This paper deals with the structure and use of indexes that facilitate the retrieval and storage of records based on a specific value, value range, or value sequence of a given field of a record within one or more data sets. Specifically, it examines general index structures, maintenance, index entry compression, and complex indexes as considered in the basic design of VSAM (Virtual Storage Access Method). Under complex indexes, indirect secondary indexes and indexes to multiple data sets are considered.

Indexing design considerations

by R. E. Wagner

Indexes in the computer environment are functionally equivalent to indexes in other environments. They are organized sets of information used to facilitate a fast search of a set of entities. For example, one of the most common indexes is the one at the end of many books. This index allows a given word or phrase in the book to be located fairly rapidly. The words in the book index represent the value or key on which the index is built, and the page number is a pointer to where that word is used.

The base data on which an index is built can vary from such unformatted material as the text of a book to such formatted material as a payroll record with fixed length fields. Generally, this paper deals primarily with a formatted base composed of a set of similar records such as found in a payroll or inventory data set. Some of the techniques, of course, do apply to indexes in general.

Although indexes have been used in computers from the beginning, the major use coincided with the availability of large, random-access storage such as the IBM 1301 device. With the availability of this storage, it was more practical to handle only the data required by a given process. This was not possible with magnetic tape because of the sequential nature of the device and of the lack of an update-in-place feature. With the ability to access small segments of the data, a mechanism was required to locate more quickly the data of interest. One of the methods that emerged is indexing. For example, with an index built on part numbers it was possible to locate a given inventory record much faster than it was in the sequential tape environment and still provide a way to process the data sequentially in part-number order.

The purpose of this paper is to present a general discussion of indexing as considered in the basic design of VSAM (Virtual Storage Access Method), and as such it examines various types of indexes and their uses in terms of general structure, maintenance, entry structure, and complexity. The section on general index structure deals with the basic parts of an index and with multiple-level, dense, and nondense indexes. The section on maintenance covers the updating of an index and basic insertion strategies. The section on entry structure considers ways in which the entry can be represented in a minimum amount of space. Under the topic of complex indexes, indirect, secondary, and multiple data-set indexes are examined.

General index structure

The basic element of an index is an *index entry*. It is composed of a single value and a pointer to a record that contains that value. To facilitate searching and maintenance, the entries are placed in the index in ascending value order. If the index is sufficiently large, it may be subdivided into index records that contain multiple entries.

Consider an index containing 40,000 entries collected into 400 records, each of which contain 100 entries. To locate a given value in this index using a binary search technique would require the examination of 16 entries in eight records. This represents a high search overhead if the examination requires an input operation for each record. To minimize this, an index can be built to index the original index. This new, second-level index contains one entry for each record of the original index. The entry contains the highest value in the original index record. For the above example, the second-level index would contain 400 entries divided into four records.

Another index, or third-level index, could be built on the second level. It would have only one record containing four entries. With this new index, only three records need to be examined to locate a given value.

index levels

The original index, the index built on it, etc. are referred to as levels of the index. The lowest level, or the original index, is named the *sequence set* since it defines an ordering of the data to which it points. The remaining levels are referred to as the *index set*.

To locate a record with a given value via an index with multiple levels, a specific set of index records is searched. This set is referred to as the search path and the records as the nodes in the path. In the above-defined index structure, the number of nodes

