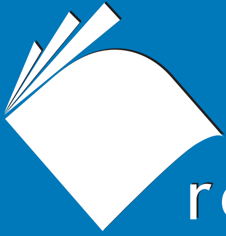


realtimepublishers.com™

The Definitive Guide™ To

Scaling Out SQL Server 2005

Don Jones

Chapter 3: Scaling Out SQL Server ... 49
Scale-Out Decisions ... 49
Real-Time Data ... 49
Cross-Database Changes ... 51
Scalable Database Designs ... 52
Redesigning Your Database ... 52
Scale-Out Techniques Overview ... 53
Distributed Databases and Replication ... 53
The Effects of Replication on Performance ... 54
Partitioned Databases ... 56
The Effects of Partitioned Databases on Performance ... 60
Distributed Partitioned Views ... 60
The Effects of Distributed Partitioned Views on Performance ... 62
Windows Clustering ... 63
Better Hardware Utilization ... 64
Four-Node Clusters ... 65
SQL Server Clusters ... 66
Effects of Clustering on Performance ... 68
Creating a Scale-Out Lab ... 69
Real-World Testing ... 69
Benchmarking ... 70
Summary ... 70

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 3: Scaling Out SQL Server

You’ve tweaked T-SQL until you’re blue in the face, upgraded your servers until they just can’t fit more memory or processors, and bought the fastest disk storage—and your database is still too slow. Whether you’re trying to support tens of thousands of users in a transactional application or quickly retrieve terabytes of data, scaling up has taken you as far as it can—it is time to scale *out.*

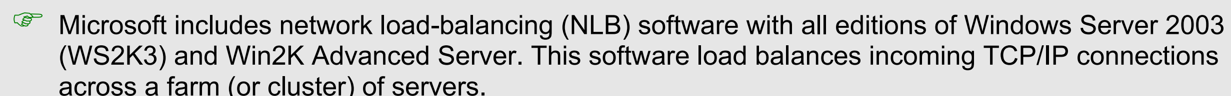
Before you start buying new servers and creating new databases, you must determine which scale-out technique is right for your environment and test the solution that you choose. You need to decide exactly how you will scale out and how to maintain a high level of fault tolerance in your new design. You will then need to perform some real-world testing to determine whether your database is ready to be scaled out and whether the scale-out decisions you’ve made result in an actual performance gain. In this chapter, we’ll explore how to select the best technique for your situation and the best methods for testing your selection to ensure it will handle the workload.

Scale-Out Decisions

Exactly how you use your data and how your data is structured will greatly impact which scale-out options are available to you. Most large database applications not only deal with a lot of data and many users but also with widely distributed users (for example, several offices that each accommodate thousands of users). One solution is to simply put a dedicated database server in each location rather than one central server that handles all user requests. However, distributing servers results in multiple copies of the data and the associated problem of keeping each copy updated. These and other specifics of your environment—such as the need for real-time data—will direct you to a particular scale-out technique.

Real-Time Data

The need for real-time data, frankly, complicates the scale-out process. If we could all just live with slightly out-of-date data, scaling out would be simple. For example, consider how a Web site is scaled out. When the number of users accessing the site becomes too much for one server to handle, you simply add Web servers. Each server maintains the same content and users are load balanced between the servers. Users aren’t aware that they are using multiple Web servers because all the servers contain the same content. In effect, the entire group of servers—the *Web farm*—appears to users as one gigantic server. A certain amount of administrative overhead occurs when the Web site’s content needs to be updated because the new content must be quickly deployed to all the servers so that they are in sync with one another, but there are many tools and utilities that simplify this process.

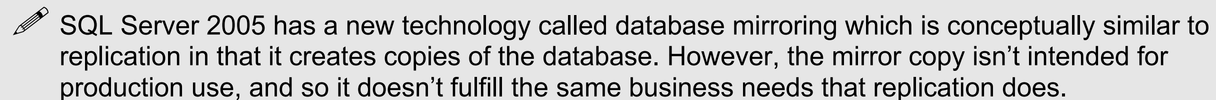


Microsoft includes network load-balancing (NLB) software with all editions of Windows Server 2003 (WS2K3) and Win2K Advanced Server. This software load balances incoming TCP/IP connections across a farm (or cluster) of servers.

realtimepublishers.com®

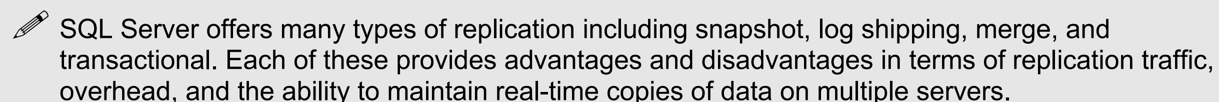
Why not simply scale out SQL Server in the same way? The real-time data requirement makes this option unfeasible. Suppose you copied your database to three new servers, giving you a total of four SQL Server computers that each maintains a copy of your database. As long as you ensure that users are accessing SQL Server only via TCP/IP and you implement Windows NLB to load-balance connections across the SQL Server computers, everything would work reasonably well—as long as your users only query records (each server would have an identical copy of the records). The minute someone needed to change a record, though, the situation would change. Now, one server would have a different copy of the database than the other three. Users would get different query results depending on which of the four servers they queried. As users continued to make changes, the four database copies would get more and more out of sync with one another, until you would have four completely different databases.

SQL Server includes technology to help with the situation: *replication.* The idea behind replication is that SQL Server can accept changes on one server, then copy those changes out to one or more other servers. Servers can both send and receive replication traffic, allowing multiple servers to accept data updates and distribute those updates to their partner servers.



SQL Server 2005 has a new technology called database mirroring which is conceptually similar to replication in that it creates copies of the database. However, the mirror copy isn't intended for production use, and so it doesn't fulfill the same business needs that replication does.

However, replication doesn't occur in real-time. Typically, a server will save up a batch of changes, then replicate those changes in order to maintain a high level of efficiency. Thus, each server will always be slightly out of sync with the other servers, creating inconsistent query results. The more changes that are made, the more out-of-sync the servers will become. In some environments, this lag time might not matter, but in corporate transactional applications, everyone must see the same results every time, and even a "little bit" out of sync is too much.



SQL Server offers many types of replication including snapshot, log shipping, merge, and transactional. Each of these provides advantages and disadvantages in terms of replication traffic, overhead, and the ability to maintain real-time copies of data on multiple servers.

What if you could make replication take place immediately—the second someone made a change, it would replicate to the other servers? Unfortunately, this real-time replication would defeat the purpose of scaling out. Suppose you have one server that supports ten thousand users who each make one change every 5 minutes—that is about 120,000 changes per hour. Suppose you copied the database across a four-server farm and evenly load balanced user connections across the four servers. Now, each server will need to process only one-quarter of the traffic (about 30,000 changes per hour). However, if every server immediately replicates every change, each of the four servers will still need to process 120,000 changes per hour—their own 30,000 plus the 30,000 apiece from the other three servers. In effect, you've bought three new servers to exactly duplicate the original problem. Ultimately, that's the problem with any replication technology: There's no way to keep multiple copies of a frequently-updated database up-to-date without putting an undesirable load on every copy.

As this scenario illustrates, the need for real-time data across the application is too great to allow a scale-out scenario to take significant advantage of replication. Thus, one of the first scale-out project considerations is to determine how up-to-date your data needs to be at any given moment.

realtimepublishers.com®

Later in the chapter, I'll explore a scale-out environment that employs replication. If you don't have a need for real-time data (and not all applications do), replication does offer some interesting possibilities for scale-out.

Cross-Database Changes

Another scale-out project consideration is whether you can split your database into functionally separate sections. For example, in a customer orders application, you might have several tables related to customer records, vendors, and orders that customers have placed. Although interrelated, these sections can standalone—changes to a customer record don't require changes to order or vendor records. This type of database—one which can be split along functional lines—is the best candidate for a scale-out technique known as *vertical partitioning*.

I'll discuss vertical partitioning shortly.

However, if your database tables are heavily cross-linked—updates to one set of tables frequently results in significant updates to other sets of tables—splitting the database across multiple servers will still require a significant number of *cross-database changes,* which might limit the effectiveness of a scale-out technique.

Vertical partitioning breaks the database into discreet sections that can then be placed on dedicated servers (technically, both a large database that is partitioned by column and several tables spread onto different servers qualify as vertical partitioning—just a different levels). Ideally, vertical partitioning will help distribute the load of the overall database application across these servers without requiring replication. However, if a large number of cross-database changes are regularly required by your application, splitting the database might not actually help. Each server participating in the scheme will still be required to process a large number of updates, which may mean that each server can only support the same (or close to the same) number of users as your original, single-server architecture.

Analyze your database to determine whether it can be logically split into different groups of functionally related tables. There will nearly always be some relationship between these sets. For example, a set of tables for customer orders will likely have a foreign key relationship back to the customer records, allowing you to associate orders with specific customers. However, adding an order wouldn't necessarily require changes to the customer tables, making the two sets of tables functionally distinct.

Scalable Database Designs

Vertical partitioning is one technique to make your database design more scalable, but it certainly isn't the most popular method. Many database tables are so heavily interdependent that vertical partitioning just isn't practical—*horizontal partitioning* is much easier. Thus, you'll need to decide how easily your database's rows can be broken up. For example, examine the largest tables in your database and consider how you might logically divide the rows into different sets. Perhaps some of your orders were placed through an Eastern call center while others come from the West, or perhaps one set of customers is managed by a different service center than others. You can also look for arbitrary divisions that aren't related to the functionality of the databases: ranges of customers by last name (A through M and N through Z, for example) or the odd- and even-numbered customer IDs as separate sets. Your database will very likely need to be split up in some fashion in order to be scaled out—you simply need to decide how that split will occur.

Horizontal vs. Vertical Partitioning

Should you split your database horizontally, by data rows, or vertically, by breaking of functionally related groups of tables? The answer largely depends on how your database is built and updated. Tables that are extremely large and comprise a majority of your database traffic are the best candidates for horizontal partitioning. If your database can be logically split into groups of functionally related tables, vertical partitioning might be a valid technique.

