

realtimepublishers.comtm

The Definitive Guidetm To

**Scaling Out
SQL Server 2005**

Don Jones

Chapter 5: Distributed and Partitioned Databases	95
Pros and Cons	95
Distributed Databases	95
Partitioned Databases.....	99
Design and Implementation	103
Designing the Solution.....	103
Distributed Databases	103
Partitioned Databases.....	107
Implementing the Solution.....	108
Distributed Databases	108
Partitioned Databases.....	113
Best Practices	116
Benchmarks.....	117
Summary	119



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 5: Distributed and Partitioned Databases

The phrases *distributed* and *partitioned* mean different things to different people when it comes to databases. Strictly speaking, *distributed* refers to any database that exists in more than one place (on more than one server), and a *partitioned* database is divided into multiple sections, with each section existing on a different server. In this chapter, I'll discuss the pros and cons of these scale-out techniques and walk you through the steps necessary to implement each.

Pros and Cons

There are a number of good and bad points about partitioned and distributed databases. In most applications, the biggest drawback to partitioned and distributed databases for database administrators and developers is the level of complexity. Involving more database servers in your environment obviously increases maintenance and administrative effort; changing the way in which data is distributed across servers can create obvious difficulties for client applications that are hard-coded to look for data in specific locations or, at least, on one server.

Distributed Databases

Distributed databases are an easy way to bring more processing power to a database application. There are two reasons to distribute:

- To place data in closer physical proximity to more users. For example, you might distribute a database so that a copy exists in each of your major field offices, providing a local database for each office's users.
- To absorb a greater amount of traffic than a single database server can handle. For example, a Web site might use multiple read-only copies of a database for a sales catalog, helping to eliminate the database back end as a bottleneck in the number of hits the Web site can handle.

Replication is used to keep the databases in sync. For example, Figure 5.1 shows an example distributed database.



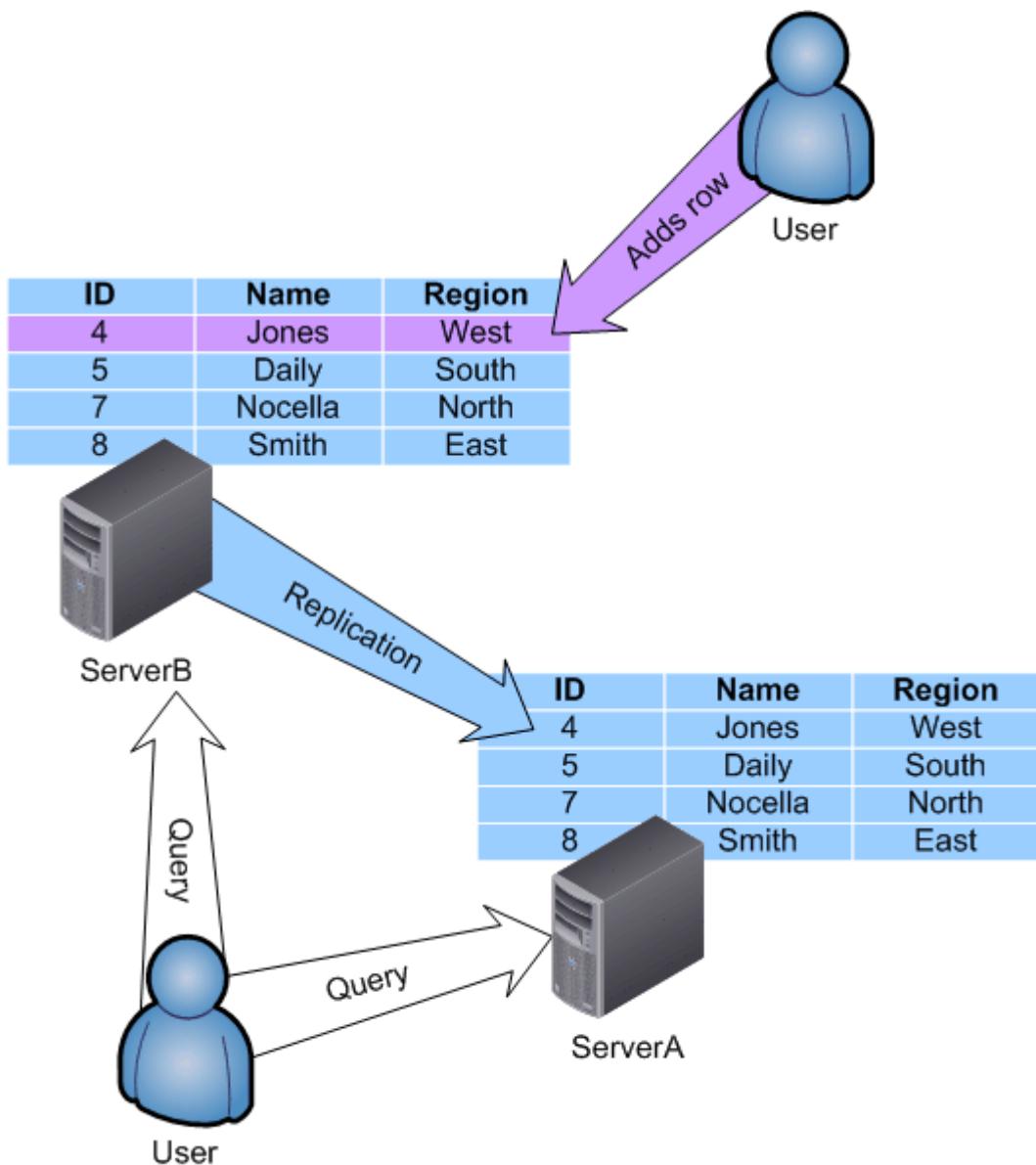


Figure 5.1: A distributed database example.

In this example, the database exists on two servers, ServerA and ServerB. Each server contains an identical copy of the database, including the database schema and the actual data contained within the database.

Suppose that a user adds a new database row to ServerB, which then replicates the changes to ServerA. Both servers again have an identical copy of the data. The downside to this arrangement is that the two database servers will always be slightly out of sync with one another, particularly in a busy environment in which data is added and changed frequently. SQL Server offers multiple types of replication (and SQL Server 2005 specifically adds database mirroring, which is conceptually similar to replication), which I'll cover later in this chapter, that each uses a different method to strike a balance between overhead and synchronization latency.



The design of your distributed database will affect its latency as well. For example, consider the four-server distributed database in Figure 5.2. In this example, the administrator has created a *fully enmeshed* replication topology, which means that each server replicates directly with every other server. Changes made on any one server are pushed out to the other three servers. This technique reduces latency because only one “hop” exists between any two servers. However, this technique also increases overhead, because each server must replicate each change three times.

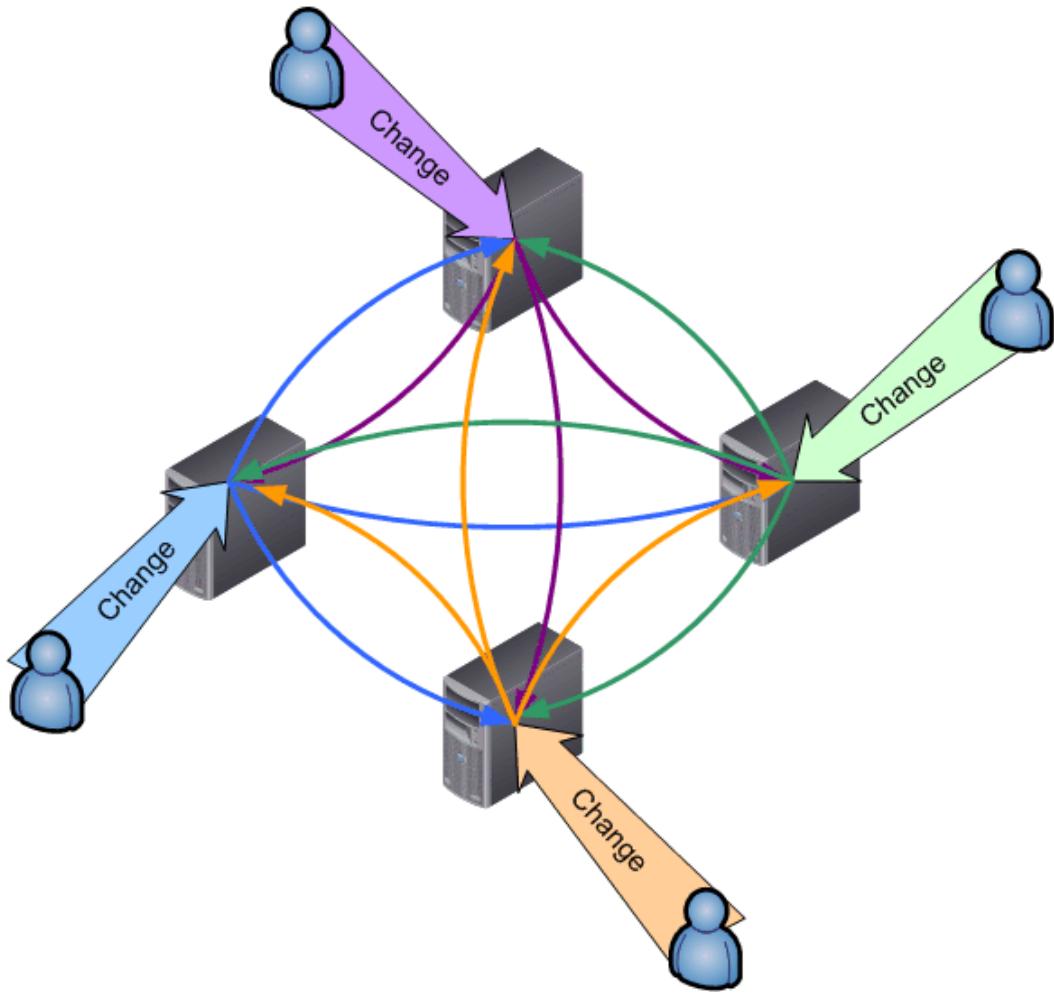


Figure 5.2: A replication topology example.

