly, the number of levels in each can be established independently.

The rules stated in Section 1.4. also govern the relative positions of a complex hierarchical file's groups, since groups of brother group-types are brothers. By application of these rules equivalent complex hierarchical files can be formed from complex single-level files. In each such transformation the hierarchical file must contain a different hierarchy corresponding to each single-level file group-type. For F7 and F2, hierarchy A,B,C,D corresponds to ITEM1; hierarchy A,E,F corresponds to ITEM2, and A,E,G,H corresponds to ITEM3. A different type of SLIE exists for each hierarchy. Figure 5, columns 7 and 8 (first part), illustrates SLIE construction for a hypothetical file of structure F7.

2. FACT Sorting Methods

FACT will generate, from suitable key specifications and a file description, a sorting routine able to rearrange any of the file types discussed above. Considerable flexibility is permitted in key specification. The following rules most affect the sorts' design:

(1) A group may never be separated from its father except by its brothers (as only their relative position identifies a group's father).

F5, a 4-level hierarchical file discussed in the text, consists of the data shown in column 1. For clarity only group name and field *value* are shown. Thus, "A2" means an A for which AF = 2. Column 2 depicts the sort items constructed from F5 for a sort by AF, BF, CF, DF. Each item is equivalent to a SLIE. Column 3 depicts the sort items rearranged by the sort. Column 4 represents the output file, returned to F5 form.

(1) *Input*	(2) *Raw Items*	(3) *Sorted Items*	(4) *Output**
A1		A1B1C1D1	A1
B1		A1B1C1D2	B1
C2		A1B1C2D1	C1
D1	A1B1C2D1	A1B1C2D3	D1
A2		A1B2C2D1	D2
B2		A1B2C2D2	C2
C2		A2B2C2D1	D1
D1	A2B2C2D1	A2B2C2D2	D3
D2	A2B2C2D2		B2
A1			C2
B1			D1
C1			D2
D1	A1B1C1D1		A2
D2	A1B1C1D2		B2
C2			C2
D3	A1B1C2D3		D1
B2			D2
C2			
D1	A1B2C2D1		
D2	A1B2C2D2		

* A dash between entries indicates that one or more redundant groups are eliminated between two groups.

Fig. 1

(2) Brothers, regardless of group-type, may be rearranged freely within the string begun by their father.

(3) Unless fields equivalent to type codes are specified as keys, the order of brothers is a function of their key values only; automatic segregation of brothers by type is not done.

(4) The relative positions of brothers having equal key values is indeterminate; if important, enough keys must be specified to break all ties.

(5) A sort's input file and its output file must have the same structure.[9]

The sorting methods applied to each type of file are now discussed.

2.1. *Sorting Simple Files.* No particular problems occur here, as each group is a complete SLIE, and all groups are to be rearranged freely within the file as a whole on the basis of the key or keys specified. Any one or more of the group-type's fields may be specified as a key of any level of significance.

A preliminary editing phase called the *pre-edit* creates a file of *sort items* from the input file's groups. The label and data block formats differ somewhat. A sort item is created from each input file group; their formats are similar, ex-

[9] The FACT sorts provide optional "last pass own coding" called *postsort procedures*. Via such a procedure the user can access each SLIE of the output file before it is filed. If he wishes the user can inhibit writing of the sort output file and call for construction of a differently structured file provided this is derivable from the SLIE.

cept for *key arrangement.* This procedure packs the group's key fields, from major to minor, into consecutive-item bit positions starting with the group base, and puts the data originally located in this "key area" into the "holes" from which the keys were drawn. If necessary it may also translate key values to assure proper sorting sequence.[10] Because the key and nonkey positions are exchanged the item's overall size remains unchanged.[11] Key arrangement allows the sorting phases to use standardized comparison coding; it thus simplifies greatly their generation. Also, if key fields are small, packing may permit decisions to be made in fewer comparisons than would be possible with unpacked keys.

The sort item file comprises one or more reels, depending on total input volume; the input may be drawn from a file of one or more reels. For a simple file structure the input and sort item files usually have about the same physical volume, as each sort item is about the same size as an input group. Differences in tape block size may affect physical volume, however.

Each reel of the sort item file is sorted separately by a slightly modified Honeywell 800 ARGUS sort routine. This consists of two phases: a presort that builds strings of ordered sort items on 2-5 work tapes, and a cascade mergesort that progressively reduces the number of strings.

