Methodology for Implementing Homogeneous Federated Database Systems Using Microsoft SQL Server

Cyrus Azarbod
Professor
Computer & Information Sciences Department
Minnesota State University at Mankato,
273 Wissink Hall, Mankato, MN 56001
Cyrus.azarbod@mnsu.edu

Jake Roerig
Software Engineer
McKesson Corporation
Minneapolis, MN

Abstract

It is quite common for e-Commerce Web sites and enterprise data-processing systems to accumulate hundreds of millions of records of data over time. If all of these records need to be readily available at all times, over time a single database server will become over-burdened and response time will suffer.
An approach to solving this problem is to scale out horizontally by partitioning data across multiple servers. When data is distributed over multiple servers in a network, this becomes what is known as a distributed database environment. In a distributed database environment, a federation of databases is made up of two or more database servers who function autonomously, but agree to share data and processing for a larger workload.
A federation of databases might be a versatile solution to this problem by scaling out the data and processing to handle the growth requirements of extremely large databases.
In this paper, a methodology is presented for designing and implementing homogeneous federated databases in SQL Server. This methodology includes design patterns for federated databases, how to implement a federation, how to design applications to use a federation, how to ensure availability, and how to handle federation maintenance.
In this methodology we are proposing eleven steps to implement a homogeneous federated database. The steps are as follows:
Profile Database, Design Partitions, Setup Servers, Setup Linked Servers, Setup Security, Set Lazy Schema Validation Server Option, Partition Data, Define CHECK Constraints, Define Data Dependent Routing Table, Create Distributed Partitioned Views, and Setup Replication.

Background Overview

A distributed database is defined as a collection of multiple, logically interrelated databases distributed over a computer network (Ozsu & Valduriez, 1999). A distributed database management system (DDBMS) is then defined as the software system that permits the management of the distributed database and makes the distribution transparent to the users (Ozsu & Valduriez, 1999).

Federated Databases

Federated databases are a particular variety of distributed database architecture. They are semiautonomous meaning they can best be described as database systems that operate independently, but also agree to cooperate with other systems in processing user requests and sharing local data.

Fragmentation

Fragmentation describes how data is partitioned across systems in a distributed environment (Ozsu & Valduriez, 1999). The goal of fragmentation is to limit the total execution time of user requests. Since joining fragments from different sites can be expensive, mutually exclusive fragments should be chosen.
Mutually exclusive fragments also promote improved performance and concurrency. When a user request is made of the distributed system, that query is broken down into sub queries. Those sub queries are run in parallel on each server which holds one or more of the requested fragments. This parallel execution decreases the total execution time of the request, which again is the goal of fragmentation. The challenging aspect of fragmentation is determining how to fragment. In order to fragment data, an appropriate unit of fragmentation must be chosen. The choices of fragmentation units are relation, attribute, and tuple.

Distributed Query Processing

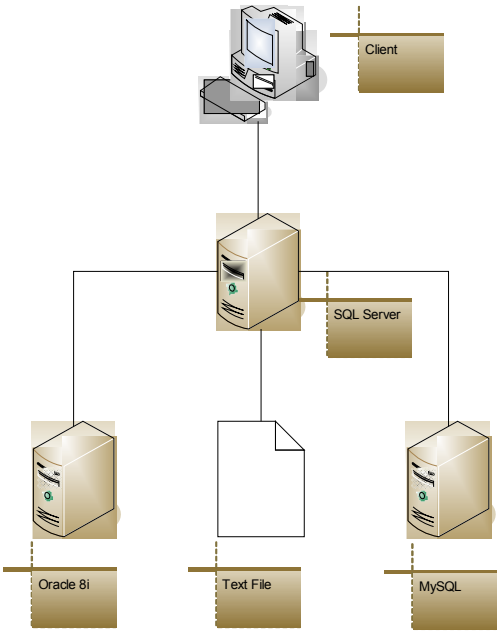
The objective of distributed query processing is to transform the user request, which appears as a single database to the user, into efficient local queries executed on the relevant partitions (Ozsu & Valduriez, 1999). In order to perform distributed queries in a federation, each member server must be able to communicate with one another, security must be enforced, transactions must be managed to preserve database integrity, and functions must be available by which to invoke distributed queries. One mechanism is called linked sever, which allows connections between participating servers in the federation.