Another technique is to have ServerA replicate changes only to ServerB; ServerB to ServerC; ServerC to ServerD; and ServerD to ServerA. This circular topology ensures that every server replicates each change only once, which reduces overhead. However, latency is increased because as many as three “hops” exist between any two servers. For example, a change made on ServerA must replicate to ServerB, then to ServerC, and then to ServerD—creating a much longer lag time before ServerD comes into sync with the rest of the servers. The amount of overhead and latency you are willing to tolerate will depend on how complex you are willing to make your environment, and how much overhead and latency your business applications and users can handle.





Latency is the most important consideration, from a business perspective, in designing replication. At the very least, users need to be educated so that they understand that the database exists in multiple copies, and that the copies won't always be in sync. Make users aware of average replication times so that they have reasonable expectations of, for example, the time necessary for their changes to be replicated.

Your business needs will determine how much latency is acceptable. For example, latency of a couple of minutes might not matter to most applications. However, applications that depend on real-time data might not tolerate even a few seconds of latency; in such cases, an alternative, third-party solution for synchronizing data will be required.

The previous examples are geared toward a database that is distributed across multiple physical locations; another technique, which Figure 5.3 shows, is to create multiple database servers to support multiple Web servers.

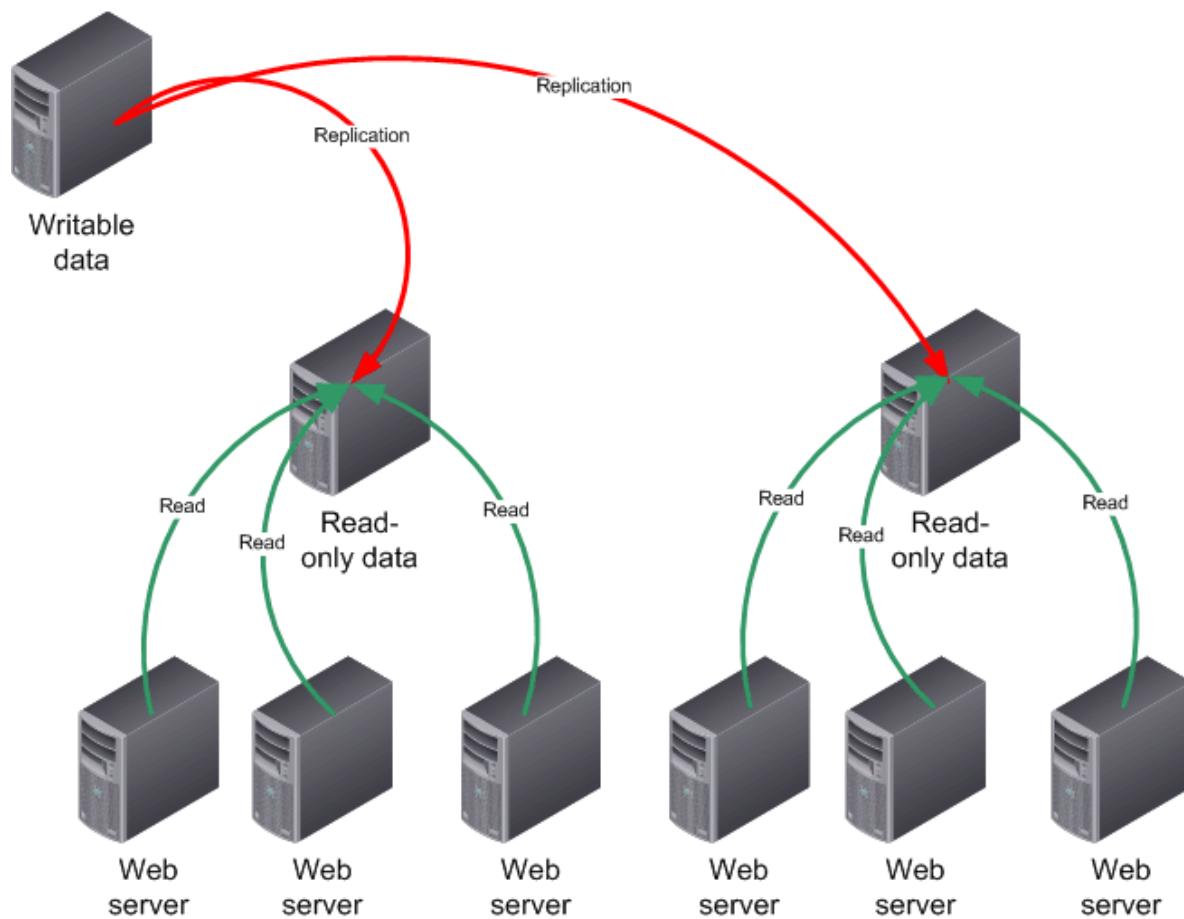


Figure 5.3: Distributed databases for a Web site.

In this example, one database server holds a writable copy of the database. Internal users make changes to this copy, and the changes are then replicated to the read-only databases accessed by the Web servers. This model is infinitely scalable; if you determine that each database server can support, for example, 50 Web servers, then you simply deploy a new database server for every 50 Web servers you add to your environment. The technique works well primarily for read-only data, such as an online product catalog. Typically, the Web servers would access a second database server with data changes, such as new orders.





Any data that doesn't change very frequently or isn't changed by a large number of users is an excellent candidate for this type of replication. A single, writable copy eliminates any possibility of conflicts, which can happen if data is changed in multiple locations. Multiple read-only copies provide an easy scale-out method for large applications, particularly Web sites that must support tens of thousands of users.

Partitioned Databases

The previous Web site example makes a nice segue into the pros and cons of partitioned databases. Figure 5.4 shows an evolution of the Web site example that includes a fourth database server used to store customer orders. This server is written to by the Web servers.

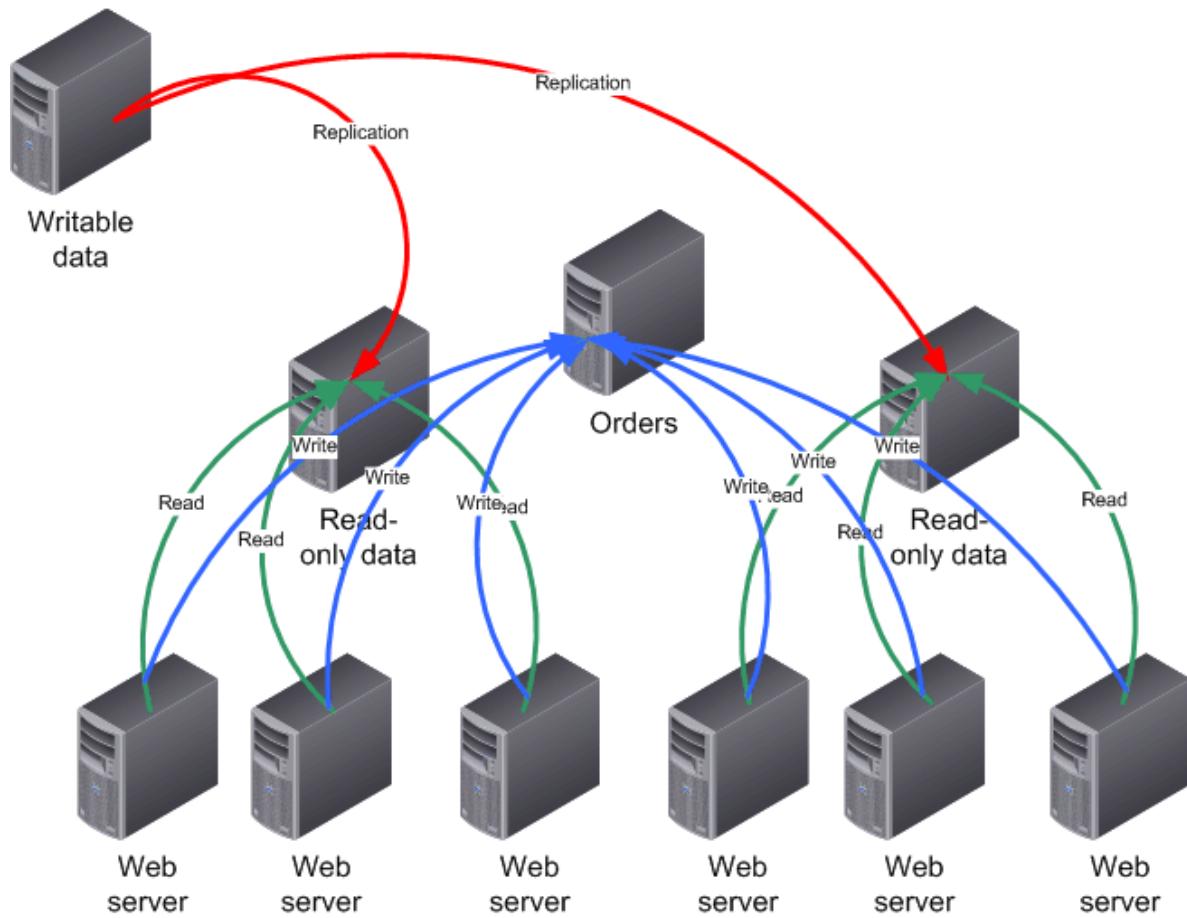


Figure 5.4: A distributed and partitioned database.



This example illustrates a form of partitioned database. Part of the database—the catalog information—is stored on one set of database servers; another part—customer orders—is stored on another server. The databases are interrelated, as customers place orders for products that are in the catalog. In this example, the purpose of the partitioning is to distribute the overall workload of the application across multiple servers; because the server storing orders doesn't need to serve up product information, its power is conserved for processing new orders. In a particularly large Web site, multiple servers might be required to handle orders, and they might replicate data between one another so that each server contains a complete copy of all orders, making it easier for customers to track order status and so forth.

