CERIA Res. Rep., October 2005

Architecture and Interface of Scalable Distributed Database System SD-SQL Server

Witold Litwin[1], Soror Sahri[2], Thomas Schwarz[3]

CERIA, Paris-Dauphine University
75016 Paris, France

Abstract. We present a scalable distributed database system SD-SQL Server. Its original feature is the scalable distributed partitioning of its relational tables. The system dynamically distributes the tables into segments created each at a different SD-SQL Server node. The partitioning is transparent to the applications. New segments result from splits following overflowing inserts. SD SQL Server avoids the periodic and cumbersome manual reorganizing of scaling tables, characteristic of the current DBMS technology. With the comfort of a single node SQL Server database user, the SD-SQL Server user may dispose of many times larger tables. We present the architecture of our system, and its user/application interface. Related work discusses our implementation and shows that the overhead of our scalable distributed table management should be typically negligible.

1. Introduction

The increasing volume of data to store in databases makes them more and more often huge and permanently growing. Typically, large tables are hash or range partitioned into segments stored over different storage sites. Current DBMSs, e.g. SQL Server, Oracle or DB2 to name only a few, provide static partitioning only [1][5][10]. 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, [1].

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, [5][6]. An SDDS scales transparently for an application through distributed splits of its buckets, hash, range or k-d based. In [7], the concept of a Scalable Distributed DBS (SD-DBS) was derived for databases. The SD-DBS architecture supports the *scalable (distributed relational) tables*. As an SDDS, a scalable table accommodates its growth through the splits of its overflowing segments, located at SD-DBS *storage* nodes. Also like in an SDDS, the splits can be in principle hash, range or k-d based with respect to the partitioning key(s). The storage nodes can be P2P or grid DBMS nodes. The users or the application, manipulate the scalable tables from a *client* node that is not a storage node, or from a *peer* node that is both, again as in an SDDS. The client accesses a scalable table only through its specific view, termed (*client*) *image*. It is a particular updateable distributed partitioned union view stored at a client. The application manipulates scalable tables using *scalable* (application) views. These views involve scalable tables through the references to the images.

Every image, one per client, hides the scalable table partitioning and dynamically adjusts to its evolution. The images of the same scalable table may differ among the clients and from the actual partitioning. The

[1] Witold.Litwin@Dauphine.fr

[2] Soror.Sahri@Dauphine.fr

[3] tjschwarz@scu.edu

image adjustment is indeed lazy. It occurs only when a query to the scalable table comes in, and finds an outdated image. Scalable tables make the global database reorganization largely useless. Similarly to B-trees or extensible hash files with respect to the earlier file schemes.

To prove the feasibility of an SD-DBS, we have built the prototype termed SD-SQL Server. The system generalizes the basic SQL Server capabilities to the scalable tables. It runs on a collection of SQL Server linked nodes. For every standard SQL command under SQL Server, there is an SD-SQL Server command for a similar action on scalable tables or views. There are also commands specific to SD-SQL Server client image or node management.

Below we present the architecture of our prototype and its application command interface in its 2005 version. The current architecture addresses more features than [7]. With respect to the interface, we discuss the syntax and semantics of each command. Numerous examples illustrate the actual use of the commands. We hope to convince that the use of the scalable tables should be about as simple as of the static ones in practice. This, despite some limitation of our current interface, with respect to a full-scale one. We have intended it as the "proof of the concept" only, as the whole prototype besides.

The related papers discussed the internal design and the processing performance of SD-SQL Server, [8], [13]. The scalable table processing creates an overhead and our design challenge was to minimize it. The performance analysis proved this overhead negligible for practical purpose. The present capabilities of SQL Server let a scalable table to reach 250 segments at least. This should suffice for scalable tables reaching very many terabytes. SD-SQL Server is the first system with the discussed capabilities, to the best of our knowledge. Our results pave the way towards the use of the scalable tables as the basic DBMS technology.

Below, Section 2 presents the SD-SQL Server architecture. Section 3 discusses the user interface. Section 4 discusses the related work. Section 5 concludes and discusses the future work.