If there is more than one reel of sorted items a collate phase is entered. This merges the separately sorted reels of sort items into a single file. If there is but one reel of sorted items the collate phase is bypassed.

Finally, a *postedit* phase returns each sort item to group form. The main task here is to reverse the key arrangement done in the pre-edit; a FACT file results.

2.2. *Sorting Complex Single-Level Files.* The group-types of a complex file have different internal structures; also, FACT lets the user specify a different sequence of keys for each group-type. Consequently the pre-edit must construct a different type of sort item for each type of group. Different pre-edit and postedit key arrangement routines are in general required for each item type. Otherwise the sort phases are the same as for simple files.

No attempt is made to segregate items by type during sorting. Thus presort mergesort and collate phases arrange the items of all types strictly according to the relative value of their packed keys. In general the user must insure the congruence of the packed keys of the different item types. He must also insure that the comparisons make sense.[12]

2.3. *Sorting Single-Hierarchy Files.* File F5 is typical of such structures. Since a sort may not separate a group from its ancestors, keys may not be specified independently for the group-types of an hierarchy. Instead, all keys specified comprise a single sequence that applies to the hierarchy as a whole. 0, 1, or more fields of each group-type may be specified as keys. The major key may be drawn freely from any group-type in the hierarchy. Then the next most significant key specified (if any) may belong to the same group-type or to a different group-type, as the user wishes[11], etc. Hence a file may be sorted into an order different from its most nonredundant order if desired. The sorting methods vary somewhat depending on how keys were selected.

2.3.1. *When Keys are Drawn from Every Group-Type.* This is the simplest case. If every group-type in the hierarchy contains at least one key field, the pre-edit constructs a sort item from each SLIE. In this way it guarantees that the sorting phases will not separate father and son, because every SLIE includes a terminating group and a copy of all its ancestors. The fields of the SLIE's most inclusive group (i.e. A of F5) are put at the top of the item. It's son's fields (i.e. BF of B of F5) are put just below them, etc., so that the SLIE'S terminating group fields are put at the sort item's end. The sort item as a whole then undergoes key arrangement as discussed above.

Presort, mergesort, and (if necessary) collate phases take place as described above. It should be noted that sort item construction usually swells the file's physical volume by an amount often hard to predict.

In the postedit each sort item undergoes key rearrangement, after which it effectively comprises a SLIE. Using the rules stated in Section 1.4. an hierarchical FACT file is created from these SLIEs. Its structure is the same as the input file's.[9]

The relative physical volumes of input and output file depend on which is more redundant. This may be hard to predict. Thus, provision is made independently for multiple input reels and multiple output reels. The output file is almost always more compact than the file of sort items. Figure 1 traces item construction, item rearrangement, and item decomposition for a possible F5, if AF, BF, CF, and DF are stated as keys in that order.

2.3.2. *When Keys are Missing From Intermediate Levels.* Omitting all fields of one or more intermediate group-types from the list of keys specified affects the procedure described in Section 2.3.1 only slightly, provided that at least one key field is drawn from the terminating group-type and at least one key field is drawn from the most inclusive group-type. As before, an item is con-

[10] For example, in Honeywell 800 FACT, signed decimal fields are packed as a leading sign and a string of 4-bit digits. So that such fields will always sort in true algebraic sequence, the key arrangement generates the bit complement of each negative signed decimal field.

[11] Two minor restrictions follow from the one-for-one exchange procedure: (a) no field may be used as a key more than once per item, and (b) no variable-length field may be used as a key.

[12] For example, if a two-digit field is specified as the major key for one group-type, and a three-digit field for another group-type, the two-digit field will be compared against the high-order two digits of the three-digit field, and any lower-order keys specified for the group-types will be mismatched when compared. Again, a field called APPLES of one group may be compared as requested against a field called ORANGES of another; their comparison is mechanically correct, but the result is probably meaningless.

When F5 is sorted by AF, DF only, raw sort items are constructed from the same data as those of Figure 1. Thus, columns 1 and 2 are the same for these figures. Column 3 and column 4 show the different arrangement of items and consequently different output file. Note that it contains more groups than the first sort's output.

