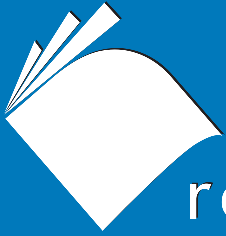


realtimepublishers.com™

The Definitive Guide™ To

Scaling Out SQL Server 2005

Don Jones

Chapter 9: Scaling Out at the Application Level 190
Applications: The Scale-Out Bottleneck 190
Common Application Problems in a Scale-Out Environment 190
Server-Centric View of the World 192
Intolerance of Longer Data Operations 194
Inflexible Data Access Models 196
Challenges in Moving to Scale-Out 197
Architecting a Complete Scale-Out Solution 198
The Data Layer 199
The Middle Tier 201
The Web Tier 204
The Client Tier 205
Converting Existing Applications for Scale-Out 206
Key Weaknesses 207
Conversion Checklist 208
Summary 208

Copyright Statement

© 2005 Realtimepublishers.com, Inc. All rights reserved. This site contains materials that have been created, developed, or commissioned by, and published with the permission of, Realtimepublishers.com, Inc. (the "Materials") and this site and any such Materials are protected by international copyright and trademark laws.

THE MATERIALS ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. The Materials are subject to change without notice and do not represent a commitment on the part of Realtimepublishers.com, Inc or its web site sponsors. In no event shall Realtimepublishers.com, Inc. or its web site sponsors be held liable for technical or editorial errors or omissions contained in the Materials, including without limitation, for any direct, indirect, incidental, special, exemplary or consequential damages whatsoever resulting from the use of any information contained in the Materials.

The Materials (including but not limited to the text, images, audio, and/or video) may not be copied, reproduced, republished, uploaded, posted, transmitted, or distributed in any way, in whole or in part, except that one copy may be downloaded for your personal, non-commercial use on a single computer. In connection with such use, you may not modify or obscure any copyright or other proprietary notice.

The Materials may contain trademarks, services marks and logos that are the property of third parties. You are not permitted to use these trademarks, services marks or logos without prior written consent of such third parties.

Realtimepublishers.com and the Realtimepublishers logo are registered in the US Patent & Trademark Office. All other product or service names are the property of their respective owners.

If you have any questions about these terms, or if you would like information about licensing materials from Realtimepublishers.com, please contact us via e-mail at info@realtimepublishers.com.

[**Editor's Note:** This eBook was downloaded from Content Central. To download other eBooks on this topic, please visit http://www.realtimepublishers.com/contentcentral/.]

Chapter 9: Scaling Out at the Application Level

A major step in creating a SQL Server-based scale-out solution is creating the client application (or applications) that end users will utilize. These applications must be designed to accommodate your back-end scale-out technique, whatever that is, and it can be one of the most difficult parts of creating a scale-out solution. This chapter will focus on the client-side (and other application tier) design techniques that can be used to create an effective scale-out solution.

You need to keep in mind, though, the precise scale-out technique you've selected for SQL Server. For example, if you've decided to scale out by creating multiple SQL Server installations, all containing the same data and using replication to keep one another updated, then you'll need to build your client applications in a particular way. If, however, you're using a federated database—where each server contains only a portion of the data, and views and other techniques are used to draw it together—there is a whole different set of client-side techniques. This chapter will examine them all, but you'll need to select the ones that are most appropriate for your particular scale-out solution.

This chapter won't address prepackaged applications. For the most part, packaged applications aren't subject to your reprogramming or rearchitecture, meaning you're pretty much stuck with what you get. Some packaged applications—such as SAP—have tremendous flexibility and can be re-architected to achieve better scale-out. However, most such applications have *very* specific and often proprietary guidelines for doing so, far beyond the scope of what this chapter can cover.

Applications: The Scale-Out Bottleneck

Applications aren't a scale-out bottleneck in the performance sense of the word, but they can be a bottleneck in your solution development process. Creating applications that effectively use a scaled-out back-end requires entirely different development and data access techniques than in a smaller SQL Server-based solution.

Common Application Problems in a Scale-Out Environment

The overall problem with typical applications is that they're designed to work with a single SQL Server computer. When an application finds itself in an environment where multiple SQL Server computers exist for the same database, all the common data access techniques and development models become useless. For a few examples, use the simple diagram in Figure 9.1 as a reference.

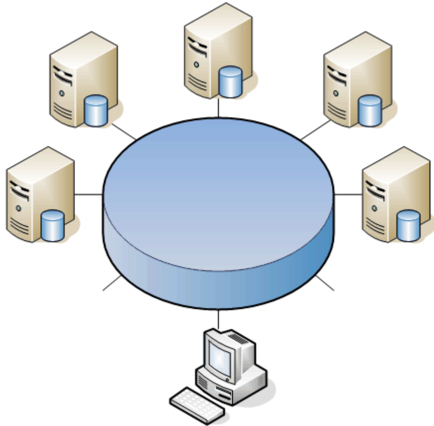


Figure 9.1: Scaled-out SQL Server environment.

In the first scenario, suppose you've designed your SQL Server database so that each server maintains a complete copy of the data, and that replication is used to keep the various copies in sync with one another. So which database server does a given client computer access? Is it a simple matter of selecting the one closest to it? How does it go about doing so? Would you prefer that it somehow select the server which is least busy at the time? That's even more difficult; while static server selection might be something you could put into the client's configuration, being able to dynamically select a server based on server workload is more difficult. You can't simply use technologies like Network Load Balancing (NLB), because that technology assumes that every server has completely identical content. In a replication scenario, servers *won't* have completely identical content—not all the time. Consider how NLB might work in this scenario:

- Client needs to add a row to a table. NLB directs client to Server1.
- Client immediately needs to retrieve that row (which has probably had some unique identifier applied, likely through an Identity column). NLB directs client to Server2 this time, but Server2 doesn't have the new row, yet, due to replication latency.

