File Number 1301/1302-07
Form A22-6784

IBM 1301 and 1302 Disk Storage

Sequential Data Organization

This manual describes a way of storing and retrieving data on IBM 1301 and 1302 Disk Storage that is largely independent of the characteristics of the application. The method will enable most users to install and operate disk storage efficiently, with minimum detailed study of the data. A common set of programs and techniques can be used for all data files in disk storage. The approach allows data to be loaded in the most useful sequence, yet allows for random or sequential access to the records. As the data file is loaded, an index is created that associates data identifiers with actual track addresses and permits expansion by an easily used overflow technique. Conversion is accomplished in an orderly and efficient transition from data handling methods presently in use.

This manual assumes knowledge of the manual, *IBM 1301 Disk Storage with IBM 1410 and 7010 Systems,* Form A22-6770-1, or the General Information Manual, *IBM 1301 Disk Storage with IBM 7000 Series Data Processing Systems,* Form D22-6576-3. Examples in the text of this manual are based on IBM 1301, Models 1 and 2, Disk Storage.

Copies of this and other IBM publications can be obtained through IBM branch offices.
Address comments concerning the contents of this publication to:
IBM Corporation, Customer Manuals, Dept. B98, PO Box 390, Poughkeepsie, N.Y.

Contents

Sequential Data Organization for IBM 1301 and 1302 Disk Storage 5
Random Processing of Exceptions 6
Sequential Processing 6
Batch Processing 6
Scan Processing 7
Programming Efficiency 7
Data File Indexes 8
Additions to Data Files 9
Data File Deletions 11
Data Directory Index 11
Data Storage Allocation 12
Conversion From One Type of Disk Storage to Another 13
Indexing Blocked Records 14
Reorganizing the Data File 14
Index Usage in Sequential Processing 15
Index Usage in Random Processing 15
Index Usage in Scanning 16
Sorting versus Distributing 16
Record Mode versus Full Track Mode 16
Input-Output Storage 17
Program Storage 18
Processing Sequential or Random Input Transactions 18
Balancing Partially Processed Data Files 18

Appendix A. Index Formats for 1410 and 7010 Systems 21
Track Index Format 21
Prime and Intermediate Index Format 21
Data Directory Index Format 21

Appendix B. Additions to Track Index 23

Appendix C. File Growth and the Track Index 24

Appendix D. A Comparison of Sequentially Processed Full Track Mode and Record Mode Operations 25

Appendix E. Estimating Access Motion Time for IBM 1301 Disk Storage 27

Sequential Data Organization for IBM 1301 and 1302 Disk Storage

The purpose of IBM 1301 and 1302 Disk Storage is to provide a convenient place for storing and retrieving data. Data records should be stored with a plan that will facilitate their later retrieval. This organization of storage can have a random or sequential plan. But before the plan is chosen, the manner in which the stored data will be processed should be considered. Figure 1 shows the relationships of data organization and processing. The approaches shown are:

Case 1: Sequential processing of sequentially organized data.

Case 2: Random processing of sequentially organganized data.

Case 3: Sequential processing of randomly organized data.

Case 4: Random processing of randomly organized data.

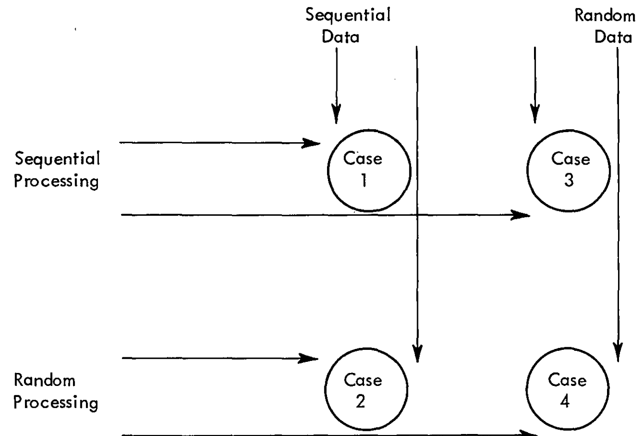


Figure 1. Four Relationships of Data Organization and Processing

Cases 1 and 2 are the major approaches. Case 3 finds limited use in most applications, but Case 4 offers unique benefits to selected applications, particularly where the data files undergo frequent additions and deletions, or where the bulk of the transactions must be processed randomly.

The great power of disk storage operations results from the freedom, provided in Cases 2 and 4, from the Case 1 requirement that input data be in the same sequence as the sequence of the stored data records. Figure 1 shows that, if data can be processed either sequentially or randomly, the system can continue to enjoy the advantages of sequential processing in addition to the benefits of random processing. This manual presents a recommended way of organizing data sequentially. Associated with the organization scheme are indexes that facilitate either sequential or random processing of the data files.

The retrieval of current information permits an in-line data processing concept, whereby all of the records affected by a transaction are posted at the same time. For example, when a receipt of goods is recorded in inventory records, the on-hand or on-order records are changed, and a record of work-in-progress, or a vendor's account can be posted. Back-order releases and vendor payment also can be triggered by such a series of actions, and sales and liabilities in journal entries can be posted.

Other disk storage advantages are equally important and may be more readily realized, thereby offering benefits to the user more quickly. The following benefits are discussed throughout this manual.

Random processing of exceptions.
Batch processing advantages retained.
Scan processing and report preparation facilitated.
Simplified conversion of data files; extensive study of identification key structure eliminated.
Simpler programs permitted — and simpler cross-referencing to inter-related data files.
Reduced response time.
Reduced effective record access time.
Reduced system set-up time.
System readily accepts growth of data.
Disk storage filled efficiently.
Simple overflow procedure for record additions.
A practical means of balancing partially processed data files.

Many of these advantages are retained from existing sequential processing methods. Others derive directly from the added capabilities of disk storage. Still others are the particular advantages that accrue to the combination of sequential, random, and scan processing within a single system.

Random Processing of Exceptions

The direct, rapid processing of exception data or of individually initiated transactions leads to increased accuracy of processing and to improved clerical efficiency. The ability to process a single transaction through an on-line terminal enables the data processing system to confirm or question the accuracy of input data while the operator still has the details of the transaction before him. The terminal operator learns correct procedure from the start and is spared multiple corrective actions; a mushrooming of errors in the input data is avoided.

Because the data file is easily accessible, random processing of exceptions in the input data may be eliminated from one run and processed as a separate run. An increase in efficiency may be realized, because the main file run is relieved from exception considerations and *fast-path* programming is possible. The exception conditions sometimes prevent processing certain items in one run, with the result that the affected record must be processed manually, for that operation and all others as well. Other items are found that fit the first run but are disqualified from the second, and so forth until a substantial body of exceptions is built up. The exceptions may appear to be a small percentage of total volume, but they can become a large job in themselves, requiring an inordinate amount of work. The writing of a relatively few routines or special programs provides for complete processing.

Sequential Processing

The most widely used system of organizing data is a sequential order based upon the alphabet, number series, or ascending values of some kind. The number can be a coded part number, customer number, etc., or it may be a serial number assigned in the sequence of arrival, such as an insurance policy number. Such a number or name, when it determines both the order of the data file and identifies the record, is called the *primary record identifier* or *key*. The key is always associated with the record.

Most data files now in use are maintained in a particular sequence. Retention of this sequence avoids radical departure from the methods familiar to people outside the data processing activity. Their procedure manuals need not be changed except where disk storage capabilities are to their advantage.

If the existing sequence is retained when loading data into disk storage, detailed study of the structure of the coding system used to identify individual records is not required. At installation time, data in disk storage and in the unconverted data file are maintained in the same sequence; precision monitoring of the conversion is made feasible.

Batch Processing

Batch processing existed long before the advent of data processing systems. Mail deliveries, messenger services, or other forms of data transmission may tend to create batches out of individual transactions. Work scheduling, cash control, and accounting control are simplified by accumulating many separate transactions into batches that are processed together in a run. In short, data processing system requirements have an effect of modifying the size of batches or the frequency of processing, but many forces external to a data processing operation continue to exert themselves in the formation of batches.

Attempting to process batches as random transactions can introduce inefficiencies and unnecessary control problems. If the transactions that form into natural batches are permitted to be processed as such, more time may become available to the system for the processing of items that are not batches, but are truly random transactions.

Rapid processing of all transactions may be unnecessary. In many instances the record-keeping can lag behind the actual transactions without adverse effect; for instance, detailed record posting on accounts receivable may lag in many applications by hours or even days, and when cash may be deposited and used before consulting the data records. Such low priority transactions can be accumulated and efficiently batch processed, thereby making the system more available for processing high priority transactions.

This processing of small batches of transactions has several advantages. Obviously, a few records can be updated in disk storage in less time than is required to pass a lengthy main file of data. The ability to process small batches of transactions efficiently means that the need to attain a high activity ratio on each run is reduced and many different types of transactions can be processed in individual batches. Allowing transactions to remain in their individual batches, and processing them as such, permits attack upon fewer logical problems at a time, simplifying programs and reducing the amount of core storage needed for processing.

