Ceria Res. Rep. 2005-12-13, December 2005

SD-SQL Server: Scalable Distributed Database System

Witold Litwin, Soror Sahri, Thomas Schwarz

Witold.Litwin@Dauphine.fr, Soror.Sahri@Dauphine.fr, tjschwarz@scu.edu

CERIA, Paris-Dauphine University
75016 Paris, France

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-partionns a table when an insert overflows its existing segments. This triggers the split resorbing the overflow by migrating tuples to new 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, its user/application interface, our implementation and its performance analysis. We show that the overhead of our scalable table management should be typically negligible.

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],[10]. The database administrator (DBA) in need to spread these tables over new nodes has to manually rede 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. The Indexed Sequential Access Method (ISAM) was in use for ordered (range partitioned) files. Likewise, only static hash access methods were known. Both approaches required file reorganization whenever the file grew too large. B-trees and extensible (linear, dynamic) hash methods were invented to avoid this need for file reorganization. Instead of reorganizing a complete file, these methods deal with file growth by incremental splitting one or a few buckets (pages, leaves, segments …) at certain inserts. These dynamic methods were successful enough to make ISAM and centralized static hashing obsolete.

Efficient management of distributed data adds specific needs. Scalable Distributed Data Structures (SDDSs) address these needs for files, [5][6]. SDDS can use hashing, range-partitioning or *k-d* trees to distribute its data in buckets spread over the nodes of a multicomputer. These nodes can form a peer-to-peer (P2P) or grid network. An SDDS grows to more buckets by splitting the overflowing ones. The splits are triggered by the (overflowing) inserts.

In [7], the analogous concept of a Scalable Distributed DBS (SD-DBS) appeared for databases. The SD-DBS architecture supports *scalable (distributed relational) tables*. As an SDDS, a scalable table accommodates its growth by splitting overflowing segments, stored each at some SD-DBS storage node DB. Like records in an SDDS, the tuples of a scalable table are assigned to segments through (scalable distributed) hashing, range-partitioning, a *k-d* tree.... The applications access a scalable table only through the specific views, called *images*. The images hide the partitioning. They are basically the updatable distributed union-all views. The applications manipulate the scalable tables by querying the images or (scalable) views of the images.

For various practical reasons, like in general in an SDDS, the splits in an SD-DBS do not modify the images. The image adjustment to the evolving partitioning is supposed lazy. It occurs only when a query

refers to an outdated image. Through all these properties, scalable tables avoid the global database reorganization, similarly to B-trees or extensible hash files with respect to the earlier ISAM and static hash file technology.

To prove the feasibility of an SD-DBS, we have built a prototype called SD-SQL Server. The system generalizes the basic SQL Server capabilities to scalable tables. We have chosen SQL Server since, to the best of our knowledge, it was the only DBMS proposing the updatable distributed union-all partitioned views. SD-SQL Server 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 the management of the SD-SQL Server client image or SD-SQL Server client node.

Below we present the architecture of our prototype and its application command interface in its 2005 version. This architecture contains more features than the reference architecture in [7]. We discuss the syntax and semantics of the commands. Numerous examples illustrate their actual use. We hope to convince the reader that the use of the scalable tables is in practice as simple as that of static tables, despite some limitation of our current interface. We view our current prototype as a proof-of-concept system.

The scalable table processing must create some overhead with respect to the processing of the static tables. We discuss our internal design of the prototype, aiming at the minimization of this overhead. This design expands that in [13] and [8]. We aim now especially also on the efficiency and the serializability of the concurrent processing, in presence of splits and of image adjustment operations. . We present the performance analysis proving that we can neglect the SD-SQL Server overhead in practice. We benchmark our processing using scalable tables loaded with experimental data from the well-known SkyServer database, [2].

The present capabilities of SQL Server apparently let it link up to 250 nodes. A scalable table may reach that many segments accordingly. This size should suffice for Petabytes. To the best of our knowledge, SD-SQL Server is the first system with the proposed capabilities. It should pave the way towards the use of the scalable tables as the standard technology.

Section 2 presents the SD-SQL Server architecture. Section 3 discusses the command interface. Section 4 discusses the processing. Section 5 follows with the related work. Section 6 presents the performance analysis. Section 7 concludes and discusses the future work. The glossary of our teminology follows.

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 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. 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 *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). 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.

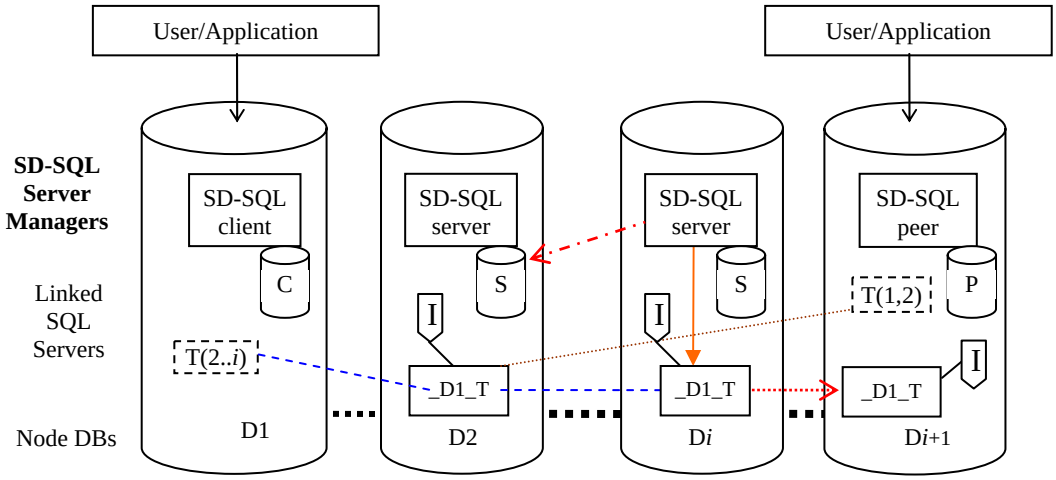


Fig 1 SD-SQL Server Architecture

3. Application Interface

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. However, the SQL Server *CREATE VIEW* command applies to both scalable and static tables. It produces a scalable view whenever a scalable table is among the base tables invoked. The SQL Server *DROP VIEW* command acts similarly.

We now present the syntax and semantics of the SD-SQL Server commands. The commands for node, SDB, NDB and image management are SD-SQL Server specific. 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. Moreover, the scalable *INSERT* command over a scalable table lets for any insert accepted by SQL Server for a distributed partitioned view. This can be a new tuple insert, as well as a multi-tuple insert through a *SELECT* expression. The *UPDATE* and *DELETE* statement offer similar capabilities. In contrast, some of SQL Server specific SQL clauses are not supported at present by the scalable commands; for instance, the *CASE OF* clause.

We discuss the syntax and semantics of the SD-SQL Server scalable commands case by case, but in the normal execution conditions only. Thus if a command addresses a node, we only discuss the case of its normal behaviour. We discuss the internal processing of each command that might address some unusual execution conditions later in Section 4.

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, [14]. Our experimental performance analysis in Section5 uses the data from the latter. 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. The primary node can be created as peer or server, but not a client. After that, the node, and then any other SD-SQL Server node created subsequently, offers the following (scalable) node management commands to the administrator, user or application.

3.2.1 Node Creation

The following command executed at an existing SD-SQL Server node appends a new to the existing collection:

sd_create_node 'new_node[, node_type]

The *'new_node'* parameter is the name of the spare on which the new node should be created. The node executing the command initiates in particular the meta-data of the new one according to its type. This latter is shown in the *'node_type'* parameter. It can be *server*, *client* or *peer*. The default type is *server*. A script file creates the first ever (primary) SD-SQL Server node at a collection of linked SQL Servers.

Example 3-1

Consider that SQL Server linked node at our *Dell1*machine contains the primary node of the SD-SQL Server configuration to create. The administrator has created the primary node by the script. The commands below executed at *Dell1* next create further nodes:

sd_create_node 'Dell2'

sd_create_node 'Dell3, 'client'

sd_create_node 'Ceria1','peer'

The SD-SQL Server node at SQL Server node '*Dell2*' is a server node, ready to store segments.

3.2.2 Node Removal

The following command removes an SD-SQL Server (scalable) node from the current collection of its (scalable) nodes:

sd_drop_ node 'node_name'

The removed node is no more an SD-SQL Server node, but remains an SQL Server linked node. The removal of a client or a peer node removes all the NDBs on it. As we discuss later, the removal of those may move some meta-data as well as some data in these NDBs to other nodes,

The need for the node removal seems rare. A (more complex) command removing simultaneously several nodes does not appear useful.

Example 3-2. The command below returns the previously created scalable *Ceria1* node, to the state of an SQL Server linked node only.

sd_drop_ node 'Ceria1'

3.2.3 Node Alteration

The application can upgrade a client or server node into a peer. It may also downgrade the peer. The command for these operations is:

sd_alter_ node 'node_name', 'ADD/DROP client/server'

Example 3-3

The command below upgrades the capabilities of the client node *Dell3* created in ExampleExample 3-1 to those of a peer:

sd_alter_ node 'Dell3', 'ADD server'

From now on, the node accepts the segments. Next after the command:

sd_alter_ node 'Ceria1, 'DROP client'

Ceria1 node that was a peer, Example Example 3-1, becomes (only) a server node.

3.3 Scalable Database Management

3.3.1 Scalable Database Creation

An SD-SQL Server scalable database administrator (SDBA) creates an SDB using the command:

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

The SDB *'db_name'* has its initial location at the node *'node_name'*. Operationally, as we detail in Section3.2.1, it creates in particular at *'node_name'* node the primary NDB of the SDB, named *'db_name'* as well. The optional *'extent'* parameter should have the value $n > 1$. It allows creating simultaneously $n > 1$ NDBs, including the primary one. All bear the name *'db_name'*. The manager chooses the location of these (secondary) NDBs on SD-SQL Server nodes randomly. The optional *'type'* parameter indicates whether the primary NDB of the scalable database is a server or peer. By default, the primary NDB inherits the type of the node *'node_name'* (that thus has to be peer or server as well). We recall that one cannot create a peer NDB at a server or a client node, or a client NDB at a server node, or vice versa. SD-SQL Server creates any secondary NDB requested in the command as a server NDB.

Example 3-4

The command below creates our *SkyServer* SDB, with its primary *SkyServer* NDB at the *Dell1* server node. As we do not mention the type, *SkyServer* NDB is a server NDB.

sd_create_scalable_database 'SkyServer, 'Dell1'

The next command alternatively attempts to rather create *SkyServer1* as a client at *Dell1* node with two secondary NDBs on some nodes. As *Dell1* is a server and *SkyServer1* is declared as client, then *SkyServer1* will have the role of a server and a client, i.e. *SkyServer1* will be a peer NDB.

sd_create_scalable_database 'SkyServer1', 'Dell1', 'Client'

As for the command below, the creation of the scalable database *SkyServer2* will be cancelled because the the node *Dell5* is server and the node where it will be created is client. This generates a conflict of the notion of the management of a scalable database and don't respect the hierarchy of clients, servers and peers.

sd_create_scalable_database 'SkyServer2, 'Dell5', 'Client'

3.3.2 Scalable Database Alteration

The alteration of an SDB consists of creating an NDB or dropping an NDB.

3.3.2.1 Node Database Creation

The following command realizes this operation:

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

The NDB enters the SDB *'sdb_name'*. Its own name at its SD-SQL Server node, as well as an SQL Server database is *'sdb_name'* as well. It is created at the *'node_name'* node. The *'node_name'* parameter is optional. By default, SD-SQL Server creates the new node database on the node of the command. SD-SQL Server can also select a node from the metabase database (MDB) to create the new node database, as we will describe later. The *'type'* parameter, may limit the capabilities of the NDB, if created at a peer node. Otherwise, the NDB inherits the node type.

Example 3-5

We create a new NDB for *SkyServer* SDB at *Dell2*. *Dell2*.*SkyServer* NDB is a server NDB.

sd_create_node_database 'SkyServer', '*Dell2*'

The next command, supposed executed at a peer *Dell7* node creates a peer *SkyServer* NDB at that node:

sd_create_node_database 'SkyServer'

3.3.2.2 Node Database Removal

An application drops an NDB from an SDB using the command:

sd_drop_node_database 'db_name', 'node'

Some meta-data, as well as the segments of tables created by applications at peers or clients at other nodes move to another NDB as we discuss in Section 4.6.2. They obviously should be preserved.

Example 3-6. The command below drops *SkyServer* **NDB** at *Dell2* node.

sd_drop_node_database 'SkyServer', 'Dell2'

3.3.3 Scalable Database Removal

This operation uses the command:

sd_drop_scalable_database 'db_name'

As the result, it removes all the NDBs of the SDB with all their content.

