ARCHITECTURE AND INTERFACE OF SCALABLE DISTRIBUTED DATABASE SYSTEM SD-SQL Server

Witold Litwin, Soror Sahri, Thomas Schwartz
CERIA, Paris-Dauphine University
75016 Paris
France
Witold.Litwin@dauphine.fr, Soror.Sahri@dauphine.fr, tjschwartz@scu.edu

ABSTRACT
We present a scalable distributed database system called SD-SQL Server. Its original feature is the dynamic and transparent repartitioning of growing tables, avoiding the cumbersome manual repartitioning characterizing the current technology. SD-SQL Server re-partions a table when an insert overflows its existing segments. With the comfort of a single node SQL Server user, the SD-SQL Server user disposes of larger tables or gets a faster response time through the dynamic query parallelism. We present the architecture of our system, and its user/application interface.

KEY WORDS
Scalable table, scalable database, dynamic table partitioning, scalable distributed data structure.

1. Introduction

Databases are now often huge and growing at a high rate. Large tables are then typically hash or range partitioned into segments stored at different storage sites. Current Data Base Management Systems (DBSs), e.g., SQL Server, Oracle or DB2, provide static partitioning only [1],[5],[11]. The database administrator (DBA) in need to spread these tables over new nodes has to manually redistribute the database (DB). A better solution has become urgent, [1].
This situation is similar to that of file users forty years ago in the centralized environment. Efficient management of distributed data present specific needs. The Scalable Distributed Data Structures (SDDSs) addressed these needs for files, [6][7]. An SDDS scales transparently for an application through distributed splits of its buckets, hash, range or k-d based. In [8], 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 image adjustment is lazy. It occurs only when a query to the scalable table finds an outdated image. 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 as it stands in its 2005 version, [14]. The current architecture addresses more features than [8]. With respect to the interface, we discuss the syntax and semantics of each command. Numerous examples illustrate the actual use of SD-SQL Server. We hope to convince that the use of the scalable tables should be about as simple as of the static ones in practice.
The related papers discussed the internal design and the processing performance of SD-SQL Server, [9], [14]. The scalable table processing creates an overhead and our design challenge was to minimize it [3]. 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 the presentation.

2. SD-SQL Server Architecture

Fig 1 shows the current SD-SQL Server architecture, adapted from the reference architecture for an SD-DBS in [8]. The system is a collection of SD-SQL Server nodes. An *SD-SQL Server node* is a linked SQL Server node that in addition is declared as an SD-SQL Server node. This declaration is made as an SD-SQL Server command or is part of a dedicated SQL Server script run on the first node of the collection. We call the first node the *primary node.* The primary node registers all other current SD-SQL nodes. We can add or remove these dynamically, using specific SD-SQL Server commands. The primary node registers the nodes on itself, in a specific SD-SQL Server database called the *meta-database* (MDB). An *SD-SQL Server database* is an SQL Server database that contains an instance of SD-SQL Server specific *manager* component. A node may carry several SD-SQL Server databases.

We call an SD-SQL Server database in short a *node database* (NDB). NDBs at different nodes may share a (proper) database name. Such nodes form an SD-SQL Server *scalable (distributed) database* (SDB). The common name is the *SDB name*. One of NDBs in an SDB is *primary*. It carries the meta-data registering the current NDBs, their nodes at least. SD-SQL Server provides the commands for scaling up or down an SDB, by adding or dropping NDBs. For an SDB, a node without its NDB is (an SD-SQL Server) *spare* (node). A spare for an SDB may already carry an NDB of another SDB. Fig 1 shows an SDB, but does not show spares.

Each manager takes care of the SD-SQL Server specific operations, the user/application command interface especially. The procedures constituting the manager of an NDB are themselves kept in the NDB. They apply internally various SQL Server commands. The SQL Servers at each node entirely handle the inter-node communication 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 internal changes of the latter.

An SD-SQL Server NDB is a *client*, a *server*, or a *peer*. The client manages the SD-SQL Server node user/application interface only. This consists of the SD-SQL Server specific commands and from the SQL Server commands. As for the SQL Server, the SD-SQL specific commands address the schema management or let to issue the queries to scalable tables. Such a *scalable* query may invoke a scalable table through its image name, or indirectly through a *scalable* view of its image, involving also, perhaps, some *static* tables, i.e., SQL Server only..