Part of the answer also depends upon how your client applications are written. Vertically partitioning a database can be expensive in terms of client application changes because you might need to heavily modify those applications to make them aware of the new database architecture. Horizontal partitioning can be simpler because you can more easily use SQL Server views to present the appearance of a single database server to your client applications.

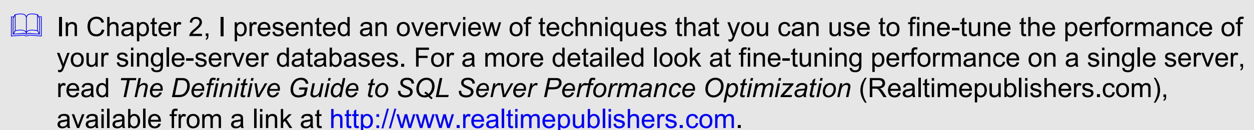
You can, of course, adopt *both* solutions, splitting your tables both horizontally and vertically to create the scale-out scenario that best fits your needs. For example, you could use both solutions for a large customer order database that you want to horizontally partition but you want to isolate your product catalog on a separate server.

Redesigning Your Database

Very few databases are successfully scaled out without some type of database redesign and client application (or middle-tier business object) modification. If it isn't obvious how to best split your current single-server database across multiple servers, plan for redesign work. Although redesigning your database can be a difficult task in a production environment, it is time well spent. Consider the following to be a "law" of scaling out:

> *Always finalize your scale-out database design in a single-server environment before attempting to move to a fully scaled-out, multiple-server environment.*

In other words, if you need to redesign tables, relationships, indexes, queries, stored procedures, views, client applications, or middle-tier objects, do so while everything is on a single database server. Finalize the architecture before you begin moving out to multiple servers. Moving a database to multiple servers presents plenty of challenges without simultaneously working out the kinks of an all-new (or even slightly modified) database design.



In Chapter 2, I presented an overview of techniques that you can use to fine-tune the performance of your single-server databases. For a more detailed look at fine-tuning performance on a single server, read *The Definitive Guide to SQL Server Performance Optimization* (Realtimepublishers.com), available from a link at http://www.realtimepublishers.com.

Scale-Out Techniques Overview

Much of the rest of this chapter is devoted to an overview of scale-out techniques so that you can consider your business needs and existing database design and get a feel for which techniques are best-suited to your environment. In Chapters 4, 5, and 6, we'll explore these scale-out techniques in more detail, including step-by-step instructions for implementing each technique on SQL Server 2000.

Distributed Databases and Replication

As I described earlier, one way—in fact, probably the easiest way—to scale out a database application is to simply add more servers and give them each a copy of the database. There are tricks and tradeoffs involved, but for some applications, this scale-out technique can provide a working environment with a minimum of re-architecting.

Figure 3.1 illustrates how a distributed database, combined with replication, can be used as a simple scale-out technique.

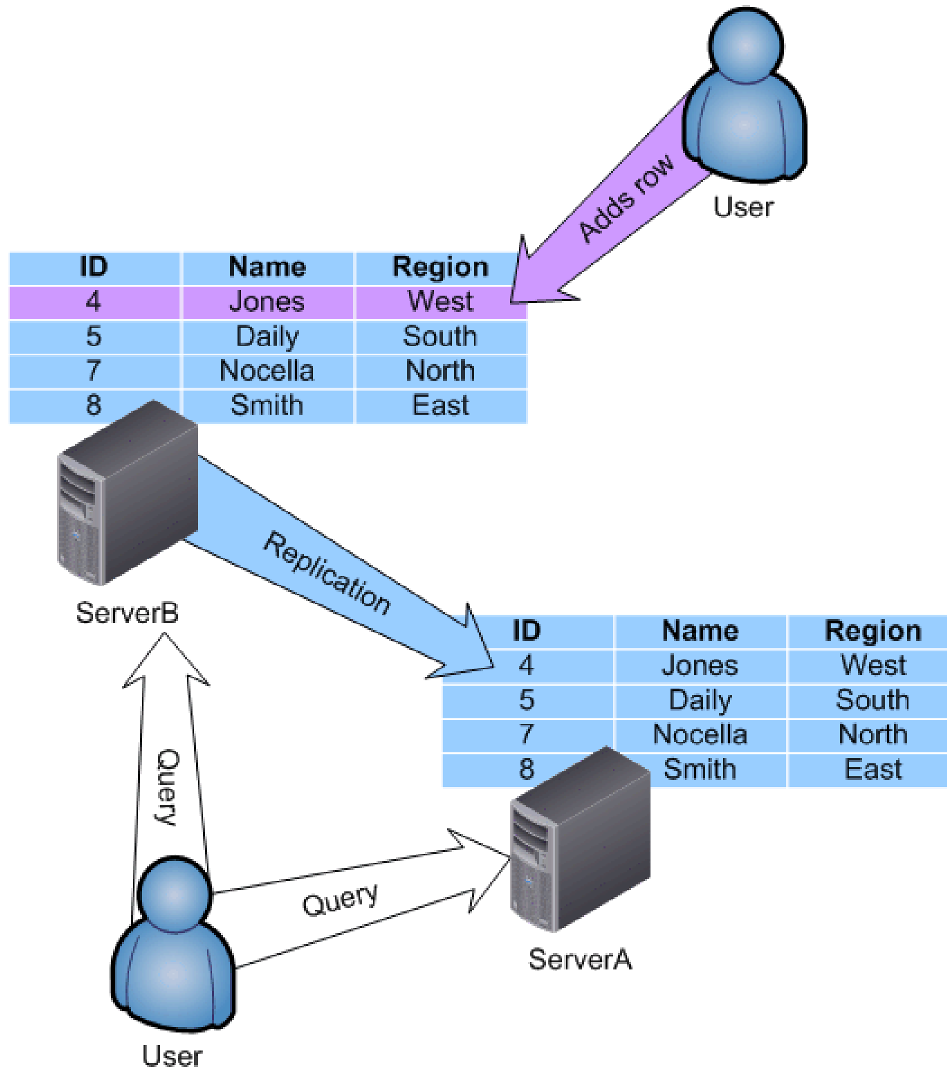
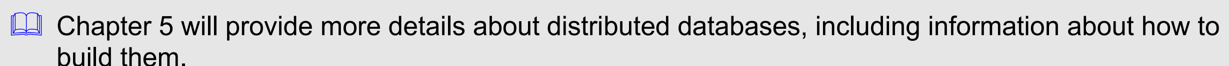


Figure 3.1: Scale-out through distributed databases.

In this example, each server contains a complete, identical copy of the database schema and data. When a user adds a row to one server, replication updates the copy of the data on the other server. Users can then query either server and get essentially the same results.

The time it takes for ServerB to send its update to ServerA is referred to as *latency.* The types of replication that SQL Server supports offer tradeoffs among traffic, overhead, and latency:

- Log shipping isn't truly a form of replication but can be used to similar effect. This technique copies the transaction log from one server to another server, and the log is then applied to the second server. This technique offers very high latency but very low overhead. It's also only available for an entire database; you can't replicate just one table by using log shipping.
- Similar to log shipping, snapshot replication essentially entails sending a copy of the database from one server to another. This replication type is a high-overhead operation, and locks the source database while the snapshot is being compiled, so snapshot replication is not a method you want to use frequently on a production database. Most other forms of replication start with a snapshot to provide initial synchronization between database copies.
- Transactional replication copies only transaction log entries from server to server. Assuming two copies of a database start out the same, applying the same transactions to each will result in identical final copies. Because the transaction data is often quite small, this technique offers fairly low overhead. However, to achieve low latency, you must constantly replicate the transactions, which can create a higher amount of cumulative overhead. Transactional replication also essentially ignores conflicts when the same data is changed in two sources—the last change is kept regardless of whether that change comes from a direct user connection or from an older, replicated transaction.
- Merge replication works similarly to transactional replication but is specifically designed to accommodate conflicts when data is changed in multiple sources. You must specify general rules for handling conflicts or write a custom merge agent that will handle conflicts according to your business rules.
- Mirroring, a new option introduced in SQL Server 2005, is primarily designed for high availability. Unlike replication, which allows you to replicate individual tables from a database, mirroring is configured for an entire database. Mirroring isn't appropriate to scale-out solutions because the mirror copy of the database isn't intended for production use; its purpose is primarily as a "hot spare" in case the mirror source fails.



Chapter 5 will provide more details about distributed databases, including information about how to build them.

