

realtimepublishers.com<sup>tm</sup>

*The Definitive Guide<sup>tm</sup> To*

# Scaling Out SQL Server 2005

*Don Jones*

---

Chapter 4: Distributed Partitioned Views .....	72
Pros and Cons .....	72
Distributed Partitioned View Basics .....	74
Distributed Partitioned View Details .....	75
Design and Implementation .....	81
Linked Servers .....	81
Partitioned Tables .....	87
The Distributed Partitioned View .....	88
Checking Your Results .....	88
Best Practices .....	89
Grouping Data.....	89
Infrastructure.....	90
Database Options .....	90
Queries and Table Design.....	90
Sample Benchmark Walkthrough.....	91
Sample Benchmark .....	91
Conducting a Benchmark.....	93
Summary .....	94

## Copyright Statement

© 2005 Realtimedpublishers.com, Inc. All rights reserved. This site contains materials that have been created, developed, or commissioned by, and published with the permission of, Realtimedpublishers.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 Realtimedpublishers.com, Inc or its web site sponsors. In no event shall Realtimedpublishers.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.

Realtimedpublishers.com and the Realtimedpublishers 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 Realtimedpublishers.com, please contact us via e-mail at [info@realtimedpublishers.com](mailto:info@realtimedpublishers.com).

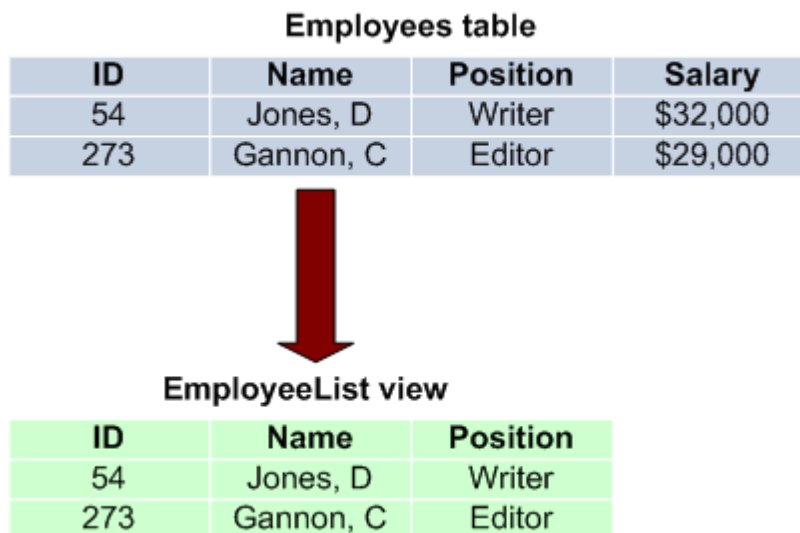
[**Editor's Note:** This eBook was downloaded from Content Central. To download other eBooks on this topic, please visit <http://www.realtimepublishers.com/contentcentral/>.]

## Chapter 4: Distributed Partitioned Views

As I've mentioned in previous chapters, distributed partitioned views can be a powerful tool in any scale-out scenario. However, they also have drawbacks, including the possibility of creating an imbalance in the amount of resources consumed by the servers that are handling the view. This chapter will go deeper into the pros and cons of distributed partitioned views, show you how to design and implement them, and offer some distributed partitioned view tips and best practices. In addition, I will provide direction for designing a performance comparison that allows you to measure the effectiveness and efficiency of distributed partitioned views in your environment.

### Pros and Cons

First, let's quickly review the pros and cons of a distributed partitioned view. A regular view is designed to act as a sort of virtual table. As Figure 4.1 shows, you can use views as a form of security mechanism, providing certain users with access to a subset of a table's columns.



*Figure 4.1: You can use a view to limit users' ability to see columns in a table.*

You can also use views to pull columns from multiple tables into a single virtual table. As Figure 4.2 shows, this type of view is most often created by using JOIN statements to link tables that have foreign key relationships. For example, you might create a view that lists a product's information along with the name of the product's vendor rather than the vendor ID number that is actually stored in the product table.

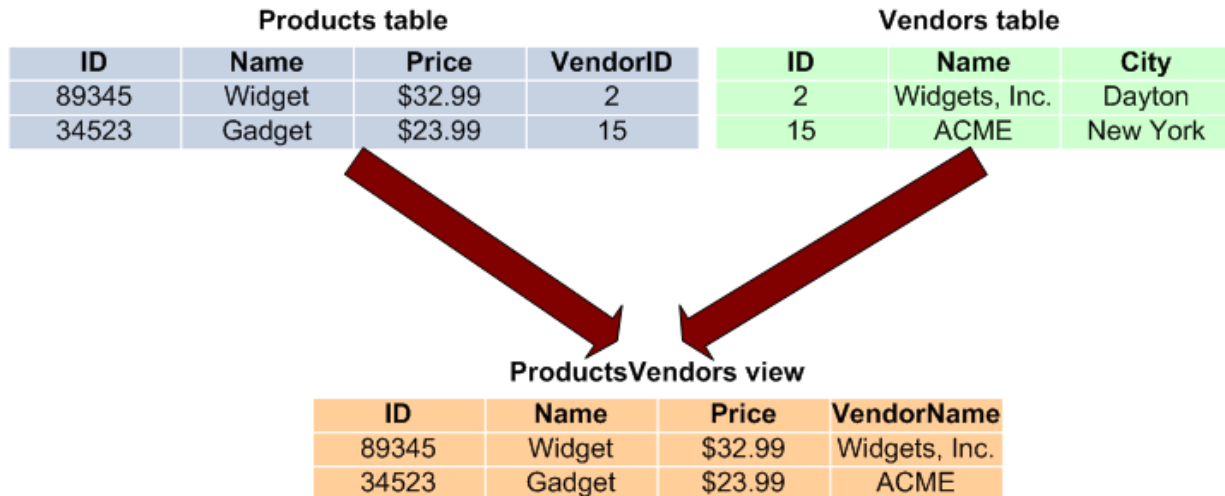


Figure 4.2: You can use a view to combine information from multiple tables.

Views can access tables that are located on multiple servers, as well. In Figure 4.3, a view is used to pull information from tables located on different servers. This example illustrates a sort of distributed view, although the view isn't really doing anything drastically different than a view that joins tables located on the same server.