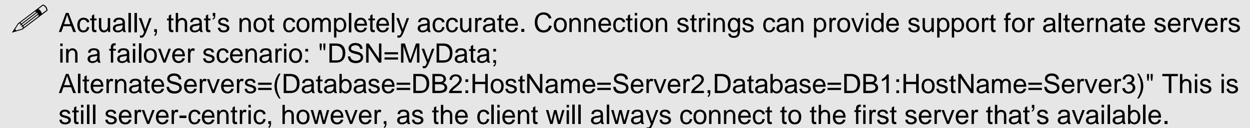
Clients would instead need some logic of their own to select a server and then stick with it (a technique referred to as *affinity*) through a series of operations; that's not something NLB (which was designed to work with Web farms) is designed to do.

Consider a second scenario, in which each server contains a *portion* of the overall database, and objects like distributed partitioned views (DPVs) are used by clients to access the database as if it were contained on a single server. As I explained in Chapter 4, the server physically containing most of the requested data can best handle the query; should the client try to figure that out and query the DPV from that server? If not, which server—as all of them are technically capable of handling the query to the DPV—should the client select? If all clients select a single particular server, you're going to bottleneck at that server eventually, so you do want some way to spread them all out.

In the next few sections, I'll discuss some of the specific components that make client applications more difficult in a scale-out solution.

Server-Centric View of the World

Figure 9.2 illustrates a common problem with client applications: Connection strings. Many client applications are designed to connect directly to a SQL Server computer using ActiveX Data Objects (ADO) or ADO.NET, and use a connection string to identify the server. Unfortunately, connection strings are, by definition, server-centric. In other words, a single connection string can connect to only a single server.



Actually, that's not completely accurate. Connection strings can provide support for alternate servers in a failover scenario: "DSN=MyData; AlternateServers=(Database=DB2:HostName=Server2,Database=DB1:HostName=Server3)" This is still server-centric, however, as the client will always connect to the first server that's available.

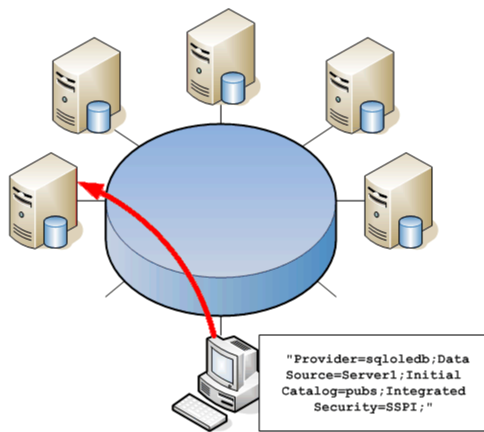


Figure 9.2: Server-centric connections.

The problem with this technique in a scale-out solution is that it restricts the client to just a single SQL Server. If the client is expected to connect to different SQL Server computers (if the client is running in a different office, for example, which has its own server), the client either has to be changed, or has to be written from the outset to have multiple connection strings to choose from.

And just because you have a multi-tier application doesn't really change this problem; while clients in a multi-tier application aren't usually server-centric from a SQL Server viewpoint, they do tend to be designed to work with a single middle-tier server, which in turn uses a standard, server-centric connection string to work with a single SQL Server computer.

In a Web farm—which is the most common model of a scale-out application—this problem would be solved by using load balancing. Clients—or middle tier servers or whatever—would connect to a single virtual host name or IP address, which would be handled by some load balancing component (such as NLB, or a hardware load balancing device). The load balancing component would then redirect the client to one of the back-end servers, often in a simple *round-robin* technique where incoming connections are directed, in order, to the next server in sequence. I've already discussed why this doesn't work in a SQL Server environment: Clients often need to perform several tasks with a given server in short order before being redirected. Sometimes, opening a connection and leaving it open will maintain a connection with the same server, but that can become difficult to manage in middle-tier servers where dozens or hundreds of connections are open at once, and where connections are pooled to improve performance.

What's the solution to server-centric connections? Well, it depends on your SQL Server scale-out design. Because at some point *somebody* has to use a connection string—whether it be a client or a middle-tier—that's somebody is going to have to incorporate logic to figure out which connection string to use (or, more accurately, which server to put into the connection string). One straightforward example of this might work for an environment where multiple SQL Servers contain the entire database and use replication to stay in sync; as illustrated in Figure 9.3, clients (or middle-tier servers) might examine their own IP address, match it to a list of server IP addresses, and thereby connect to the server nearest them (similar to the way in which a Windows client selects an Active Directory domain controller).

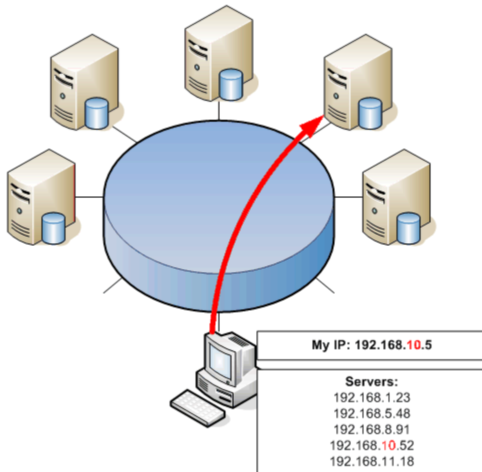


Figure 9.3: Dynamically selecting a server.

Of course, this technique requires that the application have a complete list of servers. To make the application more robust and longer-lasting, you might have it actually query the list of servers from a central database, enabling the server lineup itself to change over time without having to deploy a new application.

A more advanced solution might be to build your own equivalent of a network load balancing solution, however. Figure 9.4 illustrates this technique.

