

Scaling Out SQL Server  
A Review of Multiple Ways to Scale Out SQL Server Database Applications

SQL Server White Paper

Darmadi Komo

**Published:** March 2012

**Applies to:** SQL Server 2005, SQL Server 2008, SQL Server 2008 R2, SQL Server 2012

**Introduction:**

This white paper explains the different technologies that are available for scaling out a SQL Server® database application, focusing on the factors that go into deciding which solution(s) to use in your database application.

Copyright

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This white paper is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED, OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in, or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

© 2012 Microsoft Corporation. All rights reserved.

Microsoft and SQL Server are trademarks of the Microsoft group of companies.

All other trademarks are property of their respective owners.

Contents

[What Is Scaleout 4](#_Toc320119318)

[Types of Data 4](#_Toc320119319)

[Reference Data 4](#_Toc320119320)

[Activity Data 5](#_Toc320119321)

[Resource Data 6](#_Toc320119322)

[Factor Affecting Scaleout 6](#_Toc320119323)

[Update Frequency 7](#_Toc320119324)

[Ability to Change the Application 7](#_Toc320119325)

[Data Partitionability 7](#_Toc320119326)

[Data Interdependence and Coupling 8](#_Toc320119327)

[Scaleout Solutions 9](#_Toc320119328)

[Scalable Shared Databases 9](#_Toc320119329)

[Peer-to-Peer Replication 11](#_Toc320119330)

[Linked Servers and Distributed Queries 12](#_Toc320119331)

[Distributed Partitioned Views 14](#_Toc320119332)

[Data-Dependent Routing 15](#_Toc320119333)

[SQL Azure Federation 17](#_Toc320119334)

[Conclusion 18](#_Toc320119335)

[References: 18](#_Toc320119336)

# What Is Scaleout

Scalability is the ability of an application to efficiently use more resources in order to do more useful work. For example, an application that can service four users on a single-processor system may be able to service 15 users on a four-processor system. In this case, the application is scalable. If adding more processors doesn't increase the number of users serviced (if the application is single threaded, for example), the application isn't scalable.

There are two kinds of scalability: scaleup and scaleout. Scaleup means scaling to a bigger, more powerful server—going from a four-processor server to a 128-processor, for example. This is the most common way for databases to scale. When your database runs out of resources on your current hardware, you go out and buy a bigger box with more processors and more memory. Scaleup has the advantage of not requiring significant changes to the database. In general, you just install your database on a bigger box and keep running the way you always have, with more database power to handle a heavier load. Scaleout means expanding to multiple servers rather than a single, bigger server. Scaleout usually has some initial hardware cost advantages—eight four-processor servers generally cost less than one 32-processor server—but this advantage is often cancelled out when licensing and maintenance costs are included. In some cases, the redundancy offered by a scaleout solution is also useful from an availability perspective.

Many articles, books, and white papers have been written about SQL Server scaleout technologies, so this paper concentrates on the factors that go into deciding which solution to use in your application. The "References" section of this paper contains pointers to more in-depth information about the scaleout technologies.

# Types of Data

A critical factor in choosing a successful scaleout strategy is recognizing that applications deal with multiple types of data, and each type of data imposes different requirements on a scaleout architecture. When you only have one database, it's easier to just treat all data the same, but when you start splitting up and replicating data for scaleout, it's important to understand how data is used, so that the correct solution can be chosen. In many applications, multiple scaleout approaches are required, because multiple types of data are involved. This section discusses three different types of data, and how they influence scaleout decisions.

## Reference Data

Reference data, as its name implies, is data that is used by an application but that may not be maintained by the application. Reference data is relatively stable, and it is valid for a specified period. Good examples of reference data are parts catalogues used by order entry systems, schedules for airlines, and charts of accounts used in financial systems. This data is relatively stable, because it may be used by other applications, and therefore frequent changes can lead to confusion. For example, if a price on a price list changed many times a day, customers would be confused and dissatisfied (obvious exceptions to this are stock prices and gasoline prices, which change continuously). Reference data is often changed at fixed intervals—for example, prices may change nightly or even weekly, and account numbers may only change once a month. Reference data may also have a version label that is included in transactions that reference the data. For example, a purchase order may reference the version of the catalogue that was used to create the order so that there is no ambiguity in how prices were determined. A business may choose to accept several versions of reference data to avoid issues with customers using out-of-data reference data.

