RMS Performance: Duplicate key chains

Hein van den Heuvel

Overview

Does your application use RMS Indexed files? Do you know what a SIDR is? Do you know what a duplicate key chain is? You probably should, since SIDRs and duplicate key chains can cause thousands of read I/Os as the result of a single record insert. With that, they can have a tremendous impact not just on the application doing the insert, but also on total system performance. Application managers, of course, notice a slowdown over time, and all too often they solve that by throwing more hardware at the problem. But what if you already have the biggest box on the market? Modest file tuning and a convert can help avoid all those read I/Os and restore performance. I have yet to investigate an RMS application that did not have this duplicate key chain problem. Maybe this is because I get called in only for bad cases, or because indeed so many applications have this problem, at least to some degree.

Problem statement

In recent years, HP systems engineers have investigated and improved several applications in large commercial systems where more than half of the resources for an entire system were wasted by updating duplicate key chains. In one case, a simple CONVERT of a single indexed file changed application end-user response time from several minutes to subseconds. In another case, the total system I/O rate was reduced from 1500 I/Os per second to 200 I/Os per second, all without changing application functionality. Why did this happen? Because of duplicate key chains.

Duplicate key chains are a (long) series of (single) linked RMS data buckets containing records all with the same key value and identified only by a single index entry. (The next section clarifies this definition.) Although the problems described in this paper can occur with primary keys, in real-life applications they typically occur with secondary keys. Therefore, the illustrations in this paper show duplicate key chains in secondary keys. It may be that application designers tend to pick unique or nearly unique keys for the primary key.

To establish a frame of reference, the following section describes the internals of an RMS indexed file. Subsequent sections describe how to identify the problem and suggest a number of possible solutions. You will see that some solutions are very easy to implement and can be very rewarding.

For additional information about indexed files and tuning, refer to the *Guide to OpenVMS File Applications* in the OpenVMS documentation set.

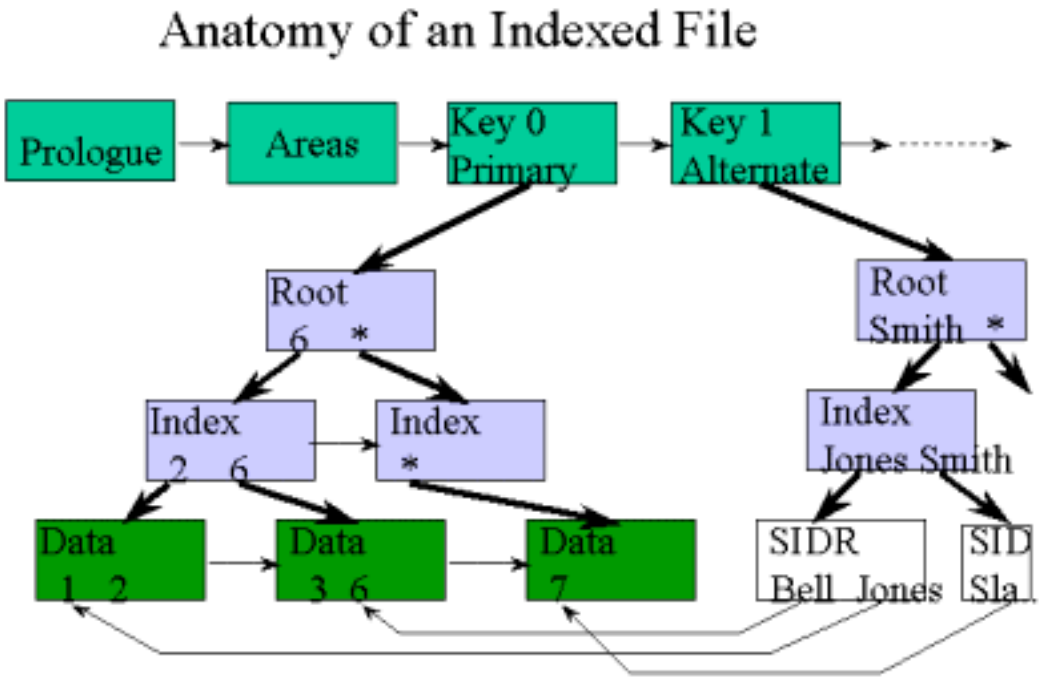
Overview of RMS Indexed File Internals

RMS stores **user data records** (UDRs) in primary-key order in **buckets**. Buckets are the unit of I/O to and from the file. Typically, a bucket contains 5 to 50 records. Records cannot cross bucket boundaries. If an entire record does not fit in a bucket, then a new bucket is added to hold the record. This process is called **bucket split.** You can identify records both by their key and by a **record file address** (RFA). The latter consists of the starting **virtual block number** (4-byte VBN) of the bucket in which the record is stored and the record's ID (2 bytes). The index structure is a balanced b-tree with pairs of key values and VBN addresses.