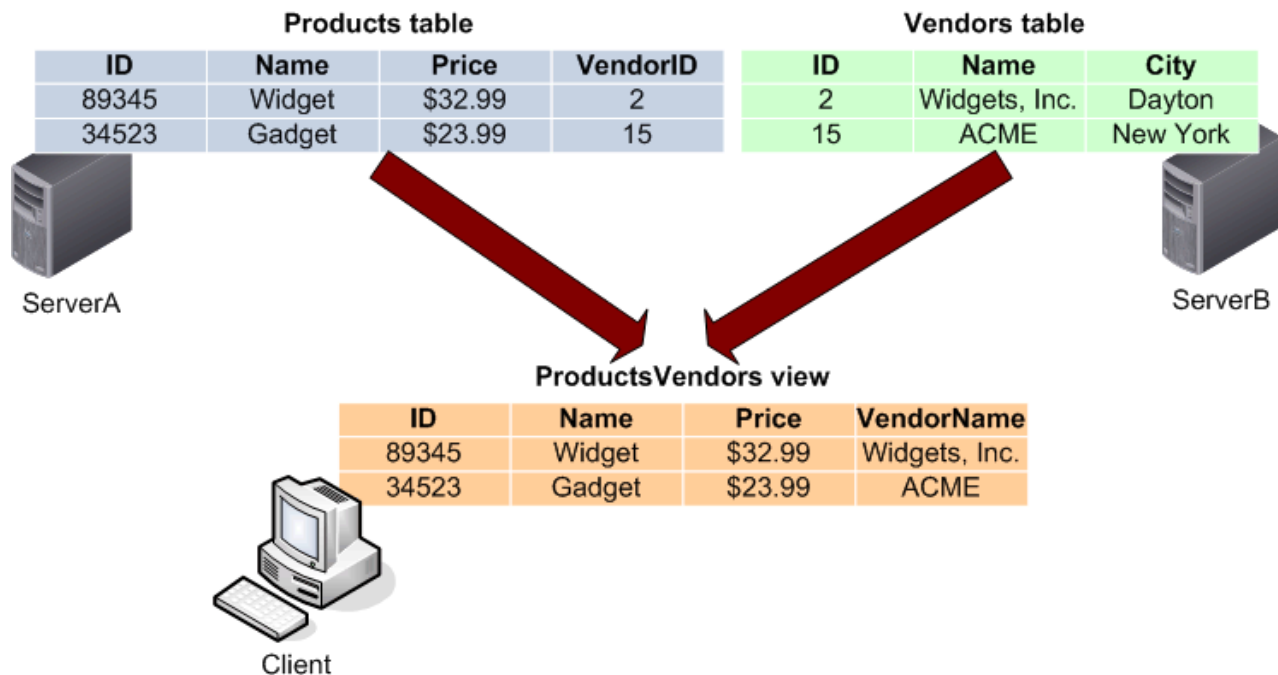
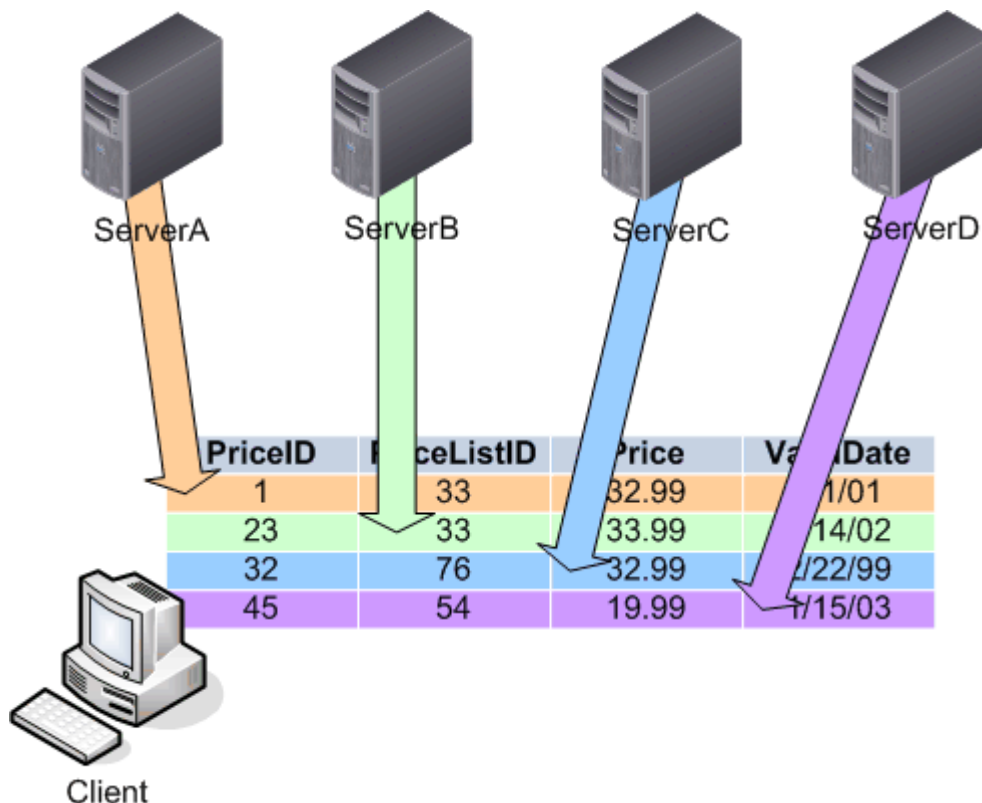


Figure 4.3: You can use a view to combine information from tables on different servers.

### Distributed Partitioned View Basics


A distributed partitioned view isn't much different than a regular view. What differentiates a distributed partitioned view is actually the nature of the underlying database. In a distributed partitioned view, you typically have two or more servers, each with tables that have the same structure, or *schema*. For example, suppose two servers have identical tables for storing products. Each server actually stores different rows within its tables. For example, one server might store all of the products with names that begin with A through M, while another stores products beginning with N through Z. The distributed partitioned view makes those two tables appear as one virtual table by employing both servers to return the results of a query. Figure 4.4 shows how a distributed partitioned view works.



**Figure 4.4:** Using a distributed partitioned view to combine information from horizontally partitioned tables on different servers.

To illustrate the power behind a distributed partitioned view, consider the following example. Suppose you have a table with several billion rows (making the table in the terabyte range), and you need to frequently execute a query that returns just 5000 rows, based (hopefully) on some criteria in an indexed column. One server will expend a specific amount of effort and will require a specific amount of time to gather those rows.

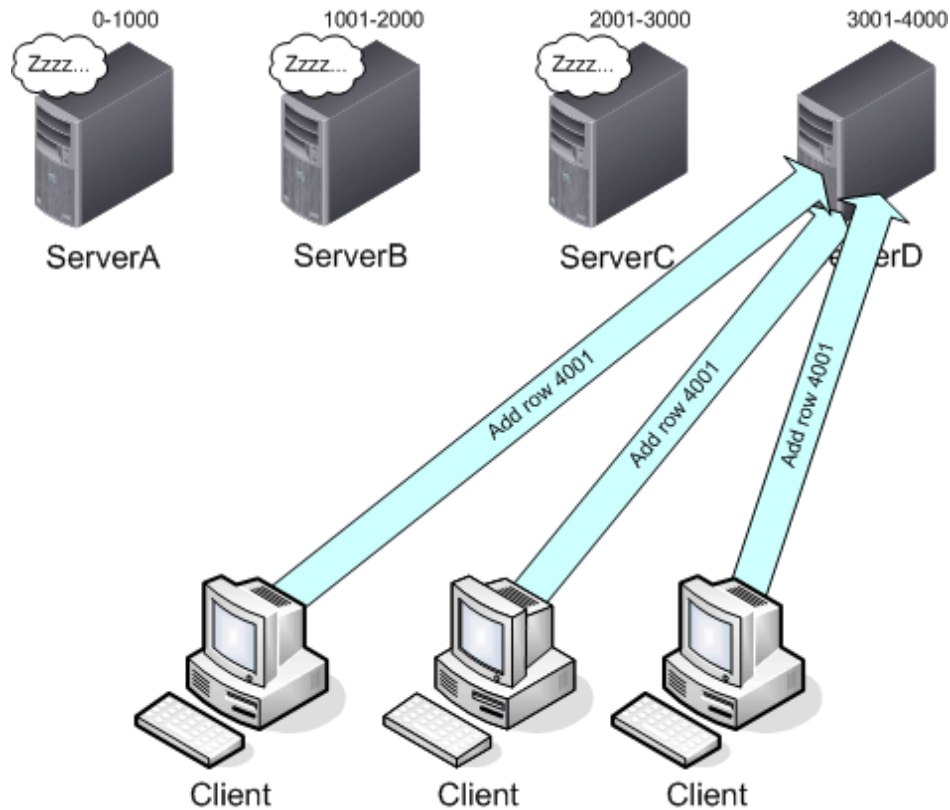
In theory, two servers—each with only half of the rows total and only a few to return—could complete the query in half the time that one server would require to return the same number of rows. Again in theory, four servers could return the results in one-quarter the time. The distributed partitioned view provides “single point of contact” to a back-end, load-balanced cluster of database servers. The distributed partitioned view is implemented on each of the servers, allowing clients to connect to any one server to access the distributed partitioned view; the distributed partitioned view makes it appear as if all the rows are contained on that server, when in fact the distributed partitioned view is coordinating an effort between all the back-end servers to assemble the requested rows.

 Microsoft uses the term *shared-nothing cluster* to refer to the group of back-end servers that work together to fulfill a distributed partitioned view query. The term describes the fact that the servers are all required in order to complete the query but that they are not interconnected (other than by a network) and share no hardware resources. This term differentiates distributed partitioned views and Microsoft Cluster Service clusters, which typically share one or more external storage devices. Shared-nothing clusters are also different than the clusters created by Network Load Balancing (NLB); those clusters contain servers that have a complete, identical copy of data (usually Web pages) and independently service user requests without working together.

What makes distributed partitioned views in SQL Server truly useful is that the views can be updated. The distributed partitioned view accepts the updates and distributes INSERT, DELETE, and UPDATE statements across the back-end servers as necessary, meaning that the distributed partitioned view is keeping track—under the hood—of which servers contain which rows. The distributed partitioned view truly becomes a virtual table, not just a read-only representation of the data.