Figure 1 Three-level balanced tree index

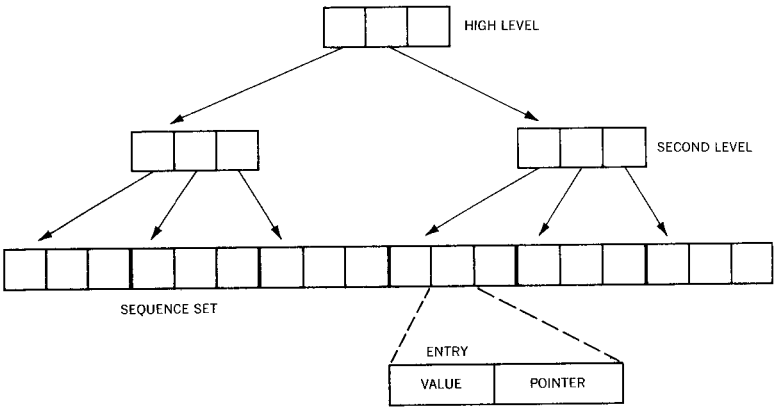
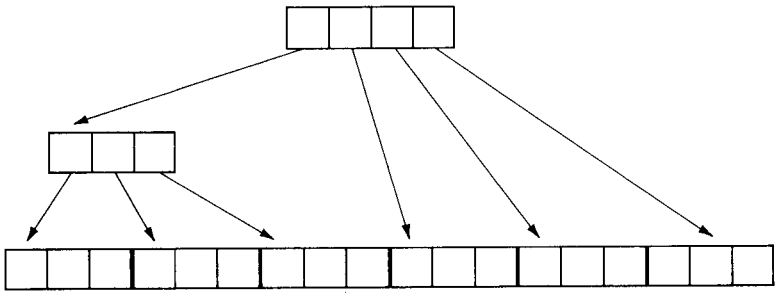


Figure 2 Unbalanced index



examined is fixed and is independent of the specific search value. This type of an index is referred to as a balanced tree index. If the number of nodes examined varies based on the specific search value, the index is said to be unbalanced. An index can be built originally as an unbalanced index to optimize the retrieval of certain records or may become unbalanced as a result of insertions and deletions of values. Figures 1 and 2 illustrate a balanced and unbalanced index, respectively.

In the index set, an entry pointed to a set of values contained within the lower-level index record. This was made possible by the ordering of the entries in the lower levels of the index. If the data itself is ordered on the indexed value, a sequence set entry can point to a set of records. In this case, the sequence set entry contains the value for the highest value in the data record set and points to the first value of the set. If the sequence set contains an entry for each data record, the index is said to be dense. If a sequence set entry only resolves to a set of two or more

records, it is said to be a nondense index. A given set of data records (data set) generally only has one nondense index since only one contiguous physical ordering of the data is possible.

The nondense index offers a number of design trade-offs. Since there are fewer entries in the index in general, it requires less space, less maintenance, and less input-output transfers. But to determine if a given key value exists in one of the records, it is necessary to search the data records themselves. Therefore, if the object of the search was to determine if a given value was or was not present, the nondense index may require more processing time than a dense index.

Maintenance

Whenever the data that is pointed to by a set of indexes is changed by update, insertion, or deletion, it may be necessary to alter the index itself. In the case of update, it involves a check to see if the indexed value itself is changed and results in an index update per index which has a value change. Deletion results in the removal of entries that point to the deleted entities. Insertion results in the adding of values. The maintenance of the index itself is secondary to the maintenance of the base data in terms of insertion.

insertion

There are two basic insertion strategies, in-place and out-of-place, and the one used is somewhat dependent on the indexes involved. An index that has its entries ordered in the same sequence as the data to which it resolves is a prime index. If the prime index structure takes advantage of this sequence, that is, it is nondense, an in-place insertion strategy is required to maintain the nondense characteristic of the index.

The simplest insertion strategy is the out-of-place insertion. This approach in its most basic form requires a single overflow area, generally at the end of the data set, into which data insertions are placed in a sequential manner. It may, however, make use of space within the data set that was made available by record deletions. This strategy works well with dense indexes, but sequential retrieval performance deteriorates as the number of insertions increase. ISAM (Index Sequential Access Method) uses a form of out-of-place insertion.[1]

The in-place insertion strategy is more complex and requires multiple insertion areas. This type of insertion generally results in data shifting and either chaining of records or the splitting of record sets. The multiple insertion areas must be local to maintain a high level of sequential performance.

There are two major drawbacks associated with in-place insertion: integrity and secondary index maintenance. The integrity problem is associated with the shifting of data, block or set splitting, and chaining in that the insertion of a given record may affect other records. This problem can be minimized by writing the updated records in a proper sequence and by minimizing the number of chains. For example, the number of chains in ISAM[1] could be reduced by increasing the number of overflow records in a physical block. Currently there is one overflow record in the physical block. In general, each overflow physical block requires a chain or pointer to the next block in the chain.

A secondary index has its entries ordered in a sequence that is different from the data to which it points. As a result, a secondary index must be dense and must be updated whenever a data record is shifted. This updating can be minimized if secondary indexes do not point directly to the data. For example, they could resolve to a unique prime key.

