A Wrinkle in Timeliness Part 2

In my previous post, I introduced the challenge: updating hundreds or thousands of tasks interconnected with tens of thousands of connections. We also took a look at a couple possible solutions, but determined that they were unworkable.

Now let's take a look at another possibility.

Trickle Up

The last approach we looked at previously was recursively looping through the tasks, starting from the base task (Concept). We decided that it would be far too slow, since it would have to update the same task potentially hundreds of times or more. But what about looping through it recursively from the other end?

This approach would have to differ from the previous recursive script. Consider this structure:

- Concept (2)
    - Task A (2)
        - Task B (4)
        - Task C (5)
    - Task D (3)
        - Task B (4)

Say we started at task C. In order to calculate its projected completion date (for brevity, I'll just call it "date"), we would need to know Task A's date. But to know Task A's date, we would need Concept's date.

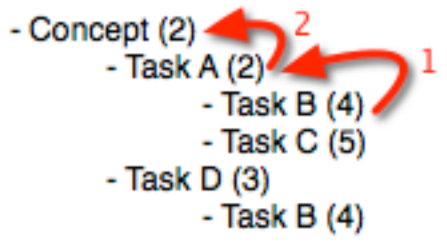
To accomplish this, we would need to use traditional recursion instead of the "tail" recursion we tried before. It would look something like this:

```
Update Tasks =

If [ not IsEmpty ( Tasks_Parents::id ) ]
        Go to related records [ Tasks_Parents ]
        Loop
                Perform Script [ Update Tasks ]
                Set variable [ $date ; Get ( ScriptResult ) + duration ]
                Set date to [ $date ]
                Set variable [ $max_date ; Max ( $max_date ; $date ) ]
                Go to Next Record [ Exit After Last ]
        End Loop
        Exit Script [ $max_date ]
Else
        Exit Script [ date + duration ]
End If
```

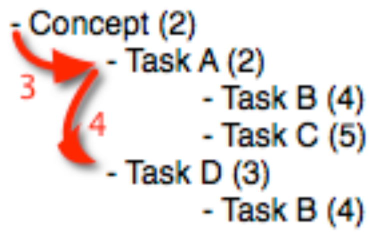
In essence, this script traces each task from the "end-level" tasks (those with no tasks dependent on them) all the way up the hierarchy until it reaches a base task (a task that is not dependent on any other task). It then returns its date to the previous iteration and continues.

In our above structure, the script starts at Task B, then goes to Task A, then Concept.



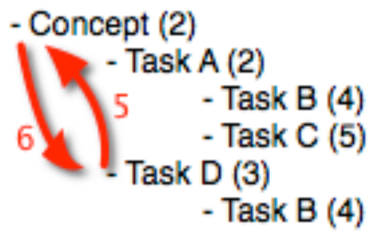
```
- Concept (2)
        - Task A (2)
                - Task B (4)
                - Task C (5)
        - Task D (3)
                - Task B (4)
```

Concept then returns its date to Task A, which sets its date to the returned date plus Task A's duration. The script then proceeds to Task D (since it's another parent of Task B).



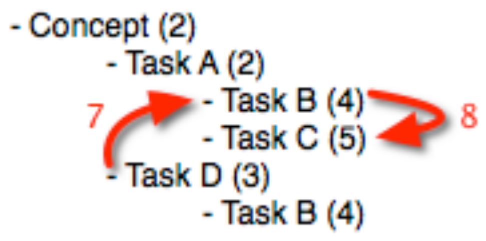
```
- Concept (2)
        - Task A (2)
                - Task B (4)
                - Task C (5)
        - Task D (3)
                - Task B (4)
```

It then goes back to Concept (since it's a parent of Task D also) and again returns its date to Task D, which sets its date to the returned date plus Task D's duration.



