Scalable Web Services Interface for SD-SQL Server

Soror Sahri
CERIA, Université Paris-Dauphine
Place de Lattre de Tassigny
75016 Paris, France
Soror.sahri@dauphine.fr

Witold Litwin
CERIA, Université Paris-Dauphine
Place de Lattre de Tassigny
75016 Paris, France
witold.litwin@dauphine.fr

Thomas Schwarz
Santa Clara University
500 El Camino Real
California, USA
tjschwarz@scu.edu

ABSTRACT
SD-SQL Server is a scalable distributed database system. Its original feature is dynamic and transparent repartitioning of growing tables. It avoids the cumbersome manual repartitioning necessary with current technology. SD-SQL Server re-partitions a (distributed) table when an insert overflows existing segments. To its user, SD_SQL offers the comfort of a single node, while allowing the larger tables and faster response time made possible by dynamic parallelism. We present the architecture of our system and its command interface. We present the Extended Web Services (EWS) Interface we have recently added to SD-SQL Server. We study the relative EWS query speed. It remains insufficient for larger data sets to be retrieved.

Keywords
Scalable table, scalable database, dynamic partitioning, scalable Web services.

1. INTRODUCTION
Today, many databases contain fast growing tables that ultimately become very large. The current state-of-the-art DBMSs (DB2, Oracle, SQL Server, Postgress, MySQL…) accommodate large tables by partitioning and distributing them over several sites. These systems allow only static partitioning [1, 4, 5, 15]. Whenever an adjustment is needed, the database administrator needs to manually repartition. Our proposal, SD-SQL Server, avoids this cumbersome task by dynamically adjusting table growth through autonomous splitting of table segments [9, 11, 12, 13, 14]. SD-SQL is a distributed system that uses the services of SQL server. The transparent growth of SD-SQL's central data structure is the defining feature of a Scalable Distributed Data Structures (SDDS) [6, 7, 8].

The salient features of an SD-DBS are *scalable* tables. Each table consists of distributed segments that are relational tables. As in any SDDS, the application (client) is aware of neither the number of segments nor their locations. Instead, the client has a local view (its *image*) of the scalable table.

The image is sufficient to manipulate the table successfully. During internal processing, an incoming query might use an outdated image at a client. However, the system checks the accuracy of the image and adjusts it if necessary as part of the query execution.

Our implementation uses SQL Server as the underlying DBMS since SQL Server allows updating partitioned views using check constraints. To our knowledge, SQL Server is the only mainstream database with this capability. For every standard SQL command under SQL Server, we provide an SD-SQL Server command for the corresponding action on a scalable table or a scalable view.

Recently, we have extended SD-SQL Server to support an Enhanced Web Services (EWS) interface. We call it the EWS-SDSQL Server interface. To implement it, we mapped each SD-SQL Server command to a Web service using the SQL endpoints new feature of SQL Server 2005. In this article, we report on its impact on SD-SQL.

We first recall the architecture of SD-SQL Server and its interface. Related papers [10, 13, 16] discuss the implementation. We then present the EWS SDSQL Server. Scalable table processing creates an overhead and our design challenge was to minimize it. The performance analysis we report proves that this overhead is negligible for practical purpose. We then discuss the EWS-SDSQL Server performance. We observed that it is several times slower when we retrieve large data sets. The slowdown of the EWS interface is so large that it seems hardly useful for practical purposes. The problem seems to be rooted deep within the SQL Server implementation. We expect that our interface will offer much better performance once SQL Server has overcome this limitation. Future work will prove or disprove our expectation.

The remainder of the paper is organized as follows: Section 2 presents the SD-SQL Server architecture. The first part of Section 3 recalls the basics of the user interface. The second one presents EWS-SDSQL, the scalable Web services interface upon SD-SQL Server. Section 4 shows the experimental performance analysis. Section 5 concludes the presentation.