Linked Servers

In order to perform distributed queries in a federated environment, each member server must be aware of the other participating member servers, and have the means to communicate with them. Therefore each member server must have a virtual server definition for each of the other member servers in the federation. In SQL Server 2000, this virtual server definition is called a Linked Server. Since linked servers can be established with any OLE DB or ODBC data source, usually they are created in

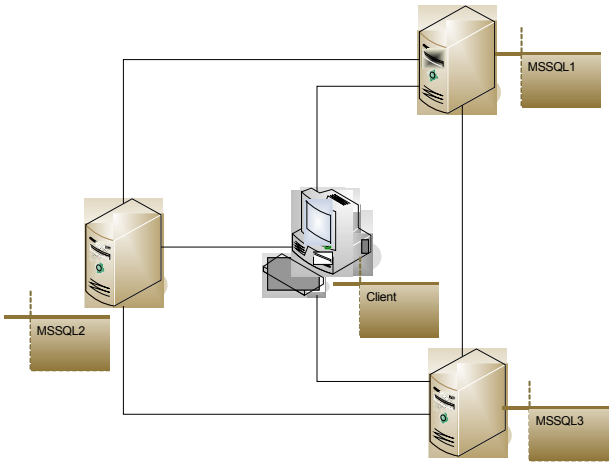
heterogeneous environments (see Figure 1). In a homogeneous environment with only SQL Server systems, they can be used to form federations (see Figure 2).
In the heterogeneous environment a client may only connect to the SQL Server a system, which then initiates communication with the other data sources. The data locations are transparent to the client, but there are no performance benefits since all connections and initiations must be made through the SQL Server system.

Figure 1: Heterogeneous Distributed Environment



In the homogeneous environment, the client may connect to any one of the SQL Server systems, which can then initiate communication with the others. The data locations are still transparent to the client, and there can be performance benefits by distributing the data and processing workload among the servers forming a federation.

Figure 2: Homogeneous Distributed Environment



Linked servers can be created, modified, and deleted using either Enterprise Manager or system stored procedures. In addition to those operations, security between the local server and its linked servers can be managed using Enterprise Manager or system stored procedures. Distributed queries can be used to retrieve and join remote data with local data, as well as perform all the other CRUD operations (create, retrieve, update, and delete). Remote data can only be used in a distributed query if the software managing the data supports an OLE DB provider which can expose the data in tabular objects called rowsets (Sheldon & Wilansky, 2001). SQL Server is able to treat these rowsets as though they were tables. There are two types of distributed queries: remote tables and pass-through queries. Remote tables are remote objects that are directly referenced in the query. When referenced, the entire object is returned to the local server for processing. Pass-through queries are queries where SQL syntax is passed to a remote server for processing where a rowset is generated and sent back to the local server for processing.

Four-Part Names

A distributed query that directly references a remote table must contain enough information to locate a particular object on a remote server. Locating a remote object can be achieved by using a Four-Part Names reference.

The four-part names reference has the following syntax (Sheldon & Wilansky, 2001):

linked_server_name.catalog.schema.object_name

The following example shows a join of a remote table with a local table using four-part naming.

```
SELECT e.lname, e.fname, b.city, b.state
FROM employee e
    INNER JOIN samwise.oes.dbo.branch b ON e.branch_no = b.branch_no
WHERE e.employee_no = 1000
```

The code for four-part naming is easy to use and follow, but since the entire remote object must be returned to the local server for processing there may be performance issues. This approach to distributed queries works well for small remote objects, but performs poorly

for large remote objects (exception to this rule is through the use of distributed partitioned views)

OPENQUERY() Function

A pass-through distributed query sends a query to a remote server to generate a rowset to be sent back to the local server. Since the query is passed to the remote server where a rowset is generated, there is opportunity to let the remote server handle some of the processing. For example, a subset of a join of two tables is needed from a remote database, the distributed query will most likely perform better if the join takes place on the remote server and only the resulting rowset be sent back instead of sending both remote tables to the client for joining.

The OPENQUERY() function has the following syntax (Sheldon & Wilansky, 2001):

OPENQUERY(*linked_server*, 'query')

The following example shows a join of a rowset generated by a pass-through query joined to a local table.

```
SELECT e.lname, e.fname, b.city, b.state
FROM employee e
        INNER JOIN OPENQUERY(samwise, 'SELECT city, state FROM branch') b
        ON e.branch_no = b.branch_no
WHERE e.employee_no = 1000
```

The syntax for OPENQUERY() is a little more difficult than that of four-part naming, however these types of queries usually perform better especially when dealing with large objects.