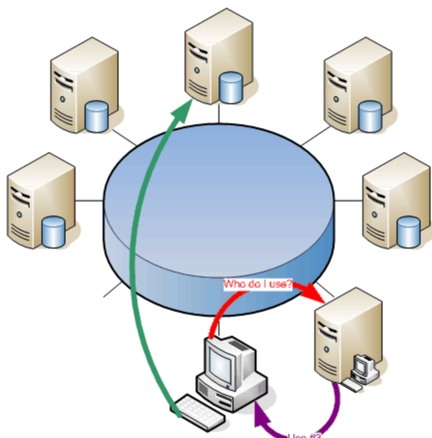


Figure 9.4: Building a redirector service.

realtimepublishers.com®

In this example, the redirector is able to determine which server is least busy, located the closest, or whatever other criteria you want to use. It then informs the client which server to use. The client then makes a direct connection to that server, and maintains the connection for however long it wants, allowing it to complete entire transactions with that server. The redirector service might provide a timeout value; once the timeout expires, the client would be required to go back and get a new server reference. This helps ensure that the redirector can continually rebalance load across servers (for example). If the list of available servers evolves over time, only the redirector needs to be updated, which helps reduce long-term maintenance.

This redirector service can also be implemented in a scenario where your scale-out solution uses a federated database. Clients might be designed to submit queries to the redirector first, which might conduct a brief analysis and direct clients to the server best capable of handling that particular query. That would require significantly more logic, and you wouldn't necessarily want the redirector to try and figure out which server contained the most data (that would reduce overall solution performance, in most cases), but the redirector might be able to realize that a client was trying to query a lookup table's contents, and direct the client to the server or servers that physically contain that data.

The idea, overall, is to find a way to remove the single-server view of the network, and to give your solution some intelligence so that it can make smarter decisions about which server to contact for various tasks. As much as possible, those decisions should be centralized into some middle-tier component (such as the redirector service I've proposed), so that long-term maintenance of the decision-making logic can be centralized, rather than spread across a widely-distributed client application.

Intolerance of Longer Data Operations

While the whole point of a scale-out application is to speed things up, especially long-running operations—such as reports that require large amounts of data to be collated—can still take a while to complete, especially when several servers must work together to provide the data. Client applications—and, more frequently, their users—are often intolerant of longer operations, though. One way to combat this problem—even in a scale-up application—is to implement asynchronous querying capabilities. Figure 9.5 shows what I'm talking about.

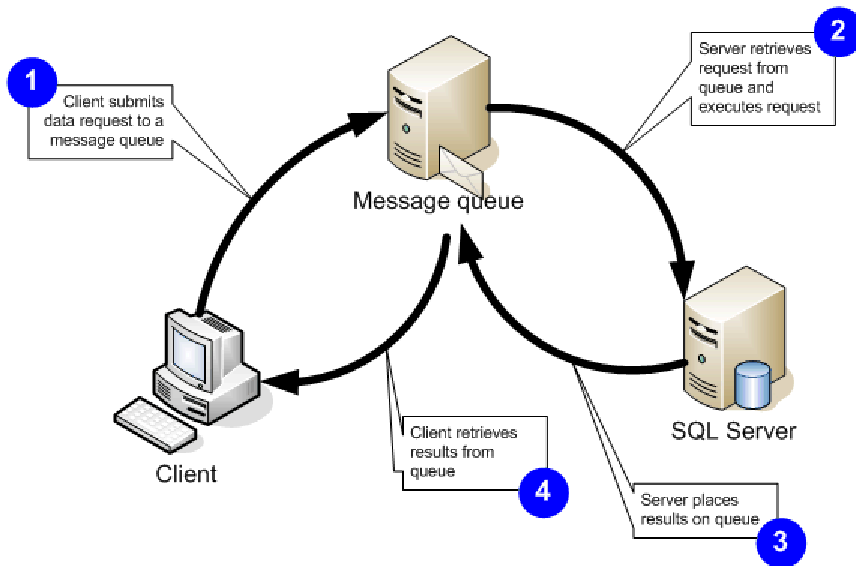


Figure 9.5: Asynchronous querying.

In this example, clients use message queuing to submit data requests. An application running on the server (or a middle tier) retrieves the requests and executes them in order, placing the results back on the queue for the client to retrieve. While this obviously isn't appropriate for typical online transaction processing (OLTP) data requests, it's perfectly appropriate for ad-hoc reports and other data that isn't needed instantly. By moving these types of data requests into an asynchronous model, you can ensure that they don't consume excessive server resources, and by building your client applications around this model you can give users an immediate response ("Your request has been submitted") and delayed results ("Your report is now ready to view") in a more acceptable fashion than simply having users stare at an hourglass.

Even certain OLTP applications can use this technique. For example, in an events-ticketing application, submitting ticket purchases to a queue helps ensure that tickets are sold in a first-come, first-served fashion. Customers might not receive instant confirmation of their purchase, especially if the queue has a lot of requests on it for a popular event, but confirmation wouldn't take long. Because the actual processing would be accomplished by a middle-tier application, rather than the client, the business logic of connecting to the scaled-out back-end could be more easily centralized, as well.

> ☞ My preference, as you'll see throughout this chapter, is to never have client applications connecting directly to the scaled-out back-end. Instead, have clients use a middle-tier, and allow that tier to connect to the back-end for data processing. This model provides much more efficient processing, eliminates the need for client applications to understand the scaled-out architecture, and helps to centralize the connectivity logic into a more easily-maintained application tier.

realtimepublishers.com®

Inflexible Data Access Models

The previous two sections have both illustrated how client applications are traditionally written with fairly inflexible data access models. Other examples of inflexibility exist, but the single best solution is to move to a multi-tier application architecture. In fact, any good scale-out solution will use at least a three-tier model of some kind, simply so that the applications used by end-users need to have as little hardcoded into them as possible. Client applications should be insulated (or *abstracted*) from the database servers, from the selection of what database server they'll use, and so forth. This allows the back-end to evolve over time, and requires you to maintain only the smaller middle-tier servers to keep up with that evolution. Figure 9.6 illustrates this basic software development concept.

