Faster with FAST KSAM

by
Stephen M. Butler
Director of Data Processing
Paradise Valley Hospital
National City, CA

BACKGROUND

Being a hospital, we maintained a manual index to patient numbers. The mechanical device was overloaded (280,000 in a 250,000 capacity Diebold), and the repair bills were worse.

During the analysis of the HP-3000 a prototype computerized Medical Index using IMAGE was written; but ALPHA or NUMERIC sequence searches could not be used--who wants a sorted chain of 280,000+ entries. A phonetic search was implemented but the chain had cases of 2000+ entries. A generic key capability for the phonetic-birthdate search mechanism was needed.

The prototype index gave us the impetus to acquire the 3000. As the conversion from a Honeywell 115 neared completion, KSAM became available. It had the capabilities needed:

1. Multiple keyed ISAM.

2. Generic keys.

We claim to be the GAMMA test site for KSAM--if such exists. It seems there were updates every other week--at least we saw our SE! Finally there was a version of KSAM clean enough to attempt a load of the 280,000+ records. It took 5 days for the disk drives to survive the shake out test; but one of the other systems started acting funny. Soon we were in the GAMMA test phase again. After several attempts to fix the bug, the SE took our test program and disappeared. (He claimed he wouldn't come back until there was a version of KSAM that would work on his machine!)

The next week he laid the update tape on our desk; we had become past masters of doing KSAM updates. The SE's parting comment was, "That 5 day load should be faster." "Did we get FAST KSAM?" "Can't say; but don't tell anybody else."

The reload took 2-1/2 days. Not the 1000% improvement expected, and there were more bugs. So, we wrangled a day at the lab to find out why a particular bug had so many facets and why it was taking upwards of a month to fix a problem we felt was critical. A lot of information was passed in both directions, and the lab thanked us for being a BETA test site. Wished somebody had told us sooner!

Faster with FAST KSAM

The new version of KSAM was quoted to be the pre-release version that would follow the MIT following 1814. That was Feb. 9, 1978. It passed all our tests and is now on the 1814 M.I.I.

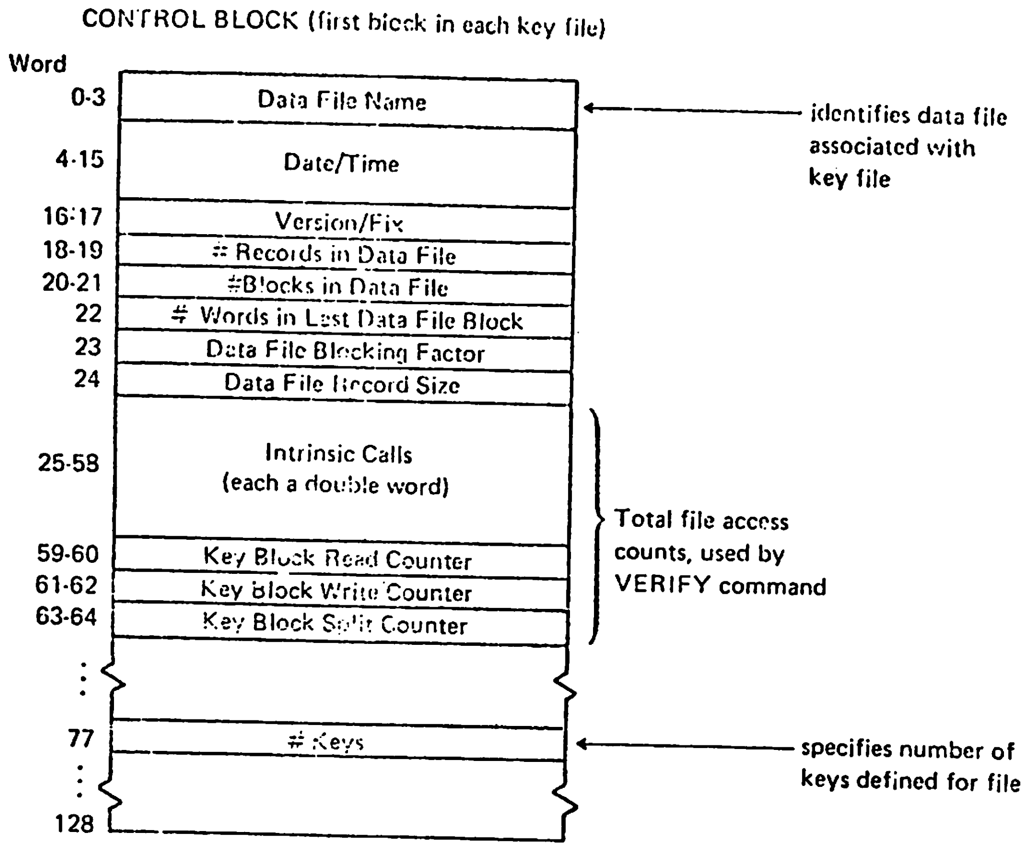
Following tips from the lab, the load took 20 hours. Not bad for 280,000 records. In retrospect we are happy to have picked KSAM.