Example 3-7. The command below drops the *SkyServer* SDB. In particular it removes all its previously created NDBs, e.g., at *Dell1*, *Dell2* and *Dell7* nodes.

sd_drop_scalable_database 'SkyServer'

3.4 Scalable table management

3.4.1 Table Creation

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

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

The parameter *'SQL: Create Table **T** clauses'* is the text of the SQL Server *CREATE TABLE* command following the command name itself. The table name may be global, i.e., prefixed by a node name and its node database, in principle, but only the local creation is supported at present. The SQL command invoked has to respect all the constraints that SQL Server imposes at a distributed partitioned view [9]. In particular, the scalable table has to have the partition key, among the key attributes, that supports the *check constraints* partitioning the partition key space. The partition key may be not (entire) primary key hence SD-SQL Server, like SQL Server, allows for tuples with the duplicate values of partition key.

The *Segment_size* parameter fixes the maximal size of a segment of *T*. The command creates a scalable table only. To create a static table, e.g., to avoid the above-mentioned restrictions on the scalable ones, one should use the SQL Server *CREATE TABLE* command. Finally, the 'Partition_Key' parameter indicates the partitioning attribute of *T*. It has to be among the key attributes, following SQL Server constraints. The parameter is optional and makes sense only for tables with composite keys . By default, SD-SQL Server 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.

Example 3-8

The user at peer *Dell1* creates the scalable table *PhotoObj*. The name comes from our benchmark for experimental performance analysis that is a fragment of SkyServer DB [2]. The user wishes that a segment contains at most 10000 tuples for the efficient distributed query processing. It applies the command:

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

Here, we have show only the beginnings of the actual SQL *CREATE TABLE* command clauses. The result is the *PhotoObj* scalable table with the O*bjid* primary key, and the segment size of 10000 tuples. It appears to the applications as being at *Dell1* node, i.e., its multi-database (global) SD-SQL Server name is *Dell1.PhotoObj*[1]. An application at another node, e.g., *Ceria2*, could also create a table named *PhotoObj*. This one would appear as at *Ceria2*, with the multi-database name of *Ceria2.PhotoObj*.

3.4.2 Table Alteration

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

Example 3-9

The user alters the scalable table *PhotoObj* initially created at *Dell1* by adding new column to it and updates its maximal size. The user may alter the table using the command:

sd_alter_table '*Dell1.PhotoObj ADD t INT, 10000*

The result of the scalable command is the scalable *PhotoObj* table with the *t* column added to all its segments, and its segment size updated to 10000 tuples. If the alteration is requested at the original node of the *PhotoObj* scalable table, i.e. *Dell1*, then the prefix indicating the multi-database name is not necessary.

3.4.3 Index Creation

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 a scalable distributed index, let it be *I*, on a column of a scalable table, let it be *T*, by executing the *sd_create_ index* command with the following syntax:

sd_ create_index ['SQL: Create Index I ON T clauses'

[1] For simplicity, we make abstraction here of the owner name part of the SQL Server global naming, as well as about the node DB name component for all examples of the Application Interface section.

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

Example 3-10. The command below, executed at the *Dell1* node, creates the *run_index* scalable index on the *run* column of the *PhotoObj* scalable table.

sd_create_ index 'run_index ON Photoobj (run)'

If this index creation is requested elsewhere, then one should use the name *Dell1.Photoobj*.

3.4.4 Index Removal

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

sd_drop_ index 'SQL: Drop Index T.I clauses'

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

Example 3-11. The command below, executed at the *Dell1*, removes *run_index*.

sd_drop_ index 'Photoobj.run_index'

3.4.5 Table Removal

To remove scalable table *T* the application uses the command:

sd_drop_ table ['SQL: DROP TABLE T clauses']

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

Example 3-12. Drop the scalable table *PhotoObj* created at peer *Dell1*:

sd_drop_ table 'Dell1.PhotoObj'

The prefix is not necessary for the command invoked at *Dell1*.

3.4.6 Image Creation

The application creates a secondary image at its client (peer) through the following command:

sd_create_image '[Primary_node]', 'Table'

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

Example 3-13. We create the (secondary) image of *PhotoObj* through the statement:

sd_create_image 'Peer1', 'PhotoObj'

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

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

3.4.7 Image Removal

The application removes a secondary image, using the command at its client (peer):

sd_drop_image 'image_name'

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

Example 3-14. The commands below delete a secondary image of our *PhotoObj* scalable table.

sd_drop_image ' Peer1.Photoobj' or **sd_drop_image** 'Peer1_Photoobj'

3.5 Scalable Queries

3.5.1 Search

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

sd_select 'SQL: Select clauses'[, Segment Size'][, 'Primary Key']

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

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. Any prefixed name is dealt with as a static table or view name.

Example 3-15

The following query brings all the data in *PhotoObj* scalable table:

sd_select '* FROM PhotoObj'

Next, the following command creates the scalable table *PhotoObj*, with the segment size of 500 tuples and *Objid* column as the primary key, from the original static *PhotoObj* table:

sd_select '* INTO PhotoObj FROM PhotoObj', 500, 'Objid'

3.5.2 Insert

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

sd_ insert 'SQL Insert clauses'

Here, *'SQL: Insert clauses'* input are the standard SQL Server *INSERT* command clauses. They may include the *SELECT* clause on scalable or static tables.

Example 3-16

The command below requested at *Dell1*, inserts a tuple into *PhotoObj* assigning the value '*225'* to its O*bjid* attribute:

sd_insert 'INTO PhotoObj (objid) values (225)'

The next command, issued at the same node, inserts perhaps many tuples into *PhotoObj* scalable table. It refreshes *PhotoObj* with the tuples in some source *PhotoObj-S* table in *Skyserver-S* DB at *Ceria5* node which are not yet in *PhotoObj*. The source table could be scalable or static only.

sd_insert 'INTO PhotoObj SELECT * FROM Ceria5.Skyserver-S.PhotoObj WHERE objid not exists in (SELECT objid FROM PhotoObj)'

3.5.3 Update

An application updates a scalable table by executing the command:

sd_update 'SQL: Update clauses'

The 'SQL *UPDATE* clauses' are the SQL Server UPDATE command clauses allowed for a distributed partitioned view.

Example 3-17

The command below executed at *Dell1* updates the *run* column of *PhotoObj*, for the tuple with *Objid = 225*:

sd_update 'PhotoObj SET run= 752 WHERE objid=2255031257923860'

The following command executed at some SD-SQL Server node changes the *run* column values to 752 for the first 10 tuples of *PhotoObj*.

sd_update 'dell1.SkyServer.PhotoObj SET run= 752 WHERE objid IN
(SELECT TOP 10 objid FROM PhotoObj')

3.5.4 Delete

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

sd_delete 'SQL: Delete clauses'

The input is the SQL Server *DELETE* command clauses. The command leaves unchanged the table partitioning, even if it deletes all the tuples (no segment merge).

Example 3-18. The command below requested at *Dell1*deletes the tuple identified by objid = 225 from *PhotoObj*:

sd_delete 'FROM PhotoObj WHERE objid=225'

4. Command Processing

We now present the processing of the commands. We start with the description of the SD-SQL Server naming rules. Next, we discuss the scalable table evolution. We follow up with the image and query processing. Finally we address the processing of the node management commands.

4.1 Naming rules

SD-SQL Server has its own system objects for the scalable table management. These are the node DBs, the meta-tables, the stored procedures, the table and index segments and the images. All the system objects are implemented as SQL Server objects. To avoid the name conflicts, especially between the SQL Server names created by an SD-SQL Server application, there are the following naming rules, partly illustrated at Fig 1 .

o Each NDB has a dedicated user account 'SD' for SD-SQL Server itself.

o The application name of a table, of a database, of a view or of a stored procedure, created at an SD-SQL Server node as public (*dbo*) objects, should not be the name of an SD-SQL Server command. These are SD-SQL server keywords, reserved for its commands (in addition to the same rule already enforced by SQL Server for its own SQL commands). The technical rationale is that SD-SQL Server commands are the public stored procedures under the same names. An SQL Server may call them from any user account.

o A scalable table T at a NDB is a public object, i.e., its SQL Server name is dbo.T. It is thus unique regardless of the user that has created it2. In other words, two different SQL Server users of a NDB cannot create each a scalable table with the proper name T. They can still do it for the static tables of course. Besides, two SD-SQL Server users at different nodes may each create a scalable table with the proper name T.

o A segment of scalable table created with proper name *T*, at SQL Server node *N*, bears for any SQL Server the table name *SD._N_T* within its SD-SQL Server node (its NDB more specifically, we recall). We recall that SD-SQL Server locates every segment of a scalable table at a different node.

[2] In the current version of the prototype.

o A primary image of a scalable table *T* bears the proper name *T*. Its global name within the node is *dbo.T*. This is also the proper name of the SQL Server distributed partitioned view implementing the primary view.

o Any secondary image of scalable table created by the application with proper name *T*, within the table names at client or peer node *N*, bears the global name at its node '*SD.N_T*'. This is also the proper name of the SQL Server distributed partitioned view implementing the secondary view.

We recall that since SD-SQL Server commands are the public stored procedures, SQL Server automatically prefixes all the proper names of SD-SQL public objects with *dbo.* in every NDB, to prevent the name conflict with any other owner within NDB. The rules avoid the name conflicts between the SD-SQL Server private application objects and SD-SQL system objects, as well as between the SD-SQL Server system objects themselves. Obviously, a private name created by an application cannot enter in conflict with an image or segment name at any SD-SQL Server node. Next, a primary image name created for the scalable table *T*, cannot conflict with the segment name at a peer (we recall that SQL Server does not allow a view and a table to share a proper name). Next, at any NDB, any two segments belonging to two different scalable tables sharing a proper name must have different segment names as well. These segments may thus share a server or peer node without creating a conflict for the underlying SQL Server table names. Next, a peer stores any segment and any image, primary or secondary, without the name conflict as well. Finally, the client cannot have the conflict between a primary image name and a secondary image name.

Example 4-1

Let *dell1*, *dell2* and *dell3* be respectively a client, server and peer nodes. Let *PhotoObj* be the proper name of a scalable table created at *dell1* in some NDB, let it be *SkyServer*, and of another one created by an application on *dell3*. *SkyServer*. Consider that *dell2* became the server of *dell1* for *PhotoObj*. Next that *dell1.SkyServer.dbo.PhotoObj* has split, creating a segment at *dell3*. Similarly, *dell3. SkyServer.dbo.PhotoObj* split creating a segment at *dell2*. Finally, *dell3* contains a secondary image of *dell1.SkyServer.Photoobj*. We have the following situation.

o Database *Dell1.SkyServer* contains primary image *dbo.Photobj*, constituting also as the distributed partitioned view *SkyServer.dbo.PhotoObj* for SQL Server at *dell1*.

o Database *dell2.SkyServer* contains the segment tables *SD._dell1_PhotoObj* and *SD._dell3_PhotoObj*,

o Database *dell3.SkyServer* contains the segment tables with the proper names (actually prefixed each with *SD.*): *_dell1_PhotoObj*, *_dell1_PhotoObj*. It also contains the distributed partitioned view actually named in *SkyServer SD.dell1_PhotoObj*.

4.2 Meta-tables

SD-SQL Server uses dedicated meta-tables constituting logically the catalogs named S, C and P at Fig 1 . These tables constitute internally SQL Server tables searched and updated using the stored procedures with SQL queries detailed in [8]. All the meta-tables are under the user name *SD*, i.e., are prefixed within their NDB with '*SD.*'.

The *S*-catalog exists at each server and contains the following tables.

o *SD.RP (SgmNd, CreatNd, Table).* This table at node *N* defines the scalable distributed partitioning of every table *Table* originating within its NDB, let it be *D*, at some server *CreatNd*, and having its primary segment located at *N*. Tuple (*SgmNd*, *CreatNd, Table*) enters *N.D.SD.RP* each time *Table* gets a new segment at some node *SgmNd*. For example, tuple (*Dell5, Dell1, PhotoObj)* in *Dell2.D.SD.RP* means that scalable table *PhotoObj* was created in *Dell1.D*, had its primary segment at *Dell2.D*, and later got a new segment *_Dell1_PhotoObj* in *Dell5.D*. We recall that a segment proper name starts with '_', being formed as in Fig 1 .

o *SD.Size (CreatNd, Table, Size).* This table fixes for each segment in some NDB at SQL Server node *N*, the maximal size (in tuples) allowed for the segment. For instance, tuple (*Dell1, PhotoObj, 1000*) in *Dell5.DB1.SD.Size* means that the maximal size of the *Dell5* segment of *PhotoObj* scalable table initially created in *Dell1.DB1* is 1000. We recall that at present, all the segment of a scalable table have the same size.

o SD.Primary (PrimNd, CreatNd, Table). A tuple means here that the primary segment of table T created at client or peer CreatNd is at node PrimNd. The tuple points consequently to SD.RP with the actual partitioning of T. A tuple enters N.SD.Primary when a node performs a table creation or split and the new segment lands at N. For example, tuple (Dell2, Dell1, PhotoObj) in SD.Primary at node Dell5 means that there is a segment _Dell1_PhotoObj resulting from the split of PhotoObj table, created at Dell1 and with the primary segment at Dell2.