### ***Distributed Partitioned View Details***

Very few technologies work as well in reality as they do in theory, and distributed partitioned views are no exception. Proper design of a distributed partitioned view is absolutely critical to ensuring a performance gain, and proper design requires an almost uncanny understanding of how your data is used. For example, suppose you create a four-server cluster (typically called a *federation*) to store a products table. You decide to partition the table by product ID so that each server contains one-fourth the total number of products and each server contains a single sequential block of product IDs. In other words, one server contains ID numbers zero through 1000, server number two stores 1001 through 2000, and so forth. Now suppose that this table is primarily used by individuals who are adding new rows and that new rows are always added to the end of the range. Thus, the fourth server in the federation—the one originally containing rows 3001 through 4000—is doing most of the work. The other three servers will more or less sit around looking alert but not doing much. Figure 4.5 illustrates the problem.



**Figure 4.5: Unbalanced updates place an uneven load on a single server in the federation.**

#### Finding the Data

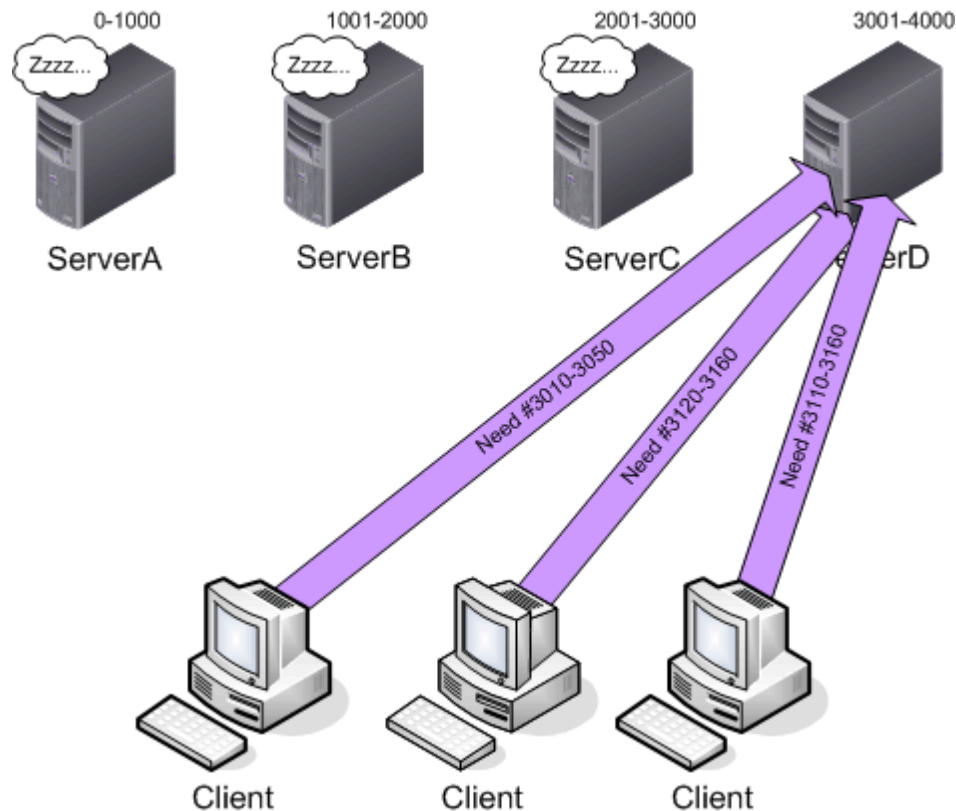
How does the distributed partitioned view know where to go for the data? The answer depends on the way the tables were created. The tables on each server in the federation aren't identical. Although the majority of the tables' schema is identical, each server has a slightly different CHECK constraint on the column that is used to partition the data. For example, a CHECK constraint might be used on a vendor name column so that ServerA could only contain values between "AAAAA" and "NZZZZ" and ServerB could only contain "OOOOO" through "ZZZZZ."

When SQL Server executes the distributed partitioned view, SQL Server analyzes this schema information—which can be queried very quickly from the federations' servers—to determine which servers need to participate in the query. This technique keeps servers that have no rows to contribute to the query from being brought into the query. This decision is made by whichever server actually receives the query against the distributed partitioned view from a client; that server's query processor does the preliminary work and makes the execution decisions, a process covered in more detail later in this chapter.


Thus, you need to create a column and CHECK constraint so that each member of the federation contains a unique and non-overlapping range of rows. If you decide to split up your data geographically, for example, you might assign regional codes to each office and include a Region column with a CHECK constraint that, on each federation member, specifies that member's region as the only valid value for rows on that server.



How the data is queried also affects performance. Suppose the users of your products table primarily query products in sequential blocks of one hundred. Thus, any given query is likely to be filled by just one, or possibly two, servers. Now suppose that your users tend to query for newer products—with ID numbers of 3000 and higher—a lot more often than older products. Again, that fourth server will handle most of the work, as Figure 4.6 shows.



**Figure 4.6: Unbalanced partitioning places an uneven load on one server in the federation.**

 Are the three uninvolved servers actually doing *nothing*? The answer depends on where the query is executed. If the clients in Figure 4.6 are submitting their queries to the copy of the distributed partitioned view on ServerD, then, yes, the other three servers won't even know that a query is happening. The reason is that the distributed partitioned view on ServerD knows that only ServerD is necessary to complete the query.

However, had the clients submitted their query to a copy of the distributed partitioned view on ServerC, for example, ServerC would execute the query, submit a remote query to ServerD, then provide the results to the clients. ServerC would be doing a lot of waiting while ServerD pulled together the results and sent them over, so it might be more efficient for the client application to have some knowledge of which rows are located where so that the application could submit the query directly to the server that physically contained the rows.

Creating a high-performance federation requires a distribution of data that might not be immediately obvious. Logical partitions—such as by name, ID number, or some other attribute—might not be the most appropriate. You want to have the most frequently accessed data spread out as much as possible to take advantage of the federation's full processing power.

### Partitions and SQL Server 2005

I want to take a moment to address a potential terminology problem with the word *partition*. This chapter speaks about partitioning in a general sense. These partitions are ones that you create yourself, not ones that SQL Server has any particular recognition for. For example, if you horizontally partition a table so that all customers with an ID of 1-100000 are on ServerA, and all customers with an ID of 100001-200000 are on ServerB, you've created two tables with identical schemas and different data. SQL Server doesn't necessarily draw any connections between those tables; you're the only one thinking of them as partitions of the same overall logical table.

SQL Server 2005 introduces a new, built-in table partitioning feature that allows you to create partitioned tables (and indexes, for that matter) and manage them as a single object. In effect, SQL Server 2005 allows you to partition tables just like SQL Server 2000 did, but SQL Server 2005 maintains a link between the tables, treating them as a single, logical unit for you, thereby making management easier. So, for example, you could run DBCC commands against the table and all partitions would automatically be included; in prior versions of SQL Server, "partitions" were really just independent tables that you had to manage independently. SQL Server 2005's partitioned tables can still use distributed partitioned views for querying; the primary purpose of the partitioning features are to make management of the partitions easier by treating them as a single unit.