Partitioning a database in this fashion presents challenges to the database administrator and the application developer. In this Web site example, the developer must know that multiple servers will be involved for various operations so that the Web servers send queries and order information to the appropriate server. Each Web server will maintain connections to multiple back-end database servers.

This complexity can be dispersed—although not eliminated—by creating multi-tier applications. As Figure 5.5 shows, the Web servers deal exclusively with a set of middle-tier servers. The middle-tier servers maintain connections to the appropriate back-end database servers, simplifying the design of the Web application. This design introduces an entirely new application tier—the middle tier—which has to be developed and maintained, so the complexity hasn't been eliminated; it has merely been shifted around a bit.



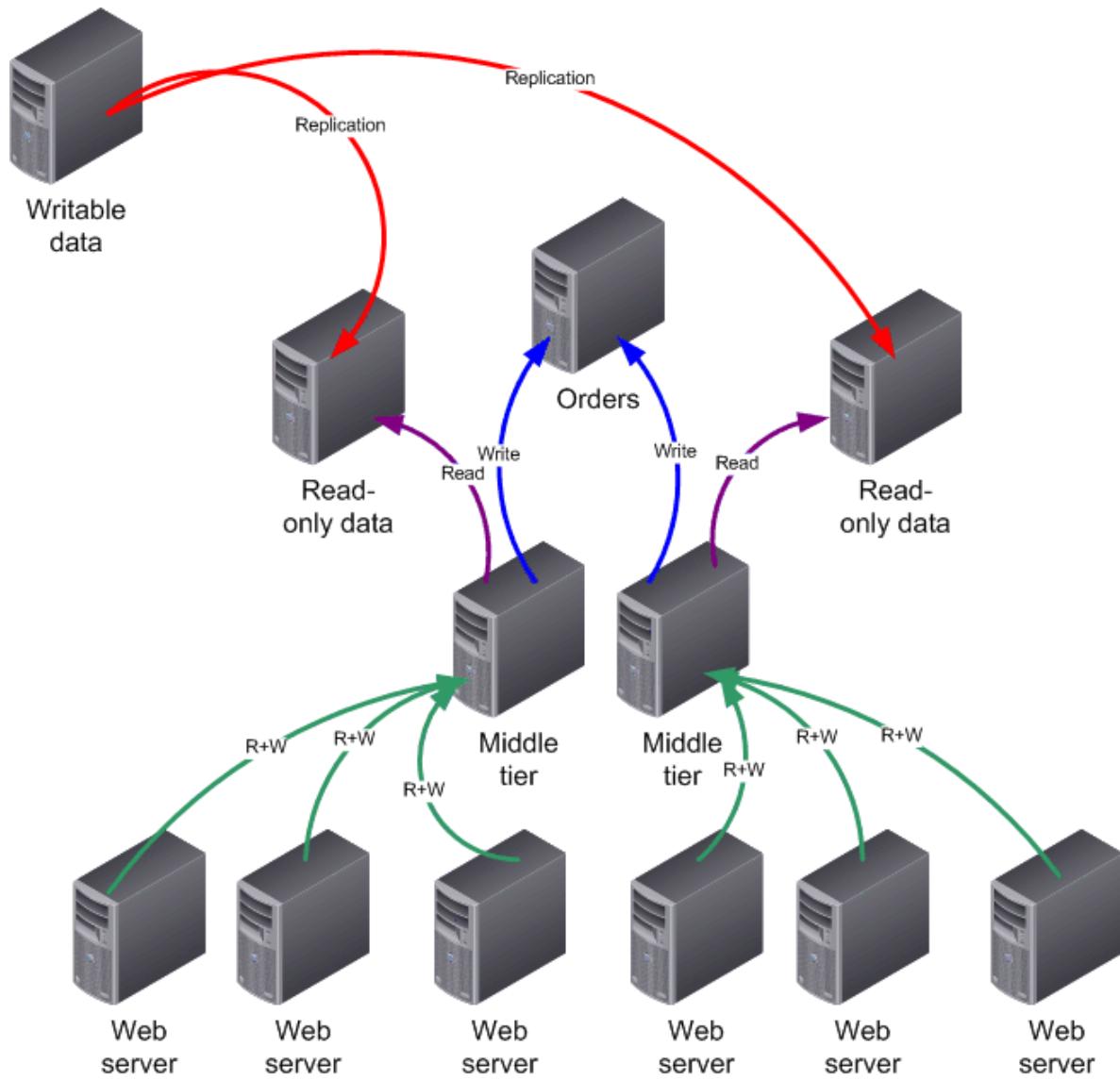


Figure 5.5: Using an n-tier design to manage application complexity.

The point is that partitioned databases always increase complexity. Data has multiple paths across which it can flow, and different servers are designated with specific tasks, such as serving up catalog data or storing order data. These database designs can allow you to create staggeringly large database applications, but you will pay for the power in more complex maintenance and software development. This Web site scenario is an example of a *vertically partitioned* database, in which different tables of the database are handled by different servers.

Figure 5.6 is a simpler model of vertical partitioning in which different tables are split between two servers. Again, the problem with this technique is that it places a burden on the software developer to know where specific bits of data are being stored.



SQL Server offers components that help to reduce this complexity. For example, you can create views that pull from multiple tables on different servers. Views work similarly to distributed partitioned views, which I covered in the previous chapter. Distributed partitioned views are designed to work with horizontally partitioned databases; you can also create regular views that help to consolidate vertically partitioned data.

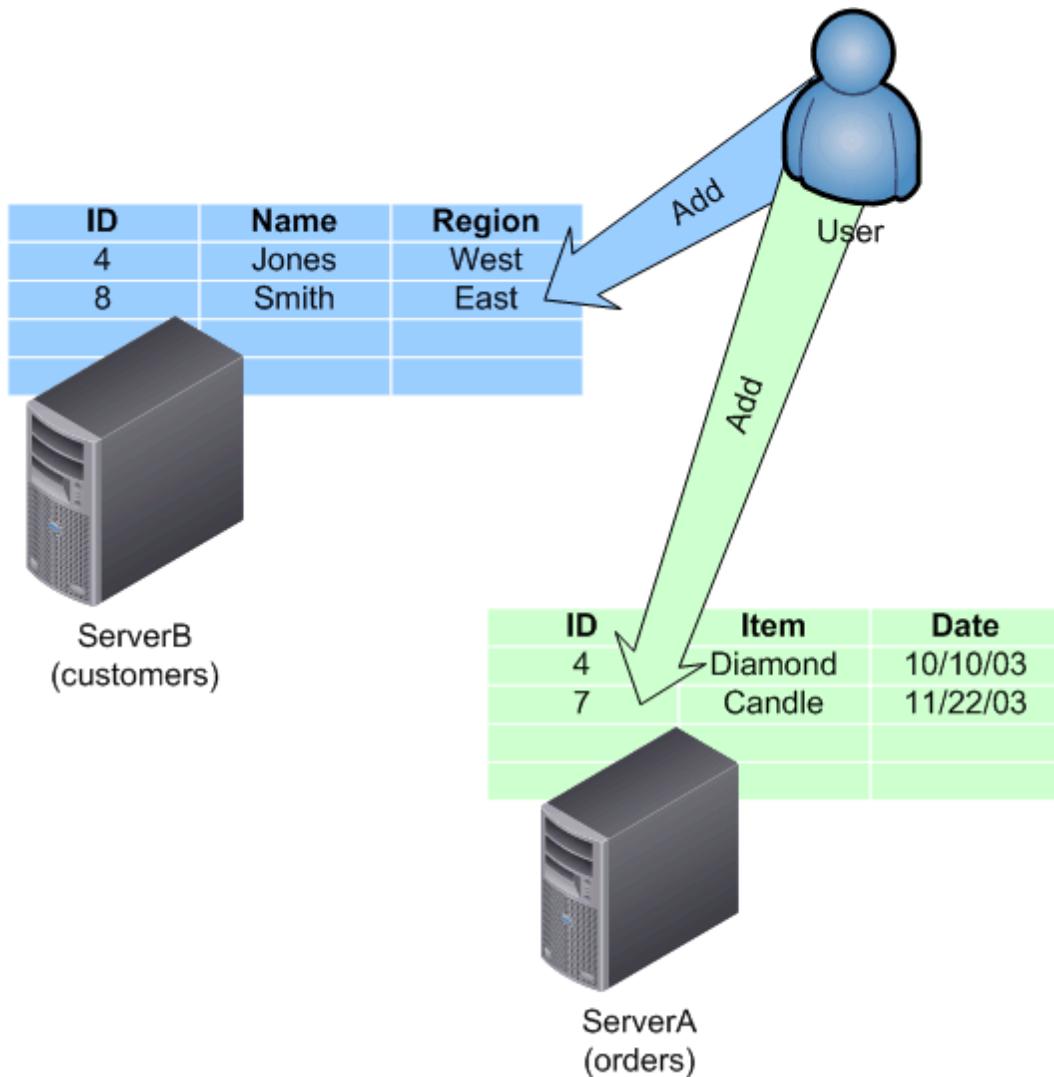


Figure 5.6: Typical vertical partitioning.

Views become a key to helping make the multiple servers appear to be one large server, a technique I'll discuss later in this chapter when I show you how to implement partitioned databases. However, views don't always work well if the servers are physically separated; partitioning a database usually precludes physically distributing the servers across WAN links for simple performance reasons.



Design and Implementation

Designing and implementing distributed or partitioned databases involves fairly straightforward decisions. Many applications, such as the Web site example I discussed in the previous section, might involve both partitioned and distributed databases. In cases of a mixed-approach scale-out method, handle the design of the distributed and partitioned portions individually. The fact that you are using both techniques as part of an overall solution doesn't appreciably affect the design decisions you will make.

