More Soup?

DRAFT 5

Michael S. Engber
Apple Computer - PIE Developer Technical Support
Copyright © 1993 - Michael S. Engber

This article was (will be) published in the January 1994 issue of PIE Developers magazine. For information about PIE Developers, contact Creative Digital Systems at CDS.SEM@APPLELINK.APPLE.COM or 415.621.4252.

Introduction

Last month's issue had an article, "Soup's On," which covered the basic things you should know about creating and using soups. This article focuses optimizing soup usage. I assume that the reader is already familiar with the chapter on using soups in the Newton Programmer's Guide and has some experience writing Newton applications.

The Nature of Soups

Soups on the Newton store flattened versions of the entry-frames you work with in NewtonScript. When you retrieve an entry from a soup, you get back a frame which is a cache of the actual data in the soup. Calling EntryChange causes the cache to be written out to the soup. Calling EntryUndoChanges causes the cache to be refreshed from the soup.

Retrieving an entry from a soup allocates a frame in the NewtonScript heap. This takes time and heap space. The current soup implementation optimizes this operation by not creating a complete frame for the entry until you access one of its slots. Once you access a slot, it brings the whole entry into memory. An obvious area for future optimization would be to bring slots into memory on an as-needed basis.

The above information may not be new to you. If you didn't know it already, after reading about soups in the Newton Programmer's Guide and a few minutes of reflection, you probably would have guessed it. What you may not have thought about, however, are some of the implications of the implementation of soups.

Organizing Your Data Across Multiple Entries

One implication of the way soup data is retrieved is that as you loop through your soup looking at all the entries, there is overhead (in both space and time) for allocating entry-frames in the NewtonScript heap. The larger the entries, the larger the overhead. If you've created a soup whose entries each contain a large amount of data, you might have noticed this already.

One way you can speed things up is to store your data in multiple soups. The built-in Calendar application uses five different soups to store its data. This is probably more soups than most applications need. I think two soups should suffice for most purposes. Call one soup the query-soup and the other the data-soup. The basic strategy is to keep the entries in the query-soup small. They hold just enough information for the queries you commonly make, plus a reference (EntryUniqueId) to an entry in the data-soup which contains the bulk of the information.

This allows you to quickly access the query-soup and pay the overhead for retrieving large amount of information only when you actually retrieve it from the data-soup.

Actually, there is no reason you have to use two soups to take advantage of this technique. You can keep both query-entries and data-entries in the same soup. If only the query-entries have the slots used for indexing, then your queries will ignore the data-entries.

The exact details of what slots go where will depend on your particular application. It will probably take some experimentation on your part to come up with an optimal organization. Another consideration is whether or not you're planning to allow your users to edit their soup data using Newton Connection. If you are, splitting your data across multiple entries may make editing the data more difficult, perhaps impractical.

Bear in mind, this type of optimization is useful only when you need to access a large number of soup entries to find relatively few entries that you are actually interested in. Whenever possible, it's better to handle this problem by using indexes. Often, proper use of indexes can allow you to construct queries that only examine the entries of interest.

Use Your Indexes

Another point to keep in mind when using soups is: take advantage of your indexes whenever possible. This seems obvious enough, but I've seen lots of programmers carefully design their soups with indexes and then fail to take advantage of them when writing queries.

StartKey vs ValidTest

A common error is to be seduced by the flexibility of using a validTest. The temptation is to write a query that relies solely on a validTest to decide which entries to return. For example, a validTest like the one below works – it returns the desired entries. However, it totally negates the benefit of having an index on slot x.

```
validTest: func(e) e.x >= "C" AND e.x <= "D"
```

If you create a cursor using this valid test, and naively use a "`while cursor:Entry()`" loop to traverse your soup, you'll end up visiting every single element in you soup. If instead you had used a startKey and an endTest, as show below, you would only have visited the entries of interest. You can't hope to do any better than that.

```
startKey: "C",
endTest: func(e) e.x < "C" OR e.x > "D"
```

Of course, not all queries are going to be as simple as these, but you should use a startKey and an endTest whenever possible. This makes the difference between an O(N) search and an O(logN) search. You may recall from your introductory data structures course how big a difference this can be – one million versus six ($\log_{10}{10^6} = 6$).

