SSIS Package Design Considerations

By:

Sean McCown

Sean@MidnightDBA.com

Microsoft CERTIFIED IT Professional | Database Administrator 2008 Database Developer 2008

Visit www.MidnightDBA.com for free SQL Server tutorial videos.

Contents

Copyright

Intro

The other night I gave a presentation to our local user group about SSIS Architecture and I thought I'd share the discussion here as well.

This is actually part of a beginner's course on SSIS I'm teaching, only instead of just zipping through the various tasks as fast as possible, I'm dedicating some real time to how to design your packages in a way that's both supportable and extensible. I know, what a concept, huh?

So the discussion the other night was on what to do when you have multiple tables to load. Do you put them in a single package or in multiple packages with a single table load in each one? And of course like everything else in databases, there's no solid answer that works across the board. You have to look at every scenario you write individually and make this decision again and again. It's really not that bad though because after a while you start getting the feel for it and it becomes much easier.

Before I get into specifics though I'd like to say a couple words on why this is even necessary. I've been supporting both DTS and SSIS for many years now and I have seen all types of architectures for processes and they're not all created equal. Some are much easier to support from the DBA side, and to maintain from the dev side. So you have to decide what your goals are for your process. Is your goal to just move the data as fast as possible with little regard to support or extensibility? Is your goal to be able to recover quickly should something go wrong? Or maybe your goal is to be able to easily move the process to different servers, or just move part of the process to a different server. Of course these are just some of the possibilities, but the point is that you can't architect a process until you know what your goals are. How would you know when you're successful unless you have something concrete to measure it against?

We see this in HA scenarios a lot don't we? Someone will say they need HA for their big box and they'll instantly put in a cluster… probably because it sounds cool. A few months later something typically corrupts the data in some way and they're shocked to find out that the cluster hasn't protected them. This is because they didn't bother writing down their HA goals so they could measure the solution against them.

I'll also add real quick that this is the same decision process you have to go through for replication scenarios. How do you want to divide up your publications? It's really impractical to think that you'll have 80 tables in the same publication because if something goes wrong and you have to re-synch the data via snapshot, then you greatly increase the chance of it taking a long time. If you break it up though, you can just re-synch the tables that got messed up. Initially though you have to decide how you want to slice it up. Do you want to do it by table activity (putting the most updated tables together), or by relationship (PK/FK tables go together), or maybe by size (big tables get their own publication)? These are the decisions you have to make when you do replication, and you have to make these types of decisions when you write an ETL process too.

Here are the general guidelines we landed on with some more detailed discussion on each one.

Single Package Considerations

Put your tables in a single package WHEN:

1. Have a lot of interdependencies between tables.

2. Network, data, and servers are really stable. (won't need to restart often)

3. May need to change package-level info frequently.

4. You have non-sql dependencies.

Now that I've given the raw list I'll discuss each item. I'm doing it this way so you advanced people out there can just look at the list and get on with your day if you wish.

1.   **Have a lot of interdependencies between tables**. It's not uncommon in ETL processes to have some interdependencies like PK/FK or value lookups. If your process has these elements it's more difficult to orchestrate them if the tables are in separate packages. Let's look at an example of a PK/FK relationship. Ordinarily you'd want the PK table to load and then the FK table. And this can be accomplished in it's simplest form by using a single package and a multiple package method. What you lose in the multiple package method though is the ability to elegantly lookup values in the FK table that don't happen to exactly match the PK table. You also lose the ability to run a query and go back and add the PK value and then try the FK insert again. Oh sure, you could do it in multiple packages by persisting the bad values to a table and then having another package go back after ETL and running through a clean-up process, but that's not elegant and you have to now delay your business cycle for an extra step. Another good scenario is where you'd want to perform replacement lookups on data coming in. So let's say that you have customer IDs and you want to do a lookup and replace them with the customer

name instead. And to make things even more complicated, let's go ahead and say that those values aren't necessarily stored in the same system. Or maybe you're looking up billing info for each customer ID and the billing system is Oracle, or just a different SQL Server on the network. You can't easily perform that lookup if you've got your table loads in separate packages. In this case you have to load the table and then in a later step read that table and do the lookups and perform updates to insert the values. This is far from elegant and I'll just go ahead and say it's stupid. You're going to grossly increase the time it takes you to do ETL and delay your business processes as well as put pressure on the DB both from a concurrency viewpoint and a log viewpoint. However, if you have the tables in the same package you can perform inline lookups as you're loading the data and the process is elegant and smooth.

I see too many people trying to force a multiple package architecture because they've seen the benefits in other areas, but they fail to see that forcing it where it doesn't belong negates any benefits they gain.

