Do migrating secondaries give you migraines?

F. Alfredo Rego
Adager Corporation
Sun Valley, Idaho 83353-3000 • *USA*
http://www.adager.com

What are migrating secondaries?

Migrating secondaries are associated with the approach (called "hashing") that IMAGE uses to store (and retrieve) entries in master datasets.

What is hashing?

Hashing allows you to quickly access a specific master entry (potentially among billions of entries) according to the value of the entry's search field (or master "key"), regardless of the entry's location in the master dataset.

IMAGE's master datasets are optimized for hashed access. For online applications, which usually serve people who are impatiently waiting over the counter or over the telephone, this kind of quick access provides excellent performance, most of the time. For some caveats, please see Fred White's papers at *http://www.adager.com/TechnicalPapers.html*

Like everything else in the universe, the advantages of hashing come tightly coupled with disadvantages. For instance, we have the very real possibility of synonyms (entries with different search-field values which, nevertheless, hash to the same location). Even though mathematicians have written countless dissertations on the desirability of "perfect" hashing algorithms that would not produce any synonyms at all, every hashing algorithm known today produces synonyms.

Are there other methods for fast storage and retrieval?

If you look on the Web for "indexing", you will be overwhelmed by the breadth and depth of the subject.

IMAGE uses a few specific approaches, listed here in order of sophistication:

- Serial access (DBGET modes 2 and 3). Compact datasets, without "holes" in the middle, perform best under serial access.
- Direct access (DBGET mode 4). This is the quickest way to get to any entry on any dataset (master or detail), provided that you know the exact address. This is the "fundamental" or "atomic" access method used — deep down inside — by all other methods.
- Chained access via doubly linked lists (DBGET modes 5 and 6). A chain consist of entries whose search-field values are equal (for details) or have the same hash value (for masters). Detail chains can be kept sorted via a sort field. You improve chained-access performance by repacking a dataset so that each chain has its entries in contiguous locations.
- Hashing (DBGET modes 7 and 8). Applicable only to masters.
- B-Tree indexing. Applicable only to masters.
- Third-Party Indexing (TPI). Applicable to masters and details. In addition to tree-based indexing, TPI supports keyword retrieval on any field ("native" IMAGE access methods apply only to search fields).

"Hashing" and "indexing" are just two examples (among many) of techniques used by designers of database management systems to try to retrieve (and to store), as quickly as possible, the entries (or rows, or records) of interest for a given query.

Is hashing better than indexing (or vice versa)?

There Is No Such Thing As A Free Lunch (TINSTAAFL) or, if you prefer, *There Ain't No Such Thing As A Free Lunch* (TANSTAAFL).

Hashing comes with migrating secondaries and the need to rehash the entire master dataset when changing its capacity. Fortunately, the issue of migrating secondaries turns out to be a non-issue most of the time. For instance, secondaries don't get migrated to the moon — they usually get relocated within the same memory cache and the cost of doing so is negligible.

Fortunately, also, most synonym chains are short (in the single digits). So, even with synonym-chain chasing, you will probably stay within your memory cache if your "data locality" is good.

The tree-like structures associated with indexing come with node splitting, balancing, and reorganization. "Bushy" trees perform better than "stringy" trees. Trees with fewer levels perform better than "tall" trees. With tree-based indexing, you also duplicate the search-field data (you keep one copy in the dataset and one copy in the index). It's a jungle out there!

The minimum number of disc accesses for tree-based indexing is two (one for the tree index, one for the dataset). The minimum number of disc accesses for hashing is one (and, most of the time, this is all it takes).

A tree has several levels (more or less, depending on statistical considerations) that must be traversed via pointers.

So, is there a perfect way to store and retrieve database entries? Unfortunately, the answer is no.

What is a synonym chain?

A synonym chain is IMAGE's method of managing synonyms (members of the class of entries whose search-field values hash to the same location).

An entry which hashes to an empty location (or to a location temporarily occupied by a secondary) becomes a ***primary***. A primary "owns" its location and can't be evicted (i.e., it never migrates to a different location).

An entry which hashes to a location that is already occupied by a primary becomes, *ipso facto,* a ***secondary***. A secondary has to find a temporary location elsewhere and is always subject to "eviction" (if a new entry hashes to the secondary's temporary location) or "promotion" (if the primary that currently occupies the secondary's proper location gets deleted and this secondary becomes a primary by taking its place).

The primary has an important job: it serves as the synonym chain's head, which contains information regarding the total number of synonyms in the chain (if any) and keeps track of the first and last members in the chain. A secondary only has to worry about its predecessor (if any) and its successor (if any).

Why do some secondaries migrate?

Some secondaries migrate because primaries (new or old) have top priority.

Fortunately, not all secondaries migrate. The rules for migrating secondaries are quite clear:

- Whenever a new legitimate primary hashes to some location which was previously "borrowed" by a secondary, IMAGE migrates the secondary elsewhere, to make room for the new primary.

- Whenever IMAGE deletes a primary with secondaries, it migrates the first secondary and promotes it to the primary location, to take over the ChainHead functions.

Are all secondaries "bad"?

No. There are "good" secondaries and "bad" secondaries, according to their demands for a very valuable computer resource with direct impact on performance: disc access.

Good secondaries are those which we may access directly in memory buffers, without having to go to disc. Bad secondaries are those which force excessive disc activity.