The Effects of Replication on Performance

Replication can be a good way to improve the performance for read-only applications. For example, you might have a central server designed to take updates and multiple additional servers that maintain read-only copies of the data. Transactional or snapshot replication—depending on the latency you're willing to tolerate—can be used to push updates to the read-only copies of the data. Web sites with read-only data are a good example of this technique (see Figure 3.2).

realtimepublishers.com®

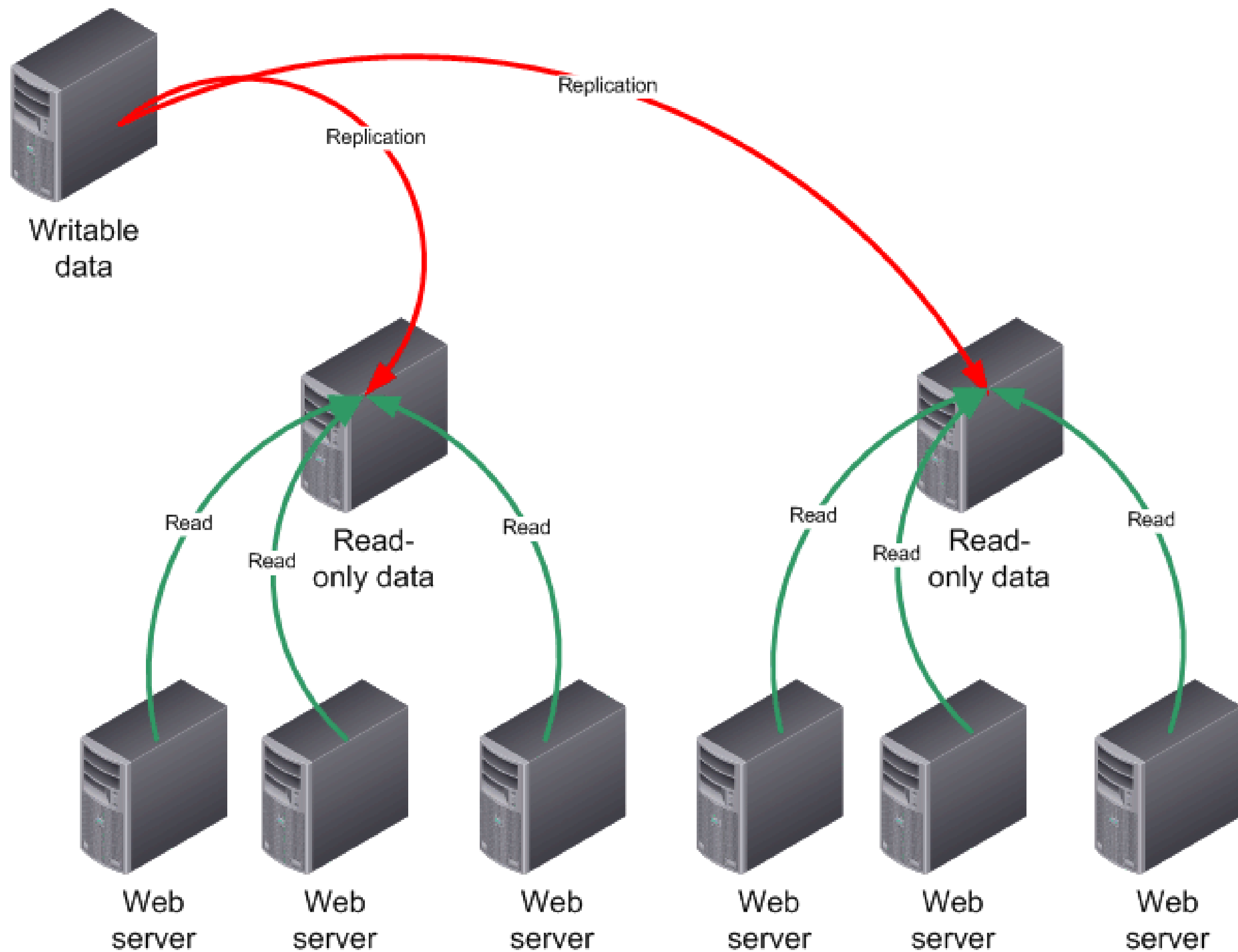


Figure 3.2: Using replication to scale out a Web back end.

However, for applications for which each copy of the data needs to support many write operations, replication becomes less suitable. As Figure 3.3 illustrates, each change made to one server results in a replicated transaction to every other server if you need to maintain a low degree of latency. This fully enmeshed replication topology can quickly generate a lot of overhead in high-volume transactional applications, reducing the benefit of the scale-out project.

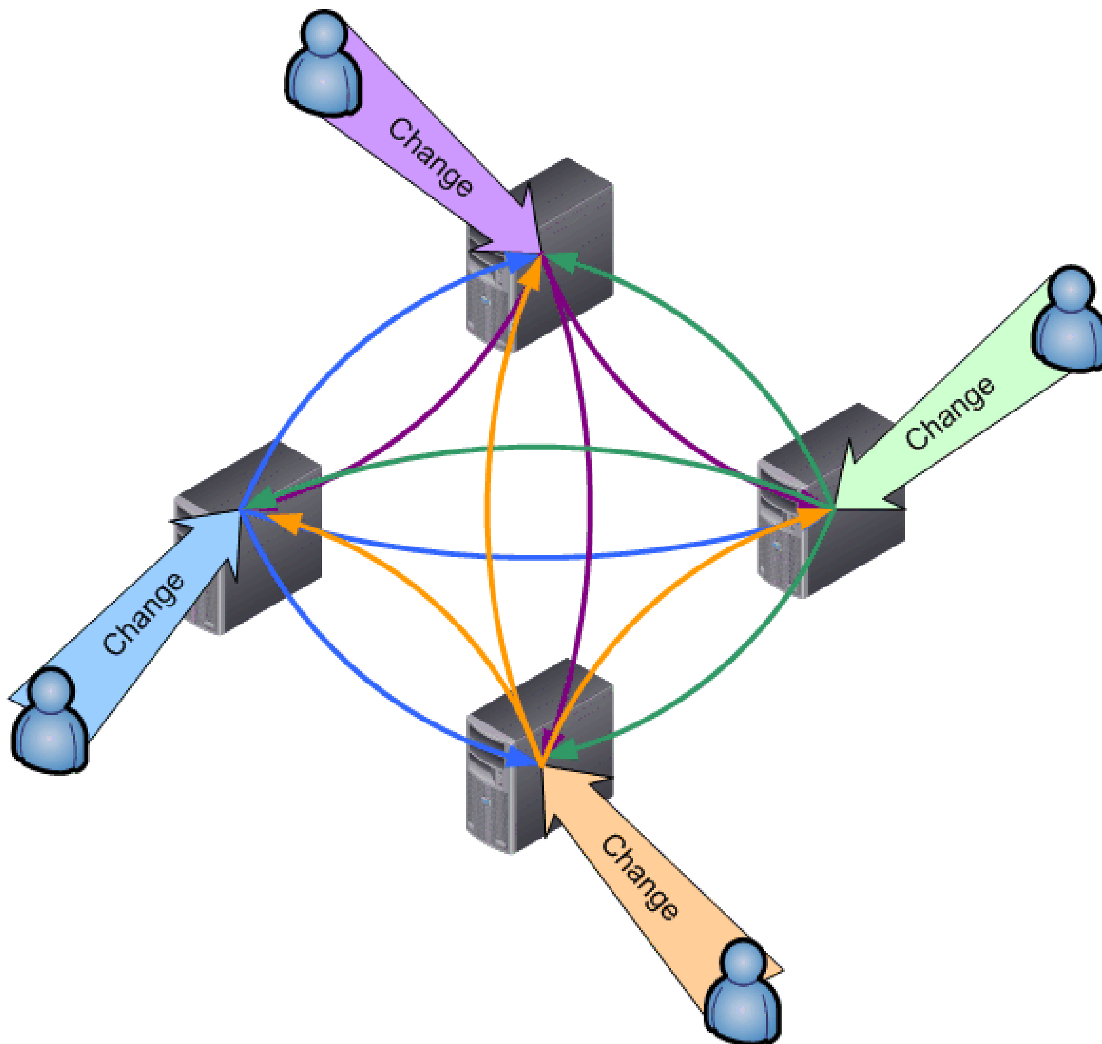


Figure 3.3: Replication traffic can become high in distributed, write-intensive applications.

To work around this drawback, a less-drastic replication topology could be used. For example, you might create a round-robin topology in which each server simply replicates with its right-hand neighbor. Although this setup would decrease overhead, it would increase latency, as changes made to one server would need to replicate three times before arriving at the original server's left-hand neighbor. When you need to scale out a write-intensive application such as this, a distributed, partitioned database—one that doesn't use replication—is often a better solution.

Partitioned Databases

Partitioning is simply the process of logically dividing a database into multiple pieces, then placing each piece on a separate server. Partitioning can be done along horizontal or vertical lines, and techniques such as replication and distributed partitioned views can be employed to help reduce the complexity of the distributed database. Figure 3.4 shows a basic, horizontally partitioned database.

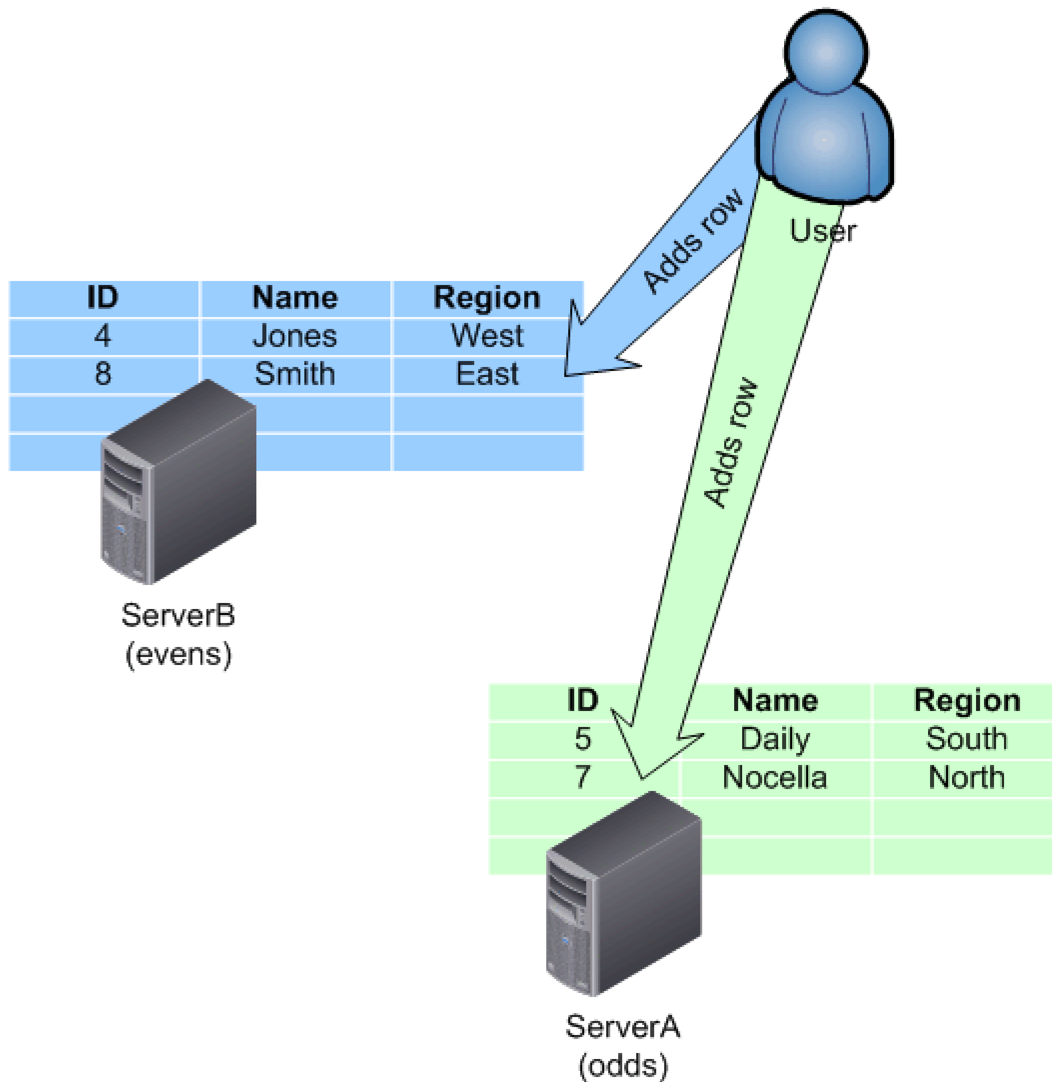


Figure 3.4: Horizontally partitioned database.

