Why is the registry a hierarchical database instead of a relational one?

devblogs.microsoft.com/oldnewthing/20110907-00

September 7, 2011

Raymond Chen

Commenter ton asks why the registry was defined as a hierarchical database instead of a relational database.

Heck, it's not even a hierarchical database!

The original registry was just a dictionary; i.e., a list of name/value pairs, accessed by name. In other words, it was a *flat* database.

`.txt`	`txtfile`
`txtfile`	`Text Document`
`txtfile\DefaultIcon`	`notepad.exe,1`
`txtfile\shell`	`open`
`txtfile\shell\open\command`	`notepad %1`

If you turned your head sideways and treated the backslashes as node separators, you could sort of trick yourself into believing that this resulted in something vaguely approximating a hierarchical database, and a really lame one at that (since each node held only one piece of data).

When you choose your data structures, you necessarily are guided by the intended use pattern and the engineering constraints. One important engineering constraint was that you have to minimize memory consumption. All of the registry code fit in 16KB of memory. (Recall that Windows 3.1 had to run on machines with only 1MB of memory.)

Okay, what is the usage pattern of the registry? As originally designed, the registry was for recording information about file types. We have the file types themselves (`txtfile`), properties about those file types (`DefaultIcon`), verbs associated with those file types (`open`), and verb implementations (`command` or `ddeexec`). Some verb implementations

are simple (`command` involves just a single string describing the command line); others are complex (`ddeexec` requires the execute string, the application, and the topic, plus an optional alternate execute string).

- Given a file type and a property, retrieve the value of that property.
- Given a file type and a verb, retrieve information about how to perform that verb.
- The set of properties can be extended.
- The set of property schemata can be extended.
- The set of verbs can be extended.
- The set of verb implementations can be extended.

Since the properties and verb implementations can be extended, you can't come up with a single schema that covers everything. For example, over the years, new file type properties have been added such as `ContentType`, `OpenWithList`, and `ShellNew`. The first one is a simple string; the second is a list of strings, and the third is a complex key with multiple variants. Meanwhile, additional verb implementations have been added, such as `DropTarget`.

Given the heterogeneity of the data the registry needs to keep track of, imposing some sort of uniform schema is doomed to failure.

"But you can just update the schemata each time the registration is extended."

That creates its own problems. For example, to support roaming user profiles, you need a single registry hive to work on multiple versions of the operating system. If version N+1 adds a new schema, but then the profile roams to a machine running version N, then that registry hive will be interpreted as corrupted since it contains data that matches no valid schema.

"Well, then include the schemata with the roaming profile so that when the older operating system sees the hive, it also sees the updated schemata."

This is trickier than it sounds, because when the profile roams to the newer operating system, you presumably want the schemata to be upgraded and written back into the user profile. It also assumes that the versioning of the schemata is strictly linear. (What if you roam a user profile from a Windows XP machine to a Server 2003 machine? Neither is a descendant of the other.)

But what kills this proposal is that it makes it impossible for a program to "pre-register" properties for a future version of the operating system. Suppose a new schema is added in version N+1, like, say, the IDropTarget verb implementation. You write a program that you want to run on version N as well as on version N+1. If your installer tries to register the version N+1 information, it will fail since there is no schema for it. But that means that when

the user upgrades to version N+1, they don't get the benefit of the version N+1 feature. In order to get the version N+1 feature to work, they have to reinstall the program so the installer says, "Oh, now I can register the version +1 information."

"Well, then allow applications to install a new schema whenever they need to."

In other words, make it a total free-for-all. In which case, why do you need a schema at all? Just leave it as an unregulated collection of name/value pairs governed by convention rather than rigid rules, as long as the code which writes the information and the code which reads it agree on the format of the information and where to look for it.

Hey, wow, that's what the registry already is!

And besides, if you told somebody, "Hi, yeah, in order to support looking up four pieces of information about file types, Windows 3.1 comes with a copy of SQL Server," they would think you were insane. That's like using a bazooka to kill a mosquito.

What are you planning on doing with this relational database anyway? Are you thinking of doing an INNER JOIN on the registry? (Besides, the registry is already being abused enough already. Imagine if it were a SQL server: Everybody would store *all their data* in it!)

ton explains one way applications could use this advanced functionality:

> An application would have a table or group of tables in relational style registry. A group of settings would be a row. A single setting would be a column. Is it starting to become clearer now how SQL like statements could now be used to constrain what gets deleted and added? How good is your understanding of SQL and DBMS?

You know what most application authors would say? They would say "Are you mad? You're saying that I need to create a table with one column for each setting? And this table would have a single row (since I have only one application)? All this just so I can save my window position? Screw it, I'm going back to INI files." What'll happen in practice is that everybody will create a table with two columns, a string called `name` and a blob called `value`. Now we've come full circle: We have our flat database again.

And how would they make sure the name of their table doesn't collide with the name of a table created by another application? Probably by encoding the company name and application name into the name of the table, according to some agreed-upon convention. Like say, the Settings table used by the LitSoft program written by LitWare would be called `LitWare_LitSoft_Settings`. So querying a value from this table would go something like

```
SELECT value FROM PerUser.LitWare_LitSoft_Settings
    WHERE name = "WindowPosition"
```

Hey, this looks an awful lot like

```
Registry.CurrentUser.OpenSubKey(@"LitWare\LitSoft\Settings")
        .GetValue("WindowPosition");
```

One of ton's arguments for using a relational database is that it permits enforcement of referential integrity. But I would argue that in the general case, you *don't want* strict enforcement of referential integrity. Suppose you uninstall a program. The uninstaller tries to delete the program registration, but that registration is being referenced by foreign keys in other tables. These references were not created by the application itself; perhaps the shell common dialog created them as part of its internal bookkeeping. If the registry blocked the deletion, then the uninstall would fail. "Cannot uninstall application because there's still a reference to it somewhere." And that reference might be in Bob's user profile, from that time Bob said, "Hey can I log onto your machine quickly? I need to look up something." Bob is unlikely to come back to your machine any time soon, so his user profile is just going to sit there holding a reference to that application you want to uninstall for an awfully long time. "Hi, Bob, can you come by my office? I need you to log on so I can uninstall an app."

So let's assume it goes the other way: The registry automatically deletes orphaned foreign key rows. (And for hives that are not currently available, it just remembers that those foreign key rows should be deleted the next time they are loaded. Nevermind that that list of "foreign key rows that should be deleted the next time Bob logs on" is going to get pretty long.)

Now suppose you're uninstalling a program not because you want to get rid of it, but because you're doing an uninstall/reinstall troubleshooting step. You uninstall the program, all the orphaned foreign key rows are automatically deleted, then you reinstall the program. Those orphaned foreign key rows are not undeleted; they remain deleted. Result: You lost some settings. This is the reason why you don't clean up per-user data when uninstalling programs.

Enforcing referential integrity also means that you can't create anticipatory references. One example of this was given earlier, where you register something on version N even though the feature doesn't get activated until the user upgrades to version N+1. More generally, Program X may want to create a reference to Program Y at installation, even if program Y isn't installed yet. (For example, X is a Web browser and Y is a popular plug-in.) The Program Y features remain dormant, because the attempt by Program X to access Program Y will fail, but once the user installs Program Y, then the Program Y features are magically "turned on" in Program X.

Consider, as an even more specific example, the "kill bit" database. There, the goal isn't to "turn on" features of Program Y but to turn them off. Imagine if referential integrity were enforced: You couldn't kill an ActiveX control until after it was installed!

Raymond Chen

Follow