Because reference data is stable for a period of time and usually has a version identifier to identify which copy of the reference data was used, it can be copied to many different systems, with little danger of becoming inconsistent. In a Web farm, every Web server may have a copy of the catalogue, in order to provide quick responses to catalogue browsing requests. In many cases, common reference data is cached in memory. Stable reference data may be transferred to a smart client in order to provide quick access and offline browsing capability. If the copy is lost or corrupted, it's easy to get a new copy. The reference data version identifier can be used to determine whether the most current version is available.

All the copies of reference data are copies of a master, and therefore none of the copies are updated, except through changes to the master. This means that update conflicts aren't an issue, and therefore either snapshot or transactional replication can be used to keep reference data updated. The owner of the master copy might expose a service to return a copy of the reference data to applications that don't store it in a database.

All this means that scaleout of reference data is easy to implement and can provide improved performance with a minimal investment. There are several other common types of data with these same characteristics that can be treat the same way as reference data. For example, history seldom changes—although it does repeat itself—so that historical data such as quarterly sales figures or stock prices can be treated like reference data.

## Activity Data

Activity data is the data associated with a particular activity or business transaction. For example, a purchase order or stock sale will generate some data that is associated with the transaction. This data is relevant within the scope of a particular business activity and, except for historical reasons; it is not very useful once the activity is completed. In general, once the activity is completed, activity data becomes reference data, and the same scaleout considerations apply.

Activity data usually has low concurrency requirements. It's unlikely that several hundred users would try to access a given purchase order at the same time. While this isn't universally true, it is common enough that it can be used as a principle when scaling out activity data. Activity data also has reasonably low update rates. Once a purchase order is created, for example, it changes only when events such as a status change or a delivery date change occur, and these events tend to be relatively infrequent—several a day vs. many updates a second. Activity data is generally easy to identify unambiguously, and it isn't often accessed outside the scope of the activity. This means that if activity data is split among several databases for scaleout purposes, it will be easy to find. A business transaction that needs to access a particular purchase order will generally know the PO number; therefore, if the purchase orders are partitioned by number range, it will be easy to find the proper database to access the required purchase order.

Activity data is often scoped by another data object. For example, it may make sense to store all the orders placed by a particular customer in the same database as the customer information for that customer, if that is the way the orders are most commonly accessed. Similarly, it may make sense to store all purchase orders for a particular supplier with the supplier database for that supplier.

All this means that it is relatively easy to replicate activity data when necessary, and in many cases, it is practical to partition activity data among several databases when required for scaleout. Which approach to scaling out activity data is most appropriate depends on the data usage factors, as described in the "Factors Affecting Scaleout" section.

## Resource Data

Resource data is the core data that your business depends on—inventory, account data, customer master, and so on. If resource data is lost, you're basically out of business; therefore, resource databases use a lot of data integrity and high availability features to ensure that this critical data is always available. Resource data normally has very high concurrency requirements, because many applications and many different users all need to access the same data. A high update rate is also very common. The available quantity of an inventory item or an account balance can change many times a day. While the large number of concurrent updates makes scaling out resource data a very attractive performance option, the need for data integrity and high availability often make scaleup a more practical alternative for resource data.

Resource data is generally only the currently active data. Inactive accounts, discontinued parts, and so on, are generally maintained in relatively static history tables and, as such, become reference data. Snapshots of resource data may be taken for historical or reporting reasons, and these are also reference data. The data integrity and high availability requirements of resource data become less important when it becomes reference data. This transformation from resource data to reference data maintains the relevance of the resource data, and it also reduces the size and need for scaleout of resource data.

# Factor Affecting Scaleout

Now that we have explored the types of data stored in database systems, we will look at the characteristics of data usage that impact the choice of scaleout technologies for data. SQL Server supports several scaleout technologies, and the choice of which one to use depends largely on the characteristics of the data and application involved.

## Update Frequency

Some data is updated very frequently. Examples are log data from Web servers, and instrument readouts from machines on a plant floor (notice that these are both INSERT intensive rather than UPDATE intensive, but for our purposes, they can be thought of as the same thing). Other data is rarely, if ever, updated, and can be thought of as essentially read-only data. An example of this is historical data such as quarterly sales figures. Data warehouse data is usually updated extensively in an off-hours batch update, but then is read-only while the warehouse is in use.

Data that is updated very frequently is often difficult to replicate very efficiently, because the overhead of replicating updates limits the scalability of the replicated copies. In other words, the database may spend so much time replicating changes that the overall performance might be less than if the data was left in a single database. You should understand the update rate of your data when considering any of the replication options for scaling out.

A database with very low update rates is easy to replicate, and therefore replication can be a good scaleout strategy. If the update rate is low enough so that the database can be marked Read Only most of the time, the Scalable Shared Database option becomes viable. This option will be covered in more detail later.

