Proceedings of the Second International Workshop on Cooperative Internet Computing (CIC 2002)

August 18-19, 2002
Kowloon Shangri-La Hotel
Hong Kong

Edited by
Alvin T.S. Chan
Stephen C.F. Chan
Hong Va Leong
Vincent T.Y. Ng

裘槎基金會
The Croucher Foundation

acm
Hong Kong

PROCEEDINGS

The Second International Workshop on Cooperative Internet Computing

August 18-19, 2002

Kowloon Shangri-La Hotel
Hong Kong

CIC 2002

Organized by:

Department of Computing
The Hong Kong Polytechnic University

Sponsored by:

Department of Computing, The Hong Kong Polytechnic University
The Croucher Foundation
ACM, Hong Kong Chapter
IEEE Hong Kong Chapter

Edited by:
Alvin T.S. Chan
Stephen C.F. Chan
Hong Va Leong
Vincent T.Y. Ng

An Architecture for a Scalable Distributed DBS: Application to SQL Server 2000

(Extended Abstract)

W. Litwin, T. Risch, Th. Schwarz
U. Paris 9, U. Uppsala, U. Santa Clara
Witold.Litwin@dauphine.fr, ; Tore.Risch@it.uu.se ; tjschwarz@scu.edu

Abstract

A table to be managed by a Scalable Distributed Database System (SD-DBS) should be able to horizontally scale over several DBMSs. The partitioning and especially its dynamic evolution should remain transparent to the application. We propose an architecture for an SD-DBS. Our architecture aims at SQL Server, but generalizes to other relational DBSs.

1 Introduction

A *parallel* DBS allows a table to be partitioned over a cluster of server-DBSs distributed over a network. Most of major DBSs are parallel, e.g. SQL Server, Oracle, DB2 and Sybase. The partitioning can be hash or range based. If a table scales, as more and more often these days, potentially to Petabytes, e.g., for the Skyserver [G02], the partitioning must be able to incrementally include a new node at a time. In present commercial parallel DBSs, the DBA has to manually redefine the partitions and typically run a utility to redistribute the data. In this sense, current parallel DBSs do not provide *scalable* data partitioning. In other common terms, their tables are not *horizontally* scalable or, in short, are not *scalable*.

On the other hand, the technology of Scalable Distributed Data Structures (SDDSs) provides for the scalable partitioning, e.g., in the SDDS-2000 system [C01]. In particular, LH* schemes provide for scalable distributed hash partitioned files of records identified by a primary key, and RP* schemes provide for scalable distributed range partitioned files. It becomes tempting to reuse this technology to provide for scalable tables as well.

We call *Scalable Distributed* Database System (SD-DBS) a relational DBS whose tables may be scalable according to SDDS principles. No SD-DBS exists at present. Our goal here is to define an architecture putting this concept into practice.

Accordingly, we consider that a table *T* of an SD-DBS resides on some SD-DBS *servers*, starting at one of them. When it grows beyond some predefined size *b* of *T*, e.g. *b* tuples, the SD-DBS mechanism *splits T*.

The split partitions *T* at a node so to create a new table with the same schema at another node and migrate there about half of *T* tuples from the splitting one. The splitting process is managed according to some SDDS schema. It keeps *T* size at any node as at most *b* for an RP* schema, or, typically, close to *b* for an LH* schema.

Likewise, the application accesses an SD-DBS through its *client* node. According to SDDS principles, each client has some private image of the actual partitioning. It uses the image to process the queries. With SDDSs client images are *not* refreshed synchronously by splits. This could be too cumbersome, e.g., because of moving or partly-available clients. Instead, the client is always allowed to make *addressing errors*. Such an error may trigger some forwarding among the servers to correctly locate the searched tuples. In particular, it may trigger an *image adjustment message* (IAM) back to the client. Data in an IAM improve the image, avoiding at least to repeat the same error.