For secondary keys (referred to in this article as alternate keys), the data records pointed to by their key structure are called **secondary index data records** (SIDRs). A SIDR consists of a key value

(optionally compressed) and an array of one or more **record retrieval vectors** (RRVs). If your application allows duplicates for the key in question, then there will be one RRV for each duplicate value that a key has. Each RRV is 7 bytes in size and consists of a flag byte plus an RFA pointing to the UDR.

The following figure illustrates an indexed file, a number as primary key, and a name as first alternate key.



Here's where the trouble starts

In addition to the file internals described here, RMS follows three rules that work very well in general but that can add up to serious performance problems in certain situations.

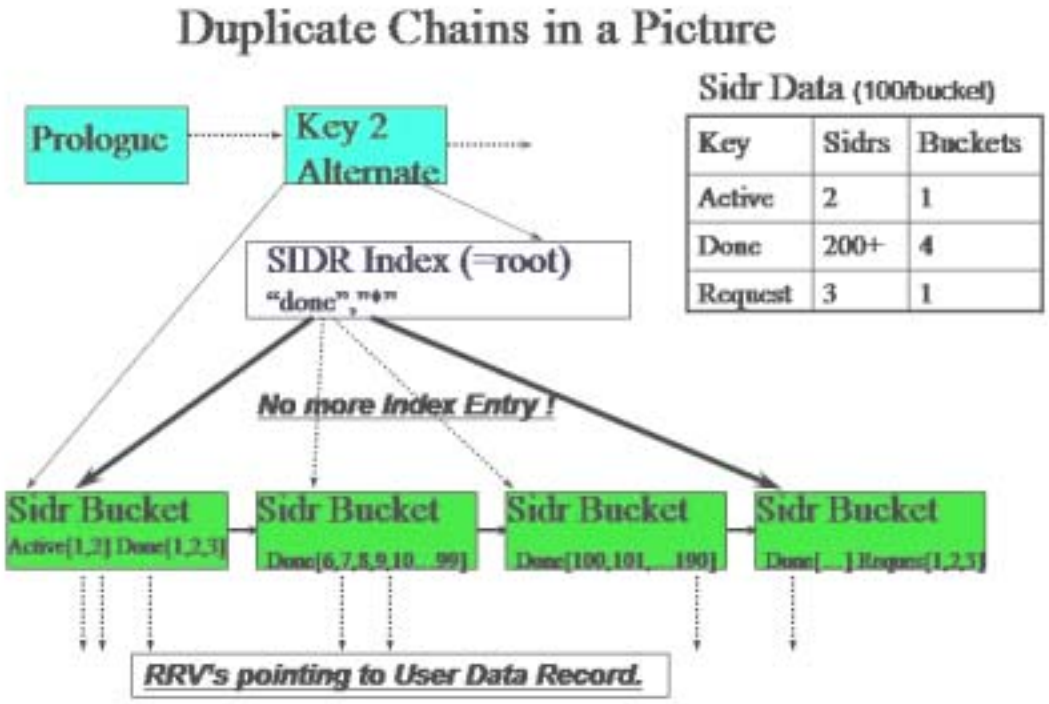
- Duplicate key values are to be added in order of arrival.
- There is only one index entry for a given key value that points to the first bucket that contains a record with that key value.
- If a new, duplicate value does not fit in the target bucket, then a new record is created in a new bucket. That bucket is pointed to by the old target bucket by using the next VBN field in the old bucket header.

What are the implications? Suppose you have an alternate key on an item file to indicate a status. That status can be 'Request', 'Active', or 'Done'. Every new item is inserted into the file with a status of 'Request'. Appropriately, according to rule 1, the item is added to the end of the array of

'Request' item. Next, the item gets processed, and the status key is updated to 'Active'. Eventually, all items are updated to the status 'Done'. Still, according to rule 1, as an item status is updated to 'Done', it is placed at the end of the 'Done' array, keeping the first item that was ever 'Done' as the first RRV in the SIDR.