The additional benefits of multi-tier design have been known for years:

- Connection pooling allows SQL Server resources to be used more efficiently, by aggregating multiple client connections across a smaller number of actual database connections between the middle tier and SQL Server itself.
- More business logic can be encapsulated into the middle tier, reducing overhead on SQL Server while maintaining fairly centralized business logic code that's easier to maintain over the long term.
- Client applications can become simpler and easier to create, and can last longer between updates since some of the solution's evolution can be restricted to higher tiers.

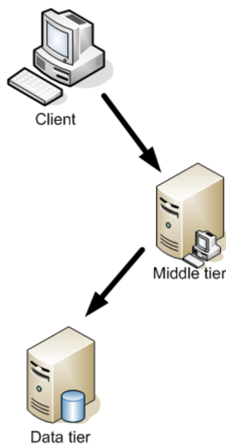
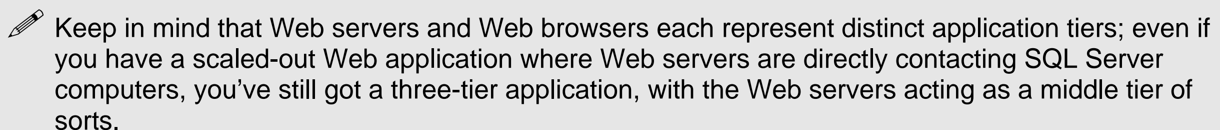


Figure 9.6: Three-tier application design.

While it's obviously possible to build effective, scaled-out, 2-tier (client and SQL Server) applications, it's not the most efficient or logical approach.



Keep in mind that Web servers and Web browsers each represent distinct application tiers; even if you have a scaled-out Web application where Web servers are directly contacting SQL Server computers, you've still got a three-tier application, with the Web servers acting as a middle tier of sorts.

Challenges in Moving to Scale-Out

Moving from an existing 2-tier application environment to a scaled-out solution (regardless of how many application tiers you build) can be painful, depending on how the original client applications were written to begin with. For example, consider the ASP.NET code shown in Figure 9.7 (I'm showing this in Notepad to help maximize the amount of code you can see).

Can you spot the problems? There are quite a few. Highlighted in red is the big one from a scale-out perspective: This application is using a common ASP.NET technique of storing the connection string in the web.config file, and then connecting directly to the database server. Under certain scale-out scenarios, as discussed previously, this simply won't work. Instead, you might need to create a new class that returns a connection, and design that class to select the appropriate database server. Or, better yet, create an entirely new tier that can be used to figure out which server is appropriate, and return the appropriate connection string. Best would be an entirely new tier that handles the data processing itself, eliminating the need for the ASP.NET application to connect directly to SQL Server in the first place.

Another problem—highlighted in yellow—is the application's use of dynamically-generated SQL statements, which are often subject to SQL insertion attacks. This has no bearing on the application's scale-out, but it is a poor practice that should be rectified when the data processing portion of the application is moved to a middle tier.

opt.aspx.vb - Notepad

File Edit Format View Help

```
Private LocRM As New ResourceManager("cc.strings", GetType(opt).Assembly)

Private Sub Page_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.

    If Request.QueryString("op") Is Nothing Or Request.QueryString("id") Is Nothing Then
        Response.Redirect("http://cc.realtimepublishers.com")
    Else
        lblID.Text = Request.QueryString("id")
        lblop.Text = Request.QueryString("op")

        'localize labels
        lblIntro.Text = LocRM.GetString("resOptOutIntro")
        lblConfirm.Text = LocRM.GetString("resOptOutConfirm")
        btnConfirm.Text = LocRM.GetString("resOptOutConfirmButton")
        btnCancel.Text = LocRM.GetString("resOptOutCancelButton")

    End If

End Sub

Private Sub btnConfirm_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles

    Dim oConnection As New SqlClient.SqlConnection(ConfigurationSettings.AppSettings("sqlconn"))
    oConnection.Open()
    Dim oCommand As SqlClient.SqlCommand
    Dim sSQL As String, sType As String
    Select Case lblop.Text
        Case "gen"
            sSQL = "UPDATE Users SET userNewsletter = 0 WHERE userID = " & lblID.Text
            sType = LocRM.GetString("resOptTypeGeneral")
        Case "ser"
            sSQL = "DELETE FROM UserSeriesAlerts WHERE userSeriesAlertID = " & lblID.Text
            sType = LocRM.GetString("resOptTypeSeries")
        Case "top"
            sSQL = "DELETE FROM UserTopicAlerts WHERE userTopicALertID = " & lblID.Text
            sType = LocRM.GetString("resOptTypeTopic")
        Case "chap"
            sSQL = "UPDATE UserPubs SET userPubNotify = 0 WHERE userPubID = " & lblID.Text
            sType = LocRM.GetString("resOptTypeChapter")
    End Select
    oCommand = New SqlClient.SqlCommand(sSQL, oConnection)
    oCommand.ExecuteNonQuery()
    oConnection.Close()

    'set status label, hide unused controls
    lblStatus.Text = LocRM.GetString("resOptOutComplete").Replace("%type%", sType)
    btnConfirm.Visible = False
    btnCancel.Visible = False
    lblIntro.Visible = False
    lblConfirm.Visible = False
```

Figure 9.7: ASP.NET application – not ready for scale out.

Figure 9.8 illustrates how this application might need to evolve to work well in a scale-out scenario.