2 SD-SQL Server Architecture

Fig. 1 shows the current SD-SQL Server architecture, adapted from the reference architecture for an SD-DBS in [7]. The system is a collection of SQL Server linked nodes. An SD-SQL Server node is an SQL Server node that runs the SD-SQL Server specific *manager* component. A linked SQL Server that is not (yet) an SD-SQL Server node is a *spare* node. Fig. 1 does not show spares. The SD-SQL Server manager handles at its node an SD-SQL Server specific *node* database (DB). Actually, every manager is a collection of SQL Server stored procedures; themselves kept in the node DBs. The manager takes care of the SD-SQL Server specific commands. These commands apply internally various SQL Server commands. The SQL Servers at each node entirely handle the inter-node communication of data and the distributed execution of SQL queries. In this sense, each SD-SQL Server runs at the top of its linked SQL Server, without any specific changes to the latter.

An SD-SQL Server node is a *client*, a *server*, or a *peer* node. The client manages the SD-SQL Server node user/application interface. This consists of the SD-SQL Server specific commands for the table and view manipulation, and from all the SQL Server commands and services. The specific commands address, on the one hand, the schema management of a scalable table or of an image. They also let to formulate the *scalable* queries, possibly involving scalable tables, directly or through views. An SD-SQL Server specific command is typically an SQL Server stored procedure involving an SQL command as the actual parameter and perhaps other parameters. These are specific to a scalable table management, e.g., the *segment size* that is the maximal number of tuples the segment should contain, beyond which the split should occur. Internally, the (client node) DB stores the images, and the local application views, scalable or not. It also contains some SD-SQL Server meta-tables constituting the catalog **C** at the figure. The catalog registers every image created at the client. Finally, the application may store at the client node all the SQL Server objects that it needs: tables, views, indexes, stored procedures or entire application databases.

When a scalable query comes in, the client checks whether it possibly involves images. If so, it checks each image whether it conforms to the actual partitioning of its table, i.e., unions its all the existing segments. The client uses **C**, as well as some server meta-tables pointed to by **C**, defining the actual partitioning. We recall that a client view may be outdated. The manager dynamically adjusts then any outdated image. Internally, it implies a change to the scheme of the underlying SQL Server partitioned

and distributed view, representing the image to the SQL Server. The manager executes the query, when all the images it uses prove up to date.

A *server* node stores the segments of scalable tables in its *server* (node) DB. Every segment at a node belongs to a different table. At each server, a segment is internally an SQL Server table with specific properties. First, SD-SQL Server refers to in the specific catalog in each server DB, named **S** in the figure. For each segment at the node, the meta-data there identify the scalable table the segment belongs to. They indicate also the segment size. Next they indicate the servers available for the segment's splits. Finally, for a *primary* segment that is the 1st one created for a scalable table, the meta-data at its server provide the actual partitioning of the table.

Next, each segment has an AFTER trigger attached, not shown at the figure. It verifies after each insert whether the segment overflows. If so, the server splits the segment, range partitioning it with respect to the table (partition) key. It moves out enough upper tuples so make the remaining (lower) tuples fitting the splitting segment size. For the migrating tuples, the server creates remotely one or more new segments that are each half-full (notice the difference to a B-tree split creating a single new segment). The new segments are each at a different server. The splitting server chooses those randomly among available server nodes, using its **S** catalog. The new segments become a part of the scalable table. Internally, the segment creation operations are the SQL commands, taken care of by the SQL Servers at the server nodes handling the split.

Furthermore, every segment in a multi-segment scalable table carries an SQL Server *check constraint*. Each constraint defines the partition (primary) key range of the segment. The ranges partition the key space of the table. These conditions let the SQL Server distributed partitioned view to be updateable, by the inserts and deletions in particular. This is a necessary and sufficient condition for a scalable table under SD-SQL Server to be updateable as well.

