SD-SQL Server:
Scalable Distributed Database System

Witold Litwin[1], Soror Sahri[2]

CERIA, Université Paris Dauphine, France
Soror.Sahri@dauphine.fr

Abstract

We present *SD-SQL Server*, a prototype scalable distributed database system. It let a relational table to grow over new storage nodes invisibly to the application. The evolution uses splits dynamically generating a distributed range partitioning of the table. The splits avoid the reorganization of a growing database, necessary for the current DBMSs and a headache for the administrators. The demo illustrates the architecture of our system, its capabilities and performance. The experiments with the well-known SkyServer database show that the overhead of the scalable distributed table management is typically minimal. To our best knowledge, SD-SQL Server is the only DBMS with the discussed capabilities at present.

1. Introduction

The explosive growth of the volume of data to store in databases makes many of them huge and permanently growing. Large tables have to be hash or range etc partitioned over several storage sites. Current DBMSs, e.g. SQL Server, Oracle or DB2 to name only a few [RZL02], provide static partitioning only. To scale tables over new nodes, the DBA has to manually redefine the partition and run a data redistribution utility. The relief from this trouble became an important user concerns, [BM00].

This situation is similar to that of file users forty years ago in the centralized environment. The Indexed Sequential Access Method (ISAM) was in use for the ordered (range partitioned) files. Likewise, the static hash access methods were the only known for the files. Both approaches required the file reorganization whenever the inserts overflowed the file capacity. The B-trees and the extensible (linear, dynamic) hash methods were invented to avoid the need. They replaced the file reorganization with the dynamic incremental splits of one bucket (page, leaf, segment…) at the time. The approach was successful enough to make the ISAM and centralized static hash files in the history.

Efficient management of distributed data present specific needs. The Scalable Distributed Data Structures (SDDSs) addressed these needs for files, [LNS93], [LNS96]. An SDDS scales transparently for an application through distributed splits, hash, range or k-d based. In [LRS02], the concept of a Scalable Distributed DBS (SD-DBS) was derived for databases. The SD-DBS architecture supports the *scalable* relational tables that accommodate their growth through the splits of their overflowing *segments* at SD-DBS storage nodes. As for an SDDS, the split can be in principle hash, range or k-d based with respect to the partitioning key(s). The application sees a scalable table through a specific type of updateable distributed view termed *(client) scalable* view. Such a view hides the partitioning and dynamically adjusts itself to the partitioning evolution. The adjustment is lazy, in the sense it occurs only when a query to the scalable table comes in and the system finds the view out of date. Scalable tables make the database reorganization useless, similarly to B-trees or extensible hash files with respect to the earlier file schemes.

To prove the feasibility of the scalable tables, we have built the prototype termed SD-SQL Server, [LS04]. The goal was to offer the usual SQL Server capabilities, but for the scalable tables. The SD-SQL Server user manipulates scalable tables as if they were SQL Server tables. However, growing scalable tables dynamically migrate on more SD-SQL Server nodes, invisibly to the user.

More precisely, when an insert makes a segment of a scalable table *T* to exceed some parameterized size of *b* tuples at a node, the segment splits. The split is range partitioned with respect to the key. It migrates all the lowest *b*/2 tuples in the segment. The moving tuples enter one or more new segments on available SD-SQL Server nodes. These segments

[1] Witold.Litwin@Dauphine.fr

[2] Soror.Sahri@Dauphine.fr

are dynamically appended to *T* maintained in a dedicated SD-SQL Server meta-table. The actual partitioning is hidden behind a scalable view of *T*. That one reuses internally an SQL Server partitioned distributed union-all view, with the Check constraints for the view updates, [M00]. The SD-SQL Server keeps the view definition in line with the actual partitioning.

The scalable table management creates an overhead with respect to the static partitioning use. The design challenge for any SD-DBS is to minimize this overhead. The performance analysis of our prototype, using in particular the well-known SkyServer database, [G02], proves this overhead negligible for practical purpose. The current capabilities of SQL Server allow an SD-SQL Server scalable table to reach at least 250 segments. This should suffice for many terabyte tables. SD-SQL Server is the first system with the discussed capabilities, to the best of our knowledge. Our results pave the way towards their successful use for other DBMSs.