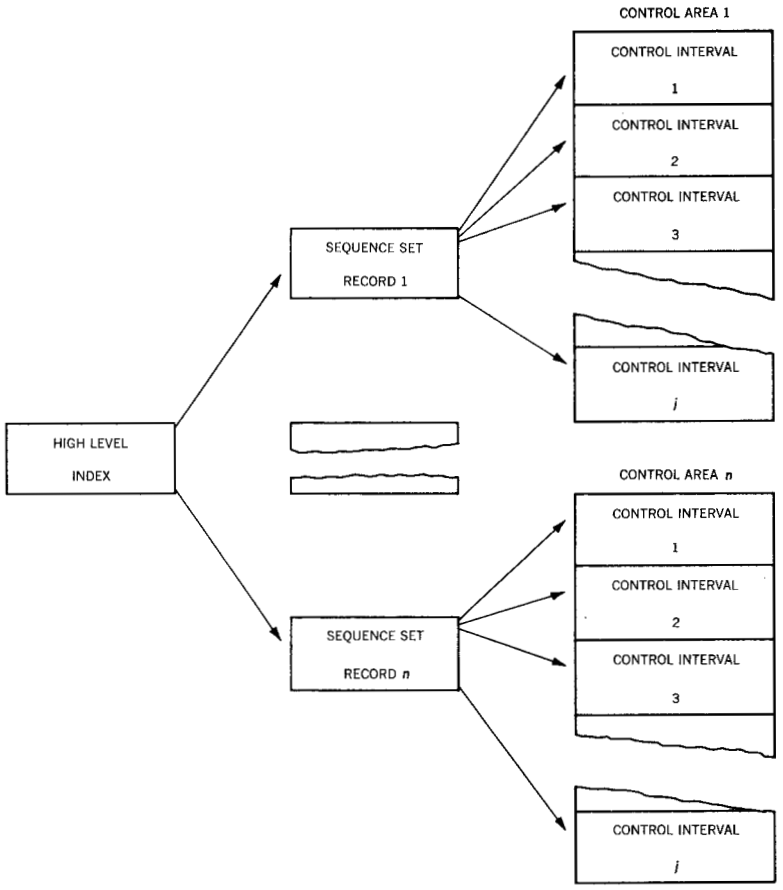
VSAM is an example of combined data and index organization that was designed for in-place inserts. The major new elements of this organization are control intervals and control areas. The control interval is a set of contiguous records in sequence order by prime key and is the set of records pointed to by a single sequence set entry. The control area is space that contains a set of contiguous control intervals in terms of the prime key. This ordering within the control area is not necessarily physical. The ordering is controlled by a single sequence set record. Figure 3 illustrates the VSAM indexed structure.[2]

To facilitiate insertion, the control interval can contain a variable number of records and free space. If a control interval contains no entries, it is available as a spare control interval within the control area. A generalization of the VSAM insertion rules is as follows:

1. Locate the control interval into which the record is to be inserted.
2. If there is space within the control interval, merge the record to be inserted into the control interval. The insertion has been completed, and no index update is required. (This assumes no multiple indexes.)
3. If there is sufficient space within the control interval, check to see if there is a spare control interval in this control area.
4. If there is a spare control interval do the following:

- Map the first half of the records from the target control interval into the spare control interval along with the new record if it fits with this sequence, and write the new control interval.

Figure 3 VSAM indexed structure



- Update the sequence set record for the control area by inserting an entry for the activated control interval, and write the updated sequence set record.
- Map the second half of the records of the target control interval into the first half of the target control interval. Merge the record to be inserted if it fits into this sequence. Write the updated target control interval.

5. If there is no spare control interval do the following:

- Allocate a new control area and a new sequence set record.
- Map the first half of the target control area intervals into the new area and build the sequence set record for the new area. Write the new control area and sequence set record.
- Insert an entry into the high-level index for the new sequence set record.
- Update the target sequence set record by making the first half of the control intervals spare control intervals and write the record.

- The insertion can now be done via Rule 4 because half of the control intervals in the original target and new control area are spares.