Designing the Solution

Designing a distributed database involves decisions that affect data latency and replication overhead; partitioning a database requires you to address concerns about application complexity and database maintenance. Three distinct designs are possible:

- A distributed database in which each copy is a complete copy of the entire database and one copy does not “own” any particular rows.
- A distributed database in which each copy is a complete copy of the entire database and each copy of the database has been assigned, through horizontal partitioning, as the “owner” of particular rows.
- A vertically partitioned database in which each server contains only a portion of the database’s schema and data.

Distributed Databases

A basic design rule is that a distributed database is useful when you need to make multiple copies of data available. Perhaps you want the copies to be physically distributed so that the copies are close to individual user populations, or perhaps you need multiple copies to support the back-end requirements for a large application. In either case, multiple copies of a database create specific problems:

- Changes to the copies must somehow be reconciled.
- Reconciliation has processing overhead associated with it.
- Reconciliation has a time factor, referred to as latency, associated with it.

SQL Server’s replication features are designed to handle data reconciliation with varying degrees of overhead, latency, and ability to handle conflicting changes.

 One way to neatly avoid most of the problems raised by distributed databases is to allow the copies of the database to be read-only. If changes are made only on one copy, then those changes are distributed to read-only copies, and you only need to be concerned about the latency in pushing out changes to the read-only copies. Some applications lend themselves to this approach; many do not.

To begin, let’s cover some basic SQL Server replication terminology. First, an *article* is the smallest unit of data that SQL Server can replicate. You can define an article to be a table, a vertical or horizontal partition of data, or an entire database. Articles can also represent specific stored procedures, views, and other database objects.



Articles are made available from a *publisher*, which contains a writable copy of the data. A *subscriber* receives replication changes to the article. A *distributor* is a special middleman role that receives replication data from a publisher and distributes copies to subscribers, helping to reduce the load of replication on the publisher. A *subscription* is a collection of articles and a definition of how the articles will be replicated. *Push* subscriptions are generated by the publisher and sent to subscribers; *pull* subscriptions are made available to subscribers, which must connect to receive the subscription's data.

In a case in which multiple servers will contain writable copies of the data, each server will act both as publisher and subscriber. In other words, ServerA might publish any changes made to its copy of the data while simultaneously subscribing to changes that occur on ServerB, ServerC, and ServerD. SQL Server has no problem with a single server both sending and receiving changes to a database. SQL Server supports different types of replication:

- Snapshot replication is designed to copy an entire article of data at once. SQL Server must be able to obtain an exclusive lock on all the data contained in the article, and can compress the replicated data to conserve network bandwidth. Because of the requirement for an exclusive lock, snapshot replication isn't suitable for high-volume transactional databases; this replication type is used primarily for data that is mostly static. Snapshots can be high-overhead when the snapshot is taken, meaning you'll schedule snapshots to occur infrequently. Subscribers to the snapshot replace their copy of the data with the snapshot, meaning there is no capability to merge copies of the database and handle conflicts. Snapshots are often a required first step in establishing other types of replication so that multiple copies of the database are known to be in the same condition at the start of replication.

 Snapshot replication is most useful for distributing read-only copies of data on an infrequent basis.

- Transactional replication begins with an initial snapshot of the data. From there, publishers replicate individual transactions to subscribers. The subscribers replay the transactions on their copies of the data, which results in the copies of the database being brought into synchronization. No facility for handling conflicts is provided; if two publishers make changes to the same data, their published transactions will be played on all subscribers, and the last one to occur will represent the final state of the replicated data. Transactional replication is fairly low-bandwidth, low-overhead, and low-latency, making it ideal for most replication situations. It is often paired with a form of horizontal partitioning, which might assign specific database rows to specific copies of the database. Doing so helps to reduce data conflicts; you might, for example, assign different blocks of customer IDs to different field offices so that the different offices avoid making changes to each others' data.

 Transactional replication offers the easiest setup and ongoing maintenance. It deals poorly with conflicting changes, so it is best if the database is horizontally partitioned so that each publisher tends to change a unique group of rows within each table. Transactional replication is also well-suited to data that doesn't change frequently or that is changed by a small number users connecting to a particular publisher.



- Merge replication is perhaps the most complex SQL Server replication technique. Also starting with a snapshot, merge replication works similarly to transactional replication except that interfaces are provided for dealing with conflicting changes to data. In fact, you can develop customized resolvers—or use one of SQL Server’s built-in resolvers—to automatically handle changes based on rules. Merge replication offers low-latency and creates an environment in which changes can be made to data in multiple places and resolved across the copies into a synchronized distributed database.

 Merge replication offers the most flexibility for having multiple writable copies of data. However, this replication type can have higher administrative and software development overhead if SQL Server’s built-in default resolver isn’t adequate for your needs.

For merge replication, SQL Server includes a default resolver; its behavior can be a bit complex. Subscriptions can be identified as either global or local, with local being the default. For local subscriptions, changes made to the publisher of an article will always win over changes made by a subscriber. You might use this method if, for example, a central office’s copy of the database is considered to be more authoritative than field office copies. However, care must be taken in client applications to re-query data for changes, and users must be educated to understand that their changes to data can be overridden by changes made by other users.

Subscriptions identified as global carry a priority—from 0.01 to 99.99. In this kind of subscription, subscribers are synchronized in descending order of priority, and changes are accepted in that order. Thus, you can define levels of authority for your data and allow certain copies of your data to have a higher priority than other copies.

 Merge replication was designed to understand the idea of changes occurring at both the subscriber and publisher, so you don’t need to create a fully enmeshed replication topology in which each copy of the data is both a publisher and subscriber. Instead, select a central copy to be the publisher and make all other copies subscribers; merge resolvers then handle the replication of changes from all copies.

SQL Server also includes an interactive resolver, which simply displays conflicting changes to data and allows you to select which change will be applied. It is unusual to use this resolver in an enterprise application, however; it is far more common to write a custom resolver if the default resolver doesn’t meet your needs. Custom resolvers can be written in any language capable of producing COM components, including Microsoft Visual C++. Of course, SQL Server 2005 integrates the Microsoft .NET Common Language Runtime, making .NET a possibility for writing merge resolvers.

 While SQL Server 2000 only supported COM-based resolvers, SQL Server 2005 supports both COM-based custom resolvers and business logic handlers written in managed (.NET) code.



As I mentioned earlier, transactional replication is by far the most popular form of replication in SQL Server, in no small part because it is so easy to set up and an excellent choice when creating distributed databases. To help avoid the problem of conflicting changes, transactional replication is often paired with horizontal partitioning of data. For example, Figure 5.7 shows how a table has been divided so that one server contains all even-numbered primary keys, and a second server contains odd-numbered keys. This partitioning represents how the data is used—perhaps one office only works with odd-numbered clients and another focuses on the evens—reducing the number of data conflicts.

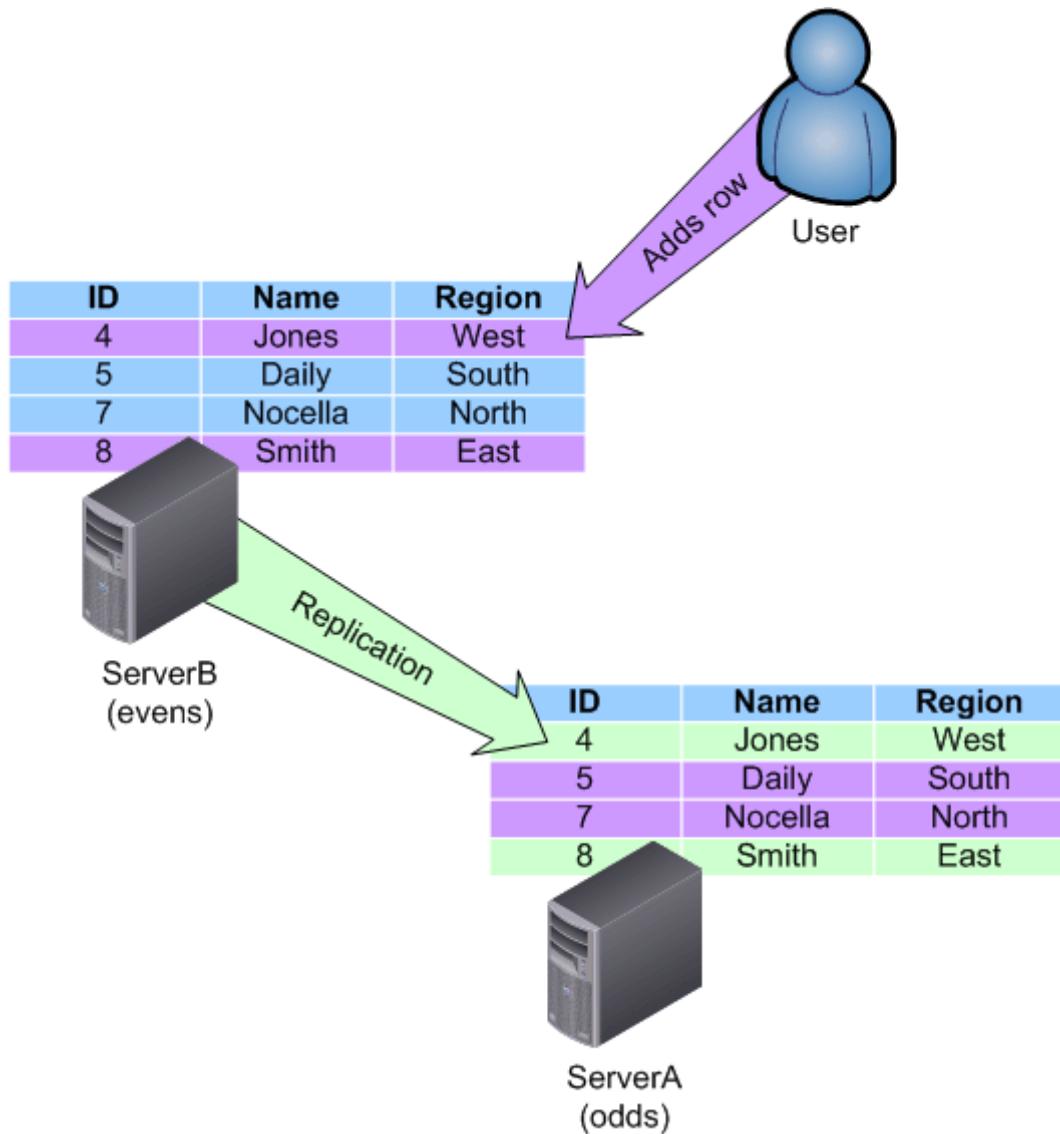


Figure 5.7: Horizontal partitioning and replication.