## Ability to Change the Application

While not strictly a data characteristic, how much flexibility you have in redesigning the application can have a large impact on which scaleout strategies are viable for you. Some scaleout strategies work with no changes to the application at all, some may require fairly minor changes to queries and stored procedures, and others may require rethinking the way the application works. Obviously, the maximum flexibility is available if you are starting out with a completely new application. For this reason, you should think about scaleout when you are designing a new application, even if it isn't required for the initial implementation, because making changes when the application is in production and runs out of resources is much more difficult than designing for scaleout from the beginning. Packaged applications and some legacy applications may be impossible to change, and therefore any scaleout strategy will have to be transparent to the application code.

## Data Partitionability

One of the most effective ways to scale out data is to partition it among several databases so that each database server can handle a portion of the data. While this is an obvious way to scale out data, not all data can be effectively partitioned, and if it can be partitioned, the way it is partitioned has a major effect on performance.

As an illustration of the effect of partitioning, let's examine how an order database might be partitioned. One possibility might be to partition orders based on what was ordered—book orders in one database, clothing orders in another database, and so on. Aside from the obvious issues, such as what happens to an order that includes both books and clothes, this scheme wouldn't work well if the majority of queries join orders to customers, because this join would have to check all the order databases for matching orders.

Another way to partition an order database would be to split the orders by order-number range. This might make sense if orders are most often accessed by order number instead of by location, but if there are a lot of joins with the customer table, this scheme would also require distributed joins. The only way to solve the join issue would be to partition the order database by customer number, so that, for a given customer, the order database to use is always known. This is especially effective if the customer database is partitioned, and the orders for each customer are in the same database as the customer. There will be other data that must be joined to the order data, and if possible, this data should be partitioned on the same scheme, to avoid distributed joins. Some of this data may be reference data—item descriptions, for example—and this can be replicated to all the order databases, to eliminate distributed joins to the inventory database.

If application data can be divided into multiple databases, and if the extra processing power provided by the multiple servers outweighs the communications costs of assembling the results, the data is partitionable. Not all application data can be effectively partitioned, and choosing the right partitioning scheme is essential for effective scaleout with partitioned data.

## Data Interdependence and Coupling

Another way to distribute data for scaleout is to split it based on usage. If parts of the database are used by different applications, it may make sense to split the database along application boundaries, so that each application has processing dedicated to the data it uses. This make sense if the data used by the different applications can be segmented in order to provide dedicated database processing without incurring the penalty of excessive network traffic caused by data interdependence. As an extreme example, if an order database is split so that all the order lines are in one database and all the order headers are in another database, virtually every query that accesses orders will have to do a distributed join between the two databases. The network traffic generated by the distributed joins will cancel out any benefits from distributing the data.

Perhaps the best way to determine how a database can be split is through a careful analysis of the data model. Relationships in an Entity-Relationship diagram represent both join paths and referential integrity constraints. Both joins and constraint enforcement are expensive to do across distributed databases. The database will not enforce referential constraints between databases; therefore, if related tables are split between databases, either the constraint will have to be ignored, or it will have to be enforced with a trigger or by the application. To avoid these issues, look for islands of related data in your data model. These are groups of tables that have few, if any, connections to the rest of the data model. In an order entry database you will generally find few relationships between the customer tables and the inventory tables, for example.

Once you have decided which groups of tables can be isolated on different databases, without too many referential integrity issues, you need to look at update patterns. If splitting your data between databases results in a large number of distributed transactions spanning the databases, the extra update overhead of the two-phase commits may cancel out some of the scaleout advantages.

The last factor to consider in data coupling is how shared tables are handled. There will be a certain number of tables that are accessed by multiple applications; therefore, when you split the data, you will have to decide where to put the shared tables. In some cases, a table may be read by many applications but only updated by one application, so that locating it with the updating application is probably the logical choice. The relational integrity and distributed issues we discussed earlier may also influence shared table placement. If the table is relatively small and used extensively by several applications, it may make sense to replicate it to multiple databases. This is easiest if the table is updated by a single application, so that transactional replication from the master copy can be used to keep the other copies up-to-date.