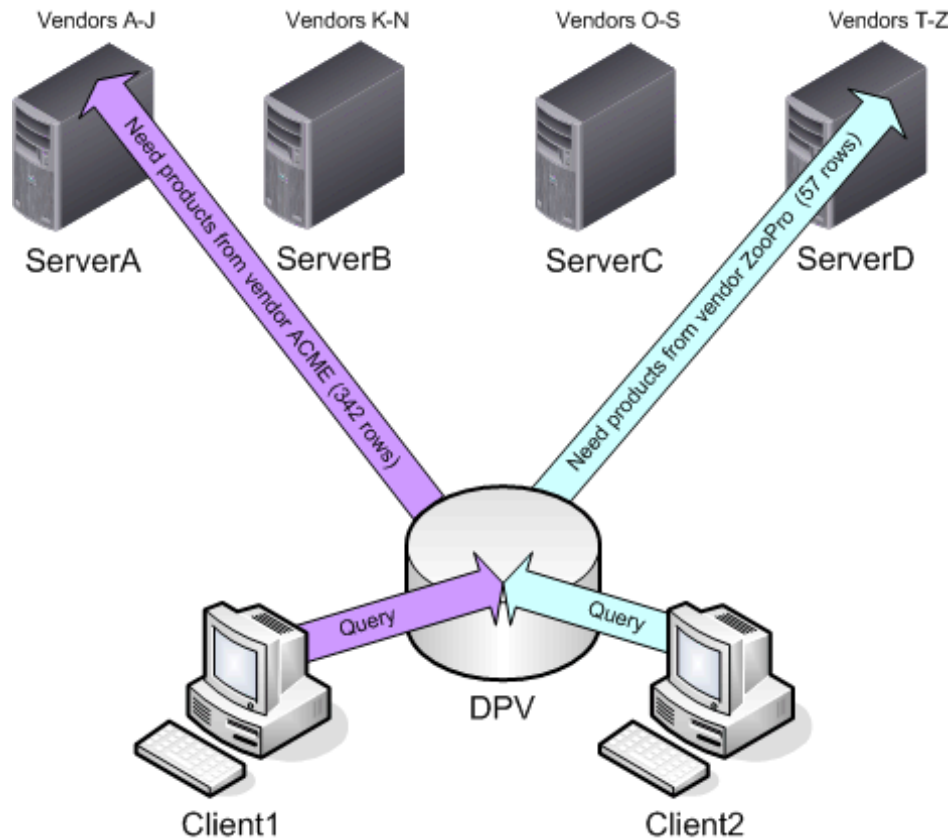
The catch is that SQL Server 2005's partitioned tables aren't designed for use in a scale-out scenario. The reason is that the partitions of a table must all live on the same server (although they can obviously live in different files, allowing them to be distributed across storage devices). SQL Server 2005 partitioned tables are suitable in a scale-up scenario, because they allow tables to be broken up across files and managed as a single unit, but they don't lend any capability to a scale-out scenario, which is why they haven't been specifically covered elsewhere in this chapter.

Don't be tempted to look at the task of partitioning from a single-query viewpoint. To design a properly partitioned federation, you want the workload of all your users' queries to be distributed across the federations' servers as much as possible. However, *individual* queries that can be satisfied from a single server will tend to execute more quickly.




Obviously, the ultimate form of success for any database project is reduced end-user response times. Balancing the load across a federation should generally help improve response times, but that improved response time is definitely the metric to measure more so than arbitrary load distribution.

The following example illustrates this point. Suppose your users typically query all of the products made by a particular vendor. There is no concentration on a single vendor; users tend to query each vendor's products several times each day. In this case, partitioning the table by vendor makes sense. Each single query will usually be serviced by a single server, reducing the amount of cooperation the servers must handle. Over the course of the day, all four servers will be queried equally, thus distributing the *overall* workload across the entire federation. Figure 4.7 illustrates how two queries might work in this situation.



**Figure 4.7:** Each query is fulfilled by an individual server, but the overall workload is distributed.

 For the very best performance, queries would be sent to the server that physically contains the majority of the rows to be queried. This partially bypasses some of the convenience of a distributed partitioned view but provides better performance by minimizing inter-federation communications and network traffic. However, implementing this type of intelligently directed query generally requires some specific logic to be built-in to client or middle-tier applications. Keep in mind that any server in the federation can fulfill distributed partitioned view queries; the performance benefit is recognized when the server that is queried actually contains the majority of the rows needed to respond.

One way to implement this kind of intelligence is to simply provide a lookup table that client applications can query when they start. This table could provide a list of CHECK values and the associated federation members, allowing client applications to intelligently submit queries—when possible—to the server associated with the desired range of values. Although not possible for every query and every situation, this configuration is worth considering as you determine which column to use for partitioning your data.

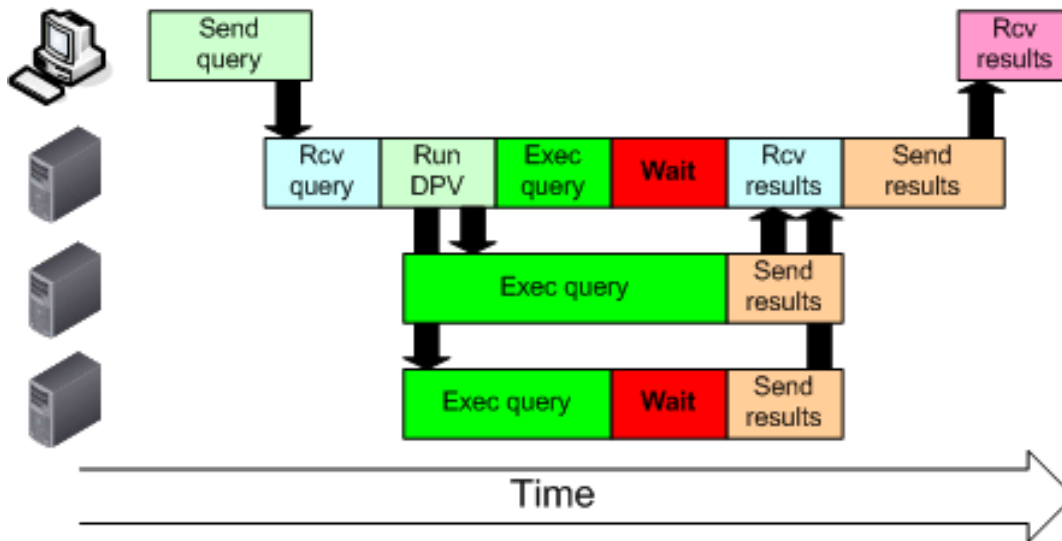
### A Great Case for Distributed Partitioned Views

One of the best examples of when to use a distributed partitioned view is a large, geographically distributed sales organization. Sales, customer information, and other data can be tagged—by using a column for this purpose—to identify the specific office to which the data belongs. Each major office would have a SQL Server computer that would hold a version of the database containing all the records that “belong” to that office. All the offices’ servers would be linked together in a federation, and a distributed partitioned view would be used to access the data.

Users in each office would primarily access their own data, meaning their queries could be resolved by their local SQL Server computer. However, queries involving other offices’ data would still be possible thanks to the distributed partitioned view (although the probably slower WAN speeds would definitely create a performance bottleneck, especially if cross-office queries became common). Although each office wouldn’t be spreading their queries across the federation, the company’s aggregate querying would be balanced across the federation.


How often users will need to query data from other offices is an important factor. If the need arises frequently, a distributed partitioned view might not be the best approach; instead, you might simply replicate data from each office to all the other offices, minimizing cross-WAN queries. However, if cross-office querying is less frequent and involves less data than replication would require, a distributed partitioned view is a useful solution. As always, exactly how your data is used is the most important factor in selecting a scale-out solution.

An uneven distribution of rows can cause more subtle problems. Using the four-server products table as an example, suppose that one of the servers hasn’t been maintained as well as the other three—its database files are heavily fragmented, indexes have a large number of split pages, and so forth. That server will typically respond to any given query somewhat more slowly than the other, better-maintained servers in the federation. When a distributed partitioned view is executed, the other three servers will then be forced to hold their completed query results in memory—a very expensive concept for a database server—until the lagging server catches up. Only when all the servers have prepared their results can the final distributed partitioned view response be assembled and provided to the client. This type of imbalance—especially if it occurs on a regular basis—can cause significant performance degradation of an application. Figure 4.8 shows a timeline for a distributed partitioned view execution, and how a lagging server can hold up the entire process.



**Figure 4.8:** A slow server in the federation causes unnecessary wait periods in distributed partitioned view execution.

A lagging server is not always caused by maintenance issues—a server with significantly less powerful hardware or one that must consistently perform a larger shard of query processing than the others in a federation might hold up the query. The key to eliminating this type of delay is, again, proper distribution of the data across the federation members.