KEYFILE

An understanding of the KEYFILE will help in knowing why the following tips work. A close reading of the new Appendix B in the KSAM manual will be useful.

First, the KEYFILE record size is one sector (128 words; 256 bytes).

The CONTROL block is described in FIGURE 1. The number of keys per record is the main item of interest.



CONTROL BLOCK (first block in each key file)
Word
0-3
Data File Name
identifies data file associated with key file
4-15
Date/Time
16-17
Version/Fix
18-19
# Records in Data File
20-21
#Blocks in Data File
22
# Words in Last Data File Block
23
Data File Blocking Factor
24
Data File Record Size
25-58
Intrinsic Calls (each a double word)
Total file access counts, used by VERIFY command
59-60
Key Block Read Counter
61-62
Key Block Write Counter
63-64
Key Block Split Counter
77
# Keys
specifies number of keys defined for file
128

FIGURE 1. CONTROL BLOCK layout. Note word 77.

The KEY DESCRIPTOR block has one entry of 8 words for each defined key. The detailed layout is in FIGURE 2. The most useful items right now are the pointers to the ROOT KEY ENTRY block for each of the defined keys.

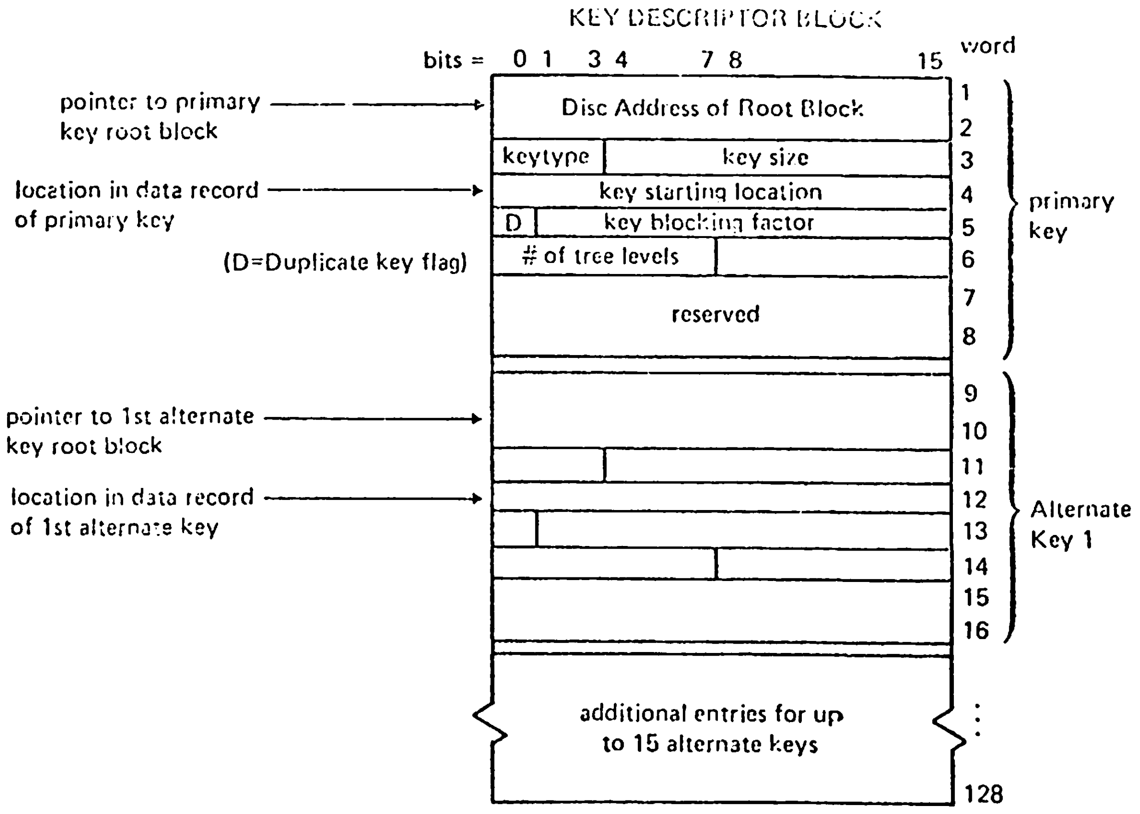

KEY DESCRIPTOR BLOCK
bits = 0 1 3 4 7 8 15
word
pointer to primary key root block
location in data record of primary key
(D=Duplicate key flag)
Disc Address of Root Block
keytype
key size
key starting location
D
key blocking factor
# of tree levels
reserved
primary key
pointer to 1st alternate key root block
location in data record of 1st alternate key
Alternate Key 1
additional entries for up to 15 alternate keys
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 128

FIGURE 2. KEY DESCRIPTOR BLOCK. Each entry consists of 8 words. The RESERVED area is a pointer to the free list chain for this key.

