Programming with Haiku

Lesson 13

Written by DarkWyrm

Queries

Haiku's parent operating system, BeOS, was far ahead of its time and possessed advanced features that were missing from Windows 95 and MacOS, the competing operating systems of its day. In addition to extended file attributes, it also provided query support. Queries are lightning-fast file searches which leverage Haiku's unique filesystem: BFS. While other operating systems were recursively searching every single file for matches within filenames and the like, BeOS users were using queries to organize MP3s, contacts, e-mail, and photos. Haiku users still do all of this, but it's often left underused except by advanced users and developers. Even now more popular operating systems cannot search for files as quickly and easily with the same power without taxing the rest of the operating system.

Query Syntax

Queries are usually run via Tracker's Find window. Most of them use the *by name* or *by attribute* modes. Hardcore Haiku users love the *by formula* setting, which uses the actual syntax of a filesystem query without all the niceties. Fortunately, the syntax of a query is relatively simple and can be learned quickly by entering a test query in one of the other modes and then changing to the formula mode.

Let's start with a simple query: find all new e-mail.

1. Open Tracker's Find window by pressing Command-F or choosing *Find...* from the Deskbar or the File menu in any Tracker window.
2. Instead of leaving it set to *All files and folders*, change the file type to *E-mail* in the text submenu.
3. Change *by name* to *by attribute*.
4. Now click on the menu field marked *Name*, choose *is* from the *Status* submenu, and type the word *new* into the text box. This is what the query will search for, but take a look at the query's real syntax by changing the mode to *by formula*.

Having done all of this, you will see the following text in the Find window:

```
((MAIL:status=="[nN][eE][wW]")&&(BEOS:TYPE=="text/x-email"))
```

For those unfamiliar with regular expressions, a set of characters within a pair of square brackets matches any single character in that set. `[nN]` matches the letter N, regardless of case, for example. This query will match any e-mail which has a status attribute of *new* without paying any attention to capitalization. Queries are case-sensitive, but any query generated with the name or attribute modes are converted so that they are not case-sensitive.

Queries can be quite complicated, but they don't have to be that way. A simpler query which would get the job done as well as the one above could look like this:

```
MAIL:status=="[nN]ew"
```

Since e-mail files are the only files which have the `MAIL:status` attribute, just searching for the attribute makes more sense.

If a query is to be run on a particular attribute, it must be indexed. The Terminal commands `lsindex`, `rmindex`, and `mkindex` manipulate and list which attributes are indexed. Note that when an attribute is added to the filesystem's index with the `mkindex` command, all files having that attribute are not automatically found by a query. They must be reindexed. This can be accomplished by copying the file somewhere and then copying it back or using the `reindex` Terminal command. *A word of warning to the wise*: don't be afraid to add an attribute to the index, but don't go crazy over them, either. Adding lots of them will slow down overall performance of all query-based searches.

Using BQuery

There are two ways that BQuery can be used to run a query. The simpler way is really easy if you understand the actual query syntax. The more complicated way is more flexible, but not necessary in most cases. We'll examine the simpler method first. Here is some very basic code for running a query on the boot volume:

```
#include <Query.h>
#include <stdio.h>
#include <String.h>
#include <VolumeRoster.h>
#include <Volume.h>

int
main(void)
{
	// Get a BVolume object which represents the boot volume. A BQuery
	// object will require one.
	BVolumeRoster volRoster;
	BVolume bootVolume;
	volRoster.GetBootVolume(&bootVolume);

	// A quick summary for using BQuery:
	// 1) Make the BQuery object.
	// 2) Set the target volume with SetVolume().
	// 3) Specify the search terms with SetPredicate().
	// 4) Call Fetch() to start the search
	// 5) Iterate over the results list using GetNextEntry(),
	//    GetNextRef(), or GetNextDirents().

	BString predicate("MAIL:subject == *");
	BQuery query;
	query.SetVolume(&bootVolume);
	query.SetPredicate(predicate.String());
	if (query.Fetch() == B_OK)
	{
		printf("Results of query \"%s\":\n", predicate.String());
		entry_ref ref;
		while (query.GetNextRef(&ref) == B_OK)
			printf("\t%s\n",ref.name);
	}
}
```

With code this easy, you would think that queries would be used more in third party applications!

The more flexible method is more complicated because of the way that the query is assembled. It uses a token stack – chaining together individual components of the search terms in a specific order called Reverse Polish Notation. Each element is added to the predicate using methods like `PushOp()`, and `PushAttr()`.

The **Reverse Polish Notation** (RPN) entry system dates back to the 1950s, but it can still be found in many different areas, particularly financial and scientific calculators. Also known as **Postfix notation**, it groups mathematical operands together and places the operator at the end. For example, what we would normally write as `(5 + 6) * 3` becomes `5 6 + 3 *` in RPN. Parentheses are not necessary if the precedence of mathematical operators is upheld. RPN is very easy for computers, but it can be a real headache for people, having had algebraic entry ingrained into us since elementary school. Luckily for us, queries normally have a simple syntax, so it isn't that much more difficult to construct a query this way.