OPENROWSET() Function

The previous two ways of performing distributed queries required linked server definitions to connect to remote servers. However, some distributed queries may be so infrequently run there's no need to create a linked server. In those cases, ideally queries could specify remote server connection information in an ad hoc way to generate a single rowset. This type of pass-through query uses the OPENROWSET() function.

The OPENROWSET() function has the following syntax (Sheldon & Wilansky, 2001):

OPENROWSET **(** '*provider_name*'
, { '*datasource*' **;** '*user_id*' **;** '*password*' | '*provider_string*' }
, { [*catalog***.**] [*schema***.**] *object* | '*query*' }**)**

The following example shows a rowset generated by an ad hoc pass-through query joined to a local table.

```
SELECT e.lname, e.fname, b.city, b.state
FROM employee e
        INNER JOIN OPENROWSET('MSDAORA', 'oracle2.cs.mnsu.edu'; 'scott';
        'tiger', 'SELECT city, state FROM branch') b ON e.branch_no = b.branch_no
WHERE e.employee_no = 1000
```

This approach does not perform as well as OPENQUERY(), and there may be security risks associated with placing login information in the query itself, but it does satisfy the need of infrequent ad hoc queries.

OPENDATASROURCE() Function
The last variety of distributed query directly references remote tables using four-part names, however the query is run so infrequently it connects to the remote server in an ad hoc manner by specifying the connection information in the query itself. The is achieved by using the OPENDATASOURCE() function. The OPENDATASOURCE() function is similar to the OPENROWSET() function, except it references remote tables directly using four-part names instead of pass-through, and it can return multiple rowsets rather than just one. The OPENDATASOURCE() function has the following syntax (Sheldon & Wilansky, 2001).

OPENDATASOURCE(*provider_name*, *init_string*)

The following example shows a remote table joined with a local table by using an ad hoc query with four-part naming.

```
SELECT e.lname, e.fname, b.city, b.state
FROM employee e
        INNER JOIN OPENDATASOURCE('MSDAORA', 'Data
Source=oracle2.cs.mnsu.edu; User ID=scott; Password=tiger') .oes.dbo.branch b
        ON e.branch_no = b.branch_no
WHERE e.employee_no = 1000
```

Partitioning Data

How data is partitioned greatly affects the performance and maintainability of the federation. There are two different ways to partition data, horizontal and vertical, which can lead to two different types of partitions, symmetric and asymmetric. Based on current database usage, there are many design considerations when partitioning data.
Once partitions are formed, a mechanism is needed to span the partitions to give the appearance of a single server to the user or application. The application must also be augmented in such a way as to allow it to connect to any server in the federation; hence the load balancing. As data and usage patterns change, partition maintenance becomes important to the continued balance of the federation. The federation should also provide high availability; this is accomplished through failover clustering.

Methodology for Design and Implementation of a Federated Database

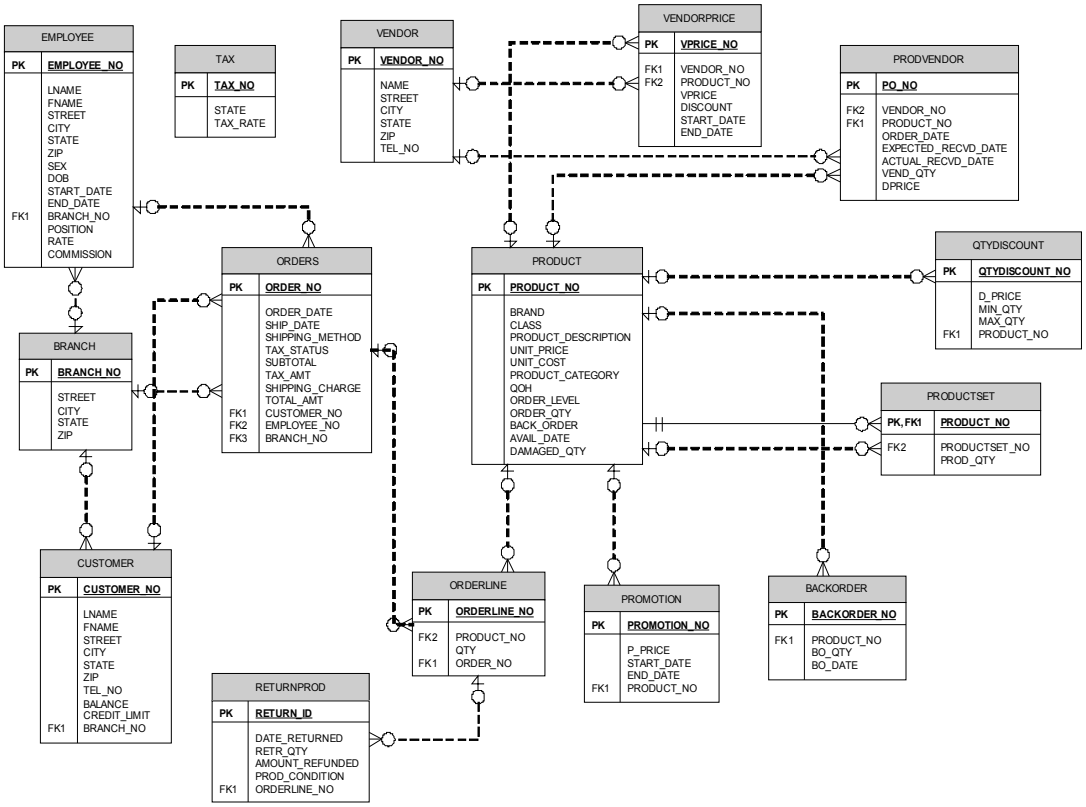
The steps needed to implement a federation follow:

- Profile Database
- Design Partitions
- Setup Servers
- Setup Linked Servers
- Setup Security
- Set Lazy Schema Validation Server Option
- Partition Data
- Define CHECK Constraints
- Define Data Dependent Routing Table
- Create Distributed Partitioned Views
- Setup Replication.

Before proceeding with the design and implementation of a federation, the current database and applications should be optimized. Some optimization techniques to consider are the use of stored procedures, indexing, improving application logic, normalization and de-normalization of the logical data model where appropriate, and rearrangement of the physical data model. The results of using these and other techniques may yield that it is not necessary to scale out to a federation. Scaling out should be the last optimization technique used.

To discuss the proposed methodology, an order entry database consisting of 15 tables is used. The ER-diagram for the OES database is shown in Figure 3.

Figure 3: OES Database



Step 1: Profile Database

The first step in creating a federation is to profile the un-partitioned database. Assume the following tables are identified as having greater than 5% of queries as inserts, updates, and deletes: orders, customer, orderline, and returnprod. These tables are good candidates for partitioning. The remaining tables are good candidates for replication.

Step 2: Design Partitions

Once the tables eligible for partitioning have been identified, a partitioning scheme must be chosen. After careful examination of the database profile, and current application usage of the database, assume usage can be divided along branch number.
Currently there are two branch numbers, 100 and 111, and usage can be nearly evenly divided between the two. Of the tables eligible for partitioning, orders and customer both contain the branch_no column, and therefore will be easy to partition. The other eligible tables, orderline and returnprod, do not contain the branch_no column, however orderline is a child of orders and returnprod is a child of orderline, therefore they can easily be partitioned by their foreign keys.

Step 3: Setup Servers

It has been identified that the federation will have two partitions, therefore two servers are needed. Each server in the federation should be roughly equivalent in terms of hardware and software since each is responsible for an equal share of the total workload.

Step 4: Setup Linked Servers
In order for the two servers to communicate with one another, each requires a linked server definition for the other.

Step 5: Setup Security
When users attempt to access data residing on different servers, they must be authenticated on each server. This is managed through the use of linked server logins.

Step: 6 Set Lazy Schema Validation Server Option
To limit meta data requests between servers until the data is needed, the lazy schema validation server option must be set.

Step 7: Partition Data
To physically partition the data between the member servers in the federation, ideally a job should be created which can periodically run to keep the federation balanced. This can be achieved via a DTS package.

Step 8: Define CHECK Constraints
Once the data has been partitioned, constraints need to be placed on the tables in each partition to enforce the partitioning scheme. This is accomplished through the use of CHECK constraints.

Step 9: Define Data Dependent Routing Table
All data dependent routing information is stored in a routing table. This table is used as a reference for partitioning and repartitioning data, and is used to route applications to appropriate partitions.