A more common technique is to create a *partitioning column*. For example, customer records might have a Region column that contains a value indicating which regional field office deals with that customer the most. Conflicting changes to the customer's data will be rare, as most changes will be made to only that region's data, with the change then replicated to other regions' database servers.



Partitioned Databases

Partitioning a database is usually performed to accomplish one of two goals:

- Distribute processing workload so that different database servers handle different portions of the database. This setup is usually accomplished through vertical partitioning.
- Segregate portions of the database so that, although copies exist on multiple servers, certain parts of the data are “owned” by only a single server. This setup is usually accomplished through horizontal partitioning and is often used in conjunction with replication, as I’ve already described.

Horizontal partitioning is a simpler matter, so I’ll cover it first. It is simply a matter of separating the rows of your database so that particular rows can be “owned” by a specific server. To do so, you follow the same process used to create distributed partitioned views (see Chapter 4 for more information about this process). You might have a specific partitioning column, as I’ve already described, which assigns rows based on criteria that is appropriated within your business (for example, a regional code, a range of customer IDs, a state, and so on).

Vertical partitioning is more difficult because you’re splitting a database across multiple servers, as Figure 5.6 shows. Usually, you will split the database along table lines so that entire tables exist on one server or another. The best practice for this technique is to minimize the number of foreign key relationships that must cross over to other servers. Figure 5.8 shows an example.

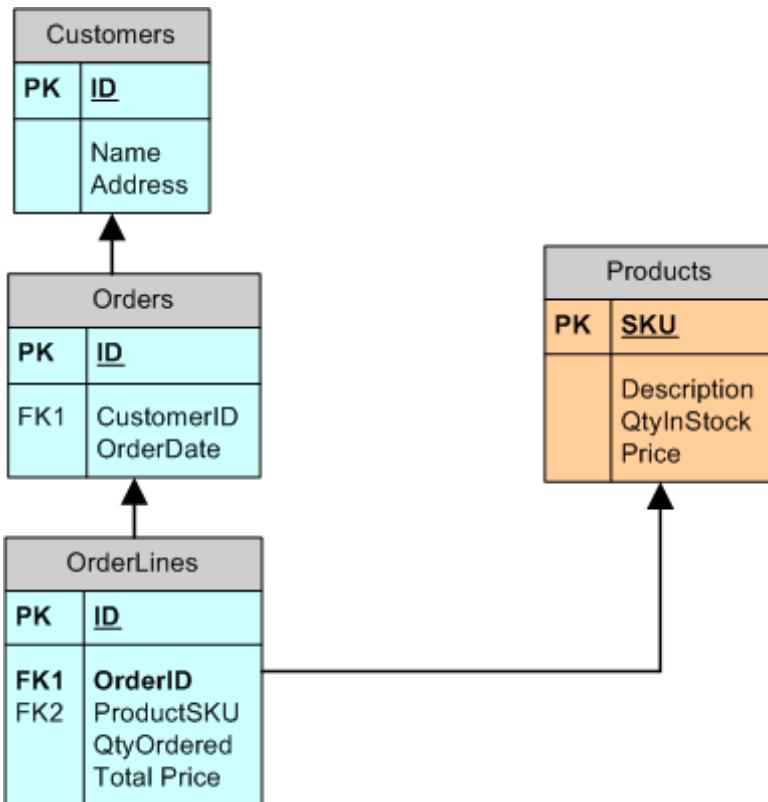


Figure 5.8: Separating tables across servers.



In this example, three tables dealing with orders and customers are kept on one server, and a table containing product information is stored on another server. This example shows only one foreign key relationship cross between servers—between the Products and OrderLines table.

☞ Depending on your needs, full partitioning might not be the best answer. For example, suppose you use the database design that Figure 5.8 shows. The reason for partitioning the database is so that the servers containing the product and order information can each handle a higher workload than if all that information was contained on a single server.

An alternative technique is to keep a copy of the product information on the server that contains the order information. Doing so would improve performance for that server because the server could maintain its foreign key relationship locally. The second server could handle actual queries for product information and replicate product changes to the order server's read-only copy of the table.

Implementing the Solution

You're ready to begin implementing your solution: What do you do first? If you're planning a blend of distributed and partitioned databases, attack the partitioning piece first because it is usually the most complicated. Once that is finished, distribution becomes primarily a matter of setting up SQL Server replication to keep your distributed copies in sync.

Distributed Databases

One of the first things you'll want to set up is replication publishing and distribution. The publisher of a subscription isn't necessarily the same server that distributes the data to subscribers; the role of distributor can be offloaded to another SQL Server computer. To configure a server as a publisher or distributor, open SQL Server Management Studio (in SQL Server 2005; for SQL Server 2000, you use SQL Enterprise Manager and the steps are slightly different). From the Object Explorer, right-click Replication, then select Configure Distribution. As Figure 5.9 shows, a wizard will walk you through the necessary steps. You can either have the publisher be its own distributor (as shown), or select one or more other servers as distributors.



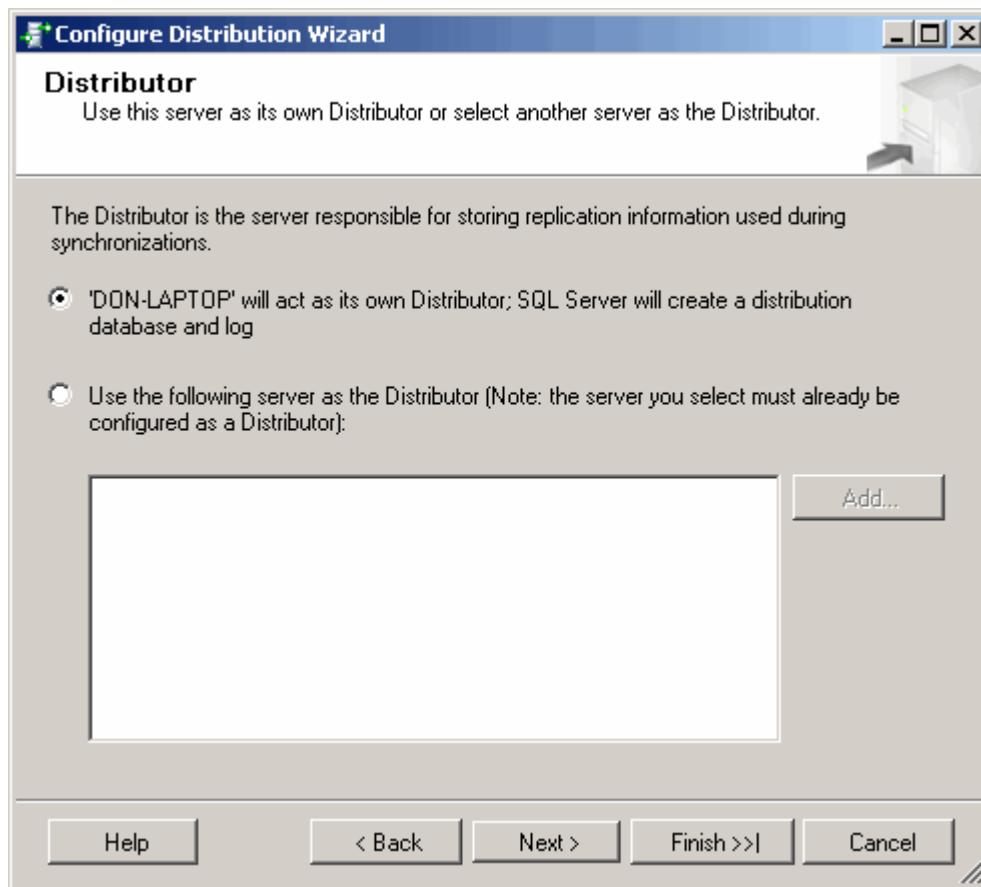


Figure 5.9: Configuring a server to be its distributor.

When configuring replication, ensure that the SQL Server Agent is configured to start using a user account that is valid on all computers that will participate in replication; generally, that will mean using a domain user account. SQL Server Agent handles much of the work involved in replication and cannot be running under the default LocalSystem account if replication is to work.

To create a publication, right-click Local Publications under Replication in Management Studio, and select New Publication. In the dialog box that appears, follow these steps:

1. Select the database from which you want to publish data.
2. Select the type of replication—Snapshot, Transactional, or Merge—that you want to use. Transactional is the most popular type, so I'll use that for the remainder of these steps.
3. As Figure 5.10 shows, select the articles you want to publish. If you're using replication to distribute an entire database, you will select all of the tables shown.