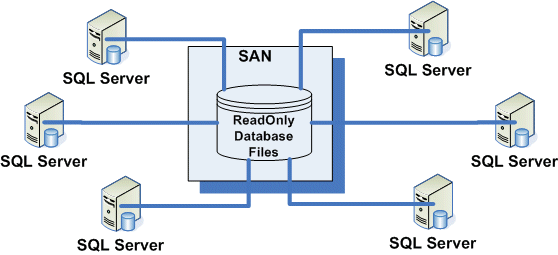
Resource data is often highly interdependent, with a lot of integrity constraints, which means a high degree of coupling. In some cases, this means that splitting resource data by application is either impossible or requires some application changes. You can scale out the reference and activity data, or partition the resource data so that scaleout is still possible if the resource data is tightly coupled, but some of the scaleout options that require splitting the data may require an application re-architecture, to go along with the data re-architecture. As we said in the section on application change, designing the data model so that it can be split up later is a good practice when building a new application.

# Scaleout Solutions

Now that we understand the factors that go into choosing a scaleout solution for SQL Server, we will discuss each scaleout solution, and what factors would lead you to choose that solution. This paper doesn't go into depth on the scaleout solutions. More in-depth information is available in the articles referenced in the "References" section.

## Scalable Shared Databases

The easiest scaleout solution to implement in SQL Server is Scalable Shared Databases. You create a database on a SAN, and up to eight SQL Server instances running on different servers attach to the database, and start handling queries. This is the classic "shared disk"–style scaleout solution, where processing power is scaled out, but only a single disk image of the data is used. At this point, those of you who understand SQL Server are saying, "But what happens to the locks? I thought each SQL Server instance kept its own locks in its own memory." This is true. Each instance will maintain its own database locks, and none of the instances will know about the other instances' locks. The only way this will work is if there are no locks, and thus Scalable Shared Databases work only if the database is attached as a Read Only database. This means that Scalable Shared Databases are great for data warehouses or reporting databases, but they are not suitable for applications that update data. Going back to our data characteristics, Scalable Shared Databases work only if the Update Frequency is zero. This data is, by definition, historical, and therefore it is all reference data. Figure 1 illustrates the use of Scalable Shared Databases as a scaleout solution.



*Figure 1. Scalable Shared Database*

Obviously, a database that never changes is of limited value, therefore, to update the database, all the SQL Server instances detach the database, one of the instances attaches it in Read-Write mode, and the database is refreshed with current data. This may take a while if a lot of data has changed; therefore, if the SAN has enough space available, it may make sense to maintain two databases, so that one can be updated while the other one is in use. The SQL Server Books Online and the articles in the "References" section explain how to set this up.

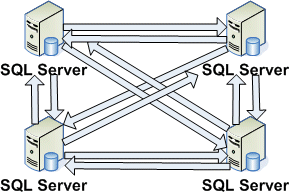
The limitation of eight database engines attached to a single database is not a technical limit, but a tested limit. It's likely that future releases will support more than eight. Because the database is never written to, and as much data as possible is cached, the actual disk IO rate is fairly low, even with eight heavily loaded database engines attached.

Many of the most demanding database applications work well in Read Only mode. The workload for a data warehouse is a relatively small number of large, complex queries against reasonably static historical data. The update frequency for most data warehouses is daily, or even weekly, so that making the data warehouse read-only while it is in use is not a hardship. A Scalable Shared Database also solves the common data warehouse problem of the user who writes a query that ties up all the database resources for hours. When this query is running on one of the database engines, the rest of the engines are still available to process queries, and therefore the impact of the "query from hell" is minimized.

Scalable Shared Databases are practical only if the update frequency is very low, because the database can't be updated at all while it is shared. Scalable Shared Databases require no application changes, provided that the application doesn't try to update the database. The database is scaled out without changes, and therefore partitionability and coupling are not factors in the decision to use Scalable Shared Databases. In summary, Scalable Shared Databases are useful in applications such as data warehouses, data marts, and reporting databases, where the data update frequency can be reduced to periodic batch changes. If this update pattern is acceptable to the application, Scalable Shared Databases are the preferred scaleout method, because they are simple to implement and require minimal application changes.

## Peer-to-Peer Replication

Scaling out data that must be updated, but that still has a relatively low update frequency, is often best handled by replication. Instead of several database engines accessing a single copy of the database, you have multiple copies of the database, so that each database engine maintains its own copy. This option provides scaleout, while also allowing updates to the data. Replication is used to propagate changes to all copies of the data. Figure 2 illustrates the use of Peer-to-Peer replication as a scaleout solution.



*Figure 2. Peer-to-Peer replication*

Peer-to-Peer replication in SQL Server propagates changes made to any copy of the data to all other copies. Peer-to-Peer replication does not provide conflict resolution, and therefore it is recommended only in configurations where only one copy of a given data element is updated. For example, if Peer-to-Peer replication is used to maintain the inventory of a chain of food stores, only the store that owns an inventory entry would be allowed to update that entry. This means that, even though all stores would be able to see the inventory of every other store, they are only able to change the inventory of their store.