The *C*-catalog has two tables:

o Table SD.Image (Name, Type, PrimNd,Size) registers all the local images. Tuple (I, T, P, S) means that, at the node, there is some image with the (proper) name I, primary if T = .true, of a table using P as the primary node that the client sees as having S segments. For example, tuple (PhotoObj, true, Dell2, 2) in Dell1.SD.C-Image means that there is a primary image dbo.PhotoObj at Dell1 whose table seems to contain two segments. SD-SQL Server explores this table during the scalable query processing.

o Table SD.Server (Node) provides the server (peer) node(s) at the client disposal for the location of the primary segment of a table to create. The table contains basically only one tuple. It may contain more, e.g., for the fault tolerance or load balancing.

Finally, the *P*-catalog, at a peer, is simply the union of C and S catalogs. Besides, each NDB has the table:

o SD.SDBNode (Node). This table points towards the primary NDB of the SDB. It could indicate more nodes, replicating the SDB metadata for the fault-tolerance or load balancing.

o *SD.MDBNode (Node).* This table points towards the primary node. It could indicate more nodes, replicating the MDB for the fault-tolerance or load balancing.

There are also meta-tables for the SD-SQL Server node management and SDB management. These are the tables:

o *SD.Nodes (Node, Type).* This table is in the MDB. Each tuple registers an SD-SQL Server node currently forming the SD-SQL configuration. We recall that every SD-SQL Server node is an SQL Server linked server declared SD-SQL Server node by the initial script or the *sd_create_node* command. The values of *Type* are '*peer*', '*server*' or '*client*'.

o *SD.SDB (SDB_Name, Node, NDBType).* This table is also in the MDB. Each tuple registers an SDB. For instance, tuple *(DB1, Dell5, Peer)* means that there is an SDB named *DB1*, with the primary NDB at *Dell5*, created by the command *sd_create_scalable_database 'DB1', 'Dell5', 'peer'*.

o *SD.NDB (Node, NDBType).* This meta-table is at each primary NDB. It registers all the NDBs currently composing the SDB. The *NDBType* indicates whether the NDB is a peer, server or client.

4.3 Scalable Table Management

4.3.1 Table Creation

We recall that a scalable table *T* is formally a tuple (*T*, ***S***), where *T* is the primary image of *T* and ***S*** are its segments. SD-SQL Server creates a scalable table as follows.

When the application of NDB *N.D* requests the creation of scalable table *T*, using *sd_create_table* ('T…,*b*), then SD-SQL Server creates the primary image of *T* at *N.D*. For all the application *T* is virtually in *N.D.dbo* as it would be the case of a public static table created by a local SQL Server user. If *N.D* is an SD-SQL client, then its manager chooses a server node, let it be at SQL Server node *N'*, among those in its *PrimServ* table. In both cases, the created segment does not have the *check constraint*. Also in both cases, there is the creation of the *AFTER* trigger, as for any segment creation. Next, the creator registers the segment with the meta-tables of node *N'*. It inserts tuple (*N,T,b*) to *N'.SD.Size* meta-table. It also insert tuple (*N',N,T*), describing *T* as having a single segment, into *N'.D.SD.RP* table. Furthermore, it inserts tuple (*N',N,T*) into *N'.D.Primary* meta-table, self-pointing thus *N'.D* as the primary node.

Next, the client creates the primary image. It first inserts tuple (*T,.true,N'*,1) to its own *Image* meta-table. The tuple defines the single segment image with the primary segment at node *N'*. Finally, the client requests from its SQL Server the creation of the distributed partitioned view as follows:

```
CREATE VIEW T AS SELECT * FROM N'._Di_T
```

Example 4-2. Consider the command to *SkyServer* SDB at its *Dell1* client node:

sd_create_table 'PhotoObj (objid BIGINT PRIMARY KEY…)', 150000

The manager at *Dell1* chooses the *SkyServer* **s**erver NDB at node *Dell2*, i.e., *Dell2.SkyServer* NDB, for the primary segment of *PhotoObj*. It performs the following actions (Fig 2):

o Creation of empty SQL Server table *_Dell1_PhotoObj* at *Dell2 SkyServer* by SQL Server.

o Creation of the following *AFTER* trigger on this table:

CREATE TRIGGER split_trigger_photoobj ON _ Dell1_PhotoObj
AFTER INSERT AS split_scalable_table 'PhotoObj', Dell2.SkyServer'

o Registration of *_Dell1_PhotoObj* by SD-SQL Server at *Dell2*, as the primary segment of *PhotoObj*. Tuples *(Dell2, Dell1, PhotoObj), (Dell2, Dell1, PhotoObj), (Dell1, PhotoObj, 150000)* are respectively inserted at *Dell2.SkyServer* NDB into the meta-tables *SD.Primary*, *SD.RP* and *SD.Size.*

o Creation by SQL Server at *Dell1* of the distributed partitioned view *PhotoObj*, addressing *Dell2.SkyServer.dbo._ Dell1_PhotoObj* as its single table.

o Registration of *hotoObj* distributed partitioned view as the primary view of *PhotoObj* scalable table. Tuple *(PhotoObj, .true, Dell2,* 1*)* is inserted into *SD.Image* meta-table.

Fig 2 shows the result of the scalable table *PhotoObj* creation.

4.3.2 Table Evolution

A scalable table *T* scales up by getting new segments, and scales down by dropping some. The dynamic *splitting* of overflowing segments performs the former. The *merge* of under-loaded segments may perform the latter. Both operations do not update the images. The existing ones become automatically outdated. They are eventually corrected later by the dedicated *image checking adjustment* operations. These occur during the query processing, as we already mentioned. The practical interest of merges appears seldom at best. We did not consider them for the current prototype (notice that the implementations of B-tree merges are very rare at best as well). We now describe the split and the image adjustment.

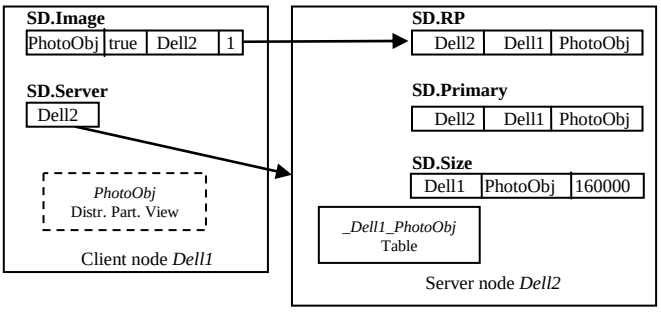


Fig 2 Creation of *PhotoObj* scalable table

4.3.2.1 Splitting

The splitting algorithm aims at several goals. We first enumerate these goals. Next we discuss the algorithm itself and show how it reaches the goals. The goals are as follows:

o The split removes the overflow from the splitting segment, leaving it at least half full. To remove the overflow, the split migrates some tuples into at least one new segment. Every new segment ends up half full. The overall result is clearly at least the typical good load factor of almost 70 %.

o The split processing aims at not delaying the commit of the insert triggering it. The split time may be typically expected much longer than that of the insert. The delay could lead to the timeout at the application level.

o The allocation of nodes to new segments of a scalable table aims at randomly balancing the node load among the clients (peers). On the other hand, the splitting algorithm allocates the same nodes to the

successive segments of different scalable tables of the same client. This policy aims at reducing the query execution time, as usually the queries tend to address the tables of the same client.

o The concurrent execution of the split and of the scalable queries is serializable. It means here specifically that a concurrent scalable query that addresses the tuples in an overflowing segment either manipulates them before the split migrates out any of them, or manipulates them only when the split is over.

We now show the algorithm, and how it achieves goals. The creation of a new segment for a scalable table *T* occurs when an insert overflows the capacity of one of its segments, declared in local *SD.Size* for *T*. At present, all the segments of a scalable table have the same capacity, defined in the *sd_create_table* command. The overflow may consist of arbitrarily many tuples, brought by a single *INSERT* command with the *SELECT* expression (unlike in a record-at-the-time operations, e.g., as in a B-tree). A single *INSERT* may further overflow several segments. More precisely, we may distinguish the cases of a (*single segment*) *tuple insert* split, of a *single segment bulk insert* split and, in the latter case, of a *multi-segment* (*insert*) split. The two latter cases correspond to the *INSERT* with the *SELECT* expression.

In every case, the *INSERT AFTER* triggers at every segment getting tuples tests for the overflow, [8]. The positive result leads to the split of the segment, according to the following *segment partitioning scheme*. The scheme adds $N \geq 1$ segments to the table, where it determines N as follows. Let T be the splitting scalable table with the segment capacity b. Let P be the (overflowing) set of all the tuples in one of, or the only, overflowing segment of T, ordered in ascending order by the key. Each server cuts its P, starting from the high-end, into successive portions $P_1 \ldots P_N$ consisting each of *INT (b/2)* tuples. It later sends each portion as a new half-full segment of T to a different server. The number N is the minimal leaving at most b remaining tuples in the splitting segment. We always have $N = 1$ for a tuple insert, and the usual even partitioning, as Fig 3 shows. A single segment bulk insert,

Fig 4,leads to $N \geq 1$ half-full new segments, while the splitting one ends up between half-full and full,. We have in both cases:

$$N = \lceil (\mathrm{Card}(P) - b) \ / \ \mathrm{INT}\ (b/2) \rceil$$

The same scheme applies to each splitting segments in the case of a multiple bucket insert, Fig 5.

The *AFTER* trigger only tests the overflow. It launches the actual splitting as an asynchronous job that we call *splitter*. If the trigger had to handle the split as well, the insert that lead to the overflow would not commit before the split ends. This could make the insert execution time unexpectedly much longer than otherwise. If the split job is dealt with asynchronously, the insert terminates in about the usual time. With some luck, the split does not then affect any concurrent update as it appears below.

The splitter gets the segment name as the input parameter. To create the new segment(s), with their respective portions, the splitter acts as follows. It starts as a distributed transaction at the serializable isolation level. SQL Server uses then the 2PL protocol for the concurrency management. The splitter first searches for *PrimNd* of the segment to split in *Primary* meta-table. If it finds the searched tuple, SQL Server puts it under a shared lock. The splitter requests then an exclusive lock on the tuple registering the splitting segment in *RP* of the splitting table that is in the NDB at *PrimNd* node. As we show later, it gets the lock if there is no (more) scalable queries or other commands in progress involving the segment. Otherwise it would encounter at least shared lock at the tuple, as we show later as well. SQL Server would then make the split waiting till the end of the concurrent operation. Very unlikely cases a deadlock may also result. As we show, the overall interaction suffices to provide the serializability of every command and of a split. If the splitter does not find the tuple in *Primary*, it ends up. As it will appear, it means that a delete of the table occurred in the meantime.

From now on, there cannot be a query in progress on the splitting segment; neither another splitter could lock it, as it should lock first the tuple in *RP*. The splitter safely verifies the segment size. An insert or deletion could change it in the meantime. If the segment does not overflow anymore, the splitter terminates. Next, it determines N as above. It finds b in the local *SD.Size* meta-table. Next, it attempts to find N an empty nodes where there is not yet any segment of T. It searches for such nodes through the query to *NDB* meta-table, requesting every node which is a server or peer and not yet in *RP* for T. Let $M \geq 0$ is the number of nodes found. If $M = N$, then the splitter allocates each new segment to a node. If $M > N$, then it randomly selects the nodes for the new segments. The selection is random, but driven, at

the base of the randomness generation, by the table creation NDB name. Any two tables created by the same client node share the same primary NDB, have their 1st secondary segments at the same (another) server as well etc… One may expect this policy to be usually beneficial for the query processing speed. At the expense however, perhaps of the uniformity of the processing and storage load among the server NDBs.

If $M < N$, it means there is no enough of NDBs in the SDB of the segment to carry out the split. The splitter attempts then to extend the SDB with new server or peer NDBs. It selects (exclusively) possibly enough nodes in the meta-database which are not yet in the SDB. We recall that the latter data are at the meta-table *Nodes* and *SDB* at primary SDB node. If it does not succeeds the splitting halts with a message to the administrator. Otherwise, it updates the *NDB* meta-table, asks SQL Server to create the new NDBs and allocates these to the remaining new segment(s).

