CERIA Res. Rep. 2004-03-28, March 2004

Implementing SD-SQL Server: a Scalable Distributed Database System

(Extended Abstract)

Witold Litwin, Soror Sahri[1]

Abstract: SD-SQL Server is a scalable distributed DBS using internally the SQL Server. The relational tables of SD-SQL Server scale through splits transparently for the application. SD-SQL Server is the only DBS with this capability at present. It constitutes an important step beyond the current technology of a parallel DBMS, long awaited by the users. The splitting and addressing principles of our system follow those of a Scalable Distributed Data Structure. We present the 1st implementation of SD-SQL Server and experimental performance analysis proving its efficiency.

Key words: Scalable Distributed DBS, Scalable Table, Distributed Partitioned View, SDDS

1 Introduction

The explosive growth of the volume of data to store in databases makes them often huge and permanently growing. This evolution requires new DBS architectures, effective for scaling databases. The proposal of Scalable Distributed Data Structures (SDDSs) addressed similar challenge for storage and file systems [5], [6]. In [1], the concept of a Scalable Distributed DBS (SD-DBS) was derived for databases. A specific SD-DBS termed SD-SQL Server was proposed for the SQL Server.

In short, a *scalable* table T of an SD-DBS in general, and of SD-SQL Server in particular, is dynamically and transparently for the application partitioned into some *segments*. This capability is crucial for the scalability and respond to a long awaited users need, [3]. It lacks to the current technology of a parallel DBMS, offering the static partitioning only. Each segment of a scalable table resides in some *segment DB*. Each segment DB is typically at a different SD-DBS *server* node. A high-speed network links the nodes. Each T typically starts as a single segment s_T. The SD-SQL Server application does not address the segments directly, but sees only the SQL Server distributed partitioned (union all) view of T. The SD-SQL *client* node manages each such *scalable* view V_T that always presents T as a single table. The scalable view has the role of the client image in an SDDS. It supports searches and updates.

An SD-SQL server checks the size of its s_T using a trigger, when an insert into s_T occurs. When s_T scales beyond some size of b tuples, fixed or node dependent, s_T splits. Half of the tuples migrate to a new segment s_T' with the schema of T, at a new node. The process typically makes any existing client view V_T outdated. It does not address all the existing segments anymore. The SD-SQL Server checks any V_T for its correctness in this sense, when a query to V_T occurs. The query activates for this purpose a dedicated trigger that further performs the adjustment if needed. This adds the missing segments to V_T prior to the query execution.

The whole mechanics allows for the scalable tables without forcing the rewriting of the SQL query optimizer. That task would be most likely daunting. The research problem we addressed since [1] was the prototype implementation of the above principles. This requires the design of some meta-tables, triggers, and stored procedures. The design should be tuned so that the overhead of segment size testing, and splitting, as well as of view correctness testing and adjustment, remain tolerable.

[1] Centre d'Etudes et de Recherches en Informatique Appliquée (CERIA), University Paris Dauphine, 75016 Paris, France. *Witold.Litwin@dauphine.fr*, *Soror.Sahri@dauphine.fr*

Submitted to WDAS-04

We describe our SD-SQL Server implementation in Section 2. In Section 3, we discuss the experimental performance analysis. The measurements show our implementation scalable and efficient. We conclude in Section 4.

2 SD-SQL Server Implementation

Figure 1 shows the gross architecture of the SD-SQL Server. At each node there is an SQL Server and an SDDS layer. Each SQL server manages the segment database that stores the local segments and partitioned views. The SDDS layer manages the views and segments so to make them scalable as above described. It behaves here as an SDDS client or server or both (a peer). As the client, it manages each scalable table *T*. It first allows the application to create *T*. This involves the creation of V_T and of 1st segment s_T. It stores both in the local SQL Server. It then manages queries to its view V_T as above described. As the server, it oversees the size b_s and performs the splits of the local segment s_T. It manages similarly the segments of other scalable tables created initially by clients at other server sites. As the peer it performs both functions.

