UNIVERSITE PARIS DAUPHINE

DFR SCIENCES DES ORGANISATIONS

Centre de Recherche en Informatique Appliquée

THÈSE

Conception et Implantation d'un Système de Bases de Données Distribuées & Scalables : SD-SQL Server

Pour obtenir le titre de

Docteur en Informatique

(Arrêté du 25 avril 2002)

Présentée et soutenue le 13 Juin 2006 par

Soror SAHRI

Composition du Jury

Directeur de Thèse :	**Witold LITWIN**
	Professeur à l'Université Paris Dauphine
Président du Jury :	**Philippe RIGAUX**
	Professeur à l'Université Paris Dauphine
Rapporteurs :	**Georges GARDARIN**
	Professeur à l'Université de Versailles
	Tore RISCH
	Professeur à l'Université d'Uppsala, Suède
Examinateurs :	**Jim GRAY**
	Directeur de E-Science Center, Microsoft Research, USA

Dédicaces

A mes très chers parents Rahima & Tahar

Remerciements

Mes premiers remerciements vont à mon directeur de thèse, Monsieur Witold Litwin, professeur à l'Université Paris Dauphine et directeur du CERIA. Sa grande disponibilité, sa rigueur et la pertinence de ses jugements ont été très constructives et m'ont permis de faire un bon travail. Merci pour tous vos conseils, votre patience et de m'avoir donnée la possibilité d'effectuer ma thèse dans de bonnes conditions. Je tiens à vous exprimer toute ma reconnaissance.

Je remercie vivement Monsieur Georges Gardarin, professeur à l'université de Versailles St-Quentin en Yvelines, d'avoir accepté d'être rapporteur de cette thèse.

Je tiens à remercier Monsieur Jim Gray, Directeur de E-Science Center, Microsoft Research, USA, récipiendaire du Prix *Turing 2000*, d'*ACM* et Docteur *Honoris Causa* de l'Université Paris Dauphine, pour les nombreuses suggestions, la mise à ma disposition de la base *SkyServer*, et pour m'avoir fait l'honneur de participer dans ce jury.

Je remercie vivement Monsieur Philippe Rigaux, professeur à l'Université Paris Dauphine, d'avoir accepté de juger ce travail et pour avoir présidé le jury.

Je remercie également Monsieur Tore Risch, professeur à l'Université d'Uppsala en Suède et directeur du Lab. UDBL, pour le vif intérêt qu'il manifeste à l'égard de mes travaux de recherche.

Je remercie chaleureusement Monsieur Thomas Schwartz, professeur à l'Université Santa Clara aux Etats Unis d'Amérique.

Je remercie chaleureusement Monsieur Pierre-Louis Xech, Responsable des Relations Universitaires de Microsoft Research., pour le soutien financier, matériel et logiciel de Microsoft.

J'adresse mes remerciements les plus vifs à Monsieur Gérard Lévy, professeur à l'Université Paris Dauphine, pour ses conseils et ses encouragements.

Toute ma gratitude et mes remerciements les plus vifs à Madame Régine Vignes-Lebbe, professeur à l'université Paris 7 et directrice du lab. LIS, pour sa confiance, son amitié ainsi que ses conseils et ses encouragements incessants. Je la remercie aussi pour le temps qu'elle m'a accordée pour corriger ce manuscrit.

Je remercie chaleureusement Monsieur Djamel Eddine Zegour, professeur à l'INI d'Alger, pour son aide et ses corrections qui ont permis d'améliorer ce manuscrit.

Un grand merci à Mahat Khelfallah pour son amitié et ses encouragements incessants. Merci aussi pour les corrections minutieuses de ce manuscrit.

Je remercie tous les professeurs qui m'ont permis d'enseigner, notamment, Mme F. Semmak, Mme J. de la Bruslerie, M. Dakhli, M. Carrard et M. Janod.

Les assistantes administratives du CERIA, Mme Janine Verepla et Mme Sylvie Bortzmeyer, pour leur disponibilité et coordination avec les services administratifs de Dauphine.

Une pensée amicale à tous les membres de l'équipe CERIA notamment Riad, Hanafi, Rim et Fatma pour leur amitié et bonne humeur.

Une pensée particulière à Hani pour son aide, son soutien ainsi que ses encouragements incessants. Je remercie aussi tous mes amis qui m'ont assistée et encouragée, avec une pensée à Ania, Nawel, Chabane et Hakim.