Our demonstration focuses on the practical use of the new capabilities of the SD-SQL Server. We show the following manipulations:

- The creation of a scalable table and its expansion through the splits.
- The on-line performance reported by the SQL Profiler. We show the overhead of an SD-SQL Server split for various segment sizes. Likewise, we show the overhead of various queries to scalable tables, in particular involving the view adjustment. To make the point about the negligible incidence of the SD-SQL Server overhead, we compare the execution times to those of the direct use of the SQL Server, whenever appropriate.
- The on-line performance, especially with respect to the SD-SQL Server overhead, measured by the SQL Profiler.

Section 2 recalls the architecture of the SD-SQL Server. Section 3 presents the demonstration more in detail. Section 4 illustrates some performance measures. Section 5 concludes the discussion.

2. SD-SQL Server Architecture

2.1 Gross Architecture

Figure1 shows the gross architecture of SD-SQL. The system is a collection of SD-SQL Server nodes. Each node has the component termed SD-SQL Server *manager*. The manager's code implements various SD-SQL Server services for the scalable table management. Every manager uses internally the services of SQL Server. The latter handles at every node the database with the SD-SQL Server specific meta-tables and various stored procedures implementing the SD-SQL Server services. It may also hold the segments, as discussed below. The SQL Servers provide furthermore the inter-node communication of data and of queries. The nodes are configured for this purpose as SQL Server linked nodes. The current limit on the number of linked SQL Server nodes seems around 250 at present. This is the cause of the above discussed scalable table size limit at present.

There are three kinds of SD-SQL Server nodes. A *client* node (and manager) handles the user/application interface, i.e. the queries and views only. In particular, when a query refers to a client view, it checks and eventually adjusts the view to fit the actual partitioning. Internally, the adjustment means a dynamic change to the underlying SQL Server partitioned and distributed view schema, representing the SD-SQL Server client view to the SQL Server.

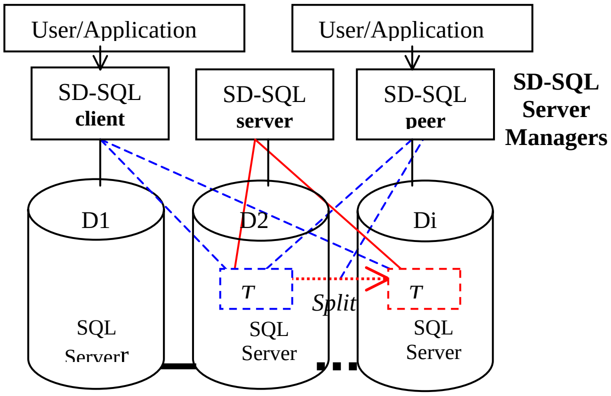

Figure1: Gross Architecture of SD-SQL Server

A *server* node only manages the segments of scalable tables, located in its *segment DB*. It manages the segment evolution, the splitting in particular. The server also executes the (sub) queries coming from (linked) SQL Servers at other nodes on behalf of SD-SQL Server client queries. Finally a *peer* is both a client and a server.

Figure1 shows the nodes termed *D1…Di*, supposed named upon their (segment) DBs. *D1* manager is a client, without segments in its SQL Server DB. *D2* is a server hence it does not carry the users/applications on its node. Finally, *Di* is a peer providing thus all the services of the SD-SQL Server manager.

Furthermore the figure shows a scalable table *T* that was initially created at *D2*. *T* splits many times since at nodes not shown. The figure shows the last split creating a segment at *Di*. Server *D2* is the only to always know the actual partitioning of *T*, symbolized with the full lines. Client *D1* happened to issue an operation after the last split. Its view (dotted lines) was adjusted and is, for the time being, up to date. Peer *D3* in contrast did not yet manipulate *T* after the split. Its client view is out of date. At least *Di* is not in it.

The SD-SQL Server architecture uses the standard SQL Server. Any contrary approach could be utopian for anyone not from Microsoft.

2.2 The SD-SQL server

2.2.1 Meta-tables

The servers carry some SD-SQL Server specific meta-tables in the segment DBs. They describe the actual partitioning of the scalable tables, the linked servers etc. The server updates the tables when a split or a creation of a segment in its segment DB occurs. To search or update the meta-tables the server uses SQL queries detailed in [LS04].