It is obvious that building a full scale SD-DBS is a potentially very challenging task. We focus therefore on the maximal reuse of the capabilities of an existing parallel DBS and on the minimal programming effort of the additional functions. We aim for the largest possible use of standard SQL manipulations. In particular, we do not intend to write an SD specific query optimiser from scratch as this might be a huge task.

Below, we base specifically on new functions that are available in SQL Server 2000 for the management of *so-called* federated DBs [MS01]. The essential idea is that an application sees a *federated* view that is defined as a union view on some *client* SQL Server 2000 of union-compatible tables at different *server* nodes. These tables are elements of a range partitioning over some *partitioning key* attribute, possibly a composite attribute. The partitioning range of each server is defined by a check constraint at the server. The processor of the union view accesses these constrains when a query or an update is issued. This allows processing the query more efficiently, directing it to only the nodes where data may reside. In particular the federated view supports inserts. The use of check constraints allows the insert to be directed to the single node where it should reside.

We call our system *SD-SQL Server*. We attempt to use the new SQL Server capabilities in the following way. We cover the SQL Server by an additional layer, we call the *SDDS* layer. The application interfaces that layer, i.e, it submits there any queries. The *SDDS* layer manages the SDDS-like table splits at the servers side, and the federated union-views representing the SDDS images at the client side. At each server side, every split alters the check constraints so they reflect the current partitioning. It also puts some input into some meta-tables. Those contain the actual image of the partitioning for the SDDS layer. The splits can create also *parity* tuples. Those provide the *k*-availability, i.e., the tolerance of unavailability of up to *k* table segments, e.g., as for the LH*$_{RS}$ schema.

On the client side, the federated union views constitute the private images. Each time an application query issues a query towards the view, the *SDDS* layer checks whether the image is adequate. The image checking is done by queries to SD-SQL meta-tables at the client and some servers. The query checks at least the adequacy between the number of tables known to the image and the actual number in the meta-tables at the servers. If the test does not match, the application query is either not issued towards the SQL Server, or aborted, if it already started, as it may happen. The SDDS layer updates then the view definition from the meta-tables. The query is (re)issued to the SQL Server by the SDDS layer, i.e., transparently for the application. It is hence optimised by the SQL Server query optimiser as usual, or re-optimised, and executed correctly.

The image adjustment strategy of SD-SQL Server is based on the checking of image correctness at the client. It is thus a departure from the current principle of an SDDS where this check takes place at a server sending an asynchronous IAM back to the SDDS client. The crucial benefit is that the query optimisation remains entirely with SQL-Server, as we wished. The cost is additional messaging. We show that its significance for the user query performance should nevertheless be rather negligible in practice.

Below, we outline the SD-SQL Server architecture. We begin with the gross architecture at servers side. We focus on the table scale-up through the splits in particular. We also outline the high-availability management. Afterwards, we outline the client side architecture. We end up with the conclusion showing how our proposals match our goal for an SD-SQL Server.

2 Gross architecture

The SD-SQL Server manages a *federation* of SQL Servers. The word federation means here its sense in the SQL Server 2000 literature. Each of the SQL Servers carries one or more databases.

Fig. 1 presents the gross architecture of SD-SQL Servers. At each node of the federation there is an additional component called *SD-DBS manager*, or *manager* in short. The manager can play the role of SDDS client, termed here *SD-DBS client*, or *client* in short. Alternatively, it can be an *SD-DBS server*. Finally, it can play both roles. As a client, the manager receives the application queries to application data in the databases of the federation. These are queries to SQL Server. If the manager acts as a server, it gets messages from the client together with the applications queries that it passes to its SQL Server for execution. SQL Servers communicate among them as usual for the federation. SD-DBS servers are from them applications like any others.