In this example, the odd- and even-numbered customer IDs are handled by different servers. The client application (or a middle tier) includes the logic necessary to determine the location of the data and where changes should be made. This particular example is especially complex because each server only contains its own data (either odd or even customer IDs); the client application must not only determine where to make changes but also where to query data. Figure 3.5 shows how replication can be used to help alleviate the complexity.

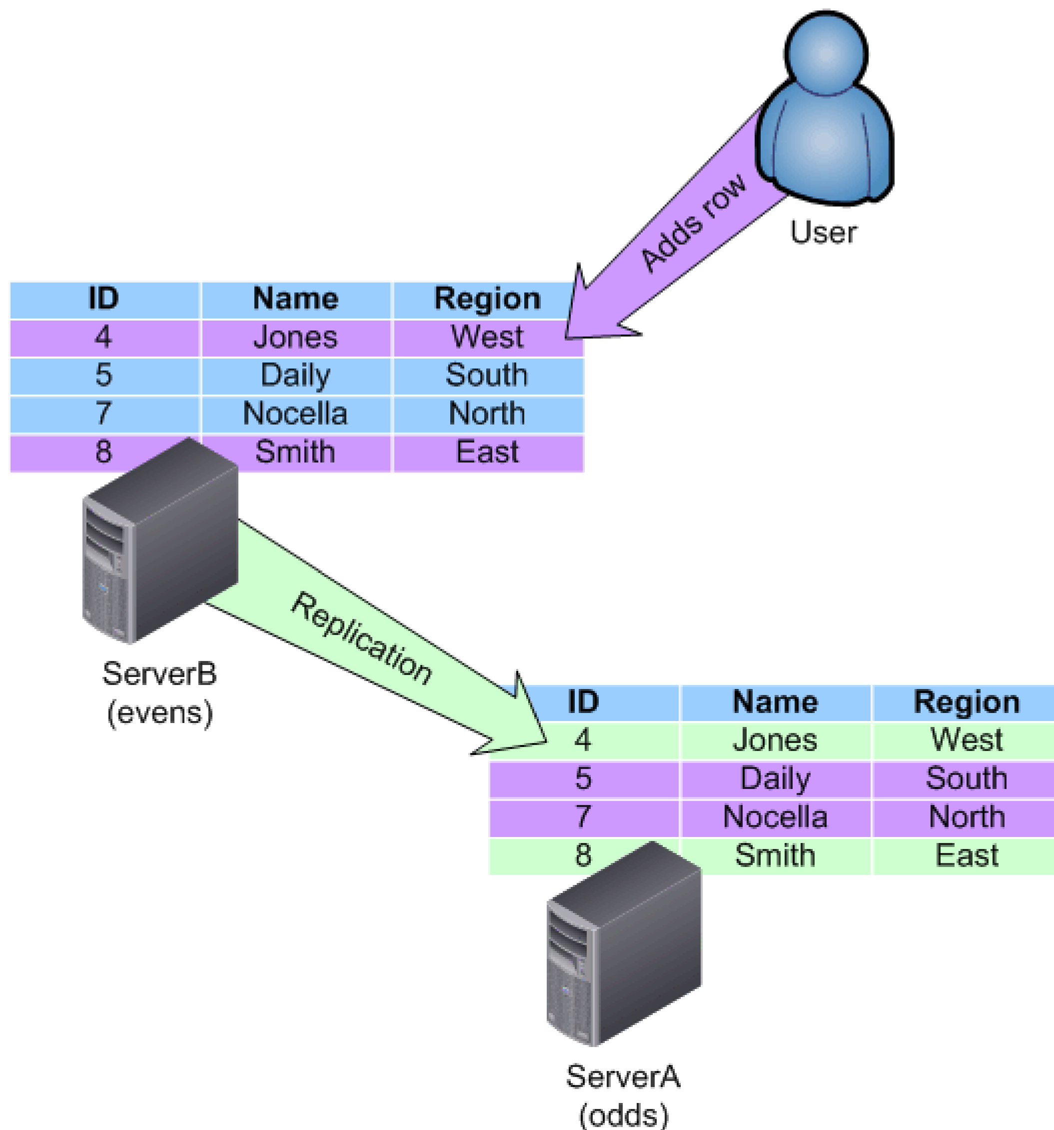


Figure 3.5: Replicating data across databases.

In this example, when a client makes a change, the client must make the change to the appropriate server. However, all data is replicated to both servers, so read operations can be made from either server. Prior to SQL Server 2000, this configuration was perhaps the best technique for scaling out and using horizontally partitioned databases. SQL Server 200x's (meaning either SQL Server 2000 or SQL Server 2005) distributed partitioned views, however, make horizontally partitioned databases much more practical. I'll discuss distributed partitioned views in the next section.

Vertically partitioned databases are also a valid scale-out technique. As Figure 3.6 shows, the database is split into functionally related tables and each group of tables has been moved to an independent server.

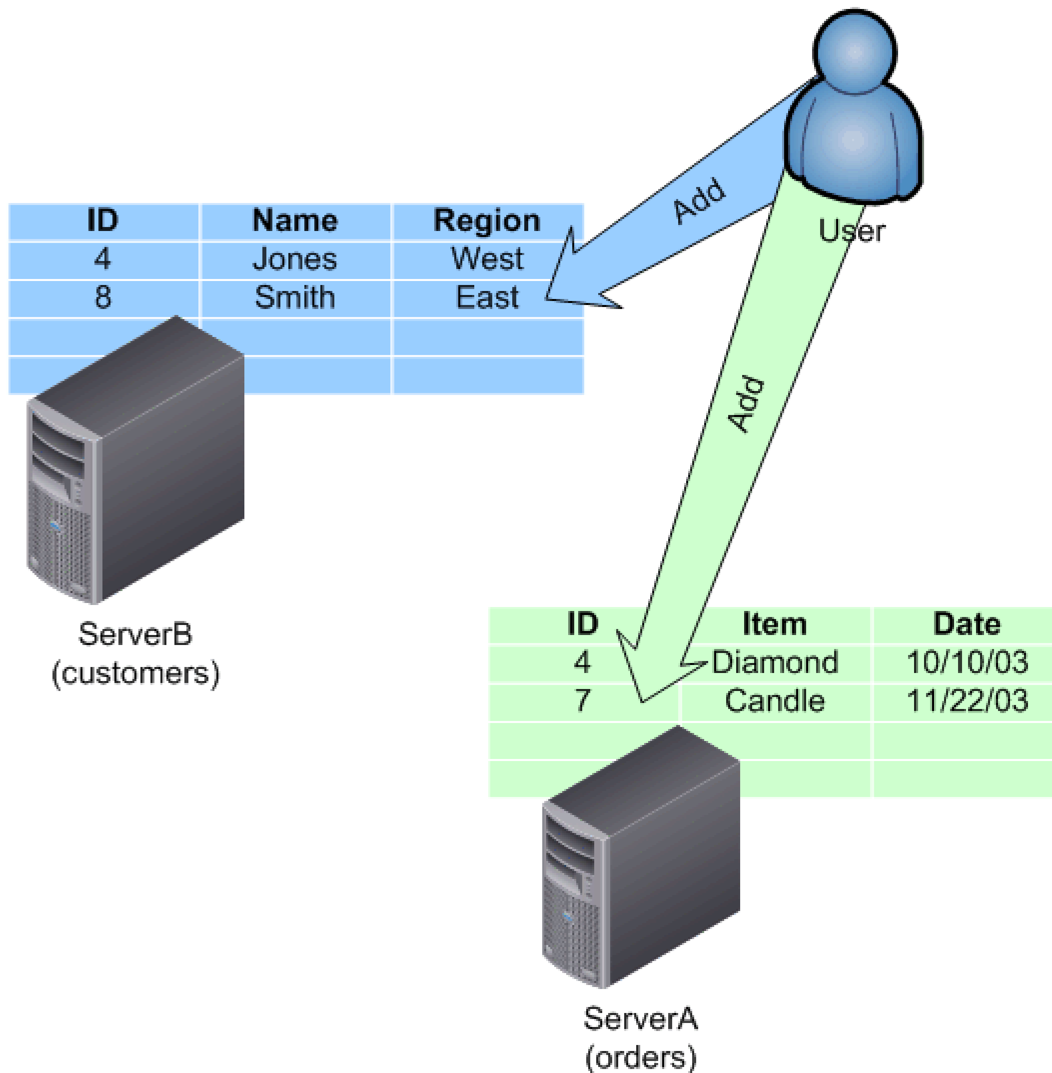
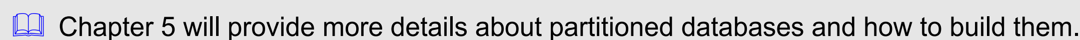


Figure 3.6: Vertically partitioned database.