Internally, each client stores the images, the local views and perhaps *static* tables. These are tables created using the SQL Server *CREATE TABLE* command (only). It also contains some SD-SQL Server meta-tables constituting the catalog **C** at the figure. The catalog registers the client images, i.e., the images created at the client.

When a scalable query comes in, the client checks whether it actually involves a scalable table. If so, it must address its image, directly or through a scalable view. The client searches therefore for the images that the query invokes. For every image, it checks whether it conforms to the actual partitioning of its table, i.e., unions all the existing segments. We recall that a client view may be outdated. The client uses **C**, as well as some server meta-tables pointed to by **C**, defining the actual partitioning. The manager dynamically adjusts any outdated image. In particular, it changes internally 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* NDB stores the segments of scalable tables. Every segment at a server 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 NDB, named **S** in the figure. The meta-data in **S** identify the scalable table each segment belongs to. They indicate also the segment size. Next, they indicate the servers in the SDB that remain available for the segments created by the splits at the server NDB. 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, by 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). 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* NDB is both a client and a server NDB. 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 form logically the catalog termed **P** at the figure. This one is operationally, the union of **C** and **S** catalogs.

To illustrate the architecture, Fig 1 shows the NDBs of some SDB, on nodes *D1…Di+1*. The NDB at *D*1 is a client NDB that thus carries only the images and views, especially the scalable ones. This node could be the primary one, being only of type peer or client. It interfaces the applications. The NDBs on all the other nodes till *Di* are servers. They carry only the segments and do not interface any applications. The nodes could be peer or server, only. Finally, the NDB at *Di*+1 is a peer, providing all the capabilities. Its node has to be a peer node. The NDBs carry a scalable table termed *T*. The table has a scalable index *I*. We suppose that *D1* carries 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 *Di+1* carry 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. None of the two images refers to this segment as yet. Each will be actualized only once it gets a scalable query to *T*. The split has also created the new segment of *I*.
Notice finally in the figure that segments of *T* are all named *_D1_T*. This represents the couple (creator node, table name). Notice here only that the name provides the uniqueness with respect to different client (peer) NDBs in an SDB.

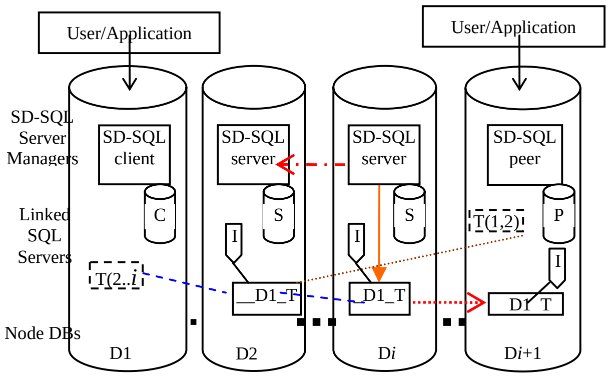

Fig 1 SD-SQL Server Architecture

3. Application Interface

3.1 Overview

The application manipulates SD-SQL Server objects essentially through new SD-SQL Server dedicated commands. The commands for the tables and views perform the usual SQL schema manipulations and queries implying however now the scalable tables (through the images) or the (scalable) views of the scalable tables. 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.
Most SD-SQL Server 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.
We now present the syntax and semantics of the SD-SQL Server commands. The rule for an SD-SQL Server command performing an SQL operation is to use the SQL command name (verb) prefixed with *'sd_'* and with all the blanks replaced with '_'. Thus, e.g., SQL *SELECT* became SD-SQL *sd_select*, while SQL *CREATE TABLE* became *sd_create_table*. The standard SQL clauses, with perhaps the additional parameters, follow the verb, specified as usual for SQL. The whole specification is however within additional quotes ' '. The rationale is that SD-SQL Server commands are implemented as SQL Server stored procedures. The clauses pass to SQL Server as the parameters of a stored procedure and the quotes around the parameter list are mandatory.

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 multi-database 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, [14], [15]. 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 illustrate the discussion of the commands by numerous examples. They have the common denominator of our benchmark application that is the SDB named SkyServer. The choice follows the actual SkyServer DB, [2]. We particularly use the data from the original *PhotoObj* table to experiment with a scalable *PhotoObj* table. In the examples, we also use our actual node names.
We start with the node management commands that create, alter or drop SD-SQL Server nodes, SDBs and NDBs. We continue with the commands for the scalable table management, including the management of the scalable indexes and of images. We end up discussing the commands for the scalable search and update queries.