We distinguish for SD-SQL Server between tables that can be partitioned and those that should not. The latter may, e.g., have "strings attached", such as indexes, triggers stored procedures, etc. They are not handled at present by SD-SQL Server since they present some degree of complexity that requires further analysis. Notice that SQL Server 2000 does not manage indexes or triggers etc. over a partitioned table neither. One may only create those objects manually for each segment. Notice also that the automatic creation of indexes for the new segment by a split, upon those of the splitting segment, does not present conceptual difficulties. In contrast, the replication by the split of triggers or stored procedures obviously needs caution.

Let D_i ; $i = 1,2\ldots$; be the databases under the SD-DBS servers. We basically consider one such DB per server, although what follows applies otherwise as well. Any scalable table T is created with some schema at some D_i. When T scales, the splits partition it over several D_i's. Each element of T then, called *segment*, is a full fledged table with the same attributes as those with which T was created.

A segment in D_i is named locally T and globally $D_i.T$. Each segment has a maximal size s, representing bucket size b for an SDDS [C01]. The value of s for each segment, is measured below as the number of tuples. The D_i server defines s when its T segment is created. The client may specify the maximal s for all the T segments.

Among D_i's some *segment* DBs serve simply as common storage pool for segments. In other words, segment DBs do not bear semantics. A segment can basically enter any segment DB. The manager may reuse for the choice an algorithm for physical allocation of buckets among SDDS servers. It may choose an existing D_i, or may create new D_i at some SD-DBS server.

In addition, the application can create *application* DBs. These are usual SQL-Server databases that typically bear some semantics. Application DBs should rather be on the clients, but could be at servers as well. They are intended for the non-scalable tables of the application and the federated views of the scalable ones. Nevertheless, an application can create these tables and views in a segment DB as well.

Fig. 1 illustrates also these principles. There is a federated view presenting some table, let it be *T,* in the application database at the client that is also the server of segment database D_1. The actual table T was initially

created at some *Di*, perhaps at D_2. It substantially scaled up thereafter, being now range partitioned into almost thousand segments $D_1.T$, $D_2.T \ldots D_{999}.T$. Perhaps, it grew up so much because of the large stream of data from a virtual telescope [G02]. The segments have different sizes, perhaps because of differences to the storage space available at the segment DBs. The dotted arrow line of the last segment symbolizes that it is just being created, and added to the view transparently for the application. At any time, the view shows to the application the tuples of all and only the segments it currently maps to. The application issues the queries to view *T* as if these tuples were in real table *T* in the application database[1].

3 Server side

3.1 SDDS-layer Meta-tables

Let $D_{i,T}$ be the segment database where one creates the initial segment of some table *T*. At every D_i there are three meta-tables at SD-SQL Server disposal. These are called at present SD-RP (DB-S, Table), SD-S (Table, S-max), and SD-C (DB-T, Table, S-size). Table SD-RP describes at each $D_{i,T}$ the actual partitioning of each table *T*. Tuple (D_i, *T*) is created in $D_{i,T}$.SD-RP by SD-Manager anytime one creates a segment of *T* at D_i. This include the initial segment, i.e., at the creation of *T* itself. Likewise, an optional tuple (*T*, *s-max*) in table SD-S fixes the maximal segment size for all the segments of *T*, if the application provides such a limit, as discussed later on. Finally, for each segment of some *T* at database D_i table D_i.SD-C points towards $D_{i,T}$. Tuple ($D_{i,T}$, *T,s*) is created in D_i.SD-C anytime one creates a segment of *T* at D_i. The size *s* is either the one found for *T* in its tuple in its SD-S table, or the D_i server defines it according to its local storage policy.

3.2 Scalable Table Management

3.2.1 Table creation

An application requests the creation of a scalable table, let it be *T*, from SD-SQL Server. It uses the usual SQL **Create Table** query with the following additional clause. The clause concerns the size of *T* segments :

Segment : size Any | *s-max*.

Size **Any** means that each server chooses the size of its segment. The choice of *s-max* fixes the limit on the size *s* of each segment of *T*. This choice may be useful if DBA wishes to make the *T* scan time about fixed. A server can nevertheless choose $s < s\text{-}max$.