2.1 Server Side

2.1.1 Meta-tables

SD-SQL Server gets queries from the application and passes them to its SQL Server for execution. Every split occurring on the server changes the current partitioning which is stored in meta-tables. The meta-tables are at each server and represent the actual image of a partition. To describe these, let $D_{i;\ i=1,2..}$ be the DBSs under *SD-SQL* servers storing the segments of a scalable table *T*, initially created in database D_i. Then, as defined in [1]:

- The SD-RP (DB-S, Table) describes the actual partitioning of each *T*. The tuple (D_i, T) enters D_i.*SD-RP* each time a segment of *T* is created at D_i.
- The SD-S (Table, S-max) fixes the maximal size of a segment for each table at the site.
- The SD-C (DB-T, Table, S-size) contains the tuple (D_i, T) at each server storing a segment s_T.
- The *SD-Site* table includes the servers a available for *T* segments.

The meta-tables are updated when any *T* is created or splits. We now describe *T* creation more in depth.

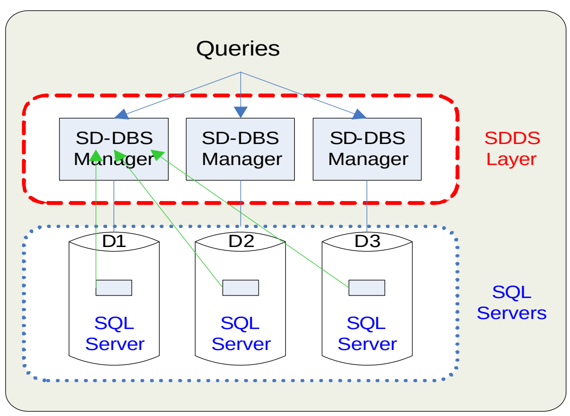

Figure 1 Architecture of SD-SQL Server

2.1.2 Scalable Table Creation

The creation of a scalable table *T*, is similar to that of any table. Nevertheless, we use a different statement and have an optional clause related to the segment size, specific to a scalable table. More precisely, the application calls instead of traditional *CREATE TABLE* statement, a stored procedure termed *create_scalable_table*.

Submitted to WDAS-04

The procedure transparently executes *CREATE TABLE* statement, but also calls a number of other distributed stored procedures with the respective goals. These are:

- Creation of a check constraint to the partitioning attribute, i.e. the primary key. The use of check constraints allows the query to be redirected to the node where it should reside.
- Insert of the size b_s into the *SD-S* meta-table.
- Creation of the trigger on *T* calling the *split_table* stored procedure. That one splits *T* when it exceeds b_s tuples.
- Insert of the tuples *(D_i, T)* into *SD-RP* and *SD-C* meta-tables respectively.
- Creation of the partitioned view V_T of *T* on server D_i.

Example 1

We create the scalable table *Customer* on *Server1.DB1*database. Let C*ustomerid* be its only attribute, hence the primary key (it is the only attribute we need for performance measurements below). We suppose that the capacity of *Customer* segment is *b* = 100 tuples at any segment DB. We call the stored procedure *create_scalable_table* with input parameters: which are the string that contains the SQL *Customer* table creation statement, and *b* value. We execute the procedure using the SQL Server *EXEC* command at *Server1.DB1*:

EXEC create_scalable_table 'CREATE TABLE Customer (Customerid numeric PRIMARY KEY)', 100

This stored procedure will:

- Create the table *Customer* as a single segment at the local site *Server1.DB1*. Once the table is created, a trigger that launches the split is added to it.
- Insert into *Server1.DB1.SD-S* the maximal size of *Customer* segment, i.e. 100.
- Insert the tuples (*Server1.DB1*, *Customer*) and (*Server1.DB1*, *Customer*) into *Server1.DB1.SD-RP* and *Server1.DB1.SD-C* respectively.
- Create the partitioned view *Customer_view* on *Server1.DB1*.