A decision to process everything in batches or at random should be avoided. A more efficient operation will result from the use of judgment in separating the total data processing task into batch elements and individual elements. Each type can then be handled appropriately and efficiently.

Scan Processing

Data file processing can be governed by factors other than the receipt and posting of transactions. The status of records can be affected by time; for example, a quarterly date can require production of social security reports or income tax statements. Premium notice and cycle billing dates are other examples. Such time considerations may be combined with other factors: accounts receivable balances, account activity and history, and credit limits. One action may be taken on accounts of a certain size when they are overdue a specified number of days, and a different action may be taken on accounts that are overdue for a longer period, regardless of the balance due.

Such actions, depending on a record status with respect to a particular point in time, require disk storage data to be brought into core storage where records can be scanned for particular conditions. This is called scanning the data file. Each record in the file is scanned to determine whether action is required.

If the coding system used as a basis for the record identifier keys has any classifying or grouping function, the use of sequential organization provides a unique advantage. When items in a group or section of the coding structure are to be processed, only the area of the file containing them needs to be scanned because the file organization preserves the intelligence of the coding sequence. The coding scheme can be presumed to have placed the data in its most frequently used sequence. Therefore, reports produced from a sequential scan usually will be in the most useful sequence without requiring special sorting.

Most sequentially oriented data processing systems combine the process of scanning records for action with batch processing procedures, because it is usually necessary to retrieve each record in the data file anyway. Such scanning is a by-product of the necessary transaction processing. But the combination of batch processing and scanning can lead to conflict when batches are withheld from processing until a scanning operation is required, or when scanning is performed more often than necessary. A data processing system with random processing abilities diminishes these problems by enabling each task to be performed with its own optimum schedule.

Programming Efficiency

The use of disk storage devices offers many new opportunities for the system designer and programmer. The use of disk storage for subroutines or programs in addition to the basic option of processing data sequentially or randomly provides a new level of flexibility in systems design.

Subroutines stored on the disk require no space in core storage except when they are needed. With suitable programming standards, a number of subroutines can share a given area in core storage that will be used by the particular subroutine needed at a given moment. The subroutines will not compete with each other nor with the main routine for core storage, and fewer compromises will be necessary in each. Also, the programmer need be less concerned with fitting his programming into as little space as possible.

This compacting of programming can be time consuming. In an effort to eliminate an additional pass of the entire main file, the programmer frequently recodes into tighter loops, and sometimes reduces the processing of some conditions. This may be required not only in the initial programming effort but in subsequent program maintenance as additional requirements appear.

With disk storage space readily available for subroutine storage, the systems designer can proceed in the initial layout of a system with confidence that all processing planned for a run can be achieved in that run. If the program exceeds a single "core load," only the time necessary for obtaining subroutines is added to the run time, instead of the time necessary to process the main file in an additional separate pass.

Records in disk storage usually are written back in complete form and in their original location. Therefore, the use of extract or intermediate records gives way to repetitive use of the complete record. This facilitates the use of standard subroutines which are modified at object time to accommodate the current condition of desired data files. Freedom to combine collections of standard subroutines can contribute dramatically to program simplification.

Reduced System Set-Up Time

Use of disk storage as a program storage device, as an input buffer, for main file storage, and as an output buffer (all attached and on-line) can reduce the set up time required to access and read a program. This time is measured in seconds and fractions of seconds rather than in the minutes required in many instances. This reduction of set up time is realized to an even greater extent by the entire elimination of the special runs for the special data files previously required to accumulate and hold exceptions.

Positive Control of File Referencing

In sequentially organized data, a data record can exist in only one place — between the next lower and

next higher numbered record. Failure to locate the desired record in such a location is adequate proof that the record does not exist. Such positive record control reduces the possibility of accidental creation of duplicate data records and facilitates their detection.

Data File Growth

The recommended methods of data storage and indexing are relatively insensitive to changes made subsequent to the initial storage assignments. If planned allocations of storage become insufficient due to growth of the data file caused by acquisition, merger, contract awards, etc., additional cylinders of disk storage can be assigned without relocating other data in storage or causing program changes. The cylinders referenced by the index need not be adjacent or even in sequence.

Disk Storage Filled Efficiently

Because data records are packed in the primary storage area without providing space for the overflow records the primary storage area is completely utilized. One track in each cylinder of 40 tracks (2.5 percent of the storage space) is used for file index records.

Several data files can effectively share a pool of available cylinders. Overflow areas are then allocated from the pool as required. This avoids prior allocation of overflow areas based upon maximum possible expansion requirements for each data file.

Data File Indexes

When a data file is loaded sequentially into a disk storage device, no particular relationship is automatically established between the record identifier keys of the data records and the addresses of disk storage tracks. In order to retrieve records without starting at the beginning of the file and searching until the desired records are found – to retrieve records randomly, a relationship between record keys and track addresses must be established. One method is to form a table that shows the storage address for each data record key, but such a table becomes a data file in its own right and has the same addressing problem: a method would be required to provide the addresses of the keys in the table if extensive sequential searching of the table is to be avoided.

A better method is to form an index that takes advantage of the sequential organization of the data and the cylinder concept of disk storage. A single entry, that of the highest key in each cylinder, will serve to identify all the records located within a cylinder. Within each cylinder a more detailed index that contains the highest key of each track of data is used to provide the track address of each record. The index to the cylinders is called the *prime* index. The index within each cylinder that identifies specific tracks is called the *track* index. With very large data files, the prime index may become too large for convenient use and *intermediate* levels of indexes may be formed. These have the same general format as the prime and track indexes.

In addition to relating record keys with track addresses, indexes permit disk storage to be filled efficiently because empty space need not be provided for future expansion. Disk storage is packed full, and additional records are placed on an overflow track. (Overflow techniques are discussed and illustrated later in this manual.)

When formatting indexes or data for IBM 1301 and 1302 Disk Storage, full track operations handle data tracks formatted in either record or full track mode and record mode operations depend on the data track being formatted for record mode.

Prime Indexes

The prime index is formed each time the data file is loaded or reloaded. It does not control the loading of data but provides a record of where the data has been placed. The prime index is composed of two basic portions, a fixed length header and a variable length entry. The header consists of an identification field, coded to indicate that it refers to other lower level indexes, and a count field that defines the number of entries in the entry portion.

The entry portion has an entry for each indexed cylinder. When intermediate indexes are used, each entry in the prime index will refer to an intermediate index. Each entry contains the record identifier key of the last record loaded in the cylinder. It is the highest record key in that cylinder. Associated with that key is the address of the track index that serves that cylinder.

Because it is easy to calculate the address from the known position of the key within the index, there is a temptation to avoid the use of the address altogether. But this imposes a requirement that the data file be located in adjacent cylinders of disk storage. Substantial flexibility in storage assignment is gained by including the address; data files can be spread among multiple modules of IBM 1301 and 1302 Disk Storage, or interspersed among data files that have diminished in size or failed to grow as originally anticipated. Inclusion of addresses provides a basic

independence from the sequence of the data file and the sequence of the track addresses.

The prime index is carried in disk storage, but can be read into core storage and retained there during periods of heavy usage. When high performance is required for inquiry responses, the prime indexes for all data files and their associated intermediate indexes may be placed in an auxiliary device such as IBM 7320 Drum Storage or in IBM 1311 Disk Storage.

Intermediate Indexes

Intermediate indexes are formed whenever it is necessary to limit the size of the prime index. The intermediate index can also be limited in length to suit the convenience of the using system, by creating another level of intermediate index. The format is identical to the prime index with a fixed header portion and a variable number of entries, one entry for each cylinder referenced by the intermediate index.

Access time is reduced by storing the intermediate indexes in the same cylinder with the prime index or in an auxiliary device.

Track Indexes

Like prime and intermediate indexes, track indexes are formed as loading of data occurs. To minimize access time to data from the track index for sequential or scan processing, the track index should ordinarily be stored in the same cylinder as its data.

Track indexes have the same format as prime indexes; the header portion is followed by the entry portions. Programming convenience requires fields in all indexes to conform in size and relative position so that they may be searched by one common subroutine.

One of the fields in the header of a track index contains a flag that identifies the index as referring to data tracks, not to an intermediate index. This flag can be tested to cause selection of a different work area in core storage or to cause the selection of a record address subroutine for the retrieval of individual records. It can also be used when data tracks carry unique home address (HA) tags in the HA2 portion of the address.

Additional fields in the header portion contain references to the prime index and to the next sequential track index. These fields provide additional flexibility in the use of the indexes in sharing the same core storage area or for sequential scanning of the data file without constant referencing of the prime index.

A count field of the number of subsequent entries in the track index permits binary searching of the index or termination of sequential searching of the index entries.

The entry portions of the indexes are similar to those in the prime and intermediate indexes. They consist of two parts: a data key that is the key of the highest numbered data record stored on the track and the track address of that data. There will be an entry for each track indexed; 39 entries are required for a fully loaded 1301 cylinder. Where multiple data files are contained in a system each data file would have different indexes. The keys in different file indexes can be of different lengths, but within a data file and its indexes the key lengths should be of equal length.