(1) *Input*	(2) *Raw Items*	(3)** *Sorted Items*	(4) *Output**
A1		A1B1C1D1	A1
B1		A1B1C2D1	B1
C2		A1B2C2D1	C1
D1	A1B1C2D1	A1B1C1D2	D1
A2		A1B2C2D2	C2
B2		A1B1C2D3	D1
C2		A2B2C2D1	B2
D1	A2B2C2D1	A2B2C2D2	C2
D2	A2B2C2D2		D1
A1			B1
B1			C1
C1			D2
D1	A1B1C1D1		B2
D2	A1B1C1D2		C2
C2			D2
D3	A1B1C2D3		B1
B2			C2
C2			D3
D1	A1B2C2D1		A2
D2	A1B2C2D2		B2
			C2
			D1
			D2

* A dash between entries indicates that one or more redundant groups are eliminated between two groups.

** Ties among items have been resolved arbitrarily.

Fig. 2

When F5 is sorted by AF, BF, CF, each sort item contains only A, B and C groups; the D's are attached as trailers to these items. The original order of a sort item's trailers is preserved through the sort. Column 1 shows the same data distribution as the previous two figures. Column 2 depicts the items. As would be expected, there are fewer items than in the previous figures. Column 3 depicts these rearranged by the sort. Column 4 represents the final output, of form F5.

(1) *Input*	(2) *Raw Items*	(3)** *Sorted Items*	(4) *Output**
A1		A1B1C1	A1
B1		D1	B1
C2	A1B1C2	D2	C1
D1	D1	A1B1C2	D1
A2		D3	D2
B2		A1B1C2	C2
C2	A2B2C2	D1	D3
D1	D1	A1B2C2	D1
D2	D2	D1	B2
A1		D2	C2
B1		A2B2C2	D1
C1	A1B1C1	D1	D2
D1	D1	D2	A2
D2	D2		B2
C2	A1B1C2		C2
D3	D3		D1
B2			D2
C2	A1B2C2		
D1	D1		
D2	D2		

* A dash between entries indicates that one or more redundant groups are eliminated between two groups.

** Ties among items have been resolved arbitrarily.

Fig. 3

structed from each SLIE. The key arrangement differs, but only because a different series of key fields was specified. Figure 2 traces the pertinent operations if F5 is sorted on AF, DF only.

2.3.3. *When Keys are Missing From the Lowest Level(s).* A fairly radical change in sort-item construction takes place when the lowest level key-containing group type is not terminating. In this case an item is issued, not for each SLIE, but for each lowest level key-containing group. The item contains the fields of this group and its ancestors. To it are attached, *as trailers*, the one or more groups belonging to its lowest level member. If F5 were sorted on the fields of A, B, and C, an item per C would be created. Each such item would have one or more D trailers. See Figure 3. Key arrangement follows the usual rules; the trailers are ignored by this process.

Since any such item may have an indefinitely long string of trailers it must be broken into subsections of convenient length for the sorting phases. After key arrangement the first subsection is prepared as follows: (1) a serial number is inserted after the last word of packed key and before the nonkey portion (if any) of the item; (2) as many trailers as possible of the item are put after its nonkey portion, provided the convenient length is not exceeded; (3) an end of item mark is laid down.

If all trailers fit into the first subsection, the item's construction is finished. Otherwise, one or more subsequent subsections are issued, prepared as follows: (1) the packed key of the first subsection is laid down, followed immediately by the augmented serial number; (2) as many trailers as possible or necessary are laid down; (3) an end of item mark is laid down.

Presort, mergesort, and collate (if needed) phases are executed as before. To these sorting phases each item subsection is a distinct sort item. Since all subsections of the same item have equal key, and since the serial numbers increase monotonically for all subsections of all items, the order of an item's subsections is preserved by the sort. (The serial numbers correctly break ties among an item's subsections.)

The postedit reverses the processes of item construction and creates a FACT file from each item and its trailers. It should be noted that these processes effectively preserve the relative order of each item's trailers.

2.3.4. *When Keys are Missing From the Highest Level(s).* Unless a key is specified for the file's most inclusive group (e.g. A of F5) all groups above the highest level key-containing group are treated very specially. Such groups are

When F5 is sorted by DF, CF, BF, all A's are treated as higher-level groups and divide the file into batches. The order of batches is preserved by the sort. Rearrangement is done only within each batch. Each item is equivalent to a SLIE, except that higher-level groups are excluded. Column 1 shows the same input file as the previous figures. Column 2 depicts the items before sorting, and column 3 the items after sorting. The higher-level groups are listed here for clarity; during actual sorting they are temporarily removed from the file. Column 4 shows the final output, in F5 form.