GotoKey

A related point is using GotoKey instead of Move. Move lets you move the entries one at a time. The one at a time part is actually hidden from you since Move accepts an arbitrary integer offset. Using GotoKey moves the cursor directly to the desired entry in one shot – a single O(logN) search.

Consider the following two code fragments for advancing a string-index based cursor to the last entry in a soup:

```
cursor:Move(0x1FFFFFF);
while cursor:Next() do;   //make sure we're off the end
cursor:Prev();            //back up one - to the last entry

cursor:GotoKey("ZZZZZZ"); //pick a "large" key
while cursor:Next() do;   //make sure we're off the end
cursor:Prev();            //back up one - to the last entry
```

These code fragments may not look very different, but for a large soup, the second one will be significantly faster.

Avoiding Explicit Search Loops

You can often avoid writing an explicit search loop by using startKeys, endTests, and validTests. This reduces the size of your code and is more efficient since looping is handled by the soup implementation. There are two examples of this in the section titled "Uniquely Identifying Entries." The following code is from one of these examples.

```
Query(soup,{type:      'index,
            indexPath: '_uniqueId,
            startKey:  entryId,
            endTest:   func(e) EntryUniqueId(e) <> entryId,
            validTest: func(e) EntryStore(e):GetSignature() = storeSig,
           }):Entry();
```

This code evaluates to either `nil` or to the entry whose id is `entryId` and whose store signature is `storeSig`. No explicit search loop is required – only a single Entry message.

When using startKeys, endTests, and validTests be sure to think carefully about your queries. Because the underlying search is handled by the soup implementation, it's much easier to inadvertently do things inefficiently than if you were writing the search loops yourself.

As was pointed out in the previous section, a common error is omitting the startKey and consequently ignoring the benefit of an index. A related error is omitting the endTest. Forgetting an endTest can force a query to do unnecessary work – continuing to search after all entries of interest have been returned. For example, consider the above query without the endTest. If there was no entry matching the criteria, the query would examine every entry in the soup whose id was greater than or equal to `entryId`. By using the endTest, it examines only entries whose id equals `entryId` – at most, two.

Simulating Joins by Using Multiple Indexes

Currently, you can only query soups using one index at a time. Sometimes you want queries involving two or more indexes at time. In relational database jargon, this is termed performing a join.

I will present a technique that is applicable only to two indexes and, even then, is practical only if one of the indexes is over a small set of values. We will call the two "conceptual" indexes of interest a director-index and a date-filmed-index. The director-index only allows the values moe, larry, and curly, indicating which stooge directed the episode. The date-filmed-index uses integers to represent the date the episode was filmed. The kinds of queries we will be interested in will be ones like: "retrieve all the episodes directed by moe in the order they were filmed."

The basic idea is to build an actual soup index for each of the distinct values of director-index. Each entry in the soup will have either a moe-slot, a larry-slot, or a curly-slot. The content of this slot will be the date the episode was filmed. Our index specification will look something like:

```
[{structure: 'slot, path: 'moe,   type: 'int},
 {structure: 'slot, path: 'larry, type: 'int},
 {structure: 'slot, path: 'curly, type: 'int},
]
```

To "retrieve all the episodes directed by moe in the order they were filmed," we simply create a cursor using the code shown below. Only entries with moe-slots will be returned by this cursor. They will be returned ordered by the contents of their moe-slot – the date on which they were filmed.

```
moeCurs := Query(stoogeSoup,'{type: index, indexPath: moe});
```

You will note that we never actually built a unified date-filmed-index. Instead, we distributed this information over three separate indexes. This means we can't do a query to retrieve all episodes in date-filmed-index order without first giving each entry another slot, call it dateFilmed, and then building an index on the dateFilmed slot. There is nothing preventing us from doing this, it will just require more space – an extra slot in each entry.

This is not the most general purpose technique. We might be able to use it for "Snow White and the Seven Dwarves," but we'll run into problems with "101 Dalmations." It's something to keep in mind for use in suitable situations.

To give credit where credit is due, this section was inspired by a posting on the Internet by Kent Borg. The posting specifically addressed implementing filing (which categorizes user data in one of up to thirteen categories). As you can see, the technique has some wider applicability.

Using Identical Frame Maps

Storing entries in a soup requires storing the contents of their slots as well as information about the structure of the slots themselves. The information describing the structure of the slots in a frame is called the map. Soups save space by reusing maps among entries with the same structure. The issue of shared maps has implications for general NewtonScript programming. This section will only discuss it with respect to minimizing soup size.