2. SD-SQL SERVER ARCHITECTURE
Figure 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

INFOSCALE 2008, June 4-6, Vico Equense, Italy
 978-963-9799-28-8
DOI 10.4108/ICST.INFOSCALE2008.3517

nodes. We can add or remove these dynamically, using specific SD-SQL Server commands. The primary node registers the nodes in a specific, local 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. Figure 1 shows an SDB, but does not show spares.

Each manager takes care of the SD-SQL Server specific operations and in particular of the user/application command interface. The procedures constituting the manager of an NDB are themselves kept in the NDB. They apply various SQL Server commands internally. The SQL Servers at each node handle the inter-node communication and the distributed execution of SQL queries entirely. In this sense, each SD-SQL Server runs on 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 only the SD-SQL Server node user/application interface. 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 issue 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. It also contains some SD-SQL Server meta-tables constituting the catalog **C** at Figure 1. 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, the query must address the local image of the table. It can do so directly through the image name, 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** that define 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 catalogue in each server NDB, called **S** in the figure. The meta-data in **S** identify the scalable table each segment belongs to. They indicate also the segment size. In addition, they give the servers in the SDB that remain available for the segments created by the splits at the server NDB. Finally, for a *primary* segment, i.e., the first segment created for a scalable table, the meta-data at its server provides the actual partitioning of the table.

Each segment has an *AFTER* trigger attached, not shown in the figure. After each insert, the trigger checks whether the segment overflows. If this is the case, then the server splits the segment by range partitioning it with respect to the table (partition) key. It moves enough upper tuples out so that the remaining (lower) tuples fit into the splitting segment. To accommodate the migrating tuples, the server creates 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 allows updates to the SQL Server distributed partitioned and in particular inserts and deletions. 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 (called **P** in the figure). This catalogue is operationally the union of the **C** and **S** catalogs.

Every SD-SQL Server node is *client*, *server* or *peer* node. The peer accepts every type of NDB. The client nodes only carry client NDBs and server nodes accept server NDBs only. Only a server or peer node can be the primary node or may carry a primary NDB. To illustrate the architecture, Figure 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, provided it is a peer. It interfaces the applications. The NDBs on all the other nodes until *Di* are server NDBs. They carry only the segments and do not interface (directly) any applications. The NDB at *D2* could be here the primary NDB. Nodes *D2*…*Di* could be peers or (only) servers. 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 called *T*. The table has a scalable index *I*. We suppose that *D*1 carries the *primary* image of *T*, named *T* at the figure. This image name is also as the SQL Server view name implementing the image in the NDB. SD-SQL Server creates the primary image at the node requesting the creation of a scalable table, while creating the table. Here, the primary segment of table *T* is supposed at *D*2. Initially, the primary image included only this segment. It has evolved since, following the expansion of the table at new nodes, and now is the distributed partitioned union-all view of *T* segments at servers *D*2…*Di*. We symbolize this image with the dotted line running from image *T* till the segment at *Di*. Peer *Di*+1 carries a *secondary* image of table *T*. Such an image interfaces the application using *T* on a node other than the table creation one. This image, named *D1_T*, for reasons we discuss below, differs from the primary image. It only includes the primary segment. We symbolize it with the dotted line towards *D*2 only. Both images are outdated. Indeed, server *Di* just split its segment and created a new segment of *T* on *Di*+1. The arrow at *Di* and that towards *Di*+1 represent this split. As the

result of the split, server *Di* updated the meta-data on the actual partitioning of *T* at server *D*2 (the dotted arrow from *Di* to *D*2). The split has also created the new segment of the scalable index *I*. None of the two images refers as yet to the new segment. Each will be actualized only once it gets a scalable query to *T*. At the figure, they are getting such queries, issued using respectively the SD-SQL Server *sd_select* and *sd_insert* commands. We discuss the SD-SQL Server command interface in the next sections.

Notice finally that the segments of *T* in the figure are all named *_D1_T*. This represents the couple (creator node, table name). It is the proper name of the segment as an SQL Server table in its NDB. Similarly for the secondary image name, except for the initial '_'. The image name is the local SQL Server view name.

