Microsoft® Windows Server System™

Microsoft® SQL Server™ 2005

Why Consider a Service-Oriented Database Architecture for Scalability and Availability

White Paper

Published: November 2005

For the latest information, please see http://www.microsoft.com/sql/

Microsoft®

Contents

Introduction 1

Scalability Overview 1
- Scalability Approaches 1
- General Hardware Issues 2
- Scale Up 2
- Scale Out 3

Transparent Scale-Out 4

Nontransparent Scale-Out 4

Scale-Out Through SODA 5

Service-Oriented Database Architecture (SODA) 5
- Introduction 5
- Elements of SODA 6

Database Services 7

Interservice Communications 7

Native XML 8

Complex Business Rules 8

Reference Data 9

Infrastructure 9
- SODA and High Availability 9

SODA Service Broker 9

Replication 10

SQL Server 2005 Database Services 10
- XML Support 10

Exposing Database Services 10

Processing XML 11
- Service Broker 11
- Microsoft .NET Framework 11
- Replication 11

Scalable Application Example 11

Conclusion 14

Introduction

Scalability is one of the key characteristics of any database system. It refers to a database system's ability to handle higher volumes of transactions, larger volumes of data, more complex queries, and more complex application requirements. Microsoft® SQL Server™ 2005 contains major improvements in all four areas of scalability and introduces a new architectural approach for handling high transaction volumes, the Service-Oriented Database Architecture (SODA). With SODA, databases become an interconnected set of highly available Web services.

SODA closely mirrors modern practice in application construction and allows for unlimited scalability by dividing database processing along Service boundaries. Services can be scaled independently or partitioned into new services to handle additional load, availability, or business requirements. Each Service can be made highly available, and the overall application can be designed to provide Continuous Availability. This white paper provides an overview of the Service-Oriented Database Architecture and the way it is supported by SQL Server 2005.

Scalability Overview

Scalability is a term used across the spectrum of computing to describe a system's ability to handle ever-increasing amounts of work. In the world of transaction processing, it is used primarily to describe a system's ability to handle higher volumes of transactions, and secondly to describe the ability to handle more complex applications. In the world of data warehousing, the term is generally used to describe a system's ability to handle larger volumes of data (VLDB) and more complex queries against that data. This white paper focuses on scalability in transaction processing systems and, thus, primarily on the issue of handling higher volumes of transactions.

In the last five years, improvements in hardware, operating systems, and database management systems have delivered a 20-fold increase in the peak volume of transactions that a single database system is able to support. While improvements in these areas will continue to drive peak transaction volumes higher, even greater improvements at lower costs are possible by changing how databases and database applications are constructed and deployed. It is this latter area where SODA comes into play. Before delving into SODA, it is helpful to review traditional techniques and issues in scalability.

Scalability Approaches

Software designers spend much of their time either trying to figure out how to exploit new developments in hardware, or how to deliver capabilities that go beyond those available from the hardware. In the area of scalability, this has resulted in two basic approaches, scale-up and scale-out. With scale-up, hardware designers provide bigger and faster computer systems. Software designers then have to figure out how to take advantage of those systems. With scale-out, software designers connect multiple computer systems together to create a larger network of systems that can handle transaction volumes far in excess of a single computer system. Both scale-up and scale-out have advantages and disadvantages, and each basic approach has several variations. High-end applications will often require a combination of these two approaches.

General Hardware Issues

At the level of an individual computer processor, performance is doubling approximately every 18 months. This phenomenon, known as Moore's Law, is the result of the rate of improvement in semi-conductor manufacturing technology. At the computer system level, the situation is far more complex. Despite the rapid growth in the power of an individual processor, many application systems have far greater requirements than can be met by a single processor. Connecting multiple processors together into a single system can be complex, costly, and result in less aggregate computing power than expected. For example, connecting four processors together may only provide three times the improvement in transaction rates over the performance of a single processor.

Transactional applications are generally very I/O intensive, and much of the attention in computer system design is focused on matching I/O capabilities to the processor capabilities. Without adequate I/O capabilities, connecting together increasing numbers of processors would yield even smaller improvements.

As the cost of connecting numerous processors together with adequate I/O increases, it collides with diminishing returns in improving transaction rates. Further, placing faster processors in an existing computer system design may not allow the overall system performance to improve as much as the processor performance might indicate, because of the overall system becoming unbalanced. This is particularly noticeable with systems utilizing a large numbers of processors, and places boundaries on the rate that hardware improvements can be turned into scalability improvements.