In this example, each server contains a portion of the database schema. Client applications (or middle-tier objects) contain the necessary logic to query from and make changes to the appropriate server. Ideally, the partitioning is done across some kind of logical functional line so that—for example—a customer service application will primarily deal with one server and an order-management application will deal with another. SQL Server views can be employed to help recombine the disparate database sections into a single logical view, making it easier for client applications to access the data transparently.


Chapter 5 will provide more details about partitioned databases and how to build them.

The Effects of Partitioned Databases on Performance

Partitioned databases can have a significant positive effect on performance. By distributing the rows or columns of a database across multiple servers, you can enlist several servers in your overall application. What makes partitioning impractical for many companies is the resulting complexity placed on client applications and middle-tier objects, which are now required to understand what data lives where, where to read data from, and where changes can be made. So while in theory partitioning is the ultimate performance boost—allowing servers to essentially specialize either in a certain type of data or in a certain set of tables, or certain rows of data within a table—the feasibility of converting a major single-server database application to one that uses multiple servers with a partitioned database is low. However, there are techniques to improve the feasibility and practicality of partitioning, including distributed partitioned views.

Distributed Partitioned Views

SQL Server 200x offers distributed partitioned views to reduce the complexity of working with highly partitioned databases—primarily horizontally partitioned databases. For example, refer back to Figure 3.4, which shows a single database horizontally partitioned across multiple servers. The primary difficulty of this scenario is in writing client applications that understand how the data has been distributed. Distributed partitioned views eliminate that difficulty from a development viewpoint, but at a significant performance cost. Figure 3.7 shows how a distributed partitioned view works.

Chapter 4 covers distributed partitioned views, including details about how to create them.

realtimepublishers.com®

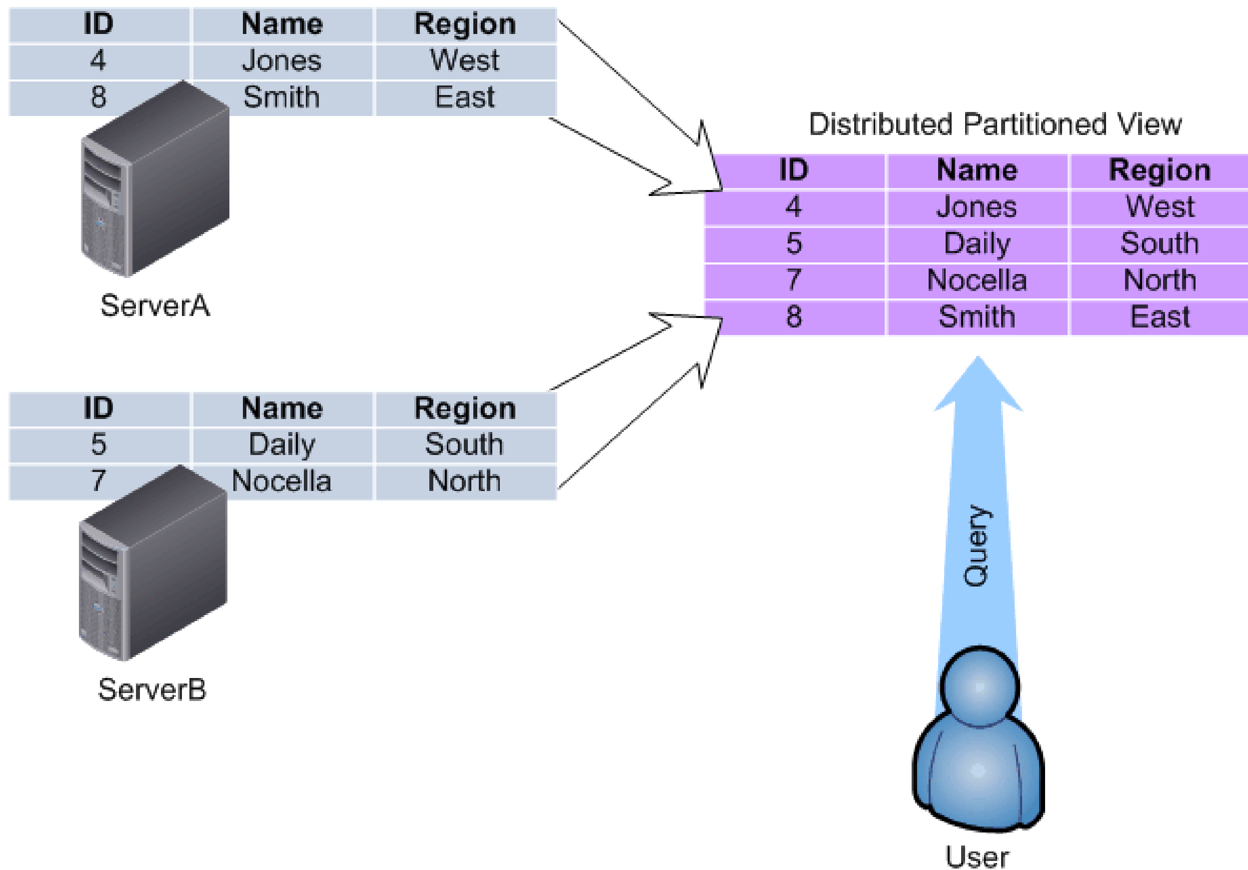
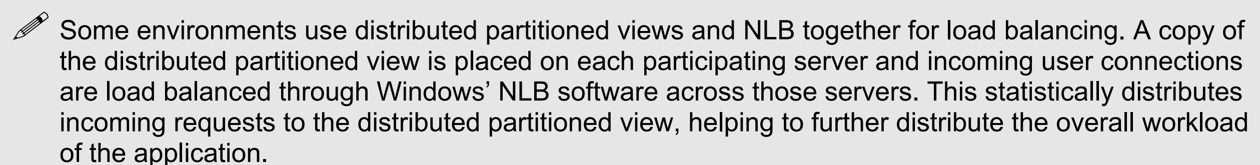


Figure 3.7: A distributed partitioned view.

In this scenario, client applications are not aware of the underlying distributed partitioned database. Instead, the applications query a distributed partitioned view, which is simply a kind of virtual table. On the back end, the distributed partitioned view queries the necessary data from the servers hosting the database, constructing a virtual table. The benefit of the distributed partitioned view is that you can repartition the physical data as often as necessary without changing your client applications. The distributed partitioned view makes the underlying servers appear as one server rather than several individual ones.



Some environments use distributed partitioned views and NLB together for load balancing. A copy of the distributed partitioned view is placed on each participating server and incoming user connections are load balanced through Windows' NLB software across those servers. This statistically distributes incoming requests to the distributed partitioned view, helping to further distribute the overall workload of the application.

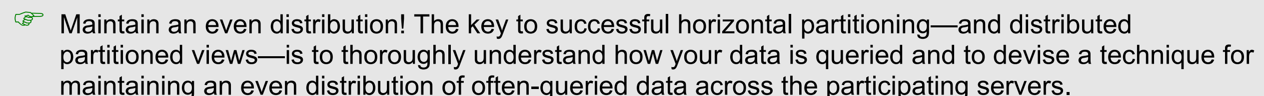
However, because of the difficult-to-predict performance impact of horizontal partitioning and distributed partitioned views (which I'll discuss next), it is not easy to determine whether the NLB component adds a significant performance advantage.

The Effects of Distributed Partitioned Views on Performance

Distributed partitioned views don’t increase performance, but they make horizontally partitioned databases—which do increase performance—more feasible. Large queries can be distributed between multiple servers, each contributing the necessary rows to complete the query and produce the final distributed partitioned view. SQL Server 200x’s distributed partitioned views are updatable, so they can be treated as ordinary tables for most purposes, and any changes made to the distributed partitioned view are transparently pushed back to the proper back-end servers.

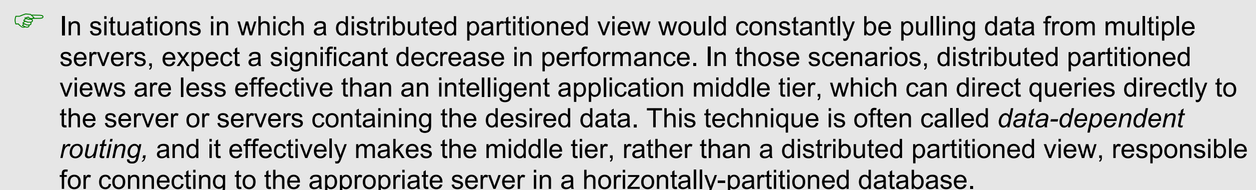
However, the ease of use provided by distributed partitioned views comes at a cost with potential performance drawbacks. First, the performance benefits of a horizontally partitioned database depend on an even distribution of rows across the various copies of the database. If, for example, the most-queried rows are all on one server, that server’s performance will become a bottleneck in the overall application’s performance. When designing partitions, you need to design an even initial split of rows based on usage, and you might need to periodically repartition to maintain an even balance of the workload across the servers.

Distributed partitioned views incur a performance hit. The servers queried by a distributed partitioned view are required to execute their query and then maintain the resulting rows in memory until all other servers queried by the distributed partitioned view complete. The rows are then transferred to the server physically containing the distributed partitioned view, and the distributed partitioned view is presented to the user or client application. A problem arises if your servers have significantly different hardware or some servers must respond with a much larger set of rows than others; in such cases, some of the servers queried by the distributed partitioned view will be required to retain rows in memory for a significant period of time while the other distributed partitioned view participants complete their portion of the query (requests made to the different servers are serial). Retaining rows in memory is one of the most resource-intensive operations that SQL Server can perform (one reason why server-side cursors have such a negative impact on performance).



Maintain an even distribution! The key to successful horizontal partitioning—and distributed partitioned views—is to thoroughly understand how your data is queried and to devise a technique for maintaining an even distribution of often-queried data across the participating servers.