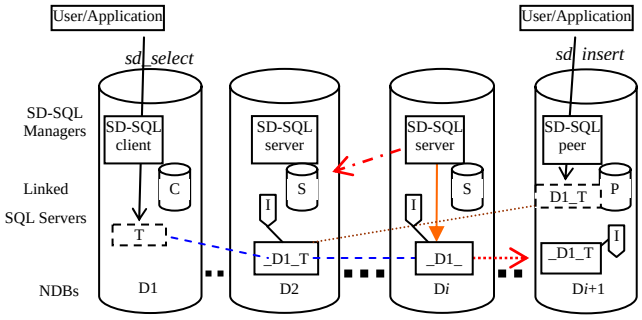

Figure 1. SD-SQL Server Architecture

3. APPLICATION INTERFACE

3.1 Overview

The application manipulates SD-SQL Server objects essentially through new SD-SQL Server dedicated commands. Some commands address the node management, including the management of SDBs, NDBs. Other commands manipulate the scalable tables. These commands perform the usual SQL schema manipulations and queries that can however now involve scalable tables (through the images) or (scalable) views of the scalable tables. We call the SD-SQL Server commands scalable. A scalable command may include additional parameters specific to the scalable environment, with respect to its static (SQL Server) counterpart. Most of scalable commands apply also to static tables and views.

Details of all the SD-SQL Server commands are in [12, 13, 14]. 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 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 should suffice for many applications. The *SELECT* statement in a scalable query supports the SQL Server allowed selections, restrictions, joins, sub-queries, aggregations, aliases…etc. It also allows for the INTO clause that can create a scalable table. However, queries to the scalable multi-database views are presently not possible. The reasons are the limitation of the SQL Server meta-tables that SD-SQL Server uses for the parsing. Moreover, the *sd_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, including the *INTO* clause. The *sd_update* and *sd_delete* commands 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 recall the SD-SQL Server command interface modelled upon our benchmark application , namely SkyServer DB, [2].

From now on, a *Dell3* user opens *Skyserver* SDB. The *Skyserver* users are now able to create scalable tables. The *Dell3* user starts with a *PhotoObj* table modelled on the static table with the same name, [2]. The user sets a segment capacity of 10000 tuples. S/he chooses this parameter for the efficient distributed query processing. S/he also sets the *objid* key attribute to be the partition key. In SD-SQL Server, a partition key of a scalable table has to be a single key attribute. The requirement comes from SQL Server, where it has to be the case of a table, or tables, behind a distributed partitioned updatable view. The key attribute of *PhotoObj* is its *objid* attribute. The user issues the command:

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

We did not provide the complete syntax, using '…' to denote the rest of the scheme beyond the key attribute. The *objid* attribute is the partition key implicitly, since it is here the only key attribute. The user creates furthermore a scalable table *Neighbors*, modelled upon the similar one in the static *Skyserver*. 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. The segment capacity should now be 500 tuples. Accordingly, the user issues the command:

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

The user indicated the partition key. The implicit choice would go indeed to *htmid*, as the first one in the list of key attributes. The *Dell3* user decides furthermore to add attribute *t* to *PhotoObj* and prefer a smaller segment size:

sd_alter_table 'PhotoObj ADD t INT, 1000

Next, the user decides to create a scalable index on *run* attribute:

sd_create_index 'run_index ON Photoobj (run)'

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

The *PhotoObj* creation command created the primary image at *Dell3*. The *Dell3* user creates now the secondary image of *PhotoObj* at *Ceria1* node for the *SkyServer* user there:

sd_create_image 'Ceria1', 'PhotoObj'

The image internal name is *SD.Dell3_Photoobj*, as we discuss in [10]. Once the *Ceria1* user does not need its image anymore, s/he may remove it through the command:

sd_drop_image 'SD.Dell3_Photoobj'

Assuming that the image was not dropped however yet, our *Dell3* user may open *Skyserver* SDB and query *PhotoObj*:

*sd_insert 'INTO PhotoObj SELECT * FROM Ceria5.Skyserver-S.PhotoObj*

*sd_select 'TOP 5000 * INTO PhotoObj1 FROM PhotoObj', 500*

The first query loads into our *PhotoObj* scalable table tuples from some other *PhotoObj* table or view created in some *Skyserver* DB at node *Ceria5*. This DB could be a static, i.e., SQL Server only, DB. It could alternatively be an NDB of "our" *Skyserver* DB. The second query creates a scalable table *PhotoObj1* with segment size of 500 and copies there 5000 tuples from *PhotoObj*, having the smallest values of *objid*. See [10] for examples of other scalable commands.