Now, for the sake of the illustration, assume a SIDR can hold up to 100 RRVs. (In actual files, this is likely to be in the 150 – 1500 range.) The SIDR with the RRVs for the first few 'Done records fits in the same bucket as the SIDR for the 'Active' key. When the number of RRVs reaches more than a few hundred, multiple buckets are needed. When the status for item 200 is updated to 'Done', RMS walks down the index to find the first 'Done' record. RMS determines that this is the first, but not the last, SIDR record for the target key, and it reads the next bucket. RMS continues to read buckets until it reaches the final bucket. It then adds the RRV for item 200 to that SIDR and writes that bucket out to the file. This is the crux of the problem. The series of linked buckets, all with SIDRs for the same key value, is called a duplicate key chain. The system will need lots of read I/Os to perform the single write that it sets out to do. The following figure summarizes this layout.

As long as RMS needs to read just a few more buckets to find the last SIDR for a key, any additional I/Os don't cause a problem. However, when there are millions of Done records with thousands of continuation buckets to store their pointers, it starts to hurt. These thousands of I/Os will bring any system to its knees, no matter how big the box.



Detecting a duplicate key chain problem

Monitor I/O rates

The biggest indicator that an application may have a problem with duplicate key chains is excessive I/O rates for seemingly basic functions. For a multiple-key indexed file insert, you can expect 2 – 4 reads per key and 1 or 2 writes per key, for a total of 10 – 20 I/Os for a typical file. If you observe an average of 100 I/Os or more per insert, then you need an explanation and a fix, preferably an easy fix.

A hot-file tool, combined with the SET FILE /STAT command and the standard MONITOR RMS command can help identify the files to analyze. You should also check out the rms_stats freeware in the RMS tools directory with the OpenVMS Freeware. The rms_stats software reports I/Os per record operation for files with RMS statistics enabled.

Analyze files

A tell-tale sign for the duplicate key chain performance problem is the presence of very short alternate keys (1- 5 bytes) in files with large numbers of records. For example, for a 1-byte field, there can be 256 distinct values in a single byte (0 – 255). Practically speaking, a single-byte key has just two key values; for example: M(ale)/F(emale) or Y(es)/N(o). If a file has a million records and just two key values for a specific index, then there will be at least a half million duplicates on one of those values. Even a 5-byte key (such as a zip code, an item code, or a date) often has but a few hundred frequently used values; again, with a million records, several values will have tens of thousands of duplicates.

The standard tool ANALYZE/RMS/FDL can help identify the problem, but it can also be misleading. Its DUPLICATES_PER_SIDR counter is reset for every new bucket, treating a continuation SIDR just like a new SIDR. When ANALYZE reports DUPLICATES_PER_SIDR=500, this is an average that, to the casual observer, suggests a flat distribution. In reality, though, a single chain of 1000 buckets each with 1000 duplicates each for each single value, and 1000 more SIDRs with a single entry, averages out to 500 but is more accurately represented by a duplicate count of 1,000,000.

A better indication within the ANALYZE stats is a large difference between the number of level 1 index records and the number of SIDR buckets. The difference indicates the number of buckets without an index, that is, those in use by duplicate key chains. The following example shows part of the analysis output for a file with a bucket size of 12:

```
ANALYSIS_OF_KEY 1
        :
        DATA_SPACE_OCCUPIED     1968
        DUPLICATES_PER_SIDR     969
        LEVEL1_RECORD_COUNT     9
```

For 1968/12 = 143 SIDR buckets, there are just 9 index pointers. This difference suggests an average of 13 continuation buckets, or an average duplicate key chain of 12000 records. More likely, there was a single duplicate key chain of 100,000 records spanning 100 or more bucket.

RMS Tune Check Tool

A powerful alternative to the standard ANALYZE is the rms_tune_check tool. This tool is available on the recent RMS tools directory with the OpenVMS Freeware. (An older tool called SIDR, which is similar to rms_tune_check, is available on earlier Freeware CD-ROMs.)

The rms_tune_check tool scans specifically for duplicate key chains larger then a specified threshold. The help text for the tool shows an example script that will help analyze all large indexed files.

The following is sample output from a single real file:

```
$1$DGA3014:[xxxxxxxxx.DATA]xxxxxx.IDX;2
- SIDR: Key 2,   203801 Dups in 704 Buckets for value "11"
- SIDR: Key 3,    99252 Dups in 332 Buckets for value "902       "
- SIDR: Key 4,    24648 Dups in 88 Buckets for value "                "
- SIDR: Key 5,   462729 Dups in 1580 Buckets for value "01"
```

The same program can also report a “top ten” list of duplicate values.

