Architecture Alternatives for Batch Scheduling

Steve Morrison, Ph.D. Chem. Eng. Info@MethodicalMiracles.com. 214-769-9081

Abstract: An inappropriate architecture can have a significant negative effect on the success of a scheduling project. In development it can delay the release date, impair testability, and increase the difficulty of software maintenance. The final product can be a very slow scheduler. Even worse, it can make a scheduling problem grow until it can no longer fit in the computer. This paper walks the user through some key architecture decisions made on a large scheduling project, so that users on their own projects can make better, more informed decisions.

Key words: batch scheduling, architecture, scheduling database access, scheduling architecture

Introduction

The mismatched application architecture can have a significant impact on scheduling. Worst case, it can affect the speed of obtaining a schedule by a factor of 20, and the memory footprint by almost as much. It is important to balance the speed of the scheduling algorithm with usability and integration with other aspects of the system. One might want a more high-performance architecture to do schedule optimization than if one is not. First we will look at the scheduler in isolation, and then see how integration with other systems should influence architecture decisions.

Tradeoffs of Architecture Building Blocks

In Microsoft’s .Net, using DataSets for all data appears attractive, because it is easy to get to and from the database, and some GUI controls work with DataSets. However, DataSets have significant overhead, and they are an order of magnitude slower than regular object instances for manipulation. In .Net DataSets or DataReaders might be appropriate for a Data Access layer, they have too large a memory footprint and too slow performance for numerically intensive applications.

On the other extreme, the simplest, fastest, and least flexible architecture is simply putting everything in arrays. While some modeling systems use this approach, this is never recommended because of the extreme clumsiness on inserting and appending rows of data. Using object instances, with an efficient way to model relationships, such as linked lists is fast, flexible, and scalable. However, in managed .Net languages such as C#, and in Java, the garbage collection looking over such large numbers of objects can be a problem. Arrays + linked lists gave insertion and append capability equal to instances, because there is no expectation of the next element physically following the current element. However, it has inefficiencies in dynamic resizing. This author performed benchmarks were performed using linked lists on 10,000 assignments 20,000 times using six architectures.
arrays + linked lists, structs + linked lists, instances + linked lists, and hybrid instance struct approach gave the following results on making 10,000 assignments 20,000 times.

<table>
<thead>
<tr>
<th>Architecture Approach</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arrays + Linked Lists</td>
<td>31 seconds</td>
</tr>
<tr>
<td>Pure Struct + Linked Lists</td>
<td>38 seconds</td>
</tr>
<tr>
<td>Hybrid instance struct approach with linked lists</td>
<td>51 seconds</td>
</tr>
<tr>
<td>Instances + linked lists</td>
<td>53 seconds</td>
</tr>
<tr>
<td>Instances + ArrayList Collections</td>
<td>79 seconds</td>
</tr>
<tr>
<td>DataSets + ArrayList Collections</td>
<td>90 seconds</td>
</tr>
</tbody>
</table>

Note that the collections were ArrayLists, which have the least overhead with respect to access. On the collections, new records were appended at the end. The performance would be expected to be much worse if inserts in the middle were required. Note that for all these results no hash tables or lookups were done.

**Subsystem Architecture Alternatives**

Figure 1 shows three ways to architect almost any complex system, not just scheduling.

The first alternative is the simplest with one common access to the database, and one layer of “landscape”. The landscape, also called the business entity layer, contains passive objects with get set, insert, delete, and similar methods. The engine in a scheduler performs the different kinds of scheduling functions at a campaign, batch, unit, and operation level. Note that the Graphical User Interface (GUI) uses the same objects as the engine. The objects in the landscape can be arrays plus linked lists, instances plus collections, datasets, or any of the alternatives discussed in the previous section.
In the second alternative, a landscape layer is optimized for the engine, and a second landscape is present for the GUI. Note that objects must be accessed twice, in different ways, from the database. Alternately, objects only need to be accessed from the database once, but there needs to be a translation layer between the two landscapes. When a bug displays in the GUI, there needs to be an efficient way to determine whether the bug is due to the GUI, the engine, or the translation between the two landscapes. Clearly decreased maintainability is a drawback of this architecture alternative, but a second drawback can be the performance of going from one landscape to another, or the doubling of the data access.

The third alternative at first looks as easy as the first, but in fact it has similar problems as the second. There may be only one “official” landscape or BE layer (Landscape 2), but hidden behind the scenes is a totally separate set of objects to support. Now when there is a bug, it is important to have a diagnostic layer to determine if it is the GUI, first engine, second engine, first landscape, second landscape, or translation layer.

The conclusion is that regardless of which architecture you use, it should merely not be one which the system accidentally evolved into, but rather the one you intelligently chose, weighing the pros and cons. Now there might be valid reasons to offset the reduced maintainability of using the second or third alternatives, but these alternatives should only be chosen after realizing the drawbacks. However, there can be additional drawbacks with the system architecture shown in the next section.