Finally a *peer* is both a client and a server. Its node DB carries all the SD-SQL Server meta-tables. It may carry both the client images and the segments. The meta-tables at a peer node, termed **P** at the figure, are those in the union of the **C** and **S** catalogs.

A client (or a peer) creates a scalable table upon the application command. The client creating table T, starts with the remote creation of the primary segment of *T*. The primary segment is the only to receive the tuples of *T*, until it overflows. The client is aware of the servers it may use through meta-tables in **C**. The client basically chooses the (primary) server randomly. A peer in contrast usually creates the primary segment in its node DB. Next, the client (or the peer) creates the *primary* client image of *T*, named *T* itself, in its node DB. The creation involves the input into SQL Server meta-tables and into SD-SQL Server **C** (or **P**) catalogs. The client itself becomes the *primary* client of table *T*.

A *secondary* client node i.e., other than the primary one, can create its own *secondary* client image. An application invokes only *T* image, we recall. The segments themselves are invisible to the applications. The splits do not adjust the images. A contrary approach would be often inefficient at best, or simply impossible in practice. As the result every split of a scalable table makes all its images outdated. This is why the client dynamically checks every query for the possibly outdated images, as we described.

A scalable table can finally have *scalable* indexes. These may accelerate searches in scalable tables like SQL Server indexes do for the static tables. The splits propagate the scalable indexes to new segments. Under SD-SQL Server a scalable index consists itself from the *index* segments, symbolized as *I* at the figure. There is one index segment per index and segment of the table. Each index segment is an SQL Server index on the table that a segment constitutes for the local SQL Server.

The interface that an SD-SQL Server client provides to the application for the scalable tables and their views, offers basically the usual SQL manipulation capabilities, up to now available for the static tables only. The client parses every SD-SQL Server (specific) command and defines an execution plan. The plan consists of SQL Server commands and of additional procedural logic. The client passes every SQL Server command produced to its SQL Server for the execution. The SQL Server parses the command in turn, produces sub-queries and forwards them for the distributed execution to the selected linked servers. If the application requests a search, then the remote servers send the retrieved tuples to the local SQL Server internally as well, i.e., among the SQL Server managers at the nodes. That one returns the overall result to the application, including perhaps also tuples found in its local segments.

To let the client to offer these operations, an SD-SQL Server server handles locally for its segments the basic SQL manipulations, embedded however typically into more complex stored procedures. The basic manipulations are the tuple updates, inserts, deletes, and searches as well as the segment indexing, and alteration. More complex procedures, involving multiple SQL commands on the segments and meta-tables, within some procedural logic, correspond to the segment creation, splitting and dropping. As we just mentioned, the split operation in particular, makes the server to remotely request the segment creation(s) on other sites.

Finally, SD-SQL Server allows for the *node management* commands. These let to create new SD-SQL Server nodes at spares, or drop such nodes. A creation installs an SD-SQL Server node with its catalogs at the spare, and makes the new node known to at least some existing SD-SQL Server servers (peers). The drop operation makes a spare node from an SD-SQL Server node. It removes the SD-SQL Server manager with its catalogs and migrates elsewhere the segments or the views stored perhaps at the node. A specific procedure creates the initial SD-SQL Server server of the whole collection.

To illustrate the architecture, Fig. 1 shows the nodes termed *D*1…*Di*+1, upon their node DBs. *D*1 is a client that thus carries only the scalable views and interfaces the applications. All other nodes till *Di* are servers that carry thus only the segments and do not sever to the applications directly. Finally, *Di*+1 is a peer, providing all the capabilities. The nodes jointly manage a scalable table termed *T*. The table has a scalable index *I*. Client *D*1 is supposed to carry the primary image of *T*, locally named *T*. The image unions the segments of *T* at servers *D*2…*Di* with the primary segment at *D*2. Peer *D*i+1 carries a secondary image of *T*. That one is supposed different, including the primary segment only. Both images are outdated. Server *Di* just split indeed its segment and created a new segment of *T* on *Di*+1. It updated the meta-data on the actual partitioning of *T* at *D*2. Such data are indeed always at the node of the initial segment of a scalable table, as we will show. None of the two images refers to this segment as yet. Each will be adjusted to the actual state of *T* only once it gets a query addressing *T* image. The split has also created the new segment of *I*.

