# Overview 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, stored each at some SD-DBS server (node DB). 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 user or the application, manipulates a scalable table from a client (node

[1] Witold.Litwin@Dauphine.fr

[2] Soror.Sahri@Dauphine.fr

[3] tjschwarz@scu.edu

DB) that is not a server, or from a peer 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 a scalable table using its image or more generally a scalable view. A scalable view involves a scalable table, always and only through the reference to its image.

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 referring to comes in, and finds the image outdated. 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 SD-SQL Server nodes. An *SD-SQL Server node* is a linked SQL Server node declared, in addition, an SD-SQL server node. Actually, we generate the links with the *lazy schema validation* option, accelerating the query processing. The declaration of a linked node as an SD-SQL Server node results from an SD-SQL Server command or a dedicated SQL Server script for the 1st node of the collection. We qualify the latter of *primary* node. The primary node registers all other current SD-SQL nodes. One can add or remove those dynamically, using specific SD-SQL Server commands. The primary node registers the nodes on itself, in a specific SD-SQL Server database termed *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 *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. An SD-SQL Server command is typically an SQL Server stored procedure involving clauses of 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. An SD-SQL Server commands for queries are typically named upon their SQL “originals”, e.g., SD_SELECT upon SELECT.

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. Finally, the application may store at the client other SQL Server objects that it might need such as its own stored procedures.

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). 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. These commands are organized so that the concurrent processing of a split and of a scalable query to the scalable table being split always remains correct (serializable).

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.

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 has only one (primary) server, otherwise it chooses randomly in its list. A peer usually creates the primary segment in its NDB. Next, the client (or the peer) creates the *primary* client image of *T*, named *T* itself, in its NDB. The creation involves the input into SQL Server meta-tables and into **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* 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 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. The procedures involve multiple SQL commands on the segments and meta-tables, within some procedural logic. They correspond to the segment creation, splitting and dropping. As we just mentioned, the split operation in particular, makes the server to remotely create the segment(s) on other sites.

Finally, SD-SQL Server allows for the *node management* commands. These let to create/drop SD-SQL Server nodes, SDBs and NDBs. A node creation command installs one or more SD-SQL Server nodes. A node can be of type peer, client or server. The *peer* node (default) accepts any type of NDB: client, server or peer. The *client* node only accepts a client NDB, while the server node only accept server NDBs. The creation of an SDB creates its primary NDB and registers SDB at the primary node. The creation of an NDB requires the existence of its SDB from which the NDB inherits the name. Internally, it registers the NDB at the primary one of the SDB. Any drop operation undoes all the above. It preserves however the every secondary segment by migrating them elsewhere. This may require a dynamic creation of an NDB elsewhere. A server NDB manager can also dynamically create an NDB during a split in progress. It may do it when the scalable table it manipulates already has a segment at every NDB within the SDB, hence it cannot find the normal location for the new one(s).

To illustrate the architecture, Fig 1 shows the NDBs of some SDB, on nodes *D*1…*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 *D*1 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 *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. 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). We discuss details of SD-SQL Server naming rules later on. Notice here only that the name provides the uniqueness with respect to different client (peer) NDBs in an SDB. These can have each a different scalable table named *T* for the local applications. Their segments named as discussed may share a server (peer) node without the name conflict.

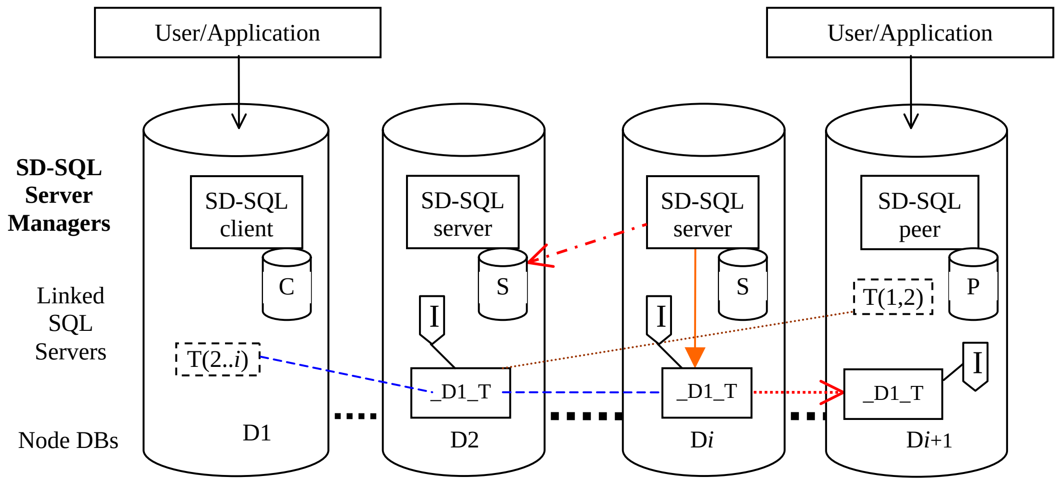

**Fig 1 SD-SQL Server Architecture**

## 3. Application Interface

To be developed in the full paper. See in the meantime [14].

## 4. Command Processing

To be developed in the full paper. See in the meantime [14].

## 5. Performance Analysis

To be developed in the full paper. See in the meantime [14].

## 6. 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. In SQL Server 2000 that we use, the partitioning is manual. SQL Server 2005 has a new capability that is the dynamic creation of a new segment, subject to the application command. For instance, the user may require that the table get a new segment every 1st of the month [15]. This is not the scalable table in our sense. It does not have the capability of splitting the overflowing segments. Splitting remains manual under SQL Server 2005. Generally, how one may create the scalable tables at the discussed systems remains an open research problem.

## 7. Conclusion

We have put into practice the SD-SQL Server, the “proof of concept” prototype of a new type of a DBMS, managing the scalable distributed tables. Through the scalable distributed partitioning, with respect to the current capabilities of an SQL Server database, as well as of the other known DBMSs, our system provides a much larger table, or a faster response time of a complex query, or both.

The SD-SQL Server commands let the user/application to easily take advantage of the new capabilities of our system. Our work in progress concerns remaining implementation and tuning issues. We also work on deeper performance analysis, using more machines and larger tables. We will measure more SkyServer benchmark queries, as well as plan to apply some well-known general benchmarks. Finally, since our prototype uses the current public SQL Server version that is the SQL Server 2000, we also plan to migrate to SQL Server 2005 when Microsoft fully releases it. This will somehow change our system design, especially the scalable index management.

## 8. Acknowledgments

We thank J. Gray (Microsoft BARC) for the SkyServer database and helpful comments. We are grateful to G. Graefe (Microsoft) for information on SQL Server linked servers’ capabilities. We also thank N. Ben-El-Kadi for information on various SQL Server capabilities. The initial support for this work came partly from research grant of Microsoft Research.

## 9. 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
14. Soror Sahri, Witold Litwin, Architecture and Interface of Scalable Distributed Database SD-SQL Server System, CERIA Research Report, November 2005.
15. Kimberly L. Tripp, Tables et Index Partitionnés dans SQL Server 2005. January, 2005.