To create table *T*, SD-DBS manager issues two queries to SQL Server layer. One **Create Table** query creates the 1st segment of *T*. It is the only one existing for *T*, as long as it does not overflow. Its scheme is that defined in the original statement. The allocation to each D_i is determined by the manager and the servers. The manager may reuse an algorithm for physical allocation of buckets in an SDDS to choose an existing D_i, or may create D_i at some server.

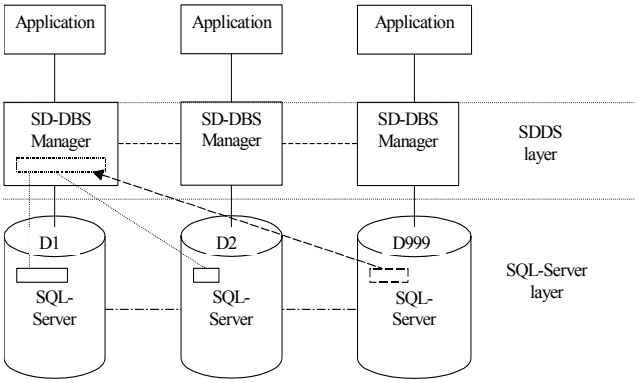

Fig. 1 Gross Architecture of SD-SQL Server

The other query is **Create View** query that creates the view *T* of table *T*. View *T* is created at *D*.

Finally, if *T* was created at database $D_{i,T}$, the manager creates at $D_{i,T}$ the trigger monitoring *T* size as discussed next. It also inserts the tuples describing *T* respectively into tables SD-RP, and SD-S and SD-C at $D_{i,T}$.

3.2.2 Table scale-up

SD-DBS manager at each D_i tests *s* anytime it gets (from SD-DBS client) the application query that could change there the number of tuples, i.e., an Insert or Delete query. To test *s*, SD-SQL Server performs simply the SQL query:

Select Count(*) From T;

Anytime the number of tuples in $D_i.T$ overflows *s*, SD-Manager at that server triggers the split. For this purpose, it first determines D_j that could handle the new segment. The algorithm may reuse those for physical allocation of buckets in an SDDS to choose an existing D_j, or may create D_j at some server. Then, let it be $s' = s / 2$ and let *C* denote the partitioning key attribute. One executes at D_i the SQL query:

Select into $D_j.T$ Top (*s'*) from *T* order by *C*;

This query creates at D_j table *T* with the same attributes as $D_i.T$ and copies there top *s'* tuples (notice it might not recreate the primary key definition). Once this process is done (by SQL Server 2000), SD manager at D_i performs the following queries. These determine the new middle key, hence the new range of $D_i.T$ and delete from $D_i.T$ the tuples that moved:

Delete * from *T* where *C* >= (Select min (*C*) from (Select Top (*s'*) from *T* order by *C*));

[1] Whether SQL Server 2000 is able at present to really manage an actual federated view of a thousand segments remains an open question.

Next, the manager at D_j executes the SQL Server 2000 Alter Table query that alters the check constraint of T at D_j. We avoid here to give the syntax of it. Likewise, the check constraint with the appropriate range (the new middle key and the former maximal key at D_i) is added to $D_j.T$ schema. Finally, the manager inserts tuple ($D_{i,T}$, T) into D_j.SD-C and tuple (D_j, T) into $D_{i,T}$.SD-RP. The former will be reused by the future splits at D_j. The latter will be used to check by the clients for the actual partitioning of T.