Toute ma reconnaissance et mon affection pour mes très chers parents. Tout mon dévouement pour mon frère et mes sœurs. Sans leur amour, leur soutien, leur confiance et leurs encouragements, je n'y serais jamais arrivée.

RESUME

Conception et Implantation d'un Système de Bases de Données Distribuées et Scalables : SD-SQL Server

Le thème de recherche de cette thèse concerne la conception et l'implantation d'un système de gestion de bases de données distribuées et scalables (ang. *Scalable Distributed Database System, SD-DBS*) que nous avons appelé SD-SQL Server. SD-SQL Server implémente une nouvelle architecture de SGBD relationnel. Une base de SD-SQL Server, dite *base scalable*, grandit par la partition dynamique, scalable et distribuée de ses tables, dites *scalables* aussi. La partition et son évolution sont invisibles de l'usager/application. A l'heure actuelle, SD-SQL Server est le seul SGBD à offrir cette possibilité à notre connaissance. Les autres SGBDs connus, n'offrent au mieux que le partitionnement statique, nécessitant des réorganisations périodiques globales. Leur maniement est en général complexe et peu apprécié des usagers.

Nos tables scalables sont organisées en segments d'une structure de données distribuée et scalable (SDDS). Chaque segment est placé sur un nœud lié de SQL Server. A l'heure actuelle, SQL Server supporte 250 nœuds liés, permettant à une table scalable de croître autant. L'expansion dynamique résulte des éclatements de segments débordant leur capacité de stockage. Chaque éclatement est déclenché par une insertion, à l'origine du débordement. Tout éclatement crée un ou plusieurs nouveaux segments. Les segments sont cachés des usagers/applications derrière les vues supportant les mises à jour, dites *images*. Les images sont des vues partitionnées et distribuées de SQL Server dynamiquement ajustées. Les usagers/applications de tables scalables n'interfacent que les images ou les vues des images.

Après l'introduction et la discussion de l'état de l'art, nous présentons l'architecture de SD-SQL Server. Nous discutons ses composants fonctionnels et les opérations qu'ils assurent. L'attention particulière concerne l'expansion d'une table scalable et la gestion évolutive des images. Nous décrivons ensuite, l'interface que SD-SQL Server offre à l'application. Notre interface consiste de requêtes à syntaxe très proche du SQL standard et de commandes d'administration. Nous présentons ensuite l'implémentation de notre prototype. Nous détaillons le traitement interne des commandes, de l'éclatement des tables scalables ainsi que de l'ajustement de leurs images. Nous montrons aussi la sérialisabilité des traitements concurrents. Nous validons ensuite le prototype par des mesures expérimentales de performances. Nous effectuons nos expérimentations sur la base de test *SkyServer*, [G02]. Enfin, nous discutons les développements possibles.

ABSTRACT

Design and Implementation of a Scalable Distributed Database System SD-SQL Server

Our thesis elaborates on the design of a scalable distributed database system (SD-DBS). A novel feature of an SD-DBS is the concept of a scalable distributed relational table, a scalable table in short. Such a table accommodates dynamic splits of its segments at SD-DBS storage nodes. A split occurs when an insert makes a segment to overflow, like in, e.g., B-tree file. Current DBMSs provide the static partitioning only, requiring a cumbersome global reorganization from time to time. The transparency of the distribution of a scalable table is in this light an important step beyond the current technology.

Our thesis explores the design issues of an SD-DBS, by constructing a prototype termed SD-SQL Server. As its name indicates, it uses the services of SQL Server. SD-SQL Server repartitions a table when an insert overflows existing segments. With the comfort of a single node SQL Server user, the SD-SQL Server user has larger tables or a faster response time through the dynamic parallelism. We present the architecture of our system, its implementation and the performance analysis.

After the introduction and the discussion of the state-of-the-art, we present our system architecture. We have based it on the scalable distributed data structures (SDDSs) principles. We present its components and their functionality. We also present our command interface for the users/applications. We discuss the syntax and semantics of each command. We then present the implementation and internal processing of our prototype. We aim the discussion in particular on the efficiency and the serializability of the concurrent processing.

Afterwards, we discuss the experiments proving the efficiency of our system. We benchmark our processing using scalable tables loaded with experimental data from the well-known *SkyServer* database, [G02]. We show that the overhead of our scalable table management should be typically negligible. We conclude through the results, published in international conferences, that our prototype attained its goals. We finally, discuss the future work that would further extend the capabilities of our system.