3.2 Node Management

A script file creates the first ever (primary) SD-SQL Server (scalable) node at a collection of linked SQL Server nodes. One can create the primary node as peer or server, but not at a client. After that, the node and then any other SD-SQL Server node created subsequently offer the following *(scalable) node management* commands, to the administrator, user or application.
Node creation. One expands the existing SD-SQL Server configuration with new nodes through the following command:

sd_create_node 'new_node[, node_type]

The *'new_node'* parameter is the name of the spare for the new node. The node executing the command initiates in particular the meta-data of the new one, according to its *'node_type'* parameter. It can be *server*, *client* or *peer*. The default is *server*.
Example 1. The script has created the primary SD-SQL Server node at SQL Server linked node at our *Dell1* machine. We could set up this node as server or peer, consider that we made the latter choice. The following commands issued at Dell1 create further nodes:

sd_create_node 'Dell2' /* Server by default */

sd_create_node 'Dell3, 'client'

sd_create_node 'Ceria1','peer'

Node removal. One removes an SD-SQL Server node from the current configuration through the command:

sd_drop_ node 'node_name'

The dropped node remains an SQL Server linked node. The removal of a node drops all the NDBs on it. Details depend on the NDB type, see below.

Example 2. The previously created *Ceria1* node quits SD-SQL Server. It remains an SQL Server linked node.

sd_drop_ node 'Ceria1'

Node alteration. An application can upgrade a client or server node into a peer. It may alternatively downgrade a peer. The command is:

sd_alter_node 'node_name', 'ADD/DROP client/server'

Example 3. We upgrade *Dell3* from Example 1:

sd_alter_ node 'Dell3', 'ADD server'

3.3 Scalable Database Management

Creation. We create an SDB using the command:

sd_create_scalable_database 'db_name', ['node_name'], ['type'] ['extent']

The SDB *'db_name'* has its primary NDB at node *'node_name'*. The (optional) *'extent'* parameter should have the value $n > 1$. By default, $n = 1$. The command creates n NDBs, including the primary one. Each bears the name *'db_name'* for SQL Server. The *'type'* indicates whether the primary NDB of the scalable database is a server or peer. By default, the primary NDB inherits the type of its node.

Example 4. We create our *SkyServer* SDB at *Dell1*.

sd_create_scalable_database 'SkyServer, 'Dell1'

As the result, our primary *SkyServer* NDB is a server NDB.

Alteration. We alter an SDB, by creating an NDB or dropping one. For the creation, we use the command:

sd_create_node_database 'sdb_name', ['node_name',] ['type',]

The name of the new NDB for SD-SQL Server, as well as for SQL Server is *'sdb_name'*. The *'node_name'* is optional. If specified, then the command creates the NDB there. By default, SD-SQL Server either creates the NDB on the node of the command, if it does not exist there yet, or randomly selects a node.

The *'type'* limits the capabilities of the NDB, if created at a peer node. Otherwise, the NDB inherits the node type. The *sd_drop_node_database* command preserves the segments of tables created at peer or client NDBs at other nodes, including related meta-data. It saves them at another NDB. These segments obviously should not disappear.

We drop an NDB by the command:

sd_drop_node_database 'sdb_name', 'node_name'

Example 5. Our above created *Skyserver* SDB has up to now only one NDB that is a server DB, To query it, one needs at least one client or peer NDB. We append a client NDB at *Dell3* to our *SkyServer* SDB. We can do it, since *Dell3* was created as a client node in Example 1.

sd_create_node_database 'SkyServer', '*Dell3*', 'client'

From now on, the *Dell3* user opens *Skyserver* SDB through the usual SQL Server USE *Skyserver* command (which actually opens *Dell3.Skyserver* NDB).

Removal. We drop an SDB using the command:

sd_drop_scalable_database 'db_name'

The command drops all the NDBs of the SDB with all their content.