- *Di.SD-RP (DB-S, Table)*. This table defines at server or peer *Di* the partitioning of every table *T* created at *Di* (as its initial segment). Tuple *(Dj, T)* enters *Di.SD-RP* each time a segment of *T* is created in segment database *Dj*. A new segment enters a *Dj* that does not own a *T* segment.
- *Di.SD-S (Table, S-max)*. This table fixes for each *T* at *Di* the maximal size (in tuples) of its segment.
- *Di.SD-C (DB-T, Table)*. This table contains the tuple *(Di, T)* for every segment at *Di*. Attribute *T* identifies the scalable table the segment belongs to. Attribute *Di* indicates the server where *T* was created. It thus points to table *Di.SD-RP* with the actual partitioning of *T*.
- *Di.SD-Site (Site)*. Each tuple in this table indicates a server node *Di* available for splits of *Di.Segments*.

2.2.2 Splitting

A split of segment *Di.T* occurs whenever *Di.T* exceeds the size stored for it in *Di.SD-S*. At present, for every *T* there is a single size. The overflow may be the result of an insert of arbitrarily many tuples by a single *INSERT* command. The server tests the overflow of a segment using a trigger, [LS04]. It then selects $N \geq 1$ servers from its *Di.SD-Site* table not already in use for *T* (note the difference to a simple B-tree like split). *N* is such that each server receives at most half of a segment capacity. The split range partitions then *Di.T* and provides each new segment with *T* scheme. It also creates the new indexes for all the declared ones at *T*. Each manager involved in the split uses the services of its local SQL Server through specific SQL queries.

Once the SD-SQL manager at *Di* completes the process, it alters the Check Constraint of the segment *Di.T*. Finally, it updates its own meta-tables and the *SD-RP* table of *T*.

2.3 The SD-SQL client

2.3.1 Scalable Distributed Views

Each client manages its scalable view of every scalable table it allows the access to. Conceptually, the scalable view T_V, of table *T* is the client image of *T* range partitioning, perhaps outdated. The client represents every T_V in its SQL Server as a distributed partitioned view of *T*. We recall that such a view in SQL Server is a *UNION ALL* view, updateable provided the check constraints at the underlying tables are set (which is the case of every SD-SQL Server segment). At the creation of *T*, let it be at server *Di*, the client image is set to:

```
CREATE VIEW T_view AS
SELECT * FROM Di.T
```

The client adds the suffix '*_view*' to avoid the name conflict internal to its SQL Server between the view and table names in the case of the peer. It deals accordingly when processing any query to *T*.

When a query at a client invokes *T*, the client checks its scalable view and adjusts it if splits occurred. The client has for this purpose the meta-table termed *C-IMAGE (Table, Size)*. When the client creates table *T* at node *Di*, or its image of *T* created by another client, it enters the tuple *(Di.T, 1)* into *C-IMAGE*. When a query to *T* comes in, the client retrieves the *T* tuple from *C-IMAGE* to match the *Size* against the actual count, let it be C_T, of *T* segments in *Di.SD-RP*. If *Size* matches C_T, the view is up to date. Otherwise, the client sets *Size* to C_T. Next, the client adds to the definition of the underlying distributed partitioned view the missing segments found in *Di.SD-RP*. The result is like:

```
CREATE VIEW T_view AS
SELECT * FROM Di.T
```

*UNION ALL SELECT * FROM Dj.T*
UNION ALL …
The clients checks and perhaps adjust in this way all the scalable tables referred to in the query.

2.3.2 Scalable View Adjustment

Except for the client that triggered the split, the existing scalable views are not adjusted synchronously at the split time. As in general in an SDDS, this could be highly ineffective in presence of many clients. Some could be unknown to the splitting server or unavailable at that time etc. The client checks therefore instead its scalable view correctness only when there is a query to it. The client has for this purpose the meta-table termed *C-IMAGE (Table, Size)*. When the client creates table *T* at node *Di*, it enters the tuple *(Di.T, 1)* into *C-IMAGE*. Later, when a query to *T* comes in, the client first retrieves the *T* tuple from *C-IMAGE* to match the *Size* against the actual count, let it be C_T, of *T* segments in *Di.SD-RP* through the specific SQL query. If *Size* matches C_T, the view is OK. If *Size* is different of C_T, then the client sets it to C_T. Next, the client adjusts the definition of the underlying distributed partitioned view accordingly, getting the locations of the missing segments from *SD-RP*. SQL Server recompiles the view on the fly.