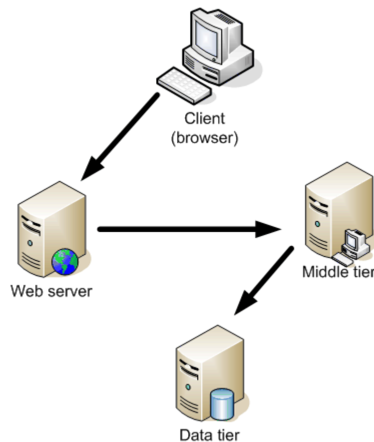


Figure 9.8: Scaling out the Web application.

Unfortunately, this sort of change is definitely nontrivial: Every page in the ASP.NET, based on the example you saw, will need significant modifications. An entire middle tier will have to be constructed, as well. Essentially, much of the application will have to be rewritten from scratch. *This* is why I refer to client applications as the bottleneck in a scale-out solution: Creating the scaled-out SQL Server tier can seem easy compared to what you have to do to make a robust client (and middle) tier that's compatible with it.

Architecting a Complete Scale-Out Solution

If you have an existing application that you're converting to scale-out, there may be a tendency to try and change as little as possible to get there. Similarly, if you're building an application solution from scratch, you may be tempted to just start with the data tier—which is the most interesting, to some folks—and worry about the rest later. Both approaches will land you in trouble, because scale-out requires a properly thought-out *solution*¸ from beginning to end, before you start implementing anything. While most of this book has focused on SQL Server's role in scale-out, that doesn't mean the rest of the solution can be ignored.

Even if you're starting with an existing application, architect your scaled-out solution from scratch, and then see what parts of your existing application fit in, and where they fit in. You may be in for a lot of reprogramming—in most cases I've see, that's what happens—but it's far better to have a properly-designed scale-out solution that requires a lot of work than to have one that *didn't* get much work, but also doesn't work very well.

In the next few sections, then, I'll discuss from-scratch scale-out architecture, taking it one tier at a time.

realtimepublishers.com®

The Data Layer

Most of this book is already focused on the data tier, so at this point I'll summarize your basic options. You do need to decide, at this point, what kind of scale-out approach is going to be appropriate for your situation. For example, Figure 9.9 illustrates a solution where multiple database servers each have a complete copy of the database, and use replication to keep one another in sync.

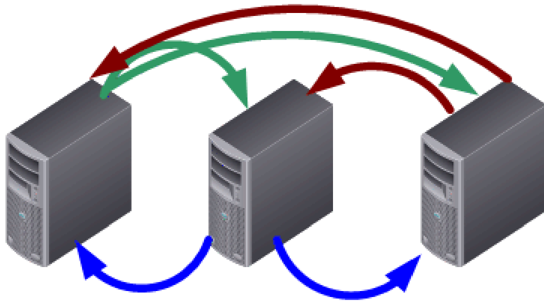


Figure 9.9: Scale-out through replication.

This scenario might be appropriate in a solution where the users are geographically distributed. Each location could have its own server, using WAN-based replication to stay in sync. Benefits include the ability for users to always access a local database server, and the remainder of the solution wouldn't be terribly different from a single-server solution. In fact, this is probably one of the easiest scale-out solutions to retrofit. However, downsides to this approach can include significant WAN utilization and high replication latency. That means users at each location have to be accepting of the fact that any data frequently updated by users at other locations may be out of date a great deal of the time.

Another possible use for this technique is load balancing. In this example, the servers would all reside in the same location, and users would be directed between them to help distribute the workload. This is also relatively easy to retrofit an existing solution into, although changes obviously need to be made to accommodate the load balancing (I discussed these points earlier in the chapter). Replication could be conducted over a private, high-speed network between the servers (a private Gigabit Ethernet connection might be appropriate), although particularly high-volume applications would still incur noticeable replication latency, meaning each server would rarely, in practice, be completely up-to-date with the others.

Figure 9.10 illustrates a different approach. Here, each server contains only a portion of the database. Queries are conducted through DPVs, which exist on each server. As needed, the server being queried enlists the other servers—through linked servers—to provide the data necessary to complete the query. This is a *federated* database.

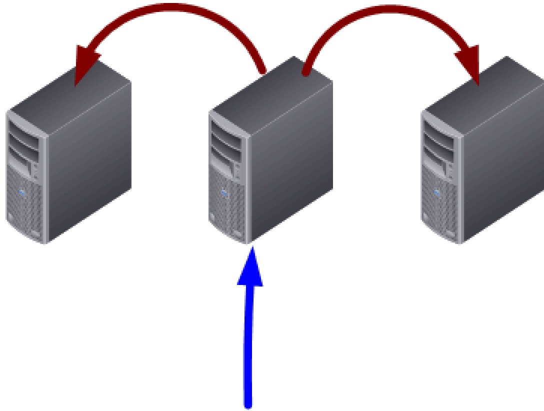


Figure 9.10: A federated database.

Figure 9.11 shows a minor variation on this them. Here, a fourth server contains the DPVs and enlists the three servers containing data to complete the queries. The fourth server might not actually contain any data; its whole function is to serve as kind of an intermediary. The fourth server might contain tables for primarily static data, such as lookup tables, which are frequently read but rarely changed. That would help the three main servers focus on the main database tables. I refer to the fourth server as a *query proxy,* since, like an Internet proxy server, it appears to be handling the requests even though it doesn't contain the data.

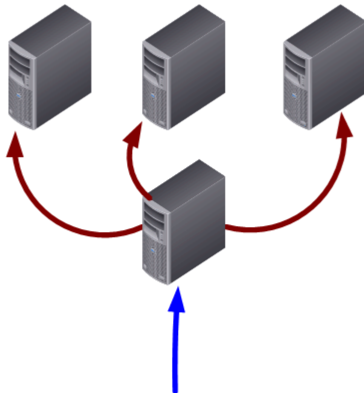


Figure 9.11: Using one database server as a query proxy.