If we wanted to create *Customer* with more attributes, we would enumerate them in the *CREATE TABLE* statement, as usual for Transac SQL.

2.1.3 Split Mechanism

Each time $D_i.T$ exceeds the maximal segment size in *Di.SD-S*, a split results. First, a new database server is selected from the *Di.SD-Site* for the new segment of *T*. The selected site should be among the SQL Servers linked to the current one. The split partitions *T* at *Di* and creates a new segment with the same schema at a new node; let it be D_{i+1}. Half of *T* tuples migrates from the split segment to thtat on the new node D_{i+1} [1]. Once the *SD-DBS* server completes this process, it alters the check constraints of the segment *Di.T* and updates the meta-tables.

2.2 Client Side

The client side of SD-SQL Server manages the already presented views of scalable tables. We call them *scalable (distributed)* views. Any scalable view V_T of a table *T*, is an SQL Server distributed partitioned view of *T*, with the additional capability of the dynamic adjustment to *T* partitioning, as resulting from the splits. We recall that distributed partitioned views of SQL Server are the range partitioned (union all) views. The ranges are defined by the check constraints at the segment DBs. The views support searches inserts and updates. This capability makes the tool crucial for SD-SQL Server design.

The default naming convention for the current implementation of SD-SQL Server is to call V_T as T_view. Assuming that one creates *T* at D_i DB, each V_T is defined initially as:

CREATE VIEW T_view AS
SELECT *FROM Di.T

Once $D_i.T$ splits, creating new segment $D_{i+1}.T$, the definition of *T_view* should be adjusted. It should become the SQL Server distributed partitioned view including $D_{i+1}.T$ segment. According to the general principles of an SDDS, the client checks the view correctness asynchronously. That is the servers performing the splits do not adjust any views. Such approach could be highly ineffective. Only the client performs this operation when a query addresses *T*. If a query involving *T* follows the *T* first split, the *T_view* definition becomes, [3]:

CREATE VIEW T_view AS
SELECT *FROM Di.T
UNION ALL
SELECT * FROM D_{i+1}.T

To adjust the scalable view, the client uses *C-Image (Table, Size)* meta-table. When a table *T* is initially created at *Di*, the tuple *(T, n)*, where *n* is the number of *T* segments in the federated view, is inserted into *Di.C-Image*. The *SD-DBS* client compares *n* corresponding to *Di.T* with the number of segments of *T* in *Di.SD-RP*, let it be *n'*. If *n<n'*, the view is adjusted to include all the *n'* segments of *T* in *Di.SD-RP*. Once the view is adjusted, the SD-SQL Server passes the query to the SQL Server for execution [1].

3 Performance Analysis

To prove the scalability and efficiency of the *SD-SQL* Server, we made a series of experiments. The goal was to determine the overhead at the servers and clients with respect to SQL Server. At the servers we measured the split time. At the client, we measured the overhead of a scalable view management during a query, i.e., of the view checking and, perhaps, of adjustment. We experimented with search and insert queries. For all the experiments, we used the *Customer* table of Example 1. We measured the timing of the operations using the SQL Server Profiler. The hardware consisted of 1.8 GHz P4 PCs, connected through 1Gbs Ethernet.

3.1 Server Side

We measured two types of splits. Both concerned the *Customer* table created in the DB termed *DB_1* at SQL Server site *Server1*. A *centralized* split created the new segment in a different DB, namely *DB_2*, at the same *Server1*. Alternatively, a *distributed* split created new segment in a DB termed *DB_1* at different site, termed *Server2*. Both cases were generated for *b* = 100, 1000, 10000.