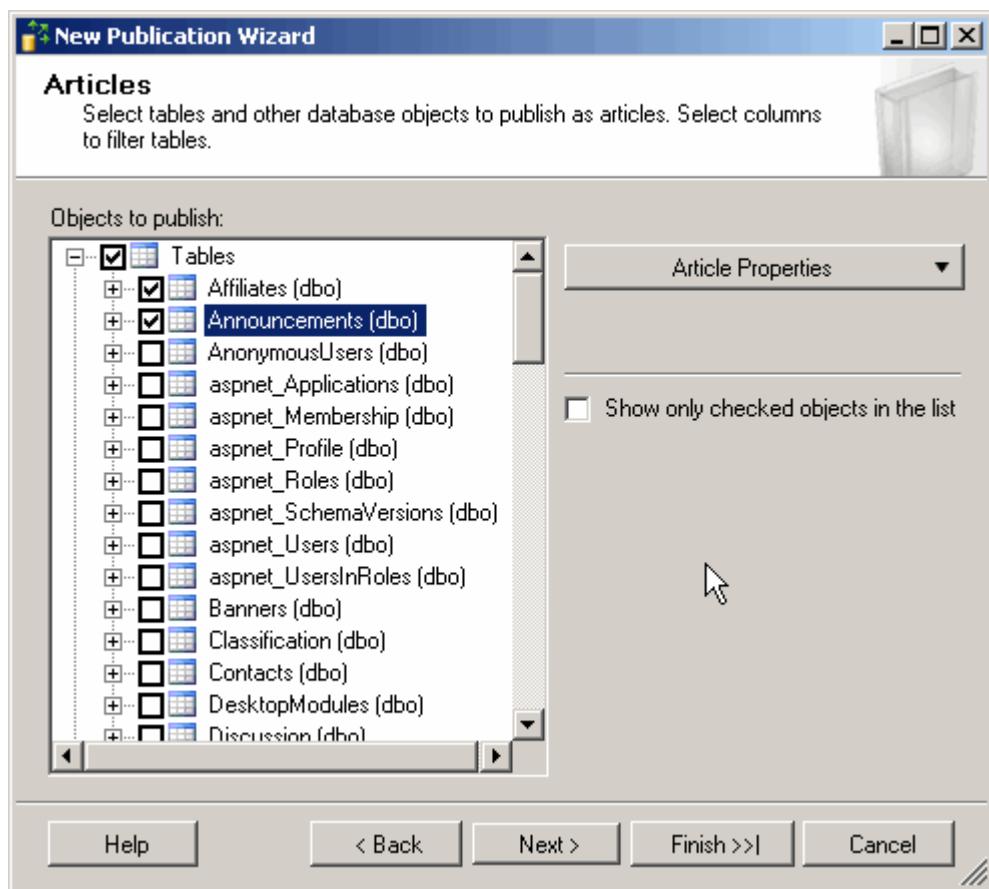


Figure 5.10: Select tables to include in the publication.

 To quickly select all tables, click the Publish All checkbox in the right-hand window, next to Tables.

4. Finish by specifying a name for the publication. You can also specify additional properties for the publication, including data filters, anonymous subscribers, and so forth. For more information about these additional properties, refer to *SQL Server Books Online*.

The Local Publications list should be updated to reflect the new publication. You can also right-click the Local Publications folder to examine the Publication Databases list (see Figure 5.11).



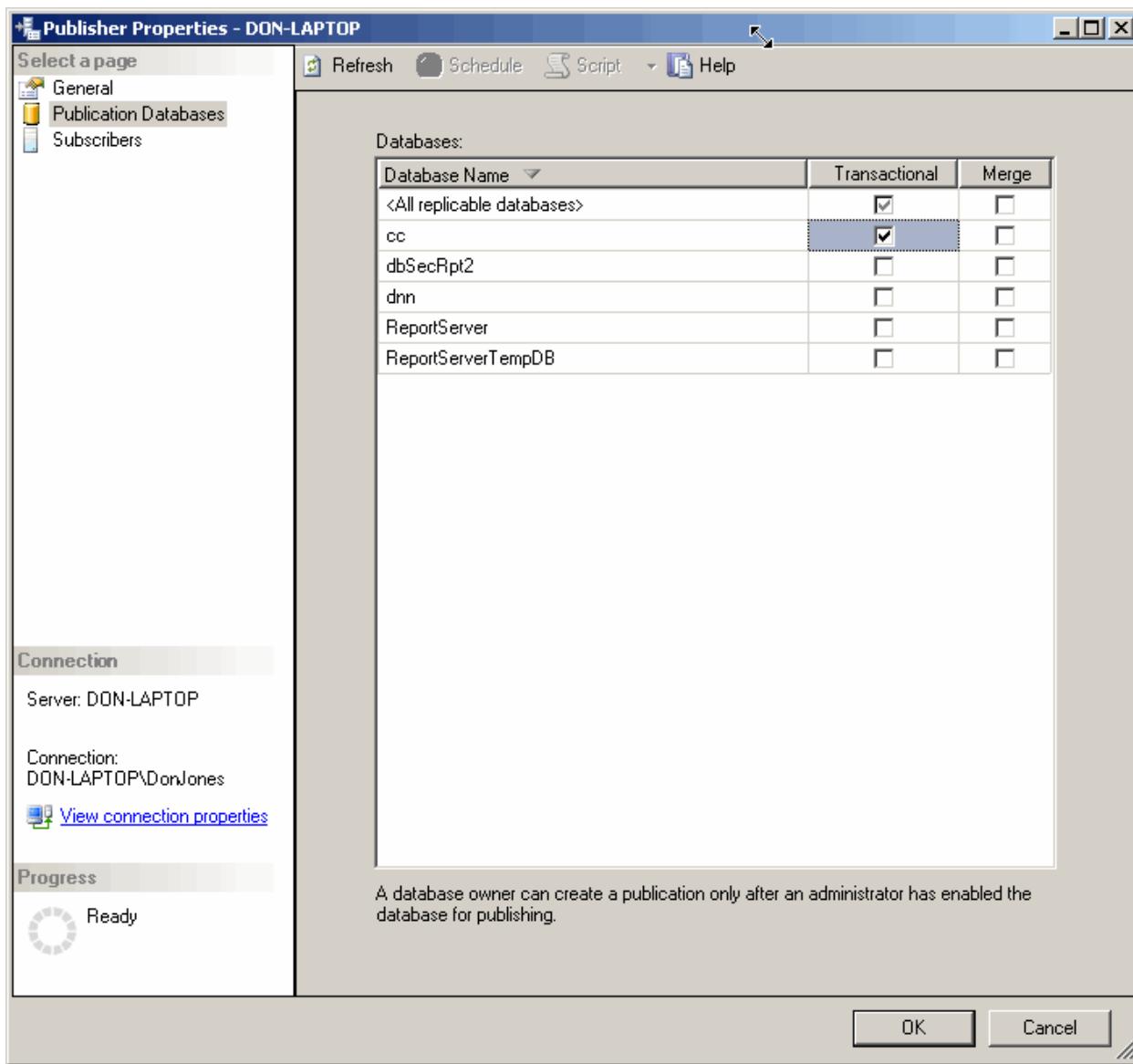


Figure 5.11: The Publication Databases list.

 There are several caveats associated with complex publications that involve multiple publishers. For example, by default, IDENTITY columns in a publication are not replicated as IDENTITY columns; they are simply replicated as normal INT columns. This default setting doesn't allow the subscribers to update the tables and create new IDENTITY values; although SQL Server can certainly handle publications in which subscribers can create new IDENTITY values, setting up these publications requires more manual effort and is beyond the scope of this discussion. For more details, consult *SQL Server Books Online*.

As an alternative, you can generate globally unique identifiers (GUIDs) to replace IDENTITY columns as unique keys. SQL Server can generate GUIDs for you, and will replicate GUIDs across servers with no conflict.

To subscribe to the publication, you will follow similar steps. For example, right-click Local Subscriptions to create a new subscription. As Figure 5.12 shows, a Wizard walks you through the entire process.



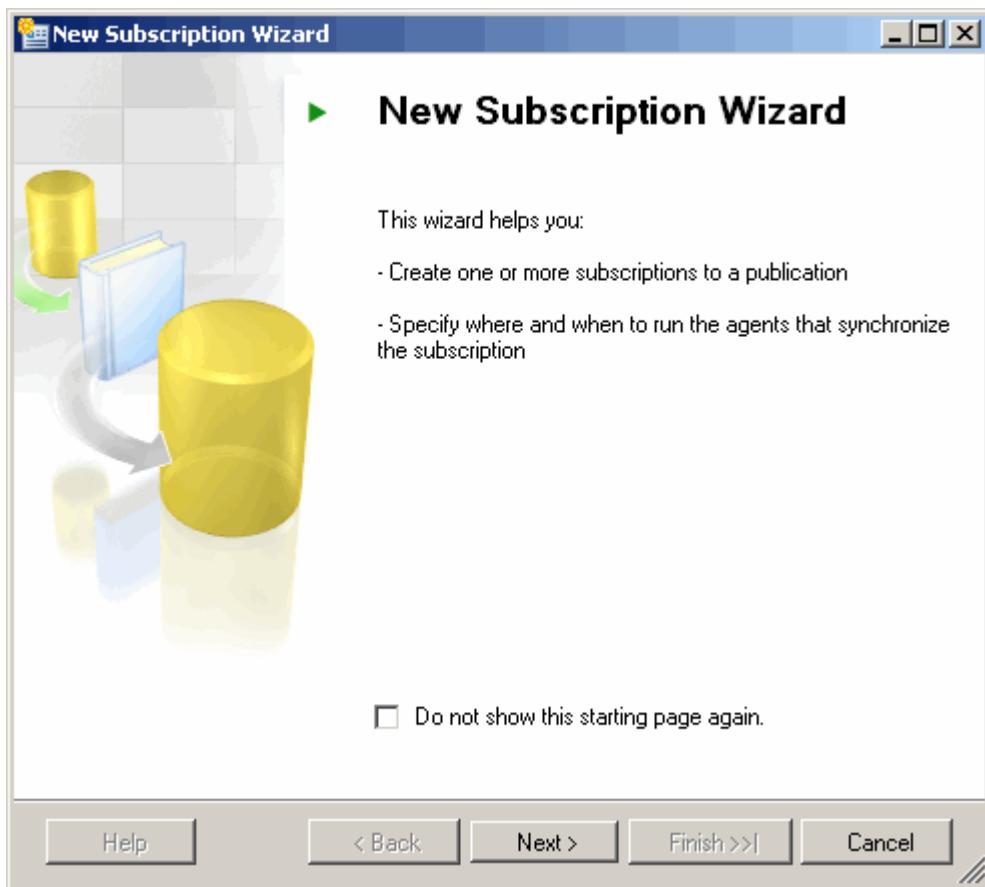


Figure 5.12: Pushing subscriptions.

