Data Structures
Storing and Retrieving: Part 1 of 3:

by J. Christopher Bell, Newton Developer Technical Support
llama@apple.com, http://www.doitall.com

Introduction

These articles focus on information useful to those who have mastered the basics of Newton data storage APIs. They assume that the reader is already familiar with NewtonScript and Newton data storage concepts in the Newton Programmer's Guide (NPG) and has some experience writing Newton applications.

The series consists of three articles entitled, “Data Structures,” “Entry Caching,” and “VBOs.” The articles include some implementation-level details with which you can optimize and design your own applications. The first article in this series focuses on internal soup data structures and how to use them.

Under the Hood

By now, you have already used soup methods like `soup:AddXmit(entry)` and `unionSoup:AddToDefaultStoreXmit(entry)`. However, you may not know what goes on “under the hood” in the Newton OS to store and retrieve soup entries.

Without any extra data structures, finding an entry in a soup would be as inefficient as searching for a specific business card after a fan blew your business cards around your office. You might not be happy if you had to look at almost every card before finding the right one.

The “pick items in a random order” algorithm would be especially slow in the Newton OS because reading a soup entry from the store has a high overhead. While writing a NewtonScript frame to a soup, the frame is flattened into a binary object, similar to the process used in the `Translate` global function. While reading in a soup entry, the system reads the flattened frame from the store and unflattens the frame into NewtonScript memory. Because of the unflattening code and the memory management required to create the NewtonScript objects, reading in a soup entry is slow. The “Entry Caching” article later in this series will discuss the details of reading in entries.

The high overhead for reading in soup entries wouldn’t be as much a problem if we read in an entry only when it was the “next” item we wanted. You might be wondering why we cannot read in all the entries into memory and sort them in place, like you might do on a desktop computer. Due to cost and mobility requirements, current Newton devices don’t have enough memory to hold all the data of an average-sized soup. Instead, creating a cursor finds only the first entry. If you send cursor messages like `cursor:Next()` and `cursor:Prev()`, the system searches for adjacent entries.

In order to help your application minimize reading in entries, Newton soups maintain separate data structures. The system stores these data structures *in addition* to the entries, and these data structures do not order or modify the entries themselves. These data structures are called indexes and tags.

Indexes: from AA to zz

The most useful soup data structure is the index. An index provides quick access to soup entries as well as a means of retrieving entries in a defined order. A designated value from each item, called the index key, defines the item's sort ordering. For soup indexes, the key value could be derived from a slot value or multiple slots within the entry. Your code can use the index to find individual soup entries quickly (if you know the key value) or iterate through entries in a sorted order without reading in unneeded entries.

Most Newton applications create indexes when the soups are created. For instance, a bank check application might specify an integer index on a `checkNum` slot. This means that the soup entry's `checkNum` slot contains an integer that provides the key value for that index. This index allows the application to retrieve the entries in check number order (ascending or descending) or quickly find a check with a specific `checkNum`.

The system maintains each index as soup entries are added, deleted or changed. Or, you could add or delete indexes to soups when entries are already in the soup. If you create indexes for soups that already have entries, the system must iterate through all the entries, examine the key values, and place them in the index.

Note that some applications that add many entries achieve better performance if they do not set up indexes in advance. If an application receives data from a remote server and must add many items to a soup, updating the index information when adding every entry can slow down the connection with the server. If connection time is expensive, an application could set up a soup with no indexes, add many items, and then add the index after the connection is complete. Even without expensive connection times, adding many entries to an empty soup will be faster if you create indexes after adding all the entries.[1]

Because the system stores index information separate from the entries themselves, we may create multiple indexes to represent different sort orders. *Multiple indexes* are useful if you want to retrieve the data in two or more different sort orders.

If you wanted a single sort order based on *multiple slots*, you'd use a multi-slot index. For example, you could retrieve employee names in alphabetical order by last name (the primary key), but use the first name (the secondary key) to decide ordering of employees with the same last name.