The rules that allow only the owner of a data item to update that data item are known as data stewardship. Stewardship is a very useful way to avoid data update conflicts. In situations where data stewardship isn't practical, merge replication can be used to handle conflicts. Merge replication imposes more overhead than Peer-to-Peer replication, because it must deal with conflicts; therefore, Peer-to-Peer replications is more desirable if conflicts can be avoided. Peer-to-Peer replication requires that replication be set up from every copy of the database to every other copy of the database, so that managing this can be a burden when many databases are involved. A single master copy that uses transactional replication to keep the other copies up-to-date is the simplest and most efficient solution if the updates to the database can be limited to a single copy of the database.

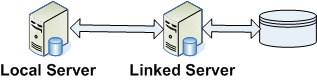
Replication is a good scaleout solution for data with a moderate update frequency. Data stewardship can be used to eliminate update conflicts so that Peer-to-Peer replication to be used. If necessary, replication can be used for all the data in a database, and therefore it is useful when data isn't easily partitionable, or when it has a high degree of coupling. In most cases, replication requires no application changes, and therefore it can be used to easily scale out existing applications. Replication can work on single tables, or even parts of tables, and therefore replication is useful for scaling out parts of an application's data. For example, reference data such as catalogues and prices lists can be replicated to improve scaleout, even if the resource data is not scaled out. Replication is also useful in combination with other scaleout solutions. If activity data is scaled out by partitioning it into different databases, it may make sense to replicate the reference data used by the activity data to all the activity databases. In general, replication is one of the easiest and most widely applicable scaleout solutions in SQL Server.

Use the configure Peer-To-Peer replication topology page to perform common configuration tasks, such as adding new nodes, deleting nodes, and adding new connections between existing nodes. This easy to use graphical user interface tool enables easy creation and maintenance of Peer-to-Peer replication topology.

Peer-to-peer replication in SQL Server has the option to enable conflict detection across a peer-to-peer topology. This option helps prevent the issues that are caused from undetected conflicts, including inconsistent application behavior and lost updates. By enabling this option, by default a conflicting change is treated as a critical error that causes the failure of the Distribution Agent.

## Linked Servers and Distributed Queries

SQL Server has the ability to query remote database objects as if they were local. This means that a scaled out database can look like a single large database to an application. The major change to the SQL Queries is that the table names will now include the name of the linked server where the data is located. This makes linked servers an attractive scaleout option when changing the application is difficult to do. In SQL Server, synonyms can be used to expose a four-part name that includes the server name as a single name, so that queries that used to address a local table can now address a remote table without changing the query. Figure 3 illustrates the use of linked servers as a scaleout solution.



*Figure 3. Linked servers*

Linked servers are most useful in an environment where the data can be split by functional area into databases with very little coupling. Referential integrity constraints don't work with remote tables, and therefore relationships between local and remote data need to be minimized. Remote queries are significantly more expensive than local queries; joins between local and remote tables can be very expensive; and updates of remote tables require distributed transactions; therefore, successfully scaling out a database with linked servers requires careful database design in order to minimize remote data access. If applications use data islands that have little coupling between then, and if relatively few queries span the islands, these data islands can be distributed between databases for an effective scaleout solution. If there are hot spots, where certain tables are used extensively by multiple applications that use different data islands, you can use replication to make most of the queries that involve these tables local.

The scaled out databases should also be designed so that most of the data used by any given application is in a single database. For example, if several applications use both customer and order data in most of their queries, it probably doesn't make sense to split customer data and order data into separate databases—even if the data is loosely coupled—because no matter which database an application opens, it will be continuously accessing some of its data remotely. On the other hand, if some of the applications access customer data almost exclusively, and other applications access order data with very little access to customer data, this split will be effective. Again, if a few queries use data from both databases extensively, replication can be used to reduce network traffic. For example, if orders require a valid customer number, and if every order includes a customer name, the customer number and customer name columns from the customer database can be replicated to the order database.

Like replication, linked servers are also valuable as part of some of the other scaleout solutions. In any distributed database, there will be a certain number of queries that have to cross database boundaries, and linked servers are a simple way to support these queries.

Linked servers are also useful when splitting data by type. For example, if all the history over 60 days old is transferred to other databases, a linked server implementation can allow the application to handle a few queries that access the historical reference data as if it were local. This strategy effectively moves reference data processing to dedicated hardware, without impacting the applications that use it. Applications that access history extensively can run on the history machines and access the live data only when required.