The KEY ENTRY blocks contain the key values and pointers used to make KSAM do its thing. A quick look at FIGURE 3 will show that a key entry is composed of:

1. Double-word relative record number of the KEY ENTRY block that sequentially comes before this entry.

2. KEY value as it is in the Data Record. A slack byte is at the end if the key length is odd. This slack byte IS NOT initialized.

3. Double-word relative record number of the data record in the data file.

4. Double-word relative record number of the KEY ENTRY block that sequentially comes after this entry.

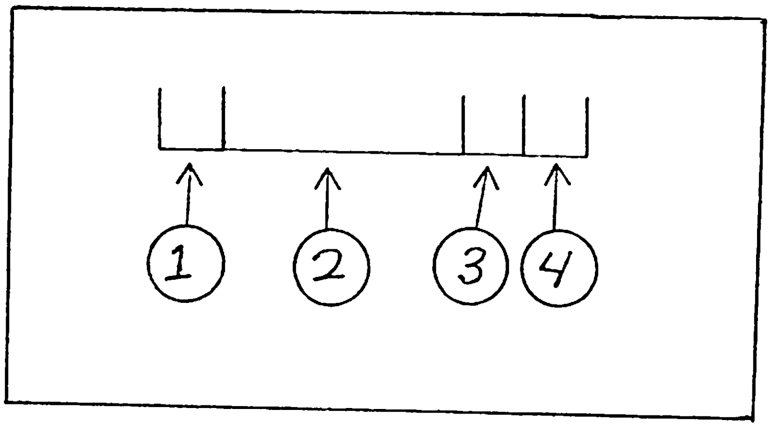


FIGURE 3. KEY ENTRY. See text for description of numbered items.

With two or more keys item 4 becomes item 1 for the next key in that KEY ENTRY block, see Figure 4. Thus there is 1 more KEY ENTRY pointer than the number of active keys in that block. Since each KEY ENTRY has a data pointer also, the number of double word pointers can be written as

$$2N + 1$$

where N is the number of keys per KEY ENTRY block.

Each KEY ENTRY block starts out with a double-word integer whose value is the relative record number of that block. The next word has a count of the number of active keys in this KEY ENTRY block. Subsequent records within the same KEY ENTRY block do not have this information. The key value within a KEY ENTRY block can be split across the physical blocks of the KEYFILE. Using FCOPY to dump the KEYFILE with ';NOKSAM; OCTAL;CHAR' options will allow a person to inspect the actual layout for a particular file. Thus a person could simulate conditions that would normally be hidden deep inside a large file. Once you know the general layout you will quickly pick up the specific pieces of information needed to navigate through the KEYFILE.