Standard Index Format

The preceding considerations lead to a standard format for indexes that can be searched by a common subroutine. These fields can be employed in the following order:

Flag: This is a two-digit field to identify the index as a prime or intermediate index or as a track index.

Address: This field carries the track address of the prime index for track and intermediate indexes. The literal PRIME can be used in this field in the prime index.

Next: In track indexes this field carries the track address of the next sequential track address. In prime and intermediate indexes this field carries the literal PRIME or INTER.

Count: This three-digit field carries a count of the number of reference entries in the index.

Entries: This field has multiple entries of highest keys and their associated track addresses. The overall length will vary with the number of keys and their length.

A specific index design is described in Appendix A.

Additions to Data Files

The indexes described pertain to sequentially loaded data files. They are formed as part of the loading process, with the prime and intermediate indexes written in a selected area and the track indexes written with their data. Additional records for the data files can be expected. They may be too few in number to warrant reorganization of the file or they may occur when reorganization of the data file is inconvenient. Provision must be made to add records to the file without disrupting the organization and without causing undue system scheduling conflicts.

One way of adding records would be to provide space at the time of loading in anticipation of the future. This space could be on each data track or space could be allowed in each cylinder. Either ap-

proach, however, assumes that additions to the data file will be evenly distributed throughout the file. In practice this will seldom be true.

When growth does not occur, any space left open is wasted, and where growth does occur any open space based upon averages will soon be insufficient. These inconveniences are inherent in the dynamics of data files. Insurance policy files generally grow at the high end of the sequential order (but even this growth is uneven for there is no requirement that the policies must be entered in the data processing system in the order of number assignment). Public utility customer files grow in areas represented by suburbs and diminish in city areas (this growth is uneven as lots are built up and housing developments are opened). Even inventory files evolve unevenly as products become obsolete and new products with different part number series take their place.

These difficulties often prevent accurate estimates of the positions of growth in data files even when the total growth can be forecast. As a result, the efficient allocation of storage for future records within the confines of a data file is nearly impossible. These problems are minimized by taking advantage of the efficiencies of packing data sequentially into data storage. An overflow area can be provided for the additional records as they occur.

Allocation of a single area into which all overflow records are stored simplifies the considerations of file growth. Growth within each portion of the data file can be ignored and concentration given to total growth of the file. This is particularly true of sequentially indexed files because they readily lend themselves to reorganization at which time the overflows are merged with the main file records and reloaded into the primary disk storage area.

The space resulting from deletions that occur in the data file can be ignored until the file is reorganized. Deletions are discussed later on in this manual. Space allocation for overflow data is based on the gross number of additions expected, instead of net file growth.

Since additions can occur anywhere in the file it is just as likely that they would occur between items stored within a track of data as it is that they would occur at the end of the tracks. But if they did always occur at the end of the track the index would still give the benefit of a positive indication of the location of the record. There would be no need to respond to a no record found condition by a further search of the index or with some type of scanning of the overflow area.

Additional records, therefore, should be inserted in their proper sequence on a track by moving higher numbered records toward the end of the track. Records at the end of the track are moved to the overflow area then in use and entries are made in the track index for the new records and for the overflow records. The overflow entry is made in its correct sequential position in the track index. Thus, insertion in the file affects three tracks: the data track that receives the addition, the track index of that cylinder, and the track in the overflow area that receives the last displaced record. Access time for index maintenance is eliminated because the track index is in the same cylinder with the data in which the change is made.

Figure 2 diagrams the effect an insertion has on the data track and its track index; BEFORE and AFTER conditions are shown. Data records significant to the Figure 2 example are represented by their identifier keys.

The record with key 2136 is inserted in the records on track 0242. It is written in proper sequence between record 2082 and record 2307. This changes the highest track index key for track 0242 from 2684 to 2307. Record 2684 is bumped from track 0242 and written on overflow track 1131.

Fields changed by the insertion procedure are underlined in the *after* part of Figure 2, including the count field. All of these changes are reflected in the track index on track 0240. Further insertions into the data track would lead to additional displacement of records from the primary area into the overflow area. Any insertions made between the last record of a track in the primary area and a record already in the overflow area would be referred automatically to the overflow area without changing the primary area. Insertions that occur between a record already in the overflow area and a record in a primary area track would be made in the primary area with the resulting displacement of a primary record into the overflow area.

No insertions will ever be made that have a key higher than the highest key already loaded into a cylinder and therefore the higher levels of indexes will not be changed by the insertion or addition of records. The reason for this, of course, is that a search for any key higher than the highest key in a cylinder will be referred to the next cylinder in the file by the higher levels of indexes.

Using a field of nine's rather than the actual highest key in the data file for the last entry in each level of indexes prevents the possibility of searching for a key that is higher than the highest key contained in the indexes.

BEFORE

Track Address	Index Header	Count	Key	Address	Key	Address	Key	Address	Key
0240	. . .	39	0117	0241	2684	0242	2893	0243	7528
0241	. . . .								0117
0242	0395	0460	0465	1839	2082	2307	2684		
0243	. . . .								2893
0279	. . . .								7528

AFTER

Track Address	Index Header	Count	Key	Address	Key	Address	Key	Address	Key
0240	. . .	40	0117	0241	2307	0242	2684	1131	7528
0241	. . . .								0117
0242	0395	0460	0465	1839	2082	2136	2307		
0243	. . . .								2893
0279	. . . .								7528

Figure 2. Record Insertion Example

Data File Deletions

A record deleted from the data file leaves a blank space unless the other data records are shifted to close the gap. Even then a space remains unless a record is brought into the primary area from the overflow area. To use the place made available by a deletion, a corresponding overflow record must exist to fill the slot; the overflow record must come from the same track in which the deletion occurs.

Additions to a data file tend to occur in clusters in specific portions of the data file or at the high end of the sequence. Deletions seldom occur in these same areas. Where growth of the data file occurs at the upper end, the deletions tend to occur throughout the file (insurance policy files are an example). Where public utility files grow in specific areas corresponding to real estate developments, the deletions occur in other clusters from slum clearance and road programs. When new products lead to a new part number series in an inventory file, discontinued products obsolete part numbers in other number series.

The effect of clustering additions and deletions upon a sequentially organized data file is more pronounced than it is upon a randomly organized file because the essential integrity of the key structure is maintained in a sequential file. Thus there is much less chance of deletions and additions offsetting each other in the data file. The additions are readily accommodated by the overflow techniques already discussed. The deletions are harmless and can be ignored until the data file is reorganized. At that time, the overflow records are merged with the records in the primary area and the file reloaded, without gaps or overflows, with all records in the primary area.

Data Directory Index

In addition to the prime, intermediate, and track indexes for each data file, an index can be created that contains information about each of the data files in disk storage. This data directory index is created independently of the loading program and is used to govern all disk storage operations. The data directory index is entered by using the code number or name of a desired data file to learn the track address of the prime index for that data file. The data directory index can also carry information about each data file that can be used to modify generalized programs that use the data files.

The data directory index provides central reference information from which a single starting point can gain access to any data file. Because the data files are referenced through the data directory index with a code or name, unintentional reference to data files is minimized; a program must request the file information for a specific data file in order to gain access to an individual record.

Like the data file indexes, the data directory index has a header portion and a series of entries. The header portion can consist of the identifying literal DIRECTORY INDEX, an overflow track address for this index, and a count field showing the number of data files referenced by this index. The header on the overflow track can specify another overflow track; as many tracks as necessary can be chained together in this manner. The last overflow header would carry the literal END in the overflow track field to identify the last track in that index.

The entry portion of a data directory index contains a reference field for every data file in disk storage. Each reference field contains the code number and name of the data file, the track address of the prime index for that data file, and other parameters used to modify generalized programs so that they will operate on all data files in disk storage.

Considerable sophistication may be employed in these parameters; fields may be provided for the following:

The length of the identifiers or record keys used.
The position of the key within the record.
The length of records in the data file.
The modes — six or eight bit and full track or record.
The effective full track length for tracks formatted in record mode.
The length of the track address fields in index references.
The address of the track currently used for overflow.
The address of the last track available for overflow use.
The date any entry in the data directory index was last changed.
The identification of the program that last changed an entry in the data directory index.
A chain address to seldom used information regarding the data file described by this entry in the data directory index. This supplementary information may be the following:
- A list of programs that reference this data file.
- Identification of the cylinders the file occupies in the primary area and in the overflow area.
- Identification of the yet unused cylinders assigned to this data file.
- An expiration data for this data file. (A permanent data file could carry the literal PERM in the expiration date field.)

This list can be expanded to meet the desires of individual installations.

The data directory index provides a common source for information on any data file in disk storage. It reduces the number of control entries that must be made to the generalized programs that manipulate the data files in disk storage. Generalized programs for random access to many data files make usage of a data directory index virtually unavoidable.