While servers that are used to host database systems typically range up to 64 processors, the "sweet spot" is the four-processor system. At the application level, four-processor systems are sufficient to meet the scalability requirements of the vast majority of customers. At the technology level, four-processor systems generally track improvements in processor performance, can be constructed using commodity technologies, and thus are offered at low cost. For application requirements that can be satisfied by a four-processor system and whose rate of growth is less than Moore's Law, using a single four-processor system and replacing it every few years is by far the simplest and most cost-effective scalability solution. For applications that do not fit on a four-processor system or whose growth rates exceed Moore's Law, either further scale-up is required or scale-out technologies need to be introduced.[1]

Scale Up

Scale-up is simply the approach of using a larger single computer system to handle increased transaction volumes. Typically, this involves the use of additional processors and complementary I/O capabilities. This can be as simple as going from a single processor to dual processors, or as complicated as employing 64 processors in a single computer system. The chief benefit of scale-up is that system software makes the addition of processors completely transparent to applications and operational personnel. You just add processors and your maximum transaction rates go up. This makes scale-up the most attractive option for most applications. If you run out of steam on one processor, you go to a dual-processor system. If you run out of steam on two processors, you move to a four-processor system. After that you can go to 8, 16, 32, or even 64 processors. With such a simple solution to scalability, why is any other solution required?

As mentioned earlier, it is quite difficult to connect large numbers of processors and get them to scale. The result is that beyond the four-processor range, the cost of systems escalates at

[1] The design complexity at the motherboard or system level scales with the number of sockets used in the system where each socket contains a processor package. Systems with more than four sockets are much more difficult to build in a way that scales effectively. With the advent of multi-core and multi-threaded processors, each socket may host a processor with multiple cores and each core may support multiple threads of hardware execution. As such, a four-socket machine with dual-core processors and each processor supporting two threads of execution will appear to the host operating system as a 16-processor machine.

a faster rate than the increased scalability they provide. Most of the incremental cost of these large systems is in the system infrastructure instead of in the cost of individual processors.

For example, if you need four processors now but next year will require eight and then sixteen processors some time in the future, you must pay for the expensive infrastructure needed to support sixteen processors up front. This creates a large step up in the entry price between systems that top out at four processors and those that top out at larger numbers. In addition, when availability considerations are included, it is often necessary to duplicate computer systems. This further magnifies the cost difference between commodity four-processor systems and larger systems. Also, a few applications have scalability requirements that exceed the capacity of even a 64-processor system. Another issue is that business or organizational requirements may dictate that an application be decentralized.

Even with its higher initial acquisition costs, scaling up to large numbers of processors may be the best approach for most customers needing greater than the scalability provided by four-processor systems. By holding operational and application costs constant while increasing scalability, scale-up keeps the total cost of ownership low.

Scale Out

Scale-out refers to increasing the amount of work that can be processed by spreading the work over additional computer systems. The most popular example of the last few years has been the advent of Web Server Farms. In their simplest form, Web servers respond to requests for pages of read-only data. Because the data is read-only, it can easily be replicated across many Web servers (the "farm"). A request for a particular page can then be routed to and satisfied by any available Web server in the farm. When growth in the request rate for pages occurs, another server is added to the farm and given a copy of the pages for the Web site. Software, or a networking box, automatically redirects some portion of requests to the new box. Scalability is almost unlimited. There are a wide variety of variations on this basic concept, though only a few are applicable to database systems.

Database systems differ from many other types of systems in that they maintain a shared, updateable, persistent state. Imagine trying to apply Web Server Farm technology directly to databases. For example, consider a database with inventory for seats on a particular airplane flight and making copies of it on a dozen different servers. What happens when the network redirector sends customer A's request to reserve seat 21A to server #1 and customer B's request for seat 21A to server #7? In a farm-type arrangement, both requests would be satisfied independently by the two servers and seat 21A would improperly be assigned to two passengers. To solve this, either all requests for seat 21A would have to go to the same server or the servers would have to coordinate their updates. This makes scale-out of databases much more difficult than for "stateless" servers such as Web and applications servers.

Database scale-out is further complicated by the notion of transparency. For Web access, transparency is at the level of the Web address (URL). When you ask for the Web page at http://www.microsoft.com, any server in the Microsoft farm can supply that page. The user (or application, in the case of a Web Service) does not have to know which server to ask the question of. What level of transparency is desired or required for database applications? If you want to print the itinerary for a customer's trip, where is the knowledge that the seat assignments for Flight 100 are on server #1 and for Flight 200 on server #7 stored? Further, who is responsible for bringing this information together into a single answer? Is it the database system or the application?

In the past, database scale-out techniques have fallen into one of two categories: transparent and nontransparent. SQL Server 2005 introduces a third type of scale-out, the Service-Oriented Database Architecture.