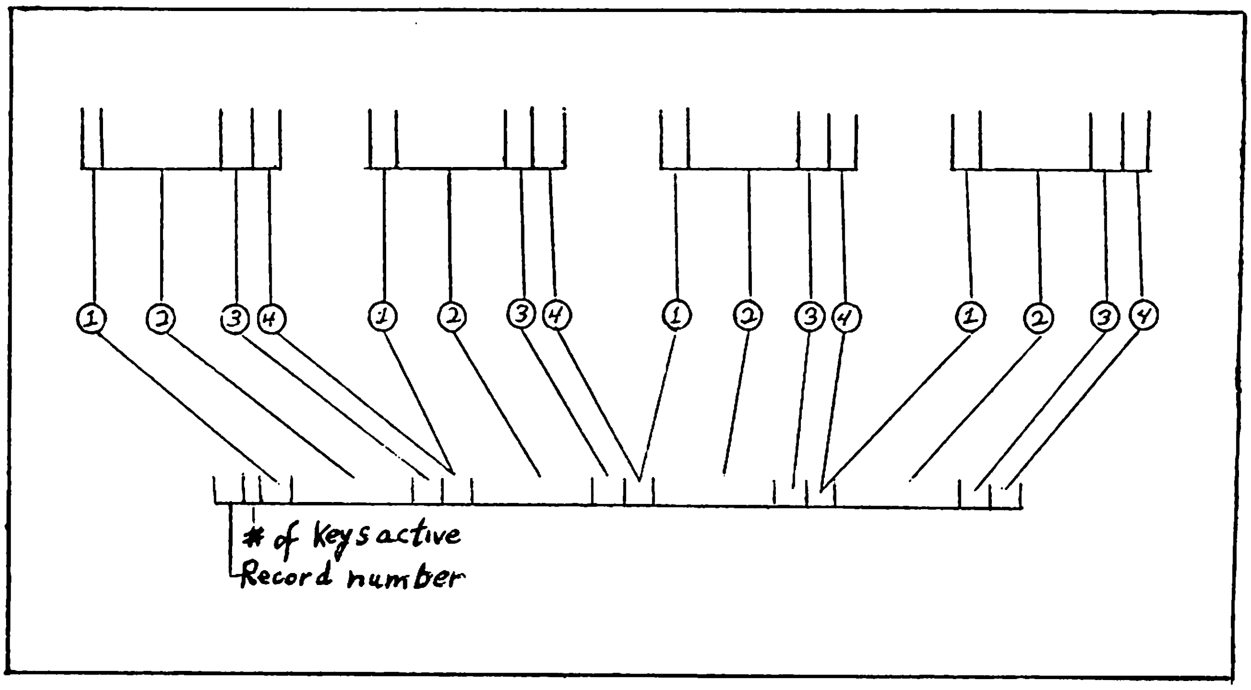


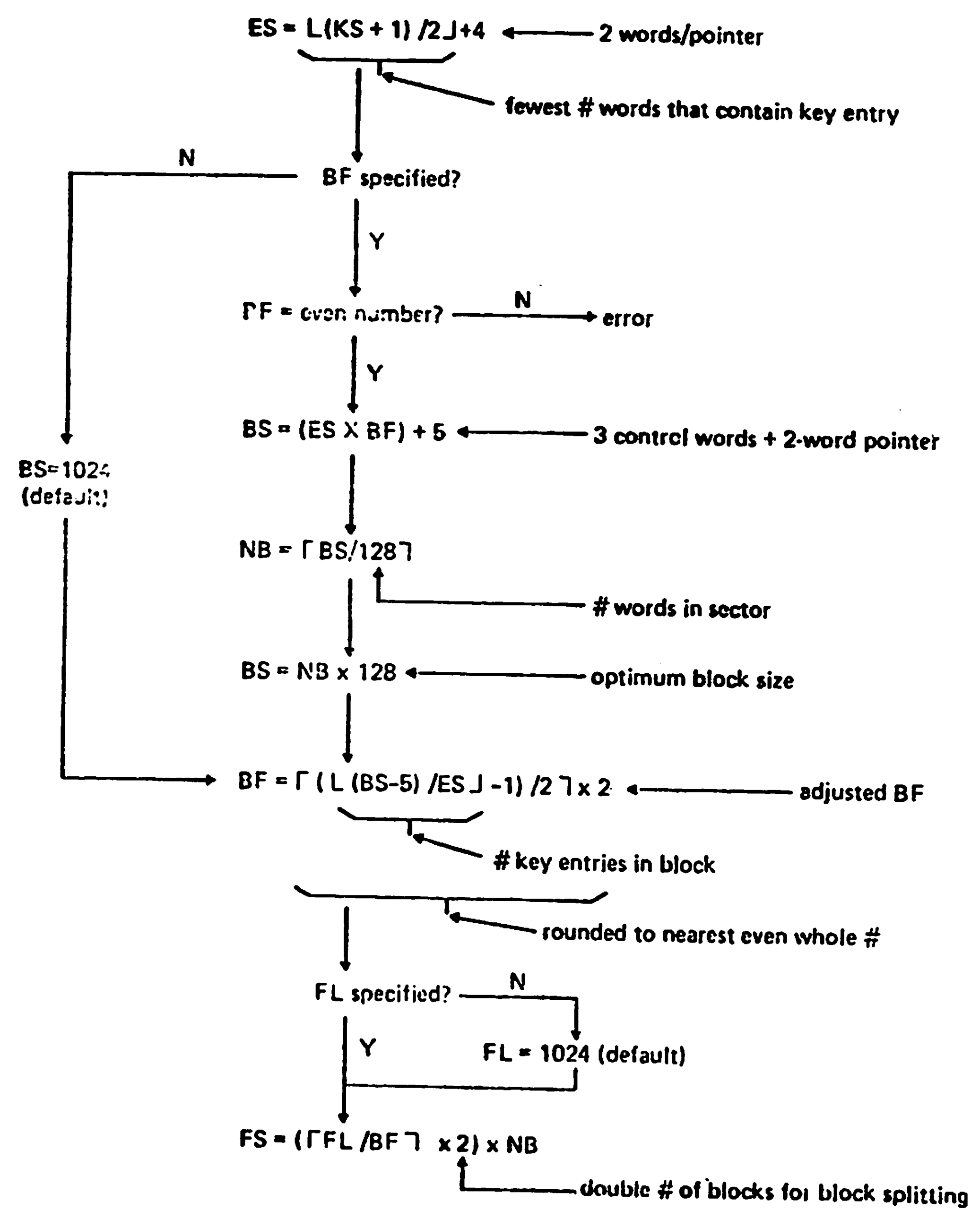
FIGURE 4. KEY ENTRY BLOCK. Words 1 & 2 contain the relative record number for this block. Word 3 contains the number of active keys. Notice how items 4 and 1 are shared by adjacent KEY ENTRYs.

We have spent upwards of two days at a time sifting through a KEYFILE in this manner in order to pin point KSAM bugs. On one occasion a bug seemed to occur only on our 280,000 record KSAM file. My boss bet a milksnake that I couldn't simulate it with less than 100 records. I did it with 8--but I first knew exactly what was happening.