EXTENDED ABSTRACT

Design and Implementation of a Scalable Distributed Database System SD-SQL Server

The Goal

Our thesis explores the design of a scalable distributed database system (SD-DBS) by constructing a specific SD-DBS termed SD-SQL Server. As its name indicates, SD-SQL Server uses the services of SQL Server. The original feature of our system is the dynamic and transparent repartitioning of growing tables, avoiding the cumbersome manual repartitioning characterizing the current technology. SD-SQL Server manages tables partitioned into segments distributed over SD-SQL Server nodes. It repartitions automatically any tables where inserts overflow existing segments. With the comfort of a single node SQL Server user, the SD-SQL Server user disposes of larger tables or gets a faster response time through the dynamic query parallelism. The transparency of the distribution of a scalable table is in this light an important step beyond the current technology of a parallel DBMS.

Thesis Contribution

We have validated the SD-SQL Server architecture by its detailed design, implementation and performance analysis. At the design level, the basic new feature of our system is the capability to manage the *scalable distributed* tables, scalable tables in short. Such a table consists of segments at SD-SQL Server storage nodes supporting dynamic splitting. Splits are triggered when inserts make segments to overflow. Scalable tables constitute in our system the *scalable* databases. These expand transparently for the application on many storage nodes, as needed for their tables.

We based the management of scalable tables on the scalable distributed data structures (SDDS) principles. The user or the application sees thus a scalable table only through a specific view, termed (client) *image*. In our system, it is a particular updateable distributed partitioned union view. The application manipulates a scalable table only through its image or a view of the image. A scalable table may have several images for applications at different nodes.

Every image hides the scalable table partitioning and dynamically adjusts to its evolution. Like in an SDDS, our images of the same scalable table may differ among them and from the actual partitioning. The image adjustment is lazy. It occurs only when a query referring to comes in, and finds the image outdated. Our scalable tables and databases make in this way the global database reorganization largely useless. Similarly to B-trees or extensible hash files with respect to the earlier file schemes.

Our system runs on a collection of SQL Server linked nodes. For every standard SQL command under SQL Server, there is an SD-SQL Server command for a similar action on scalable tables or views. There are also commands specific to SD-SQL Server image or node management. We have implemented all these commands and we have also managed their concurrent processing.

The scalable table management creates an overhead with respect to the static partitioning. The image processing affects the query processing time, as it may be the case of a concurrent splitting. Our design challenge was to make this overhead usually negligible. We have validated our prototype by experimental performance analysis. The performance measurements used especially the well-known *SkyServer* database, [G02]. The measures have basically confirmed the efficiency of our design choices.

The current capabilities of SQL Server allow an SD-SQL Server scalable table to reach at least 250 segments. This should suffice for scalable tables reaching very many terabytes. SD-SQL Server is the first system with the discussed capabilities, to the best of our knowledge. Our results pave the way towards the use of the scalable tables as the basic DBMS technology.

Future work should lift some current limitations of our "proof-of-concept" type prototype and expand its functional and processing capabilities. For instance, one should enhance the current query error processing. Our experimental performance analysis, applied to the *SkyServer* database as benchmark, should be expanded towards other well-known benchmarks. Our implementation on SQL Server 2000 should now be ported to SQL 2005 recently released. Our design should be also ported towards other DBSs. With respect to the functional capabilities, we currently expand our interface so to enable SD-SQL Server to be a server of *GovML* documents. It should act as core component of a virtual repository of *eGov* documents in a prototype system under development [LMS06].

Publications

We have published four papers about our work in international conferences [LS04, LSS06a, SLS06, LSS06b].

In [LS04], we report on the gross architecture and performance results of our first SD-SQL Server prototype. The architecture extends and puts into practice the one of

[LRS02]. The performance measures essentially confirm the efficiency of splitting and of image adjustment operations.

In [LSS06a], we detail the internal processing of the SD-SQL Server commands. We discuss in particular our concurrent command processing, using basically the repeatable read isolation level. We prove the serializability of our approach.

In [SLS06], we expand the SD-SQL Server architecture with new concepts, especially that of a scalable database. We present the user/application interface and illustrate its commands with examples.

Finally, in [LSS06b], we give an overview of SD-SQL Server. We essentially report performance results related to the new improved architecture. We also discuss the related works.

We have also presented our proposed prototype, SD-SQL Server, at the *Microsoft Research Academic Days 2006*.