Transparent Scale-Out

Transparent scale-out is an attempt to duplicate the scaling properties of scale-up solutions across a set of networked, or loosely connected, computer systems. In an ideal situation, gaining additional capacity would be achieved by networking in an additional computer system. Neither the application nor the database schema would require any changes, and the system would dynamically rebalance as workloads and resources changed. Applications would just address the database, not even knowing that it was spread across numerous computer systems. As with scale-up solutions, scalability would improve uniformly across applications and workloads. Unfortunately, even after 20 years of development, this dream has not been achieved. From a scaling perspective, the primary issue is that the cost of communications required to coordinate access to shared data and combine the results from (potentially) multiple nodes into a response to a database request can exceed the raw computing power available from connecting the computer systems.

Today's transparent scale-out solutions scale reasonably in only two environments. The most successful environment for database scale-out is when a workload overwhelmingly consists of read requests. When little or no update activity is taking place, minimal cross-node communications is required and the benefits of the additional compute power are available to satisfy requests. For many applications, replication provides a superior solution to the problem of handling larger volumes of read requests.

The other successful environment is for databases that can be carefully partitioned across nodes, with application requests directed to the specific partition containing the required data. If requests are routed to the correct server partition and very few of these requests require cross-partition data, much of the benefit of the additional compute power may also be used to satisfy requests.

Although this technically provides transparency at the database request (SQL) level, for applications to scale well the application must be aware of the partitioning scheme and avoid requests that cross partition boundaries. As loads change, both the database and application may have to be modified in response. And, application requirement changes can invalidate the careful partitioning and result in severely degraded performance. This has resulted in transparent scale-out being more successful in the static environment of benchmarking than in the dynamic environment characteristic of most real applications.

Nontransparent Scale-Out

Nontransparent scale-out comes in several variations, but it generally refers to using multiple independent servers to run various databases. An application that requires data in two or more different databases directly connects to those databases and issues two separate database requests. The application is responsible for combining the results of those requests. Most often, the databases are partitioned by function. For example, one database might contain inventory data while another might contain customer data. In other cases, the data might be partitioned by value, such as one database containing data for customers in the eastern region and the other for customers in the western region. Scaling of these systems can be quite good, but the burden on application development and maintenance is very high.

The most successful examples of nontransparent scale-out are those where a single application does not directly address more than one database. Instead, application components directly access only a single database and make requests to other application components against data in databases not directly under their control. These requests may be through Web Services, Message-Oriented Middleware, RPC, or even batch processing. Replication is often used to make read only data, such as reference or catalog data, available on multiple nodes, and thus avoid a central database becoming a bottleneck for this data.

Scale-Out Through SODA

With the Service-Oriented Database Architecture, or SODA, databases are partitioned according to well-defined service boundaries that meet application requirements. These database services can then be hosted on a single server node, or multiple server nodes, to achieve the desired level of scalability.

Unlike transparent scale-out, SODA avoids SQL-level cross-partition operations, and all the scalability limitations they bring, in favor of well-defined requests between database services. Additionally, unlike nontransparent scale-out, SODA integrates support for service interfaces and interservice communications, and also routing directly into the database system, thus relieving the application development and maintenance burden. The rest of this white paper explains SODA and provides an overview about the way SODA is supported by SQL Server 2005.

Service-Oriented Database Architecture (SODA)

Introduction

For more than a decade, application architecture has trended toward independent application systems interconnected by some type of middleware. For example, an e-Commerce application may have completely independent application systems for Web-based order entry, customer profile, internal “heads down” order-entry, distribution, shipping, manufacturing, and credit processing. Instead of having a centralized database that supports all these applications, each application system has one or more of its own databases. There are no cross-database operations.

Instead, application systems communicate with each other through well-defined interfaces to obtain data required to complete their own operation and also to make changes in the data owned by the other application system. Some application systems, such as shipping, may actually be owned by third parties that preclude direct database access. Increasingly, the interactions between application systems are being expressed in terms of exchanges of XML documents. Historically, database systems have provided little support for this application architecture, creating a large burden on the application developer. The Service-Oriented Database Architecture puts support for this application architecture directly into the database system.

From the 1970s to the mid 1980s, database systems exposed little in the way of application architecture. They were passive stores that were called by an Application Programming Interface (API) with Data Language (the most popular being SQL) requests to read or write data. All business and data integrity rules were the responsibility of the application.

As a result of the proliferation of ad hoc and 4GL application development tools, database systems in the mid 1980s started to incorporate the ability to create data integrity rules and small amounts of business logic. This allowed multiple applications, or even end users, to directly access the database without corrupting its contents. By the late 1980s, the proliferation of personal computers that wanted direct access to databases lead to the development of a full application architecture for the database system. Business logic and data integrity rules could be incorporated into stored procedures using procedural extensions to SQL, which the application invoked using a database remote procedure call (RPC). This became the classic two-tier Client/Server application model. For multitier applications, the model has remained a mix of direct application data access through SQL and the use of stored procedures, as well as triggers and constraints, for maintaining data integrity and enforcing some common business rules. While many refinements to this basic model have

been made, overall database application architecture has changed little over the past 15 years.