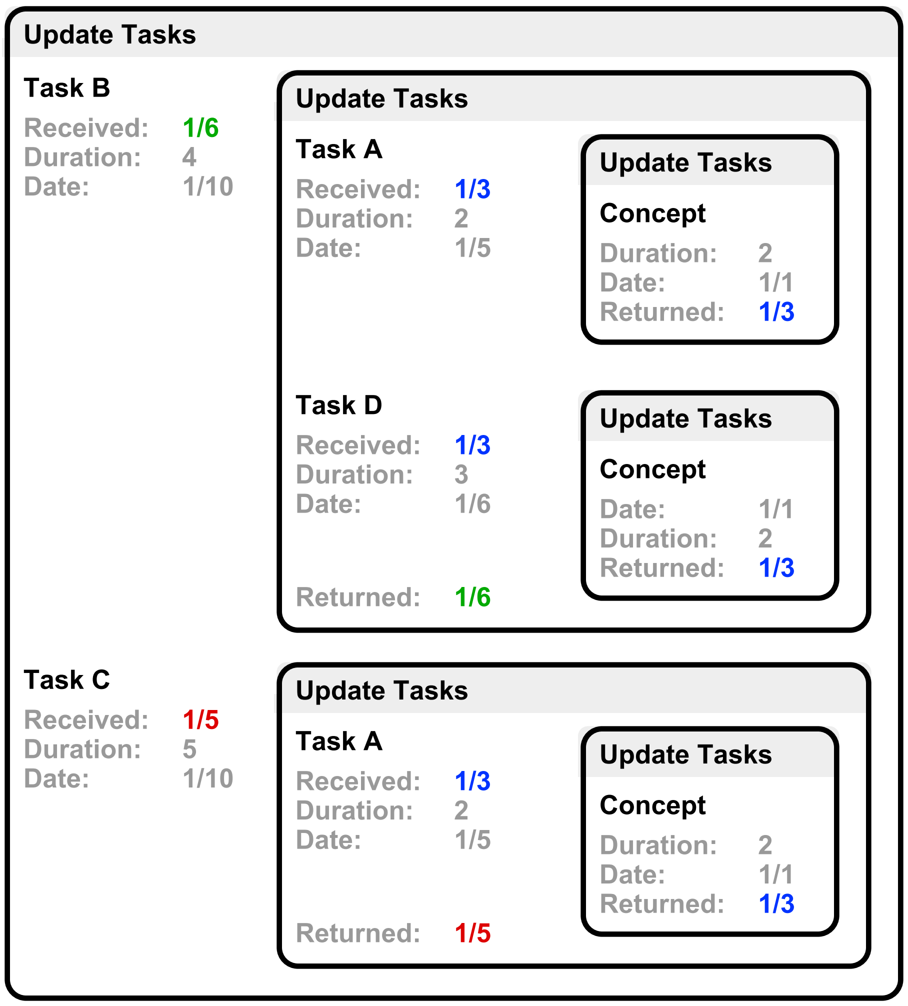
```
- Concept (2)
        - Task A (2)
                - Task B (4)
                - Task C (5)
        - Task D (3)
                - Task B (4)
```

FInally, it returns the maximum of Task A and Task D dates to Task B, which sets its date to the returned date plus its duration. Then it continues on to Task C to repeat the process.



- Concept (2)
    - Task A (2)
        - Task B (4)
        - Task C (5)
    - Task D (3)
        - Task B (4)

Here's another way to visualize what's happening (each rounded rectangle represents one instance of the Update Tasks script):



Update Tasks

Task B
Received: 1/6
Duration: 4
Date: 1/10

Update Tasks

Task A
Received: 1/3
Duration: 2
Date: 1/5

Update Tasks

Concept
Duration: 2
Date: 1/1
Returned: 1/3

Task D
Received: 1/3
Duration: 3
Date: 1/6

Update Tasks

Concept
Date: 1/1
Duration: 2
Returned: 1/3

Returned: 1/6

Task C
Received: 1/5
Duration: 5
Date: 1/10

Update Tasks

Task A
Received: 1/3
Duration: 2
Date: 1/5

Update Tasks

Concept
Duration: 2
Date: 1/1
Returned: 1/3

Returned: 1/5

I'm Thinking… I'm Thinking…

Now, an enterprising individual might object that this script would have to update the same tasks tens of thousands of times just like the previous recursive script; at first glance, they'd be right. However, this approach has one crucial advantage: *once a date has been set, it's final.* Unlike the previous approach, a date isn't set until every ancestor has been taken into account. This means that once a date is set, we don't need to calculate it again.

This means we could clear all dates before running the script. Then, if a date had a value, we wouldn't have to update it (or any of its ancestors). This alone would save a huge amount of time.

Don't Worry, I Crash Better Than Anyone I Know

That same enterprising individual might also point out that this approach still has the same scope requirement: it would have to generate a new window for each iteration, slowing things down and potentially crashing FileMaker. This one is more problematic; how do we maintain each instance's scope without creating new windows, while still maintaining speed?

In our next installment, we'll explore how to accomplish this.