Notice finally in the figure that segments of *T* are *_D1_T*. The name reflects the global name of *T* that is the couple (creator node, table name). It provides the uniqueness with respect to different client (peer) nodes possibly creating each a scalable table named *T* for the application local to each node. The segments of these tables may share a node without the name conflict. Similar name conflict could concern the secondary image name. We show our current solution later on.

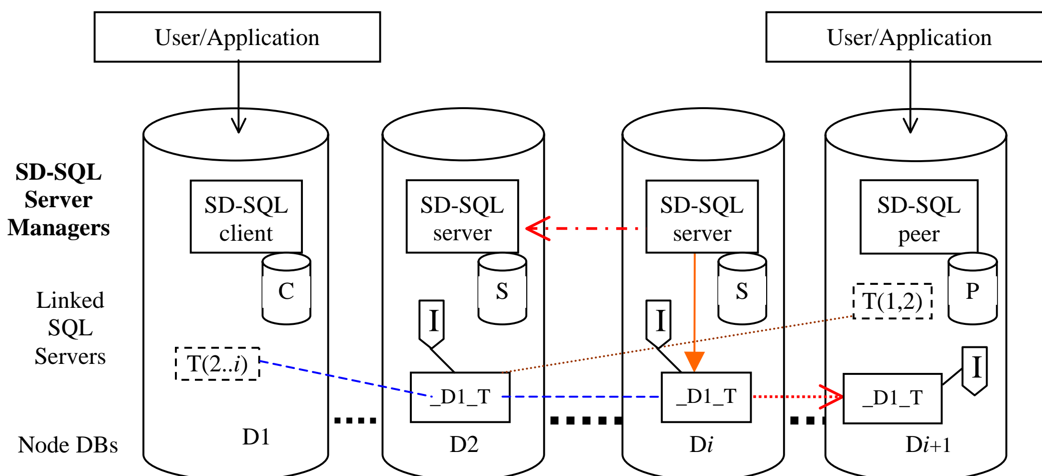

Fig. 1 SD-SQL Server Architecture

3 Application Interface

3.1 Overview

The application manipulates scalable tables and their views essentially through new SD-SQL Server dedicated commands. They perform the usual SQL schema manipulations and queries implying however

now the scalable tables (through the images) or the scalable (application) views. We qualify these commands of *scalable*. They address all the existing segments, regardless of their actual number, and their effects may propagate to the future ones. A scalable command may include additional parameters specific to the scalable environment, with respect to its original static counterpart.

The scalable commands in SD-SQL Server are the SQL Server stored procedures. Most commands apply also to the static tables and views. The application using SD-SQL Server may also directly invoke the (static) SQL Server commands. These calls are transparent to SD-SQL Server managers. Their use should remain limited to the static tables. However, the SQL Server CREATE VIEW command applies to both scalable and static tables. It produces a scalable view whenever a scalable table is among the base tables invoked. The SQL Server DROP VIEW command acts similarly.

We now present the syntax and semantics of the scalable SD-SQL Server commands. For a scalable table schema manipulation, there is a scalable command per every standard SQL command, i.e., for table creation, alteration, indexing or removal (drop). There are furthermore specific scalable commands for image management. For other scalable views, one should use the SQL Server CREATE VIEW and DROP VIEW, as just mentioned. Finally, there are scalable commands for the search, insert, update and delete queries respectively. The operational capabilities of SD-SQL Server scalable commands are sufficient for most applications.

The SELECT statement in a scalable query supports the SQL Server allowed selections, restrictions, joins, sub-queries, aggregations, aliases…etc. However, the queries to the scalable multidatabase views are not possible at present. The reasons are the limitation of the SQL Server meta-tables that SD-SQL Server uses for the parsing. Moreover, the scalable INSERT command over a scalable table lets for any insert accepted by SQL Server for a distributed partitioned view. This can be a new tuple insert, as well as a multi-tuple insert through a SELECT expression. The UPDATE and DELETE statement offer similar capabilities. In contrast, some of SQL Server specific SQL clauses are not supported at present by the scalable commands; for instance, the CASE OF clause.