To bring database systems into synch with modern application architecture requires a number of substantial changes to the traditional database application architecture. Specifically, this includes the following:

- Existing models for direct data access and the use of database RPCs for calling stored procedures have to be supplemented with native support for exposing data through Web Services. This allows middle-tier, external partner, and even client applications to communicate with the database system at the same level that applications communicate with one another.
- These Web or database services have to be interconnected through a mechanism that allows for the composition of complex application systems implemented totally within the database systems.
- The database system must natively deal with the lingua franca of modern applications, XML.
- The database system must support the incorporation of arbitrarily complex business logic written by application programmers, who are not necessarily database specialists, into these database services.\
- The database system must provide support for dealing with reference data that may be required across service boundaries.
- The database system must provide the infrastructure to make these services highly available and allow them to be monitored and maintained.

Combined, these elements make up the Service-Oriented Database Architecture (SODA).

SODA is an approach to application architecture that allows for the development of highly scaleable and available database applications. By encouraging that databases be designed along logical application service boundaries, SODA makes clear how to scale the overall database system as throughput requirements rise.

An application may initially consist of several database services running on a single physical server. As requirements increase, individual database services may be moved to other servers without requiring changes in the applications or databases. SODA turns the decision about when to apply scale-up or scale-out into an economic question. As throughput requirements grow, the server hosting the database services can be scaled up, or additional servers can be added and the database services reallocated among them. Even in a very high throughput application, most individual database services will comfortably fit on a commodity (four-socket or less) server. If a particular database service has higher throughput requirements, it can be run on a larger server or designed to allow partitioning into several services.

Elements of SODA

SODA incorporates six elements that together provide a new, database application architecture. These elements can also be used individually within an application and can be combined with traditional database usage. Because scalability is the focus of this white paper, greater attention will be given to some of these elements as opposed to others.

Database Services

The notion of a database service is central to SODA and its scalability model. At a logical level, a database service exposes a well-documented application level interface to data. This is not a general database interface for reading and writing data, but instead provides very specific application functionality. For example, an inventory database service might expose methods for checking inventory levels, reserving inventory, removing products from inventory, recording receipt of new shipments, and managing back-ordered items. While this may sound similar to the stored procedures and database RPCs of the Client/Server world, there are two very important differences.

The first difference between database services and traditional models is that access to data under the control of one database service is completely isolated from access to data under the control of a different database service. For example, an Order Entry database service never directly manipulates the tables associated with Inventory. It always manipulates Inventory by calling the Inventory database service. As shown later, this makes scaling out very easy.

The second difference is that requests to database services are not made over a database connection but, instead, the services are exposed as Web Services. By breaking the traditional association of a database and its stored procedures, with a connection, it makes it possible to redistribute the services over multiple servers without requiring changes to application code. For example, in a traditional model there might be a set of stored procedures for Inventory and another set for Order Entry. To call these stored procedures, you establish a connection to a specific database server and then call both sets of procedures over that same connection. If you later want to split the Inventory and Order Entry capabilities over two database servers, you have to modify the calling application to establish two separate connections and call the stored procedures over the appropriate connection. This is not necessary with database services, because individual services can be moved without requiring application changes.

From a design standpoint, it has long been possible to think about applications in terms of services, design them to have independent data, establish independent connections for each service, and more recently even expose your stored procedures with Web Services. However, all responsibility for composing such multiple services into an application was the responsibility of the application code, with little help from the database system. What makes database services a truly practical approach is the addition of interservice communications.

Interservice Communications

A distinguishing characteristic of SODA over traditional database application architecture is a rich interservice communications mechanism for composing applications out of database services. SODA relies on an asynchronous transactional messaging system for communications between database services. This is similar to the use of message-oriented middleware between applications. However, the SODA Service Broker is deeply integrated into the database system.

The Service Broker enables one database service to make a request of another service without concern for the physical location of the other service or, in many cases, its current availability. It supports asynchronous requests where an immediate response is optional or unnecessary, as well as dialogs in which two services work together to complete a single request. It also provides a transaction model for communications that is integrated with the transaction model of the database system, but yet does not have the limitations of the traditional two-phase commit mechanism.

Consider the problem of updating data stored in two different databases with SODA and with traditional application architecture. With SODA, the Order Entry application makes a Web

Service request to the interface of an Order Entry database service to process the order. For each line item in the order, the Order Processing database service sends a message to the Inventory Management database service requesting that the inventory be reserved for the order. The Order Entry database service will also send messages to the Credit Management database service to verify the customer's account status, to Distribution to get an estimated ship date, to Shipping, and so on.