Data Storage Allocation

In the indexes described, each entry carries a direct disk storage address. Many techniques are available to eliminate these addresses and thereby lessen the storage space required for the indexes. These techniques depend, however, upon the sequential nature of the addresses. They must be in some sort of computable series and if a common routine that can search any data file index is to be used, they must be in continuous series. Disk storage addresses are included in the indexes in order to eliminate this requirement of continuity. Addresses in the indexes permit the data file sequence to be independent of the track address sequence in disk storage.

The use of addresses provides many advantages:

Freedom from commitment to the initial disk storage allocations.
Ability to relocate data files without reprogramming.
Ability to intersperse data files.
Ability to distribute data files into multiple modules of disk storage.
Simplified conversion from one type of disk storage to another.

Independence from Initial Allocation of Storage

In the course of developing applications for disk storage it is frequently impossible to develop the entire set of programs for the ultimate system at one time. Provisions must be made, therefore, for the data files that will eventually reside in disk storage and often these provisions must be made before the characteristics of the files can be studied adequately.

Many questions arise: How fast will the files grow and how much space will they require when placed in disk storage? What will be the exact record requirements for the next series of applications and what arrangement will provide adequate space without undue waste? What must be done if a data file sandwiched between two other data files grows beyond those boundaries?

The use of an indexed file with addresses in the index provides ready solutions to these problems. A file can be loaded into the first ten cylinders plus the seventeenth, or the last 50 cylinders plus the 23rd, 36th, and the 8th without affecting the programming of that file. Even the effect upon access time is negligible in sequential processing and in file scanning. In random accessing, the difference between accessing adjacent and distant cylinders is small when the proportion of distant cylinders is small.

Ability to Relocate Data Files without Programming Changes

When a data file that is between two other files grows beyond its bounds, or when it is being pressed for space by one of the other files, it may be desirable to move a file from its current location and relocate it in another area of disk storage. This relocation requires only a change in the content of the index without a

change in the format of the index, and the change can be made easily with no effect upon the programs that search the indexes.

Ability to Intersperse Data Files

When storage allocations have been made conservatively and data files have failed to grow to full occupancy of their assigned space, it is possible to recapture that space by loading a data file in the available space between other files. It is easy to relocate the file if it becomes necessary or advisable.

Where references are frequently made to related data files, it may be desirable to intersperse them in some manner. This is particularly true in disk storage systems with a single module of storage. When additional modules are added later it may be more effective to place the data files in different modules. This would have no effect upon the programs using the data. One file would be relocated into the additional module; the other file would be consolidated within the original module and the space remaining would become available for an additional data file. Complete flexibility of the allocation and subsequent manipulation of location data file is obtained.

Ability to Distribute Data Files into Multiple Modules of Disk Storage

Although this manual is devoted to description of sequential organization of data in disk storage and may seem to emphasize sequential processing and scanning of those files, the importance of random processing should not be overlooked. When random processing is desirable and multiple modules of disk storage are available, the opportunity to minimize access delays by overlapping accesses to different modules should be considered. This can be done by dispersing a data file into portions of several modules of storage in such a way as to cause an approximately equal number of random transactions to reference each disk storage module. If the random activity is distributed equally throughout the data file, its distribution should also be equal throughout the available modules.

Such distribution of data files requires knowledge of the processing dynamics of the files that may be unknown at the start of the system. The facility for relocating files can be used to study the system after installation by trying various allocations and timing the procesing under actual conditions; some advantage can be realized even without consideration of overlap. Dispersing the file over multiple modules can decrease the span of cylinders that any one access mechanism need cover.

If a given data file occupies cylinders 0-19 in one module of storage, the average random access time within the data file is 82.5 milliseconds. For the same file distributed in cylinders 0-9 in each of two modules of disk storage, the average random access time within the data file is 45 milliseconds. (These figures are based on the calculations in Appendix E.)

Conversion From One Type of Disk Storage to Another

New paths of growth for data files are provided by IBM 1302 Disk Storage. The use of indexed sequential files and the use of generalized programs to function with the data files minimizes the problems of disk storage conversion. For if all data files are loaded with a common routine and are retrieved with a common subroutine, only these parts of disk storage programming need be changed in order to accommodate the new storage device. If these routines are modified at *object time* with data directory index information, conversion from 1301 to 1302 disk storage is simplified. This is accomplished in relation to whether data is formatted in 1301 disk storage in record mode or in full track mode.

Conversion of Record Mode Data

Data formatted in record mode in 1301 disk storage can be readily formatted in record mode for 1302 disk storage. Two 1301 cylinders for data will usually occupy one 1302 cylinder (Figure 4).

A problem may exist, however, where programming has handled track indexes with full track operations. For example, the track index associated with ten 1301 records of 245 characters each is a 2,518 character "record" as shown in Figure 8 (Appendix B) when handled in full track mode. If this track index is placed in the same 1302 cylinder as the prime/intermediate indexes, it can be formatted to be read into the same size core storage area as was used in 1301 index operations; otherwise, if the track index is put in the same cylinder as its data, the full track operation would attempt to move a 1302 track with a format length of two tracks of 1301 data (5,036 characters).

Consideration may be given to putting the record mode 1301 data into 1302 disk storage as if it had been recorded in full track mode in the 1301, as follows.

Conversion of Full Track Mode Data

The following approach is suggested for conversion of indexed sequential data files stored in 1301 disk storage in full track mode. Format the 1302 data tracks to accept two records, each 2,800 characters in length. This will permit the data files to be loaded into

the 1302 in record mode. Two cylinders of 1301 indexed data will fit into one 1302 cylinder with a track left over for data or indexing (Figure 3). This track is formatted for two 2,800 character records; one record can be the track index.

A converted track index is different than the indexes previously described because it has two entries for each track of 1302 disk storage. Each entry refers to a block of data records that can overflow. Previous description has shown overflow to occur at the end of a track; the method now described overflows at the end of block instead of at the end of a track.

A consequence of this method is that a track index will have twice as many entries. Adequate space is available, however, as shown in Figure 8 (Appendix B). In fact, the track index can be recorded twice on the 1302 track, thereby reducing rotational delay time to 8.5 milliseconds.

The prime or intermediate indexes must be changed to indicate that they refer to index records written in record mode instead of in full track mode. This change can be indicated by flag bits in the prime/intermediate indexes.

Furthermore, because the data was stored originally in full track mode and now is stored in record mode, the address length shown in the data directory index should be changed to specify seven characters instead of six. This permits the units position of the seven character address to serve as a one-character record address. It can be a 1 or a 2, depending upon whether it is for the first or second block of 2,800 characters.

Indexing Blocked Records

The procedures discussed for use in placing 1301 data files in 1302 storage can also be adapted for indexing groups of small records in 700 or 1,400 character blocks for 1301 operations.

Reorganizing the Data File

Periodically, it will be necessary to reorganize the data file. This requires unloading the data file, merging the overflow records, and reloading the file into the original area. If the file has grown since the previous loading, additional space will be required for the data file. The overflow records are merged during the data file unloading operation to produce a complete and usable magnetic tape record of the data file that can be useful in a normal reporting procedure or auditing function. This tape is used directly for reloading by the loading and indexing program. It may then be filed to fulfill record retention protection requirements.

Reorganizing the file can be both a very simple operation and a regular part of normal operations. The time spent to reload and reindex the file may be wasted if reorganizing is done too frequently, but processing time may be wasted – particularly during random processing – if overflow records cause too many additional accesses to the overflow areas. The time for each of these additional accesses can be determined quite accurately because it depends on the distance between the primary storage area and the overflow area. Since these overflow accesses are pro-

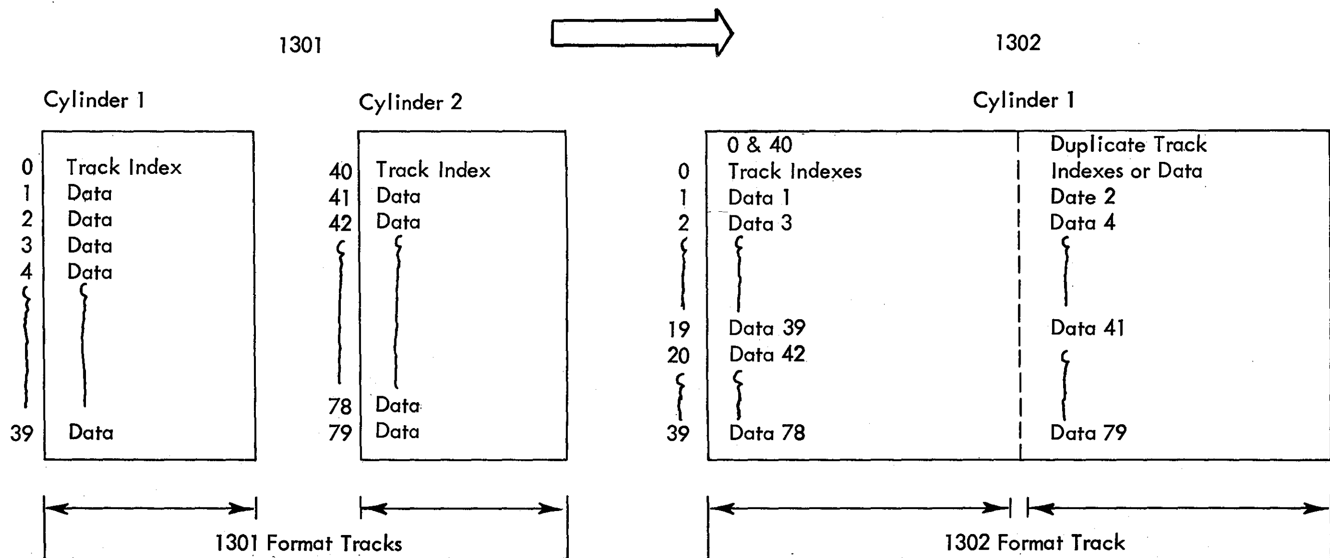


Figure 3. Example of Conversion of 1301 Full Track Format to 1302 Record Mode Format

gram controlled, they can be counted and the total time for them calculated. Whenever that time exceeds the time to reorganize the file, reorganization is in order. When a projection of future time losses that would occur if reorganization waited for the next convenient time to reload exceeds the time required for reorganization, then reorganizing is also advisable.