Example 6. The command below drops the *SkyServer* SDB. It thus removes all the above created NDBs, e.g., at *Dell1*, and *Dell2*.

sd_drop_scalable_database 'SkyServer'

3.4 Scalable table management

Table Creation. The application on client (peer) node *D* creates a scalable table *T* by invoking:

sd_create_table 'SQL: Create Table **T** clauses', 'Segm_size' [, 'Partition_Key']

The parameter *'SQL: Create Table **T** clauses'* is the text of the SQL Server *CREATE TABLE T* command clauses following the command name itself. The prototype supports the local creation only at present, i.e., of scalable table *T* in NDB currently in use at *D*. The SQL command clauses have to respect all the constraints that SQL Server imposes at an updatable distributed partitioned view [10]. The scalable table has to have its partition key among the key attributes. The *check constraints*, [10], defined at each segment (automatically for SD-SQL Server, but not for SQL Server) should partition the partition key space. The partition key may be not (entire) primary key. SD-SQL Server, like SQL Server, allows therefore for the duplicated values of the partition key in the scalable table.

The *Segment_size* parameter fixes the maximal size of a segment of *T*. The '*Partition_Key*' parameter indicates the partition key. It is optional and makes sense only for tables with composite keys . By default, SD-SQL Server chooses the 1st key attribute appearing in the attribute declaration clause of *T*. Clever choice of the partitioning key may speed up some queries, e.g., with joins on the primary and foreign key attribute.

The command creates a scalable table only. To create a static table, e.g., to avoid the above-mentioned constraints on the scalable ones, one should use the SQL Server *CREATE TABLE* command.

Example 7. The *Dell3* user of *Skyserver* wishes to create the scalable table *PhotoObj*, upon the static one with the same name, [2]. The user wishes the segment capacity of 10000 tuples, for the efficient distributed query processing. It applies the command:

sd_create_table 'PhotoObj (objid BIGINT PRIMARY KEY…)', 10000

The partition key of *PhotoObj* is its *objid* attribute.

The user creates furthermore the scalable table *Neighbors*, modelled upon the static table with the same name in *Skyserver*, [2]. That table has three key attributes. The *objid* is one of them and is the foreign key of *PhotoObj*. For this reason, the user wishes it to be the partition key. Finally, s/he chooses the segment capacity to be 500 tuples. Accordingly, the user issues the command:

sd_create_table 'Neighbors (htmid BIGINT, objid BIGINT, Neighborobjid BIGINT) ON PRIMARY KEY…)', 500, 'objid'

Table Alteration. To alter scalable table *T*, the application executes the command:

sd_ alter_table ['SQL:'ALTER TABLE T clauses], [new segment_size]

The command carries at least one of its 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 effect of the decrease to the segment size is lazy. No segment splits before next insert into it.

Example 8. The *Dell3* user adds a column to *PhotoObj* and changes its segment size:

sd_alter_table *'PhotoObj ADD t INT, 10000*

Indexes. SQL Server does not allow for indexed distributed partitioned views at present. It does use however the local indexes on the tables under the view to accelerate the query processing, whenever they exist. SD-SQL Server lets therefore the scalable tables to have the scalable distributed indexes. The segments of such an index are the local indexes on the segments of the table. An application creates or removes a scalable distributed index *I*, for a column of a scalable table *T*, by the commands:

sd_create_index ['SQL: Create Index I ON T clauses']

sd_drop_index 'SQL: Drop Index T.I clauses'

Example 9 We issue, e.g., at *Dell3* node, the following command, to create *run_index* scalable index on *run* column of *PhotoObj*.

sd_create_index 'run_index ON Photoobj (run)'

Splits of *PhotoObj* will propagate *run_index* to any new segment.

Table Removal. The user removes scalable table *T* using the command:

sd_drop_table ['SQL: DROP TABLE T clauses']

The command removes the primary image if *T* and all *T* segments. The syntax and semantics of the parameter are those of the SQL Server *DROP TABLE* clauses.