In some cases, the Order Entry database service will wait for a response and not complete the transaction until the response is received (for example, "reserve inventory"). In other cases, it will attempt to receive a response before completing the transaction, but not hold up completion (for example, "credit check"). In some cases, it will not even attempt to receive a response and will send the message as a means of kicking off an asynchronous process (for example, "fulfill this order"). However, all communications can be tied to the transaction and aborted if the transaction does not complete. For example, the send of "fulfill this order" will only be completed if the local database transaction in the Order Entry database service is committed. Contrast this with the traditional database application architecture where connections would have to be made to each individual database, all communications are synchronous, and coordination must be achieved by including all the databases in a single transaction controlled by a two-phase commit.

The SODA Service Broker allows database services to be composed into more powerful applications without the limitations of having all data reside in a single database and without the clumsy use of multiple connections and a two-phase commit. It adds further transparency to the location of the database services and provides a mechanism that can be used to implement applications that are continuously available. These points will be further explored after the other elements of SODA are described.

Native XML

XML is rapidly becoming the lingua franca of business applications, and SODA relies on having deep native support for XML. The use of XML in SODA can be shallow or deep. In the shallow case, XML is just used as an encoding for parameters in a Web Service call or a messaging service request. In the deep case, the parameter itself is a complex XML document that is to be processed. In either case, SODA requires that the database system provide extensive native facilities for dealing with XML. These facilities include the ability to parse, process, and generate XML documents and also includes facilities to directly store and manipulate XML in the database. However, this aspect of SODA is not directly related to scalability and will not be discussed further.

Complex Business Rules

Database services encourage far more business logic to be managed by the database system than has classically been done. To support Client/Server application architecture within database systems, the ability to implement business rules by using procedural extensions to SQL was added to database systems. In theory, this allowed large amounts of business rules to be implemented in the database system. However, in practice, only those rules associated with data integrity were usually implemented in this way. The reasons for this were two-fold.

First, typical application programmers generally program in languages such as Microsoft® C#®, C++®, and Visual Basic® and not in procedural SQL languages such as Transact-SQL. Second, procedural SQL provides weak support for computationally complex business rules. In SODA, both limitations are removed by supporting the use of general purpose programming languages for creating business rules inside the database system. This aspect of SODA is not directly related to scalability and will not be discussed further.

Reference Data

SODA recognizes that there are significant amounts of reference data that require extremely broad and low-cost access. Reference data is generally data that is updated, infrequently, at a central location and then used in read-only fashion at multiple locations. Examples include price lists that are updated quarterly, catalog information such as order numbers and product descriptions that may be updated daily or hourly, and business rules such as the credit guidelines for new customers. Reference data may also include read-mostly data where infrequent updates are allowed against any copy. An example of this might be the customer profile in an e-Commerce system. SODA provides support for both read-only and read-mostly reference data through the use of replication technology. This data can be accessed from within a database service or directly from middle-tier application servers.

Support for reference data in SODA can be a key enabler for scalability, as well as availability, and will be discussed further later in this paper.

Infrastructure

The SODA application architecture is practical only in light of a rich, supporting infrastructure. It must be possible to implement, debug, deploy, monitor, and maintain database services. It must also be possible to make them highly available and robust. Furthermore, it must be easy to reconfigure their topology for scalability. Because infrastructure plays only a supporting although vital role in SODA, it will not be specifically dealt with in this paper. However, some specific parts of infrastructure will be discussed in regards to scalability and availability.

SODA and High Availability

One of the major advantages achieved by applications designed with SODA is their natural ability to achieve very high levels of availability. As with traditional database servers, individual database services can be protected from hardware and software failure by using technology such as failover clusters or database mirroring. However SODA-based applications can be further designed to continue to operate even when individual services fail. This allows SODA-based applications to achieve near continuous availability. The two key SODA technologies for achieving continuous availability are the Service Broker and Replication.

SODA Service Broker

As previously described, the SODA Service Broker provides an asynchronous transactional messaging system between database services. This messaging system can be used in two ways to improve the availability of an application. First, the messaging system provides for complete transparency between services during temporary failures. For example, if a Web Order Entry service makes a request to the Inventory service while the Inventory service is in the middle of recovering from a failure (such as through a failover cluster transition), the request will transparently be responded to by the secondary server when it takes over. Even in the case of a disaster, such as a power outage leading to a failover to a mirror database in another geography, the Web Order Entry service's request will be transparently routed to the Inventory service.