The update frequency has little impact on a linked server scaleout solution, provided that the updates don't span databases; therefore, two-phase commit isn't required. This means that a linked server solution can be used effectively in some OLTP applications. Linked servers also require little or no application change, and therefore they are appropriate for legacy applications. Partitioning data by key value is not generally used in linked server implementations, and therefore partitionability isn't a factor. The key factor in deciding whether a linked server scaleout solution is appropriate is data coupling. If coupling is high, the overhead of distributed queries will cancel out the performance gains of scaleout. A clear understanding of data relationships and application data usage is required in order to determine whether linked servers are a viable scaleout option.

## Distributed Partitioned Views

Distributed Partitioned Views (DPVs) were put into SQL Server specifically to support transparent scaling out of partitioned data. The data in a table is partitioned among tables in several distributed databases, based on a partitioning key. For example, a customer table might be partitioned on customer number ranges (1–1000 in one database, 10001–20000 in a second database, 20001–30000 in a third, and so on). Check constraints are used to tell SQL Server which customers are located in which database. A query that accesses a particular customer number will be executed only on the database that contains the desired customer number. Queries that don't reference a particular customer number will have to be run on all the partitions. For example, if you need to look up information about George Bush, and you don't know his customer number, SQL Server would have to look in all the databases that contain a partition of the customer table. Similarly, a join between two tables on the customer number can execute in parallel on each database, with the results returned to the database where the query originated.

Like most scaleout solutions, successfully using DPVs involves minimizing the number of queries that require moving data between databases. If all the data in the DPV databases is partitioned with the same partitioning key, any joins on that key can execute locally. If you partition an order entry database so that all the tables are partitioned on the customer key, and if all database queries contain a customer key, any query can be satisfied by going to the database that contains that customer key. Of course, if you execute the query from a database that doesn't contain that key; the query will be a distributed query. In practice, it's not possible to partition all the data on the same key, and therefore some tables may be partitioned on different keys, and many tables will not be partitioned in most implementations. This means that many queries will have to access multiple databases so the success of the scaleout effort will depend on the relative mix of distributed and local queries. A join between a partitioned table and a non-partitioned table can end up being expensive because the non-partitioned data may have to be sent to all the servers where the partitions are located. Achieving a high proportion of local queries usually involves not only carefully designing the partitioning schema, but also changing the application to take advantage of the partitioning. This means that, even though DPVs were designed to provide transparent scaleout, in many cases, the applications have to be changed anyway. Coming up with a partitioning schema that works for many different applications is difficult; therefore, DPVs are generally more useful if a limited number of related applications are using the partitioned database. One way of limiting the number of applications that access a DPV scheme is to split the data by application, as well as partitioning it by key value. If this is done, it is likely that only a few of the resulting application databases will be large enough to benefit from DPVs, so that a combination of partitioned and non-partitioned schemas will result.

DPVs provide good performance in update-intensive applications, because most update transactions affect a small number of rows, and thus are likely to execute on a single database. This makes DPV the first scaleout solution that works with high update frequencies. In theory, a DPV implementation shouldn't require any application changes, because the scaleout is handled by SQL Server and is transparent to the application. In reality, some application changes may be required in order to take full advantage of the scaleout. The key decision point for a DPV implementation is data partitionability. If there aren't good partitioning keys that evenly partition the data and minimize cross-partition queries, DPVs are not practical. A lot of up-front analysis and planning is required in order to pick the correct partitioning keys. Data coupling doesn't have much impact on a partitioned view, but if there are a lot of unpartitioned tables in the databases, a high level of coupling can make it difficult to distribute the data efficiently.

OLTP applications are often the best candidates for Distributed Partitioned Views, because they have large amounts of data and high update frequencies. In general, a partitioned view takes a lot more management effort than the same data in a single table. Backup and restore have to be able to synchronize all partitions of a table in order to be able to restore to a consistent state, for example. This extra management effort, coupled with the relatively high effort to get a DPV solution running efficiently, are the main reasons why not many applications use DPVs for scaleout.

## Data-Dependent Routing

With Distributed Partitioned Views, SQL Server understands how the data is partitioned, and decides where to go to find the data. With Data-Dependent Routing (DDR), the data is partitioned among databases, and either the application or some middleware services routes queries to the correct database. DDR is obviously much less transparent to the application, but if the application is involved in the decision about where to execute a query, the deeper knowledge that the application has about the data can lead to a better decision. DDR can also support options that SQL Server does not currently support directly. For example, if there are multiple copies of the same data available, DDR may distribute SELECT's across the copies while directing updates to a single master copy. Figure 4 illustrates the use of Data-Dependent Routing as a scaleout solution.