Entry structure and compression

The index entry is composed of a specific value and a pointer and is the basic building block of the index. Since an index generally is too large to reside in primary storage, performance can be optimized by minimizing the size of the index. The previous section defined a nondense index that accomplished this by minimizing the number of entries, but this approach is restricted to prime indexes. In this section, reduction of the size of indexes is examined in terms of the entry itself. This can be done by making either the pointer or entry smaller. Three approaches are considered. The first deals with pointer compression, the second and third with key compression.

In contrast to the savings in terms of I/O transfers and space, the compression techniques in general require additional CPU time. Also, the key compression approaches require that the data itself must be examined to determine if a given key value does or does not exist.

pointer compression

The first method, concerning pointer compression, only applies to nondense prime indexes. Since the entry in a sequence set points to a set of contiguous records, the pointers only have to resolve to a set rather than to individual records. If the record sets pointed to by a given sequence set record are fully contained within a single area and the record set size is fixed, the contents of the area can be implied from the sequence set record rather than the individual entries. Under these conditions, the pointer within an entry can be reduced to n bits where the maximum number of record sets that can be addressed by a single sequence set record is less than two to the nth. The resulting record format is as follows:

	ENTRY 1		ENTRY 2		
BASE POINTER	VALUE	RRS	VALUE	RRS	

To calculate the address of the record set, the RRS (Relative Record Set) is multiplied by the RRS size and is added to the base pointer.

key compression

The second and third methods deal with the values within the entries for key compression. Thus far an entry contained a complete value. This is not necessary if the value can be repre-

Figure 4 Binary search index across 15 values

Entry	*Bit position*	*0 Pointer*	*1 Pointer*	*Data values*
1	1	E-2	E-12	1. 0010 0010
2	2	E-3	E-8	2. 0010 1000
3	4	E-4	E-6	3. 0010 1011
4	5	D-1	E-5	4. 0011 0010
5	7	D-2	D-3	5. 0011 0100
6	5	E-7	D-6	6. 0011 1010
7	6	D-4	D-5	7. 0100 1000
8	4	E-9	E-10	8. 0100 1010
9	7	D-7	D-8	9. 0101 0011
10	5	D-9	E-11	10. 0101 1100
11	8	D-10	D-11	11. 0101 1101
12	2	E-13	D-15	12. 1001 0000
13	3	E-14	D-14	13. 1001 0101
14	6	D-12	D-13	14. 1010 0101
				15. 1101 0110

Figure 5 Search paths for the index illustrated in Figure 6

	Data value	*Search entries in order*	*Path length*
1.	0010 0010	E1, E2, E3, E4	4
2.	0010 1000	E1, E2, E3, E4, E5	5
3.	0010 1011	E1, E2, E3, E4, E5	5
4.	0010 0010	E1, E2, E3, E6, E7	5
5.	0011 0100	E1, E2, E3, E6, E7	5
6.	0011 1010	E1, E2, E3, E6	4
7.	0100 1000	E1, E2, E8, E9	4
8.	0100 1010	E1, E2, E8, E9	4
9.	0101 0011	E1, E2, E8, E10	4
10.	0101 1100	E1, E2, E8, E10, E11	5
11.	0101 1101	E1, E2, E8, E10, E11	5
12.	1001 0000	E1, E12, E13, E14	5
13.	1001 0101	E1, E12, E13, E14	5
14.	1010 0101	E1, E12, E13	3
15.	1101 0110	E1, E12	2
		Average path length	4 1/3

sented in a path to the degree necessary to locate the data. Method two deals with a binary compression technique and method three with a character compression technique.

binary compression

An approach using binary compression (binary trees) was developed by L. J. Woodrum. The basic concept is as follows. Given two values, they can be placed into two slots based on the value of the first bit (starting from the left) in which the bit value differs. For example, the values "1010" and "1001" can be placed into two slots based on the third bit in each. The index entry for these values then could be represented in the following manner.

