Integer Keys: The Final Chapter

Fred White
Senior Research Scientist

Adager Corporation
Sun Valley, Idaho 83353-3000 U.S.A.
Tel. (208) 726-9100 *Fax* (208) 726-8191
fred@adager.com *http://www.adager.com*

Introduction

The calculation of primary addresses for IMAGE keys of data types X, U, P and Z is performed by a hashing algorithm whose goal is to generate a uniform distribution of primary addresses on the closed interval [1,C] where C is the capacity of the master dataset.

Despite what you may have read or heard from various IMAGE evangelists, this is not true for keys of data types I, J, K and R. Keys of these types are called "non-hashing" keys for the simple reason that they are ***not*** hashed! IMAGE makes ***no*** attempt to distribute them uniformly! The user has ***absolute*** control over their primary address assignment! This control is exercised by the user's method of assigning key values and his choice of master dataset capacity.

There are two kinds of non-hashing keys: "type R" and "types I, J and K". I shall refer to key types I, J and K as "integer keys".

With the proper tools and knowledge, integer-keyed master datasets can be created so that they have no synonyms and are not wasteful of disc space. The ill-advised use of integer keys typically leads to performance disasters!

History

In January of 1972, the IMAGE/3000 project team agreed to provide for non-hashed master datasets in which the primary address calculation would be in the hands of the user rather than controlled by IMAGE's hashing algorithm.

After considering various options, we decided on the following:

1. IMAGE keys of types I, J, K and R would ***not*** be hashed.
2. These keys could be of any length acceptable to IMAGE.
3. Only the rightmost 31 bits (the "determinant") would be used to calculate the primary address. (For one-word keys, their 16 bits are padded on the left with zeroes.)

4. The determinant is then divided by the dataset capacity yielding a remainder which becomes the primary address unless it is zero, in which case the capacity is assigned as the primary address.

Notice that, for a given capacity C, if we use determinant values N between 1 and C, the primary address for each N is N.

Furthermore, if these determinant values are all unique, the user will have taken advantage of IMAGE's integer key facility to provide himself with the good old, traditional, Direct Access Method (DAM).

However, IMAGE does not demand uniqueness of determinants nor does it restrict their values to the range 1 to C. We shall see that this “loosening up” of the constraints on the values of N, if used, will typically lead to a horrible performance problem unless the user is armed with a tool for intelligently selecting a capacity which will enable him to avoid such a performance pitfall.

Non-hashing Key Performance Pitfalls

In this section I address two distinct examples of bad uses of IMAGE's integer key facility. Both of these appeared in an earlier paper of mine “The Use and Abuse of Integer Keys”.

The first example demonstrates that our choice to not hash keys of IMAGE type R was a horrible design decision.

Example 1: The Synonym Pitfall

This “pitfall” arises whenever a user elects to use a key of IMAGE type R4 whose key values are, for the most part, integers.

To understand why, one must be knowledgeable about the format of 64-bit reals as represented on the HP3000 family of computers.

The leftmost bit is the sign bit, the next 9 bits are the exponent, and the rightmost 54 bits constitute the mantissa (excluding the most significant bit).

As a consequence, the floating point format of all integers of magnitude less than 8388609 (2**23+1) is such that the low order 31 bits are all zeroes. All entries with keys like this will be in a single synonym chain having the dataset capacity as its primary address!

To add a new entry to this chain, DBPUT must traverse the entire synonym chain to ensure that the key value of the new entry is not a duplicate before adding it to the chain. This has an impact on performance proportional to the number of entries in the chain (which could be in the thousands) and inversely proportional to the blocking factor.

Also, each DBFIND (or mode 7 DBGET) will, on average, be forced to traverse half of the chain to locate the desired entry!

The picture improves somewhat if an R2 field is used. In this case, the rightmost 31 bits (which are reduced modulo the capacity) include the exponent bits and all bits of the mantissa.

Consequently, the various key values will not all be assigned the same primary address. However, if these values have only a few significant binary digits in their mantissa, the rightmost bits will tend to be all zeroes which will lead to a high percentage of synonyms regardless of the capacity. This is especially true if the capacity is a power of 2 because we are treating the R2 field as a double integer and, if several key values have zeroes in their rightmost N bits, they are all divisible by 2**N and thus will all be synonyms.