The results are in Figure 2. The time for each *b* is the average one over several experiments. We recall that the split operation involves (i) the creation of the segment, (ii) the move of *b*/2 tuples, (iii) the alteration of the check constraints of all the segments, and (iv) updates to the meta-tables. As expected, the overhead of a distributed split is systematically greater. This is probably due to the internal dialog of the linked SQL Servers. The difference is about four times for larger *b*'s. The split remains nevertheless fast, e.g. 2 sec. at most in our experiments. Notice that the scale is logarithmic, hence the curves are sub-linear. Thus the scalability is good with respect to the segment size.

3.2 Client Side

We have created again the table *Customer* on *Server1.DB_1*. Next, we have generated its (distributed) splits towards *Server2.DB_1*. This was done for three segment capacities *b* measured above. We also used two scalable partitioned views of *Customer* table, both termed *Customer_view,* one in *Server1.DB_1* and one in *Server2.DB_1*. Both were as follows:

CREATE VIEW Customer_view AS
SELECT * FROM Server1.DB1.Customer

The view in *Server1.DB_1* was in fact the initial one generated by ***create_scalable_table*** from Example 1.

Next, to test the basic search performance, we used the following Transac SQL query to *Customer* table[2], [4]:

(Q 1) ***SELECT * FROM Customer_view WHERE Customerid=90***

[2] In the 1st SD-SQL Server implementation used for the measurements, the query to a scalable table *T* must refer explicitly to the name of its scalable partitioned view. We recall that this name is *T_view* by default.

The query was distributed. It executed at *Server2.DB_1* and looked up for the tuple in *Server1.DB1.Customer*. In particular its execution under SD-SQL Server had to adjust the scalable partitioned view. For this purpose, SD-SQL Server client side at *Server2* consulted the meta-table SD-RP at *Server1*. This query was thus more time consuming than if it performed at *Server1* site.

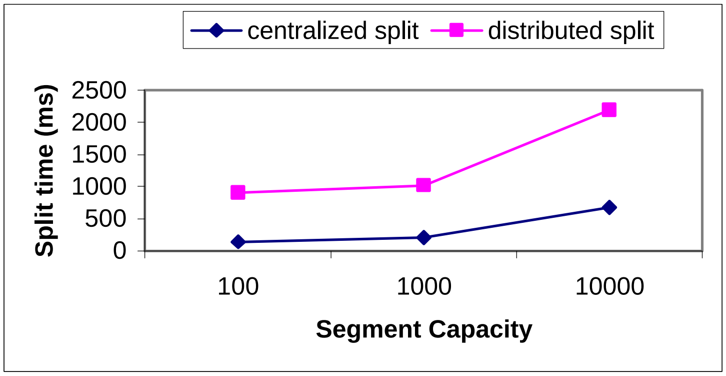

Figure 2 SD-SQL Server segment split time

Likewise, to measure the insert overhead, we have used the query:

(Q 2) ***INSERT INTO Customer_view VALUES (25)***

This query was also distributed, executing at *Server2.DB_1*, while inserting the tuple into *Server1.DB1.Customer*. Again (Q 2) was more time consuming than if it executed at *Server1*.

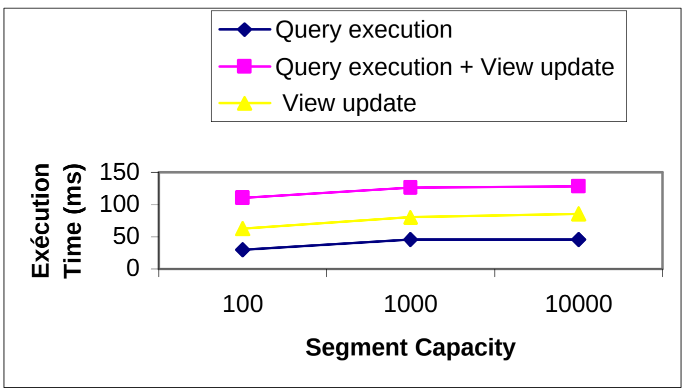

Figure 3 SD-SQL Server search query (Q 1) execution time

Figure 3 and Figure 4 show the results, averaged over several experiments. The line “query execution” shows the time to execute the query with the view test, but without the view update. Other lines show the view update time and the total time. It appears from the figures that for both queries, the overhead of view adjustment is rather low. Notice however that it always takes more time than (Q 1) itself. This query is indeed particularly simple. We recall that view updates should be nevertheless infrequent. In any case, the scalability of SD-SQL Server appears good, being largely sub-linear.