FIGURE 5 can be followed to calculate the blocking factor (BF) for each key. A specified BF is used as a minimum and is adjusted upwards to make full use of any remaining area in the last sector. The default BF is chosen so that the KEY ENTRY block will span 8 sectors--1024 words (2048 bytes). If the KEY ENTRY block spans more than 16 sectors (2048 words or 4096 bytes), the BF is reduced so a maximum of 16 sectors is used.

With multiple keys the largest KEY ENTRY block size is used to calculate the BF for all keys. Thus all KEY ENTRY blocks occupy the same number of sectors. This along with the 16 sector maximum are by-products of requirements for using the KSAM extra data segment.

KS	=	key size in bytes
ES	=	key entry size in words
BF	=	blocking factor (number of key entries per block)
BS	=	key block size
FL	=	data file limit in records
NB	=	number of sectors per key block
FS	=	key file size in sectors
$\lceil\,\rceil$	=	round up $\lfloor\,\rfloor$ = round down



$ES = \lfloor (KS + 1)/2 \rfloor + 4$
2 words/pointer
fewest # words that contain key entry
BF specified?
N
Y
BF = even number?
N
error
Y
$BS = (ES \times BF) + 5$
3 control words + 2-word pointer
BS=1024 (default)
$NB = \lceil BS/128 \rceil$
# words in sector
$BS = NB \times 128$
optimum block size
$BF = \lceil (\lfloor (BS-5)/ES \rfloor - 1)/2 \rceil \times 2$
adjusted BF
# key entries in block
rounded to nearest even whole #
FL specified?
N
Y
FL = 1024 (default)
$FS = (\lceil FL/BF \rceil \times 2) \times NB$
double # of blocks for block splitting

FIGURE 5. Calculating blocking factor (BF) and file size (FS) for one key.

Assume a file with 2 keys defined as:

KEY = B,1,53,12
KEY = B,54,13,20

For Key 1:

KS=53
FL=1024 (default)
BF=12

Calculation of FS:

$ES=\lfloor (53+1)/2 \rfloor +4 = 27+4 = 31$

$BS=(31\times 12)+5 = 377$

$NB=\lceil 377/128 \rceil = \lceil 2.9 \rceil =$ 3 sectors

$BS= 3\times 128 = 384$

*$BF=\lceil (\lfloor (384-5)/31 \rfloor -1)/2 \rceil \times 2$

$= \lceil (\lfloor 12.2 \rfloor -1)/2 \rceil \times 2$

$= \lceil (12-1)/2 \rceil \times 2$

$= \lceil 5.5 \rceil \times 2 = 6\times 2 = 12$

$FS=(\lceil 1024/12 \rceil \times 2)\times 3$

$=(\lceil 85.3 \rceil \times 2)\times 3$

= 516 sectors

For Key 2:

KS=13
FL=1024 (default)
BF=20

$ES=\lfloor (13+1)/2 \rfloor + 4 = 7+4 = 11$

$BS= (11\times 20)+5 = 225$

$NB=\lceil 225/128 \rceil = \lceil 1.75 \rceil =$ 2 sectors

$BS= 2\times 128 = 256$

*$BF=\lceil (\lfloor (256-5)/11 \rfloor -1)/2 \rceil \times 2$

$= \lceil (\lfloor 22.8 \rfloor -1)/2 \rceil \times 2$

$= \lceil (22-1)/2 \rceil \times 2$

$= \lceil 10.5 \rceil \times 2 = 11\times 2 = 22$

$FS=(\lceil 1024/22 \rceil \times 2)\times 2$

$=(\lceil 46.5 \rceil \times 2)\times 2$

= 188 sectors

Since key 1 has the largest block size (384 words in 3 sectors), its blocking factor is unchanged. The blocking factor for key 2 is adjusted so it has the same block size. The following values are used:

ES=11 ⟵ entry size calculated for key 2
BS=384 ⟵ block size of key 1 (now used for key 2, also)
FL=1024 ⟵ default file size in words
NB=3 ⟵ number of sectors needed for each block of 384 words

Calculate the new blocking factor for key 2:

*$BF=\lceil (\lfloor (384-5/11 \rfloor -1)/2 \rceil \times 2$
$= \lceil (\lfloor 34.4 \rfloor -1)/2 \rceil \times 2$
$= \lceil 16.5 \rceil \times 2 = 17\times 2 = 34$

$FS=(\lceil 1024/34 \rceil \times 2)\times 3$
$=(\lceil 30.1 \rceil \times 2)\times 3 = 186$ sectors

Summing the two file sizes and adding two sectors for control and key descriptor information, the total file size in sectors is:

516 + 186 + 2 = 704 sectors

*The algorithm to calculate BF can be expressed more simply if the result can be checked for an even number:

$BF=\lfloor BS-5/ES \rfloor$ If BF is an odd number, set BF=BF-1

FIGURE 6. Calculating file size (FS) for multiple keys.