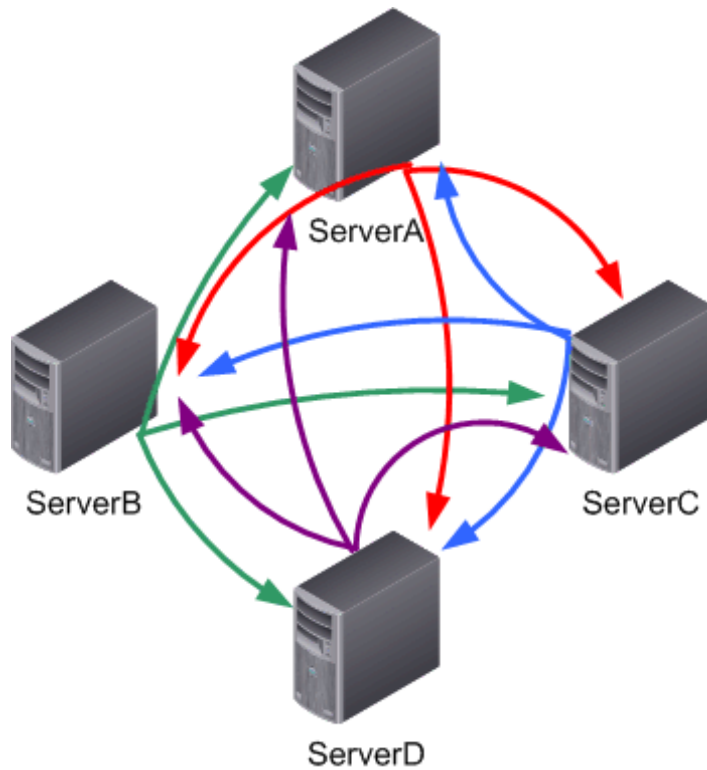
 This example reinforces the idea that step one in a scale-out strategy is to ensure that your servers are properly tuned. For more information, see Chapter 2.

## Design and Implementation

At this point, it is clear that proper planning is absolutely critical for a successful distributed partitioned view implementation. Once you have figured out the perfect balance for your partitions and are ready to begin implementing a scale-out environment that uses distributed partitioned views, how do you begin?

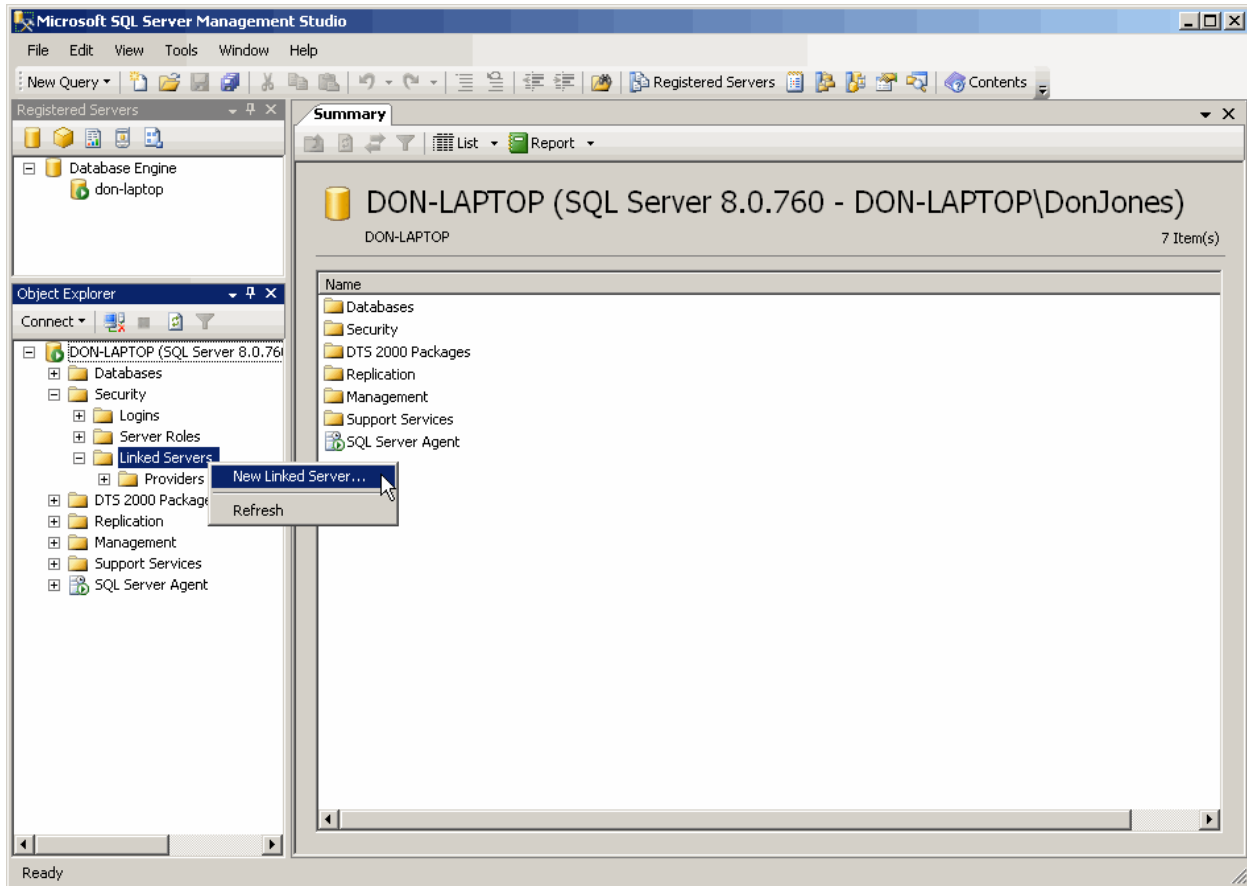
### Linked Servers

A distributed partitioned view requires multiple servers to communicate, so you need to provide the servers with some means of communication. To do so, you use SQL Server's *linked servers* feature, which provides authentication and connection information to remote servers. Each server in the federation must list all other federation members as a linked server. Figure 4.9 shows how each of four federation members have pointers to three partners.



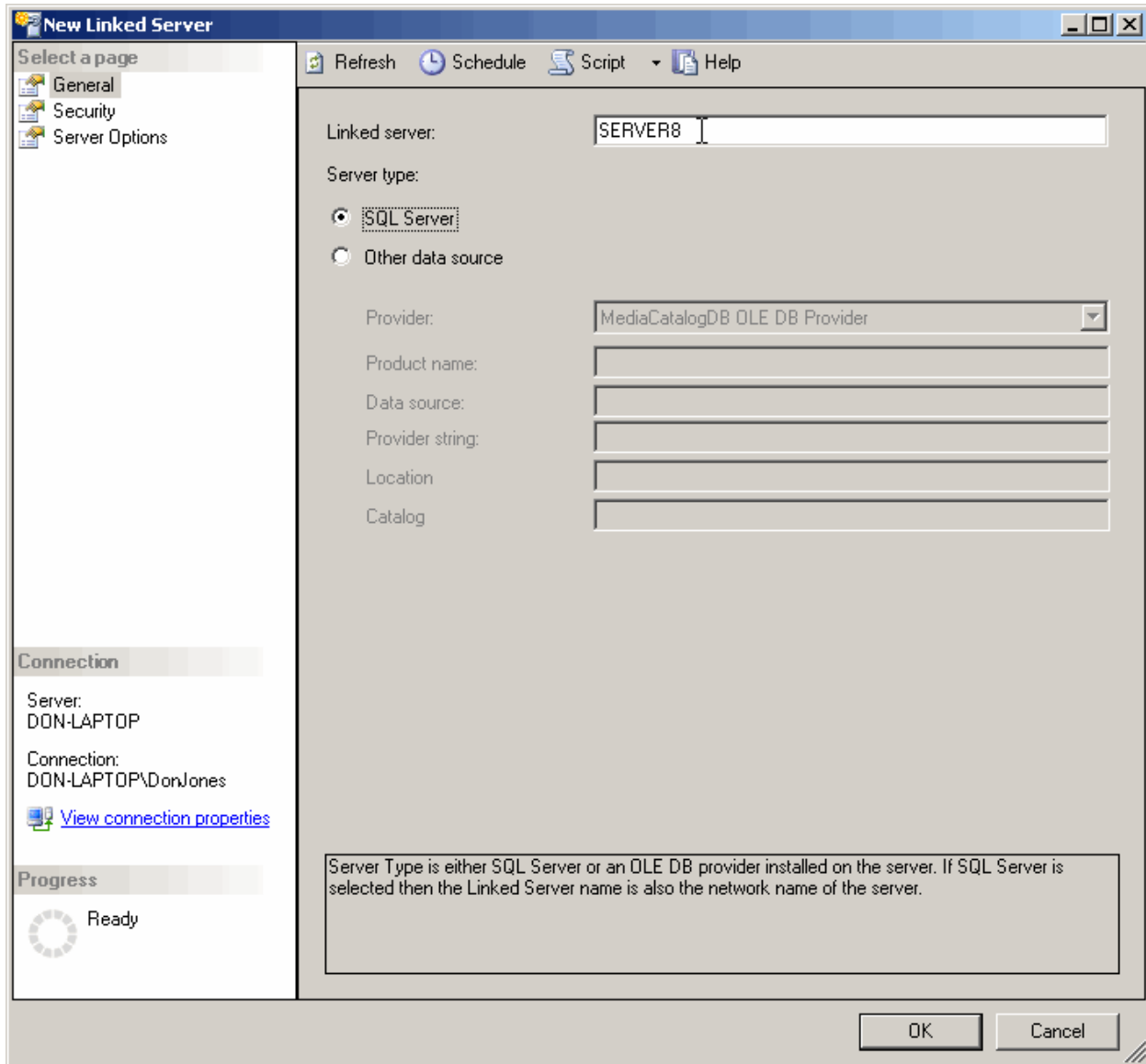
**Figure 4.9: Setting up linked servers in a federation.**

To begin, open SQL Server Management Studio. In the Object Explorer, right-click Linked Servers, and select New Linked Server, as Figure 4.10 shows.