Step 10: Create Distributed Partitioned Views
Distributed partitioned views are the principle mechanism by which the data in the federation is combined.
The following scripts create two distributed partitioned views, one unions the orders table, the other unions the customer table.

```
--executed on server FRODO
CREATE VIEW v_orders AS
SELECT order_no, order_date, ship_date, shipping_method, tax_status, subtotal,
        tax_amt, shipping_charge, total_amt, customer_no, employee_no, branch_no
FROM orders
UNION ALL
SELECT order_no, order_date, ship_date, shipping_method, tax_status, subtotal,
        tax_amt, shipping_charge, total_amt, customer_no, employee_no, branch_no
```

```
FROM samwise.oes.dbo.orders

CREATE VIEW v_customer AS
SELECT customer_no, lname, fname, street, city, state, zip, tel_no, balance,
credit_limit, branch_no
FROM customer
UNION ALL
SELECT customer_no, lname, fname, street, city, state, zip, tel_no, balance,
credit_limit, branch_no
FROM samwise.oes.dbo.customer

--executed on server SAMWISE
CREATE VIEW v_orders AS
SELECT order_no, order_date, ship_date, shipping_method, tax_status, subtotal,
        tax_amt, shipping_charge, total_amt, customer_no, employee_no, branch_no
FROM orders
UNION ALL
SELECT order_no, order_date, ship_date, shipping_method, tax_status, subtotal,
        tax_amt, shipping_charge, total_amt, customer_no, employee_no, branch_no
FROM frodo.oes.dbo.orders

CREATE VIEW v_customer AS
SELECT customer_no, lname, fname, street, city, state, zip, tel_no, balance,
credit_limit, branch_no
FROM customer
UNION ALL
SELECT customer_no, lname, fname, street, city, state, zip, tel_no, balance,
credit_limit, branch_no
FROM frodo.oes.dbo.customer
```

In order for the distributed partitioned views to support the full behavior of the underlying tables (insert, update, and delete operations), INSTEAD OF triggers need to be defined on the views.

Step 11: Setup Replication

The tables that were not partitioned were instead replicated on each member server. To maintain exact copies on each server, either SQL Server Replication can be specified, or INSTEAD OF triggers can be defined on each table.

```
CREATE TRIGGER t_taxinsert ON tax INSTEAD OF INSERT
AS
DECLARE @state AS CHAR(2), @tax_rate AS DECIMAL(5,5)
SELECT @state = state, @tax_rate = tax_rate
FROM INSERTED
BEGIN DISTRIBUTED TRANSACTION
```

```
        INSERT INTO tax(state, tax_rate) VALUES(@state, @tax_rate)
        INSERT INTO samwise.oes.dbo.tax(state, tax_rate) VALUES(@state,
@tax_rate)
COMMIT TRANSACTION
```

Conclusion

An alternative to scaling up is scaling out. Single database servers eventually become bottlenecked with increasingly large transaction databases. Rather than continually upgrading a single database server, adding new servers can linearly increase the processing power of the system (Gray & Waymire, 2004). SQL Server 2000 offers the functionality to satisfy the requirements of scaling out with a federated architecture. This functionality makes it possible to partition data across multiple servers while maintaining data location transparent to users. Some extensions of this research have been left for future work. One such extension is a more robust implementation where a Partitioning Agent automatically maintains federation balance without database administrator intervention. Another extension is the implementation of a failover cluster accompanied by a study of high availability. The methodology presented in this paper could also be modified in such a way to support a heterogeneous federation where other database management systems participate in the federation**.**

References

1. Gray, J., & Waymire, R. SQL Server Megaservers: Scalability, Availability, Manageability. Microsoft SQL Server TechCenter. Retrieved April 20, 2004, from,http://www.microsoft.com/technet/prodtechnol/sql/2000/plan/ssmsam.mspx?pf=true

2. Microsoft Corporation. Federated SQL Server 2000 Servers. MSDN Library. Retrieved December 30, 2004, from http://msdn.microsoft.com/library/en-us/architec/8_ar_cs_4fw3.asp

3. Microsoft Corporation. Designing Applications to Use Federated Database Servers. MSDN Library. Retrieved December 30, 2004, from http://msdn.microsoft.com/library/default.asp?url=/library/en-us/acdata/ac_8_qd_10_48oj.asp

4. Oracle Corporation. (2002). Database Architecture Federated vs. Clustered. Retrieved March 2, 2004, from http://www.oracle.com/technology/tech/windows/rdbms/ClusterComp.pdf

5. Ozsu, T. M., & Valduriez P. (1999). *Principles of Distributed Database Systems* (2nd ed.). New York, NY: Prentice Hall.

6. Sheldon, R., & Wilansky E. (2001). *MSCE Microsoft SQL Server 2000 Database Design and Implementation Training Kit*. Redmond, WA: Microsoft Press