Figure 6 Binary tree index with a sequential list

Entry	*Bit position*	*0 Pointer*	*1 Pointer*
E-1	1	E-2	E-12
E-2	2	E-3	E-8
E-3	4	E-4	E-6
E-4	5	L-1	E-5
E-5	7	L-2	L-3
E-6	5	E-7	L-6
E-7	6	L-4	L-5
E-8	4	E-9	E-10
E-9	7	L-7	L-8
E-10	5	L-9	E-11
E-11	8	L-10	L-11
E-12	2	E-13	L-15
E-13	3	E-14	L-14
E-14	6	L-12	L-13

List

L-1	L-2	L-3	L-4	L-5	L-6	L-7	L-8	L-9	L-10	L-11	L-12	L-13	L-14	L-15
D1	D2	D3	D4	D5	D6	D7	D8	D9	D10	D11	D12	D13	D14	D15

Bit Position	3
Zero Entry	Address of "1001" record
One Entry	Address of "1010" record

If a new value "0111" was added to the index, the resulting two-level index can be built:

Level 1

Bit Position	1
Zero Entry	Address of "0111" record
One Entry	Address of level 2 entry

Level 2

Bit Position	3
Zero Entry	Address of "1001" record
One Entry	Address of "1010" record

Figures 4 and 5 illustrate this type of an index and its search paths. The notation $E - n$ refers to entry n and $D - n$ to data value n.

Although the binary tree approach provides a satisfactory means of direct retrieval of a specific value, it presents some problems in terms of sequential value ordered retrieval because the pointers to the data are not in sequence order within the index. One method of minimizing this shortcoming is to make the entry pointer offsets into a list of data pointers ordered by key value. The list of pointers could then be used for sequential processing and the binary tree for direct processing. Figure 6 illustrates this modification to the index illustrated in Figure 5.

Figure 7 Full key compression

Key	*P*	*N*	*F*	*L*	*Value*
(low value)	–	–	–	–	–
AEGER	1	2	0	2	AE
ALESS	2	2	1	1	L
ANNET	2	2	1	1	N
ARENA	2	1	1	0	
BAKEN	1	3	0	3	BAK
BANGS	3	3	2	1	N
BARBA	3	4	2	2	RB
BARLO	4	4	3	1	L
BARNE	4	4	3	1	N
BARTH	4	3	3	0	
BATES	3	2	2	0	
BEATY	2	4	1	3	EAT
BEAUD	4	3	3	0	
BEHEN	3	3	2	1	H
BENDE	3	3	2	1	N
BERBE	3	1	2	0	
(high value)					

character compression

Character compression was developed by H. K. Chang, W. A. Clark, C. T. Davies, K. A. Salmond, and T. S. Stafford.[3-5] It eliminates characters from a key based on the actual characters that identify it from other keys. For example, to tell the difference between "apples" and "application" only the fifth letter is required. There are two types of character compression, front and rear. The front compression of "apples" and "application" would result in "es" and "ication". The rear compression is "apple" and "appli". These two forms of compression can be combined with the resultant values "e" and "i". Of course, the values "e" and "i" by themselves are not sufficient, but it illustrates the general concept.

Figure 8 Full key compression index

FL/Value/Pointer	
02/AE/PTR	AEGER
11/L/PTR	ALESS
11/N/PTR	ANNET
10//PTR	ARENA
03/BAK/PTR	BAKEN
21/N/PTR	BANGS
22/RB/PTR	BARBA
31/L/PTR	BARLO
31/N/PTR	BARNE
30//PTR	BARTH
20//PTR	BATES
13/EAT/PTR	BEATY
30//PTR	BEAUD
21/H/PTR	BEHEN
21/N/PTR	BENDE
20//PTR	BERBE

To build an index using this concept, it is necessary to expand it to allow a given compressed entry to identify the given entry from all previous and all following entries. Consider the three values "annex", "apples", and "application". The difference between "annex" and "apples" is a "p" in the second position. The difference between "apples" and "application" is an "e" in the fifth position. The two differences can be combined to form the value "pple," which has one character front-compressed and has a length of four. The resultant compressed value can be represented in a number of different ways. One of these is "ppli." In the previous case, an equal condition results in a hit, and in the latter case, a less-than condition results. Figures 7 and 8 illustrate full key compression and the resultant index based on the following set of definitions and rules.