Finally, the last scale-out model is a distributed database, as pictured in Figure 9.12. Here, the database is distributed in some fashion across multiple servers, but the servers don't work together to federate, or combine, that data. Instead, anyone accessing the database servers knows what data is stored where, and simply accesses it directly.

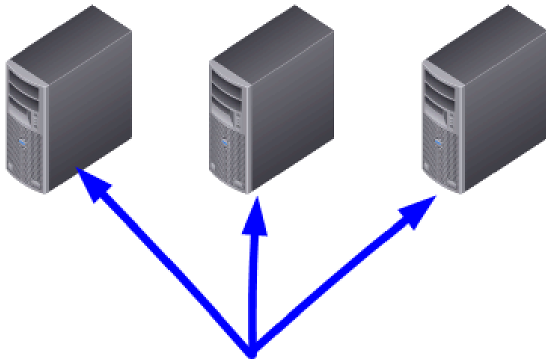


Figure 9.12: Distributed database.

This model has two main permutations. First, the database might be broken up by tables, so that (for example) customer data resides on one server, while order data lives on another. The second way is for the data to be manually partitioned in some fashion. Customers "A" through "M" might be on one server, while the remainder are on another server.

These models don't necessary stand alone, either. For example, you might create a solution where customer data is federated between three servers, and uses DPVs to present a single, combined view of the data. Vendor data, however, might only exist on the second server, while lookup tables live on the third server. This model would combine a distributed database model with a federated database model. You can be creative to help your database perform at its best.

The Middle Tier

Scale-out application need a middle tier. Yes, it's absolutely possible to build a scale-out solution *without* a middle tier, but it'll operate better, and be easier to maintain, with one than without one. Although they can be simple proxies that help remove the need for client applications to understand the back-end (as I've discussed already in this chapter), middle tiers can provide a great deal of logic, help to centralize important business rules, and much more.

Figure 9.13 shows a basic middle tier in operation. Here, one middle tier server receives a request (in blue) that's filled by a DPV. The middle-tier server could technically contact any server hosting that DPV to query it, but it might do a few quick ping operations to see which server is responding fastest, and query the DPV from that one. Or, it might do a quick analysis to determine which back-end server physically contains the majority of the data being requested, since that server can complete the DPV query with the least effort. The back-end then does whatever it's supposed to, enlisting the other back-end servers to complete the query. All that time, whatever application was querying the middle tier needs to know nothing about this back-end work; it simply instantiated some remote object to obtain some data, and it got the data.

realtimepublishers.com®

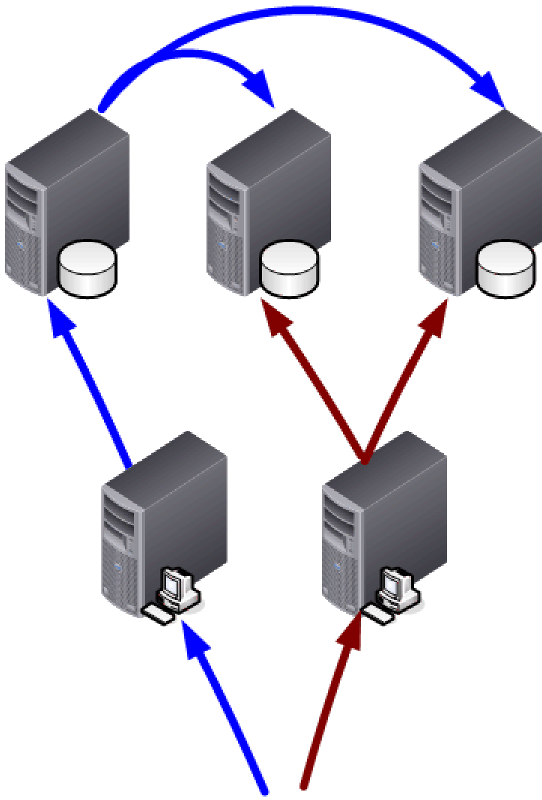


Figure 9.13: Using a middle tier.

A second request (in red) goes to a second middle-tier server. This request might be for data which isn't handled by a DPV, but is rather distributed across two back-end servers. The client application doesn't need to understand this at all; it simply instantiates a remote component on the middle tier server (or accesses a Web service, or something similar), and it gets its data. The middle tier knows where the data is located, and retrieves it.

Using middle tiers offers some distinct advantages:

- Maintaining connections requires resources on SQL Server, which could often be used better elsewhere. While client applications would require, at minimum, one connection to SQL Server per client, a middle tier can get by with fewer connections than the number of clients it services. This is called *connection pooling,* and it can make the back-end operate more efficiently.

How Does Connection Pooling Help?

Any application accessing SQL Server has two choices when it comes to connections: Create one (or more) and leave it open at all times, or just create one when it's needed.

In the first case, the connection requires resources to maintain even when it isn't being actively used. If a client application does this, in fact, the connection will *mainly* be unused, because client applications spend far more time waiting for their user to do something than they do querying data.

In the second case, creating and tearing down connections takes processing power, something SQL Server could be using more efficiently elsewhere.

A middle tier often creates a number of connections and keeps them open. However, it doesn't create one per client; instead, as requests come in, the middle tier selects a currently-idle connection to service the request. This allows connections to remain open, but helps prevent them from being idle, which derives the maximum benefit from the SQL Server resources diverted to maintaining the connection.