**Figure 4.10: Setting up linked servers in SQL Server Management Studio.**

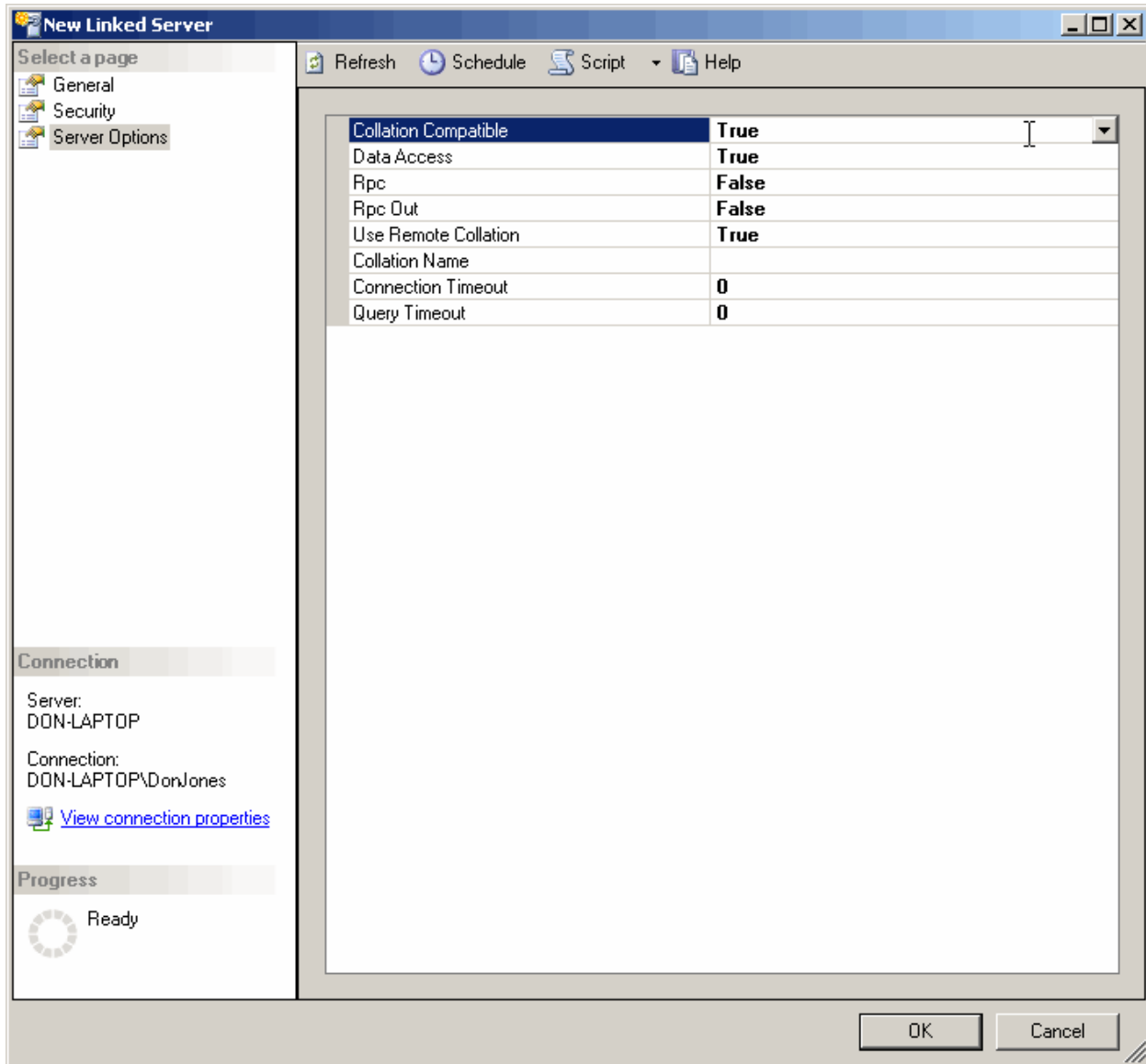
In a dialog box similar to that shown in Figure 4.11, type the name of the server to which you this server linked, and indicate that it is a SQL Server computer. SQL Server provides built-in linking functionality, so most of the rest of the dialog box isn't necessary.



**Figure 4.11: Specifying the linked server name and type.**

You should also specify the collation compatible option (which I describe later under Best Practices). Doing so will help improve performance between the linked servers, which you should set to use the same collation and character set options. Figure 4.12 shows how to set the option.





**Figure 4.12:** Setting collation options for the link.

Finally, as Figure 4.13 shows, you can set various security options. You can use these settings to specify pass-through authentication and other options so that logins on the local server can map to logins on the linked server. Ideally, set up each server to have the same logins—especially if you’re using Windows Integrated authentication; doing so would make this tab unnecessary.

👉 You’ll make your life significantly easier by maintaining consistent logins across the members of the federation. I prefer to use Windows Integrated authentication so that I can add domain groups as SQL Server logins, then manage the membership of those groups at the domain level. By creating task- or role-specific domain groups, you can add the groups to each federation member and ensure consistent authentication and security across all your SQL Server computers.

Be sure to set a complex password for SQL Server's built-in sa login even if you're setting SQL Server to use Windows Integrated authentication. That way, if SQL Server is accidentally switched back into Mixed Mode authentication, you won't have a weak sa login account as a security hole.