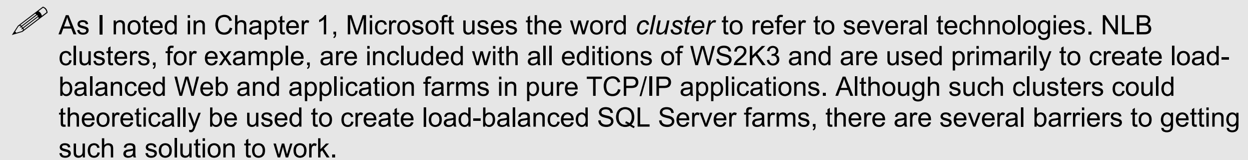
When are distributed partitioned views a good choice for scaling out? When your data can be horizontally partitioned in such a way that most users’ queries will be directed to a particular server, and that server will have most of the queried data. For example, if you partition your table so that East coast and West coast data is stored on two servers—knowing that West coast users almost always query West coast data only and that East coast users almost always query East coast data only—then distributed partitioned views provide a good way to scale out the database. In most cases, the view will pull data from the local server, while still providing a slower-performance means of querying the other server.



In situations in which a distributed partitioned view would constantly be pulling data from multiple servers, expect a significant decrease in performance. In those scenarios, distributed partitioned views are less effective than an intelligent application middle tier, which can direct queries directly to the server or servers containing the desired data. This technique is often called *data-dependent routing,* and it effectively makes the middle tier, rather than a distributed partitioned view, responsible for connecting to the appropriate server in a horizontally-partitioned database.

Windows Clustering

I introduced Windows Clustering in Chapter 1, and Chapter 6 is devoted entirely to the topic of clustering. Clustering is becoming an increasingly popular option for scale-out scenarios because it allows you to employ many servers while maintaining a high level of redundancy and fault tolerance in your database server infrastructure. WS2K3 introduces the ability to support 4-way and 8-way clusters in the standard and enterprise editions of the product, making clustering more accessible to a larger number of companies (8-way clusters are available only on SQL Server Enterprise 64-bit Edition).



As I noted in Chapter 1, Microsoft uses the word *cluster* to refer to several technologies. NLB clusters, for example, are included with all editions of WS2K3 and are used primarily to create load-balanced Web and application farms in pure TCP/IP applications. Although such clusters could theoretically be used to create load-balanced SQL Server farms, there are several barriers to getting such a solution to work.

To complicate matters further, Windows Server 2003, x64 Edition, supports a new clustering technology called *compute cluster* which is completely different from Windows Cluster Server-style clustering. I'll cover that in Chapter 7.

In this book, I'll use the term *cluster* to refer exclusively to what Microsoft calls a Windows Cluster Server. This type of cluster physically links multiple servers and allows them to fill in for one another in the event of a total hardware failure.

The idea behind clustering is to enlist several servers as a group to behave as a single server. With Windows Cluster Server, the purpose of this union isn't to provide load balancing or better performance or to scale out; it is to provide fault tolerance. If one server fails, the cluster continues to operate and provide services to users. Figure 3.8 shows a basic 2-node cluster.

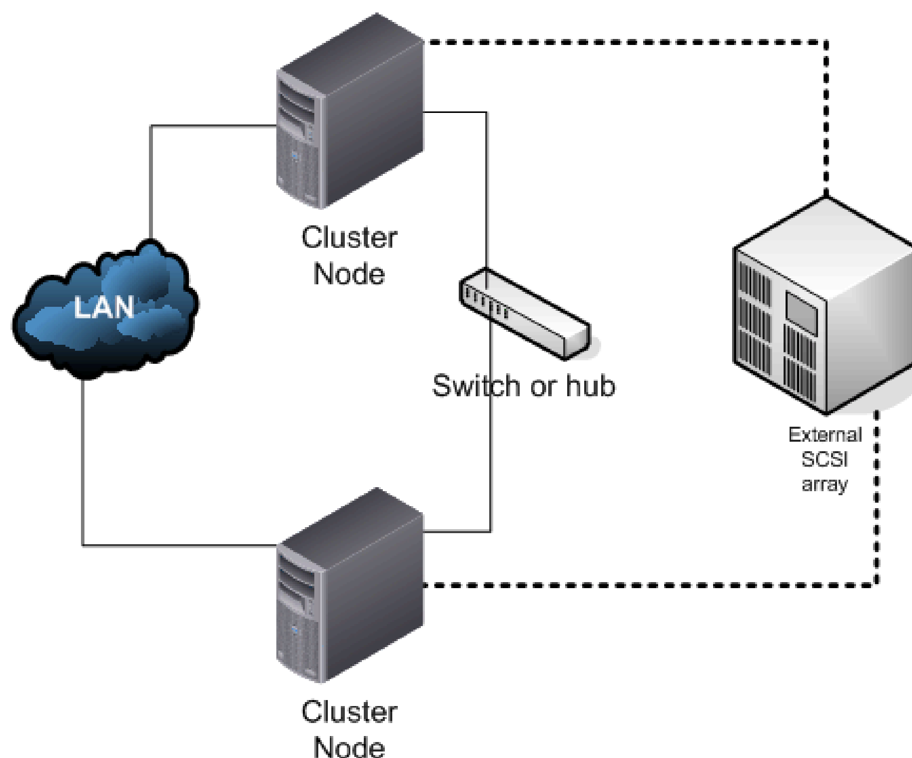
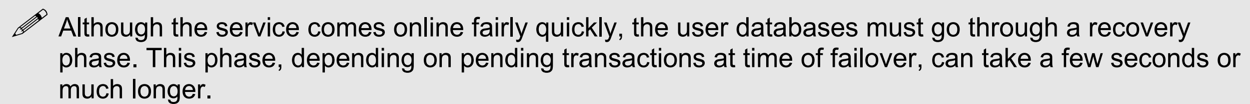


Figure 3.8: A basic 2-node cluster.

This diagram shows the dedicated LAN connection used to talk with the corporate network, and the separate connection used between the two cluster nodes (while not strictly required, this separate connection is considered a best practice, as I'll explain in Chapter 6). Also shown is a shared external SCSI disk array. Note that each node also contains its own internal storage, which is used to house both the Windows OS and any clustered applications. The external array—frequently referred to as *shared storage* even though both nodes do not access it simultaneously—stores only the data used by the clustered applications (such as SQL Server databases) and a small cluster configuration file.

Essentially, one node is active at all times and the other is passive. The active node sends a *heartbeat signal* across the cluster's private network connection; this signal informs the passive node that the active node is active. The active node also maintains exclusive access to the external SCSI array. If the active node fails, the passive node becomes active, seizing control of the SCSI array. Users rarely notice a cluster failover, which can occur in as little as 30 seconds.


Although the service comes online fairly quickly, the user databases must go through a recovery phase. This phase, depending on pending transactions at time of failover, can take a few seconds or much longer.

Better Hardware Utilization

An inactive node isn't the best utilization of expensive hardware. For this reason, many companies build *active-active* clusters (see Figure 3.9).

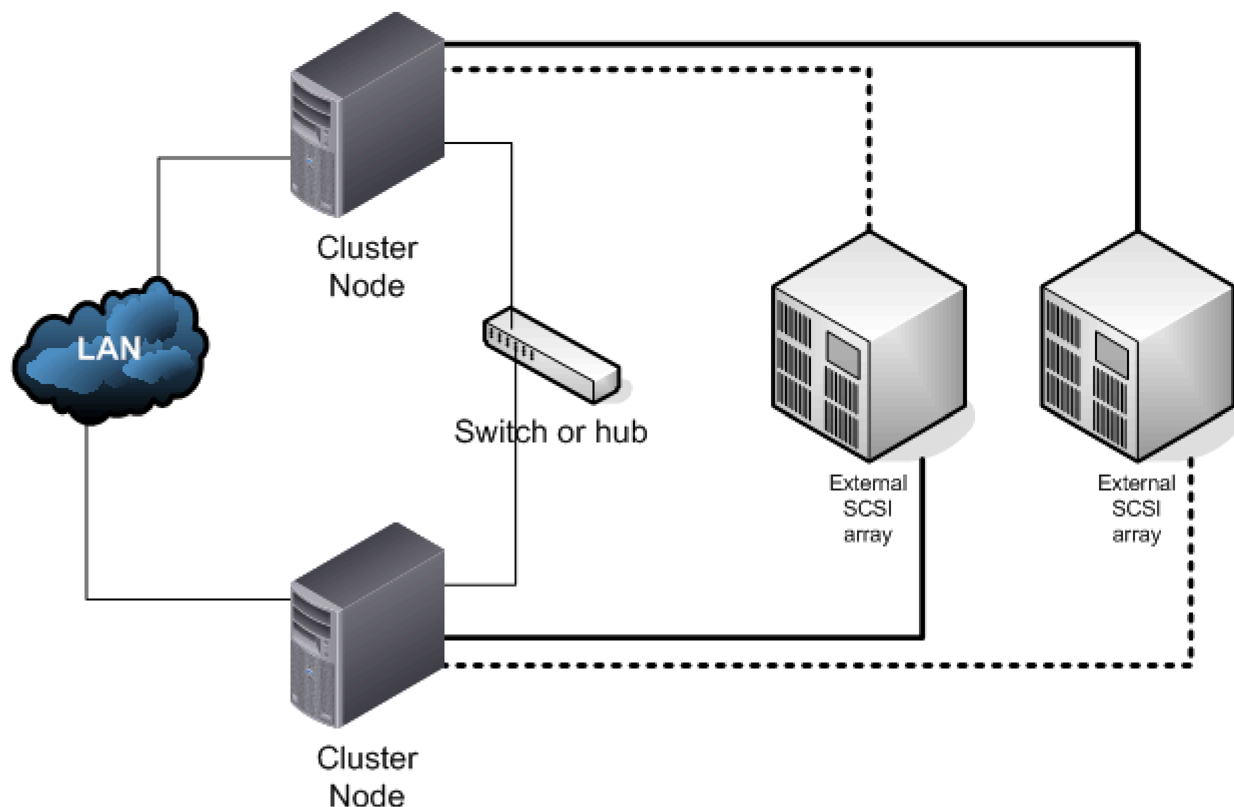
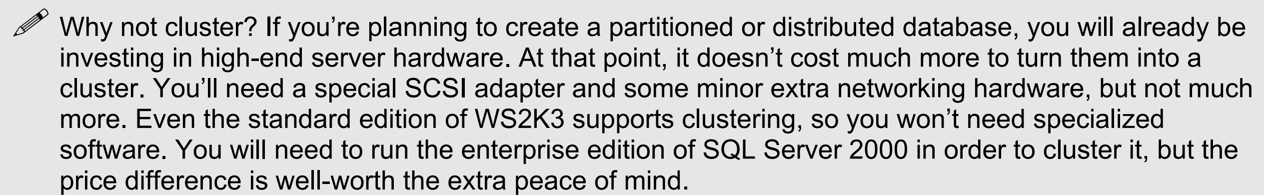


Figure 3.9: An active-active cluster.