- The middle tier is often easy to scale out. Simply create an identical middle-tier server and find a way to load-balance clients across it (perhaps hardcoding some clients to use a particular server, or by using an automated load balancing solution).
- The middle tier can contain business and operational logic that would otherwise require more complex client applications, or would place unnecessary load on SQL Server. For example, the middle tier can be designed to understand the back-end data layout, allowing it to access the data it needs. This removed the need for client applications to have this logic, and allows the back-end to change and evolve without having to redesign and redeploy the client. Instead, the middle tier—which is a much smaller installed base—is reprogrammed. Similarly, operations like basic data validation can take place on the middle tier, helping to ensure that all data sent to SQL Server is valid. That way, SQL Server is wasting time validating and rejecting improper data. If business rules change, the middle tier represents a smaller installed base (than the client tier) that has to be modified.
- You can get creative with the middle tier to help offload work from SQL Server. For example, the middle tier might cache certain types of data—such as mainly-static lookup tables—so that SQL Server doesn't need to be queried each time. Or, clients could cache that information, and use middle-tier functionality to determine when the data needed to be re-queried.

Middle tier applications used to be somewhat complex to write, and involved fairly complicated technologies such as Distributed COM (DCOM). However, with today's .NET Framework, Web services, and other technologies, middle tiers are becoming markedly easier to create and maintain, giving you all the more reason to utilize them in your scale-out application.

☞ A middle tier, can, in fact, be an excellent way of migrating to a scale-out solution. If you can take the time to redesign client applications to use a middle tier, and create the middle tier properly, then the back-end can be scaled out without having to change the client again.

The Web Tier

Many application solutions, these days, are incorporating a Web component, even if they aren't specifically a Web application. In general, the Web tier should be treated as a client tier. In other words, you still should consider using a middle tier, and then allow Web servers to connect to *that*. Figure 9.14 illustrates this arrangement.

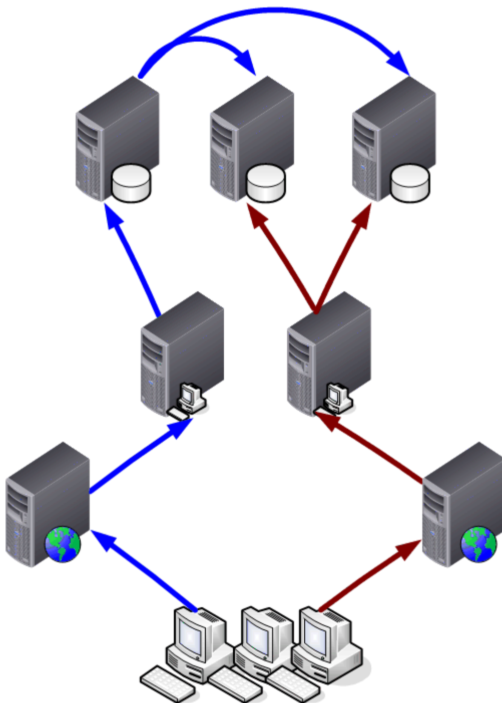


Figure 9.14: Web servers in a multi-tier scale-out solution.

It's very important that Web applications follow the same best practices as any other client application: Minimizing data queried, no ad-hoc queries, and so forth.

✐ There are a few Web applications that are special cases. For example, a SQL Server Reporting Services Web site typically needs direct connectivity to SQL Server, rather than accessing data through a middle tier. When this is the case, you can typically make the back-end more robust to accommodate the direct access. For example, reports might be pulled from a static copy of the database that's created each night (or each week, or however often), rather than querying the OLTP servers.

As your Web tier scales out—Web farms being one of the easiest things to create and expand, thanks to the way Web servers and browsers work—always take into consideration the effect on the middle and back-end tiers. For example, you might determine that each middle-tier server can support ten Web servers; so as you scale out the Web tier, scale out the middle tier appropriately. Always pay attention to the resulting effect on the back-end, which is more difficult to scale out, so that you can spot performance bottlenecks before they hit, and take appropriate measures to increase the back-end tier's capacity.

The Client Tier

The client tier may consist of numerous client applications, but they should all access data exclusively through a middle tier server, as shown in Figure 9.15.

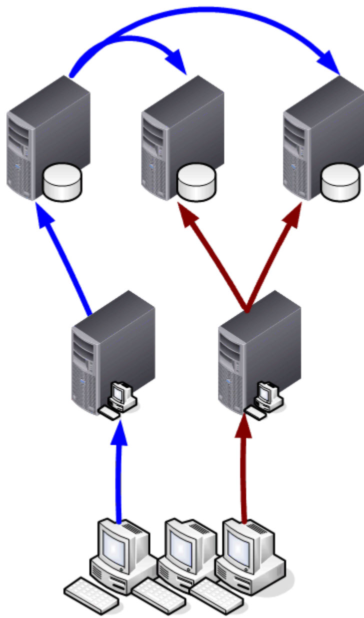


Figure 9.15: Clients access data through a middle tier.

There are a number of pieces of functionality which typically exist in client applications, but which can and should, whenever possible, be moved to the middle tier:

- **Data validation.** When possible, move this functionality to the middle-tier. The middle-tier might provide functionality that allows clients to query data requirements (such as maximum field lengths, allowed formats, and so forth), so that clients can provide immediate feedback to their users, but in general try to avoid hardcoding data validation in the client tier. As the most widely-deployed tier, the client tier is the most difficult to maintain, so eliminating or reducing data validation—which can change over time—helps to improve long-term maintenance.
- **Business rules.** As with data, client-tier maintenance will be easier over the long term if business logic exists primarily on the middle tier.
- **Data access logic.** Clients should have no idea what the data tier looks like. Instead, data access should all be directed through the middle tier, allowing back-end structural changes to occur without affecting how clients operate.

Client should not use (and the middle tier should not allow the use of) ad-hoc queries. Instead, clients should be programmed to use middle-tier components (or Web services, which amounts to the same thing) to query the exact data they require. This helps to ensure that clients are fully abstracted from the data tier and have no dependencies on anything, including table names, column names, and so forth. This technique provides the maximum flexibility for the data tier, and truly makes the middle tier a "wall" between the clients and the data.