Obviously, in order for two entries to share the same map they must have the same set of slots. However, the current of soup implementation also requires entries that share a map to have their slots in the same internal order. This means that even if your entries all use the same set of slots, it's possible to have a different map stored for each permutation of slot ordering.

The internal ordering of slots in a frame is something that programmers normally don't think about. NewtonScript purposely leaves the details of slot ordering unspecified – open to changes in the future. Programmers are not supposed to write code that relies on slot ordering and there are no functions provided to manipulate slot ordering. So it may seem unreasonable to suggest optimizations the rely on slot ordering. However, the way in which you create a frame gives you some control in the area of slot ordering. If you create your entry frames in an identical way, you can assume they use the same slot ordering.

For example, if you use the following line of code to create all your entries you can be sure that each entry will use the same slot ordering.

```
newEntry := {slot1: x, slot2: y, slot3: z};
```

You can achieve the same effect by cloning a dummy frame and then filling in the slots individually as show in the following code:

```
newEntry := Clone('{slot1: nil, slot2: nil, slot3: nil});
newEntry.slot1 := x;
newEntry.slot2 := y;
newEntry.slot3 := z;
```

If the set of slots needed varies from entry to entry, you might consider giving all entries the same set of slots and filling in unused slots with nil. Indexed slots containing nil will be ignored by indexes. (Note that this last point is only true for the last system update (1.05). Prior to that, using nil in an indexed slot would cause errors.) You have to weigh the benefits of map sharing against the cost of the extra slots.

For instance, if you know that over time, all the entries in your soup will end up with the same set of slots, it's probably better to give each entry this set of slots when it's initially created. On the other hand, if the entries in your soup tend to fall into a few different categories, with a different set of slots for each category, then it's wasteful to force every entry in the soup to use the same set of slots. You can still achieve space savings by ensuring that entries in the same category share a map.

Generally speaking, most programs create their soup entries using the same few lines of code – much like the ones shown above. You only need to worry about the issues in this section if you build up entries in a piecemeal fashion, using a large number of different of slot sets or slot orderings.

Alternatives to Soups

Remember that soup data is stored as frames. Consequently, there is a certain amount of overhead associated with each soup entry. For example, using a soup to store a list of <zip-code, state> pairs (e.g. < 95051, CA >) would be a poor choice. You may be better off storing large, read-only data sets in a slot in your base view as a single array, frame, or binary object. Information stored this way will be compressed along with your package and will not be brought into the NewtonScript heap when it is accessed. The primary disadvantages of such a scheme are that the data will be read-only and that you won't have any of the nice conveniences that soup queries provide.

If your application uses a large initial data set and allows additions by the user, you might consider a hybrid approach: Keep the initial data set in your base view and use a soup only for the user's additions.

If you decide not to store your data in a soup, here are some points to think about:

• If you keep an array sorted, you can use binary search to find elements quickly. Once again, this is the difference between an O(N) and O(logN) search.

• Don't be too quick to discount frames as your data structure. Slot lookup is fast. A binary search is used for large frames (as soon as there are enough slots to make it faster than a linear search). Also, remember that slot names (symbols) can consist of any characters if you use vertical bars to escape them. For example, `{|1|: "Reg"}` is a legal syntax for a frame.

• If you're storing a lot of repeated strings, consider using symbols instead. This way, only one copy of the actual string will be stored in your package – all the symbols will reference it. Again, remember you can use vertical bars to allow arbitrary characters in your symbols.

• Storing your data as a binary object can avoid some of the overhead of the array and frame data structures. You'll have to use the various ExtractXXX functions to retrieve your data. If strings are part of the data in your binary object, extracting them with ExtractCString or ExtractPString will create a string object in the NewtonScript heap. In general, binary objects may let you store your data more compactly, but you'll pay more to access it.

Uniquely Identifying Entries

Sometimes soup entries need to refer to each other. Consider a soup whose entries represent people. One person may need to refer to another person in the soup – for example, their father. You may recall that when a frame is added to a soup, a deep copy is made. This means if the father slot contains a reference to the father's frame, you would end up copying the father's entire frame into the son's entry. This is undesirable. You want each person to be represented by a single entry in the soup and for other entries only to reference that single entry.