3.2 SCALABLE WEB SERVICES INTERFACE

Here, we present the SD-SQL Server commands through a full extended Web services (EWS) interface. The WS-SDSQL is our generic name for such interface. Through its methods, WS-SDSQL interface would act as document repository protocol for data storage and retrieval. The messaging should thus use EWS compatible SOAP specs and the service description should be available through WSDL. The EWS-SDSQL methods could be called from a more extensive interface, using also EWS protocols.

As SD-SQL Server is implemented upon SQL Server, we use the new feature of SQL Server 2005 to create the EWS-SDSQL interface. SQL Server 2005's HTTP/SOAP endpoints provide SQL Server with new capabilities for using Web services within SQL Server. An endpoint is a service that listens for requests natively within the server. Each endpoint supports a protocol, which can be TCP or HTTP, and a payload type, which can include support for database mirroring, service broker, T-SQL, or SOAP.

To set up a Web service, we create an HTTP endpoint on the server. That endpoint can expose a stored procedure as a Web method. As the SD-SQL Server commands are SQL stored procedures, so we map these stored procedures to web methods by creating an SQL endpoint. To do so, we use the *SkyServer* database benchmark presented as a scalable database on SD-SQL Server. We map all its SD-SQL Server commands to web methods as in **Erreur ! Source du renvoi introuvable.**. For example, we map the *sd_select* command to *SdSelect* Web method.

Once the SD-SQL Server commands are mapped into Web methods through the *SkyServer* endpoint above, we can call each of the corresponding command as a Web service from a user interface. We have developed this interface using C# language.

The limitation with Web services, in this context, is the use of distributed partitioned views. The call of a Web service functions only when a partitioned view addresses local segments. If a partitioned view addresses distributed segment, the Web service call fails.

```
CREATE ENDPOINT SkyServer
   STATE = STARTED
AS HTTP
(  PATH = '/SkyServer,
   AUTHENTICATION = (INTEGRATED),
   PORTS = (CLEAR),
   SITE = 'localhost'  )
FOR SOAP
(    WEBMETHOD 'SdCreateTable'
      (NAME='eGovBus.dbo.sd_create_table'),
     WEBMETHOD 'SdAlterTable'
      (NAME=SkyServer.dbo.sd_alter_table'),
     WEBMETHOD 'SdCreateIndex'
      (NAME= SkyServer.dbo.sd_create_index'),
     WEBMETHOD 'SdDropIndex'
      (NAME= SkyServer.dbo.sd_drop_index'),
     WEBMETHOD 'SdDropTable'
      (NAME= SkyServer.dbo.sd_drop_table'),
     WEBMETHOD 'SdSelect'
      (NAME= SkyServer.dbo.sd_select'),
      WEBMETHOD 'SdInsert'
      (NAME= SkyServer.dbo.sd_insert'),
     WEBMETHOD 'SdUpdate'
      (NAME= SkyServer.dbo.sd_update'),
     WEBMETHOD 'SdDelete'
      (NAME= SkyServer.dbo.sd_delete'),
     BATCHES = DISABLED,
   WSDL = DEFAULT,
   DATABASE = ' SkyServer ',
   NAMESPACE = 'http://SkyServer/SkyServer' )
```

Figure 2. Mapping of the SD-SQL Server commands to Web Services

4. PERFORMANCE ANALYSIS

To validate the SD-SQL Server architecture, we evaluated its scalability and efficiency over some *SkyServer* DB data [2]. Our hardware consisted of 1.8 GHz P4 PCs with either 785 MB or 1 GB of RAM, linked by a 1 Gbs Ethernet. We used the SQL Profiler to take measurements.