Multi-slot indexes do not define multiple indexes. Instead, the index extracts the key value for the index from two or more slots. Although this is oversimplifying the implementation, you can think of it as creating concatenated values for use as the key value for that entry. For instance, if the entries use separate slots for last names and first names, the system extracts both slots from each entry, creating single index key values like "Simpson*Bart" which define the sort order.[2]

A common misconception is that this is equivalent to multi-index queries, which are not currently supported. Multi-slot indexes define a single sort order. When you use a multi-slot

---

[1] If the soup already contains entries, based on the number of entries already in the soup, it may or may not be faster to 1) remove the index 2) add many entries, 3) re-add the index. Or, you could add new entries to a temporary soup (with no indexes) during connections, and move the entries to the indexed soup afterward (this might take longer overall, but it might allow shorter connection times if that was important for your application)

[2] You can also define directionality for multi-slot indexes. For instance, you could retrieve employee data sorted by last names alphabetically (ascending), and when there is a tie, show the employee salary sorted reverse alphabetically (descending).

index, you must create the entire index with the proper sort order specification before querying with that index.

An example of this misconception is trying to make an efficient query with the last-name-first index described above, retrieving all people with *first* names beginning with the letters A through C. You can think of the names as listed on a sheet of paper sorted by last name and then first name. Because there is only one sort order, the people with first names beginning with D through Z are interspersed throughout the sort order. The many undesired entries must be examined and then skipped over using a filtering method (which we will discuss later in this article), which is inefficient.

Indexes are B-trees

Indexes are implemented in a balanced tree structure called a B-tree. The important thing to know about B-trees is that finding an entry based on a key value is described as O(log n) in big oh notation (pronounced "order log n"), where n is the number of indexed entries. Moving from one item to the next item in sorted order can be done in O(1), or "constant time." For more information about big O notation and B-trees, see the cross-references section for more information (Baase, Sedgewick).

For instance, let us compare it to a hypothetical "brute force" index designed to minimize reading in entries. Imagine we created a simple index by storing each entry's key value and the soup entry unique identifier (unique ID) in elements of an unsorted array. We could iterate over the unsorted array for a particular key value but we might search most of the unsorted array for the desired key value. In comparison to this "brute force" index search time of O(n), a real soup index requires touching only a small fraction of nodes in O(log n) time, and subsequent searches for the next or previous entry in sorted order are O(1) time.

Note that for multi-slot indexes, search speed is approximately the same as for single-slot indexes. Multi-slot indexes use the same structure as for single-slot indexes, except that the system takes the key values from multiple slots in the entry.

When Entries Participate In Indexes

For single slot indexes, if the key value is non-`nil`, the entry will participate in the index. If the key is `nil` or not present, that index will not reference that entry.

For multi-slot indexes, the entry will participate in the index if any of the subkeys are non-`nil`. If an entry is in a multi-slot index but a subkey is `nil`, it will sort before any otherwise identical item with a non-`nil` value in that subkey. For instance, in the multi-slot index for the employees soup described above, the following entries would appear in this order:

```
{last: nil, first: "Madonna"} // nil value for primary key
{last: "Simpson", first: "Bart"}
{last: "Smith"} // nil value for secondary key
{last: "Smith", first: "Abigail"}
```

Urban Legends About Indexes

Here are few common misconceptions about indexes:

1) *Indexes do not order the entries themselves.* Soup entries themselves are stored as flattened frames in no defined order. The system copies the key values from the entries into the index structure separate from the soup entries themselves.

2) *Indexes are not free*. Indexes take time to maintain when entries are added, deleted, or changed. Indexes take space on your internal or PC Card store. Indexes are not automatically added for all slots in soups; you must add one index for each desired sort order.

3) *Indexes do not provide instant access to soup entries*. Finding an entry based on its key value takes O(log n) time (although it takes O(1) time to find the next/previous entry in the index). Note that if you are not filtering out unneeded entries in the index (see the next section), index operations are very fast, low-level operations in comparison to calling NewtonScript functions or reading in entries.

4) *Multi-slot indexes are not multi-index queries*. A multi-slot index must already be created for a particular set of slots before querying the index. When querying that index, there is only *one* sort order and only *one* key value derived from multiple slots.