FIGURE 6 shows how the size of the KEYFILE is calculated. Since each block can be a minimun of half full, twice as many KEY ENTRY blocks are assigned as would be needed if each block were full.

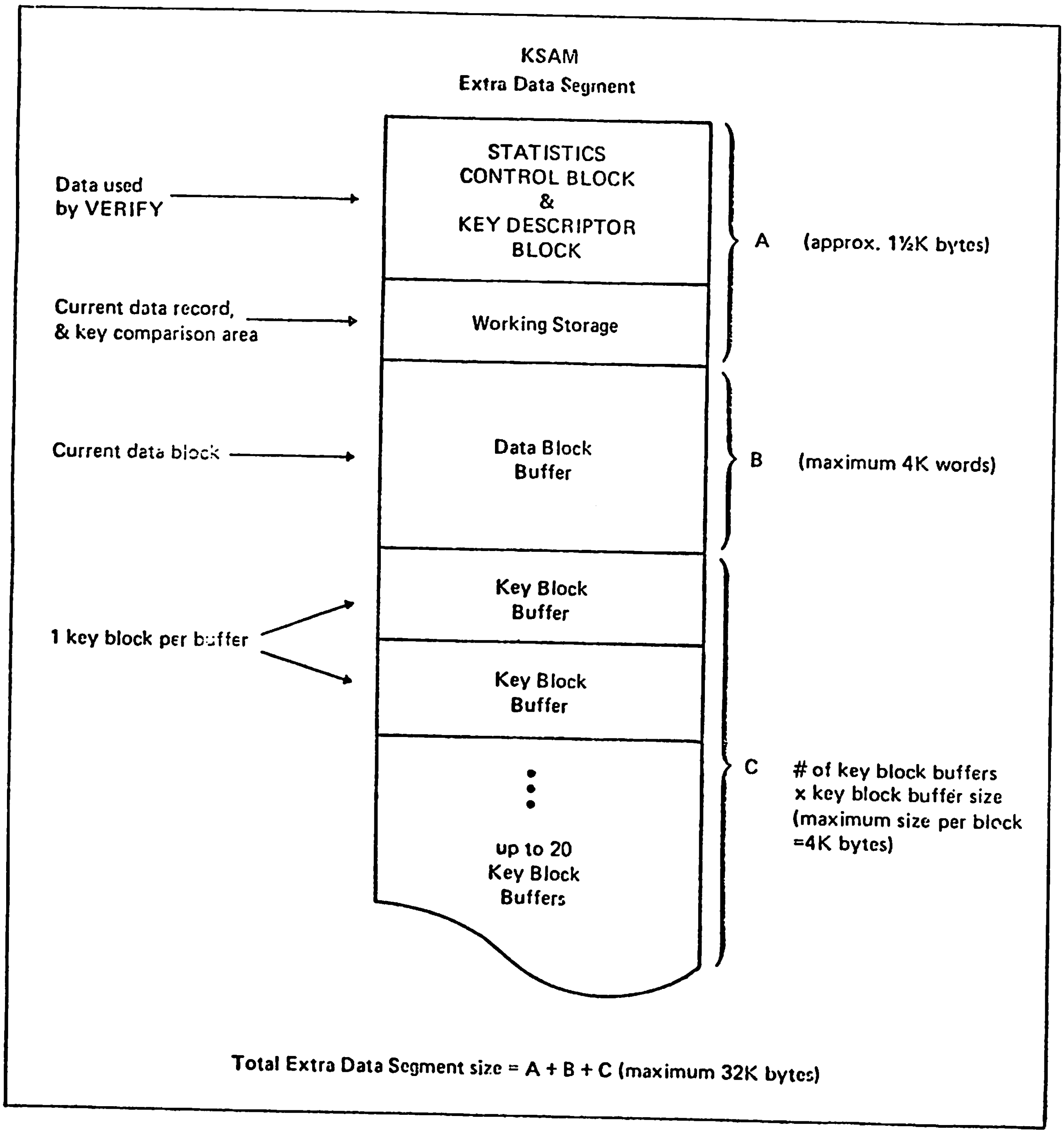

KSAM
Extra Data Segment
Data used by VERIFY
STATISTICS CONTROL BLOCK & KEY DESCRIPTOR BLOCK
Current data record, & key comparison area
Working Storage
A (approx. 1½K bytes)
Current data block
Data Block Buffer
B (maximum 4K words)
1 key block per buffer
Key Block Buffer
Key Block Buffer
up to 20 Key Block Buffers
C # of key block buffers x key block buffer size (maximum size per block =4K bytes)
Total Extra Data Segment size = A + B + C (maximum 32K bytes)

FIGURE 7. KSAM XDS.

KEY BUFFERS

The new features of FAST KSAM can now be put to use. An extra data segment (XDS) is used to handle all I/O to the KSAM file. The size of this XDS is limited to 16K words. Approxiametly 1-1/2K are used for overhead and control information. Only one buffer is used for the DATAFILE; it has a maximum of 4K words and is the size of one block from the DATAFILE. We use a program that calculates the best BF that will fit in 8 or fewer sectors so the data buffer will be 1K or less. The rest of the XDS can be used for KEY ENTRY buffers.