The query measures included the overhead of the image checking alone, of image adjustment and of image binding for various queries, [12, 13]. Here, we discuss this query:

(Q) *sd_select 'COUNT (*) FROM PhotoObj'*

We have measured (Q) on our *PhotoObj* scalable table as it grows under inserts. It had successively 2, 3, 4 and 5 segments, generated each by a 2-split. The query counted at every segment. The segment capacity was 30K tuples. We aimed at the comparison of the response time for an SD-SQL Server user and for the one of SQL Server. We supposed that the latter (i) does not enter the manual repartitioning hassle, or, alternatively, (ii) enters it by 2-splitting manually any time the table gets new 30K tuples, i.e., at the same time when SD-SQL Server would trigger its split. Case (i) corresponds to the same comfort as that of an SD-SQL Server user. The obvious price to pay for an SQL Server user is the scalability, i.e., the worst deterioration of the response time for a growing table. In both cases (i) and (ii) we studied the SQL Server query corresponding to (Q) for a static table. For SD-SQL Server, we measured (Q) with and without the LSV option.

Figure 2 displays the result. The curve named "SQL Server Centr." shows the case (i), i.e., of the centralized *PhotoObj*. The curve "SQL Server Distr." reflects the manual reorganizing (ii). The curve shows the minimum that SD-SQL Server could reach, i.e., if it had zero overhead. The two other curves correspond to SD-SQL Server.

We can see that SD-SQL Server processing time is always quite close to that of (ii) by SQL Server. Our query-processing overhead appears only about 5%. We can also see that for the same comfort of use, i.e., with respect to case (i), SD-SQL Server without LZV speeds up the execution by almost 30 %, e.g., about 100 msec for the largest table measured [3]. With LZV the time decreases there to 220 msec. It improves thus by almost 50 %. This factor characterizes most of the other sizes as well. All these results prove the immediate utility of our system.

Notice further that in theory SD-SQL Server execution time could remain constant and close to that of a query to a single segment of about 30 K tuples. This is 93 ms in our case. The timing observed practice grows in contrast, already for the SQL Server. The result seems to indicate that the parallel processing of the aggregate functions by SQL Server has still room for improvement. This would further increase the superiority of SD-SQL Server for the same user's comfort.

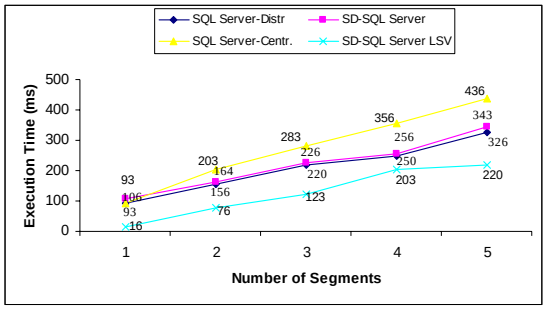

Figure 3. Query (Q) execution on SQL Server and SD-SQL Server

4.1 PERFORMANCE ANALYSIS OF EWS-SDSQL INTERFACE

We have measured the time of the *SdSelect* Web service corresponding to the scalable query (Q) where *PhotoObj* scalable table contained only one segment. We recall that we cannot address distributed segments in a partitioned view if we query it from a Web service.

The execution time of this Web service presents about three times the execution time of the query (Q) directly on SD-SQL Server whatever the scalable table size.

We have also measured the time of the Web service that corresponds to the query below:

(Q1) *sd_select 'TOP N * FROM PhotoObj'*

In the *TOP* clause, we vary *N* to have the values 50, 100, 1000…tuples.

If *N=50* tuples, the execution time of the Web service, that corresponds to (Q1) is about two times longer than the execution time of (Q1) directly on SD-SQL Server.

If *N=100* tuples, the execution time of the Web service is about four times longer than the execution time of (Q1) directly on SD-SQL Server.