2. **Network, data, and servers are really stable. (won't need to restart often).** One of the main advantages of using multiple packages in a process is that it can make it easier to recover from errors. However, if your load process has proven to be really stable, then that's not as much of a concern for you. In this case you've got a well-managed frontend, with good, solid data and it ETLs with no problems. On top of that, perhaps both servers are in the same data center and are managed well by the network and server teams. So recovering from a failure is something you just don't have to deal with often enough to worry about. Sure, it happens maybe once ever 9mos or 1.5yrs, but big deal. It takes a little extra effort to get it going again, but what you gain in management of the process more than outweighs that occasional inconvenience. So if your scenario is stable, I urge you to strongly consider just throwing everything into a single package.

3. **May need to change package-level info frequently.** If you're in the type of place where you have to change things at a package level more than usual, then a single package is where you want to put your load. Package-level stuff includes logging, or security level, or event handling, etc. Now, the question comes up right away, why would you be changing these things a lot? Well, the most common reason is because you're in a shop that has little experience writing and supporting SSIS and you're still finding your way. I saw this happen with my last gig. They had never really done much with SSIS and they chose to put every table into its own package (over 100 tables to load)

so there were about 120 or so packages in total. Well, they changed logging options something like 4 times before they decided what they wanted to actually see. And they had some turnover and had to change the alert emails that got sent out. And they even had to change the security level a couple times while they discovered what all the levels meant. Now, I told them to not use multiple packages, but they knew best so it's their game. And of course, every change they made they had to make 120 times. So simple things became very tedious. Now of course, most of them are on more of a single package model, but there are still some hangers. So be honest with yourself and if you know your shop doesn't have much experience with it, then lean towards the single package model.

4. **You have non-SQL dependencies.**

Do you have non-SQL dependencies? Things like FTP or file copies, deletes, etc? If so then using multi file doesn't gain you anything for recovery really. You can setup those dependency steps in the package with each table sometimes, but it can get complicated fast. A master package can help with this, but you're back to not gaining much really because you still have to run the entire package. the problem here is having to run the dependency steps to get the table going again. So what are you gonna do, put all the dependency steps into separate packages… that makes the whole thing much harder to read and to support. The point here is to have a process you can support easily and that means being able to figure out what's going on.

OK, so that's mostly it for single package design. It may or may not be inclusive, but you guys get the idea. If you come up with something major I left out send it to me and I'll add it to the list.

Multiple Package Considerations

Now on to multiple package design. Here's the list of factors that could lead you down this road with discussions following:

1. Network, data, servers aren't reliable and process is likely to need to be restarted often.

2. Tables are more autonomous (not deciding factor on its own).

3. Not doing any logging, or logging is set in stone and won't change.

1. **Network, data, servers aren't reliable and process is likely to need to be restarted often.** What can I say? Sometimes you just know that your data is messed up or that your network goes down a lot, or something else typically happens to halt your process on a regular basis. So if your environment, for whatever reason, proves itself to be unreliable in a way that hinders your ETL, then it would probably be easier for you if you broke that process up into as many separate steps as possible so you'll be able to easily pick up where it left off and not have to change anything. And I have no real number I can quantify this for you with. It really just depends on your personal threshold for putting up with BS. I've met quite a few DBAs (I'm certainly not one of them) who really don't mind getting pulled out of bed every other night to get a process going again. I've even managed one of them at my last company. And when you explain to them that we need to find a way to make the process more robust they answer with, I don't mind getting up all the time so there's no reason to make my job easier in that way. Well, the company minds, and the business minds because it's a delayed process so let's fix it. Let's at least speed the troubleshooting and resolution as much as we can if we can't do anything about it. Anyway, that was just a small tangent. The point is that having your table loads in separate packages can help you recover from frequent failures.

2. **Tables are more autonomous (not deciding factor on its own).** If your tables have nothing to do with each other, that is, if they can be loaded in any order, then you are able to consider putting them all in their own packages. Now, that doesn't make it the deciding factor, but it means you're able to do it without it adversely effecting your ETL. However, you still have the supportability angle to consider so I'd personally still consider a single package, especially if all other factors (reliability, etc) remain the same. If you prefer multiple packages though, this would make it a good candidate.

3. **Not doing any logging, or logging is set in stone and won't change.** This is mainly for those of you who have your logging levels set in stone. I know none of you are stupid enough to not do any logging in your packages. The truth is that errors do happen and to leave yourself in such a bad position as to not be logging anything is just irresponsible and like I said, stupid. However, one of the big reasons I gave above for choosing a single

package is that you may be making frequent package-level changes. So this is just the opposite side of that argument. If you're not making frequent changes, then that's something you could use as a deciding factor on whether to use multiple packages or not. Think about it this way, if you've got package-level changes to make would you rather make them once, or 80 times?

