A Wrinkle in Timeliness Part 3: The Final Reckoning

In my first post of this three part series, I posed a challenge: how do you update hundreds or thousands of tasks interconnected with tens of thousands of connections? In my second installment, we explored a possible solution: recursively looping through each task from the end-level tasks, tracing each one to its source, then returning back down the stack and updating the dates along the way.

However, one issue was still in the way. Recursion requires that each iteration maintain it's "scope". In this case, the current record and found set would have to be unique for each iteration. This would mean each iteration would have to open a new window; the proliferation of hundreds of windows would be excruciatingly slow and potentially would crash FileMaker.

In this final installment, we'll explore an elegant solution that adds lightning-fast speed, avoids crashing FileMaker Pro and includes a free set of Ginsu knives if you read now*!
**Ginsu knives not included, additional processing and handling charges may apply.*

A Virtual Planetoid

The solution lies in caching (sometimes referred to as virtualizing). In general computing, caching refers to storing a duplicate of data in a more quickly accessible location. In FileMaker, caching usually refers to temporarily storing record data in global variables.

In our scenario, we'll cache the dependency data (which tasks each task is dependent on) and the offset data (how long each task will take). This will allow us to do all our calculating without touching the database again until the very end. The benefits of this are twofold:

1. We can maintain scope in each iteration because we will no longer be relying on the FileMaker found set. We'll be able to use the script's inherent scope, in essence maintaining our own "found set" by storing a list of record ids in local variables.

2. Because we don't touch the database engine until the very end, this approach is lightning fast. In my implementation, all tasks are updated within 1 to 2 seconds.

Caching In

Implementing the caching is fairly straightforward (the "Dependencies" table is a join table connecting a task with one or more tasks it depends on):

1. Create a calculation field with the information we want to cache, formatted like a Let() statement:

dependency_entry =

"Let ($$dep_" & id & " = \"" & List (Dependency_Child::id) & "\"; 0)"

This will create a Let() function ready to be executed. It would look something like this:

Let ($$dep_123 = "4¶5¶6" ; 0)

When executed with the Evaluate() function, it will create a global variable that is named after the record id and contains a list of its dependencies.

The offset cache is similar:

offset_entry =

"Let ($$off_" & id & " = \"" & offset & "\"; 0)"

2. We then run this script:

Create Caches =

Freeze Window

Go to Related Record [From table: “Dependencies_Project”; Using layout: “Dependencies” (Dependencies)] [Show only related records]

Go to Record/Request/Page [First]

Loop

Set Variable [$x; Value:Evaluate (Dependencies::dependency_entry)]

Set Variable [$x; Value:Evaluate (Dependencies::offset_entry)]

Go to Record/Request/Page [Next; Exit after last]

End Loop

Go to Layout [original layout]

This loops through the dependency records, executing the code in dependency_entry and offset_entry to create the variables.

To retrieve the data, we simply use Evaluate() again to calculate the name of the variable we want and retrieve it, such as:

```
Evaluate ( "$$dep_" & $record_id )
```

We'll also be creating more caches as we go, which will contain the calculated dates and their corresponding task id.

Let Me Check My Script

We need two scripts to update the dates: a recursive script and a master script to wrap it in.

First, the master script:

Update Dates (master) =

Go to Related Record [From table: “Project”; Using layout: “Project” (Project)]

Perform Script [“Create Caches”; Parameter: ""]

Create the caches

Set Variable [$base_dependencies; Value:List (Dependency::base_id)]

This gets a list of ids for all base dependencies, i.e., those with no child dependencies. "base_id" is a calculation that equals the id if that record has no children.

If [IsEmpty ($base_dependencies)]

Go to Layout [original layout]

If no base dependencies exist, don't proceed

Else

Set Variable [$$timeliness_cache; Value:""]

Set Variable [$counter; Value:1]

We will be caching the dates as we calculate them and storing them here, so we need to make sure it's clear

Loop

Set Variable [$current_parent; Value:getValue ($base_dependencies ; $counter)]

We'll be looping through the list of base dependencies one by one

Exit Loop If [IsEmpty ($current_parent)]

Exit the loop once we've gone through all base dependencies

Perform Script [“ops: update dates (recursive)”; Parameter: $current_parent]