For either R2 or R4 keys there will always be significant synonym problems unless the rightmost 31 bits of the keys, in and of themselves, form a set of doubleword values which represent a uniform distribution over the closed interval [1,2**31-1]. In this event, the primary addresses assigned will tend to be uniformly distributed over the master dataset even though no hashing occurs.

The bottom line is: if your R2 or R4 values don't fit this pattern, avoid using them as keys.

When Hewlett-Packard finally introduces an IEEE real data type in IMAGE, similar warnings will apply since the data formats differ from the Classic 3000 reals only in the number of bits in the exponents. The 32-bit IEEE real has an 8-bit exponent, the 64-bit IEEE real has an 11-bit exponent and the 128-bit IEEE real has a 15-bit exponent.

The next example demonstrates the problems which can arise when using integer keys (IMAGE types I, J or K).

Example 2: The Primary Clustering Pitfall

One Friday in 1978 I received a phone call from an insurance firm in the San Francisco Bay Area. I was told that their claims application was having serious performance problems and that, in an attempt to improve the situation, they had, on the previous Friday, performed a DBUNLOAD, changed some capacities and then started a DBLOAD which did not conclude until the early hours of Tuesday morning!

They were a US$100 million-plus company which couldn't stand the on-line response they were getting and couldn't afford losing another Monday in another vain attempt to resolve their problems.

Investigation revealed that claims information was stored in two detail datasets with paths to a shared automatic master. The search field for these three datasets was a double integer key (IMAGE type I2) whose values were all of the form YYXXXXX (shown in decimal) where YY was the two-digit representation of the year (beginning with 71) and where, during each year, XXXXX took on the values 00001, 00002, etc. up to 30000.

Although the application was built on IMAGE in late 1976, earlier claims information (from 1971 through 1976) was included to be available for current access. I do not recall the exact capacity of the master dataset but, for purposes of displaying the nature of the problem (especially the fact that it didn't surface until 1978) I will assume a capacity of 350000.

Although the number of claims per year varied, the illustration will also assume that each year had 30000.

The first claim of 1971 was claim number 7100001 to which (using a capacity of 350000) IMAGE would assign a primary address of 100001. This is because 7100001 is congruent to 100001 modulo 350000.

The 30000 claims of 1971 were thus assigned the successive record numbers 70001 through 100000 (a cluster of primaries).

Similar calculations show that the claims for each year were stored in clusters of successive addresses as follows:

Claim numbers	Record Numbers
7100001 through 7130000	100001 through 130000
7200001 through 7230000	200001 through 230000
7300001 through 7330000	300001 through 330000
7400001 through 7430000	50001 through 80000
7500001 through 7530000	150001 through 180000
7600001 through 7630000	250001 through 280000
7700001 through 7730000	1 through 30000

Note that no two records had the same assigned address and thus that there were no synonyms and that all DBPUTs, DBFINDs and keyed DBGETs were very fast indeed!

Along came 1978!!!

Unfortunately 7800001 is congruent to 100001 modulo 350000 so that the first DBPUT for 1978 creates the very first synonym of the dataset. Claim 7800001 is, in fact, a synonym of claim 7100001.

DBPUT attempts to place this synonym in the block occupied by claim 7100001 but that block is full so DBPUT performs a serial search of the succeeding blocks to find an unused location. In this case, it searches the next 60000 records before it finds an unused address at location 130001! Even with a blocking factor of 50, this requires 1200 additional disc reads making each DBPUT approximately 200 times as slow as those of all previous years!

Note that the next claim of 1978 (claim 7800002) is congruent to 100002 and is thus a synonym of claim 7100002. This also leads to a serial search which ends at location 130002! Thus the DBPUT of each claim for 1978 results in a search of 60000 records 59999 of which were inspected during the preceding DBPUT!

Primary clustering had claimed another victim! The designer of this system had unknowingly laid a trap which would snap at a mathematically predictable time, in this case 1978. After struggling with this problem for months, the user escaped the clustering pitfall by converting to “hashed keys” (in both the database and the software); a very expensive conversion!

Note that the problem was NOT a “synonym” problem in the sense that synonym chains were long nor was it a “fullness” problem since the master dataset was less than 69% full when disaster struck.

The problem was due to the fact that the records were severely clustered when the very first synonym occurred and DBPUTs serial space searching algorithm is efficient only in the absence of severe clustering.