That SIDR data, together with a minimal understanding of the application, makes fixing the performance relatively easy. The following two examples demonstrate a problem situation as found in a real file earlier this year. Once you have read the next section I believe the solution for both cases will become obvious. (Hint for later: think null keys and adding segments.)

```
  Duplicate count, Buckets, Key value
-------------------------------------------------------
   1759748         4045     000000000
        46           1      292164044
        27           1      211941745
        25           1      211147595
        22           1      220995050

  Duplicate count, Buckets, Key value
-------------------------------------------------------
    220461         189      CA
    182738         156      NY
    167123         143      TX
    104023          89      FL
     86792          73      PA
     85524          72      MA
```

How to Solve a Duplicate Key Chain Problem

It is not always possible to solve this performance problem entirely in all cases but more often than not we *can* optimize the performance to a large extent. Here are a few techniques to consider.

Drop the Key

The easiest and most effective solution for this duplicate key chain problem is to drop the key altogether. You laugh; but it might just work for you.

- Maybe you have a key on a stray field in a file where some data (perhaps a back reference or additional date stamp) was going to be stored. However, that functionality in your application

was not implemented and the field was left filled with blanks all along. The blank key never seemed to cause problems when the application was tested with a few thousand records, but now that the file has grown over time to contain millions of record, it is slowing the system down.

- How about that key in the Country or State field of an address? Already the set of values to choose from is limited, and maybe not all are used yet. For example, a company in the United States might do business with 30 out of 50 states, but in reality the bulk of the records are from only a handful of states. Perhaps this key is used only by a weekly batch job that reports business across the states or for a particular state. Consider changing that job to read the whole file by primary key and to filter for the selected state. Alternatively, you could have it to pass records to (callable) sort. Consider putting a process in place to convert the file to add a key with the state field just before it is needed, instead of maintaining it for each record inserted. In all likelihood, there is very little business value in an online lookup (such as, "Find the first customer in California.")

Use a NULL KEY value

The null key is a mechanism RMS has always provided specifically to avoid duplicate key chain problems. Although it is restrictive, it is frequently useful.

The null key value is a double-barreled key attribute you can define with FDL (or XABs for the diehard programmers). First specify NULL_KEY yes, then specify a single null key byte. For example, use NULL_VALUE ' ' for a single space. ANAL/RMS/FDL will report this as follows:

```
KEY X
  CHANGES                 yes
  DUPLICATES              yes
  :
  NULL_KEY                yes
  NULL_VALUE              32
  :
  SEG0_LENGTH             LL
  SEG0_POSITION           PP
```

What's the consequence for RMS? *If* a new record is inserted into the file (via the RMS $PUT operation) *and* all bytes of this key's value are identical, *and* this byte value is that of the one defined as the NULL KEY value (=SPACE=32 decimal), then no alternate index entry is made for that record. This solution works immediately for cases with a single-byte key. It also works for the unimplemented field (as in the stray field example), since such fields often contain a string of space (or null) characters. This solution does not work directly for the Status=Done example or for the State=CA scenario, described earlier. For those cases, you need to adapt the application to replace a single, frequently recurring word by a special, reserved series of repeating characters; for example, DDDDDD instead of Done or XX for a state. This works around the single-byte restriction.

For more information, refer to the EDIT/FDL section on null values in the *OpenVMS Record Management Utilities Reference* manual.

Increase the Bucket Size

Strictly speaking, increasing the bucket size is not a good solution. Rather, it is only an effective workaround that hides the underlying problem. However, it might provide enough time for you to implement a real solution. By making the buckets larger, RMS needs far fewer I/Os to find where to insert a new RRV. There are just as many kilobytes of SIDR data to wade through, but it can be done with much less overhead. A small alternate-key bucket size of just 2 blocks (found in some legacy applications) holds fewer than150 RRVs in a SIDR. A typical (and more appropriate) bucket size of 12 blocks holds almost 900 RRVs. Remember that the maximum bucket size is 63 blocks. Each such bucket can hold about 4600 pointers.

Note that this should be only a temporary solution, since the system is still doing excess work.

Deduplicate the Key Values

First, let's remember that a duplicate key value is not a bad thing in itself. Duplicate keys are, in fact, relatively efficient storage. They tend to be shorter than unique keys and, since all key values are identical, they allow for 100% key compression. As long as all duplicate pointers fit in a single bucket, there is no problem using them. Also, many applications can easily tolerate a chain that spans a few buckets. Only when an application frequently adds duplicate values to an already long list that spans dozens or hundreds of buckets will duplicates cost too much to update.

Key values are deduplicated by adding additional, changing, bytes to a key field. Sometimes this is done simply by increasing the key length. For example, suppose a State field is followed by an adjacent zip code field. By adding a 5-character (or 9-character) zip code to the 2-byte state field key, the combined key clearly does not become unique. And our goal is not to make them unique. The millions of duplicates for California will be reduced to a few thousand and will become manageable.