To create a pull subscription, open Management Studio on the subscriber. From the Replication sub-menu, select Pull Subscription. You will see a dialog box similar to the one in Figure 5.12 listing current subscriptions. Click Pull New Subscription to create a new subscription.

Once replication is set up, it occurs automatically. SQL Server includes a Replication Monitor within Management Studio (see Figure 5.13) that you can use to monitor the processes involved in replication. In this case, the Log Reader agent is the service that monitors the SQL Server transaction log for new transactions to published articles; when it finds transactions, it engages the distributor to distribute the transactions to subscribers of the published articles.



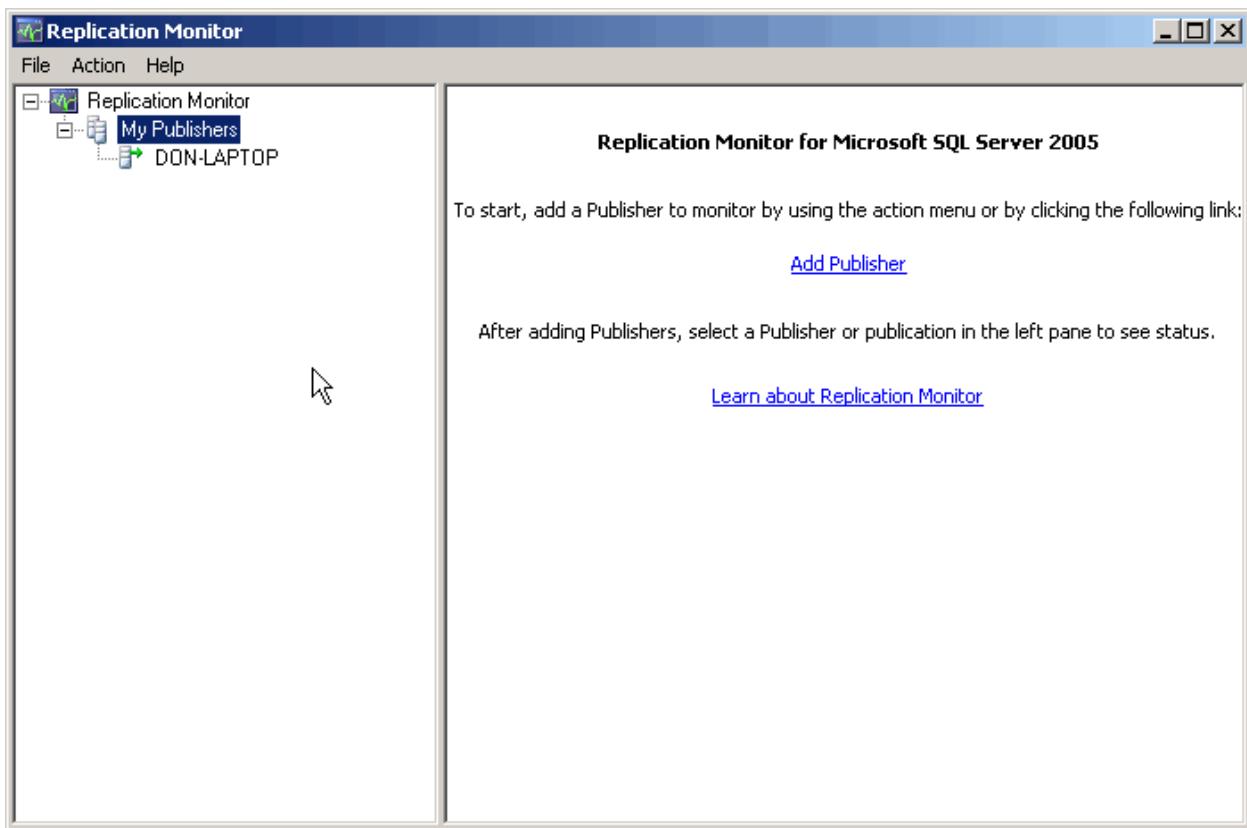


Figure 5.13: Monitoring replication.

Partitioned Databases

Vertically partitioned databases are very easy to create—simply move tables from one server to another. Deciding which tables to move is the difficult part of the process, and reprogramming client applications to deal with the new distribution of data can be a major undertaking.

Unfortunately, there are no tools or rules for designing the partitioning of a database. You will need to rely on your own knowledge of how the database works, and perhaps performance numbers that tell you which tables are most often accessed as a set. Spreading commonly-accessed tables across multiple servers is one way to help ensure a performance benefit in most situations.



There are also no tools for reprogramming your client applications to deal with the newly partitioned database. However, SQL Server does make it possible to create an abstraction between the data a client application sees and the way in which that data is physically stored, partitioned, or distributed.

One technique to help make it easier for programmers to deal with partitioned databases is views. Figure 5.14 shows an example of a vertically partitioned database in which different tables exist on different servers. A view can be used to combine the two tables into a single virtual table, which programmers can access as if it were a regular table. Stored procedures can provide a similar abstraction of the underlying, physical data storage. Applications could be written to deal entirely with the actual, physical tables; the virtual tables represented by views; or a combination of the two, depending on your environment. Keep in mind that the server hosting the view uses a bit more overhead to collect the distributed data and assemble the view; be sure to plan for this additional overhead in your design and place the views accordingly.

☞ It's also possible to use SQL Server as middle tier in partitioned database schemes. For example, you might have tables spread across ServerA and ServerB, and construct views on ServerC. Client applications would deal solely with ServerC, and ServerC would assemble virtual tables from the data on ServerA and ServerB. This setup requires significant planning but can provide a useful abstraction so that software developers don't need to be concerned with how the data is physically distributed. In addition, this configuration prevents either ServerA or ServerB from hosting all the views related to the database application.



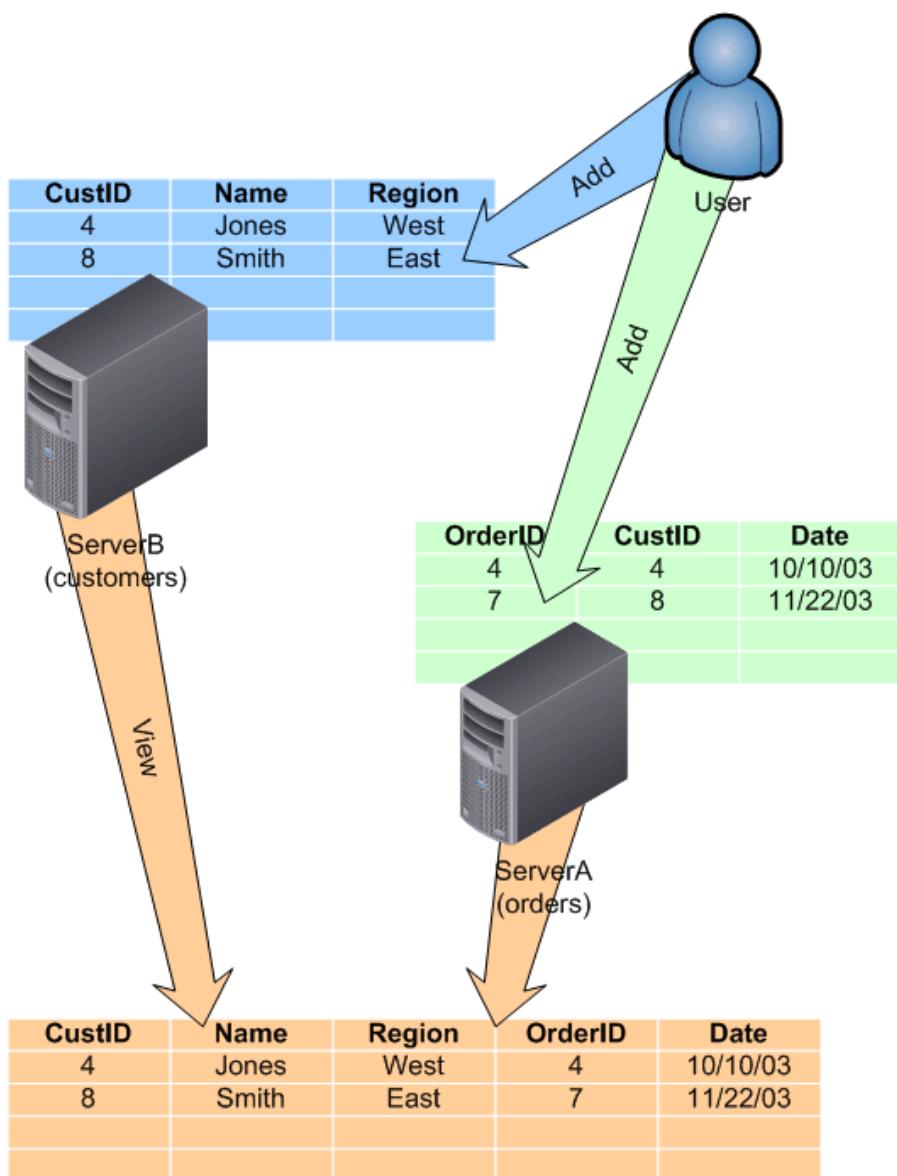


Figure 5.14: Using a view with a partitioned database.