We discuss the syntax and semantics of the SD-SQL Server scalable commands in the normal execution conditions only. If a command addresses a node, we only consider the case of an available one. Likewise, we consider that every SQL Server node involved in a command execution works, as it normally should. We discuss the internal processing of each command later in Section 4.

3.2 Scalable table management

♣ **Table Creation**. The application, let it be on a node D, creates a scalable table, let it be T, by executing the *create_table* command at its client (peer) with the following syntax:

EXEC create_table *'SQL: Create Table **T** clauses, Segment_size'*

The parameter *'SQL: Create Table **T** clauses'* is the text of the SQL Server CREATE TABLE command following the command name itself. The table name may be global, i.e., prefixed by a node name, in principle, but only the local creation is supported at present. Also, the table name should not start with '_' (to avoid the name conflict with a segment name on a peer). The SQL command invoked has to respect all the constraints that SQL Server imposes at a distributed partitioned view [9]. In particular, the scalable table has to have the primary key that supports the CHECK constraints partitioning the key space. The *Segment_size* parameter fixes the maximal size of a segment of *T*. The command creates a scalable table only. To create a static table, e.g., to avoid the above mentioned restrictions on the scalable ones, one should use the SQL Server CREATE TABLE command.

Example. The user at peer *Peer1* creates the scalable table *PhotoObj*. The name comes from our benchmark for experimental performance analysis that is a fragment of SkyServer DB [2]. The user wishes that a segment contains at most 10.000 tuples for the efficient distributed query processing. It applies the command:

Exec create_table *'PhotoObj (objid BIGINT PRIMARY KEY…)', 10000*

Here, we have show only the beginning of the actual SQL CREATE TABLE command clauses. The result is the *PhotoObj* scalable table with the *objid* primary key, and the segment size of 10.000 tuples. It appears to the applications as being at *Peer1* node, i.e., its multidatabase (global) SD-SQL Server name is

Peer1.PhotoObj[4]. An application at another node, e.g., *Client1*, could also create a table named *PhotoObj*. This one would appear as at *Client1*, with the multidatabase name of *Client1.PhotoObj.*

♣ **Table Alteration.** To alter a scalable table, let it be table *T*, the application executes the command:

***EXEC alter_table** [‘SQL:’ALTER TABLE T clauses], [new segment_size]*

The command carries at least one of the clauses. The ‘*SQL: ALTER TABLE* clauses’ parameter contains the SQL Server ALTER TABLE clauses with their usual syntax. Accordingly, the SD-SQL Server command provides the same capabilities for a scalable table. SD-SQL Server propagates the alteration to every segment. However, the effects of the decrease to the segment size are lazy. The commands actually affect a newly overflowing segment only when an insert to it occurs, triggering a split.

Example. The user alters the scalable table *PhotoObj* initially created at Peer1 by adding new column to it and updates its maximal size. The user may alter the table using the command:

***EXEC alter_table** ‘ Peer1.PhotoObj add t int’, 10000*

The result of the scalable command is the scalable *PhotoObj* table with the *t* column added to all its segments, and its segment size updated to 10 000 tuples. If the alteration is requested at the original node of the scalable *PhotoObj* table, i.e *Peer1*, then the prefix indicating the multidatabase name is not necessary.

♣ **Index Creation.** The application creates a scalable distributed index, let it be *I*, on a column of a scalable table, let it be *T*, by executing the *create_ index* command with the following syntax:

***EXEC create_index** [‘SQL: Create Index I ON T clauses’*

The command creates the index *I* on a *column* of the scalable table *T*. The input parameter *‘SQL: Create Index I ON T clauses’* are the clauses of the SQL Server CREATE INDEX command. The application may use the local or global name of *T*. The latter is necessary if one invokes the command at a node another than the primary one *T*

Example. The command below, executed at the *Peer1* node, creates the *run_index* scalable index on the *run* column of the *PhotoObj* scalable table.