Index Usage in Sequential Processing

The indexes used in random processing are also necessary for sequential processing. During sequential runs, the overflow addresses on the track indexes provide the means of retrieving these records in their proper sequential order. Where only a portion of the file is to be scanned, a comparison with the prime index permits skipping to the proper starting point. When processing input data, reference to the cylinder index can show whether a given cylinder has any activity at all and reference to the track index can eliminate reading many tracks in the primary or overflow storage areas. This permits efficient processing of files that have low activity.

Index Usage in Random Processing

Indexes provide the ability to access randomly an individual record without scanning the entire data file. If the cylinder index is in core storage during the processing of random transactions, then the random capability of disk storage is retained and comes close to the performance of a system deliberately designed for random processing. Only an additional read cycle, that of the track index, has been added.

Index usage in random processing has several timing considerations. For example, assume a file stored in a single module of 1301 disk storage, and assume the processing of 50,000 transactions.

Random Processing, Random File Organization: The time to access the tracks on which the records are located is a function of the number of records, the average number of accesses to locate the records, and the average access time per seek. For the purposes of this example, the estimated number of effective seeks per item retrieved is 1.2. Therefore, random access time = 50,000 x 1.2 x 0.165 seconds = 9,900 seconds = 2.75 hours.

Sequential and Random Processing, Sequential File Organization: Assume that eighty percent of the transactions can be accumulated and processed in batches. Five batch processing runs are assumed, with 8,000 items in each batch. The remaining 10,000 items will be processed randomly, as they arrive.

To access all the cylinders in a 1301 module sequentially takes 14.9 seconds (minimum access times are used for all cylinders except the first which is considered to be a maximum). Therefore, batch processing access time = 5 x 14.9 seconds = 74.5 seconds = 1.25 minutes.

To access 10,000 records randomly requires 10,000 x 1.0 (average number of seeks) x 0.165 = 1,650 seconds.

Additional time is required to read and search the track index stored on each cylinder. Average rotational delay time plus read time equals 51 milliseconds and search time is 34 milliseconds. These two factors total 85 milliseconds and 10,000 times 0.085 equals 850 seconds. Therefore, total random access time = 1,650 +850 = 2,500 seconds = 41.6 minutes.

In order to achieve minimum access times in sequential processing, the input data must be in the same sequence as the data file and the time to sort these items must be added. 1301 sorts can sequence 8,000 items (100 character records) in approximately 4 minutes. Taking a conservative magnetic tape sorting time (1410-729 VI) of 6.5 minutes, 6.5 minutes x 5 = 32.5 minutes.

Finally, allowance must be made for additional accesses for records that are in overflow locations. This time cannot exceed 24 minutes for the entire job, including both sequential and random processing; reorganization of the file would be called for. Therefore, an average time of 12 minutes is used.

To recap:

Random Processing, Random File Organization	
Total Access Time	2.75 hours
Sequential and Random Processing, Sequential File Organization	
Sequential processing access time	1.25 minutes
Sequential index read and search time	1.77
Batch sorting time (magnetic tape)	32.5
Random processing access time	41.6
Additional access time for overflow records	12.0
Total Access Time	89.12 minutes = 1.5 hours

Note that the example deals only with access times. The process times which would be involved are not considered in the timings used; they would be approximately the same for sequential or random file organizations after the data records had been accessed.

The example shows an advantage in total access time in favor of sequential processing. The advantage is contingent upon the ability to process the bulk of the work in batches. It is also contingent upon the size and location of the data file.

On the other hand, a further advantage to sequential processing can be realized by performing the sorting operations on auxiliary equipment, thereby relieving the central processing system of this task. In the example above, 32.5 minutes of sorting could be transferred to auxiliary systems, leaving approximately 50 minutes of access time on the main system. This is contrasted to the 165 minutes of access time required if a random processing approach is used for all transactions.

Some additional time is required in the random processing of sequentially stored data because the indexes must be consulted in order to locate a data record. It is assumed that the cylinder index can be retained in core storage during the processing of random transactions but the additional time to read and search the track index cannot be avoided. Where the individual response times are critical, such a factor should not be overlooked. Processing low-priority items sequentially, however, reduces the requirement for random processing and increases system availability for random processing.

Index Usage in Scanning

In sequential scanning of a data file the indexes are the guide to the location of the data file. The location of the initial track index is learned from the prime index specified by the data directory index.

Track indexes are used to control scanning because cylinders may not be assigned to the data file in ascending sequence. Also, the use of track indexes facilitates handling overflow records.

Each track index contains the address of the next track index in sequence. When completing the processing of a cylinder, the track index in use provides the track index for the next cylinder of data. This chaining of track indexes facilitates scanning.

Data from primary areas in disk storage is readily scanned in core storage, but records from overflow areas may present a problem. One way of solving the problem is to transform the overflow record key in the track index for use as a record address or for use in selecting a record from a block in full track mode.

Sorting versus Distributing

Random processing of input data can eliminate sorting runs and lists can be used to control scanning for the generation of reports from a random file, but the function of sorting, the business of putting data in order, must still be accomplished in order to process the data efficiently.

In sequential systems, input data are arranged in the same order as the data in the file in order to efficiently match transactions against the affected records. With random processing, input data is matched with the stored data record by accessing the proper data record, matching it with the input, and then distributing the data record back to its proper location. This procedure of placing the input transactions into their proper pigeon-holes is similar to sorting; the purpose is to put the input transactions into order.

When using sequential file organization with both sequential and random processing capability, the two methods of putting data into order can be compared and the better one selected. Sorting time depends upon the number and size of the records as well as the equipment configuration. Random access time – the time used to distribute records – depends upon the number of records, the size of the data file, and, in some instances, upon the location of the cylinders assigned.

The following example points up some of the considerations, but a comparison of conditions existing in an operating environment is required to support a final choice between the two methods.

Assume that 8,000 items consisting of 80 characters each accumulate for processing. Further assume that the data records occupy an entire module of 1301 disk storage.

Random Processing
Random Access Time
165 ms per item x 8,000 = 1,320 seconds = 22 minutes

Sequential Processing
Magnetic Tape Sorting Time (1410-729 VI)

Sort 8,000 items (80 characters)	6.5 minutes
Access Cylinders Sequentially	0.25
	6.75 minutes (approx.)

Disk Sorting Time
1410-1301 II

Sort 8,000 items (80 characters)	4.00 minutes
Access Cylinders Sequentially	0.25
	4.25 minutes (approx.)

Sorting the data and sequentially accessing each cylinder is three to five times faster than eliminating the sorting and processing randomly; the reverse is true if a quantity of only ten items instead of 8,000 is assumed, or if access time to the data is minimal. Therefore, a calculation should be made for each case before deciding to include or eliminate sorting.

Record Mode versus Full Track Mode

Records can be stored individually with record addresses or grouped into single physical records that comprise a full track of 1301 disk storage. Three considerations enter into the choice: storage utilization, data transfer time, and processing time.

Figure 4 shows the number of records and their lengths that can be stored on a track. Full track mode provides more characters per record than does record mode. If the fullest possible use of character space were the only consideration, full track mode would be superior.

The second consideration is the channel requirement to read and write the record. A complete disk rotation is required to read a full track record from disk storage into the computer. This takes 34 milliseconds. The time to read a single record is proportionately less, depending upon the record length. If ten records are stored on a track the time to read one record is approximately one-tenth of the full rotational time. Writing and checking times are similarly calculated. In addition to the actual data transfer time through the channel, the channel will be occupied while it waits for the proper record to arrive at the read-write head. The average waiting time will be half a revolution or 17 milliseconds for the initial reading of a record. The rotational delay for writing a record is the difference between total process time and the next multiple of 34 milliseconds. When the channel is occupied by reading, writing, or checking data, it is

Number of Records	1301 Full Track Mode	1301 Record Mode	1302 Full Track Mode	1302 Record Mode
1	2800	2800	5850	5850
2	1400	1381	2925	2900
3	933	908	1950	1916
4	700	671	1462	1425
5	560	529	1170	1130
6	466	435	975	933
7	400	367	835	792
8	350	316	731	685
9	311	277	650	605
10	280	245	585	540
11	254	220	531	486
12	233	198	487	441
13	215	180	450	403
14	200	164	417	371
15	186	151	390	343
16	175	139	365	318
17	164	128	344	297
18	155	119	325	277
19	147	111	307	260
20	140	103	292	245
21	133	97	278	230
22	127	91	265	218
23	121	85	254	206
24	116	80	243	195

6 Bit Character Record Size*

*Maximum sizes of uniform length records per track are listed (six character RA's; two character HA2's)

Figure 4. Record Size and Number of Records per Track

not available for operations with other devices. Thus, record mode requires less channel time and is preferable for random processing or low activity sequential processing where channel time considerations are paramount.