Once done with the allocation phase, the splitter creates the new segments. Each new segment should have the schema of the splitting one, including the proper name, the key and the indexes, except for the values of the *check constraint* as we discuss below. Let *S* be here the splitting segment, let *p* be p = INT (b/2), let *c* be the key, and let S_i denote the new segment at node *Ni*, destined for portion P_i. The creation of the new segments loops for $i = 1…N$ as follows:

o It performs the SQL Server query in the form of

SELECT TOP *p* (*) INTO *Ni.Si* FROM *S* ORDER BY *c* ASC

o It finds the key of *S* using the SQL Server system tables and alters S_i scheme accordingly. It uses SQL Server system tables *information_schema.Tables* and *information_schema.TABLE_CONSTRAINTS*. The splitter will join these tables on the *TABLE_SCHEMA*, *CONSTRAINT_SCHEMA* and *CONSTRAINT_NAME* columns and then returns the primary key of *S*.

o It determines the indexes on *S* using the SQL Server stored procedure *sp_helpindex*. It then creates the same indexes on S_i using the SQL Server *create index* statements.

o It creates the check constraint on S_i as we describe below.

o It registers of the new segment in the SD-SQL Server meta-tables. It inserts the tuples describing S_i into (i) *Primary* table at the new node, and (ii) *RP* table at the primary node of *T*. It also inserts the one with the S_i size into *Size* at the new node.

o It deletes from *S* the copied tuples.

o The splitter computes each *check constraint* as follows. We recall that, if defined for segment *S*, this constraint *C(S)* defines the low *l* and/or the high *h* bounds on any partition key value *c* that segment may contain. We have: $C(S) = \{ c : l < c \leq h \}$. The creation of a primary segment does not set the constraint, unless the command defines the bounds. Let thus h_i be the highest key values in portion $P_{i>1}$, perhaps, undefined for P_1. Let also h_{N+1} be the highest key remaining in the splitting segment. Then the low and high bounds for new segment S_i getting P_i is $l = h_{i+1}$ and $h = h_i$ .The splitting segment keeps its *l*, if it had any, while it gets as new *h* the value $h' = h_{N+1}$, where $h' < h$. The result makes *T* always range partitioned.

Once the loop finishes, the splitter commits which makes SQL Server to release all the locks.

Notice that the scheme may lead to a distributed deadlock, in the following case. One may reasonably expect it fortunately very rare, and it would still preserve the serializability in practice. Consider that we have a multiple segment split, let us say of two segments, *S* and *S'*. This would launch two splitters. They may start concurrently. The splitter of *S* may get the lock on tuple of *S* in *RP*. In about the same time, the splitter of *S'* may get the lock on the *S'* tuple. Next, both splitters may concurrently issue the queries searching for the available NDBs. The query of *S* splitter would block on the *S'* lock, and vice versa.

The executions would then wait during the timeout till SQL Server at the node of *RP* automatically aborts and restarts one of the splits, after some delay. Let us say, it would restart the split of *S'*. Depending on this delay, this one would either (i) wait on the shared lock on *S'* tuple obtained in the meantime till the split of *S* commits, or would (ii) restart once the split of *S* already committed, or (iii) would got for any reason the exclusive lock on the S' tuple again before the *S* splitter's query comes into, being re-aborted and restarted with a delay. In any case, at some point *S'* would get its lock. It then would normally find

that either *S'* does not need the split anymore, or would eventually execute its split. This, although, in theory, it could deadlock again with a new split of *S* etc.

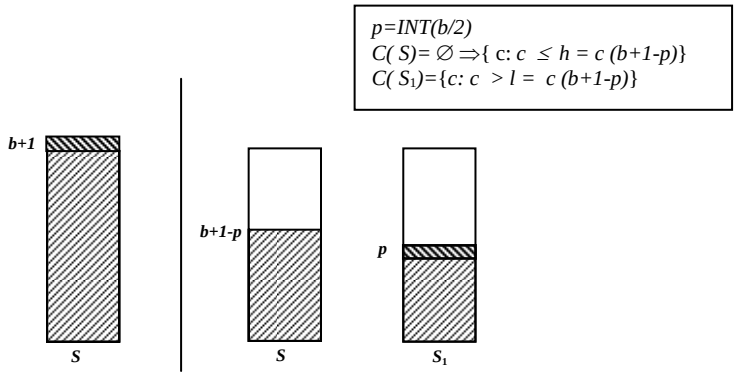


Fig 3 Tuple insert split of the primary segment (thus without the check constraint)

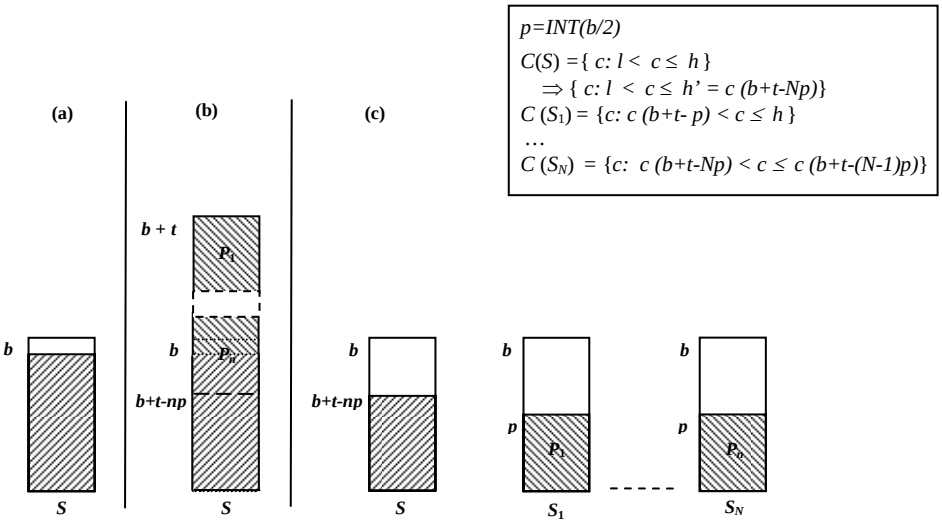


Fig 4 Single segment bulk insert split creating $N \geq 1$ new segment. The splitting segment already has a check constraint. (a) Segment before the insert, (b) after the insert, (c) after the split

Example 4-3. Consider an insert of 160000 tuples into empty *_Dell1_PhotoObj* primary segment of *PhotoObj*. Suppose the segment capacity of *PhotoObj* to be b = 150000 tuples. The insert will trigger the split. Let the query to *Nodes* and *RP* find that there is no *_Dell2_PhotoObj* segment in *SkyServer* NDB at node *Ceria2*. Let suppose that the splitter chooses this node. It then proceeds as follows:

o It creates *_Dell1_PhotoObj* segment at *Ceria2.SkyServer* with the *PhotoObj* table scheme.

o It transfers the portion of p = 75000 tuples to the new segment. These are the upper tuples in the segment with respect to the order on *Objid*.

o It defines the new *Check Constraints* on the splitting and the new segment. On the splitting one, it sets the high value *h* to the highest remaining value of *Objid*. This is *Objid* of the tuple with rank 85000 that we note *Objid* (85000). The low value *l* remains undefined, as it was before. On the new segment, *Objid* (85000) becomes in turn the low value *l*, while *h* remains undefined there.

o It registers the new segment, i.e., it inserts (i) the tuple *(Dell2, Dell1, PhotoObj)* into *Ceria2.SkyServer.SD.Primary* meta-table, pointing thus to the primary node of *PhotoObj*, and (ii) the tuple *(Ceria2,Dell1,PhotoObj)* into *Dell2. SkyServer.SD.RP* meta-table, updating thus the definition of the current *PhotoObj* partition.

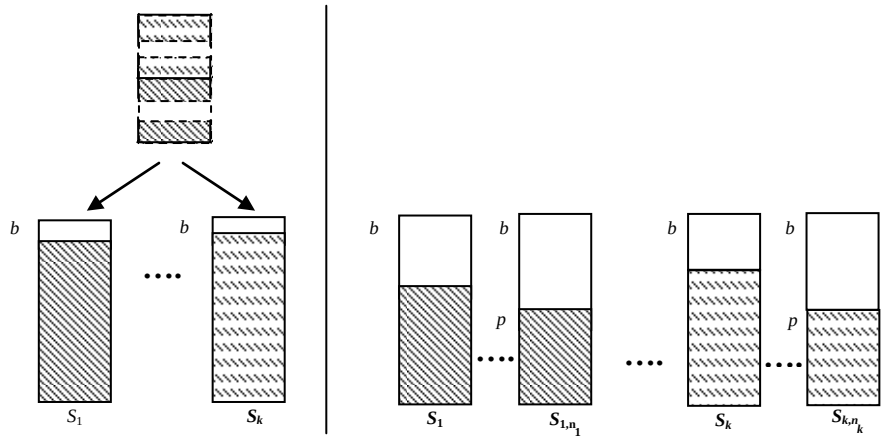


Fig 5 Multi-segment split

o It deletes the transferred tuples from the splitting *_Dell1_PhotoObj* segment at *Dell2* node.

Consider now the same insert, but suppose b = 100 000. Now the portion size is p = 50000 and the split creates 2 new segments with the capacity of b = 100 000 each. Let *Ceria2 and Dell3*nodes be chosen for these segments. Let the first portion go again to *Ceria2*. These are the uppest tuples in the segment, with respect to the order on *Objid*. The portion fills again half of the new segment. The splitting segment would remain now with 110 000 tuples. It is more than b hence the splitting continues. Next portion of 50000 tuples is cut off and sent to *Dell3*. It also fills there half of the new segment. The splitting segment contains now 60000 tuples. This is under b, hence the splitting ends up.

4.3.2.2 Image Adjustment

This operation occurs when a scalable query, i.e., a user query using any command of SD-SQL Server with the selection expression, invokes an image of some scalable table, let it be *T*. The image can be primary or secondary, invoked directly or through a (scalable) view. SD-SQL Server produces from the scalable query, let it be *Q*, an SQL Server query *Q'* that it passes for the actual execution. *Q'* actually addresses the distributed partitioned view defining the image that is *dbo.T*. It should not use an outdated view. *Q'* would not process the missing segments and *Q* could return an incorrect result.

Before passing *Q'* to SQL Server, the client manager first checks therefore the image correctness with respect to the actual partitioning of *T*. *RP* table at *T* primary server let to determine the latter. The manager retrieves therefore from *Image* the presumed size of *T*, in the number of segments, let it be S_I. It is the *Size* of the (only) tuple in *Image* with *Name* = '*T*'. The client also retrieves the *PrimNd* of the tuple. It is the node of the primary NDB of *T*, unless the command *sd_drop_node_database* or *sd_drop_node* had for the effect to displace it elsewhere. In the last case, the client retrieves the *PrimNd* in the *SD.SDB* meta-table. We recall that this NDB always has locally the same name as the client NDB. They both share the SDB name, let it be *D*. Next, the manager issues the multi-database SQL Server query that counts the number of segments of *T* in *PrimNd.D.SD.RP*. Assuming that SQL Server finds the NDB, let S_A be this count. If $S_A = 0$, then the table was deleted in the meantime. The client terminates the query. Otherwise, it checks whether $S_I = S_A$. If so, the image is correct. Otherwise, the client adjusts the *Size* value to S_A. It also requests the node names of *T* segments in *PrimNd.D.SD.RP*. Using these names, it forms the segment names as already discussed. Finally, the client replaces the existing *dbo.T* with the one involving all the newly found segments.

The view *dbo.T* should remain the correct image until the scalable query finishes exploring *T*. It implies, that no split modifies *T* partitioning since the client requested the segment node names in *PrimNd.D.SD.RP*, till *Q* finishes manipulating *T*. Giving our splitting scheme, it means in practice that no split starts in the meantime the deletion phase on any *T* segment. To ensure it, the manager requests from

SQL Server to processes every *Q* as a distributed transaction at the serializable isolation level. We recall that SD-SQL Server uses the same level for the splits. The counting in *PrimNd.D.SD.RP* during *Q* processing generates then a shared lock at each selected tuple. Any split of *T* in progress has to request an exclusive lock on some such tuple, registering the segment where the split should start deletion. According to its 2PL protocol, SQL Server would then make any *T* split waiting till *Q* ends up. Vice versa, *Q* in progress would not finish, or perhaps even start, the S_A count and the *T* segment names retrieval, till any split in progress ends. *Q* will take then to the account the newly added *T* segments as well. In both cases, the query and split executions remain serializable.