3.3 High-availability

High-availability allows an application to use an entire partitioned table despite unavailability of some of its segments. SD-SQL Server may provide *scalable k*-availability which means that the simultaneous unavailability of up to $k \geq 1$ segments is tolerated where k scales with T. This is done by reusing the technique developed for LH*$_{RS}$ SDDS, [LS00], [S02]. Namely, for each T there are k additional *parity* tables each located at some DB other than those supporting any of T segments. When a tuple is inserted into T, or updated or deleted, a trigger and a stored procedure compute the so-called δ-tuple and send it to k parity tables. There, tuples from up to $m >> 1$ distinct segments of T constitute logically the *tuple group*. Each tuple group gets a value called *rank* and one *parity* tuple identified in the parity table by its rank. Ranks are attributed by auto-numbering of parity tuples. Each parity tuple contains the *parity* attribute whose content is calculated by a stored procedure at the parity DB as for LH*$_{RS}$. One uses these tuples to recover the tuples the query needed, if segments searched by the query are unavailable.

4 Client side

4.1 Client Image Structure

The SD-DBS client manages the image of each partitioned table. The image presents table T as if it were entirely in the client's DB. This is done through the SQL Server union view of T segments known to the client:

Create View T as
select * from $D_{i,T}.T$ union all select $D_{i,2}.T$… select select * from $D_{i,n'}.T$;

Here, $D_{i,T}$ is the database where T was created and $D_{i,n}$ is the database where n'-th segment of T resides. Notice the use of **union all** despite lack of duplicates which is due to possible presence of OLE attributes, not supported by **union**. The view is stored in the SQL Server and used by the application queries. It is however created by the SD-DBS client upon the application query. Afterwards, it is altered dynamically and transparently for the application.
To manage images SD-DBS client has a meta-table termed C-Image with the schema:

C-Image (DB-T, Table, Size).

For every partitioned table T that the client manages, there is a tuple ($D_{i,T}$, T, n') in C-Image. Initially, when T is created, $n' := 1$.

4.2 Image Adjustment

Data in C-Image allow the client to check whether its image of T is the actual one. If not, query optimisation by SQL Server at the client could silently lead to erroneous results for some queries involving T. *The necessary condition for the correct image is that the number of T segments n' known to the client and hence used in the T view schema, is the actual one.* The actual number of T segments is that of the number of tuples created by T splits in table SD-RP of T. It thus results simply from the query:

(Q1) Select count (*) into @n
From $D_{i,T}$.SD- RP Where Table = 'T';

Here, variable n (denoted as @n for Transac SQL) will contain the actual number of T segments. If $n' = n$, then the application query can execute safely. Otherwise, n' should be adjusted. The client has to find then the location of the new segments. It seems nevertheless more efficient at present to rather locate all the T segments. The client can do it using the following query:

(Q2) Select DB-F into Temp-T
From $D_{i,T}$.SD-RP Where Table = 'T' ;

The client then uses the tuples in the temporary table Temp-T to trivially alter its T view definition so it includes all the n existing segments.

4.3 Query Processing

The client checks the image correctness whenever the application submits a query. A query, let it be A, may address several tables, some partitioned others not. The client first parses therefore all the FROM clauses for partitioned table names[2]. It then prepares all the (Q1) queries and executes them as dynamic SQL statements. The negative result should be infrequent. The application query, should be also, typically; more complex than (Q1). The client issues therefore all (Q1) queries and A *in parallel*. Typically, (Q1) results should come back before those of A. If not, the client waits with A commitment. If any image reveals incomplete, the client aborts A. It then adjusts the image and restarts A. This is done transparently to the client.

The testing of the image correctness within SD-SQL Server occurs in this way at the client. This is a departure from the current principles of an SDDS with the checks at servers. The latter would require checking the subqueries received by the server. One would need to add these SDDS specific capabilities to the current SQL Server query optimiser. The former strategy allows the query

[2] The present scheme does not permit queries to views over the partitioned tables.

optimisation to perform entirely as at present. The SQL Server does not need any modifications, which is a crucial benefit.