In an active-active cluster, a separate external SCSI array is required for each active node. Each node "owns" one external array and maintains a passive link to the other node. In the event that one node fails, the other node becomes active for both, owning both arrays and basically functioning as two complete servers.

Active-active is one of the most common types of SQL Server clusters because both cluster nodes—typically higher-end, pricier hardware—are serving a useful purpose. In the event of a failure, the databases from both servers remain accessible through the surviving node.


Why not cluster? If you're planning to create a partitioned or distributed database, you will already be investing in high-end server hardware. At that point, it doesn't cost much more to turn them into a cluster. You'll need a special SCSI adapter and some minor extra networking hardware, but not much more. Even the standard edition of WS2K3 supports clustering, so you won't need specialized software. You will need to run the enterprise edition of SQL Server 2000 in order to cluster it, but the price difference is well-worth the extra peace of mind.

Four-Node Clusters

If you're buying three or four SQL Server computers, consider clustering all of them. Windows clustering supports as many as 8-way clusters (on the 64-bit edition), meaning you can build clusters with three, four, or more nodes, all the way up to eight (you will need to use the enterprise or datacenter editions for larger clusters). As Figure 3.10 shows, 4-node clusters use the same basic technique as an active-active 2-node cluster.

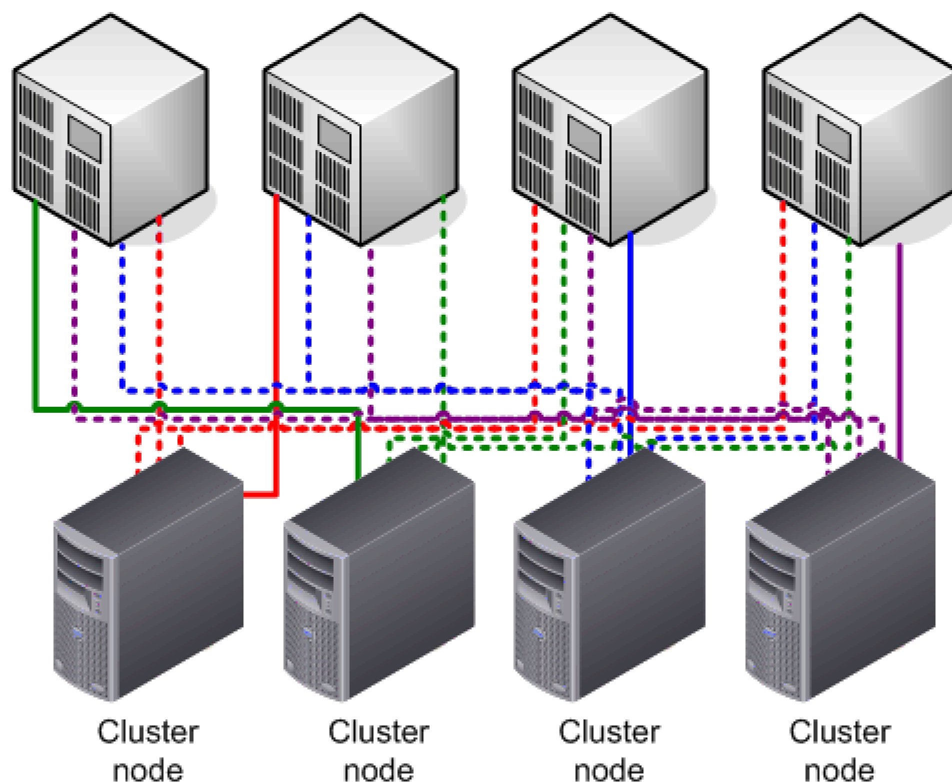


Figure 3.10: A 4-way cluster configuration.

realtimepublishers.com®

I've left the network connections out of this figure to help clarify what is already a complex situation: each node must maintain a physical connection to each external drive array, although under normal circumstances, each node will only have an active connection to one array.

It is very uncommon for clusters of more than two nodes to use copper SCSI connections to their drive arrays, mainly because of the complicated wiring that would be involved. As Figure 3.11 shows, a *storage area network (SAN)* makes the situation much more manageable.

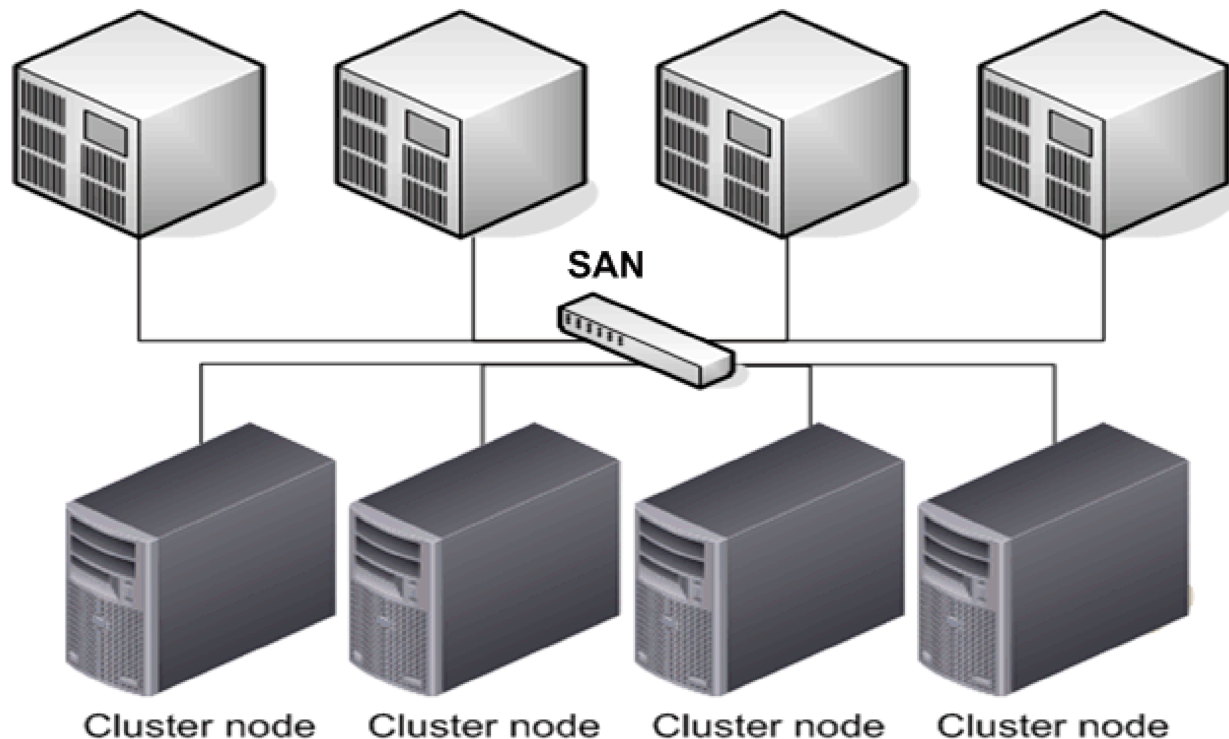


Figure 3.11: A 4-way cluster using a SAN.

In this example, the external disk arrays and the cluster nodes are all connected to a specialized network that replaces the copper SCSI cables. Many SANs use fiber-optic based connections to create a Fibre Channel (FC) SAN; in the future, it might be more common to see the SAN employ new technologies such as iSCSI over less-expensive Gigabit Ethernet (GbE) connections. In either case, the result is streamlined connectivity. You can also eliminate the need for separate external drive arrays, instead relying on external arrays that are logically partitioned to provide storage space for each node.

SQL Server Clusters

In a SQL Server cluster, each cluster node runs at least one *virtual SQL Server*. In a 2-node, active-active cluster, each node runs two virtual servers; in a 4-node cluster, each node runs four virtual servers. In a simple configuration, only one virtual server per node is actually running (although that doesn't have to be the case; multiple instances of SQL Server can be clustered). When a node fails, another node in the cluster runs the corresponding virtual server and takes over the operations for the failed node. Figure 3.12 provides a basic illustration of this concept.