It should be apparent by now that designers may avoid this cluster collision pitfall by carefully (mathematically) investigating the consequences of their assignment of integer key values together with their choice of master dataset capacity.

Look Mom, No Synonyms

As we have seen, the use of integer keys of IMAGE types I, J or K, coupled with the assignment of key values created by concatenating pairs (or even triplets) of integer subfields whose values are sequential, always leads to these clusters of primaries; a new cluster arising whenever a new value is assigned to any but the last subfield.

There are, however, many situations which lend themselves to the use of integer keys in this manner. In our example, the YY major values form the sequence 71, 72, 73,... and the XXXXX minor values form the sequence 00001, 00002, 00003,...

Notice that, for any particular value of YY, the primary addresses for keys with values YY00001, YY00002,... form a set of (circularly) consecutive record numbers X, X+1,... where X is the primary address generated by reducing YY00001 modulo the capacity C. In other words, they form a cluster of consecutive primaries. I will refer to such a cluster as a “run”.

Notice that each increment of the minor value by 1 merely lengthens the run by 1 and that each increment of the major value marks the beginning of a new run with the minor value restarting at 1.

This works great (i.e., no synonyms) until a new run collides with a pre-existing run. When this happens, you have a performance disaster on your hands as shown in our example.

The question arises: “Is there a way to determine a capacity such that for a specified range of major, intermediate and minor values, the resulting dataset will have no run collisions (i.e., no synonyms) and yet not be unsatisfactorily wasteful of disc space?”.

Some IMAGE evangelists have simplistically “answered” this question by recommending that you select a capacity equal to or greater than the largest key value. We shall refer to this technique as “Method 1".

Applying Method 1 to our 1978 example above would have yielded a capacity of at least 7830000 to hold 240000 entries (8 years worth). Unfortunately, the dataset would only be about 3.0% full which, because of its size, would be very wasteful of disc space.

A better “answer”, which we shall refer to as “Method 2", is to choose a capacity 1 greater than the difference between the highest and lowest key values. In our example, this would equal 7830000-7100001+1 = 730000 and the dataset would be about 32.8% full. Not nearly as bad and yet not great either.

The question then arises: “Is there a way to calculate the smallest capacity which will yield no synonyms?”.

To any mathematician worth his salt, the answer to this question is, “Hell, yes”.

To prove this, note that we have already established that the set of all capacities which will yield datasets without synonyms is not empty. There is, in fact, an infinite number of answers satisfying Method 1.

Next, since capacities are always positive, the set of all successful capacities must have a smallest value since the set of positive integers is “well ordered” and bounded below by zero.

Lastly, since the method of key value assignment is well defined, as is the method of primary address calculation, all that remains to be done is to convert this knowledge to a programmable algorithm which will calculate this smallest capacity.

Clearly, we could simply start with the minimum possible capacity C which equals the product N x L, where N is the “number of runs” and L is the “run length”. By calculating the first address of each run, we can determine whether any two runs collide. If they do, we can increment C and try again. This will ultimately yield a successful value of C which is, for multiple runs, generally far better than the capacity determined by Method 2.

Some months ago, after years of procrastination and wearying of the simplistic and inadequate advice being peddled by others, I finally developed and programmed an algorithm which converges on a minimum C value in a matter of seconds.

We refer to this algorithm as “Method 3".

If I had possessed Method 3 in 1978, I could have applied it to our earlier example of a “pitfall” to achieve a no-synonym dataset

90% full by ***decreasing*** the capacity from 350000 to 265000 yielding the following runs:

Claim numbers	Record Numbers
7100001 through 7130000	210002 through 240001
7200001 through 7230000	45002 through 75001
7300001 through 7330000	145002 through 175001
7400001 through 7430000	245002 through 10001
7500001 through 7530000	80002 through 110001
7600001 through 7630000	180002 through 210001
7700001 through 7730000	15002 through 45001
7800001 through 7830000	115002 through 145001

If the user wanted to maintain 15 years of claims, Method 3 yields a 79% full dataset with a capacity of 565000 and no synonyms:

Claim numbers	Record Numbers
7100001 through 7130000	320002 through 350001
7200001 through 7230000	420002 through 450001
7300001 through 7330000	520002 through 550001
7400001 through 7430000	55002 through 85001
7500001 through 7530000	155002 through 185001
7600001 through 7630000	255002 through 285001
7700001 through 7730000	355002 through 385001
7800001 through 7830000	455002 through 485001
7900001 through 7930000	555002 through 20001
8000001 through 8030000	90002 through 120001
8100001 through 8130000	190002 through 220001
8200001 through 8230000	290002 through 320001
8300001 through 8330000	390002 through 420001
8400001 through 8430000	490002 through 520001
8500001 through 8530000	25002 through 55001

By now it should be clear that, under the right circumstances and with the proper tools, integer keys are superior to hashing keys since we can guarantee a dataset with no synonyms.