The third major consideration is process time. The normal processing sequence with full track mode is to read the record from disk storage into the central processing unit, process the record and write it back to disk storage, and check it. Because of processing delay, the minimum possible time to perform these operations would involve four revolutions of the disk. With most systems, however, there is insufficient time between the passing of the end of the track and the index point at the beginning of the track to allow for initiation of the checking operation. Therefore, an additional rotational delay is incurred. And at the completion of checking, insufficient time is available to initiate a read of the next track and additional rotational delay occurs.

In record mode, the IBM 7631 File Control need not reference the index point to initiate the reading, writing, or checking of the record. Therefore, the total delay time is only that required to readdress the same record. If the record is not at the far end of the track, there is time after the completion of the checking operation to initiate a seek and obtain another record from any other track in the cylinder. This avoids a second rotational delay.

Figure 5 shows a substantial processing time advantage for record mode. This is in addition to the decreased requirement for input/output core storage space and an absence of a requirement to *deblock* the desired record from a group stored full track. The advantage of single record processing holds true even for sequential processing of the track, when the average number of active records per track is less than two.

Figure 5 further suggests that a program should look ahead at the input data and compare it with the track index to learn whether there are more than two items being processed on a track. When this is true, the full track should be read, but when only one item is active on the track, the item should be read in record mode. Conversely, when a file is being scanned for data file maintenance, the full track, with addresses, should always be read and where only one item was active in that particular group, it should be written back separately into the proper place in disk storage.

The process time relationships in Figure 5 are shown graphically for a specific case in Figure 10 (Appendix D).

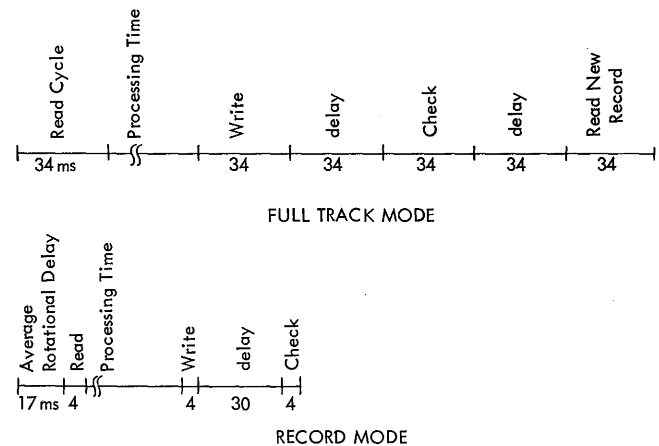


Figure 5. Time Considerations for 1301

Input-Output Storage

Disk storage can be used for the accumulation of input transactions or output items into batches. As input transactions are accumulated, control totals and a total count of the items are maintained. When a batch is large enough to process, the system can be signalled to schedule the operation, or conversely, whenever the system finds time, it can test these counts to determine if a batch is large enough to process.

Output records can be accumulated in the same way and processed when convenient. For example, random transactions may have been processed during the course of a day which call for the generation of output documents such as confirmations of sale, special billings, or similar documents. If these were produced at the time, multiple setups of equipment or the continuous reservation of a magnetic tape drive would be necessary to receive these documents. For convenience in scheduling a printing operation, records can be retained in a section of disk storage until printing is warranted or until another batch that produces similar documents is run. The accumulated document data can then be added to a tape and printed in a combined run.

Program Storage

Disk storage is a good storage medium for programs and subroutines. Obvious advantages include reduction of setup time, elimination of searching time for a program stored on tape, and elimination of handling time when a program tape must be dismounted to make the tape drive available for data input and output. The use of disk-stored programs and subroutines can simplify programming in a number of ways.

Processing Sequential or Random Input Transactions

Programs should be written to capitalize upon the ability to process random transactions economically. Conditions arise under which it may be desirable to have a program process a single input or a very small number of transactions despite the program having been designed to process only large batches of transactions. Special payroll checks or vacation checks may require all the processing of the regular payroll procedure. If the payroll system is designed only for large batches, processing of individual checks may be so inconvenient as to necessitate manual handling with subsequent adjustment of the records. If substantial editing of the transaction is accomplished in the sorting routine, it may not even be possible to run individual items or batches that are too small to sort.

If the processing programs are written to be independent of the method of data matching and retrieval, then single items or large batches can be processed by the same program. In this way, random transactions as well as large batches can be accommodated without any need to write and maintain two separate programs.

Another benefit of such an approach is that the sorting versus distribution calculations previously discussed can be performed to choose the best procedure. The addition of the concepts of program storage and input buffering to the concept of sorting versus distribution points to programming that will permit the system to choose automatically an efficient course of action.

Balancing Partially Processed Data Files

When processing transactions against a data file on cards or magnetic tape, every file record, inactive as well as active, is read and a total balance is established based upon the details of each record. The whole process is assumed to be in balance if the total of original balances, plus or minus the transactions, equals the total of the resulting balance. When random transactions or batches are processed against records in disk storage, only the active records are consulted, and control procedure modification is required. Since the inactive records are not read, the balancing procedure must depend upon the assumption that they are correct. This assumption is proven by trial balancing all accounts on some cyclic basis that is frequent enough to enable corrective action.

The means for detecting record errors are provided by establishing balance fields in addition to detailed item fields. For accounts receivable records, a total amount due field is established that is the *cross foot* total of the gross amounts of the individual invoice items.

Then all processing of such records would include cross-footing the record before and after processing to assure that the record was in balance and that it remains in a balanced condition. A before total of all balances of the affected records is reconciled with the changes and the total of the after balances. When this is done, the total of the changes can be posted to the total control records which will then reflect the correct total of all record balances. An example follows:

Accounts Before Processing:

	ITEM	ITEM	BALANCE
Account A	50.00	00.00	50.00
Account D	75.00	75.00	150.00
Total Old Balance of All Accounts			10,000.00

Two Cash Receipts to be Processed:

Transaction A for 40.00
Transaction D for 75.00.

Accounts After Processing:

	ITEM	ITEM	BALANCE
Account A	10.00	00.00	10.00
Account D	00.00	75.00	75.00
Total Balance of Affected Accounts "Before"			200.00
Total Balance of Affected Accounts "After"			85.00
Total Transactions			115.00

Since the old balance minus the new balance equals $115, the amount posted, the procedure checks, and the new total control balance of all accounts is reduced from $10,000.00 to $9,885.00.

Such a balancing procedure is fundamentally the same as that used in manual bookkeeping systems or in tape systems where the total main file is split into daily cycles and a total control balance covers all cycles.

The sequential organization of data lends itself particularly well to this kind of balancing because subledger controls can be instituted to cover portions of the total file. These controls should be reconciled to the total control before and after processing or at periodic intervals during processing. Provisions should be made for retaining the original control figures during the course of any interval of processing as well as the new figure that reflects the changes. These are then reconciled at any desired balancing point. The general philosophy used is that if the changes balance in detail they can be used in the subledger totals. If the subledger totals balance similarly, the change can be posted to the grand total.

If errors occur, only the subledgers that are in doubt need be balanced to the details supporting them, and searches of the entire file are substantially eliminated.

An inactive account that is out of balance will go undetected. But the procedure outlined guarantees that the last time it was legitimately processed, the record was correct, and the next time it is processed or trial balanced, the error will be detected. Good bookkeeping practice requires nothing more.

Audit Trail on Output Documents

Random processing can make the trail back through output documents somewhat obscure. There is no requirement that every account be reflected in every document, nor is there a requirement that the items on output listings be in any sequence. Under such circumstances the job of searching back through historical documents may be difficult if not impossible.

It is recommended, therefore, that a record of the most recent output document that has a reference to a given account be maintained within the disk stored record of that account. For added convenience the line number of that reference can also be included. When a new reference is made the previous record is printed out with the result that the most recent reference is carried in disk storage, the next most recent is shown on the referenced report, and that report shows the previous report reference, etc. (See Figure 6.)

Program Signature

The previous example shows that any program that produced an output document can be traced in its relationship to other programs that also produced output documents. The sequence in which several different programs operated upon a particular data record can be ascertained quickly.