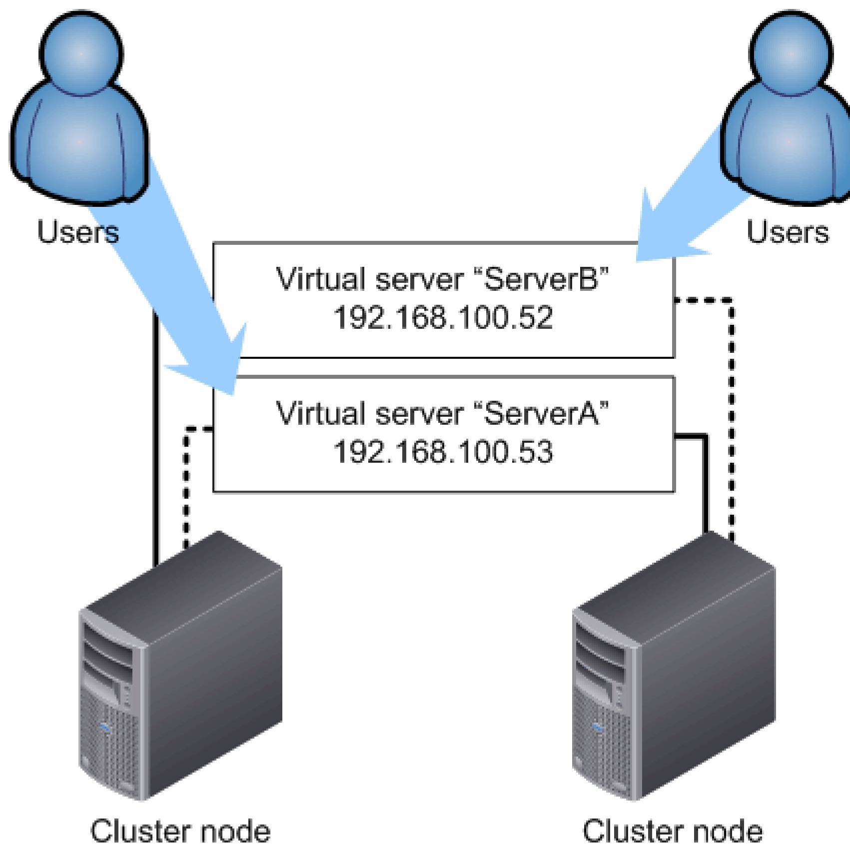
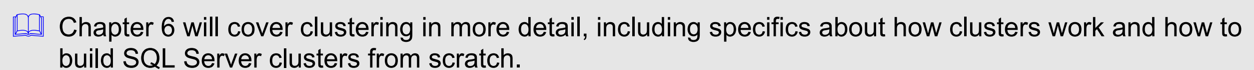


Figure 3.12: Virtual SQL Server failover cluster.

Users access the virtual servers by using a virtual name and IP address (to be very specific, the applications use the virtual name, not the IP address, which is resolved through DNS). Whichever node "owns" those resources will receive users' requests and respond appropriately.


Chapter 6 will cover clustering in more detail, including specifics about how clusters work and how to build SQL Server clusters from scratch.

Discussing clustering in SQL Server terminology can be confusing. For example, you might have a 2-node cluster that represents one logical SQL Server (meaning one set of databases, one configuration, and so forth). This logical SQL Server is often referred to as an *instance.* Each node in the cluster can "own" this instance and respond to client requests, meaning each node is configured with a virtual SQL Server (that is the physical SQL Server software installed on disk).

A 2-node cluster can also run two instances in an active-active configuration, as I've discussed. In this case, each node typically "owns" one instance under normal conditions; although if one node fails, both instances would naturally run on the remaining node. A 2-node cluster can run more instances, too. For example, you might have a 2-node cluster acting as four logical SQL Server computers (four instances). Each node would "own" two instances, and either node could, in theory, run all four instances if necessary. Each instance has its own virtual server name, related IP address, server configuration, databases, and so forth.

This capability for clusters to run multiple instances often makes the *active-passive* and *active-active* terminology imprecise: Imagine an 8-node cluster running 12 instances of SQL Server, where half of the nodes are "doubly active" (running two instances each) and the others are merely "active" (running one instance apiece). SQL Server clusters are thus often described in terms of nodes and instances: An 8 × 12 cluster, for example, has 8 nodes and 12 instances.

Effects of Clustering on Performance

Clustering doesn't directly impact performance and clustering of this type doesn't offer load balancing. However, SQL Server clustering does offer a very high degree of tolerance for hardware failure. Larger clusters tend to utilize fast SANs to reduce cost and complexity, and those SANs can have a very positive effect on performance (I'll discuss this idea in greater detail in Chapter 7).

Clusters can *hurt* performance through over-engineering. A best practice is to size cluster nodes so that each node will normally operate at about 50 percent capacity. That way, if another node fails, the node that picks up the failed node's share of the workload won't exceed 100 percent capacity. If your nodes are all built to 70 to 80 percent capacity (a more common figure for standalone servers), a node failure will result in one server trying to carry 140 to 160 percent of its usual load, which is obviously impossible. The result is drastically reduced performance.

> ☞ Buy pre-built, commodity clusters. Microsoft's Windows Cluster Server can be a picky piece of software and has its own Hardware Compatibility List (HCL). Although building a cluster isn't necessarily difficult, you need to be careful to get the correct mix of software in the correct configuration. An easier option is to buy a preconfigured, pre-built cluster (rather than buying pieces and building your own). Many manufacturers, including Dell, IBM, and Hewlett-Packard, offer cluster-class hardware and most will be happy to work with you to ship a preconfigured cluster to you. Even if you don't see a specific offer for a cluster, ask your sales representative; most manufacturers can custom-build a cluster system to your specifications.
>
> Also look for clusters built on commodity hardware, meaning servers built on the basic PC platform without a lot of proprietary hardware. In addition to saving a significant amount of money, commodity hardware offers somewhat less complexity to cluster configuration because the hardware is built on the basic, standard technologies that the Windows Cluster Server supports. Manufacturers such as Dell and Gateway offer commodity hardware.

Creating a Scale-Out Lab

Once you've decided how to create your scale-out solution, you need to test it. I'm a strong advocate of creating a *throwaway pilot,* meaning you build a complete pilot in a lab, test it, gather the facts you need, document what worked and what didn't, and then ditch the pilot. You'll start fresh when building your production solution, keeping all the good things you discovered in your pilot and leaving out all the bad. To do so, you'll need to be able to conduct real-world testing and you'll need to have some benchmarks available to you.

Real-World Testing

Perhaps the toughest part of conducting a scale-out pilot is getting enough data and users to make it realistic. Try to start with a recent copy of the production database by pulling it from a backup tape because this version will provide the most realistic data possible for your tests. If you're coming from a single-server solution, you'll need to do some work to get your database backups into their new scaled-out form.

> ☞ Whenever possible, let SQL Server's Integration Services (called Data Transformation Services, called DTS, prior to SQL Server 2005) restructure your databases, copy rows, and perform the other tasks necessary to load data into your test servers. That way, you can save the DTS packages and rerun them whenever necessary to reload your servers for additional testing with minimal effort.

It can be difficult to accurately simulate real-world loads on your servers in a formal stress test to determine how much your scaled-out solution can handle. For stress tests, there are several Microsoft and third-party stress-test tools available (you can search the Web for the most recent offerings).

For other tests, you can simply hit the servers with a good-sized portion of users and multiply the results to extrapolate very approximate performance figures. One way to do so is to run a few user sessions, then capture them using SQL Server's profiling tool (SQL Profiler in SQL Server 2000). The profiling tool allows you to repeatedly replay the session against the SQL Server, and you can copy the profile data to multiple client computers so that the session can be replayed multiple times simultaneously. Exactly how you do all this depends a lot on how your overall database application is built, but the idea is to hit SQL Server with the same type of data and traffic that your production users will. Ideally, your profile data should come from your production network, giving you an exact replica of the type of traffic your scaled-out solution will encounter.

Benchmarking

The Transaction Processing Council (TPC) is the industry's official bench marker for database performance. However, they simply provide benchmarks based upon specific, lab-oriented scenarios, not your company's day-to-day operations. You'll need to conduct your own benchmarks and measurements to determine which scale-out solutions work best for you. Exactly what you choose to measure will depend on what is important to your company; the following list provides suggestions:

- Overall processor utilization
- Number of users (real or simulated)
- Number of rows of data
- Number of transactions per second
- Memory utilization
- Network utilization
- Row and table locks
- Index hits
- Stored procedure recompiles
- Disk activity

By tracking these and other statistics, you can objectively evaluate various scale-out solutions as they relate to your environment, your users, and your database applications.

Summary

In this chapter, you've learned about the various scale-out techniques and the decision factors that you'll need to consider when selecting one or more techniques for your environment. In addition, we've explored the essentials of building a lab to test your decisions and for benchmarking real-world performance results with your scale-out pilot.

A key point of this chapter is to establish a foundation of terminology, which I will use throughout the rest of the book. The terms *distributed* and *partitioned* come up so frequently in any scale-out discussion that it can be easy to lose track of what you're talking about. The following points highlight key vocabulary for scale-out projects:

- *Partitioned* refers to what breaks up a database across multiple servers.
- A *vertically partitioned* database breaks the schema across multiple servers so that each server maintains a distinct part of the database, such as customer records on one server, and order records on another server. This can also be referred to simply as a partitioned database.
- A *horizontally partitioned* database breaks the data rows across multiple servers, which each share a common schema.

- *Distributed* simply refers to something spread across multiple servers. Generally, anything that is partitioned is also distributed (however, you can partition on a single server); the partitioning method tells you how the database is broken up before distribution.
- A *simple distributed database* may refer to one which has been horizontally partitioned, perhaps with rows from different regions physically contained on servers in those regional offices. *Replication* is often used in these situations so that each server contains a copy of the other servers' rows.
- Another kind of distributed database may simply replicate all of a server's data to one or more other servers. This configuration allows users in various locations to query a local copy of the server and increases the number of users that can be supported. The database isn't partitioned, and techniques such as merge replication may be used to handle multiple conflicting updates to the data.
- A complete *distributed partitioned database* is one that combines the techniques we've explored in this chapter. The database is typically partitioned horizontally, with each server containing different rows and using the same database schema. However, replication isn't typically used, and each server doesn't contain a complete copy of the data. To access the complete database as a single unit, *distributed partitioned views* are used to virtually recombine rows from multiple servers. In cases in which distributed partitioned views aren't suitable due to performance reasons, an intelligent middle tier takes on the task of directing queries to the appropriate server or servers.

In the next chapter, I'll dive into scaling out by using distributed partitioned views. I'll discuss in more detail how they work and will provide step-by-step instructions for creating a scaled-out database that uses distributed partitioned views to reduce complexity and potentially provide radically increased performance.

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/.