The second way the SODA Service Broker can be used to achieve continuous availability is to specifically incorporate points where communication with another service can be lost for long periods of time. For example, in a traditional Order Entry system the orders are entered directly into the same database that the Order Fulfillment system uses to process the order. If this database is unavailable, both the acceptance and fulfillment of orders comes to a stop. In

a SODA-based system, the Order Entry service makes asynchronous requests to the Order Fulfillment system. If the Order Fulfillment system is unavailable, these requests will be retained in the database of the Order Entry system until communication is reestablished. Even if Order Fulfillment is unavailable for hours or days, the Order Entry system will remain fully operational. Beyond dealing with unplanned failures outside of those handled by failover clustering or mirroring, this process enables an application to continue in operation during many planned outages.

For example, the Order Fulfillment system can be taken offline for a major upgrade without affecting the ability of a business to accept orders. Likewise, a disruption to the system that is accepting orders will not affect the ability of the fulfillment system to process current orders.

Replication

As previously discussed, replication is used to make multiple copies of reference data available on various servers. This allows for multiple servers with each able to provide the equivalent database service for both scalability and availability. Using the previous example, the Order Entry part of an application is well suited for this kind of solution. By replicating catalog and customer profile information across several servers, each of those servers is capable of accepting an order from any customer. The failure of any individual server may result in the loss of the shopping carts on that server that have not yet been through the checkout process. However, the application is able to continue accepting orders through the other servers.

SQL Server 2005 Database Services

Although it is possible, conceptually, to implement databases as services with little or no support from the underlying database management system, they only become truly practical when the underlying system provides rich support. SQL Server 2005 is the first database management system to focus on providing rich support for databases as services. Specifically, the extensive XML support and new Service Broker of SQL Server 2005 provide the backbone for creating database services. Support for the Microsoft® .NET Framework provides the rich environment necessary to implement complex business rules, and the replication enhancements provide improved support for reference data. Furthermore, new features, such as peer-to-peer transactional replication, provide superior support for scaling out read-mostly data. Finally, these capabilities are available along with the other scalability, availability, programmability, and manageability improvements of SQL Server 2005.

XML Support

SQL Server 2000 provided limited support for XML and XML Web services, primarily through a midtier component called SQLXML. SQL Server 2005 natively provides extensive support for XML in the database engine. This includes support for native XML Web Services access, a native XML data type, and XQuery support.

Exposing Database Services

One of the major requirements for implementing SODA is the ability to expose the database as a set of Web Services. SQL Server 2005 supports this requirement with its implementation of Native HTTP SOAP Access. The Native HTTP SOAP Access allows direct access to SQL Server 2005 Web Services without the need for a midtier IIS server, and without the need for client-side database software. Any application capable of invoking a Web Service, any

development toolset that supports generating calls to Web Services, and any platform (such as Windows, UNIX, and Linux) can access SQL Server 2005 Web Services without any special programming.

Processing XML

Because a database service is exposed as a Web Service, it generally is expected to deal with XML documents. SQL Server 2005 provides extensive native support for XML. Specifically, a new XML data type allows for native storage of XML in the database. The XQuery language is provided for querying data stored in the XML data type and is extended with the ability to insert, update, and delete XML documents or fragments. The FOR XML and OPENXML Transact-SQL support is provided for transforming data between XML and relational forms. Combined, these capabilities allow a database service to deal with the external world in XML while storing and manipulating data internally in the most appropriate form.

Service Broker

The key to scaling with SODA is the ability to compose an overall application system from a set of database services. The SQL Server 2005 Service Broker was designed to provide a rich interservice communications mechanism for SODA. Service Broker provides a reliable, persistent, transactional, and asynchronous communications facility for SQL Server. Service Broker also provides location transparency. This allows the calling application to be unaware if a message will be processed locally, in another SQL Server instance on the same server, or on a remote server.

Microsoft .NET Framework

SODA encourages the implementation of complex business logic in the database service. Transact-SQL is one choice for implementing business logic. SQL Server 2005 provides a second choice by hosting the Microsoft .NET Framework Common Language Runtime (CLR). This allows business logic to be implemented in traditional programming languages such as C#, J#, and Visual Basic .NET.

Replication

One of the major ways to improve scalability is to have multiple copies of reference data. SQL Server 2005 replication provides the tools for maintaining multiple copies of read-only (such as catalog data) and read-mostly (such as customer profile) data. One new feature in particular is the peer-to-peer transactional replication. This provides excellent support for read-mostly data by allowing updates to be directed to any copy of the data. This has the added benefit of allowing any copy to be taken offline for maintenance (such as software patching) without disturbing overall system operation.

Scalable Application Example

This section examines an application example that takes advantage of the SQL Server 2005 Service-Oriented Database Architecture to create a subset of an n-tier, e-Commerce Web application. This module performs the order acceptance and order processing tasks. The goal is to design the application so that it facilitates extensibility, scalability, and integration, while effectively serving the core business requirements.