Figure 9 Search for "BATES"

FL/Value/Pointer	Input j	Input l	Output j	Output l	Rule	Condition
02/AE/PTR AEGER	0	1	0	1	1.c.	A(1) > E(1)
11/L/PTR ALESS	0	1	0	1	3.	j < F
11/N/PTR ANNET	0	1	0	1	3.	j < F
10//PTR ARENA	0	1	0	1	3.	j < F
03/BAK/PTR BAKEN	0	1	2	3	1.c	A(3) > E(3)
21/N/PTR BANGS	2	3	2	3	1.c	A(3) > E(3)
22/RB/PTR BARBA	2	3	2	3	1.c	A(3) > E(3)
31/L/PTR BARLO	2	3	2	3	3.	j < F
31/N/PTR BARNE	2	3	2	3	3.	j < F
30//PTR BARTH	2	3	2	3	3.	j < F
20//PTR BATES	2	3	2	3	1.a	j = F and L = 0
13/EAT/PTR BEATY						
30//PTR BEAUD						
21/H/PTR BEHEN						
21/N/PTR BENDE						
20//PTR BERBE						

Definitions for construction

1. The current key is the value to be compressed.
2. The previous key is the previous boundary value to be distinguished from the current key.
3. The next key is the next boundary value to be distinguished from the current key.
4. The initial value is the lowest and the last value is the highest within the constraints of the system.
5. Character positions are numbered from the left starting with one.

Construction rules

1. Calculate P, the position in which the previous and current key differ.
2. Calculate N, the position in which the current and next key differ.
3. If P equals N, save the Pth character of the current key, set the front compression to $P - 1$, and the length to one.
4. If P is greater than N, set the front compression to $P - 1$, and set the length to zero. It is not necessary to save any portion of the current key. An alternate rule is to set the front compression to $N - 1$. Since VSAM uses the $P - 1$ value as the front compression value, the examples in Figures 9 and 10 use it.
5. If P is less than N, set the front compression to $P - 1$, save the Pth through the Nth characters of the current key, and set the length to $N + 1 - P$.

Figure 10 Search for "BEHEN"

FL/Value/Pointer	Input j	Input 1	Output j	Output 1	Rule	Condition
02/AE/PTR AEGER	0	1	0	1	1.c	A(1) > E(1)
11/L/PTR ALESS	0	1	1	1	3.	j < F
11/N/PTR ANNET	0	1	0	1	3.	j < F
10//PTR ARENA	0	1	0	1	3.	j < F
03/BAK/PTR BAKEN	0	1	1	2	1.c	A(2) > E(2)
21/N/PTR BANGS	1	2	1	2	3.	j < F
22/RB/PTR BARBA	1	2	1	2	3.	j < F
31/L/PTR BARLO	1	2	1	2	3.	j < F
31/N/PTR BARNE	1	2	1	2	3.	j < F
30//PTR BARTH	1	2	1	2	3.	j < F
20//PTR BATES	1	2	1	2	3.	j < F
13/EAT/PTR BEATY	1	2	2	3	1.c	A(3) > E(3)
30//PTR BEAUD	2	3	2	3	3.	j < F
21/H/PTR BEHEN	2	3	3	4	1.b(4)	L exhausted
21/N/PTR BENDE						
20//PTR BERBE						

Figures 9 and 10 illustrate the searching of the index defined in Figures 7 and 8 for the values BATES and BEHEN based on the following set of definitions and rules.

Definitions and initializations for searching

1. The search value is **A**, which is a vector of characters $A(1)$ through $A(m)$. The subscript j is used to reference the individual characters and has a range of values from 0 to $m-1$.
2. The index entry value is **E**, which is a vector of characters $E(1)$ through $E(L)$ where L is the length of the compressed value. The subscript k is used to reference the individual characters and has a range of 1 to L.

Rules for searching

1. If j equals F then do the following:
   a. If L equals zero, this is the entry of interest.
   b. If $A(j+1)$ equals $E(k)$, then do the following:
      (1) Step j to $j+1$.
      (2) Step k to $k+1$.
      (3) If the compressed key is exhausted ($k < L$), repeat the rule 1 b.
      (4) If the compressed key is exhausted ($k = L$), this is the entry of interest.
   c. If $A(j+1)$ is greater than $E(k)$, step to the next entry and repeat the rules.
   d. If $A(j+1)$ is less than $E(k)$, this is the entry of interest.
2. If j is greater than F, this is the key of interest.
3. If j is less than F, then step to the next entry and repeat the rules.