DISK RECORD:

Account Number	Name	Date	Run	Line Number
XXXX	AAAAA	0731	Cash	1097

PRINTED REPORT:

Cash Journal July 31

Line Number	Account Number	Name	Previous Reference			Paid	Balance
			Date	Run	Line #		
1096							
1097	XXXX	AAAAA	0625	JnEntry	109	XXXX	XXXX

(On the Journal Entries report for June 25th a similar reference leads to the previous report.)

Figure 6. Report Line Number Audit Trail Example

Such a relationship is not so easy to establish with programs which do not produce an output document, but when difficulties are suspected some trail information is invaluable. An additional field can be used in a data record to identify the last program that acted upon the data record. A sequence number can be used to identify multiple references to the same record. In tracing unusual conditions, this field will eliminate questions about how a record achieved its status if the most recent program *signed* the record with a date, number, and unique program designation. This information should be passed on to the audit records that are created from the transactions even though it is not planned to print the records. This program signature procedure can provide a single, common trail for all programs, similar to that discussed for printed reports.

Unintentional Data Reference

While the methods just described give assistance in tracing unintentional data referencing, it is also true that the organization scheme itself provides considerable protection of the data. If the system uses a data directory index, the addresses of data in disk storage can be located only by referencing the data file by name. The directory thus performs a fundamental role in limiting the data access to those programs that have

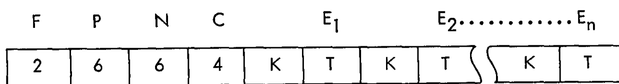
specifically requested records from that particular data file.

Detailed information on accounting controls appears in the IBM manual, *In-Line Electronic Accounting, Internal Control and Audit Trail,* Form F20-2019 (prepared by Price Waterhouse at IBM's request). This manual draws upon experience with the IBM 305 RAMAC® Data Processing System.

Appendix A. Index Formats for 1410 and 7010 Systems

The fields below are recommended for 1301 and 1302 data indexes to facilitate random and sequential data processing, as well as sequential scanning of all records in a data file.

Track Index Format



F – This two character flag refers to data and contains D, bl.

P – Address of prime index (AMTTTT).

N – Address of *next* track index (AMTTTT) or, END left justified for the last track index of the data file.

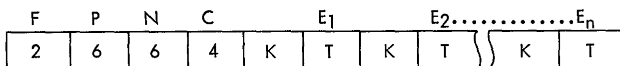
C – Count of entries (E-fields) in this track index.

E – This variable length field provides index information for data tracks. One E-field is created for each data track in the primary cylinder plus one E-field for each overflow record associated with this track index. The E-field has two portions, K and T:

K – A variable length field for the alphanumeric key of the record with the highest key in the primary cylinder or of the overflow record. Blanks are permitted in the key.

T – The track address (AMTTTT) for the track associated with K. When K refers to an overflow area, the tens position of this T-field has a zone (B-bit).

Prime and Intermediate Index Format



F – This flag field contains I, bl, to refer to a lower level of index.

P – For a prime index, this field contains PRIME left justified. For an intermediate index, the field contains the address of the prime index (AMTTTT).

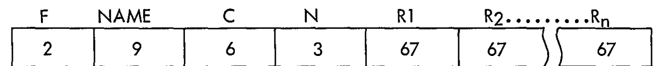
N – This *name* field carries the literal PRIME or INTER.

C – Count of entries (E-fields) in this index.

E – This variable length field serves a function similar to that of the E-fields in the track index. In an index of data cylinders, the key portion of the field (K) contains the last (highest) key in the track index and the track portion of the field (T) contains the track address of the track index for the data. An index to a data cylinder is complete when the data file is completely loaded or when a maximum length is attained. In the former case, the index would be a prime index; in the latter case, the index would be an intermediate index. A hierarchy of indexes can be created to meet the needs of the data file with regard to system limitations.

Data Directory Index Format

This index is created independently of the data loading program to provide control information regarding the data files in disk storage. It has this format:



F – This two character flag field is for identification.

NAME – Contains the literal DIR bl INDEX.

C – Track address of overflow portion of this index or END.

N – Number of R-fields in this index.

R – An R-field is created in this index for each data file in disk storage (Figure 7).

Code	Name	Address	Key Length	Key Position	Record Length	Mode	Full Track Length	Track Address Length	First Track for Overflow	Last Track for Overflow	Data of Change and Signature	Chain Address
3	10	6	2	3	4	1	4	2	6	15	8	5

Figure 7. Format of R-Field

Code – Date file code.

Name – Data file identification.

Address – Address of the prime index of the data file.

Key Length – The length of the key fields for the data file in character positions.

Key Position – Units position of the key for the low-order position of the records in core storage.

Record Length – Length of the data record in characters.

Mode – Full track or record mode; 6- or 8-bit.

Full Track Length – Length of a full track of the data file under format control. Lenth includes record addresses.

Track Address Length – This field has 00 for normal six character addresses (AMTTTT). When indexes for data files are stored in record mode, additional track address characters can be used to provide record addresses. Such record mode storage of indexes is convenient for data files converted from 1301 to 1302 disk storage. When track addresses are longer than six characters in length, this field should be incremented by one for each extra character.

First Track for Overflow – Track address (AMTTTT) for the beginning of the data overflow area in disk storage. The address is updated as overflow tracks are filled. Therefore, this field always indicates the address of the next overflow track. As this field is updated, its contents are compared with the contents of the next field, Last Track for Overflow. A message is typed when the remaining area for overflow becomes minimal.

Last Track for Overflow – Track address (AMTTTT) for the end of data overflow area in disk storage.

Date of Change and Signature – The first five characters of this field indicate the date that the reference was last changed. The last three characters are the initials of the program making the change.

Chain Address – This field is reserved for future use in track address (AMTTTT) chaining to additional information.

When only two indexes are required the prime index can be read into core storage. The area required can be calculated in this manner:

Number of characters = $W + N(K + 6)$
- W = Length of prime index header.
- N = Number of cylinders of data.
- K = Length of key.

The key portion of the last entry in any index should be all 9s in order to terminate a search of keys.

Appendix B. Additions to Track Index

Figure 8 shows the number of track references that can be packed into a 1301 track index before the track index overflows. No allowance has been made for a track index header. The cylinder carrying the track index is formatted in record mode.

The following formula was used to compute the numbers of references:

$$N = \frac{L+6}{K+6} \times R$$

where: N = Number of references per track.
L = Number of characters per record plus six characters for record address.
K = Number of characters per key plus six characters for track address in index.
R = Number of records per track.
6 = Characters per record address.
6 = Track address.

Maximum No. of Uniform Length Records per Data Track	Effective Area for Index Storage	Number of References per Index Track by Key Size							
		5 Char Key	6 Char Key	7 Char Key	8 Char Key	9 Char Key	10 Char Key	11 Char Key	12 Char Key
1	2806	255	233	215	200	187	175	165	155
2	2774	252	231	213	198	184	173	163	154
3	2742	249	228	210	195	182	171	161	152
4	2710	246	225	208	193	180	169	159	150
5	2678	243	223	206	191	178	167	157	148
6	2646	240	220	203	189	176	165	155	146
7	2614	237	217	201	187	174	163	153	145
8	2582	234	214	198	184	172	161	151	143
9	2550	231	212	196	182	170	159	149	141
10	2518	228	209	193	179	167	157	148	139

Figure 8. Maximum Number of References Per 1301 Index Track by Key Size When Formatted in Record Mode

Appendix C. File Growth and the Track Index

The percentages of allowable 1301 data file growth shown in Figure 9 are based upon the following factors:

Percentage of allowable record growth per

$$\text{cylinder} = \frac{N - 39}{R}$$

where N = Number of overflow index references per track.
R = Number of records per cylinder (of 39 tracks).

Maximum No. of Records per Track	No. of Records per 39 Track Cylinder	Percentage Allowable Growth in Cylinder Records from Track Index Capacity 5 Char Key	6 Char Key	7 Char Key	8 Char Key	9 Char Key	10 Char Key	11 Char Key	12 Char Key
1	39	554	497	451	413	379	348	323	297
2	78	273	246	223	203	185	171	158	147
3	117	176	158	144	133	122	112	104	96
4	156	132	121	109	100	91	84	78	72
5	195	104	94	85	77	71	65	60	55
6	234	85	77	70	64	58	53	49	46
7	273	72	65	59	54	49	45	41	38
8	312	62	56	50	46	42	39	35	33
9	351	54	49	44	40	37	34	31	29
10	390	48	43	39	35	32	30	27	25

Figure 9. Percentage of File Growth Before 1301 Track Index Overflow When the Cylinder is Formatted in Record Mode

Appendix D. A Comparison of Sequentially Processed Full Track Mode and Record Mode Operations

The relationships in Figure 10 show that record mode operations are favorable for low ranges of activity. For higher ranges, full track mode is more attractive. The two modes of operations are comparable at an activity ratio of approximately 40 percent. Figure 10 refers to 1301 operations.

The assumptions for Figure 10 are as follows:

1. Record size — 367 characters.
2. File size — 70,000 records.
3. Processing time for one record — 17 ms.
4. The data file is stored in sequential order.
5. The input transactions are pre-sorted in the same sequence as the data file.
6. Processing time includes:

   a. Read, process, write, and write check the data record; and

   b. Rotational delay periods associated with the operations.