To find how many buffers could be used (all calculations in words):

1. Subtract the 1-1/2K of overhead.
2. Subtract the size of the data buffer. Lets assume 1K.
3. Divide what's left by the size of a KEY ENTRY block--default is 1K.
4. Round the answer down to the next integer.

   So, (16K - 1-1/2K - 1K) / 1K = 13 buffers.

To get them:

```
        :FILE ksamfile;DEV=,,13
```

If this would cause the XDS to be larger than 16K words, KSAM will automatically decrease the number of buffers.

Since KSAM does have a fairly good algorithm for choosing the default number of key buffers (see FIGURE 8), once the file has stablized you may wish to restrict the use of the FILE equation to loading or otherwise making large numbers of changes to the file. If the file is empty, KSAM will default to the minimal number of buffers for the type of open specified. For this reason you should specify the number of buffers you will actually need as KSAM will not allocate more buffers as the file is filled.

Each process that opens the KSAM file gets its own XDS. The number of buffers in these XDSs are dependent upon the type of open specified and the number of keys in the file at the time of opening. Therefore, these XDSs could have differing numbers of key buffers.

Access Type	Buffers Assigned
Read Only Access	1 buffer per level in *primary* key structure
Write Only Access	3 buffers per primary key + 3 buffers per alternate key + 3 buffers
Other Access (Read/Write or Update)	1 buffer per level in primary key structure + 1 buffer per level in alternate key structure + 3 buffers
	(up to a maximum of 20 buffers)

FIGURE 8. Default key buffers allocated at FOPEN.

DUPLICATES

The other ISAM packages that I am familiar with do not allow for duplicate keys. At first glance, one would think it is a blessing that KSAM does; but to paraphrase the LAB: If there are more than 10 duplicates for a particular key, then don't have this key or make it unique.

Whenever a key is added to the file it is added after any duplicates that exist for that value. KSAM must search the KEYFILE to find that last entry. A START causes a search for the first entry.

Two of the most common ways of making duplicates unique are:

1. Put a time stamp (HR:MM:SS) after each key. For calls less than 1 sec. apart, this would still leave them duplicates.

2. Put a copy of the primary key after the other keys.

In COBOL the primary key must be unique. In the Medical Index case it was 7 bytes long so we were not any worse off than using the time stamp. Another method will be proposed in the ENHANCEMENT section.

TIMINGS

A stand-alone environment is not readily available on our system. The following timings show both CPU seconds and WALL TIME to load 10,000 records into an empty KSAM file.

	Default Buffers	13 Buffers
KSAM expected duplicate keys and duplicate keys were loaded.	1111/5333	533/1010
KSAM expected duplicate keys; but all keys loaded were unique.	870/4627	326/484
KSAM expected unique keys and unique keys were loaded.	1183/5992	389/567

FIGURE 9. This shows the CPU seconds/WALL seconds to load 10,000 records into an empty KSAM file. Three keys were used--7 bytes, 20 bytes, and 43 bytes. The BASIC procedures were used to load the file.

ONLINE BENEFITS

By correctly specifying the number of key buffers and utilizing unique keys there will be a marked improvement in throughput. But the other benefits even outweigh this.

An example please: two users will access a KSAM file that has 4 records in it. We will assume 1 defined key and a KEY ENTRY blocking factor of 4. Therefore, the ROOT KEY ENTRY block is full. Any new records added to the file will cause a key-block-split. We proceed:

1. Both users open the file for shared access.

2. User A LOCKs the file and reads the first two logical records.

3. User A UNLOCKs the file and User B LOCKs it.

4. User B writes a record whose logical value places it #3.

5. User B UNLOCKS the file and posts the updated buffers.

6. The file now has 1 KEY ENTRY in the ROOT block. This points to two other blocks. The first block contains the keys User A just read. The second block contains the two keys User A expects to see. In actual practice he should get the record that User B just posted.

7. User A LOCKs the file again and calls for the next READ (sequential of course).

8. The next KEY ENTRY that User A would previously have used would have been #3 in the ROOT. At least that is all that KSAM knows. But the ROOT now has only one entry. Since the 3rd entry no longer exists we are at the end of the file, so return an EOF condition.

User A was lucky. If there had been many KEY ENTRY blocks and User A had been down several levels, the following possibilities could have happened (we have seen results to indicate they have happened to us):

1. The current block would no longer be included in the key structure; but the process is not aware that is has been placed in a free buffer list, so the process uses it.

2. The same for a previous level; ie, the ROOT or one of the intermediate levels was moved away from where we expected it after the last access.

3. The current or a higher level was reshuffled. All blocks are active; but not necessarily in the same tree structure as before.