Finally, we use the *lazy schema validation* option for our linked SQL Servers, as we mentioned in Section 2. When starting *Q'*, SQL Server drops then the preventive checking of the schema of any remote table referred to in a partitioned view. The run-time performance obviously must improve, especially for a view referring to many tables [9]. The potential drawback is a run-time error generated by discrepancy between the compiled query used towards the view, hence *dbo.T* in our case, and some alterations of *T* schema by SD-SQL Server user since, requiring *Q* recompilation on the fly.

Example 4-4. Consider query *Q* to the *SkyServer* SDB at the peer node *Peer1*:

sd_select '* from PhotoObj'

Suppose that *PhotoObj* is here a scalable table created locally, and with the local primary segment, as typically for a scalable table created at a peer. Hence, *Q'* should address *dbo.PhotoObj* view and is here:

SELECT * FROM dbo.PhotoObj

Consider that *Peer1* manager processing *Q* finds *Size* = 1 in the tuple with of *Name* = *'PhotoObj'* retrieved from its *Image* table. The client finds also *Peer1* in the *PrimeNd* of the tuple. Suppose further that *PhotoObj* has in fact also two secondary segments at *Peer2 and Peer3.* The counting of the tuples with *Table* = '*PhotoObj'* and *CreatNd = 'Peer1' in Peer1.SkyServer.SD.RP* reports then $S_A = 3$. Once SQL Server retrieves the count, it would put on hold any attempt to change *T* partitioning till *Q* ends. The image of *PhotoObj* in *dbo.PhotoObj* turns out thus not correct. The manager should update it. It thus retrieves from *Peer1.SkyServer.SD.RP* the *SgmNd* values in the previously counted tuples. It gets {*Peer1, Peer2, Peer3*}. It generates the actual segment names as '*_Peer1_PhotoObj*' etc. It recreates *dbo.PhotoObj* view and updates *Size* to 3 in the manipulated tuple in its *Image* table. It may now safely pass *Q'* to SQL Server.

4.3.2.3 Table Alteration

The *sd_alter_table* command alters a scalable table, let it be *T*. The alteration of a scalable table offers all the traditional functions of *ALTER TABLE*: add, drop or alter a column…. In addition, it allows updating the maximal size of a scalable table. Its processing starts by determining if it modifies the segment capacity of a scalable table or its schema.

In the former case, it acts as below:

o First, it gets the primary NDB of the scalable table to alter from the *SD.Image* table where the command is executed.

o Next, it goes on the *SD.Size* table of the NDB of the scalable table to alter and updates the tuple describing its segment capacity. It replaces its maximal size by the new entered one in the command.

o In the case of the alteration of the scalable table schema, SD-SQL Server acts as below:

o As for the first case, it gets the primary NDB, let it be *D*, of the *T* scalable table to alter.

o It goes on the *SD.RP* table of the *D* primary NDB and loops over all the existing segments of the *T* scalable table. To preserve the serializability with the other operations, in particular the split, the *sd_alter* command puts a shared lock at all the tuples that it uses in *RP* meta-table. This put on wait any split to get its exclusive lock on the tuple, describing the splitter segment, in *SD.RP* table. Once the alteration finishes with *RP* meta-table, the split gets its exclusive lock. If at this moment another alteration starts, it will be made on wait until the splitter finishes its process. Consequently, the segment added from the split should also be altered. Thus, the concurrency schema allows the serializability of both alteration and splitting.

4.3.2.4 Index Creation and Removal

SD-SQL Server follows the same steps for the index creation and removal command processing. Let the command concern the scalable index I at the scalable table *T*. The manager executing it basically iterates the SQL Server index creation or removal command at each *T* segment. More in detail, it goes through the following steps:

o It retrieves the *PrimND* in the *SD.Image* table at its node.

o It loops through *SD.RP* at the NDB at *PrimND* node. For each tuple describing a segment of the *T* scalable table, it gets its value of *SgmNd*. Next, it requests SQL Server to create or remove index *I* at *T* segment on the *SgmNd* node.

The processing respects the serializability. Here, it means, in the nutshell, that when the command processing ends either all the existing segments of *T* get the index, or none of them. Indeed, the above accesses to the meta-tables generate the shared locks. The SQL Server *Create Index* commands requests exclusive locks at the segments. A shared lock request at a tuple of *T* segment in *SD.RP* may find an exclusive lock of a split in progress or of a command moving a segment of *T* into another node, or dropping the segment. For details, see the processing of those commands below. If there is such lock, the command waits and later processes the new or moved segments. Otherwise, the loop cannot miss any segment of *T*. With respect to the actual index creation at each segment, SQL Server manages the locks at each segment internally as usual, providing the serializability accordingly.

4.3.2.5 Table Removal

The *sd_drop_table* command removes a scalable table, let it be *T*, as previously described. Internally, it calls the standard SQL Server *DROP TABLE* command for every existing segment. It also removes all the related scalable indexes and meta-data in the SD-SQL Server meta-tables. It acts in detail as follows.

o It searches *Image* for *PrimNd* of the table. Let *N* denote the node found and let *D* be the SDB of *T*, The client issues then the query to *N.D.SD.RP*. If it does not find this NDB (rare case), it searches for the NDB with *RP* with *T* through the meta-table *SD.SDB*, defining the current SDB. Once the *RP* is located, the query will act as follows:

o It gets an exclusive lock on the tuples describing the *T* scalable table segments to remove in the *RP* table. Thus, if a splitter comes to put its exclusive lock on a tuple describing one of these segments, it will be made on wait. Otherwise, if the *sd_drop_table* finds the exclusive lock put by a splitter on one of these tuples, then it will be put on wait until the splitter finishes.

o It goes on each node where the segments of the scalable table to remove are located. At each node, it drops the *T* segment and also removes the tuple describing its primary node in the Primary table.

o After removing each segment of the *T* scalable table, it removes the tuple describing it in the *RP* table of the primary NDB.

o It removes the tuple that describe the segment capacity of *T* in the *SD.Size* meta-table.

Notice that *sd_drop_table* doesn't remove the scalable table images. This is due to the state of the client that detains the image. If a client is not connected for example the image removal will fail. Thus, the image removal is not done with its scalable table removal. It is only done when the client doesn't find that the related scalable table, thus it will need to remove itself later, as we will describe latter.

Example 4-5. The following command drops the *PhotoObj* scalable table. *Dell2.SkyServer* is the original node where *PhotoObj* was initially created by the *Dell1* client. Let *PhotoObj* be a scalable table partitioned into 2 segments at *Dell2.SkyServer* and *Ceria2.SkyServer* nodes with the public user (*.dbo*) as in the previous example..

sd_drop_table 'Dell2.SkyServer.dbo.PhotoObj'

The processing of the *sd_drop_table* command for the *PhotoObj* scalable table follows the steps below:

o Drop the *_Dell1_PhotoObj* segments at *SkyServer* NDBs of *Dell2* and *Ceria2* respectively.

o Remove the tuples *(Dell2,Dell1,PhotoObj)* from *Dell2.SkyServer.SD.Primary* and *Ceria2.SkyServer.SD.Primary* respectively.

o Delete the tuple *(Dell1, PhotoObj, 160000)* from *SD.Size* meta-table.

o Delete the tuples *(Dell2, Dell1, PhotoObj), (Ceria2, Dell1, PhotoObj)* from *SD.RP* meta-table.

4.4 Secondary Image Management

4.4.1 Image Creation

We recall that an image can be primary or secondary. Both have the same role, i.e. defining their scalable table. What differs is that the primary images are created on the client NDB of their scalable table creation at the same time. The secondary images are created on clients that are not those where the related primary image is created. Otherwise, a primary image has the same name of their scalable table, *T* for example, and prefixed with the *SD* user. However, the secondary image name is like *C_T* and prefixed by the *dbo* users unlike.

The *sd_create_image* scalable command provides then only the creation of a secondary image. These are the steps that the secondary image creation follows:

o The image is created on a client NDB, let it be *C1*, which executes the scalable table creation, let it be *T*, and related to this image. As already described, the secondary image name is then *C_T* (C is the client NDB on which the scalable table *T* is created). To get the secondary image definition, the client uses the NDB parameter entered in the *sd_create_image* command that indicates the primary NDB, let it be *D* found at *N* node, of the *T* scalable table. Thus, the image definition will be as follow:

```
CREATE VIEW dbo.C_T AS SELECT * FROM N.D.dbo._C_T
```

o Next, the *C1* client adds a tuple describing the created image in *SD.Image* table, i.e *(PhotoObj .false,D,1)*. In order to avoid the concurrency conflict with other queries that use *SD.Image* table at the same time, the client puts an exclusive lock on this table. So if another query, an alter one for e.g., comes to alter the *T* scalable table, it finds the exclusive lock of the image creation. Thus, it waits the end of this operation to get its shared lock.

4.4.2 Image Removal

The *sd_drop_image* command removes a secondary image as already shown. It also removes all meta-data related to the secondary image to remove. Let *D_T* the secondary image to remove. We recall that the removal operation is executed on the client of the secondary image. These are the steps it follows:

o It gets from the secondary image name the client node, i.e. *D*, of the related primary image *T*.

o Next, it goes to the *SD.Image* table and puts an exclusive lock on the tuple describing *D_T* image. This will put on wait any concurrent operation on *SD.Image*. If the *sd_drop_image* finds a shared lock put by a delete query on *SD.Image*, it waits the end of this query to get its exclusive lock. Thus, the query can use the image before that the *sd_drop_image* command removes it. The serializability is then preserved with this schema.

o Finally it removes the *D_T* image.

4.5 Scalable Query Processing

A *scalable* query consists for an SD-SQL Server client (peer) from query command followed with the (scalable) *query expression.* We recall that these commands are *sd_select*, *sd_insert*, *sd_update* and *sd_delete*. Each command names the stored procedure implementing it. The query expression constitutes one or more input parameters of the procedure. The syntax of the valid expressions follows the standard SQL rules for the query expression of the related SQL (static) command, e.g. *select* for *sd_select*. It is also command dependant and has for each command additional SD-SQL Server rules, seen in Section 4.1. Every scalable query, unlike static one, starts with the *image binding* phase that determines every image on which a table or a view name in a query depends. The client verifies every image before it passes to SQL Server any query to the scalable table behind the image. We now present in depth the processing of scalable queries under SD-SQL Server. We first discuss the image binding. Next, we discuss each command specifically.

4.5.1 Image Binding

The client (manager) parses every *FROM* clause in the query expression, including every sub-query, for the table or view names it contains. The table name can be that of a scalable one, being then that of its primary image. It may also be that of a secondary image. Finally, it can be that of a static (base) table. A view name may be that of a scalable view or of a static view. Every reference has to be resolved. Every image found has to be verified and perhaps adjusted before SD-SQL Server lets SQL Server to use it, as already discussed.

The client searches the table and view names in *FROM* clauses, using the SQL Server x*p_sscanf* function, and some related processing. This function reads data from the string into the argument locations given by each format argument. We use it to return all the objects in the *FROM* clause. The list of objects is returned as it appears in the clause *FROM*, i.e. with the ',' character. Next, SD-SQL Server parses the list of the objects and takes every object name alone by separating it from it *FROM* clause list. For every name found, let it be *X*, assumed a proper name, the client manager proceeds as follows:

o It searches for *X* within *Name* attribute of its *Image* table. If it finds the tuple with *X*, then it puts *X* aside into *check_image* list, unless it is already there.

o Otherwise, the manager explores with *T* the *sysobjects* and s*ysdepends* tables of SQL Server. Table *sysobjects* provides for each object name, its type (*V* = view, *T* = base table…) and internal *Id*, among other data. Table s*ysdepends* provides for each view, given its *Id*, its local (direct) dependants. These can be tables or views in turn. A multi-database base view does not have direct remote dependants in *sysobjects*. That is why we do not allow for scalable multi-database views at present. The client searches, recursively if needed, for any dependants of *X* that is a view that has a table as dependant in *sysobjects* or has no dependant listed there. The former may be an image with a local segment. The latter may be an image with remote segments only. It then searches *Image* again for *X*. If it finds it, then it attempts to add it to *check_image*.

o Once all the images have been determined, i.e., there is no *FROM* clause in the query remaining for the analysis, the client verifies each of them, as usual. The verification follows the order on the image names. The rationale is to avoid the (rare) deadlock, when two outdated images could be concurrently processed in opposite order by two queries to the same manager. The adjustment generates indeed an exclusive lock on the tuple in *Image*. After the end of the image binding phase, the client continues with the specific processing of each command that we present later.