***EXEC create_ index** ‘run_index ON Photoobj (run)'*

If this index creation is requested elsewhere, then one should use the name *Peer1.Photoobj*.

♣ **Index Removal.** The application drops an existing scalable index *I* on table *T*, by executing the command:

***EXEC drop_ index** ‘SQL: Drop Index T.I clauses’*

The input parameter is the SQL Server *DROP INDEX* command.

Example. The command below, executed at the *Peer1*, removes *run_index*.

***EXEC drop_ index** ‘Photoobj.run_index’*

♣ **Table Removal.** To remove scalable table *T* the application uses the command:

***Exec drop_ table** [‘SQL: DROP TABLE T clauses’]*

The command removes all *T* segments with their tuples and its primary image. The syntax and semantics of the input parameter that of the SQL Server DROP TABLE clauses.

Example. Drop the scalable table *PhotoObj* created at peer *Peer1*:

***EXEC drop_ table** ‘Peer1.PhotoObj’*

The prefix is not necessary for the command invoked at *Peer1*.

[4] For simplicity, we make abstraction here of the owner name part of the SQL Server global naming, as well as about the node DB name component.

♣ **Image Creation**. The application creates a secondary image at its client (peer) through the following command:

EXEC create_image *'[Primary_node].Table'*

At any node, only one image of a given table can exist. The application wishing to use another name for an image may do it through CREATE VIEW. It should then use the local image name, formed as follows. If the primary node is *D* and the table name is *T*, then the local image name is ***__D_T***. The convention eliminates the name conflict with table and segment names that could exist at the peer where the new image is being created.

Example. We create the (secondary) image of *PhotoObj* through the statement:

EXEC create_image 'Peer1.'PhotoObj'

To name it locally *PhotoObj* as well, one could use the SQL Server command:

CREATE VIEW *PhotoObj* as Select * from __ *Peer1_PhotoObj*

♣ **Image Removal.** The application removes a secondary image, using the command at its client (peer):

EXEC drop_image **'**image_name*'*

The command cannot remove the primary image. The **'**image_name*'* parameter may be the global table name. It can also be the local image name.

Example. The commands below delete a secondary) image of our *PhotoObj* table.

EXEC drop_image *' Peer1.Photoobj'* or ***EXEC drop_image*** *'__ Peer1_Photoobj'*

3.3 Scalable Queries

♣ **Search.** An application can submit a select query to scalable tables by executing *select* command at its client (peer) with the following syntax:

EXEC select *'SQL: Select clauses', Segment Size', 'Primary Key'*

The *'SQL: Select clauses'* parameter is the SQL SELECT command clauses with their usual syntax. The application may invoke in the scalable query the aggregations, joins, aliases, sub-queries…etc. The *'Segment Size'* and *'Primary Key'* input parameters are optional. The application may use them when it issues a SELECT INTO query and wishes to make the result a scalable table. One use of this capability may be to recreate a static table as a scalable one. The *'Segment Size'* indicates the size of the table to change into a scalable one. The *'Primary Key'* indicates the (partitioning and primary) key column(s) in the new table.. These columns can be the original ones, or the ones dynamically named in the Select clause, e.g., produced by the aggregate functions.

Example. The user may execute a select query on a scalable table as below:

EXEC select *'* FROM PhotoObj'*

The result of the scalable command is the execution of the select query *'SELECT * FROM PhotoObj'*.

Next, the following command copies the tuples of a static SQL Server table *PhotoObj*, into to a new scalable table *S_PhotoObj* with the segment size of 500 tuples and *objid* column as the primary key:

EXEC select *'* INTO S_PhotoObj FROM PhotoObj', 500, 'Objid'*

To achieve the recreation, the application may further drop *PhotoObj* and rename *S_PhotoObj*.