Solutions

So with all of this discussion you'll be glad to know that you're not limited to just those 2 choices. It's more than just a decision to put your tables all together or by themselves. That's the good news. The bad news is that now your choice on how to architect your package just got a lot harder.

Here's more or less your full list of choices with discussions.

1. **You can use a single package for everything if it's small and stable enough.** I think I've discussed this one enough above, but I wanted to give a full list of choices in one place.

2. **Put all tables in their own package.** This is the same thing I just discussed above. Again, it's part of the list for completion.

3. **Use a master package.** If you're going to use multiple packages, then the best way to control them is with a master package that calls each one in order. You maintain workflow, parallel execution, and you can even add back in your non-SQL dependencies. If you absolutely insist (or are forced) on using multiple packages, then this is your best bet for making it a success. Your other option is to call each package as a separate agent job step. I'll discuss that later. Using a master package also allows you to document the process inside the package itself, which is something you just don't get with multiple packages by themselves. Unfortunately, even though you can use a master package and have many more centralized control at your fingertips, and even though you get sequence containers, foreach loop containers, and the like, using master packages flattens your process. What I mean by that is that instead of seeing the actual tasks that are being run in the package, all you see is a field of 'execute package' tasks. So while you still have the control flow tasks at your disposal, your package is flat because you can't tell what

each of the steps is doing by looking at it. Also, if you do use a master package to tie all of your individual table loads together, you lose the recoverability you sought in the first place by splitting up your tables… or you can at least. If you've got just plain table loads inside your master package, then you can always run the packages individually if you need to recover in the middle of the process. However, if you've got non-SQL dependencies or foreach loops, etc, then you haven't gained anything because you've strapped yourself to the constraints of a single-package process.

Serially in Agent Job – If you find yourself with the multiple package design, your choices again are using a master package or setting each package as a different step inside an agent job. This is the worst way you could possibly implement this because you lose everything that makes SSIS what it is. You have no parallelism at all. That is to say that all your loads run serially and nothing runs concurrently. I remember I had this one process with this exact design that took 1.5hrs. I changed it to run some of the loads concurrently and shaved 40mins off the time right away. So this is definitely worth looking into. Of course, you could use this method to force a serial run if your server doesn't have the power to handle parallel loads, but it's still easier to handle this in a single or master package design. The only real advantage to this design is that you gain the ability to very easily pick up in the middle of the process if something fails. You can simply restart the job from the failed step and all's well. So from that aspect it's very easy to support, but in every other aspect it's more difficult. So I don't think it's worth it except in the most limited of circumstances, but as long as you're aware of all the shortcomings and you've weighed your options, I see no reason why you couldn't choose this if you see no other viable option for your environment. It's really a catch-22 situation though. This solution really isn't that bad if you've only got a handful of steps, but if you've only got a handful of steps then you really don't need it. And this comes in handy when you've got a lot of steps, but having a lot of steps makes this harder to sustain.

4. **You can group tables in packages.** Another viable solution is to split the difference and go with a hybrid approach. Instead of everything being in a single package or in its own package you can group like tables into their own packages. So let's take the example of loading 80 tables. You can put them all in a single package, or you can put them into 80 different packages, or you can group them into any number of packages; in this case let's say 8 packages of 10 tables each. The advantage of this is that in finding logical split points to separate your package into, you can recover from errors much

easier than with a single package process, or from a multiple package process. This way you can setup groupings by either PK/FK, or by size, or even by server. So if you've got a load that pulls data from different servers, then having the load from each server in its own package is a good way to split the process. This way you can also move the different pieces of the process to different servers if you need to. You can also reuse the process for other ETLs if you need the data on more than one server. And it doesn't have to be even, right? You can put 3 tables in one package, and 15 in another, etc. Just split them logically. One large process I split went from a source box, to a staging server, to a destination. So I logically split them into 2 packages: one that went from the source to staging, and one that went from staging to the destination. So each one was a supportable process that could easily be moved from one server to another.

OK, so that's a lot of discussion on a topic that seems easy on the surface, but you can see that you've got some things to consider. Just remember to always have a plan so you'll know what you want to get out of your process. Then you can evaluate your process against your plan and see if you've accomplished what you set out to do.

Feel free to email me with questions or comments if you have any.

Also, here's the video of the presentation if you'd like to hear all of this happening in realtime.