The following are the business requirements for this module:

- Accept orders
- Process orders (credit card processing, inventory, distribution, and shipment)
- Read customer profile
- Read product catalog
- Send e-mails confirming the order and the order progress reports
- Allow other applications and systems to query inventory levels
- Shorten the interactive response time as much as possible during order acceptance
- Maximize the performance by caching parts of customer profile and product catalog data

Before looking at the SQL Server 2005 SODA-based solution, the following diagram illustrates how this functionality is achieved with SQL Server 2000:

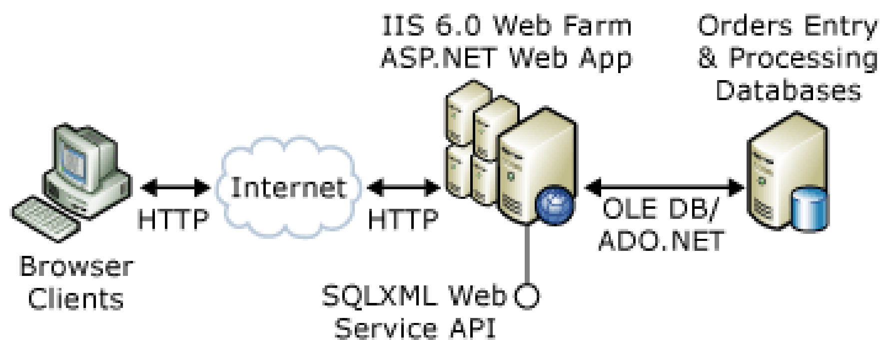

Figure 1: Order accepting and order processing module using SQL Server 2000: Design 1

As illustrated in Figure 1, the browser-based client sends requests over HTTP to IIS 6.0 Web servers, which host an ASP.NET Web site to accept orders. The ASP.NET code uses ADO.NET to communicate with the order entry and order processing a SQL Server 2000 database. The SQLXML 3.0 Web services support is used to expose the inventory level querying API. The ASP.NET code implements caching to reduce roundtrips to the database server for customer profile and product catalog details. SQL Mail is set up on SQL Server and sends e-mails confirming order submission and order progress reports.

Following are some of the limitations of this kind of architecture:

- The order entry and order processing databases are hosted on a single server, limiting the application's ability to scale. The only option available is to scale up. In addition, this approach creates a single point of failure. If SQL Server is offline, the Web site will not be able to even accept orders.
- SQLXML is used to expose inventory-level querying API by mapping Web service methods to existing stored procedures. SQLXML places an additional setup and maintenance requirement. ASP.NET could be used to provide this API. However, it requires writing and maintaining additional code.
- SQL Mail requires a MAPI client to be installed on the SQL Server computer. In addition, SQL Mail is not scalable.
- The ASP.NET code caches parts of the product catalog and customer profile as it is accessed. The current technologies do not offer a solution to automatically refresh the cache when the data changes. The ASP.NET code has to either periodically refresh the cache or poll the database for changes, or implement a custom solution that indicates that the cached data has changed.

The order processing can be separated onto a different server. However, that can require enlisting in two-phase commit and implementing distributed transactions. Customer profile and product catalog has to be accessible to both the servers. One solution to this is to keep the data on one server and use a linked server and distributed queries from a second server to access the data. A better approach is to keep a local copy of data on both of the servers and use transactional replication to replicate customer profile and product catalog data. However, transactional replication is not well suited for read-mostly data. This approach is illustrated in Figure 2.

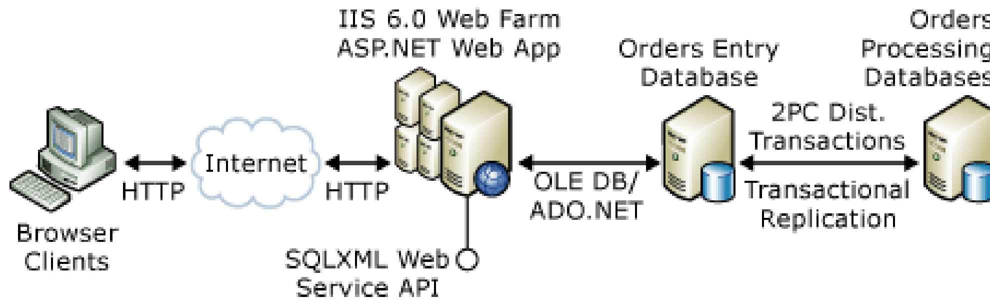

Figure 2: Order accepting and order processing module using SQL Server 2000: Design 2

Now consider how SQL Server 2005 addresses the previously described limitations and also allows the creation of scalable and loosely coupled database applications.