Example 10. Drop the scalable table *PhotoObj*, created by *Skyserver* user at *Dell3*:

sd_drop_table 'Dell3. SkyServer.PhotoObj'

Secondary Image. The application creates a secondary image of scalable table *T*, created at node *N* by issuing the command:

sd_create_image '[image_node]', '[*N*,]' 'T'

At any node, only one image of a given table can exist. *N* is not necessary for the command invoked at this node. Likewise, *image_node* is not necessary for the command at the image node. The local secondary image name is *SD.N_T*, [14]. This naming avoids the conflict between images and segment of scalable tables sharing the proper name while created at different client or peer NDBs. The application wishing to use another name for an image, e.g., *T*, may do it through *CREATE VIEW*.
The application removes a secondary image using the command:

sd_drop_image '[image_name]'

Example 11. The user at *Dell3* creates the (secondary) image of *PhotoObj* at *Ceria1* node through the command:

sd_create_image 'Ceria1', 'PhotoObj'

The *Skyserver* user at *Ceria1* wishing to remove this image issues the command:

sd_drop_image 'SD.Dell3_Photoobj'

3.5 Scalable Queries

Scalable Table Search. An application searches scalable tables through *sd_select* command with the following syntax:

sd_select 'SQL: Select clauses'[, Segment Size'][, 'Primary Key']; [, 'Partition 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'* etc parameters are optional. They serve *SELECT INTO* clause creating a scalable table. The *'Primary Key'* and *'Partiton Key'* address the new table, whenever needed. The columns can be inherited from the source tables, or created by the query e.g., by the aggregate functions. The application may use in the query any SQL Server table or view name, i.e., local or prefixed. Only the local names may however designate a scalable table or view at present.

Example 12. Once our *Dell3* application opens *Skyserver* SDB (Example 5), it queries for all the data in *PhotoObj* simply as follows.

USE *Skyserver* /* SQL Server command */
sd_select '* FROM PhotoObj'

Next, the application creates the scalable table *PhotoObj1*. It chooses the segment size of 500 tuples. The new table inherits *Objid* as both the primary and the partition key.

sd_select '* INTO PhotoObj1 FROM PhotoObj', 500

Scalable Table Modification. An application modifies a scalable table through the SD SQL Server commands:

sd_ insert 'SQL Insert clauses'

sd_update 'SQL: Update clauses'

sd_delete 'SQL: Delete clauses'

Here, an SQL clauses input are respectively the standard SQL Server *INSERT*, *UPDATE* and *DELETE* command clauses. They may include the *SELECT* clause on scalable or static tables.

Example 13. The following command executed at some SD-SQL Server node changes the *run* column values to 752 for 10 tuples of our *PhotoObj*, leading with respect the ascending order on *objid*.

```
USE dell3.SkyServer
sd_update 'PhotoObj
SET run= 752 WHERE objid IN
       (SELECT TOP 10 objid FROM PhotoObj')
```

4. Related Works

We discuss our implementation of the SD-SQL Server commands and we show it efficient in [9], [14] and [15]. More generally, the parallel and distributed database partitioning has been studied since years, [13]. It naturally triggered the work on the reorganizing of the partitioning, with results already in 1996, [12]. The aim was at a global reorganization, unlike for our system.

The editors of [12] 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 [12], 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 [11]. 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. All well-known DBSs support the distributed partitioned union-all views. Only SQL Server let such views be updatable. This is the rationale for our choice. How the scalable tables may be created at other main DBSs remains an open problem.

5. Conclusion

The 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 constrained by the internal processing capabilities of our "proof of concept" prototype, [14]. It is simplified with respect to a full-scale system. Further work could lift these limitations.

Acknowledgments. We thank J.Gray (Microsoft BARC) for the original SkyServer database and for advising this work from its start. MS Research partly funded this work, relaying the support of CEE project ICONS.

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
7. Litwin, W., Neimat, M.-A., Schneider, D. Linear Hashing for Distributed Files. ACM-SIGMOD International Conference on Management of Data, 1993
8. 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
9. 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
10. Microsoft SQL Server 2000: SQL Server Books Online
11. Rao, J., Zhang, C., Lohman, G. and Megiddo, N. Automating Physical Database Design in a Parallel Database, ACM SIGMOD '2002 June 4-6, USA
12. Salzberg, B & Lomet, D. Special Issue on Online Reorganization, Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, 1996
13. Özsu, T & Valduriez, P. Principles of Distributed Database Systems, 2nd edition, Prentice Hall, 1999.
14. Litwin, W., Sahri, S., Schwarz, Th. SD-SQL Server: a Scalable Distributed Database System. CERIA Research Report 2005-12-13, December 2005.
15. Litwin, W., Sahri, S., Schwarz, Th. Scalable Command Processing in SD-SQL Server: a Scalable Distributed Database System. 7th Intl. Workshop on Distributed Data and Structures (WDAS-7) Santa Clara, CA, 2006.