For convenience, we have attach our two latest publications, [SLS06] and [LSS06b] at the end of the Thesis. Our extensive *Research Report*, [SLS05], is available on-line at: *http://ceria.dauphine.fr*

Thesis Outline

The Thesis dissertation is divided into eight chapters:

The first Chapter of the Thesis, "*Introduction*", discusses the basic issues and the motivation for the thesis work, as well as our contribution.

In Chapter 2, "*Introduction aux SGBDs Parallèles*", we cover the basic features of data partitioning in the parallel database systems. We also present the partitioning schemes in the principle current DBMSs.

Chapter 3, "*Structures de Données Distribuées et Scalables*" covers with special interest the SDDSs, being one of our thesis fundamentals. We mainly present the RP* SDDS on which is the basis of our work.

Chapter 4, "*Architecture de SD-SQL Server*", describes the SD-SQL Server architecture with all its components. It presents the originality of this system among all current DBMSs. It also discusses our choice of SQL Server to conceive our SD-DBS.

Chapter 5, "*Interface de SD-SQL Server*", describes the syntax and semantics of each command of the SD-SQL Server interface. Numerous examples illustrate the actual use of SD-SQL Server.

Chapter 6, "*Le Prototype SD-SQL Server*", covers the SD-SQL Server implementation. It presents all the steps to build SD-SQL Server. It also describes its internal processing.

Chapter 7, "*Mesures de Performances de SD-SQL Server*", discusses the numerous experiments we made to prove the efficiency of our prototype. We used for this the *SkyServer* DB benchmark. The scalable table processing creates an overhead. The performance analysis proved this overhead negligible for practical purpose.

Finally, the last chapter "*Conclusion & Perspectives*" concludes the thesis dissertation and gives some future research directions.

The Thesis contains Appendix A, "*Traitements Internes des Commandes SD-SQL Server*", that presents the internal processing of the SD-SQL Server commands. After this appendix, the glossary of our terminology follows. At the end, we attach two of our published papers ([SLS06] and [LSS06b]).

Table des Matières

LISTE DES FIGURES

LISTE DES TABLEAUX

1 INTRODUCTION

1.1 Motivations

Lors de cette dernière décennie, la croissance rapide du volume de données à stocker et à traiter a rendu les bases de données de plus en plus volumineuses et difficiles à gérer. Toutefois, les progrès technologiques impressionnants réalisés ne sauraient faire face aux traitements de tels volumes de données. En effet, les performances des microprocesseurs augmentent d'environ 50% par an et la capacité mémoire par un facteur de 16 tous les six ans. Ce qui est loin d'être le cas des mémoires secondaires dont le temps de réponse et le débit n'ont augmenté que d'un facteur de deux durant les dix dernières années [V93].

La solution qui s'imposait était alors de distribuer les données, organisées dans des tables, sur différents sites de stockage. L'ensemble de ces sites constitue les systèmes de bases de données distribuées. Ces systèmes permettent aux utilisateurs de manipuler des données sur plusieurs bases distribuées dans un réseau de manière transparente, comme une base de données globale.

Le problème essentiel auquel ont été confrontés les systèmes de bases de données distribuées est le problème d'entrées/sorties qui constituent un véritable goulot d'étranglement. En effet, le temps d'accès disque est environ cent mille fois plus lent que le temps d'accès mémoire. Les solutions matérielles à ce problème ont vite été abandonnées à cause du rapport prix/performance qu'elles induisent, et la recherche s'est vite orientée vers l'utilisation du parallélisme afin d'augmenter la bande passante des entrées/sorties. Ainsi, les systèmes de bases de données parallèles ont apparus. En effet, si on stocke une base de données de taille D sur un seul disque de débit T, le débit du système sera borné par T. Cependant, si on partitionne la base de données sur n disques, chacun de capacité D/n et de débit T', on aura un débit de $n*T'$ qui pourra être mieux exploité par plusieurs processeurs.

L'obstacle majeur à l'obtention de bonnes performances dans les systèmes de bases de données parallèles réside dans l'équilibrage de la charge entre les différents processeurs. Ceci est dû au critère de partitionnement de données. Les SGBDs commercialisés actuels (Microsoft SQL Server, Oracle, DB2…) fournissent uniquement un partitionnement statique des données [BM00, L03, LB05]. Si une table monte en échelle, l'administrateur de base de données doit redéfinir manuellement la partition et exécute des utilitaires de

redistribution de données. Ceci est devenu une préoccupation croissante pour les utilisateurs des bases de données parallèles [SL96, RZLM02, GG05].