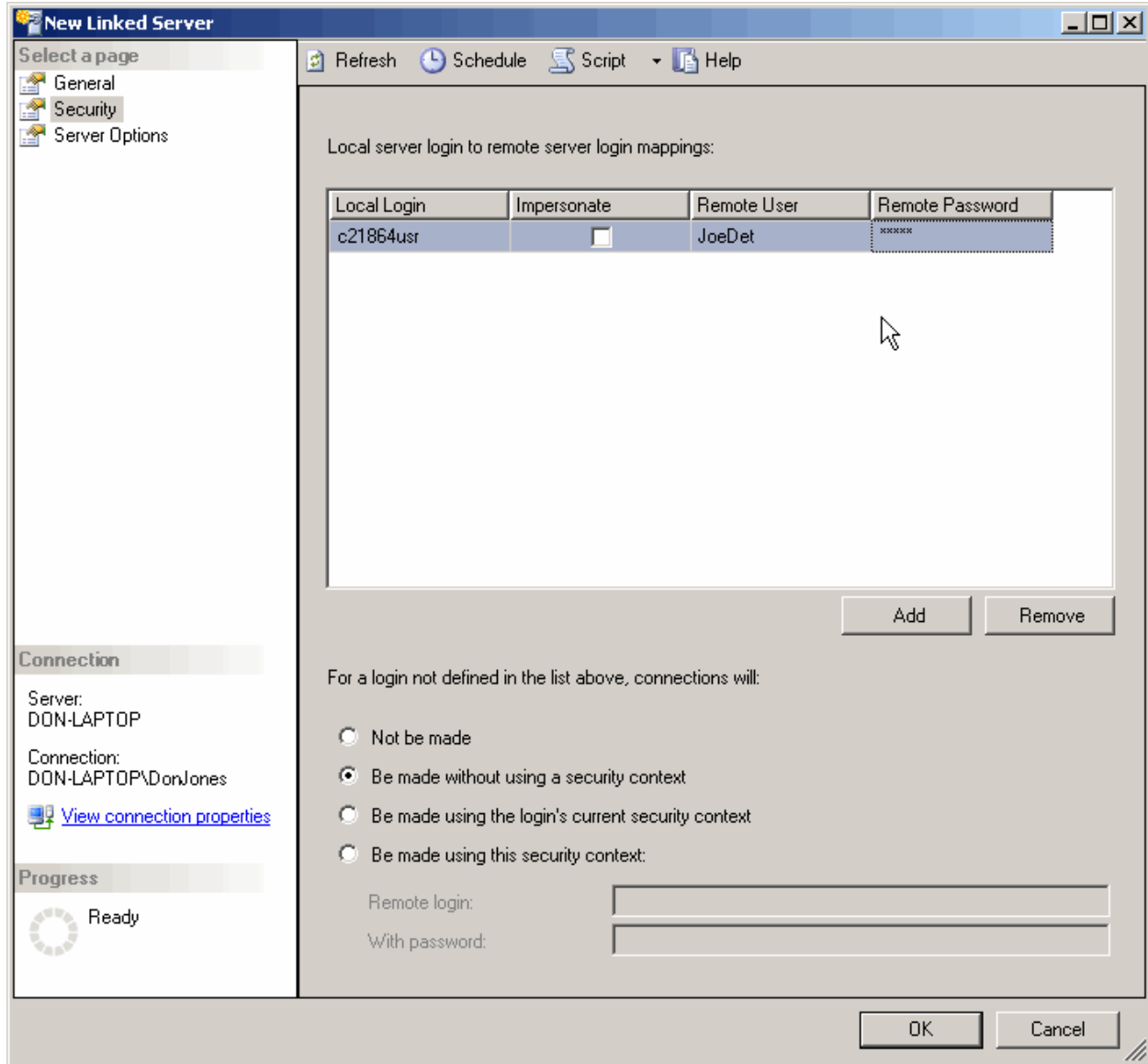


Figure 4.13: Specifying security options for the link.

You can also set up linked servers by using the `sp_addlinkedserver` stored procedure. For more information on its syntax, consult the Microsoft SQL Server Books Online.

Once all of your links are established, you can move on to creating the partitioned tables.

## Partitioned Tables

Partitioned tables start out just like any other table, but they must contain a special column that will be the *partitioning column*. SQL Server will look at this column to see which of the federation's servers contain (or should contain, in the case of added rows) specific rows. The partitioning column can be any normal column with a specific CHECK constraint applied. This CHECK constraint must be different on each member of the federation so that each member has a unique, non-overlapping range of valid values for the column.

A UNION statement is used to combine the tables into an updateable view—the distributed partitioned view. Keep in mind that SQL Server can only create updateable views from a UNION statement under certain circumstances; in order to create a distributed partitioned view on SQL Server 2000, you'll have to adhere to the following rules:

- The partitioning column must be part of the primary key for the table, although the primary key can be a composite key that includes multiple columns. The partitioning column must not allow NULL values, and it cannot be a computed column.
- The CHECK constraint on the column can only use the BETWEEN, OR, AND, <, <=, =, >, and >= comparison operators.
- The table that you're partitioning cannot have an identity or timestamp column, and none of the columns in the table can have a DEFAULT constraint applied.

Here is an example CHECK constraint that you might include in a CREATE TABLE statement:

```
CONSTRAINT CHK_VendorA_M CHECK (VendorName BETWEEN 'AAAAA' AND 'MZZZZ')
```

Again, this constraint must exist in the table in each member of the federation, although each member must supply a different, non-overlapping range of values.

### The Distributed Partitioned View

The last step is to create the distributed partitioned view. You'll do so on each server in the federation so that clients can connect to any one server and successfully query the distributed partitioned view. Assuming you're starting on ServerA and that the federation also contains ServerB, ServerC, and ServerD, you would use the following commands on ServerA:

```
CREATE VIEW VendorDataDPV_A
AS SELECT * FROM VendorData_A
UNION ALL
SELECT * FROM ServerB.TestDB.dbo.VendorData_B
UNION ALL
SELECT * FROM ServerC.TestDB.dbo.VendorData_C
UNION ALL
SELECT * FROM ServerD.TestDB.dbo.VendorData_D
GO
```

☞ It is important to use UNION ALL rather than specifying some additional criteria because you want all of the data in each table to be included in the distributed partitioned view.

Each server's version of the distributed partitioned view will be slightly different because each server will start with the local copy of the table, then link to the other three (or however many) members of the federation.

☞ There is no requirement that the tables on each federation member have different names; SQL Server uses the complete server.database.owner.object naming convention to distinguish between them. However, from a human-readable point of view, coming up with a suffix or some other indicator of which server the table is on will help tremendously when you're creating your distributed partitioned views and maintaining those tables.

### Checking Your Results

The distributed partitioned view can then be queried just like any other table, using the following command, for example:

```
SELECT * FROM VendorDataDPV
```

It is interesting to look at the query execution plan that SQL Server generates. You can view the plan in SQL Query Analyzer, keeping in mind that the plan will only exist on the server you connect to in order to query the distributed partitioned view. You'll see SQL Server issue a query against the local computer and issue parallel queries to each of the federation members. The cost of each query should be pretty much evenly divided for a SELECT \* query, because each server is returning all of its rows and those rows are, hopefully, evenly spread across the servers.

The results of the query and remote query operations will feed to a concatenation operation, then a SELECT operation, both of which will show a cost of zero. These operations are simply the local server accepting and combining the data from the federation's query responses.

Next, try a slightly more complex query:

```
SELECT * FROM VendorDataDPV WHERE VendorName = 'ACMEA'
```

This query will return a more interesting query execution plan. In fact, the graphical plan can be a bit misleading. You will see something similar to the first, simpler `SELECT *` query, but with the addition of a yellow filter icon filtering the results of those queries prior to feeding the data to a concatenation operation. The graphical plan makes it appear as if all servers in the federation executed a remote query, but such is not the case. You're querying against the partitioning column, so SQL Server's query processor should have been able to determine—by examining `CHECK` constraints—which server contained the requested data.

Check out the text version of the execution plan for more details, and you will see that SQL Server only executed the query against the server that contained the desired rows. Because the distributed partitioned view is bound to the underlying tables' schema, SQL Server can easily determine which of the servers is allowed to contain the requested data (the column queried is the partitioning column; had another column been used in the `WHERE` clause, SQL Server would have had to submit the query to each member of the federation). This illustrates how distributed partitioned views can provide a performance benefit, especially when queries will use the partitioning column as selection criteria.



Clients will realize better performance if they query directly against the partitioning column because SQL Server can make an immediate and certain determination as to which server will handle the query. In the previous example, only one server could possibly contain vendors with the name ACMEA because that is the column that was used to partition the table and the servers must have non-overlapping ranges.

## Best Practices

Designing databases and distributed partitioned views can be a complex task, often filled with contradictory goals, such as improving performance and reducing database size (which are rarely completely compatible). To help you create the best design for your situation, and to configure your SQL Server computers to execute distributed partitioned views as quickly as possible, consider the following best practices.

### Grouping Data

It is not enough to simply partition your primary tables across the servers in the federation. Ideally, each server should also contain a complete copy of any lookup tables to enable each server to more readily complete queries on its own. You'll likely need to make some design decisions about which lookup tables are treated this way. For example, tables with a small number of rows or ones that aren't updated frequently are good candidates to be copied to each server in the federation. Tables that change frequently, however, or that contain a lot of rows, may need to be partitioned themselves. The ideal situation is to horizontally partition the primary table that contains the data your users query most and to include complete copies of all supporting tables (those with which the primary table has a foreign key relationship). Not every situation can meet that ideal, but, in general, it is a good design strategy. The fewer external servers any particular server has to contact in order to complete its share of a distributed partitioned view query, the better the queries' performance.

## **Infrastructure**

SQL Server computers that are part of a federation must have the highest possible network connectivity between one another. Gigabit Ethernet (GbE) is an inexpensive and readily available technology that provides today's fastest LAN connectivity speeds and should be part of the base configuration for any server in a federation. Use fast, efficient network adapters that place as little processing overload on the servers' CPUs as possible. In the future, look for network adapters that implement TCP/IP offload engines (TOE) to minimize CPU impact.

As I've already mentioned, try to keep servers in a federation as equal as possible in terms of hardware. Having one server with a memory bottleneck or with slower hard drives makes it more difficult for the federation to cooperate efficiently.