A major shortcoming of the compressed-value index as currently defined is the requirement to search it in a serial fashion. This means that for an index with 1000 entries, the number of entries searched to locate a given key on the average is 500; the maximum is 1000. This effect can be minimized by dividing the index into records and building higher-level indexes. The difficulty with this approach is the number of levels required and the length of the search path. For example, if the index contains 40,000 entries and the record contains 20 entries, a four-level index is required.

Another approach to overcoming this shortcoming is to construct a different type of index record in which two or more levels are incorporated in a single record.[6] To do this, two additional rules must be added to the compression rules, and the definition of the previous key must be clarified. The additional rules for two levels per record are as follows:

1. Divide the record into n sections that on the average can contain n entries.
2. In the beginning of each section, construct an offset to the last (high) entry of the section.

The selection of the previous key to be used in compression is integral to the construction of multiple-level indexes and nondense indexes as well as multiple levels within the same record. As originally defined, the previous key is the previous boundary value. For nondense indexes, it is the last key compressed. For multiple-level indexes, it is the last key compressed at this level. When two levels are incorporated into a single record, the previous key depends on the type of entry being built. If it is a section entry (the last in the section), the previous key is the last key compressed in the previous section. Otherwise, it is the previously compressed key.

Take note that the definition of the next key has not been altered. Figure 11 illustrates the entries for a two-level, sectionalized index with a section size of three. Figure 12 illustrates the search through the index for the entry "BEHEN" using the previous search rules plus one additional rule. The new rule is that the values of j and k are reset to their input values after the processing of a level that is a high level or a section entry. Input values are defined to be the values used to start the evaluation of a given entry.

As a result of sectionalizing the index record to include two levels, the average number of entries examined in searching for a given key is the square root of n where n is the number of entries in the record. Table 1 compares the search time in terms of entries examined for an index record with 50 entries.

Figure 11 Two-level sectionalized index with compressed keys

	First level						Second level					
	Normal			Section			Normal			Section		
	F	*L*	*Value*	*F*	*L*	*Value*	*F*	*L*	*Value*	*F*	*L*	*Value*
AEGER	0	2	AE									
ALESS	1	1	L									
ANNET				0	2	AN						
ARENA	1	0										
BAKEN	0	3	BAK									
BANGS				0	3	BAN						
BARBA	2	2	RB									
BARLO	3	3	L									
BARNE				2	2	RT	0	4	BARN			
BARTH	3	0										
BATES	2	0										
BEATY				2	2	EAT						
BEAUD	3	0										
BEHEN	2	1	H									
BENDE				2	1	N						
BERBE	2	2	RB									
BERMA	3	0										
BETZA				2	0		1	1	E			
BIBLI	1	1	I									
BJORK	1	2	JO									
BJUNE				1	1	J						
BLOMQ	1	1	LOM									
BLOOM	3	0										
BODIN				1	2	OD						
BONNE	2	1	N									
BORET	2	2	RE									
BORNA				2	1	R				0	3	BOR
BOWMA												

Complex indexes

A complex index is used to obtain a search value for another index that is the merger of two or more indexes, or that is a combination thereof. An index that yields a search value for another index is called a *cascading index*. A merger of two or more indexes results in concatenated keys or pointers to multiple data sets, or a combination of both.

cascading index

An index is a cascading index[7] if its sequence set entry pointers yield a value on which another index is built. For example, if an index based on the field values for name yields a sequence set

Figure 12 Search for "BEHEN"

Entry examined	Input j	Input k	Output j	Output k	Condition
03BOR	0	1	1	2	A(2) > E(2) CHANGE LEVELS AND RESET j AND k
04BARN	0	1	1	2	A(2) > E(2)
11E	1	2	2	3	L EXHAUSTED CHANGE LEVELS AND RESET j AND k
13EAT	1	2	2	3	A(3) > E(3)
2IN	2	3	2	3	A(3) < E(3) CHANGE LEVELS AND RESET j AND k
30	2	3	2	3	f < F
21H	2	3	3	4	L EXHAUSTED

Table 1 Search time comparison

	Full key binary search	Compressed linear	Keys sectionalized
Minimum	1	1	1
Maximum	6	50	16
Average	6	25	8

pointer in the form of serial number and another index based on serial number points to the actual data records, the original index based on name is a cascading index.