The processing time elements have the following relationships:

Full Track Mode:

READ	PROCESS	WRITE	DELAY	CHECK	DELAY	
34	34	34	34	34	34	= 204 ms

Single Record Mode:

DELAY	READ	PROCESS	WRITE	DELAY	CHECK	
17	4	30	4	30	4	= 89 ms

Index Retrieval — Full Track or Record Mode:

DELAY	READ	PROCESS	
17	34	34	= 85 ms

7. For both modes of operations, the procedure is as follows: The track index for a cylinder of data is referenced once for all transactions affecting records contained in that cylinder. Processing proceeds sequentially from one cylinder to the next. No overflow records are assumed.

The calculation considerations for Figure 10 are:

Record size — 367 characters
Number of records per file — 70,000 records
Number of tracks — 10,000
Process time per record = 17 ms
Cylinder access time = 14,600 ms for 250 cylinders
Time to read indexes = 85 ms per track
Index retrieval time including seek time = 14,600 +85 (250) ms = 35.85 seconds

Full Track

N = Number of active records
Process = (34)6 = 204 ms per record

N = 250 T = Process Time = 14,600 +85(250) +0.204(250) seconds
= 86.85 seconds
= 1.45 minutes

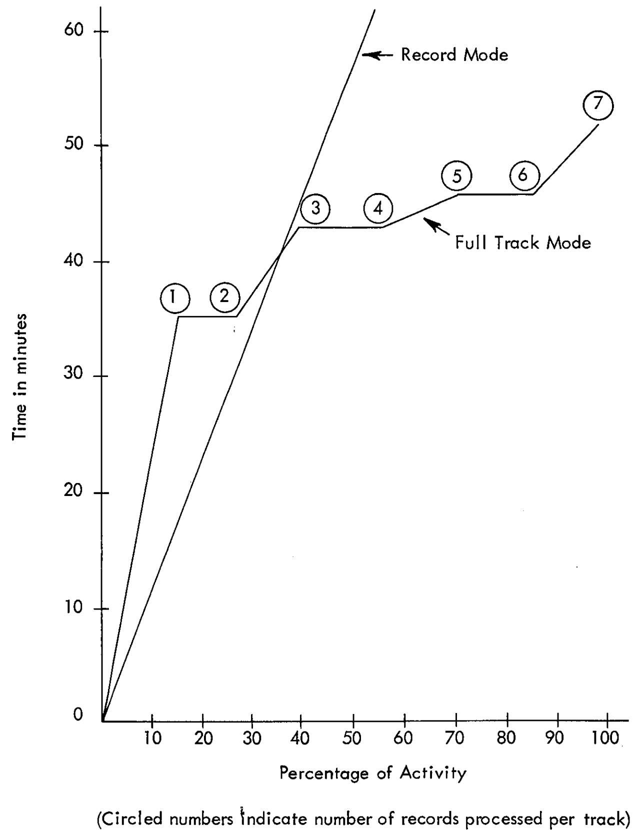


Figure 10. Activity Considerations with Record Mode and Full Track Mode Operations

N	T
N = 1,000	T = (14.6 +21.25) +0.204(1,000) seconds = 4 minutes
N = 5,000	T = 35.85 + 0.204(5,000) seconds = 17.6 minutes
N = 10,000	T = 35.85 +0.204(10,000) seconds = 34.6 minutes
N = 20,000	T = 35.85 +0.204(10,000) seconds = 34.6 minutes
N = 30,000	T = 35.85 +0.238(10,000) seconds = 40.2 minutes
N = 40,000	T = 35.85 +0.238(10,000) seconds = 40.2 minutes
N = 50,000	T = 35.85 +0.272(10,000) seconds = 45.9 minutes
N = 60,000	T = 35.85 +0.272(10,000) seconds = 45.9 minutes
N = 70,000	T = 35.85 +0.306(10,000) seconds = 51.6 minutes

Single Record

Process = 17 +4 +30 +4 +30 +4 = 89 ms per record

N = 250 T = 35.85 +0.089(250) seconds
= 0.97 minutes

N = 1,000 T = 35.85 +0.089(1,000) seconds
= 2.06 minutes

N = 5,000 T = 35.85 +0.089(5,000) seconds
= 8.01 minutes

N = 10,000 T = 35.85 +0.089(10,000) seconds
= 15.43 minutes

N = 20,000 T = 35.85 +0.089(20,000) seconds
= 30.26 minutes

Appendix E. Estimating Access Motion Time for IBM 1301 Disk Storage

Access motion time is the time required for access arm motion to get from the track last used to the next track. There are three levels of motion time: cylinder, group, and zone.

No motion time occurs between tracks in the same cylinder. Motion time between tracks in cylinders in the same group is 50 milliseconds, between tracks in cylinders in different groups in the same zone motion time is 120 milliseconds, and between tracks in different zones it is 180 milliseconds. Thus, motion time varies between 0 and 180 milliseconds, depending on the motion involved. Six examples of various access situations are shown below.

Example 1: Average motion time within a five cylinder group.

Five cylinders can be accessed from any of the five cylinders in the group. The starting cylinder requires 0 milliseconds motion time and the other four each require 50 milliseconds motion time. Assuming that the starting and ending cylinders are chosen at random, the average time is:

$1/5 \times 0 + 4/5 \times 50 = 0 + 40$ ms.

A similar relationship exists for the tracks. Of 200 tracks (40×5) in the group, 40 tracks have zero motion time and 160 have 50 milliseconds motion time. The average is $40/200 \times 0 + 160/200 \times 50 = 40$ milliseconds, the same result obtained with the cylinder calculation.

Example 2: Average motion time within a ten cylinder group.

Here the chances are nine in ten of going to a different cylinder against one in ten of accessing a track on the same cylinder. The average time is:

$1/10 \times 0 + 9/10 \times 50 = 0 + 45$ ms.

Example 3: Twenty cylinders in two groups of ten within the same zone.

Half the tracks are in the starting group and the rest are in the other group. Within the same group the average time is 45 milliseconds (Example 2); between groups it is 120 milliseconds. The average time is:

$1/2 \times 45 + 1/2 \times 120 = 22.5 + 60 = 82.5$ ms.

Example 4: Twenty-five cylinders in two groups of ten and one group of five, all within the same zone.

This example illustrates how to take into account different sized starting groups.

Case A: If the starting track is in a ten cylinder group. Ten of the cylinders that can be accessed are in the same group and have an average time of 45 milliseconds. Fifteen are in one of the other groups with an average time of 120 milliseconds. The average motion time for this situation is:

$10/25 \times 45 + 15/25 \times 120 = 18 + 72 = 90$ ms.

Case B: If the starting track is in the five cylinder group. Five of the cylinders are in the same group with an average access time of 40 milliseconds (Example 1). Twenty are in the other groups having 120 milliseconds access time. The average time is:

$5/25 \times 40 + 20/25 \times 120 = 8 + 96 = 104$ ms.

Cases A and B must now be combined in proper proportion to get the overall average. Since there are 20 of the 25 cylinders in the groups of ten, on the average 20 out of 25 starts will be Case A. The other five will be Case B. Therefore, overall average access time for this example is:

$20/25 \times 90 + 5/25 \times 104 = 72 + 20.8 = 92.8$ ms.

The next example is of the same type.

Example 5: A zone of 50 cylinders. Two groups of ten, one of five, two more of ten, and one of five.

Case A: With the starting track in any one of the ten cylinder groups, ten of the cylinders that can be accessed are in the same group with an average time of 45 milliseconds and 40 are in other groups with 120 milliseconds access time. The average time is:

$10/50 \times 45 + 40/50 \times 120 = 9 + 96 = 105$ ms.

Case B: With the starting track in one of the groups of five cylinders, five of the 50 cylinders can be accessed in an average of 50 milliseconds (Example 1) and the remaining 45 require 120 milliseconds. The average time is:

$5/50 \times 40 + 45/50 \times 120 = 4 + 108 = 112$ ms.

In this example, ten of the 50 starting tracks are in groups of five (two groups) and the other 40 are in groups of ten (four groups). The overall average is:

$10/50 \times 112 + 40/50 \times 105 = 22.4 + 84 = 106.4$ ms.

Example 6: A file of 250 cylinders. Five zones of 50 cylinders each.

Fifty of the cylinders are in the same zone with an average access time of 106.4 milliseconds (Example 5). The other 200 are in different zones requiring 180 milliseconds of access time. The average time is:

$50/250 \times 106.4 + 200/250 \times 180 = 21.3 + 144$
$= 165.3$ ms.

It is necessary to consider the proportion of starting tracks only when the average time to all other tracks in the data file depends on the location of the starting track. While this is generally the case, by using the group and zone averages calculated in examples 1, 2, and 5 which take this effect into account, the computations can be simplified as in example 5. The calculations give *average* times; the motions can take times varying from 0 to the maximum for the boundary crossed in such a manner that they average out as indicated by the calculations.

Printed in U.S.A. A22-6784

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, New York