The clients checks and perhaps adjust in this way all the scalable tables referred to in the query. The query may refer to the table directly in main *FROM* clause, or through a view name or through an alias, or in a sub-query *FROM* clause etc. The parsing is quite simple to implement for the first type of queries. It is more tedious for the latter options. We will show the experiments with queries of the first kind. More general parsing capabilities are under implementation. The issue was felt secondary for the prototype, appearing rather programming than research challenge.

3. SD-SQL Server Demo

The demo illustrates the new capabilities of the SD-SQL Server. We demonstrate its SQL interface and prove the efficiency of the distributed scaling of the underlying tables. We measure the overhead of distributed splitting and of view adjustment. We show that the incidence should be typically negligible in practice.

We experiment the system on peers, named *Peer1*, *Peer2…* distributed at the machines at the stand. We use the 120 GB fragment of the SkyServer database and some of its benchmark queries, [G02]. The demo script is as follows:
(0) Start the SD-SQL Server at each peer. This starts also all the linked SQL Servers underneath.
(1) Create a scalable table. We use the *create_sd_table* function of SD-SQL, operationally implemented as SQL Server stored procedure also termed *create_sd_table***.** The input is the traditional *CREATE TABLE* statement and the maximal segment size. The procedure executes *CREATE TABLE* statement, and a number of other distributed stored procedures. The execution is normally transparent, but we disclose it at the demo. We apply the creation procedure to make *PhotoObj* table of SkyServer DB a scalable table. *PhotoObj* is a table of 158,426 tuples (about 260 MB). We fix the segment size at 158,000 tuples. The result is the following demo execution of the *create_sd_table* at *Peer1*:

EXEC create_sd_table 'CREATE TABLE PhotoObj(objid BIGINT PRIMARY KEY,….)', 158000

As *PhotoObj* is not prefixed, SD-SQL Server creates it in the segment DB of Peer1. The demo shows the following steps:

- Creation of *PhotoObj* empty segment at *Peer1*.
- Update of the server meta-tables.
- Update of *C-IMAGE* at *Peer1* and creation there of the initial scalable view *PhotoObj_view*.

We then insert by SD-SQL Server command *INSERT INTO* 158,000 tuples from the original *PhotoObj* into ours. It should take about 30 sec.
(2) Split a scalable table. We insert one more tuple into *PhotoObj*. We show how *Peer1* splits then *PhotoObj*, creating a new segment at Peer2. We measure the time by the SQL Profiler.
(3) Creation of a scalable view. We create 2nd scalable view *PhotoObj_view* on Peer*2*. We use SD-SQL Server stored procedure:

EXEC create_scalable_view 'PhotoObj', 'Peer2'

(4) Creation of an index on a scalable table. We create one at *PhotoObj* using *create_scalable_index* stored procedure. We show the local indexes appearing at all the existing segments of *PhotoObj*. We generate an additional split and show the propagation of the index to the new segments. Again we measure the time by the SQL Profiler.
(5) Various SQL queries to the scalable tables. We show benchmark queries with aggregations, joins… to various SkyServer DB scalable tables, [G02]. These tables are prepared earlier to save time, or created at the demo, time permitting. We display the underlying operations (normally transparent) and execution times. We also display the execution time by SD-SQL Server with and without view adjustment, and directly by SQL Server. We prove that the SD-SQL Server overhead is small in practice. Among the queries, we show the following distributed ones proving their measures in the next section. Query (Q1) benchmarks the fast side, bringing only a few tuples from nodes different from

that of the query. Query (Q2) is in turn at the expensive side, bringing 129,470 tuples from all the nodes. We execute these two queries at *Peer1*.

(Q1) *SELECT TOP 10 objid FROM PhotoObj_view*
WHERE objid
NOT IN (SELECT objid FROM Photoobj)

(Q2) *SELECT * FROM PhotoObj_view*
WHERE (status & 0x00002000 > 0)
AND (status & 0x0010 > 0)

(6) Finally, we demonstrate the availability and performance of ALTER and DELETE SQL commands for scalable tables. These are implemented as SD-SQL Server stored procedures *alter_scalable_table* and *drop_scalable_table*. We show how they act at all the segments of a scalable table. Again, these are actions normally transparent to the SD-SQL Server user/application.

4. Performance Measurements

We have measured in various ways SD-SQL Server access performance. The hardware consisted of 1.8 GHz P4 PCs with 512 MB and 1 GB of RAM, linked by 1 Gbs Ethernet.