Ce problème est similaire à celui rencontré, il y a 40 ans, pour les systèmes de gestion de fichiers dans l'environnement centralisé. La méthode d'accès séquentiel indexée (ISAM) était utilisée pour le partitionnement ordonné des fichiers [IBM87]. De même, les méthodes d'accès basées sur le hachage statique ont été connues pour le partitionnement des fichiers. Ces deux approches de partitionnement demandent la réorganisation des fichiers lorsque les insertions provoquent des dépassements de la capacité des fichiers. Suite à cela, les arbres-B et les méthodes de hachage dynamique (extensible et linéaire) ont été introduits pour pallier aux problèmes de partitionnement statique. Ils remplacent la réorganisation des fichiers par des éclatements incrémentaux de chaque fichier [BM72, L80, FNPS79].

La recherche dans le domaine de la gestion des données distribuées a montré les mêmes besoins particulièrement avec l'apparition des multi-ordinateurs qui représentent des réseaux de stations de travail ou ce qui est connu aujourd'hui sous le nom de *Peer-to-Peer* [AD01, SMKB01] ou *Grid Computing* [F01, A01]. Leur but est la conception de systèmes distribués. La recherche sur les multi-ordinateurs a proposé l'utilisation de nouvelles structures de données pour fournir, à haut débit, un bon temps de réponse, sur les gros volumes de données gérées par ces systèmes. Il s'agit des structures de données distribuées et scalables (SDDSs) [WBW94, KW94, LNS94, KLR94]. Les données d'une SDDS sont réparties dans la mémoire distribuée d'un multi-ordinateur. Les fichiers SDDS s'adaptent dynamiquement à l'accroissement du volume des données. Ils montent en échelle d'une manière transparente à travers des éclatements distribués, d'où l'appellation de *fichiers scalables*.

Les SDDSs supportent le traitement parallèle et assurent des temps d'accès aux données beaucoup plus rapides que ceux aux fichiers traditionnels sur disques. Elles permettent le partitionnement dynamique ainsi que la scalabilité dans les multi-ordinateurs. La *scalabilité* (ang. *scalability*) ou la montée en échelle est donc la capacité d'une application de maintenir le même niveau de performances, lorsque la charge augmente et la configuration matérielle évolue par l'ajout de processeurs et/ou de capacités de stockage [G93, G99].

Le problème du partitionnement dynamique et de la scalabilité a été largement étudié pour les systèmes de gestion de fichiers [LS90]. Il n'en est pas de même pour les SGBDs. Dans un système distribué à grande échelle, les bases de données sont nombreuses. La gestion de données dans un tel contexte pose des problèmes de gestion difficiles car les techniques doivent passer à l'échelle tout en supportant les nouveaux besoins liés à la distribution des données. Ces problèmes constituent les axes principaux de notre recherche.

1.2 Contribution de la Thèse

Dans cette thèse, nous nous intéressons au problème du partitionnement dynamique et de la *scalabilité* (ou montée en échelle) des bases de données et des tables ainsi que leur accès. Nous avons donc proposé, mis en œuvre et évalué un système de gestion de bases de données distribuées et scalables. Nous avons étendu l'architecture de bases de données parallèles dans le but d'obtenir une nouvelle architecture de SGBD capable de gérer des bases de données et des tables distribuées et scalables. Nous avons appelé ce système SD-SQL Server.

Nous avons conçu SD-SQL Server à partir du SGBD SQL Server. Les nœuds de stockage SD-SQL Server représentent les instances du SGBD SQL Server liées entre elles, qui sont connues sous le nom de *serveurs liés*. L'architecture du système SD-SQL Server supporte des bases de données et des tables distribuées et scalables. Une base du système SD-SQL Server, dite *scalable*, grandit par la partition dynamique de ses tables, dites scalables aussi. Une *table scalable* est organisée en segments. Chaque segment est placé sur un nœud de stockage SD-SQL Server. Nous avons utilisé les principes des SDDSs pour gérer les tables scalables. Une table scalable est un ensemble de segments d'une structure de données distribuées et scalable (SDDS).

SD-SQL Server permet à une table scalable de croître sur ses différents nœuds distribués. Cette expansion résulte des éclatements de segments débordant leur capacité de stockage. Chaque éclatement est déclenché par l'insertion de tuples, qui est à l'origine du débordement, et crée un ou plusieurs nouveaux segments. Ces nouveaux segments ont le même schéma que les segments éclatés. Ce qui diffère entre eux est leur plage de valeurs définies par les contraintes d'intégrité au niveau de chaque segment d'une table scalable. Les contraintes d'intégrité définissent alors le partitionnement par intervalle de chaque segment d'une table scalable.