♣ **Insert.** An application can insert tuples into a scalable table by executing scalable *insert* command at its client (peer) with the following syntax:

EXEC insert *'SQL Insert clauses'*

Here, *'SQL: Insert clauses'* input is the SQL Server INSERT command. The target table may be static or scalable. In the latter case, a split may result. The application may in particular use the command with the SELECT clause on scalable or static tables.

Example. The command below inserts the tuple (2255031257923860) in the *objid* column of the scalable table *PhotoObj*.

EXEC insert *'INTO PhotoObj (objid) values (2255031257923860)'*

The next command inserts perhaps many tuples into *PhotoObj* scalable table, at *Peer1*, from another *PhotoObj* table on *Peer5* node. The source table could be scalable or static, i.e., the SQL Server table only.

EXEC insert *'INTO PhotoObj SELECT * FROM Peer5.PhotoObj WHERE objid not exists in (SELECT objid FROM PhotoObj)'*

♣ **Update.** An application updates a scalable table by executing the scalable *update* command:

EXEC update *'SQL: Update clauses'*

The input parameter is the SQL Server UPDATE command, including eventually the SELECT clause and, for a scalable table, respecting the SQL Server constraints on updates to distributed partitioned tables.

Example. The command below updates the value of the *run* column, in the scalable table *PhotoObj,* for the tuple containing the value '*2255031257923860'* in its *objid* column:

EXEC update *' PhotoObj set run= 752 where objid=2255031257923860'*

As for the command below, it shows how to update *PhotoObj* scalable table by changing the *run* column values to 752 for the first 10 tuples of the table.

EXEC update *'PhotoObj set run= 752 where objid in (SELECT TOP 10 objid FROM PhotoObj'*

♣ **Delete.** An application deletes tuples from a scalable (or static) table using the command:

EXEC delete *'SQL: Delete'*

The input is the SQL Server DELETE command. The command leaves unchanged the schema of a scalable table, the table partitioning in particular.

Example. The command below deletes a tuple from *PhotoObj* scalable table:

EXEC delete *'FROM PhotoObj WHERE objid=2255031257923860'*

3.4 Node Management

A script file creates the first ever (initial) SD-SQL Server node at a collection of spares. After that, the SD-SQL Server nodes offer the following commands to the administrator, user or application.

♣ **Scalable Server Creation.** The application creates new scalable servers on given spares by executing the command:

EXEC create_server *'new_server1[, new_server2]…',*

Example. The script has created the first ever SD SQL client at spare *Node*1 in our collection of spares. The command below could then create a scalable server at the spare *Node*2.

EXEC create_server *'Node*2*'*

From now on, '*Node*2' is a server node, ready to store segments.

♣ **Scalable Client Creation.** An application creates a new (scalable) client on a spare by executing the command:

EXEC create_client *'new_client'*

The *'new_client'* is the name of the spare on which the new client should be created. The node executing the command initiates in particular the meta-data of the new one. It does not seem useful at present to let the creation of several clients in one command.

♣ **Scalable Peer Creation.** The application creates new scalable peers at given spares using the command:

EXEC create_scalable_peer *'new_peer1[, new_peer2]...'*

The command semantics is similar to the above one for the server nodes. In the absence of an existing scalable server, a script file creates the first ever (initial) SD-SQL Server peer at a collection of spares.

- ♣ **Scalable Node Removal.** The application can remove a scalable node from SD-SQL Server by executing the *drop_ node* command as below:

EXEC drop_ node *'node_name'*

The result of the scalable command is an SD-SQL Server without the node *'node_name'*. As the need for the operation seems rare, a (more complex) command removing simultaneously several nodes does not appear useful at present.

Example. The command below returns *Node211* node, perhaps a client, or server or peer, to the state of a spare.

EXEC drop_ node *'N211'*

3.5 Related Works

Efficient parallel and distributed database partitioning has been studied for many years, [12]. It naturally triggered the work on the reorganizing of the partitioning, with notable results as early as in 1996, [11]. The common goal was a global reorganization, unlike for our system.