Filtering: All You Ever Think About is Sets

Some applications want to query using an index but retrieve entries only if they satisfy certain criteria. You might say that you are looking for a certain “set” of entries in the index. For instance, you might want to find the set of all employees who “are married, don't have children, have the ABC health plan, but don't have a dental plan." You could add an index for every possible set-based query, but it would cost both speed and store space to maintain those indexes. Instead of using indexes to filter out unwanted items *and* define an order, you can use an index to define a sort order (and a range) and then filter out unwanted items with one of the query filtering methods.

When used on their own, filtering methods take O(n) time to find the next entry, since the system might examine and reject 999 out of 1000 items in the index before finding the first item in the desired set. Filtering methods do not create a sort order. If you want to retrieve the entries in a sorted order (or limit the range with query limiters like `beginKey` and `endKey`) , you *must* create an index.

The simplest and least efficient filtering method is a `validTest`. A query's `validTest` function must determine whether the entry is or is not in the appropriate set. It is the most powerful filtering method because it can run an arbitrary function to determine whether the item should be filtered out. However, it is slow because the functions execute a NewtonScript function and usually require the system to read in every entry. Reading in entries is slow, so use this as a last resort.

A faster filtering tool is the `indexValidTest`. This is similar to a `validTest`, except that the NewtonScript function uses the index key(s) instead of the entry itself. This is faster because the system will not read in the entry. However, it is less flexible because the `indexValidTest` can only use the index key(s). Also, because index keys may be truncated after 80 bytes, the `indexValidTest` must be prepared for subkeys to be truncated or missing.

Tags: Not Just For Luggage

An even better solution for filtering entries is to use tags, boolean flags you can attach to soup entries. Tags queries provide a way to test these boolean flags using a simple syntax. Instead of providing a NewtonScript function to determine whether the item is in the set, the system checks the tags and determines whether to filter out the item. A query just using tags takes O(n log m) time to iterate through an index looking for appropriate tags, where n is the number of indexed entries and m is the number of entries in that soup with at least one tag. However, it is very fast (the constant is low) due to the help of an extra low-level data structure created especially for tags.

If we added the appropriate tags to the soup entries, we can create a `tagSpec` to search for the example “employees who are married, have no children, have the ABC health plan, but don't have a dental plan:”

```
myQuery := unionSoup:Query({tagSpec: {
  all: ['isMarried, 'hasABCHealthPlan],
  none:['hasChildren,'hasDental]
});
```

Since all the tags are booleans, the tags mechanism can store tags for each entry using compact bit fields. The system stores these bit fields in a data structure called a tags data structure[3]. The tags data structure uses an entry identifier as the key value for a B-tree, with the tags bit fields as the data within each node.

As you might guess from our earlier discussions of B-trees, the system requires O(log m) time to find the tags for the entry and O(log n) time to find the first entry in the current index. Note again that n is the number of indexed entries and m is the number of entries in that soup with at least one tag. In a query that just uses tags, the system must repeat this process for potentially every entry (O(n) time) until your `tagSpec` matches the tags for an entry. This means that the system takes O(n log m) time to find the first match for a tags query. You might note that this may *sound* slower than `validTests`. However, a tags query does not need to execute NewtonScript code, read in entries, or manipulate NewtonScript objects, and the system uses efficient index and tags data structures to test entries. Overall, the result is *much faster* in comparison to any other filtering method we have discussed so far because of these extra data structures.

Textual Queries

Because it is similar to the other filtering types already discussed, I want to briefly mention textual queries. Like the other filtering methods, textual queries do not define an order. If you want to retrieve the entries in a sorted order (or limit the range with query limiters like `beginKey` and `endKey`), you *must* create an index.

The most important thing to note about textual queries is that strings are not flattened with the rest of the soup entry. There’s a separate data structure for storing strings. Unlike the other soup data structures using for retrieving data, this data structure is created for all strings and requires no extra store space nor explicit creation.

You can think of the strings stored as one long concatenated string for each soup entry. While writing entries to the store, the system writes string data (from anywhere in the entry) to a separate place on the store.

During textual queries, the system need not examine individual frame slots nor read in the entry. Because of this, the system can perform textual queries at a low level using this structure. Textual queries might potentially search all indexed entries, but the text-searching speed is very fast in comparison to filtering mechanisms that use NewtonScript functions to evaluate whether an item should be filtered out.