Finally, to determine also the overhead of the SD-SQL Server view test at the client side, and of its insert overhead at the server side, we have experimented with the execution time of (Q 1) and of (Q 2) directly on SQL Server. The average execution time was about 30 ms for a search query, and about 76 ms for an insert

Submitted to WDAS-04

query. These times are only slightly inferior to the time of the same query to SD-SQL Server. They thus terminate the proof of low overhead of our implementation.

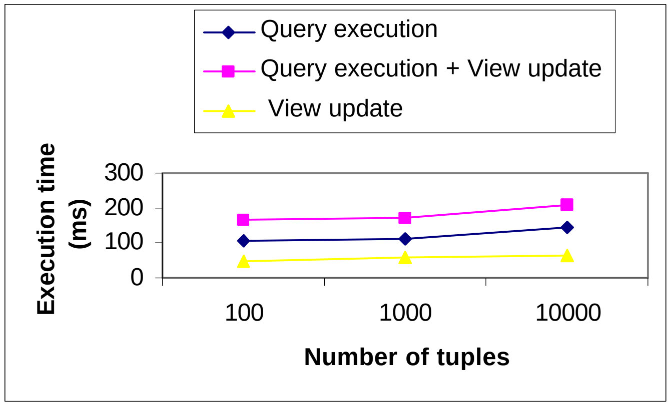

Figure 4 SD-SQL Server insert query (Q 2) execution time

4 Conclusion

The SD-SQL Server is the first DBS we are aware of putting into practice the scalable distributed database partitioning. The transparency of the distribution is an important step beyond the current technology of a parallel DBMS. Lack of this capability is felt by users as one of most important limitation of DBS technology at present, [3]. We have presented our architecture and validated it through basic experiments. These show the scalability and the efficiency of our approach. As a result, SD-SQL Server opens new prospects for the applications.

The work on the implementation is far from finished. In particular the splitting algorithm will be expanded to better deal with inserts of a large (much greater than b_s) number of tuples at once. Likewise, we will expand the query processing that is limited at this time, e.g., to queries without subqueries and without views in FROM clause. We will also add to the interface more SQL operations on the scalable tables, e.g., ALTER, DELETE. We also continue with the performance study, using more complex SQL queries, including TPC benchmarks. Finally, we plan to apply our architecture to SkyServer database, [2].

Acknowledgments

This work was partly supported by the research grants from the European Commission project ICONS project no. IST-2001-32429 and from Microsoft Research.

References

[1] Litwin, W., Rich, T. and Schwarz, Th. An Architecture for a scalable Distributed DBSs/ application to SQL Server 2000. 2nd Intl. Workshop on Cooperative Internet Computing (CIC 2002), August, 2002, Hong Kong.
[2] Gray, J. & al. Data Mining of SDDS SkyServer Database. WDAS 2002, Paris.
[3] Ben-Gan, I., And Moreau, T. Advanced Transact-SQL for SQL Server 2000. Apress Editors, 2000.
[4] SQL Server Books Online of Microsoft SQL Server 2000.
[5] Litwin, W., Neimat, M.-A., Schneider, D. Linear Hashing for Distributed Files. ACM-SIGMOD International Conference on Management of Data, 1993.
[6] Litwin, W., Neimat, M.-A., Schneider, D. A Scalable Distributed Data Structure. ACM Transactions on Database Systems (ACM-TODS), Dec. 1996.