The SD-SQL Server image adjustment strategy checks the image at the client. The basic strategy for an SDDS, is in contrast to check the image at the server. The price for the SD-SQL Server strategy is the additional messaging. The basic strategy triggers indeed the additional messages among the servers and to the client only when an incorrect image was detected. The former one implies basically at least two additional messages per query. However, these messages correspond to a typically simple query compared to that of the user. Next, assuming several servers and about uniform distribution of SD-RP tables among them, no server should constitute a hot-spot delaying the replies to the clients. Hence, these replies should typically come fast enough to avoid delaying the user query. Also, they do not block the start of the processing of the user query and should rarely lead to a restart because of an incorrect image. They should therefore have a rather negligible incidence at the user query performance.

5 Conclusion

SD-SQL Server attempts to put into practice the scalable distributed database partitioning. The SDDS layer reuses the SDDS technology, as present in SDDS-2000 prototype. The SQL Server layer applies new capabilities for federated multidatabase management of SQL Server 2000. Both layers uses also standard SQL capabilities coupled with a few meta-tables to perform crucial SDDS operations that are splitting and image adjustment.

As the result, the outlined architecture appears attractive and, hopefully, simple to put into practice. It paves thus the way towards experimental confirmation. In particular, the scalability of SD-SQL Server appears limited only by the size of the federation that SQL Server capabilities allow to manage in practice. It is a crucial advantage of SD-SQL Server gross architecture that it also gracefully incorporates future improvements to these capabilities. This includes especially the progress in the parallel query optimisation, new capabilities for federated views, e.g. the referential integrity constraints on such views, triggers at them, or hash partitioned views.

The exposed principles of SD-SQL Server functioning are just the basic scheme. Variants are easy to see that provide additional capabilities or can optimise performance. For instance, the split operation can easily recreate local indexes following their schemes at the splitting segment, as well as validity constraints on attributes etc. Likewise, an image checking query may easily concern several tables sharing an SD-RP table at once.

SQL Server uses the federated union views for the range partitioning with check constrains defining the ranges at the servers available for the query optimiser at the client. These are the key properties to the SD-SQL Server architectural simplicity. Parallel DB2 and Oracle uses a different approach through their clauses “Partitioning Key” of Create Table statement. DB2 manages furthermore only hash partitioning, while Oracle allows for both types. This approach has potential advantages, e.g., allowing for a global index over a partitioned table, impossible at present for SQL Server. Our principles should generalize to these DBSs as well, probably, however, at the expense of a more extensive implementation effort.

Acknowledgments

We thank Jim Gray for fruitful discussions during WDAS-2002 workshop, and for 100 GB of SQL Server SkyServer data to experiment with. This work was partly supported by the research grants from Microsoft Research, the European Commission project ICONS project no. IST-2001-32429, and by Vinnova project no. 21297-1.

References

[C01] CERIA: SDDS-2000 prototype and related papers.

[G02] Gray J. & al. Data Mining the SDSS SkyServer Database. To appear in Proceedings of 4-th Workshop on Distributed Data & Structures (WDAS-2002), Carleton Scientific (Publ.), 2002.

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

[LNS96] Litwin, W., Neimat, M-A., Schneider, D. RP* : A Family of Order-Preserving Scalable Distributed Data Structures. VLDB-94, Chile. With Neimat, M-A., Schneider, D.

[LS00] Litwin, W., J.E. Schwarz, T. LH*$_{RS}$: A High-Availability Scalable Distributed Data Structure using Reed Solomon Codes
ACM-SIGMOD-2000 Intl. Conf. On Management of Data. With Thomas J.E. Schwarz, S.J.

[R02] Risch T., Koparanova M., Thide B. High performance GRID database manager for scientific data. To appear in full in Proceedings of 4-th Workshop on Distributed Data & Structures (WDAS-2002), Carleton Scientific (Publ.), 2002.

[S02] Schwarz T. Generalized Reed Solomon code for erasure correction. To appear in full in Proceedings of 4-th Workshop on Distributed Data & Structures (WDAS-2002), Carleton Scientific (Publ.), 2002.

[MS01] Federated SQL Server 2000 Servers.