3 You’ll see other documentation that calls these “tags indexes”, but they are not indexes in the way we’ve been discussing them. You cannot use them to retrieve items in a sorted order. However, to use tags you must treat `tagSpecs` as `indexSpecs` when used with `store:CreateSoupXmit(...)` or `soupOrUnionSoup:AddIndex(...)`.

If Speed Matters: Some Raw Data

To compare different filtering methods, here is some speed data with multiple methods and a similar search. The test is based on a 1000 entry soup with about 700 bytes of data in each entry, including 200 bytes of text, and string searches are performed on a 40 character myString slot. Each search answers the question: how many entries have its myString slot start with the string "blah?"

Note that all searches were designed to search the entire soup as the “worst case scenario” and these timing samples apply *only* to this particular set of test data. Speed of particular filtering methods will vary based on your data, the complexity of your searches, and on each Newton device. These tests were performed on a MessagePad 120 with Newton 2.0 OS.

A validTest query takes 2225 ticks. The validTest read in every entry and tested the myString slot with the BeginsWith function.

An indexValidTest query takes 354 ticks. The indexValidTest tested the indexed myString key and tested it with the BeginsWith function

A text query takes 654 ticks. It had to search 95 bytes of text in each entry.

A words query takes 255 ticks. The words query had to search 95 bytes of text looking for "blah" at the beginning of a word. It is faster than the text query because the test data had one long word in each string, so most of the time was just searching for word beginnings and not comparing text.

A tags query optimized for this test takes 128 ticks. When adding the data, we determined whether this particular entry had its myString slot start with the string "blah." If it did, we added a special hasBlah tag to the entry. Since we were prepared for this query, we used tags to precalculate the filtering test and then did a tags query to retrieve the items.

An index query on the myString slot takes 5 ticks. The string index query used beginKey and endKey to limit the range in the index to the ones starting with "blah."

An index query optimized for this test takes 1 tick. When adding an entry, we determined whether this entry had its myString slot start with the string "blah". If so, we set a new hasBlahString slot to an integer instead of nil. If we add an integer index on the hasBlahString slot, we can use an index to iterate over *only those entries* which match our required criteria (and in a certain sort order). In this example, there was only one entry that matched the desired criteria so the index was very small. Note that no query filtering methods were used.

In the previous tests, no more than one filtering mechanism was used at one time. In your own projects, you can use several filtering methods within one query, and also use range limiters like beginKey and endKey if desired. Check the NPG for full details about the various query options.

I strongly recommend that you experiment with several designs for your soups and combinations of query methods before deciding on an implementation. Also, remember that you can store data in multiple soups if that makes your queries easier to conceptualize and faster to query.

If Size Matters: Some Raw Data

Most of the previous information focused on information to help you optimize your code for speed. Some applications must make performance tradeoffs based on the size of the data and the soup data structures. Of the soup data structures mentioned in this article, the ones which take up extra space are indexes and tags.

Indexes always copy key values for the data. For integers, characters, and symbols, the copied data is small. For reals and strings, more binary data is copied to the index (8 bytes for reals, up to 80 bytes for strings). The maximum amount of data copied for a key value is limited. For instance, no more than 39 characters would be used in a string index, and for multi-slot indexes, subkeys in multi-slot indexes may be truncated or missing in the index.

String indexes are particularly large, and this could significantly affect storage requirements for some applications. For instance, the Newton 2.0 ROM Time Zones soup is a package store part created in NTK. The Time Zones soup contains strings for city names and country names, and most of the names are short. Since the strings are indexed to retrieve city names in sorted order, the short strings are stored twice: once as part of the soup entry and once in the index. Including the size increase when converting ASCII strings to double-byte Unicode, the size of the Time Zones data was between 3 and 4 times larger in a package than in an ASCII text file.

On the topic of storage size, I want to mention an optimization used to store strings in soups (but not in string indexes). If the characters in the strings in an entry are Unicode characters with the high byte 0 (chars 0-255), only one byte is stored per Unicode character. The system transparently optimizes such strings when reading and writing soup entries to the store. This is the only transparent compression for soup data (other than the frames themselves).[4]

The system also tries to optimize the indexes themselves. When multiple entries have identical index keys (for instance, people with the same last & first names), the system stores the index information for those entries in a more compact way than saving a copy of the key for each entry.