Tout comme les SDDSs, les segments d'une table scalable sont cachés des usagers/applications derrière des images représentées par des vues supportant les mises à jour. Les images sont les vues partitionnées et distribuées de SQL Server dynamiquement ajustées pour toute évolution des partitions. Elles rendent transparent le partitionnement des tables scalables et ceci en se basant sur les contraintes d'intégrité de leurs segments. Les usagers/applications n'interfacent que les images ou leur vues. SQL Server est le seul SGBD qui permet la mise à jour des vues partitionnées, d'où notre utilisation de ce SGBD.

Les usagers/applications manipulent les objets du système SD-SQL Server en utilisant des commandes dédiées à SD-SQL Server. Ces commandes permettent de gérer les nœuds de stockage de SD-SQL Server, ses bases de données scalables, ses tables scalables ainsi que leurs images. Les commandes sur les tables scalables et leurs images (vues) permettent les

mêmes manipulations que les schémas et les requêtes SQL usuels. Cependant, ces commandes sont personnalisées afin de traiter la scalabilité et le partitionnement des tables sur SD-SQL Server. De plus, elles peuvent aussi être exécutées sur des tables non scalables. Nous avons également pris en compte la concurrence entre les différentes commandes. Afin de préserver la sérialisabilité de l'exécution de ces commandes, nous avons proposé des schémas de concurrence qui évitent les conflits entre les différentes commandes s'exécutant en parallèle.

La gestion des tables scalables par SD-SQL Server permet de créer un *overhead*. Minimiser cet *overhead* (ou surcoût) était notre principal objectif. L'analyse des performances du prototype, en utilisant la base de test (ang. *benchmark*) *SkyServer* [G02], montre que le surcoût est négligeable. Les mesures de performance prouvent que la gestion de tables scalables peut offrir une amélioration de 25% du temps de réponse des requêtes sur des tables scalables.

1.3 Plan de la Thèse

Cette thèse est organisée en huit chapitres. Dans le chapitre 2, Nous présentons en détail l'ensemble des notions nécessaires à la compréhension de cette thèse. Nous passons en revue les concepts des SGBDs parallèles. Nous décrivons leurs architectures matérielles, les schémas et stratégies de partitionnement qu'ils adoptent. Ensuite, nous donnons quelques exemples de SGBDs parallèles commercialisés ou issus de la recherche. Nous insistons sur le problème de partitionnement dynamique sur les SGBDs existants, objet d'étude de cette thèse, en présentant un bilan à la fin de ce chapitre.

Le chapitre 3 est consacré à la présentation des structures de données distribuées et scalables, une nouvelle classe d'organisation de données, définie spécifiquement pour les multi-ordinateurs. Nous présentons en premier lieu les architectures des systèmes répartis sur lesquelles peuvent reposer de telles structures. Ensuite, nous exposons les principes de base de ces structures. Nous insistons en particulier sur la famille des SDDSs RP* sur laquelle notre travail repose.

Le chapitre 4 présente le système de gestion de bases de données distribuées et scalables SD-SQL Server que nous proposons. Nous définissons tout d'abord les objectifs de ce système et les règles structurelles nécessaires à sa conception. Ensuite, nous décrivons en détail l'architecture de SD-SQL avec tous ses composants.

Le chapitre 5 présente l'interface d'application dédiée au système SD-SQL Server. Nous détaillons toutes les commandes du système SD-SQL Server qui comprennent : la gestion des nœuds, des bases scalables, des tables scalables, des images ainsi que la gestion des requêtes.

Le chapitre 6 décrit le prototype SD-SQL Server. Nous commençons par présenter nos choix techniques pour la mise en œuvre de SD-SQL Server. Ensuite, nous détaillons les traitements internes de chaque commande SD-SQL Server.

Le chapitre 7 présente l'environnement expérimental et les mesures de performances.

Le chapitre 8 conclut ce document et présente les perspectives et les axes de recherche qui pourront être poursuivis.