(1) *Input*	(2) *Raw Items*	(3) *Sorted Items*	(4) *Output**
A1	A1	A1	A1
B1		B1C2D1	B1
C2			C2
D1	B1C2D1		D1
A2	A2	A2	A2
B2		B2C2D1	B2
C2		B2C2D2	C2
D1	B2C2D1		D1
D2	B2C2D2		D2
A1	A1	A1	A1
B1		B1C1D1	B1
C1		B2C2D1	C1
D1	B1C1D1	B1C1D2	D1
D2	B1C1D2	B2C2D2	B2
C2		B1C2D3	C2
D3	B1C2D3		D1
B2			B1
C2			C1
D1	B2C2D1		D2
D2	B2C2D2		B2
			C2
			D2
			B1
			C2
			D3

* A dash between entries indicates that one or more redundant groups are eliminated between two groups.

Fig. 4

called *higher-level groups*. They are never included in sort items. Instead, they divide the file into *batches*. The contents of each batch is arranged separately and the order of batches is preserved. When a file contains two or more levels of higher level groups two or more such groups can occur in a sequence between groups from which keys will be drawn. For example, if A and B held no keys but C and D did, each batch would begin with a sequence A, B or else a single group, B. The relative position of the groups in a sequence must not be changed by the sort. These problems are handled as follows:

(1) The pre-edit assigns each batch a monotonically increasing batch number.

(2) Each higher-level group is marked with the current batch number; the higher level groups are copied to a work tape in the order read to get them out of the sort's way; this tape is called the *higher level group tape.*

(3) Sort items are constructed from the other groups in accordance with one of the methods discussed above, except that no item contains higher-level group information; also, the appropriate batch number becomes the high order portion of each item's key.

(4) At the pre-edit's end a higher-level group file and a sort item file have been created. Presort, mergesort, and collate phases handle the latter as usual. Because the major key of all items was their batch number, batch order is preserved.

(5) During the postedit the higher-level group file and the rearranged file of sort items are collated and the items rearranged. The expected output file results.

Figure 4 illustrates these processes for F5.

2.4. *Sorting Complex Hierarchical Files.* No new principles are required to sort complex hierarchical files. With a few extensions and restrictions those previously stated apply.

First, a separate type of sort item is normally constructed for each file hierarchy.[13] For file F7, separate types of item are made from the three hierarchies A,B,C,D; A,E, F; and A,E,G,H.

Second, a separate key list must be specified for each type of item. Each key list must be drawn from fields of the group-types in the hierarchy from which the corresponding items will be made. If a group-type is common to more than one hierarchy it may be a key source for all items, or for none, as the user wishes. If a common source, the same field(s) may or may not be drawn from it for the different items. If a common field is stated, it may or may not have the same relative significance in the key lists of the different items. These possibilities are restricted somewhat, below.

The pre-edit code requires that a group-type be classified consistently for all item types as either (1) a higher-level group, (2) a sort item member or (3) a sort item trailer. Thus, if A contains a key for one item type it must contain a key for all. Similarly, E must be a sort item member for both of the last two item types or for neither.

Again, the highest-level key-containing groups for all item types must be at the same level in the file. In F7, if no A, B, or E keys were stated, but a key of C were stated, keys of F and G would also need to be stated. This rule is unnecessarily restrictive in certain cases, but relatively harmless; its use simplifies analysis.

The pre-edit for a complex hierarchical file sort depends as usual on obtaining a SLIE per item issued. (As above, if certain ancestors are higher-level groups they are omitted from the items constructed; similarly, items having trailers omit these from item construction.) As the pre-edit obtains each group from the input file it stores the group. The group's type-code is used to enter a table that tells whether an item is to be issued (or continued, if the group is an item

[13] But in applying this rule, branching below the level of the lowest-level key containing group along any path is ignored. Consider these examples for different sorts of file F7: (a) if AF is the sole key, only one type of item is made; (b) if AF, BF, and EF are the keys, two types are needed, corresponding respectively to the first hierarchy and to the last two hierarchies as a group; (c) if all fields are keys, three types are needed.