If *N=1000* tuples, the execution time of the Web service is about forty times more important than the execution time of (Q1) directly on SD-SQL Server.

Each time we increase the number of tuples in the TOP clause, the overhead resulted from the call of the Web service is very noticeable.

Table 1 below shows the measurements pf the execution of (Q1) over the SD-SQL Server interface and the EWS-SDSQL interface when *N= 50, 100 and 1000* in the TOP clause of (Q1).

	on SD-SQL Server directly	**over EWS-SDSQL Interface**
N = 50	2.1648	5.278
N = 100	2.5076	9.664
N = 1000	2.3125	90

Table 1. Execution Time (sec) of (Q1) on SD-SQL Server and on EWS-SDSQL interface

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. A video demonstration of our system is available in our Web site [17, 18]. We believe that the proposed SD-SQL Server capabilities should become a standard feature of a modern DBMS.

The current design of our interface is geared towards a "proof of concept" prototype. It is naturally simpler than a full-scale system. We have extended the SD-SQL Server interface to be accessed upon a scalable Web services.

The performance measurements of the SD-SQL Server interface show that the overhead related to the additional processing of SD-SQL Server is negligible. However, the preliminary measurements related to the Web services interface to SD-SQL Server show that the overhead of the processing of Web services is very pronounced.

Our performance analysis should be expanded. The measurements we took for Web services upon SD-SQL Server remain preliminary. We should extend them and study the performances of EWS-SDSQL more in dept. If the resulted overhead remains important, we should perform our measurements method of Web services and find ways to improve them. Further work will also concern the application of Web services on scalable partitioned views. This should work on partitioned views with distributed scalable table segments.

Finally, while SD-SQL Server acts at present as an application of SQL Server, the scalable table management could alternatively become part of the SQL Server core code. Obviously, we could not do it, but the owner of this DBS can. Our design could apply almost as is to other DBSs, once they offer the updatable distributed partitioned (union-all) views.

6. 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

[3] Gray, J. The Cost of Messages. Proceeding of Principles Of Distributed Systems, Toronto, 1989.

[4] Guinepain, S. and Gruenwald, L. Research Issues in Automatic Database Clustering. ACM-SIGMOD, 2005.

[5] Lejeune, H. Technical Comparison of Oracle vs. SQL Server 2000: Focus on Performance, 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.).

[10] Litwin, W., Sahri, S., Schwarz, T. SD-SQL Server: Scalable Distributed Database System. CERIA Research Report 2005-12-13, December 2005.

[11] Litwin, W., Sahri, S., Schwarz, T. 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.

[12] Litwin, W., Sahri, S., Schwarz, T. Architecture and Interface of Scalable Distributed Database System SD-SQL Server. The Intl. Ass. of Science and Technology for Development Conf. on Databases and Applications, IASTED-DBA, Insbruck, 2006.

[13] Litwin, W., Sahri, S., Schwarz, T.: An Overview of a Scalable Distributed Database System SD-SQL Server. In: Flexible and Efficient Information Handling: 23d British National Conference on Databases, BNCOD 23, Belfast, Northern Ireland, UK, July 2006 Proceedings, Bell, D. and Hong, J. (Eds.), Lecture Notes in Computer Science 4942, Springer-Verlag, Berlin, Heidelberg, and New York, 2006, p. 16-35.

[14] Litwin, W., Sahri, S. & Schwarz, Th. *New Features for a Scalable Distributed Databases Management in SD-SQL Server*. The 3rd Biennial Conference on Innovative Data Systems Research, CIDR 2007, January 7-10, Asilomar.

[15] Loney, K & Bryla, B. Oracle Database10g DBA Handbook, Oracle Press, 2005.

[16] Sahri, S. SD-SQL Server : Conception de Bases de Données Distribuées et Scalables. Phd Thesis, June 2006.

[17] SD-SQL Server Video demo. http://ceria.dauphine.fr

[18] SD-SQL Server installation Readme. http://ceria.dauphine.fr (submitted to DBWorld).