2 INTRODUCTION AUX SYSTEMES DE GESTION DE BASES DE DONNEES PARALLELES

2.1 Introduction

Un système de gestion de bases de données (SGBD) est aujourd'hui un logiciel de base essentiel dans un système informatique de gestion. De façon intuitive, il permet à des utilisateurs concurrents de manipuler, rechercher, modifier, insérer efficacement des données contenues dans une base de données. Historiquement, les SGBDs ont connu plusieurs organisations selon la technique employée pour leur implantation physique. Ainsi, on est passé des SGBDs centralisés aux SGBDs parallèles en transitant par les SGBDs distribués [GV91, Gar93]. L'approche centralisée permet à un ensemble d'applications distribuées d'accéder de façon efficace à un serveur de base de données unique. Toutefois, cette approche souffre des limitations traditionnelles des systèmes centralisés. Les développements remarquables faits dans des domaines aussi divers que les réseaux de communication, les mini et micro-ordinateurs ont permis à l'approche bases de données réparties de devenir une solution alternative à la centralisation [V93]. Outre ce besoin de décentralisation, d'autres problèmes étaient posés par la suite. Il s'agit en particulier des volumes de données qui s'accroissent de jour en jour ainsi que des requêtes complexes. La solution qui s'imposait était alors de combiner la gestion de bases de données et le traitement parallèle afin d'augmenter la performance et la disponibilité des données [OV99]. Et c'est ainsi que sont apparus les SGBDs parallèles.

Ce chapitre passe en revue les principaux concepts des systèmes de bases de données parallèles. Dans la première section, nous présentons les architectures matérielles utilisées par les SGBDs parallèles. Ensuite, nous nous intéressons aux stratégies et schémas de

partitionnement des données dans la deuxième section. Enfin, nous présentons les concepts utilisés dans quelques SGBDs parallèles connus.

2.2 Architectures Matérielles

Le choix d'une architecture destinée à supporter un SGBD parallèle est guidé par le souci d'atteindre tous les objectifs de traitements parallèles des transactions et des opérations tout en garantissant le meilleur rapport prix/performances. Trois architectures ont, jusqu'à présent, été utilisées comme support pour les SGBDs parallèles. Nous décrivons dans ce qui suit ces trois architectures et les architectures hybrides qui tentent de combiner les avantages de chaque architecture, telles qu'elles sont décrites dans la littérature [DG92, NZT96, OV99].

2.2.1 Architecture à Mémoires Partagées

Dans l'architecture à mémoires partagées (ang. *shared memory*), la mémoire est accessible et partagée par l'ensemble des processeurs de la machine comme illustré dans la Figure 2-1. Chaque processeur a un coût uniforme d'accès à la mémoire. Un bus relie entre eux les éléments matériels de la machine : processeurs, mémoires et disques. Cette architecture permet un accès rapide aux données. De plus, comme les informations de contrôle et les informations globales concernant les données sont accessibles par tous les processeurs, les principes de conception ne sont pas très différents de ceux d'un SGBD centralisé. Par contre, la complexité du réseau d'interconnexion due à la nécessité de relier chaque processeur à tous les modules augmente le coût d'une telle architecture et limite son nombre de processeurs.

Parmi les systèmes de gestion de bases de données qui utilisent ce type d'architecture, nous citons : XPRS, DBS3 [BCV91] et Volcano [G90]. Pour les systèmes commerciaux, Oracle [D92] et Informix [Da92] proposent des solutions sur ce type d'architecture.