Let's look at some other examples.

First, suppose we have an application where access is keyed on “day-of-year”. We can define a master dataset with an I1 key and a capacity of 366 (remember leap year). If we subsequently reference all 366 days of the year, the master will be 100% full with no synonyms. In this simple case, all three methods yield the same result.

Suppose, however, we want to key on “day-of-month”. In this case our I1 (or J1 or K1) key values (in decimal notation) could be in the form MMDD where 1<=MM<=12 and 1<=DD<=31. The smallest value represented will be 101 and the largest 1231.

Method 1 yields a capacity of 1231. Since the dataset has exactly 366 entries, it is thus only 29.73% full.

Applying Method 2, we achieve a capacity of 1231-101+1 = 1131 and a master which is 32.36% full.

Applying Method 3 yields a capacity of 431 and a master which is 84.91% full with no synonyms:

Key Values	Record Numbers
0101 through 0131	101 through 131
0201 through 0231	201 through 231
0301 through 0331	301 through 331
0401 through 0431	401 through 431
0501 through 0531	70 through 100
0601 through 0631	170 through 200
0701 through 0731	270 through 300
0801 through 0831	370 through 400
0901 through 0931	39 through 69
1001 through 1031	139 through 169
1101 through 1131	239 through 269
1201 through 1231	339 through 369

Let's go one step further. Suppose again that you wish to key on “day-of-year” but also to distinguish by year and want to span 10 years. Here we can have a key of type I2 represented by a decimal format of YYDDD (or YYYYDDD).

Choosing the YYDDD format with YY values between 87 and 96, leaves us with a minimum value of 87001 and a maximum of 96366.

Method 1 leads to a capacity of 96366 and a dataset 3.79% full.

Method 2 yields a capacity of 96366-87001+1 = 9366 and a dataset 39.07% full.

Method 3 yields a capacity of 5366 and a dataset which is 68.2% full with no synonyms. Not great, but much better than 39.07% and fantastically better than 3.79%:

Key Values	Record Numbers
87001 through 87366	1145 through 1510
88001 through 88366	2145 through 2510
89001 through 89366	3145 through 3510
90001 through 90366	4145 through 4510
91001 through 91366	5145 through 144
92001 through 92366	779 through 1144
93001 through 93366	1779 through 2144
94001 through 94366	2779 through 3144
95001 through 95366	3779 through 4144
96001 through 96366	4779 through 5144

Now, in anticipation of the next century, let's use a YYYYDDD format for a 20-year span starting with 1986 and ending with 2005, inclusive.

Method 1 results in a capacity of 2005366 and a dataset 0.36% full. Wow!

Method 2 yields a capacity of 2005366-1986001+1 = 20000 and a dataset 36.6% full.

Method 3 yields a capacity of 10366 and a dataset 70.61% full with no synonyms:

Key Values	Record Numbers
1986001 through 1986366	6095 through 6460
1987001 through 1987366	7095 through 7460
1988001 through 1988366	8095 through 8460
1989001 through 1989366	9095 through 9460
1990001 through 1990366	10095 through 94
1991001 through 1991366	729 through 1094
1992001 through 1992366	729 through 2094
1993001 through 1993366	2729 through 3094
1994001 through 1994366	3729 through 4094
1995001 through 1995366	4729 through 5094
1996001 through 1996366	5729 through 6094
1997001 through 1997366	6729 through 7094
1998001 through 1998366	7729 through 8094
1999001 through 1999366	8729 through 9094
2000001 through 2000366	9729 through 10094
2001001 through 2001366	363 through 728
2002001 through 2002366	1363 through 1728

Key Values	Record Numbers
2003001 through 2003366	2363 through 2728
2004001 through 2004366	3363 through 3728
2005001 through 2005366	4363 through 4728

Now let's look at a few keys which involve three subfields such as date fields of the forms YYMMDD and YYYYMMDD.