With respect to the concurrent command processing, the image binding phase results for SQL Server in a distributed transaction with shared locks or exclusive locks on the tuples of the bound images in *Image* tables and with the shared locks on all the related tuples in various *RP* tables. The image binding for one query may thus block another binding the same image that happened to be outdated. A shared lock on *RP* tuple may block a concurrent split as already discussed. We'll show progressively that all this behaviour contributed to the serializability of the whole scalable command processing under SD-SQL Server.

4.5.2 Scalable Search

The *sd_select* command carries a *select* expression and, perhaps an *INTO* clause. The latter names a scalable table if the query expression provides also the segment size and the partition key for the table to create. Otherwise, the clause names a static table. The manager processes a scalable search, let it be *Q*, with the INTO clause or with INTO clause creating a static table basically as follows:

o It forms an SQL query, let it be *Q'*, with the query expression provided, following the SQL *select* command.

o It binds the images referred to in *Q'*.

o It passes *Q'* for the execution to SQL Server at the serializable isolation level.

This execution scheme is obviously amended if errors occur, e.g., the name used in the query expression does not designate any object.

If there is *INTO* clause creating a scalable table, let it be *T*, then the segment size and the partition key for *T*, constitute the additional input parameters. Once the client finds these parameters, it proceeds also as follows:

o It registers T as a scalable table, i.e., T gets a primary segment site and the entries into SD-SQL Server meta-tables.

o It replaces T name in the expression with that of its primary segment, let it be _N_T, prefixed with N.D, where D is the name of SDB used (through the USE D clause of SQL Server).

o It then proceeds as for the scalable search already discussed.

o Next, it creates the trigger on the N.D.dbo._N_T table.

o It searches for any tuple in T, e.g. using TOP 1 predicate, deletes it and reinserts.

The last step triggers the split if *T* turned out to overflow.

Example 4-6

Here, we show a set of scalable select queries. We give different cases that can be presented for users. We use for all the examples below the *PhotoObj* scalable table. We suppose that it is partitioned into 3 segments at *Peer1*, *Peer2* and *Peer3* respectively. Let *Peer1* be the initial node where *PhotoObj* was created. We execute scalable select query on *Peer2* node:

(S1) **sd_select** '* FROM PhotoObj, T1'

In *(S1)* statement, *PhotoObj* is a scalable table and *T1* is a static table. We suppose that *PhotoObj* contains three segments at *Peer1*, *Peer2* and *Peer3*. Thus, the definition of the *PhotoObj* secondary image (scalable view) at *Peer2* is as below:

CREATE VIEW PhotoObj AS SELECT * FROM _Peer1_PhotoObj

The *PhotoObj* secondary image doesn't contain all the segments of the *PhotoObj* scalable table in its definition and thus should be adjusted to include all the scalable segments. Applying the query processing described above, SD-SQL Server will get the objects in the *FROM* clause of the input parameter of the *select* command, i.e *PhotObj* and *T*. Next, it applies the algorithm described above on each object. It finds that *PhotoObj* is a scalable table and it is not adjusted, thus it adjusts it before it executes the query.

(S2) **sd_select** 'COUNT (*) FROM T2'

In (S2), we suppose that *T2* is a static view defined as below:

CREATE VIEW T2 AS SELECT * FROM PhotoObj

When parsing the query, SD-SQL Server will find *PhotoObj* as a depending object of *T2*. As *PhotoObj* is a scalable table, thus it adjusts at first *PhotoObj*, which is not already adjusted, and then executes the query.

(S3) **sd_select** 'TOP 5 P.objid FROM PhotoObj as P'

(S3) gives an example of a query with top and alias. The execution of this statement will select the first five tuples from the *PhotoObj* scalable table.

(S4) **sd_select** 'TOP 10 objid INTO S FROM PhotoObj', 150, 'Objid'

The execution of *(S4*) will create a new table *S* as a copy of *PhotoObj* with only the 10 first tuples. The new *S* table is created without primary keys, thus we use the key attribute entered by the user (the 3rd parameter: '*Objid*') to define the primary key. A trigger and a primary image *S*, related to the *S* scalable table, are also created on a new server, let it *D*, selected from the *PrimServ* meta-table of the current client *Peer2*. Next, we put data related to *S* table in the meta-tables. Tuples *((D, Peer2, S), (Peer2, S, 150)* and *(D, Peer2, S)* are respectively inserted into *SD.Size, SD.RP, SDprimary* of the *D* node where the primary segment of the *S* scalable table is created. The *(S, .true, D, 1)* tuple is inserted into *SD.Image* of the node which creates the *S* scalable table, i.e. *Peer2* client.

4.5.3 Scalable Updates

The *sd_insert, sd_update and se_delete* commands follow the same processing phases as follows:

o The manager binds the images to (i) the modified table name, (ii) all the table names in selection expression of the command.

o It generates the SQL Server command acting according to the scalable command. For *sd_insert* it is thus INSERT etc. The clauses of the generated command are these in the scalable one.

o It launches the execution of the generated command.

The commands execute as the distributed transactions at the repeatable read isolation level. It is quite easy, thought rather tedious to analyse in depth the concurrent execution of different queries and the other operations, proving the overall serializability especially. In the nutshell, two concurrent scalable queries never block on *RP* and might block without deadlock on common *Image*, as already observed. They may block on common segments. The image binding phase is clearly serializable. The segment processing should be as well. It is purely managed by the SQL Server, trusted serializable. The concurrent processing of a query and of a split was already analysed. The analysis of the concurrency with respect to the node management commands will follow.

Example 4-7

Let the insertion below into the *PhotoObj* scalable table with the *sd_insert* command:

sd_insert 'PhotoObj (objid) VALUES (2255031257923860)'

SD-SQL Server gets the *PhotoObj* table used in the insert query. It finds that is a scalable table by looking for it in *Image* table. Then, it checks if *PhotoObj* image is adjusted. It gets its segments number from the *RP* table and compares this number with the one found in Image table. We suppose that it is the same number, and then PhotoObj image is adjusted. Finally, the *sd_insert* command will execute the *INSERT* clause.

4.6 Node Management

These SD-SQL Server capabilities involve the creation or the removal of a node, an SDB or an NDB.

4.6.1 Node, SDB or NDB Creation

The application creates an SD-SQL Server node through the script for the primary one and through the command otherwise. The script creates the MDB with its tables and loads into all the stored procedures needed at a peer node. The node creation command let it be of node *N*, simply registers *N* with *SD.Nodes*. The manager getting the SDB creation command, let it be named *D*, of type *T*, and at node *N* again, acts as follows:

o It creates an SQL Server database *D* and registers its name, type and node in *SD.SDB* in the MDB. The database becomes the primary NDB for SDB *D*.

o It creates in *D* the user *SD* and SD-SQL Server meta-tables. Besides, *SD.NDB* and *SD.MDBNode* these depend on the type of *D*. It initiates *SD.NDB* and *SD.MDBNode*.

o It copies into *D*, from MDB, the SD-SQL Server stored procedures constituting the manager of *T*.

o The NDB creation command acts similarly, except that it registers the NDB in *SD.NDB* of its primary NDB.

4.6.2 NDB Removal

Dropping an NDB is a more complex operation than to create one. If it is a client NDB with some scalable tables, then the manager should also drop all the remote segments, if any, of these tables. If it is a server NDB with some segments, the manager should preserve elsewhere these segments. Finally, it it is a peer NDB, the manager should act both as for the client and as for the server. As for the already discussed operations, the processing of the NDB removal should also preserve the serializability of a concurrent execution of any commands anywhere under SD-SQL Server. These could possibly involve the NDB to drop or any meta-data that the manager has to use while processing the command. As usual, the processing of the command should besides let the concurrent operations to be least affected, and should possibly avoid deadlocks. With the aim at all these constraints, the NDB removal acts as follows.

The manager first tests the NDB type, using *SD.NDB*. If it is a client, then the manager first, loops over its *Image* table. It fetches a tuple requesting an exclusive lock. If it gets it, it means there is no client's query in progress on the scalable table referred to. The manager tests then the image type. If it is a secondary

image, it simply deletes the tuple. Otherwise, for each primary image thus, the manager drops the scalable table referred to. This can be done only if there was no concurrent query or split in progress over the dropped table. Once done with the tuple, the manager removes it and fetches the next one, if any remains. Once done with the loop, the manager attempts to remove the tuple registering the *NDB* at its *SD.NDB*. If it succeeds it means that there was no lock at this tuple, hence no split somewhere already in progress of segment creation at the NDB. Finally, it requests SQL Server to drop the database underlying the NDB. This may drop the manager itself. The manager acts therefore for the whole operation in fact through the launch of an asynchronous job.

If the NDB to drop is a server NDB, the manager acts in the nutshell as follows. It finds a spare node (for the SDB) and creates an NDB there. It then copies there every remaining segment. It recreates the key, the check constraint and the indexes. It also updates accordingly the meta-data at the new NDB, and those on other nodes affected by the move, the *Primary* tables especially. Finally it drops the existing NDB. In detail, because of the already discussed concurrency control constraints, this process is quite complex. It goes through the following steps. We denote the dropped NDB as *N.D* and the new one as *N'.D*:

o The manager creates *N'.D* using internally the *sd_create node_database* command, without the commit. This operation inserts the tuple with *N'.D* into *NDB* table at the primary *NDB* of SDB *D*.

o Next the manager deletes the tuple with *N.D* from *SDBNode* table at the primary node. It finds its address in its *SDBNode* table or through its *MDBNode* table (there is a rare possibility that an *SDBNode* pointer is outdated, if the primary *NDB* was dropped, since the client last attempted to access it). Once performed, the operation prevents any concurrent splitter at any node from getting *D* as an NDB for a new segment. Likewise no concurrent NDB or node removal can get it. Every concurrent splitting already in progress on *D* had to finish.

o The manager loops over the tuples in *Primary* table in *D*. If it does not find any tuple, it means there is no segment to save. Otherwise, for any tuple found, (i) it requests an exclusive lock, and (ii) once it gets it, then (ii) it requests a shared lock at the segment, and (iii) it copies the segment the tuple registers into the new *D*, as above discussed. If it does not get the lock in step (i), it normally means that there is a shared lock of a split in progress started at *D*. This would make the drop to wait till the split ends up. If it does not get the lock (ii), it means there is an update query in progress. This would make the copying to wait, and could lead to a deadlock with the query. The query may have to finish the update on another segment that the manager locked in the meantime. The shared lock however does not affect any search over the segment.

o The manager updates in *Primary* every tuple registering a primary segment at *D*. It sets the *PrimNd* value to that of the new *D* node name. Once done, the manager copies *Primary* into new *D*.

o The manager requests an exclusive lock on *RP*. Once it gets it, it means there is no segment at *D* serving a query. From now on, and only from this point, any new query involving a segment at *D* will be put on wait at the stage of its image adjustment, so before it accesses any data.

o The manager now loops again through *Primary*. For each tuple, it updates the related tuple in *RP*, pointing to *D*, so it points to new *D* instead. Next, it copies *RP* to new *D*.

o The manager determines from *SDB* whether *D* is primary in its SDB. If so, it requests the exclusive lock on *SDB* table at *D*. Once it gets it means there is no split in progress anywhere in the SDB. Any split coming from now on and needing to search for nodes in *D* waits. Next, the manager copies *SDB* and all the remaining meta-tables to new *D*. Next, it updates *SDBNode* at every NDB so it points to new *D*.

o Finally, it requests SQL Server to drop *D*. If it goes OK, the manager commits the transaction. At this time, SQL Server nodes let continue any query or other command waiting on a tuple in any meta-table manipulated above.

The scheme lets the queries to execute during possibly relatively long operations of segment copying. The queries have to wait only during a relatively short time, when the manager copies *RP* table and some others, and drops *D*. Next, notice that the operation does not adjust any client image. A client could be unavailable anyhow at the time of the operation. A side effect of a server NDB dropping may be that every existing image of the scalable tables whose segment moved becomes outdated. SD-SQL Server may detect this fact at the SQL Server query execution time. First, SQL Server may not find the segment

pointed to by a clause in the distributed partitioned view. Then, the manager at the query node forces the image adjustment. Next, the segment moved elsewhere could be the primary one. The client with the primary or secondary image to verify will not then find the *RP* table at all at the location it should be, or will not find the registration of the segment in *RP* at the NDB at the node. The latter case could occur if an NDB is dropped and then another created at the same node and for the same SDB. In both cases, the manager searches then every *RP* of an NDB currently in the SDB to find the displaced primary segment. Next it updates the client meta-tables, verifies and adjusts the image, as in Section 4.3.2.2, and finally executes the query.

Finally, if the NDB to drop is a peer, then the manager acts as for both cases above.