Here are the methods which are used to construct queries using RPN:

```
void PushAttr(const char *attrName);
void PushOp(query_op operator);

void PushUInt32(uint32 value);
void PushInt32(int32 value);
void PushUInt64(uint64 value);
void PushInt64(int64 value);
void PushFloat(float value);
void PushDouble(double value);
void PushString(const char *string, bool ignoreCase = false);
```

The top two methods are for adding attribute names and comparison operators; the rest are for adding values of different types.

The operators which can be used with `PushOp()` are as follows:

Operator	Operation
`B_EQ`	==
`B_NE`	`!=`
`B_GT`	>
`B_LT`	<
`B_GE`	>=
`B_LE`	<=
`B_CONTAINS`	The same as the regular expression `*value*`
`B_BEGINS_WITH`	The same as the regular expression `*value`
`B_ENDS_WITH`	The same as the regular expression `value*`
`B_AND`	&&
`B_OR`	`||`
`B_NOT`	`!`

Modifying the previous query example to use RPN results in this code:

```
#include <Entry.h>
#include <Query.h>
#include <stdio.h>
#include <String.h>
#include <Volume.h>
#include <VolumeRoster.h>

int
main(void)
{
	// Get a BVolume object which represents the boot volume. A BQuery
	// object will require one.
	BVolumeRoster volRoster;
	BVolume bootVolume;
	volRoster.GetBootVolume(&bootVolume);

	// A quick summary for using BQuery:
	// 1) Make the BQuery object.
	// 2) Set the target volume with SetVolume().
	// 3) Specify the search terms with Push*().
	// 4) Call Fetch() to start the search
	// 5) Iterate over the results list using GetNextEntry(),
	//    GetNextRef(), or GetNextDirents().

	BString predicate("MAIL:subject == *");
	BQuery query;
	query.SetVolume(&bootVolume);
	query.PushAttr("MAIL:subject");
	query.PushString("*");
	query.PushOp(B_EQ);
	if (query.Fetch() == B_OK)
	{
		printf("Results of query \"%s\":\n", predicate.String());
		entry_ref ref;
		while (query.GetNextRef(&ref) == B_OK)
			printf("\t%s\n",ref.name);
	}
}
```

Be aware that the two techniques cannot be mixed. Any search terms added to a BQuery by way of a `Push` method takes precedence over anything passed to `SetPredicate()`.

Searching in Real Time Using Live Queries

The above code examples use queries statically – the results are read and do not change, but a query can also do live updates. Live queries send update messages to your program when a file suddenly matches the query or disappears off the face of the earth.

Making a query live is easy: pass a valid BHandler or BLooper to the `SetTarget()` method before calling `Fetch()`. Handling updates requires some finesse, though. When an update message is received, it may be while you're still busy reading the results using one of the `GetNext` methods, so message handling needs to be synchronized with the reading of the results. Also, do be careful that the BMessenger object passed to SetTarget() is not deleted or allowed to go out of scope until you are done with your query. Deleting it any sooner will cause update messages to not be sent any more.

The query update message has the identifier B_QUERY_UPDATE. When one is received, you need to then read the 32-bit integer field opcode to find out what other data fields the message contains.

Opcode B_ENTRY_CREATED:

Field Name	Type	Description
opcode	int32	Identifier for the message, equaling B_ENTRY_CREATED in this case.
name	string	The name of the new entry
directory	int64	The ino_t number for the directory in which the entry resides.
device	int32	The dev_t number of the device on which the entry exists.
node	int64	The ino_t of the entry itself.

Opcode B_ENTRY_DELETED:

Field Name	Type	Description
opcode	int32	Identifier for the message, equaling B_ENTRY_DELETED in this case.
directory	int64	The ino_t number for the directory in which the entry formerly resided.
device	int32	The dev_t number of the device on which the entry used to exist.
node	int64	The ino_t of the removed entry.

At first glance, you might think that live queries are almost always the best choice. There is a drawback to using them: there is some overhead which must be maintained for these messages to actually be useful. All of the information which is obtained from the creation messages – the name, node, and so forth – needs to be stored away because there is no name field sent for B_ENTRY_DELETED. This critical piece of information makes it impossible to create an entry_ref, which happens to be the most common way to store the location of a file or directory outside of strings.

When using a live query, store away the information gleaned from whichever GetNext method you use and do the same for any B_ENTRY_CREATED messages. This way, if and when you receive B_ENTRY_DELETED messages you will be able to look up the proper item from the information that you are given.

Concluding Thoughts

Queries are easy to use, fast, and powerful. If you have to look up a collection of files on the system, such as all people files in the case of a contact manager, they provide an easy way of finding and retrieving files. Keep them in mind as you work on projects – you may find a new use for them which blows people's minds.

Going Further

- Run the `lsindex` command in the Terminal. Most, if not all of them, are pretty self-explanatory. What uses can you think of for some of them?
- If you were going to write a music organizer, how could you use queries and attributes, both indexed and not, to get the most power out of the app as possible?
- How could you go about leveraging queries in these parts of a personal organizer app: contacts, appointments, tasks, and e-mail?