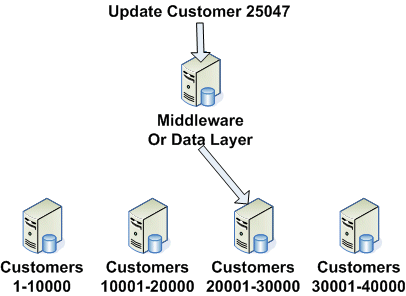


Figure 4. Data-Dependent Routing

Many of the biggest databases with huge transaction volumes use variations of the DDR scaleout option to distribute processing across hundreds, or even thousands, of database servers. With this many servers, almost all queries will have to be directed to a single server, and therefore the data model must be designed so that all the data needed for a query or update is located on the same server. For example, a large online retailer might partition its databases on the customer ID. This means that all information about that customer, including all the orders the customer has placed, the customer's billing status, credit card info, and so on, are located in the same database as the customer data for that customer. This is often referred to as a customer entity. Partitioning data into entities—self-contained groups of data that can be referenced by a single identifier—is the way most mega-scale databases are designed. Each database engine handles queries for the customer entities in its database. When the number of entities gets too large, a new server is added, and the entities are redistributed.

Now that we have all our customer data partitioned into entities identified by customer ID, the next issue is, "How does the application know which customer ID to use?" In most cases, that's pretty trivial. If the customer is going to be placing orders, you can assume that a login has taken place, and that the customer's ID is in a login record somewhere—in the Web server or a cookie, for example. If there isn't a login process, a table mapping the customer's identity (name, Window SID, and so on) to a customer ID is easy to implement. The last piece of the puzzle is a table that maps customer IDs to the database server where that customer entity is located. Once this is in place, a data access layer—either in the application or in a middle-tier application—can expose an API that allows the application to access the data stored in a given customer entity. Requests are routed based on the data the request addresses—hence the name Data-Dependent Routing.

The obvious question at this point is, "How do I search across hundreds of databases to get summary data or find all the orders?" The short answer is, "You don't." As part of designing a DDR application, you must design the facilities needed in order to handle these queries. A typical approach would be to replicate summary data for each order into an order database along with the customer ID, so that most order queries can access the summary data. If an application needs the detail for an order, it can use the customer ID from the summary data to locate the order. Summary data is also commonly loaded into a data warehouse or reporting database for use in generating reports, data mining, data analysis, and so on.

While using DDR to achieve mega-scaleout to thousands of database servers isn't common, using the same principles to scale out to tens of database servers is a viable solution for many applications. Relocating data, managing replication, extracting summary data, and so on, make this solution relatively complex to manage, but much of this work is repetitive and can be automated.

The DDR solution is designed for high transaction volumes, and therefore it is a clear winner for applications with very high update frequencies. DDR requires a data layer that understands how to locate and access data entities from the databases where it is stored, so that DDR is most suitable for a new application. The use of a middle-tier application to handle the data routing will reduce the number of changes to the application code itself, but adapting a legacy application to DDR will require extensive changes. The requirement for data partitionability is extremely high for a DDR, because this solution won't work unless the data is partitioned so that a partitioning key can be used to locate all data entities. Low data coupling may also be required, because it is generally not possible to partition the whole database on a single key. In our order entry example, it makes no sense to partition the inventory by customer ID, for example, and therefore inventory data will have to be isolated in its own set of database servers—possibly partitioned on catalog number. In general, DDR is one of the most effective ways to scale out a database, but it is also one of the most difficult to implement. It is generally most appropriate when applications are being redesigned or modified. DDR is also suitable for scaling out only part of an application. For example, you may want to use DDR for the order database, and either use another scaleout strategy for the rest of the data, or simply scale up the remainder of the data if removing the order data makes the remaining data small enough to manage.

## SQL Azure Federation

For highly scalable cloud computing need, Microsoft introduced Federation in SQL Azure for highly scale out requirements to address challenges in repartition/redistributing data and the need of robust connection routing. Using Federation, applications can scale from 10s to 100s of SQL databases. Federation also provides easy repartitioning of data without downtime. For multi-tenancy requirements, Federation provides repartitioning operations for easy management of tenant placement and re-placement without any application downtime. Few examples of database applications that can use Federation are software vendors implementing multi-tenant SaaS solutions, large web scale database solutions to support peaks, bursts, spikes or new flood of users, and NoSQL applications.