Converting Existing Applications for Scale-Out

There's no question that converting existing applications can actually be more difficult than just starting from scratch, especially if you're inheriting an application that you weren't responsible for in the first place. Still, sometimes conversion can be more cost-effective, and so in the next few sections I'll touch on the key areas you'll need to pay attention to in converting your applications.

It probably goes without saying, but just in case: Applications should use all the best practices that I discussed in Chapter 2, such as using stored procedures rather than ad-hoc queries, retrieving the minimum amount of data, and so forth. These practices help applications perform better no matter what kind of environment you're working in.

Key Weaknesses

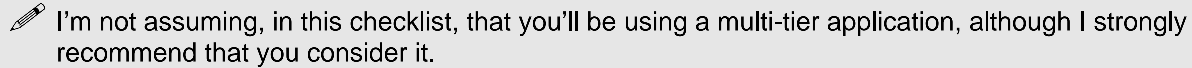
Applications already written for a 3- (or more) tier environment are less likely to have significant weaknesses with regard to scale-out operations, although the tier which accesses data will likely need a decent amount of work to accommodate a scaled-out SQL Server solution. However, many applications are simple, client-server applications that may require extensive work. Here are some of the key weaknesses usually found in these applications, which you'll need to address during your conversion:

- **Direct connectivity.** Applications connecting directly to a data source will need to have that connectivity removed or modified, as appropriate, to understand your new solution architecture.
- **Ad-hoc queries.** Many client applications make use of ad-hoc queries, which are out of place in any database application, but especially in a scale-out scenario. Replace these with calls to stored procedures or to middle-tier components.
- **Caching.** Client applications rarely cache data, although in a scale-out solution—when retrieving data might require the participation of multiple servers—doing so can help improve overall throughput. Clients may be able to cache, for example, relatively static data used for drop-down lists and other lookups, helping to improve overall throughput of the solution.
- **Poor use of connection objects.** Client applications often make poor use of ADO or ADO.NET connection objects, either leaving them open and idle for too long or too frequently creating and destroying them. A middle tier, which can help pool connections, makes connection resources more efficient.
- **Intolerance for long-running operations.** While scale-out solutions are designed to improve performance, sometimes long-running operations are inevitable. Client applications must be designed not to error out, or to use asynchronous processing when possible.
- **Dependence on data tier.** Client applications are often highly dependent on specific data tier attributes, such as the database schema. Clients should be abstracted from the data tier, especially the database schema, to improve solution flexibility.
- **Multi-query operations.** Clients typically perform interrelated queries, requiring them to remain connected to a single server while each successive query completes. This creates a connectivity dependence and eliminates the possibility of the client being load-balanced to multiple servers throughout its run time.

realtimepublishers.com®

Conversion Checklist

Here's a short checklist of things you'll need to change when converting an existing application to work in a scaled-out environment:



I'm not assuming, in this checklist, that you'll be using a multi-tier application, although I strongly recommend that you consider it.

- Remove all direct connections to servers and implement logic to connect to the proper server in the back-end. In a multi-tier application, all database connectivity will need to be replaced by use of remote middle-tier components.
- Examine the application for data which can be locally cached and updated on demand. Implement components that check for updated data (such as lookup data) and re-query it as necessary.
- Redesign applications to use asynchronous processing whenever possible and practical. This provides the middle- and back-end tiers with the most flexibility, and allows you to maximize performance.
- Remove schema-specific references. For example, references to specific column names or column ordinals should be removed, or rewritten so that the column names and ordinals are created by a middle tier, stored procedure, or other abstraction. The underlying database schema should be changeable without affecting client applications.
- Make operations as short and atomic as possible. If a client needs to execute a series of interrelated queries, try to make that a single operation on the client, and move more of the logic to the back-end or middle tier. By making every major client operation a "one and done" operation, you make it easier to re-load balance clients to a different middle-tier or SQL Server computer (if that's how your scale-out solution is architected).

Summary

This chapter focused on the last topics needed in a scale-out solution—the actual applications that will use your data. In general, a good practice is to apply a multi-tier approach to help isolate clients from the data, thus providing the maximum flexibility for the back-end data tier. Although the remainder of this book has focused primarily on SQL Server itself, you can't ignore the fact that SQL Server is only *part* of an application solution, and the design of the remainder of the solution plays an equally important role in the solution's overall scalability.

Throughout this book, the focus has been on scalability and flexibility. This guide has presented you with options for scaling out the back end, explained technologies that can help SQL Server perform better and more consistently, and introduced you to techniques that can help in both scale-up and scale-out scenarios. You've learned a bit about how high availability can be maintained in a scale-out solution, and about how critical subsystems—particularly storage—lend themselves to a better-performing solution. Although building a scale-out solution is never easy, hopefully, this guide has given you some pointers in the right direction, and as SQL Server continues to evolve as a product, we'll doubtless see new technologies and techniques dedicated to making scale-out easier and more efficient. In the meantime, the very best of luck with your scale-out efforts.

Content Central

Content Central is your complete source for IT learning. Whether you need the most current information for managing your Windows enterprise, implementing security measures on your network, learning about new development tools for Windows and Linux, or deploying new enterprise software solutions, Content Central offers the latest instruction on the topics that are most important to the IT professional. Browse our extensive collection of eBooks and video guides and start building your own personal IT library today!

Download Additional eBooks!

If you found this eBook to be informative, then please visit Content Central and download other eBooks on this topic. If you are not already a registered user of Content Central, please take a moment to register in order to gain free access to other great IT eBooks and video guides. Please visit: http://www.realtimepublishers.com/contentcentral/.