If you knew, for example, that every person in the soup had a unique social security number you could store the father's social security number and then be able to retrieve the father's entry. Some soups you design will naturally have a slot that uniquely identifies an entry. If the soup you're working with doesn't have such a slot, there may not be a need to waste space by artificially creating one.

Every soup entry has a unique id that uniquely identifies it within its soup. This id is stored in a `_uniqueId` slot, but you should access it with the EntryUniqueId entry function. Every soup has an index on the `_uniqueId` slot. It's important to know that these id's are not unique across union-soups, they are unique only within a single soup.

In the "Soup's On" article I said that "since applications normally use union-soups, the `_uniqueId` slot isn't of much practical use." I should have said "the `_uniqueId` slot, by itself, isn't of much practical use." You can uniquely identify an entry in a union-soup using its id plus the signature of the store on which it resides. Given an entry, you can obtain this information using the EntryUniqueId and EntryStore entry-functions and the GetSignature store method. The following code illustrates getting this information.

```
fatherStoreSig := EntryStore(fatherEntry):GetSignature();
fatherId := EntryUniqueId(fatherEntry);
```

Given an id and store signature, you can retrieve the entry either by getting it directly from its store (using GetSoup, bypassing union-soups) or by examining the store signatures of all the entries in the union-soup with that id.

The following function retrieves an entry by getting it directly from its store.

```
func(storeSig,soupName,entryId)
begin
 pos := ArrayPos(GetStores(),storeSig,0,func(id,store) id = store:GetSignature());
 if pos then
  begin
   soup := GetStores()[pos]:GetSoup(soupName);
   if soup then
    Query(soup,{type:      'index,
                indexPath: '_uniqueId,
                startKey:  entryId,
                endTest:   func(e) EntryUniqueId(e) <> entryId,
                validTest: func(e) EntryUniqueId(e) = entryId,
               }):Entry();
 end;
end;
```

The following function retrieves an entry by examining all the entries a union soup with the specified id.

```
func(storeSig,soupName,entryId)
begin
 local soup := GetUnionSoup(soupName);
 if soup then
   Query(soup,{type:      'index,
               indexPath: '_uniqueId,
               startKey:  entryId,
               endTest:   func(e) EntryUniqueId(e) <> entryId,
               validTest: func(e) EntryStore(e):GetSignature() = storeSig,
              }):Entry();
end;
```

Notice both of the above functions use endTests and validTests instead of explicit search loops. Therefore, a single Entry message is sufficient to return the entry of interest or `nil`.

These functions are only meant to illustrate the technique I've been describing, not as code you should blindly paste into your application. They are not efficient for retrieving large numbers of entries – they creates a cursor each time they're called. It would easy to generalize these functions to accept arrays as arguments and return arrays of entries. Depending on your particular application, you can probably make other optimizations.

Before you start adapting this technique to your application, you should consider its major drawback. Most applications let users move their data between stores using the "Move to card" and "Move from card" items in the routing menu. If you plan to support this feature in your application, and you probably should, then any saved store signatures and entry id values will be incorrect if the user moves data. (Note that when an entry is moved to a new store, its unique id must be changed if it's already in use by an entry in the new store.) You could try to fix up all the references to an entry when it is moved, but this may prove impractical unless you have a quick way to find all the references.

Indexing on Modification Time

To support date finds, you need to time stamp your soup entries. There is no need to create a special slot for this purpose. Soups already maintain this information.

In addition to the `_uniqueId` slots, entries that have been modified have a `_modTime` slot. This slot contains the time the entry was last modified as the number of minutes since January 1, 1904. Normally, you should access this value with the EntryModTime entry-function.

To query your entries in the order of their modification dates you should build an index on the `_modTime` slot. Remember, entries don't automatically have a `_modTime` slot; only entries that have been modified have one. Entries without `_modTime` slots will be skipped by a query on the `_modTime` index so you have to ensure that every entry in your soup has one. You can do this by creating the slot yourself in the frames you pass to AddToDefaultStore or you can immediately call EntryChange on entries after you add them.

Summary

Newton's soups provide a lot of functionality for free. The down side of this is that people often fail to think about the nature of the underlying soup implementation when they are designing their data structures and applications. Understanding these issues and designing your code appropriately can make a huge difference in the performance you realize from soups.