IMAGE takes excellent advantage of the optimized disc-access methods provided by MPE/iX. It is a good idea to invest in as much main memory as possible, to minimize time-consuming trips to/from disc.

Messy synonym chains, with entries scattered all over, will probably contribute numerous bad synonyms. Cleanly-packed synonym chains, on the other hand, may contribute good synonyms which will be, for all practical purposes, equivalent to primary entries. Intra-memory operations are, after all, significantly faster than disc operations.

Under any circumstances, short and tidy synonym chains are much better than long and messy synonym chains. Use Robelle's HowMessy (available in the Robelle and Adager installation packages) to get a good view: *http://www.robelle.com/smugbook/howmessy.html*

Why do secondaries (migrating or not) give you headaches?

There are three fundamental operations on entries:

- addition
- deletion
- finding

We want to do these operations as quickly as possible. Therefore, we want to avoid excessive disc accesses. Unfortunately, secondaries (migrating or not) tend to increase disc activity. As a consequence, the response time for online applications may deteriorate and the throughput of batch jobs may decline.

Quick review: Adding and deleting entries.

If a new entry's appointed location is vacant, we are home free. We just add the new entry on the spot and mark it as "occupied". The new primary entry will have tenure for life.

If a new entry's appointed location is already occupied, we must do a lot of work. There are two possibilities:

- The current occupant *is* at its appointed place and, since it arrived before the new entry, it has "seniority" and tenure

for life. The new entry, sadly, becomes a synonym and we must place it elsewhere as a secondary, linked to the primary by means of a synonym chain. Before we add the entry, though, we must make sure that it will not duplicate an existing entry's search-field value. We must, therefore, scan the whole synonym chain before we can add the new entry. If the synonym chain is long, this may take a while.

- The current occupant did *not* hash to this location but was placed here as a secondary (subject to migration). Sorry: its time to migrate has come and must go elsewhere. After we evict the secondary, we can place the new entry in its appointed spot, as a primary.

Notice that the innocent-sounding expression "must go elsewhere" is easier said than done. Finding the "elsewhere" may take a long time if the dataset has a long, uninterrupted cluster of occupied entries. This usually happens if the dataset is quite full or if the distribution of search-field values pushes the limits of IMAGE's hashing algorithm.

Having found the "elsewhere" does not guarantee tenure there, since any secondary is subject to migration in the future. If we add a new entry which hashes to a spot occupied by a secondary, we must migrate the secondary elsewhere. If we delete a primary with secondaries, we must move the first secondary into the spot previously occupied by the deleted primary (since we *must* have a primary). IMAGE, under extreme circumstances, may spend a significant amount of time chasing its own tail while migrating secondaries all over the place.

Quick review: Finding existing entries.

Even in a static dataset (where we never add or delete entries), we may have performance problems when we simply want to find an existing entry.

If the entry happens to be at its appointed location, we are in good shape. If the entry is not at its appointed location, there are two possibilities:

- The appointed spot is empty, in which case we know, immediately, that our entry does not exist (this is a valid "entry not found" result in a "find" task).
- The appointed location contains some other synonym entry (which happens to hash to the same spot as the entry which interests us). Since this entry is the primary in a synonym chain, it keeps track of the number of synonyms. If there are no synonyms, we know that our entry does not

exist. If there are synonyms, we must scan the synonym chain until we either find our entry or exhaust the chain. In either case, we may have to go to disc more than once (depending on the length of the chain, the messiness of the chain, the dataset's blocking factor, and so on).

What can you do about nasty secondaries?

If you are a theoretical type, you can spend endless pleasant hours dedicated to the fine art of figuring out the ideal mix of capacity, percent-full, search-field value distribution, and so on. This is a worthy endeavor, of course. But the sad thing is that your dataset, most likely, is *dynamic*. The minute you add or delete a few thousand entries, most of your glorious conclusions become invalid.

If you are a practical type, you might just as well accept reality, bite the bullet, and periodically repack your master datasets to balance the synonym load (just as you periodically repack your detail datasets, your disc drives, or the drawers in your desk). If your dataset is static, you are in luck: you just repack once.

As another alternative, you can enable your dataset for master dynamic capacity expansion (MDX). See Fred White's paper at *http://www.adager.com/TechnicalPapersHTML/DDX-FW.html*

If the widths of master ventilation shafts (free areas reserved for future secondaries) are narrow and suffocating, you may consider changes in the dataset capacity which, together with dataset repacking and MDX, may improve the situation.

An ideally-packed master dataset allows sufficient ventilation space for DBPUT's sake (so that it does not take forever to find an empty spot for a new secondary or for a migrating old secondary) without having so much empty space that a serial scan (or a backup) will take forever.

Furthermore, the ventilation space has to be intelligently distributed throughout the entire dataset.

The good news and the not-so-bad news.

Fortunately, it turns out that repacking a master dataset is a very efficient operation. Smart repacking of a master dataset takes only slightly longer than an optimized serial dataset scan.

Unfortunately, repacking does not seem to be an *a priori* activity. We can only repack master datasets *a posteriori*. We must repack *periodically*, and (unless we quit adding or deleting entries) we must keep repacking *ad infinitum*.

The same applies, of course, to tree-based indexing methods. The price of good performance is constant fine-tuning.