4.6.3 SDB Removal

To drop an SDB, the manager removes (through an asynchronous job) all its peer and server NDBs with everything they contain. It does not remove any client NDB . The latter can be unavailable for access anyhow while the SDB removal takes place. The client NDB will need to remove itself later, when it will not find its SDB through any of its pointers to.

As above the processing has to preserve the serializability of the concurrent operations and the related constraints. The steps of an SDB removal are therefore as follow:

o The manager requests an exclusive lock on the tuple describing the SDB to remove in the *SD.SDB* meta-table. Once it gets it, no split can be in process of creating a new segment in some NDB. Notice that a split could be holding an exclusive lock on some *RP* tuple. This lock would need to be rolled back by SQL Server *drop database* command, or would require SQL Server to break the deadlock. Besides, there cannot be anymore a creation of a DB by SQL Server in progress, on behalf of an NDB creation or removal command. Likewise, it cannot be any concurrent removal of the SDB in progress beyond the same step.

o Next, the manager loops over *NDB* table of the primary NDB of the SDB to remove. For each tuple there, it accesses *RP* table at NDB pointed to. It locks there exclusively every tuple registering a primary segment. Once done, there cannot be any query in progress beyond the search for an image it needs.

o The manager restarts the loop over *NDB*. For each tuple that neither points to the primary NDB, nor to itself, it issues the SQL Server *drop database* command. Next, it drops the primary NDB. It also removes the tuple registering the SDB from *SDB* table in the MDB. Finally, it requests through an asynchronous job from SQL Server to remove its own DB, provided the job does not find the SDB still registered in *SDB* which would mean that the command was rolled back, and it commits.

4.6.4 Node Removal

The command *sd_drop_node* can concern any node, let it be *N*. It selects at *Nodes* in MDB a spare (yet empty) node, let it be *N'*, provided there is one. Next, it deletes *N* from *Nodes*. This blocks every incoming command requesting to create or remove an *NDB* at *N*. The manager moves then to *N'* every NDB at *N*. It also reconstructs at *N'* any meta-table at *N*, with the tuples conveniently updated if the need occurs. To process the command, the manager iterates over *N.SD.NDB*. For each NDB found, except perhaps its own, it loops over its removal, without committing the operation. For itself, it only copies its NDB.

At the end, the manager updates *Nodes* table. If needed, it updates *SDB* as well (at the primary node). The update may concern its own NDB as if it is already moved to *N'*. If *N* was the primary node itself, the manager copies also its meta-tables to the new one. Finally, if its NDB was itself to move, the manager requests as the asynchronous job its DB removal from SQL Server, provided the job does not find the node in *Nodes*. The scheme preserves the serializability, since the removal of an NDB does it. Notice that, while the removed node is no more an SD-SQL Server node, it remains an SQL Server linked node.

5. Experimental Performance Analysis

To validate the SD-SQL Server architecture, we evaluated its scalability and efficiency over some Skyserver DB data. We measured split times and query response times under various conditions. For the former we dealt with different segment capacities and split types, comparing in particular the timing to that of the manual repartitioning by an SQL Server user. For the latter, we aimed at the overhead of the

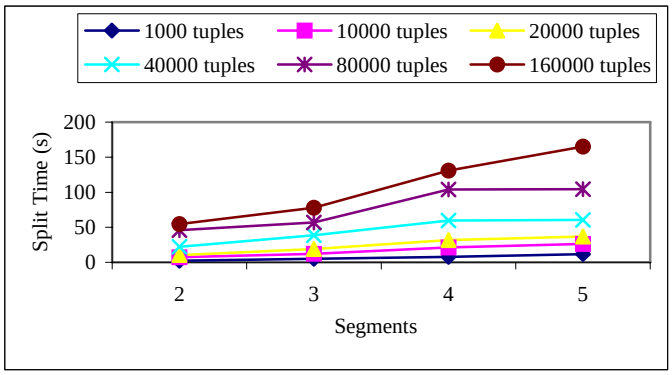
image management. Our hardware consisted of 1.8 GHz P4 PCs with either 785 MB or 1 GB of RAM, linked by 1 Gbs Ethernet. We used the SQL Profiler to take measurements.

5.1 Split Time

We used a 120 GB fragment of the SkyServer database and some of its benchmark queries, [2]. We studied the bulk inserts of 1000…160,000 tuples (260 MB) from the original *PhotoObj* table into an empty 1-segment scalable *PhotoObj* table. We varied the segment capacity *b* for these inserts and the number of tuples loaded. The inserts triggered splits into 2..5. segments accordingly, Fig 6 and Table 1.

For every insert, the split time is predictably the fastest for the smaller 2-segment splits. Likewise, an *i*-segment split, $i = 2…5$, is faster for a smaller insert. In the best case that is the 2-segment split of 1000 tuples, a few seconds suffice. It remains relatively fast, about three minutes for the most demanding one, moving around over 200 MB of tuples.

We furthermore determined the overhead of an SD-SQL Server split, with respect to the same manual repartitioning by an SQL Server user. We have compared our split time with the one of the necessary SQL Server commands on the same tuples. The SQL Server time includes the transfer time of the portions, and the time to delete the transferred tuples. The SD-SQL Server overhead is basically the time to perform a few others above discussed SD-SQL operations. We expected the latter much smaller than the former. The time of the transfer and of the deletion commands took about 40 sec on SQL Server for a 2-segment split of 80000 tuples. The split time under SD-SQL Server was 46 sec (Table 1). Thus, the difference imputable to SD-SQL Server confirmed the efficiency and utility of SD-SQL Server, being only about 6 sec, i.e., about 15 %.



o

Fig 6 Split time (s) as function of split segment size (tuples) and number of resulting segments

	Result			
Segment Size	**2 seg**	**3 seg**	**4 seg**	**5 seg**
1000	2.45	4.83	7.84	11.62
10000	7.11	12.15	21.40	26.42
20000	10.55	18.94	32.08	37.12
40000	22.42	38.88	59.59	60.73
80000	46.17	56.79	104.02	104.15
160000	54.65	77.86	130.89	165.11

Table 1 Split time (s) as function of split segment size (tuples) and number of resulting segments.

Next, we studied the splitting of an indexed segment. We aimed at the additional cost of the update of the existing indexes, and the creation of these on the new segments Fig 7 and Table 2 show the result. The insert size is 160 000 tuples. The blue curve in Fig 7 matches the brown one in Fig 6. The conclusion is that the split time increases naturally with the number of existing indexes and the number of resulting segments. The index management overhead remains nevertheless negligible, under 10 %, for up to 4-segment split. The increase for the 5-segment split is more important the experiment, but remains quite moderate, about 22 %.

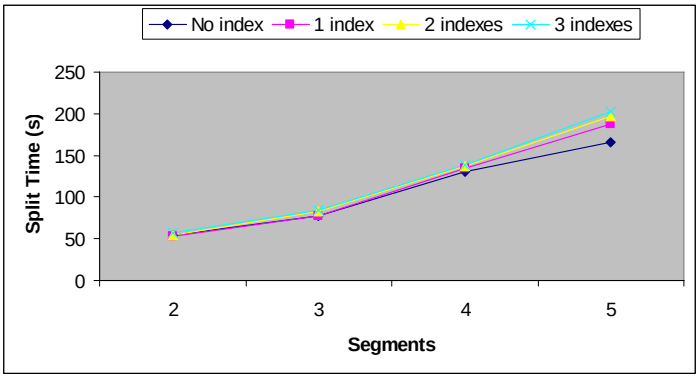


Fig 7 Split time as function of existing indexes and of number of resulting segments

	2	3	4	5
No index	54,656	77,86	130,89	165,11
1 index	52,33	77	133,86	187,813
2 indexes	54,816	82,596	137,626	196,783
3 indexes	56,486	83,61	139	202,65

Table 2 Split time as function of existing indexes and of number of resulting segments

5.2 Image Binding

To study the overhead of image binding, we used queries to our scalable SkyServer DB, modelled upon the actual ones [2]. The following two queries benchmarked the image checking and adjustment overhead:

(Q1) sd_select 'TOP 10 objid FROM PhotoObj WHERE objid not in

(SELECT objid FROM PhotoObj WHERE objid <= @objidMax'

(Q2) sd_select '* FROM PhotoObj WHERE (status &0x00002000>0) AND (status &0x0010> 0)'

Query (Q1) represents the rather fast non-trivial queries. It evaluates through the distributed processing with the response time that has to scale up with *PhotoObj*, as we show below, but brings finally to the application always only a few tuples. The image binding overhead should affect such queries the most. The experimental analysis seems the easiest way to find how much. Especially, whether at least the image checking overhead may turn out to be usually negligible. Query (Q2) represents the expensive queries. It brings 130K tuples from all the segments. The theoretical analysis points towards negligible image binding overhead for both image adjustment and checking. We launched both queries at *Dell2* node where we had the secondary *PhotoObj* image. The table had two segments of various sizes, Fig 8. The node *dell1* was the primary one, thus with the *RP* table. The execution time of (Q1) depended on the *PhotoObj* size because if the subquery. Furthermore, for every segment capacity studied, the parameter *@objidMax* was the maximal key in the 1st segment. Since *objid* was the primary key, the subquery always addressed all and only first segment. SQL Server evaluated the query most likely using the (automatic) indexes on *objid* at the segments only.

The measures of (Q1) show its response time with (i) the image checking only and (ii) with the image adjustment. We compared these times at Fig 8 to (iii) that of the (Q1) within the SQL Server, i.e., that of the generated SELECT query. The user of SQL Server would formulate that one. The difference between (i) and (iii) appeared always negligible. Both case (i) and (iii) execute in about 300 msec for our largest tupleset. If the LZV option was set, the experiments discussed later show that this bound should reduce to under 200 msec. The SD-SQL Server overhead should be constant, as it corresponds always to the same operations that are independent of the query semantics. The curves show thus that, even for short queries, usually, i.e., without the image adjustment, the overhead of query processing by SD-SQL Server should be negligible.

The adjustment overhead in case (ii) dominates the query time that becomes constant. Again, one could expect this result. The total time appears about 0.7 sec. The query time became thus substantially longer. It remains however still largely negligible in practice, given also that the image adjustment should remain a rare operation. The image adjustment overhead time itself appears about 0.5 sec. It is about constant, as it should be. We recall that this time is due to the distributed query to *RP* to bring the new nodes, followed by the SD-SQL Server processing preparing the new image, and SQL Server queries dropping the existing distributed partitioned view and creating the new one. The individual incidence of these operations remains to be measured.

The execution of (Q2) took predictably much longer than for (Q1), about 45 sec with the image adjustment. The execution without required 44 sec. The time of (Q2) within the SQL Server was practically the same. The image checking time stayed naturally the same. It should be so, since it corresponds to the same operations as for (Q1). The incidence of the image checking became proportionally even more insignificant than for a short query. The image adjustment took again under 1 sec, since it corresponds to the same operations as well. But now it represents only about 2% overhead, becoming thus negligible in practice as well. The use of LZV option would get unnoticed. Since the image adjustment processing is independent of the query semantics, the overhead of image adjustment should remain negligible in practice for any long query.

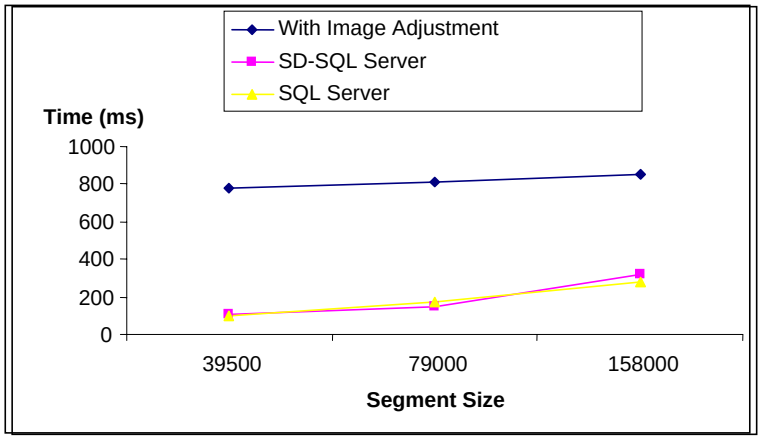


Fig 8 SD-SQL Server query (Q1) execution time

Next, we studied the execution time and the image binding overhead of the queries to scalable views. We were interested in the overhead of a query addressing a view of an image, then a view of a view of an image etc. We call such views of *level i* = 1, 2... The image itself is here at level 0. The image binding loops recursively through SD-SQL Server and SQL Server meta-tables when *i* increases, using the potentially expensive and joins, as discussed in Section 4.5.2. This was the rationale for our set up of this experiment.