Suppose we want to span the five years 1989 through 1993 and, for simplicity, ignore the fact that not all months have 31 days. Each of the five "years" will have 12*31 = 372 days and the number of entries will be 5*372 = 1860.

Note that "fullness" percentages based on such 372-day years will always be about 1.6% high since 6 or 7 of the days of these "years" do not exist and hence don't require data entries.

Method 1 requires a capacity of 931231 for a dataset 0.19% full.

Method 2 requires a capacity of 931231-890101+1 = 41131 for a dataset 4.52% full.

Method 3 yields a capacity of 2241 for a dataset 82.99% full with no synonyms.

The charts showing the Record Numbers for the runs of examples involving three subfields take up too much space to include in this paper. If you should want them, please contact me.

If we span ten years from 1989 to 1998, inclusive, Method 3 yields a capacity of 5221 for a dataset 71.25% full with no synonyms. I will no longer bore you with the results of Methods 1 and 2.

Proceeding to the YYYYMMDD format we find that, to span fifteen years from 1989 to 2003, a capacity of 6246 yields a dataset which is 89.33% full with no synonyms. To include twenty years, say from 1989 to 2008, a capacity of 8746 yields a dataset 85.06% full with no synonyms.

Before leaving this section, I would like to show one final chart based on an integer key of the form YYXXXXX where the values of YY span the 15 years 1971 to 1985 and the XXXXX values go from 1 to 25000. Method 3 yields a capacity of 375000 and a dataset 100% full with no synonyms:

Key Values	Record Numbers
7100001 through 7125000	350002 through 1
7200001 through 7225000	75002 through 100001
7300001 through 7325000	175002 through 200001
7400001 through 7425000	275002 through 300001
7500001 through 7525000	2 through 25001
7600001 through 7625000	100002 through 125001
7700001 through 7725000	200002 through 225001
7800001 through 7825000	300002 through 325001
7900001 through 7925000	25002 through 50001
8000001 through 8025000	125002 through 150001
8100001 through 8125000	225002 through 250001
8200001 through 8225000	325002 through 350001
8300001 through 8325000	50002 through 75001
8400001 through 8425000	150002 through 175001
8500001 through 8525000	250002 through 275001

Summary

There are several gurus writing papers or giving talks or publishing books about IMAGE internals, features and idiosyncrasies. Some of these gurus stick to the facts. Others cloud the issue by peddling information which, however plausible and amusing, is either false, imprecise or overly simplistic. I call such gurus “IMAGE evangelists”.

Having been the project leader of the IMAGE development team and co-developer of the primary address calculation algorithm for both hashing and non-hashing keys, I am amused when I hear or read the bad trash being peddled by such evangelists.

Some of the statements and recommendations emanating from these evangelists which this paper has shown to be false, imprecise or simplistic are:

1. “If you use integer keys, choose a capacity at least as large as the maximum key value.”

Simplistic. This advice leads to Method 1. Also ***imprecise*** since the author meant to refer to the maximum “determinant”.

2. “If you use integer keys, IMAGE hashes them to determine the primary address.”

False. Keys of types I, J, K and R are NOT hashed.

3. “Clustering is bad.”

False. When used properly, clustering is the ***virtue*** of integer keys.

4. “If you use integer keys, always choose a prime number for the capacity.”

False. None of our three capacity selection methods care about the “primeness” of numbers.

5. “If you use keys of type R4, IMAGE uses the leftmost 32 bits to calculate primary addresses.”

False. IMAGE uses the rightmost 31 bits.

6. “If DBPUT has to search for a free spot in which to add a new entry (or to move an existing secondary), it inspects the next higher block and then the next lower and then the second higher and then the second lower and continues this ping-pong style of searching until it succeeds.”

False. This statement is pure Science Fiction; truly bad trash. See also either of my previous papers “The Three Bears of IMAGE” or “The Use and Abuse of Non-hashing Keys in IMAGE”. I wrote DBPUT's space-searching routine, so I know it doesn't “ping-pong”.

7. “Don't let master datasets become more than 80 to 85% full.”

Imprecise. This is true only if there is a synonym problem. You may also get good performance up to 95% full if you have a large blocking factor such as 20 or 30. Integer keyed masters can be great even if 100% full.