We'll look at this script below

Set Variable [$result; Value:Get (ScriptResult)]

Set Variable [$x; Evaluate ("Let ($$time_" & $current_parent & " = \"" & $result & "\"; 0)")]

Cache the result in a new global variable named with the id of the current dependency

Set Variable [$counter; Value:$counter + 1]

Increment $counter so we can move onto the next value

End Loop

Go to Layout [original layout]

End If

Now, the recursive script:

Update Dates (recursive) =

Set Variable [$current_node; Value:Get (ScriptParameter)]

Set Variable [$parents; Value:Evaluate ("$$dep_" & $current_node)]

Retrieve the child dependencies for the current dependency

Set Variable [$counter; Value:1]

Loop

Set Variable [$current_parent; Value:GetValue ($parents ; $counter)]

Loop recursively through the child dependencies

Exit Loop If [IsEmpty ($current_parent)]

Set Variable [$result; Value:Evaluate ("$$time_" & $current_parent)]

Check to see if the current dependency has already been calculated and cached. If it has, we can retrieve it without going further up the tree

If [isEmpty ($result)]

Otherwise, we have to continue recursively up the tree

Perform Script [“ops: update variable timeliness (recursive)”; Parameter: $current_parent]

Set Variable [$result; Get (ScriptResult)]

Set Variable [$x; Evaluate ("Let ($$time_" & $current_parent & " = \"" & $result & "\"; 0)")]

Cache the result in a new global variable named with the id of the current dependency

End If

Set Variable [$max_parent_date; Value:Max (GetAsNumber ($max_parent_date) ; GetAsNumber ($result))]

Keep track of the maximum date of the parent dependencies

Set Variable [$counter; Value:$counter + 1]

End Loop

Exit Script [Result: $max_parent_date + Evaluate ("$$off_" & $current_node)]

Retrieve the corresponding offset for the current node, add it to the latest date of the parents and return it as the script result

They'll Go Back Into Storage

The final piece of the puzzle is getting all those dates global variables into actual fields. This is simpler than it looks. We just have to

1. Create an unstored calculation in the **Dependency** table that retrieves its corresponding date from the cache:

    date_timeliness =

    Value:Evaluate ("$$time_" & id)

2. Create a simple **Timeliness** table with only two fields:

    dependency_id
    date

This table will have a one-to-one relationship with Dependency, based on id = dependency_id.

3. Do a self-import from the **Dependency** table to **Timeliness,** with a matching import based on id to dependency_id.

    id <-> dependency_id
    date_timeliness -> date

This will refresh the data in Timeliness::date with the data in the $$time_ variables, and it will do it very quickly. Also, because it's in a dedicated table, we don't have to worry about record locking issues.

There you have it! Our script

1. Loops through the tasks, tracing each to its source if necessary
2. Goes back down through the tree, adding an entry to the date cache for each task, based on the dates of its parent tasks plus its own duration (offset)

3. If it encounters an entry that already has a cached date, it retrieves it without further calculation
4. When it's finished, it stores the results in a dedicated table.

Let Me Sum Up

It's been a bit of a journey, but our approach, combining recursion and caching, solves the problem and does it quite speedily. It has the additional advantage of performing almost equally well when the solution is hosted over a WAN. The server is only touched at the beginning and end of the script; all the complex calculations take place on the client side with already-cached data.

I encourage you to take a closer look at caching with FileMaker Pro. It has many potential applications when speed is crucial, especially in solutions hosted over a WAN. Recursion, while a more specialized solution, is extremely powerful in certain situations.

I hope you found this series useful and educational! Please feel free to share your comments, questions, and feedback on our eX-Cetera blog.

Andy Persons is a Senior Lead FileMaker Pro Developer with Excelisys: Andy has been an industry leading FileMaker Pro developer creating FileMaker Pro solutions for over 17 years. In addition to being one of the lead developers of three top-rated and most-downloaded FileMaker Pro solutions of all-time; the FileMaker Business Tracker and the Excelisys eX-BizTracker & eX-BizTracker Pro jump-start solutions, he has shared his incredible and advanced talents by authoring numerous Tips-n-Tricks files and white papers, including Hierarchical Portals, Recursive Calcs, Audit Logs and Drag-and-Drop using FileMaker Pro.