F7 has the structure:

```
F7
  *A
    AF
      *B
        BF
          *C
            CF
              *D
                DF
      *E
        EF
          *F
            FF
          *G
            GF
              *H
                HF
```

and the initial distribution of File #1 shown in column 1. This file undergoes four successive sorts, as follows:

(a) Sort #1 reads File #1 and creates File #2 (column 3). AF is the sole key. A single type of item, consisting of A groups, is created. All other group-types form trailers. Column 2 shows item formation and rearrangement.

(b) Sort #2 reads File #2 and creates File #3 (column 5). BF, AF; and EF, AF are the keys of the first and second item types formed. Type 1 items contain A and B fields; they have C and D trailers. Type 2 items contain A and E fields; they have F, G, and H trailers. Column 4 shows the items.

(c) Sort #3 reads File #3 and creates File #4 (column 7). BF, CF; EF, FF; and EF, GF, HF are the respective key lists of the three item types former. Type 1 items contain B and C groups; they have D trailers. Type 2 items contain E and F groups; they have no trailers. Type 3 items contain E, G, and H groups; they have no trailers. The A groups are higher-level groups and divide

(1) File No. 1	(2) Sort No. 1 Raw Items	Sorted Items	(3)* File No. 2	(4) Sort No. 2 Raw Items	Sorted Items	(5)* File No. 3	(6) Sort No. 3 H L G Raw Items	H L G Sorted Items	(7)* File No. 4	(8) Sort No. 4 Raw Items	Sorted Items	(9)* File No. 5
A2	A2	A1	A1		A2B1	A2	A2	A2	A2		A1E4G1H1	A1
B2	B2	E4	E4	A1E4	C1	B1		E1G1H1	E1			E4
C2	C2	G1	G1	G1	D2	C1	B1C1		G1		A1E4G1H2	G1
D2	D2	H2	H2	H2		D2	D2	E1G1H2	H1	A2E1G1H1		H1
D1	D1	H1	H1	H1	A2E1	E1			H2	A2E1G1H2	A2B1C1D1	H2
D3	D3	A2	A2		F2	F2	E1F2	B1C1	B1			A2
C1	C1	B2	B2	A2B2	F1	F1	E1F1	D2	C1		A2E1G1H1	B1
D1	D1	C2	C2	C2	G3	G3			D2	A2B1C1D2		C1
D4	D4	D2	D2	D2	H1	H1	E1G3H1	B1C1	D1	A2B1C1D1	A2B1C1D2	D1
B1	B1	D1	D1	D1	H2	H2	E1G3H2	D1	D4	A2B1C1D4		E1
C1	C1	D3	D3	D3	G1	G1		D4	D2	A2B1C1D2	A2B1C1D2	G1
D2	D2	C1	C1	C1	H2	H2	E1G1H2	D2	D3	A2B1C1D3		H1
E1	E1	D1	D1	D1	H1	H1	E1G1H1	D3	E1		A2E1G3H1	B1
F2	F2	D4	D4	D4	F3	F3	E1F3		F1	A2E1F1		C1
F1	F1	B1	B1	A2B1		B1		E1F1	F2	A2E1F2	A2B1C1D3	D2
G3	G3	C1	C1	C1	A2B1	C1	B1C1		G3			D2
H1	H1	D2	D2	D2	C1	D1	D1	E1F2	H1	A2E1G3H1	A2B1C1D4	E1
H2	H2	E1	E1	A2E1	D1	D4	D4		H2	A2E1G3H2		G3
G1	G1	F2	F2	F2	D4	D2	D2	E1G3H1	F3	A2E1F3	A2E1F1	H1
H2	H2	F1	F1	F1	D2	D3	D3		B2			B1
H1	H1	G3	G3	G3	D3	B2		E1G3H2	C1		A2E1G1H2	C1
F3	F3	H1	H1	H1		C2	B2C2		D1	A2B2C1D1		D3
A1	A1	H2	H2	H2	A2B2	D2	D2	E1F3	D4	A2B2C1D4	A2E1G3H2	D4
E4	E4	G1	G1	G1	C2	D1	D1		C2			E1
G1	G1	H2	H2	H2	D2	D3	D3	B2C1	D2	A2B2C2D2	A2E1F2	F1
H2	H2	H1	H1	H1	D1	C1	B2C1	D1	D1	A2B2C2D1		G1
H1	H1	F3	F3	F3	D3	D1	D1	D4	D3	A2B2C2D3	A2E1F3	H2
A2	A2	A2			C1	D4	D4		A1			G3
B1	B1	B1	B1	A2B1	D1	A1	A1	B2C2	E4		A2B2C1D1	H2
C1	C1	C1	C1	C1	D4	E4		D2	G1			F2
D1	D1	D1	D1	D1		G1		D1	H1	A1E4G1H1	A2B2C1D4	F3
D4	D4	D4	D4	D4	A1E4	H2	E4G1H2	D3	H2	A1E4G1H2		B2
D2	D2	D2	D2	D2	G1	H1	E4G1H1	A1			A2B2C2D1	C1
D3	D3	D3	D3	D3	H2			E4G1H1				D1
					H1						A2B2C2D2	D4
								E4G1H2				C2
											A2B2C2D3	D1
												D2
												D3