The editors of [11] contributed themselves with two on-line reorganization methods, termed respective *new-space* and *in-place* reorganization. The former method created a new disk structure, and switches the processing to it. The latter approach balanced the data among existing disk pages as long as there was room for the data. Among the other contributors to [11], concerned a command named '*Move Partition Boundary*' for Tandem Non Stop SQL/MP. The command aimed on on-line changes to the adjacent database partitions. The new boundary should decrease the load of any nearly full partition, by assigning some tuples into a less loaded one. The command was intended as a manual operation. We could not find whether it was ever realized.

A more recent proposal of efficient global reorganizing strategy is in [10]. One proposes there an automatic advisor, balancing the overall database load through the periodic reorganizing. The advisor is intended as DB2 offline utility. Another attempt, in [4], the most recent one to our knowledge, describes yet another sophisticated reorganizing technique, based on the database clustering. Termed AutoClust, the technique mines for the closed sets, then groups the records according to the resulting attribute clusters. The AutoClust processing should to start when the average query response time drops below a user defined threshold. It is unknown whether AutoClust was put into practice.

With respect to the partitioning algorithms used in other major DBMSs, the parallel DB2 uses the (static) hash partitioning. Oracle offers both, hash and range partitioning, but over the shared disk multiprocessor architecture only. How the scalable tables may be created at these systems remains an open research problem.

5 Conclusion

The proposed syntax and semantics of SD-SQL Server commands make the use of scalable tables about as simple as that of the static ones. It lets the user/application to easily take advantage of the new capabilities of our system. Through the scalable distributed partitioning, they should allow for much larger tables or for a faster response time of complex queries, or for both.

The current design of our interface is geared towards the "proof of concept" prototype. It is naturally simplified here and there with respect to a full-scale system. Further work could concern alternate design choices. It may also address additional scalable commands. Both directions concern especially the capabilities for the image and node management.

For the images, one could design for instance a scalable *Create_View* command, providing the transparency of the local image names. The command could furthermore allow for the full transparency of the secondary images, making them a kind of a local caches. For the nodes, one could design, for

instance, the command *Connect* uniting separate configurations of SD-SQL Server nodes into one. All such issues lead to various implementation level choices, to study in consequence.

Acknowledgments

We thank J. Gray (Microsoft BARC) for providing the SkyServer database and for the counselling crucial to this work, and G. Graefe (Microsoft) for information on SQL Server linked servers' capabilities. The initial support for this work came partly from the research grant of Microsoft Research.

References

1. Ben-Gan, I., and Moreau, T. Advanced Transact SQL for SQL Server 2000. Apress Editors, 2000
2. Gray, J. & al. Data Mining of SDDS SkyServer Database. WDAS 2002, Paris, Carleton Scientific (publ.)
3. Gray, J. The Cost of Messages. Proceeding of Principles Of Distributed Systems, Toronto, Canada, 1989
4. Guinepain, S & Gruenwald, L. Research Issues in Automatic Database Clustering. ACM-SIGMOD, March 2005
5. Lejeune, H. Technical Comparison of Oracle vs. SQL Server 2000: Focus on Performance, December 2003
6. Litwin, W., Neimat, M.-A., Schneider, D. LH*: A Scalable Distributed Data Structure. ACM-TODS, Dec. 1996
5. Litwin, W., Neimat, M.-A., Schneider, D. Linear Hashing for Distributed Files. ACM-SIGMOD International Conference on Management of Data, 1993
7. 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
8. 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
9. Microsoft SQL Server 2000: SQL Server Books Online
10. Rao, J., Zhang, C., Lohman, G. and Megiddo, N. Automating Physical Database Design in a Parallel Database, ACM SIGMOD '2002 June 4-6, USA
11. Salzberg, B & Lomet, D. Special Issue on Online Reorganization, Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, 1996
12. Özsu, T & Valduriez, P. Principles of Distributed Database Systems, 2nd edition, Prentice Hall, 1999.
13. Soror Sahri, Witold Litwin, «*SD-SQL Server: a Scalable Distributed Database System*», CERIA Research Report 2005-03-05, March 2005