We have made a series of experiments to prove the scalability and efficiency of the SD-SQL Server. We show the overhead at the SD-SQL servers and clients. At the servers, we measured the split time. At the clients, we measured the overhead of a scalable view management during a query. We have particularly experimented with the *PhotoObj* table. Figure 2 shows the split time of this table for different segment sizes *b*. The split time remains always fast. It is sub-linear with respect to the segment size, thus the scalability is good.

We have also measured the execution time of (Q1) and (Q2) from the previous section. The measures of (Q1) show that the time difference between the execution on SD-SQL Server with view checking only and directly on SQL Server is negligible, Figure3. The query executes in about 300 ms. The view adjustment that implies the largest overhead for the query execution takes here about 0.7 s and is constant as one could expect. It makes the substantially longer, but we recall, it remains a rare operation. The execution in turn of the expensive (Q2) takes about 45 sec with the view adjustment and 44 sec by SQL Server directly. Thus the view adjustment took again about 1 sec, but now it represents about 2 % of the execution time only.

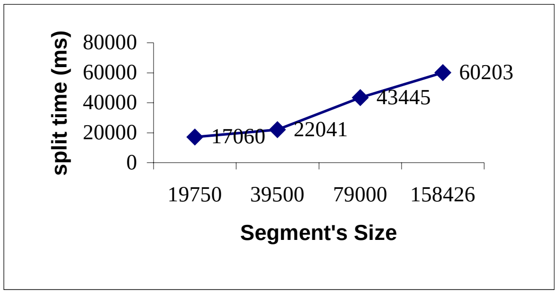

Figure 2 SD-SQL Server Segment split time

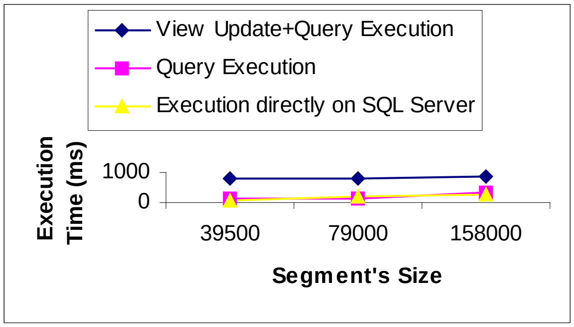

Figure 3 SD-SQL Server query (Q1) execution time

5. Conclusion

Our demonstration shows the prototype implementation of SD-SQL Server: a first relational database system with dynamically splitting tables. We show its main concepts of scalable tables and views. We illustrate how they actually function for an application, using the SkyServer database. We demonstrate the availability and performance of the usual SQL commands for scalable tables under our system. Further work concerns more benchmark queries, various implementation and optimization issues, and deeper performance analysis.

6. Acknowledgments

We thank J. Gray (Microsoft BARC) for providing the SkyServer database and for the counselling crucial to this work, and G. Graefe for info on SQL Server linked servers' capabilities. The support for this work came partly from the research grant of Microsoft Research, and from the European Commission ICONS project no. IST-2001-32429.

7. References

[BM00] Ben-Gan, I., and Moreau, T.
Advanced Transact SQL for SQL Server 2000.
Apress Editors, 2000.

[G02] Gray, J. & al. Data Mining of SDDS SkyServer Database. WDAS 2002, Paris, Carleton Scientific (publ.).

[LNS93] Litwin, W., Neimat, M.-A., Schneider, D. Linear Hashing for Distributed Files. ACM-SIGMOD International Conference on Management of Data, 1993.

[LNS96] Litwin, W., Neimat, M.-A., Schneider, D. LH*: A Scalable Distributed Data Structure. ACM-TODS, Dec. 1996.

[LRS02] Litwin, W., Rich, T. and Schwarz, Th. Architecture for a scalable Distributed DBSs application to SQL Server 2000. 2nd Intl. Workshop on Cooperative Internet Computing (CIC 2002), August 2002, Hong Kong.

[LS04] Litwin, W & Sahri, S. Implementing SD-SQL Server: a Scalable Distributed Database System. Intl. Workshop on Distributed Data and Structures, WDAS 2004, Lausanne, Carleton Scientific (publ.), to app.

[M00] Microsoft SQL Server 2000: SQL Server Books Online.

[RZL02] Rao, J., Zhang, C. , Lohman, G and Meggido, N. Automating physical database design in a parallel database, ACM SIGMOD '2002 June 4-6, Madison, Wisconsin, USA.