Also consider connecting the servers in a federation via a storage area network. SANs provide the best speed for storage operations and can eliminate some of the bottlenecks often associated with the heavy-load data operations in a federation.

## **Database Options**

Ensure that each server participating in a federation has the same collation and character set options. Then set the server option `collation compatible` to true, telling SQL Server to assume compatible collation order. Doing so allows SQL Server to send comparisons on character columns to the data provider rather than performing a conversion locally. To set this option, run the following command using SQL Query Analyzer:

```
sp_serveroption 'server_name', 'collation compatible', true
```

Another option you can set is lazy schema validation. This option helps improve performance by telling SQL Server's query processor not to request metadata for linked tables until the data is actually needed from the remote server. That way, if data isn't required to complete the query, the metadata isn't retrieved. Simply run the following command to turn on the option:

```
sp_serveroption 'server_name', 'lazy schema validation', true
```

Of course, you'll need to set both of these options for each server in the federation.

## **Queries and Table Design**

To help improve the efficiency of distributed partitioned views, avoid queries that contain data conversion functions. Also avoid using SELECT statements that utilize the TOP *n* or TOP *n*% clauses. Eliminating these items from your queries makes the queries easier for a distributed partitioned view to process.

For example, using a TOP clause forces each server in the federation to return the complete results of the TOP clause just in case that server is the only one with any results. The server actually executing the distributed partitioned view must then combine the federation's results, sort them, then truncate them to provide the requested TOP *n* or TOP *n*% rows. It's an inefficient operation.

Also avoid using the bit, timestamp, and uniqueidentifier data types in tables that sit behind a distributed partitioned view; distributed partitioned views deal less effectively with these data types than with others. Also, when using binary large object (*blob*) data types—such as text, ntext, or image—be aware that SQL Server can incur a significant amount of additional processing simply to transmit the large amount of data between the servers in the federation. Although these object types aren't forbidden in a distributed partitioned view, they certainly won't provide the best possible performance for queries.

## Sample Benchmark Walkthrough

It is critical that you pilot and benchmark your database in order to ensure that distributed partitioned views are giving you a performance benefit. In addition, you will need to use multiple physical servers (not a shortcut such as virtual machines) in order to see real performance numbers.

### Design vs. Test

When you're designing distributed partitioned views and their associated databases, you can take shortcuts. For example, keep in mind that multiple instances of SQL Server can run on a single computer. The first instance is referred to simply by using the computer's name, such as ServerA. Subsequent instances are given unique names that are appended to the computer name (ServerA\$InstanceTwo, ServerA\$InstanceThree, and so forth). Each instance of SQL Server is essentially the same as a separate physical server, at least from a logical point of view.

Using multiple instances allows you to test various partitioning schemes along with your distributed partitioned view designs to ensure functionality. In other words, you can perform tests to ensure that a DTS package can successfully transfer existing data into the new, partitioned tables; perform tests to ensure that the distributed partitioned view itself returns the expected results; and perform tests with client applications to ensure they are able to successfully query—and, if necessary, post updates to—the distributed partitioned view.

Obviously, separate physical computers would have far more computing power than multiple instances of SQL Server sharing a single processor; for performance measurements, you need the real thing, and working from multiple instances on one machine will provide inaccurate performance results.

### Sample Benchmark

I conducted an informal benchmark of a database on both a single server and a 2-node federation using distributed partitioned views. My database was fairly small at a mere ten million rows. For hardware, I utilized identical Dell server computers, each with a single Pentium 4 processor and 1GB of RAM. Obviously not high-end server hardware, but it is the relative difference between single-server and distributed partitioned view performance that I wanted to examine, not the overall absolute performance values. In fact, using this hardware—as opposed to newer 64-bit hardware—makes the performance differences a bit easier to measure. I created a simple table structure within the database using the SQL statements included in Listing 4.1.

```

CREATE TABLE [dbo].[VendorData] (
[VendorID] [numeric](19, 0) NOT NULL ,
[VendorName] [varchar] (40) COLLATE SQL_Latin1_General_CP1_CI_AS NOT
NULL ,
[AddedDate] [datetime] NOT NULL ,
[Commission] [money] NOT NULL
) ON [PRIMARY]
GO
ALTER TABLE [dbo].[VendorData] WITH NOCHECK ADD
CONSTRAINT [PK_VendorID] PRIMARY KEY CLUSTERED
(
[VendorID],
[VendorName],
[AddedDate]
) ON [PRIMARY]

```

**Listing 4.1: Example SQL statements used to create a simple table structure within the database.**

Certainly not a mega-table but sufficient to provide measurable performance numbers. I performed all my queries from a separate laptop computer, which was connected to the servers via a switched Fast Ethernet connection. No other computers were on the network, and the two servers were running Windows Server 2003 as standalone machines (not as members of a domain). I linked the two tables, then created the view using the following SQL statement:

```

CREATE VIEW TestView
AS
SELECT * FROM [ServerA].TestDB.dbo.VendorData
UNION ALL
SELECT * FROM [ServerB].TestDB.dbo.VendorData

```

I executed each query ten times, and took the average response time from all ten queries. Between each execution, I performed a number of other unrelated queries to ensure that SQL Server's data caches were receiving a fair workout, and that my test queries couldn't be satisfied entirely from the caches. Here is my first query:

```

SELECT
VendorName,
COUNT(*)
FROM
TestDB.dbo.VendorData
GROUP BY
Commission
ORDER BY
VendorName

```



For the first query, I queried the base table directly. The base table was located on a third identical server containing all ten million rows, and the query completed in an average of 58 seconds. Next, I queried the view. Each of the two servers in the federation contained half of the ten million rows, more or less at random. All my queries were returning all the rows, so achieving a fair distribution of rows wasn't particularly important. The view responded in an average of 40 seconds, which is about 30 percent faster. So, two servers are able to query five million rows apiece faster than one server is able to query ten million rows. Keep in mind that my response time also includes the time necessary for the server executing the view to compile the results and provide them to my client.

My second query was simpler:

```
SELECT
  DISTINCT Commission
FROM
  TestDB.dbo.VendorData
```

The base table responded in 42 seconds; the view responded in 30 seconds, which is about 18 percent faster. Although far from a formal test of scale-out capability, my tests indicated that distributed partitioned views provide an average 20 to 30 percent faster response time than a standalone server. Not incredibly impressive, but this example is fairly simplistic—more complex queries (and databases) would generate a greater performance increase as the complex query operations became spread across multiple servers. My example doesn't have much in the way of complex query operations, so I realized a fairly minimal performance gain. Keep in mind that my goal with this test wasn't to measure absolute performance, but rather to see whether any basic difference existed between distributed partitioned views and a straight, single-server query. As you can see, distributed partitioned views provides a performance benefit. Also notice that my queries were specifically chosen to pull all the rows in the database (or to at least make every row a candidate for selection), creating a fairly even distribution of work across the two servers. I deliberately avoided using queries that might be more real-world because they might also be affected by my less-than-scientific partitioning of the data.

### ***Conducting a Benchmark***

Make an effort to perform your own tests using data that is as real-world as possible, ideally copied from your production environment. In addition, run queries that reflect actual production workloads, allowing you to try several partitioning schemes until you find one that provides the best performance improvement over a single-server environment. Carefully document the environment for each test so that you can easily determine which scenario provides the best performance gains. Your tests should be easily repeatable—ideally, for example, running queries from saved files to ensure that each one is identical—so that you can make an even comparison of results. Your benchmarks should also be real-world, involving both queries and updates to data. You will be building a scale-out solution based on your benchmark results, so make sure that those results accurately reflect the production loads that your application will see.

## Summary

Distributed partitioned views are a powerful tool for scaling out, providing shared-nothing clustering within the base SQL Server product. Distributed partitioned views require careful planning, testing, and maintenance to ensure an even overall distribution of querying workload across federation members, but the planning effort is worth it— distributed partitioned views allow you to grow beyond the limits of a single server. Using less-expensive, “commodity” PC-based servers, you can create increasingly large federations to spread the workload of large database applications—a tactic already employed to great effect in Web server farms throughout the industry.

In the next chapter, I’ll look at additional scale-out techniques that distribute and partition data across multiple servers. These techniques are a bit more free-form and can be adapted to a variety of situations. We will explore how to set up replication and other SQL Server features to implement various additional scale-out scenarios.

## 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.realtimedpublishers.com/contentcentral/>.