We turned this in as a bug--and promptly got laughed at. This is one of those dubious features we all enjoy. KSAM will not keep track of any reorganization that may occur while the file is unlocked. The buffers are refreshed by the physical blocks that were last used in the XDS. KSAM will not check to be sure that these contain the logical values last used. So, you must reposition the pointer yourself. You can do that by using the START procedure with a relop of strictly greater than the key that was returned in the last read even though a number of changes may have occurred to the point of deleting the record last read. This is a multi-user online environment, right! Again:

Faster with FAST KSAM

1. LOCK the file.

2. Do a START using last key read and greater than relation.

3. Now do that sequential READ.

4. If you were going to do a READBYKEY or a REWRITE in random/dynamic mode, then items 2 & 3 are not needed.

5. UNLOCK when done.

That process can't be done with duplicate keys. If the last READ was in the middle of a duplicate key, the START would bypass the duplicates not read. Unique keys are a MUST in order to do the above.

In the case of updates to the file, one more item is needed--RECORD LOCKING. Set aside one byte in the record to be a locked/unlocked flag. When a record is read prior to updating it:

1. Check that field--if it is locked, then report it as being locked or work out a mechanism to hang until it frees up. We don't like hanging up, since a process might abort and leave a record flagged as locked. It hasn't happened in 6 months at our site, but? If unlocked, then continue.

2. Set the flag for lock.

3. REWRITE the record.

Now you can UNLOCK the file and KNOW that when you're ready to update that record it WILL BE the SAME. P.S. Be sure to reset that flag!

PROPOSED_ENHANCEMENTS

If we find time between this writing and the International meeting, we may have a set of COBOL copylibs to simulate this. We want KSAM to do a lot of the dirty work for us, so:

1. It should automatically call the LOCK for us if we failed to. Of course, only we know when to UNLOCK, so this is only a onesided benefit. It would still be useful.

2. The first call following a LOCK (whether directly or by #1) should cause a call to the START to reposition the pointer, unless this is a READBYKEY or START or REWRITE in random/dynamic mode.

3. Force all keys to be unique by:
   A. Assigning a double-word integer as the primary key.
   B. Appending this integer (4 bytes) to the end of every key. (Of course, if one key ends at the place the primary key takes off, then just increase the length of that key.)

4. Allow record locking (if necessary, set aside one byte) ie, a read with lock option.

If everything else in KSAM worked the same, we could then define a next step that would reuse space left by deleted records.

1. The record with a primary key of zero should hold two pointers:
   A. The next integer to be used for the primary key (ie, EOF pointer).
   B. The primary key value of the most recent record deleted.

2. The alternate keys for primary key of zero and all deleted records should be set to HIGH-VALUES except for the last 4 bytes which would be a copy of the primary key.

3. An EOF pointer would be returned upon reading a record with HIGH-VALUES in all bytes except the last 4 of any key.

4. Every deleted record would have the primary key value of the previously deleted record. (A push down stack or free list chain.)

5. Whenever a REWRITTE occurs it would be keyed off of the primary key.

6. A WRITE would first use up the free list chain before incrementing the primary key.

You will notice the above has been specified in such a manner that a user of the current KSAM could write a set of procedures to make KSAM function as suggested. Now for the bombshell. KSAM already appends a double-word integer to <u>ALL</u> keys. It is more properly called the data record pointer.

If the lab would do the above, they could do it at a more simple level (ie, they wouldn't need to use the primary key).

1. They could use the EOF pointer as now.

2. They would need to set up a free list chain for the DATAFILE (they already have one for the KEYFILE).

3. They would have to keep track of the key and record number of the last logical record read. Then automatically reposition according to the first set of proposed enhancements.

4. They already append the record pointer to the keys so no physical change would be necessary--as would be if one of us users was to try.

5. In short, the lab has all the information necessary to do the job except for the free chain list in the DATAFILE.

Whether or not the lab does this enhancement, we already do something similar by using the primary key to append to all others; but we intend to write the procedures necessary to make KSAM look like our proposal. Anybody interested?

There are some enhancements that only the lab can do:

1. A central XDS similar to the recent IMAGE update. Only one XDS per file no matter how many users are using that same file. A small XDS may be needed for each process to keep track of the last logical key value and other local data.

2. Implement a LOCKing similar to IMAGEs.

3. OK, let's go for it! USE THE IMAGE CALLS TO HANDLE KSAM. This means:
   A. A schema processor to look for data sets type K or KSAM.
   B. We could have FAST sorted chains.
   C. In fact we could now have sorted Master files (just a KEYFILE to point to the Master set entries).
   D. DBFIND would also function as START for KSAM files.
   E. DBGET would take over as:
      - the current calculated DBGET would work for READBYKEY.
      - a new mode for sequential reads as opposed to serial.
   F. The DBLOCK would give the same type of locking scheme for KSAM as is now being enhanced for IMAGE.

These ideas will be implemented in our shop as far as possible. We plan to write a set of routines that will handle both the KSAM and IMAGE procedures. If the lab beats us, we shall be very happy to conceed the race!