The cascading index provides a number of advantages in terms of maintenance of secondary indexes when the primary index is a nondense index. Since nondense indexes require that inserts are made in place and that this may result in the shifting of data, indexes that point directly to the data in this environment must be updated as a result of the shift. This updating can be eliminated if the secondary index resolves to a primary key instead of a direct pointer.

This solution does affect the efficiency of searching secondary indexes because both the secondary and primary indexes must be manipulated. This cost can be minimized by the physical structure of the data in secondary storage and by additional primary storage.

concatenated keys

In a number of cases, a search of a given set of data is based on two or more keys called concatenated keys. This type of searching can be handled via multiple indexes, one per search key. If the searching requirements are such that multiple indexes are only needed for searches involving multiple keys, these indexes can be merged into a single index. The entry in this type of an index is composed of multiple values and a single point. For example, if the merged indexes were based on the keys a, b, and c, the entry could be the concatenated key a.b.c followed by a single data pointer.

This type of an index has the two disadvantages of size and limited search. If the size of the key is increased, the entry becomes larger, and the index tends to have more levels. This can be minimized by using fully compressed keys. Past observations indicate that the length of the fully compressed key is more or less independent of the size of the key. The search of an index with concatenated keys is limited in terms of the order of concatenation. For example, if the key is a.b.c, the index can be searched on a, a.b, and a.b.c but cannot be efficiently searched on b, b.c, or c. If the compression results really eliminate the size constraint and there is a requirement for this type of searching, multiple, concatenated key indexes may be the answer.

concatenated pointers

In certain environments, multiple sets of data may have a value in common. In many cases, this value is a data identifier and, as such, is a candidate for indexing. For example, a payroll set of data and a personnel set both contain name and serial number. An index built on a common value could be constructed so that its sequence set entries contain pointers for both sets of data resulting in concatenated pointers.

The major advantages of this type of an index are minimum space requirements and accessing when both sets of information are desired. The space savings are only in terms of eliminating the multiple occurrences of the key. The entry is therefore only smaller in terms of the size of the multiple entries; it is larger than one of the original entries. The major disadvantage is in the resulting size of the entry due to increasing the size of the total index such that the number of levels may increase. This problem becomes significant if the number of accesses to the single sets are more important than space and access to the multiple sets.

In certain cases, the pointer itself can be reduced in size. Consider the above example in which personnel and payroll had the serial number and name fields in common. If they both had a simple prime index based on serial number, they could share an index based on name provided that its sequence set entries had its pointers in the form of prime key values, i.e, the serial numbers.

This type of an index can be used to interrelate data records that have common data fields in a way similar to that of hierarchical data organizations.

Summary comment

A significant amount of work has been done in the area of general indexing in terms of simple indexing. This is reflected in the development of access methods such as VSAM. Very little work has been done on complex indexes. The only example given in this paper is the cascading indexes of CICS.[7] Future work could include an expansion of the items discussed under complex indexes. In particular, some attention should be given to the relationships between different sets of data. A limited set of these relationships are provided by hierarchical data structures provided by systems such as the Information Management System.

CITED REFERENCES

1. *IBM System/360 Operating System Indexed Sequential Access Method*, Form No. Y28-6618, IBM Corporation, Data Processing Division, White Plains, New York.
2. *OS/VS Virtual Storage Access Method (VSAM) Planning Guide*, Form No. GC26-3799, IBM Corporation, Data Processing Division, White Plains, New York.
3. H. K. Chang, "Compressed indexing method," *IBM Technical Disclosure Bulletin* **II**, No. 11 (April 1969).
4. W. A. Clark, IV, C. T. Davies, Jr., K. A. Salmond, T. S. Stafford, *High-Level Index-Factoring System*, United States Patent Number 3,646,524 (February 29, 1972).
5. W. A. Clark, IV, K. A. Salmond, T. S. Stafford, *Methods and Means for Generating Compressed Keys*, United States Patent Number 3,593,309 (January 2, 1969).
6. R. E. Wagner, "Searching a compressed index," to be published in the *IBM Technical Disclosure Bulletin* (1973).
7. *Customer Information Control System; OS-Standard Program Description*, Form No. SH20-0605, IBM Corporation, Data Processing Division, White Plains, New York.