Index size will vary on individual ROMs and vary based on the number of repeated index keys, but here are approximate index sizes on an Apple MessagePad 120 with Newton 2.0 OS. I have included index sizes for both short (15 chars) and long (40 chars) strings. Note that for some data types, I have included "with repeats" versions which means that the 1000 entries have only 1 of 10 possible values instead of unique values. For single-slot indexes of various index types using a 1000 entry soup, the index sizes are: Int: 15K, Char: 15K, Real: 17K, ShortString: 46K, LongString: 100K, LongString (with repeats): 4.5K, Symbol: 12.5K, Symbol (with repeats): 4.5K. Adding 2 tags each of 20 tags took 27K in the test soup.

Note that if you have read-only data, I strongly recommend that you consider storing your data in packages. See the “Lost In Space” article and sample for more information. For more information about making store parts and using package compression, see the NPG and the Newton Toolkit User’s Guide.

Designing for Performance

4 There is another store optimization for storing frames called “frame map sharing” that is outside the scope of this article. Also, Virtual Binary Objects (VBOs) can be explictly compressed when created (see the NPG for more info), and symbols are shared in symbol tables. See the NTK Users Guide for more info about compression and object-sharing options for packages.

There is too much variety in application data for me to recommend specific implementations. However, I can offer some things to consider when designing your soup structure.

1) *What sort orders will you need?* This may help you design your soup entry format and your indexes.
2) *What sets will you need to retrieve?* Determining what queries you'll make will help you design indexes and tags. I recommend that you use `indexValidTests`, `validTests`, and textual queries as a last resort only.
3) *Experiment.* Before implementing your entire application, test several implementations with real data or dummy data that accurately reflects the size and types of data in your final application.
4) *Determine if any data is read-only data.* If so, I recommend that you consider storing your data in packages. Check out the “Lost In Space” article and sample for more suggestions and information.
5) *Use the fastest filtering method that can do what you need.* For overall speed, here is the order of preference for filtering methods: tags, textual queries, `indexValidTest`, `validTest`. You might want to use more than one.

Conclusion

Here are the things which are most important to remember:
√ Indexes are required to retrieve data in sorted order.
√ Indexes are stored as B-trees and take O(log n) to find an entry.
√ Multi-slot indexes define a single sort order and use a single B-Tree.
√ Tags are important for retrieving sets, but are not indexes. Tags do not define a sort order.
√ Tags are used in conjunction with an index, even if it is the default index.
√ Indexes and tags data structures aren't free. They take time and storage space to maintain, so decide carefully what you need.

I want to reiterate that if you are ever in doubt of how to optimize your application for speed and/or size, experiment with several different implementations. In the words of the composer Cole Porter, "Experiment. Make it your motto day and night."

Thanks to Bob Ebert for his help with this article.

———————————————————

Cross-references

Apple Computer, Newton Programmer's Guide (NPG) 2.0 and Newton Programmer's Reference 2.0, 1996. Apple Computer, Inc. Available on-line on Newton Developer CD-ROMs. The NPG 2.0 will also be published in hard copy by Addison Wesley.

Baase,Sara, Computer Algorithms, 1988. Publisher: Addison-Wesley. Find more information on-line at http://heg-school.aw.com/cseng/authors/baase/compalgos/compalgos.html and http://www.amazon.com/exec/obidos/ISBN=0201060353

Engber, Michael, Lost In Space article and LostInSpace code, 1995. Publisher: Apple Computer, Inc. Find them on Newton Developer CDs.

Porter, Cole, Experiment. Copyright 1993 (Renewed) by Warner Bros. Find more information about Cole on-line at my Cole Wide Web, http://www.doitall.com/cole

Sedgewick, Robert, Algorithms, 1988. Publisher: Addison-Wesley. Find more information on-line at http://heg-school.aw.com/cseng/authors/sedgewick/algo-in-c/algo-in-c.html and http://www.amazon.com/exec/obidos/ISBN%3D0201066734