**System Architecture Alternatives**

It is usually not desirable to have scheduling software in isolation from the rest of the world. Some degree of integration with batch history, the process control system, laboratory Information Management, shipping, modeling, data mining, planning, and ERP systems is almost taken for granted. Translation data agents to retrofit between incompatible architectures are usually unavoidable for some systems, but designed in compatibility gives gains in maintainability, usability, and performance. Figure 2 presents three sample alternatives.
The first alternative is simplest, with three subsystems (the scheduler being one) all using the same landscape, data access, and database. Testing for the landscape and data access only needs to be done once. One can share the context between applications, with the transfer simply being reading objects in memory, with no translation layers or object brokers required.

The second alternative is what can naturally occur if three different programming teams develop three applications with little communication between them. The code maintenance might be three times as much, and the only way to transfer information from one subsystem to another is by writing to the database and then reading what was just written.

The third alternative tries to have some synergy on the data access layer, have different landscape layers for the different applications. There might be reasons for this if one application was constrained by memory size, or a second application needed greater speed for iterations in an optimizer. However, different landscapes will probably exist simply because different programmers worked on them, if there is no overall system architect.

Again, the lesson here is not that the first alternative is always the best, but when a more complex architecture is chosen than the first architecture, it should be chosen intelligently, weighing the benefits and drawbacks, and not simply an accidental result due to lack of design. But just like these strategic issues should be decided before the applications are coded, there are tactical issues that should be agreed to also.

**Tactical Issues**

For each application, some rough measure of the memory footprint is desired. For example, for the system for a large pharmaceutical company, it was totally impractical to have all of the
relevant plant-level S-88 recipes, with all their details, in memory at the same time. Likewise, having the capability to have all schedule details for the next year or too demonstrated that on a 500 Megabyte physical RAM system, Windows XP does not work well once it needs 2.6 Gigabytes of RAM.

There are a variety of ways to reduce the memory footprint, often improving performance at the same time. Some of these ways are more painful than others.

**Denormalizations:** One database denormalization is to store the “root” key of the schedule, recipe, or other object in the database. This leads to faster loading and saving from the database. However, the root and some other denormalized fields should be eliminated from the memory footprint. Of course there will be other variables that are memory only, but think twice before making them instance variables. Instance variables will be in memory for every single instance, but if the memory variable is only needed within the context of one or two methods, then use local variables instead.

**Variable granularity:** One subsystem might need a lot more detail of a model or schedule than another. Rather than getting all detail, the data access layer can selectively only get the detail required. However, if context is shared between subsystems, then a flag must exist saying how much detail is in memory, and there needs to be a query to get the rest of the detail.

**Rehydration:** Instead of storing the entire output of a schedule of disk, store only what is needed to run a simulation to “rehydrate” the schedule. Likewise, instead of having all the detail of every operation, phase, and batch phase connection in memory, store only what is needed, perhaps at the campaign level. Then when something is needed for display purposes, it can be selectively created or “rehydrated”.

**Sparse storing of anchors:** If anchors are allowed at the batch unit, operation, and phase level, and if there are both start and duration constraints, that can add significant memory footprint to a schedule. However, if it is understood that anchors on these lower levels are rarely used, then they can be stored in their own structure. A drawback is that a lookup may be required to access them, which can decrease performance.

**Caching of master recipes:** In my experience with a large pharmaceutical company, one major problem was due the very large footprint of what corresponded to master recipes. One problem is that much of the detail was not needed. However, even leaving out the unnecessary detail, it was not practical to put in every master recipe that could be used for the production orders.

**Storing history:** Each batch, batch unit, operation, and phase has an expected start time and duration, and eventually the actual start time and duration from history. If it is decided to have a field for the start time, duration, and end time for both the predicted schedule and actual history that can add to the memory footprint.

**Further reductions:** Some other, more fundamental things that can reduce the footprint by around a factor of two are changing the time to be an integer (which is fine for resolution down to one minute), using intrinsic linked lists instead of external classes of collections, using real
values as floats instead of doubles, and reducing the need for lookups with hash tables. However, if these measures appear too drastic, remember that it is typically only three or four classes that comprise 90% of the memory footprint. Thus, some of these reductions could be done on only those classes.

**Use of Code Generation**

With these more complex strategies, sparse storage, and further reductions, one can foresee problems with maintainability of the landscape code. This problem can be nearly completely eliminated by using code generation for the landscape(s). The most obvious benefit, more rapid development of the code, is of small importance compared to improvements in code reliability and consistency. In my experience writing ten code generators, one ends up regenerating for the same project twenty times, fifty times, or more, due primarily to schema changes.

**Summary**

There are many choices for the architecture of the scheduling subsystem, but these must be weighed against either using the same architecture for other subsystems, or translating between different landscapes. Choosing an inappropriate architecture is expensive, when code that relies on that architecture needs to be rewritten. However, with code generation, maintaining the landscape code should be a non-issue, much like maintaining the assembly language code behind a high level language.

**References**