* A dash between entries indicates that one or more redundant groups are eliminated between two groups. The relative position of sorted items of equal key has been established arbitrarily.

Fig. 5

the file into batches. Sort item construction and arrangement is shown in column 6.

(d) Sort #4 reads File #4 and creates File #5 (column 9). AF, BF, CF, DF; AF, EF, FF; and AF, EF, HF are the respective key lists of the three item types formed. Type 1 items contain A, B, C, and D groups. Type 2 items A, E, and F groups, and type 3 items A, E, G, and H groups. There are no trailers.

---

trailer); if not, the next group is obtained and stored; when an item is to be issued the table specifies its type and the location of the key arrangement routine to be used. Thus, items of different structure are created. The pre-edit puts them into a single file. If necessary, it also creates a higher-level group file.

After presort, mergesort, and collate phases, the post-edit reverses the pre-edit's operations and recreates the complex hierarchical file. If indicated the sort item file and the higher-level group file are collated. As each sort item is obtained its item type code, created in the pre-edit, is used to select the proper subroutine to rearrange it.

Figure 5 shows the changes effected when F7 is sorted successively according to four different sets of key specifications.

DISK SORTING

Sorting with Large Volume, Random Access, Drum Storage*

Joel Falkin and Sal Savastano, Jr.
Teleregister Corporation, Stamford, Conn.

An approach to sorting records is described using random access drum memory. The Sort program described is designed to be a generalized, self-generating sort, applicable to a variety of record statements. This description is divided into three parts. The first part presents the operating environment; the second defines the general solution; the third part describes the internal sort-merge technique.

1. Operating Environment

The Teleregister Telefile data processor includes drum storage whose capacity is far in excess of the requirements for sorting. This storage is randomly addressable and includes an automatic program interrupt feature which allows data transfers to occur simultaneously with processing. This feature is available for all peripheral data transfers, including tape and automatic typewriter functions. The peripheral transfer, once initiated, is autonomously governed by the slower peripheral device. The main program processing is interrupted only for the relatively short time required by a peripheral device to access one memory position.

The Telefile data processor provides 16,000 positions in memory, each position storing one binary coded decimal character. A floating accumulator arrangement allows the accumulator to contain any field in memory from 1 to 100 characters in length. All indexing is accomplished programmatically. Input and output tape blocking is fixed at 300 characters per block. For this reason, permissible record lengths are restricted to integral submultiples of 300.

2. General Description of Solution (Fig. 1)

The string length in records (N) is given by the formula $N = 13000/(25 + M)$ where: 13000 is the number of available core memory positions for storing source records and the code required to sequence them; M specifies the record length in characters. The constant 25 allows for 3 sequencing instructions at 8 characters per instruction plus a safety factor of 1. The maximum order of the merge is given by the formula $b_0 = 12000/(30 + 2Y)$ where: 12000 is the number of available core memory positions for storing the merge bin and the merge instructions, and Y is the key length in characters. The constant 30 includes the following: three merge instructions at 8 characters per instruction, and 6 characters for a drum address. The value b_0 is the only limiting facter in calculating the number of sequenced strings allowed per program pass; i.e., the capacity per program pass. Note that b_0 is a function of the sorting criteria Y, not the record length M. A 300-way merge can occur at $Y = 5$.

3. Detailed Description of Program Technique (Fig. 5)

The general program flow is illustrated in the Block Diagram. The Sort program is designed to function as part of a library of utility routines controlled by an executive system called the *Locator*. One of the functions of the Locator is to search and transfer control to the programs requested by an operator. This request is made by

* Presented at an ACM Sort Symposium, November 29, 30, 1962.