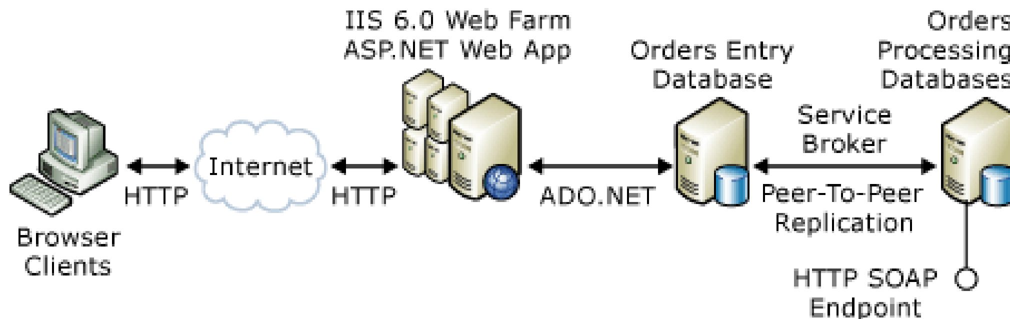

Figure 3: Order accepting and order processing module using SQL Server 2005 SODA

As illustrated in Figure 3, the order entry and order processing now is split between two servers, thus making it a more scalable solution. In addition, this design increases the availability, because even if the order processing server is down, the Web site can continue to accept the orders.

When an order is submitted, an entry is made to the order entry database. At the same time, a message is placed in a Service Broker queue. The order entry process ends here. This queue message is then routed to the SQL Server 2005 server hosting the order processing databases. If the order processing server is offline, the messages are held in the queue at the order entry database until the order processing database becomes available.

When a message is received by the order processing database, various tasks such as inventory, distribution, and shipping are performed. Finally, a response message is sent back to the order entry server using Service Broker, which then uses SQLiMail to send the order confirmation e-mail. This decoupling of tasks facilitates scalability and allows for updating and upgrading the functionality individually.

SQLiMail uses Service Broker, making it a scalable e-mailing solution. Alternatively, SQL Server Notification Services can be used to send e-mails. The use of Service Broker for asynchronous messaging across servers eliminates the need to use distributed transactions and two-phase commits. Furthermore, Service Broker aids in scaling out the architecture and

in reducing the interactive response time. This architecture increases the availability, eliminates the need for an external messaging solution or product, and simplifies implementing the distributed database applications.

The order processing server uses HTTP SOAP endpoints to expose API for query inventory levels. HTTP SOAP endpoints eliminate the need for SQLXML and IIS, and also provide better performance by eliminating extra IIS and SQLXML layers.

The new peer-to-peer transactional replication is set up between the order entry and order processing databases to duplicate the customer profile and product catalog data. Peer-to-peer transactional replication is superior and performs better for read-mostly data compared to traditional transactional replication.

The ASP.NET Web site code uses the new Query Notification mechanism to refresh the cached product catalog and customer profile data. Query notification uses Service Broker to asynchronously notify the subscribed ASP.NET code about the change in the database.

The previous design is loosely coupled and supports standards-based Web services facilitating integration and extensibility. The use of Service Broker, Query Notification, and SQLiMail makes it a solution that can scale well when the load increases. The use of peer-to-peer replication offers superior solution for sharing read-mostly product catalog and customer profile data. In addition, wherever applicable, the previous design can take advantage of XML data type and XQuery functionality to natively store and query unstructured data. It can also use SQLCLR to extend the type system or aggregate functions, or to implement complex business rules.

Conclusion

The Service-Oriented Database Architecture provides for the construction of highly scaleable, highly available databases. By moving the transparency boundary to the service level, SODA avoids the scaling limitations of SQL statement-level scale-out solutions. With its highly reliable interservice communications mechanism, SODA supports the design of databases that can achieve near-continuous availability.

Although this paper has focused on SODA as the basis for achieving scalability, SODA offers far more than that. It is an architecture well suited for integrating legacy and new database applications. Furthermore, it is an architecture that allows for better evolution to support new business requirements by “rewiring” the various services, along with new database services, to meet those requirements.

These attributes make SODA the ideal architecture for building scalable SQL Server 2005 database applications.

For more information:
http://www.microsoft.com/sql/

Did this paper help you? Please give us your feedback. On a scale of 1 (poor) to 5 (excellent), how would you rate this paper?

Integrated and manageable server software products designed to reduce IT complexity and total cost of ownership so you can focus your resources on other priorities for you and your business.

www.microsoft.com/windowsserversystem

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 OR IMPLIED, 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 per 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.

© 2005 Microsoft Corporation. All rights reserved.

The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

Microsoft, C++, C#, J#, and SQL Server are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

All other trademarks are property of their respective owners.