If no useful adjacent field is available, a key segment can be added. (See the description of xab$w_pos in the *OpenVMS Record Management Services Reference Manual*). In the Status example, you might want to add a Done date or an *MMDD* from a date field to the Status key, thereby reducing the number of duplicates from 99% of the file to the number of records processed every day. (This example assumes that records are purged yearly. If not, a year indicator also might be needed.)

Please note that no data is added to the record; the Status field in the application is not extended. Only the definition of the key that used to map directly, and only onto the state field, changes to point to more data. No change in application code is required.

For other applications, you might be able to add a frequently changing single byte from an unrelated binary field. With a perfect distribution, this divides the number of duplicates by a factor of 256. Even with a skewed distribution, you can still expect an improvement of two orders of magnitude. If only ASCII/decimal bytes are added, then each byte will give only a factor of 10, and you will need 3 or 4 bytes to sufficiently reduce duplicates. Again, in the Status example, you can add all or part of an item code (or similar) field as an additional segment.

Adding a segment can be entirely transparent to the application accessing the file by that key as RMS allows for partial or generic key lookup. The specified key length does not have to match the full key size; rather, it can be equal to the original key size. (See the description of rab$b_ksz in the *OpenVMS Record Management Services Reference Manual*.)

Possible snags associated with adding key segments:

- RMS now honors the order of arrival within the new key definition. It takes both the original field as well as the added segments into account, which can result in a new sort order. This may or may not be relevant for the application.

- Some languages verify that the key specification in the program exactly matches the definition in the file itself when opening an existing file. This verification creates no problem in MACRO, C, or BASIC. Programs written in other languages might need to be adjusted and recompiled.
- Some languages do not support segmented keys.

Reminder: The goal is not to make unique keys. To have some duplicates, even hundreds, is fine.

Why Me, Why Now?

Because it's your turn to be a hero!

There are three main reasons for the occurrence of the duplicate key problem: Neglect, Time, and Fear of All Things New.

As time passes, applications scale to unimagined sizes, and they run with millions of records although they were designed and tested only for thousands. Brute-force hardware solutions can ease the pain, but at a price, and eventually hard limits will be reached. Perhaps it will be an IOLOCK8 VMS internal bottleneck after doing too many I/Os per second. Perhaps the duplicate key chain used to fit completely in the disk controller cache and now no longer fits. You can buy still more cache, which would be the "trusted" solution. However, it is far more advisable and rewarding to change the application so that RMS no longer does all those read I/Os.

Many believe that RMS tuning is "black magic," but it is not. In a few days of effort, most of us can pick up the essentials. If you don't make this effort, then tuning is done only once, shortly after implementation. Others believe that all that's required for RMS tuning is a modest automated procedure with ANAL/RMS … EDIT/FDL/NOINTERACTIVE … CONVERT. Such procedures are great, but they are not sufficient over years of change.

You will have to make changes to get changes, there is always risk involved with that. You will also need buy-in from operations, development, and management. But if you find that missing null-key bit, you could save your company millions of dollars. So use the tools, analyze the data, and make the change. What a nice change it will be!

Notes

1. The RMS tools directory with the OpenVMS Freeware contains several tools that may be useful for RMS work, as well as a PowerPoint presentation about RMS tuning and an Excel spreadsheet to review file design.
2. When dropping an unselective key, forcing an application to read all records by primary key can help avoid I/Os. Unless there is an associated primary-key order to the alternate key, you get 1 I/O per record read by an alternate key. Reading a file by primary key, each single I/O will return a bucket full of records, 5 – 50 records depending on the bucket and record sizes. Thus, even if an alternate key selects only10% of all records, it may still be faster to read all records, since more than 10 records fit in a bucket.
3. Duplicate key chains tend *not* to be cached by global buffers. The subtle reason for this is that RMS targets buckets to the local cache, if they are not in the global buffer cache yet, and are requested with write intent. When the duplicate key problem occurs, there is write intent. On one hand, this is unfortunate, since it would provide a seemingly easy workaround. On the other hand, it would only hide a problem that ultimately needs to be fixed. Furthermore, adding thousands of duplicate key buckets to the global buffer cache can easily exceed the capacity of that cache and make matters much worse for other users of that cache (that is, thrashing can occur).

4. Duplicate key chains are not restricted to alternate keys; they can also happen with the primary key. But they don't, because application designers tend to pick unique or near-unique primary keys.