# Conclusion

The key thing to take away from this discussion of SQL Server scaleout is that there are different kinds of data in any application, and an effective scaleout solution may include different approaches for different data. Reference data may be replicated and cached in many different places; historical data may be exposed through distributed queries on low-cost, high-capacity storage; activity data may be partitioned across a number of servers; and resource data may be split by application.

The decision to use a scaleout solution is influenced by several factors. Table 1 summarizes the importance of these factors for each solution.

*Table 1. Factors influencing the selection of scaleout solutions*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Update Frequency** | **Ability to Change Application** | **Data Partitionability** | **Data Coupling** |
| **Scalable Shared Databases** | Read Only. | Little or no change required. | No requirement. | No requirement. |
| **Peer-to-Peer Replication** | Read mostly, no conflicts. | Little or no change required. | No requirement. | No requirement. |
| **Linked Servers** | Minimize cross-database updates. | Minor changes. | Not generally required. | Very important to have low coupling. |
| **Distributed Partitioned Views** | Frequent updates OK. | Some changes may be required. | Very important. | Little impact. |
| **Data-Dependent Routing** | Frequent updates OK. | Significant changes possible. | Very important. | Low coupling may help some applications. |

Remember that some scaleout architectures may incorporate multiple solutions. Replication and linked servers are generally part of any scaleout architecture.

With a good understanding of the data, requirements, and constraints for an application, an effective SQL Server solution can be designed to meet almost any level of scaleout. Even if scaleout isn't an initial requirement for an application, including scaleout considerations in the design can pay dividends later as the application and your business grows.

For high scale out solution for database applications in the cloud environment, SQL Azure Federation enables the concept of elastic computing with the ability to scale out to hundreds of nodes as well as down.

# **References**:

**SQL Server Web Site** http://www.microsoft.com/sqlserver/en/us/default.aspx

**Scalable Shared Database**

* [Scalable shared databases](http://msdn.microsoft.com/en-us/library/ms345584.aspx) <http://msdn.microsoft.com/en-us/library/ms345584.aspx>

**Peer-to-Peer Replication**

* [Peer-to-Peer Transactional Replication](http://msdn.microsoft.com/en-us/library/ms151196(v=sql.110).aspx) <http://msdn.microsoft.com/en-us/library/ms151196(v=sql.110).aspx>
* [Configure Peer-to-Peer Transactional Replication Topology](http://msdn.microsoft.com/en-us/library/ms183664.aspx) <http://msdn.microsoft.com/en-us/library/ms183664.aspx>
* [Conflict Detection in Peer-to-Peer Transactional Replication](http://msdn.microsoft.com/en-us/library/bb934199(v=sql.110).aspx) <http://msdn.microsoft.com/en-us/library/bb934199(v=sql.110).aspx>

**Linked Servers**

* [Linking Servers](http://msdn.microsoft.com/en-us/library/ms188279.aspx) <http://msdn.microsoft.com/en-us/library/ms188279.aspx>

**Distributed Partitioned Views**

* [Implementing Federated Database Servers](http://msdn.microsoft.com/en-us/library/ms191185.aspx)  <http://msdn.microsoft.com/en-us/library/ms191185.aspx>
* [Resolving Distributed Partitioned Views](http://technet.microsoft.com/en-us/library/ms187836.aspx) <http://technet.microsoft.com/en-us/library/ms187836.aspx>
* [Modifying Data in Distributed Partitioned Views](http://msdn.microsoft.com/en-us/library/ms187067.aspx) <http://msdn.microsoft.com/en-us/library/ms187067.aspx>

**Data-Dependent Routing**

* [Scaling Out SQL Server with Data Dependent Routing](http://technet.microsoft.com/en-us/library/cc966448.aspx) <http://technet.microsoft.com/en-us/library/cc966448.aspx>

**SQL Azure Federation**

* [Federation in SQL Azure](http://msdn.microsoft.com/en-us/library/windowsazure/hh597452.aspx)  <http://msdn.microsoft.com/en-us/library/windowsazure/hh597452.aspx>

Did this paper help you? Please give us your feedback. Tell us on a scale of 1 (poor) to 5 (excellent), how would you rate this paper and why have you given it this rating? For example:

Are you rating it high due to having good examples, excellent screen shots, clear writing, or another reason?

Are you rating it low due to poor examples, fuzzy screen shots, or unclear writing?

This feedback will help us improve the quality of white papers we release.

[Send feedback](mailto:sqlfback@microsoft.com?subject=White%20Paper%20Feedback:%20SQL%20Server%20Scale%20Out).