We have therefore created the scalable views:

CREATE VIEW T1 AS SELECT * FROM PhotoObj

CREATE VIEW T2 AS SELECT * FROM T1

CREATE VIEW T3 AS SELECT * FROM T2

(Q3) sd_select 'COUNT (*) FROM PhotoObj'

(Q4) 'sd_select COUNT (*) FROM T1'

Etc.

A *PhotoOb* segment had in this example, 39.500 tuples (in 2 segments). Fig 9 shows the result with and without the image adjustment. The curves at the figure increase slightly, but remain about flat in practice. The incidence of the scalable view level on the image binding time, hence the query time is thus negligible. Notice also that the overall times are about those at Fig 9.

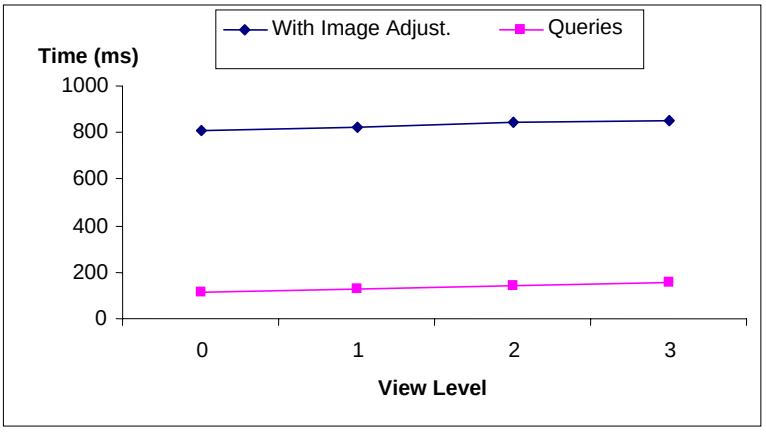


Fig 9 Query (Q3) and queries (Q4) with scalable views time

We have finally executed query (Q3) on growing *PhotoObj* table in various conditions, Fig 9. 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 enters 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 thus 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 (Q3) for a static table. For SD-SQL Server, we measured (Q3) with and without the LSV option.

The figure 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. 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.

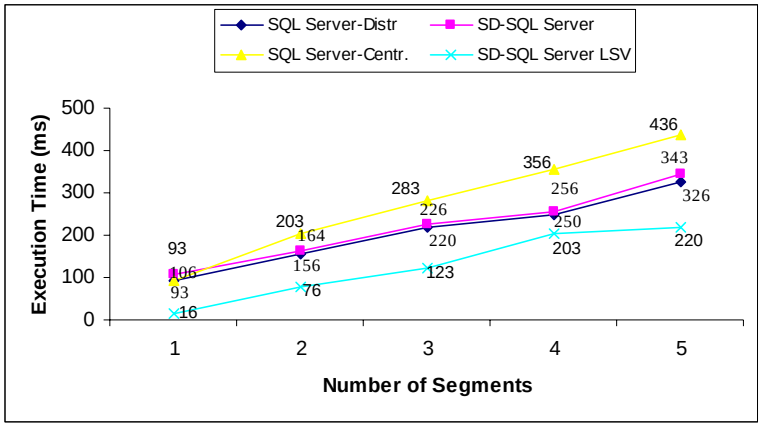


Fig 10 Query (Q4) execution time on SQL Server and SD-SQL Server, including the "Lazy Schema Validation" option.

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. How the scalable tables may be created at these systems remains an open research problem.

7. Conclusion

SD-SQL Server makes the use of scalable tables about as comfortable as that of the static ones. Its user interface lets the user/application to easily take advantage of the new capabilities of our system. Its efficiency of the scalable distributed partitioning processing should allow for much larger tables than those of SQL Server applications today, or for a faster response time of complex queries, or for both.

The current design of our interface is geared towards the "proof of concept" prototype. It is naturally simplified with respect to a full-scale system. Further work could expand it. It may concern the existing commands or add new ones. Both directions should address especially the capabilities for the image and node management. For the images, one could design a scalable sd_*Create_View* command, providing the transparency of the secondary images. It would make any such image a kind of a cache. For the nodes, one could design the command sd_*Connect* uniting separate configurations of SD-SQL Server nodes into one. All such issues lead to various processing choices, to study in consequence.

The processing level has also interesting challenges on its own. There is no user account management for the scalable tables at present. Next, the concurrent query processing could be perhaps made faster, especially during the splitting or NDB removal. We aimed at using the exclusive locks as little as it seemed necessary for the correctness, but there is perhaps still a better way. Our performance analysis should be expanded, uncovering perhaps further directions for our current processing optimization. Next, while SD-SQL Server acts at present as an application of SQL Server, the scalable table management could alternatively enter the SQL Server core code. Obviously we could not do it, but the owner of this DBS can. On the other hand, our design could apply almost as is to other DBSs, provided they also offer the updatable distributed partitioned (union-all) views. On the other hand, we did not address the issue of the reliability of the scalable tables. More generally, there is a security issue for the scalable tables, as the tuples migrate to places unknown to their owners.

Acknowledgments

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

References

1. Ben-Gan, I., and Moreau, T. Advanced Transact SQL for SQL Server 2000. Apress Editors, 2000
2. Gray, J. & al. Data Mining of SDDS SkyServer Database. WDAS 2002, Paris, Carleton Scientific (publ.)
3. Gray, J. The Cost of Messages. Proceeding of Principles Of Distributed Systems, Toronto, Canada, 1989
4. Guinepain, S & Gruenwald, L. Research Issues in Automatic Database Clustering. ACM-SIGMOD, March 2005
5. Lejeune, H. Technical Comparison of Oracle vs. SQL Server 2000: Focus on Performance, December 2003
6. Litwin, W., Neimat, M.-A., Schneider, D. LH*: A Scalable Distributed Data Structure. ACM-TODS, Dec. 1996
5. Litwin, W., Neimat, M.-A., Schneider, D. Linear Hashing for Distributed Files. ACM-SIGMOD International Conference on Management of Data, 1993
7. Litwin, W., Rich, T. and Schwarz, Th. Architecture for a scalable Distributed DBSs application to SQL Server 2000. 2nd Intl. Workshop on Cooperative Internet Computing (CIC 2002), August 2002, Hong Kong
8. Litwin, W & Sahri, S. Implementing SD-SQL Server: a Scalable Distributed Database System. Intl. Workshop on Distributed Data and Structures, WDAS 2004, Lausanne, Carleton Scientific (publ.), to app
9. Microsoft SQL Server 2000: SQL Server Books Online
10. Rao, J., Zhang, C., Lohman, G. and Megiddo, N. Automating Physical Database Design in a Parallel Database, ACM SIGMOD '2002 June 4-6, USA
11. Salzberg, B & Lomet, D. Special Issue on Online Reorganization, Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, 1996
12. Özsu, T & Valduriez, P. Principles of Distributed Database Systems, 2nd edition, Prentice Hall, 1999.
13. Soror Sahri, Witold Litwin, SD-SQL Server: a Scalable Distributed Database System, CERIA Research Report 2005-03-05, March 2005
14. Alexander S. Szalay, Jim Gray, Ani R. Thakar, Peter Z. Kunszt, Tanu Malik, Jordan Raddick, Christopher Stoughton & Jan vandenBerg, The SDSS SkyServer – Public Access to the Sloan Digital Sky Server Data., Technical Report, MSR-TR-2001-104, February 2002

Glossary

Application Interface

The application interface manipulates scalable tables and their views essentially through the SD-SQL Server commands.

Client DB

A client (N)DB manages the user/application interface only.

Client Node

A client node is a node which stores only client DBs.

Image Binding

The image binding determines every image on which a table or a view name in a query depends. It concerns the *FROM* clause of *SELECT* expressions.

Install Script

An install script is an SQL Server script whose execution installs the primary node of SD-SQL Server.

Meta-database

A *meta-database* (MDB) is a specific SD-SQL Server database at the primary node. It contains basic meta-data on all the nodes and SDBs currently under the SD-SQL Server management. It also contains the SD-SQL Server code itself, in the form of stored procedures.

Node Database

A node database (NDB) is a database registered as an element of an SDB. An NDB can be a *client*, *server* or *peer* DB. Under SD-SQL Server, every NDB is an SQL Server DB. It shares there the (proper) name of the SDB. It also contains an instance of SD-SQL Server specific *manager* component.

Peer DB

Peer (N)DB unions the capabilities of the server and client DB.

Peer Node

A peer node is a node that is both server and client node. It can store a client, server or peer DBs.

Primary Image

A primary image is the definition of a scalable table partitioning. It is created at the SD-SQL client, which creates the scalable table. Internally, it is a partitioned and distributed view. It defines the union all of the scalable segments that constitute a scalable table. If any segment of the scalable table is not defined in the primary image, thus the image will be adjusted.

Primary Node

A primary node is the first ever node created for the current SD-DBS configuration. It is created under SD-SQL Server by executing the *install* script. It registers the other (secondary) nodes in its meta-database.

Scalable Database

A scalable (distributed) database (SDB) is a dynamically defined collection of SD-SQL Server *node databases*. SD-SQL Server and its applications may dynamically add or remove node databases to or from an SDB.

Scalable Distributed Database System

A Scalable Distributed Database System (SD-DBS) applies the principles of the Scalable Distributed Data Structures (SDDSs) to database systems. It manages scalable databases with scalable tables dynamically spreading over SD-DBS nodes.

Scalable Query

A scalable query is a query that may invoke a scalable table, through its image name, or indirectly through a *scalable* view of its image. It executes correctly for any dynamic change to the number of the segments in the invoked scalable table. Under SD-SQL Server, the user/application formulates a scalable query using a dedicated SQL Server command. A scalable query may also involve a *static* tables, i.e., created under SQL Server only.

Scalable Table

A scalable table *T* is formally a tuple (*T*, ***S***), where *T* is the primary image of *T* and ***S*** are the segments distributed each in some NDB. It scales through the splits of its overflowing segments, stored each at some NDB.

Scalable view

A scalable view is a view that may depend on a scalable table, directly or through a view. Otherwise a view is *static*. A scalable view may refer to a scalable table through the name of its primary or secondary image.

SD-DBS Manager

An SD-DBS manager is the software component that takes care of SD-DBS specific capabilities. We distinguish between the *client* manager, residing at a client DB, a *server* manager, and a *peer* manager.

SD-SQL Server

SD-SQL Server is an SD-DBS designed for data in SQL Server databases. It manages scalable databases and scalable tables at a collection of SD-SQL Server nodes.

SD-SQL Server command

It is a user/application interface command offered by SD-SQL Server. These commands let to manage the scalable tables, NDBs, SDBs, and the nodes. The commands for the scalable tables generalize these of SQL Server (and SQL more generally) for the static tables. Such commands are typically named upon their originals, e.g., *SD_SELECT* upon *SELECT*.

SD-SQL Server Manager

An SD-SQL Server manager is an SQL Server specific prototype implementation of the SD-DBS manager concept. An SD-SQL Server manager is operationally a collection of stored procedures within each NDB.

SD-SQL Server Node

An SD-SQL Server node is a linked SQL Server node declared for SD-SQL Server as a node. It is the SQL Server specific implementation of the concept of an SD-DBS node.

Secondary Image

A secondary image is a distributed partitioned view, which defines the actual partitioning of a scalable table. It is created on a client, which is not the one creating the scalable table.

Segment

A segment is a table that is fragment of a scalable table. Under SD-SQL Server, the segments range partition the scalable table. A segment can be *primary* or *secondary*. The client creates the primary segment while it creates the scalable table. A secondary segment results from a split.

Segment Capacity

The segment capacity is the maximal tolerated number of tuples in the segment. If an insert makes the segment to exceed its capacity, the segment becomes overloaded and it splits. Under SD-SQL Server all the segments of a scalable table have the same capacity.

Server DB

A server (N)DB stores the segments of scalable tables, without having the user/application interface.

Server Node

A server node is a node which stores only server DBs.

Spare Node

For an SDB, a spare node is a node without an NDB belonging to the SDB. It can carry an NDB of another SDB.

Split

The split of a segment occurs when it overflows. It creates one or more new segments. These absorb the overflowing tuples, as well as up to half of the tuples of the overflowing segment. The split range partitions the splitting segment. SD-SQL Server launches a split by an AFTER trigger at each segment, checking for an overflow after each insert. Every split is carried out by the splitter.

Splitter

The splitter is an asynchronous job which is launched by the splitting trigger. It achieves the split asynchronously with the insert that leads to the overflow.

Static Table

A static table is a table which is unknown by SD-SQL Server. Any relational table, in current DBMSs, are static for SD-SQL Server, thus are not scalable.

.Static View

Under SD-SQL Server, any view which is not scalable is a static view. A static view is an SQL Server view unknown by SD-SQL Server.