Best Practices

Creating best practices for distributed and partitioned databases is difficult; every business situation has unique needs and challenges that make it difficult to create a single set of beneficial rules. However, there are certainly guidelines that have proven effective in a wide variety of situations. Don't consider these hard and fast rules—take them as a starting point for your designs:

- Reduce the number of subscribers that a publisher must deal with when it is also acting as a database server for users or database applications. If necessary, create a standalone distributor so that the publisher only needs to replicate data once (to the distributor), after which the distributor handles the brunt of the replication work to the subscribers.
- If latency is an issue, employ transactional or merge replication and create a fully enmeshed replication topology. If latency is not an issue—for example, a product catalog being distributed to read-only copies might only need to be replicated once a week—then use snapshot replication.
- As I've already mentioned, minimize the number of cross-server foreign key relationships and other cross-server object references when vertically partitioning a database. Cross-server references pass through SQL Server's Linked Servers functionality (which I described in Chapter 4) and can have a negative impact on overall performance if overused.
- Minimize the potential for data conflicts in replication so that you can use simpler transactional replication rather than the more complex merge replication. Horizontally partitioning tables so that each copy of the database "owns" particular rows can go a long way toward reducing data collisions (or conflicts) and can make transactional replication more viable in an environment with multiple writable copies of a database.
- Reduce the programming complexity of vertically partitioned databases by making use of views and stored procedures. These objects can abstract the underlying physical database structure so that software developers deal with a single set of objects (views and stored procedures) regardless of where the underlying data is actually situated.

Working with distributed or partitioned databases can be especially difficult for software developers, so make sure you include them in your initial scale-out design processes. They will need to understand what will need to change, if anything, in their client applications. In addition, perform basic benchmark testing to determine whether your proposed scale-out solution provides tangible performance benefits for your end users; how client applications function will play a major role in that performance. Including software developers in the planning and testing stages will help ensure more accurate results.



Benchmarks

Measuring the performance of a scale-out solution that uses distributed and/or partitioned databases can be complex because it is difficult to determine what to measure. For example, suppose you've created a distributed database like the one that Figure 5.3 illustrates. The purpose is to allow more Web servers to exist by having multiple copies of a database. All hardware being equal, a new database server should double the potential throughput of your Web site, because the new database server can support the same number of servers as the original database server. Similarly, if your existing Web farm can handle 10,000 users per hour with one back-end database and 10 Web servers, having two back-end database servers and 20 Web servers should provide the power for 20,000 users per hour.

 The main thing to measure is end-user response time because that metric is ultimately the sign of success or failure in any IT project.

This type of calculation becomes less straightforward when you move into more complex—and realistic—scenarios like the one that Figure 5.4 shows. In this case, the central Orders database server could serve as a performance bottleneck, preventing you from exactly doubling your site's overall user capacity.

You could also be using distributed databases in a scenario like the one I showed you in Figure 5.2, with multiple database servers housed in different physical locations. Again, hardware being equal, each database server should be able to handle an equal number of users. However, the actual performance gain from such a scenario can be greater than simply providing more power at the database tier. For example, suppose you start out with a single database server located in a central office, and field office users connect via WAN. And suppose that your database server is approaching its performance limits with several thousand company users connecting each day. Adding a server at your two major field offices would provide two performance benefits: the workload of the database application would be distributed across three servers (which will allow each server to maintain peak efficiency) and users will be accessing data across a LAN—rather than a WAN—which will create at least the perception of improved application performance. Figure 5.15 illustrates how network speed provides the performance gain.



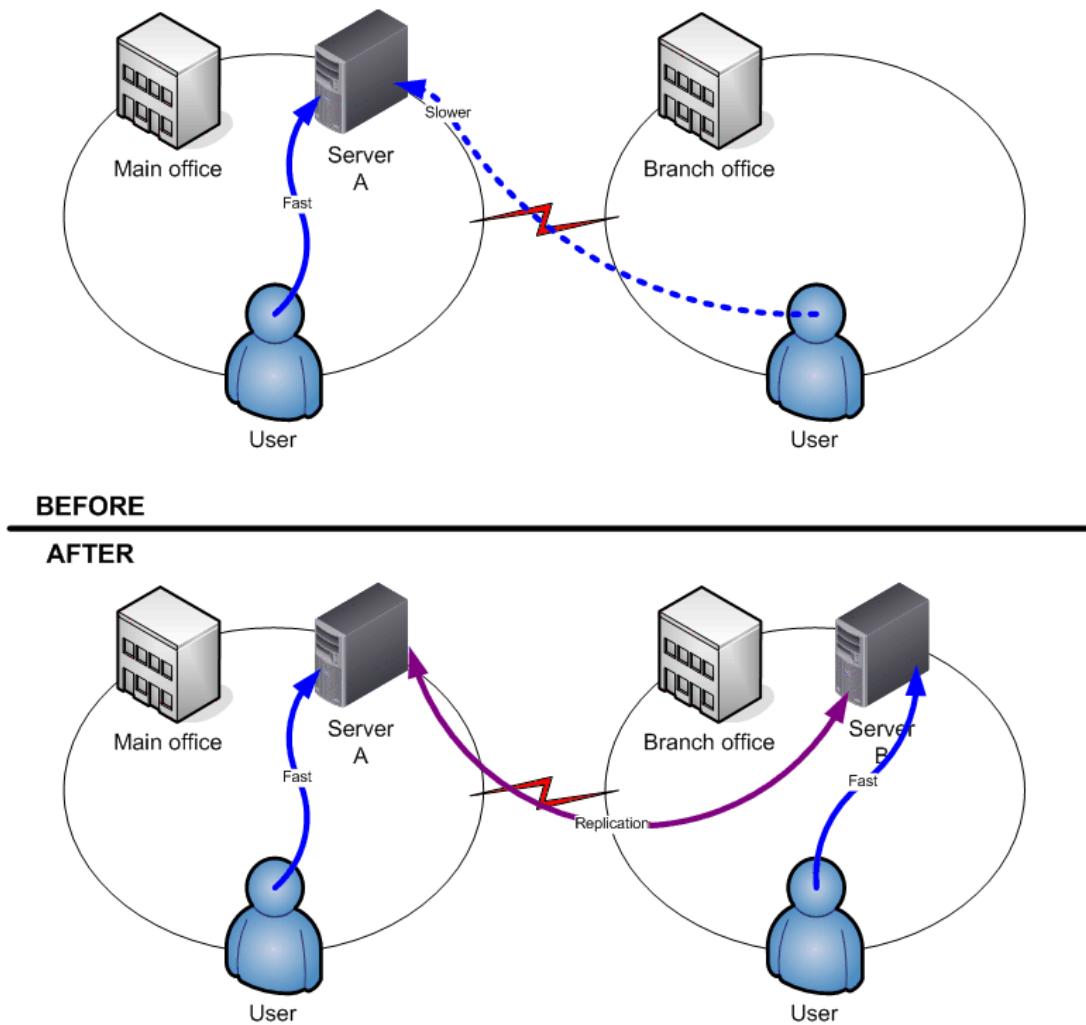


Figure 5.15: Local SQL Server computers have an impact on perceived performance.

To illustrate this concept with another example, suppose your original server, located at one of your company's two offices, can support all 5000 of your company users, which is far from the server's limit. Half of the users access the data across a WAN link. Now suppose you get another identical server and place it in your other office. Neither server will be working close to its capacity, but the second office will definitely see a performance benefit from the distributed database because they are now accessing data across the LAN instead of across the slower WAN link. The first office's users won't see any performance change at best; at worst, they might see a slight decrease in performance as a result of the additional load of replication (performance degradation is unlikely in this case; replication isn't *that* big of a burden in a scenario such as this). This setup illustrates how it can be difficult to measure the performance gains of a distributed database scale-out solution—there are several factors completely unrelated to SQL Server that can affect users' perception of performance.



Measuring the success of a vertically partitioned database can be even more difficult. It's nearly impossible to measure the performance each table contributes to an application's performance. For example, if you were to divide a database between two servers so that exactly half the tables were on each server, it's unlikely that you would double performance. The reason is that some tables are more heavily used than others. Additionally, a poorly designed partitioning scheme can *hurt* performance by forcing servers to rely too much on remote foreign key tables, which must be queried across the LAN.

The only accurate way to measure the performance benefits—or drawbacks—of a vertical partitioning scheme is to objectively measure the performance of the database application as a whole. In other words, construct metrics such as maximum number of users or average response time for specific user activities. By measuring these end user-based metrics, you will be able to account for all of the various factors that can affect performance, and arrive at an objective performance measurement for the application as a whole.

Summary

Distributing and partitioning databases are time-tested flexible ways to increase the performance of a database application. In fact, distributed partitioned views, which I discussed in the previous chapter, are an outgrowth and refinement of the database distribution and partitioning techniques I've discussed in this chapter. Distributing a database gives you the flexibility to place multiple copies of data in a single location and balance workload between the copies. Alternatively, you can distribute data across locations to provide faster access to different groups of users.

Partitioning—both horizontal and vertical—can also provide a performance gain, particularly for well-designed databases that offer logical divisions in either tables or rows.

It is not a straightforward task to predict performance gains from distributing and partitioning databases. It's difficult to fire off sample queries against a non-distributed copy of a database and compare the results to the performance of a distributed copy; the nature of distribution is to increase potential capacity, not necessarily to increase the performance of individual queries. When making performance comparisons, consider the total activity of an entire application to determine the effectiveness of your scale-out solution.

In the next chapter, I'll focus on Windows Clustering. Clustering is a common addition to scale-out solutions, as it prevents the single point of failure that a database server can represent. By clustering SQL Server computers, you can create a multiple-server scale-out solution that isn't vulnerable to the failure of a single piece of server hardware.



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/>.