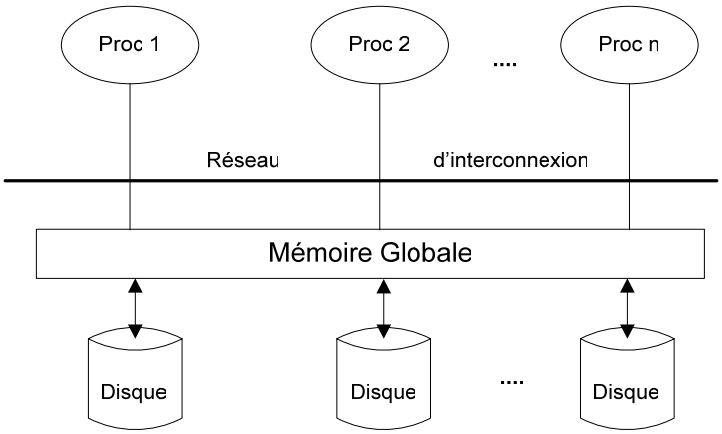

Figure 2-1 : Architecture à mémoires partagées

2.2.2 Architecture à Disques Partagés

L'architecture à disques partagés (ang. *shared disk*), illustrée dans la Figure 2-2, est basée sur l'hypothèse que chaque processeur a sa mémoire centrale privée, mais les disques sont partagés par tous les processeurs. Ceci facilite la répartition de la charge de travail tandis que la mémoire reste distribuée pour ne pas pénaliser l'extensibilité [OV99]. Le disque partagé offre donc une excellente utilisation des ressources. De plus, il fournit un bon niveau de tolérance aux pannes avec l'ensemble des données qui restent accessibles tant qu'il y a au moins un nœud qui fonctionne. Cependant, comme la mémoire n'est pas partagée, une page disque peut être dupliquée sur plusieurs nœuds de la machine. Ceci nécessite la gestion de cohérence entre toutes les copies d'une page. Le surcoût de la maintenance de cette cohérence est bien sûr l'inconvénient majeur de cette architecture.

Parmi les systèmes de gestion de bases de données s'exécutant sur des machines à disques partagés, nous citons principalement Oracle (*Oracle Parallel Server*) [D92].

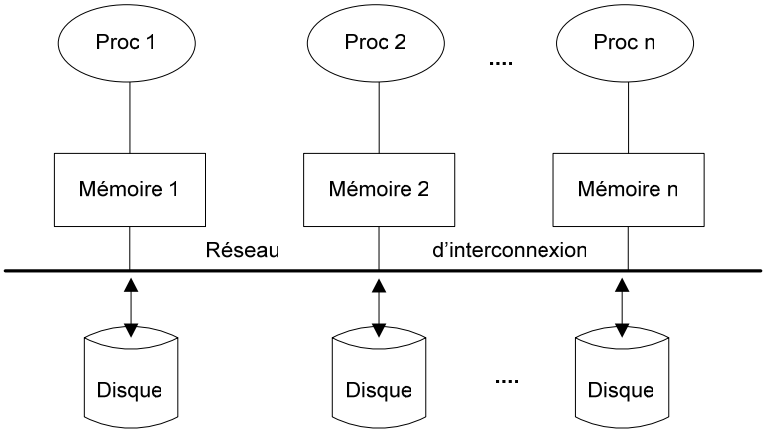

Figure 2-2 : Architecture à disques partagés

2.2.3 Architecture à Mémoires Distribuées

Dans cette architecture, dite aussi à partage de rien (ang. *shared nothing*), excepté le réseau d'interconnexion, rien n'est partagé entre les processeurs. Chaque processeur a sa propre mémoire centrale et son propre disque comme le montre la Figure 2-3. Le partage de ressources matérielles et logicielles entre les différents processeurs est donc limité au minimum. Cette architecture est ainsi facilement extensible. De plus, une bonne disponibilité des données est assurée grâce à leur réplication au niveau de plusieurs nœuds. Cependant, pour équilibrer la charge de travail entre les processeurs, la redistribution des données est très complexe.

De nombreux systèmes de gestion de bases de données utilisent ce type d'architecture notamment NonStopSQL de Tandem [BACC+90], Gamma [DG86] et PRISMA/DB [AVFG+92].

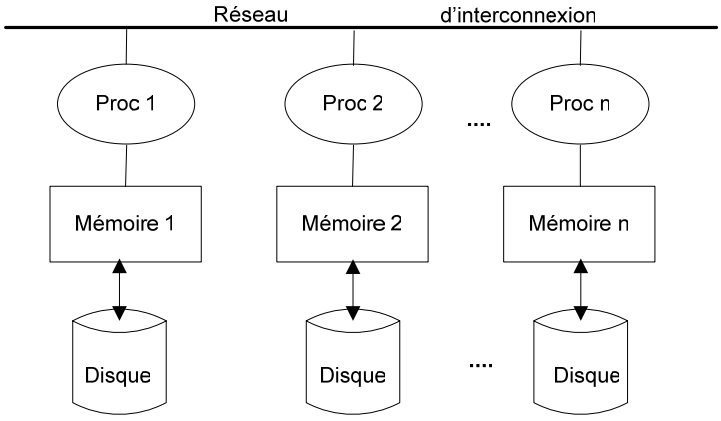

Figure 2-3 : Architecture à mémoires distribuées

2.2.4 Architecture Hybride

Une architecture hybride est une combinaison des architectures à mémoires distribuées et à mémoires partagées. Une telle architecture combine les avantages de chaque architecture, et compensent leurs inconvénients respectifs. L'architecture hybride illustrée dans la Figure 2-4 combine l'équilibre de charge des architectures à mémoire(s) partagée(s) et l'extensibilité des architectures à mémoire(s) distribuée(s).

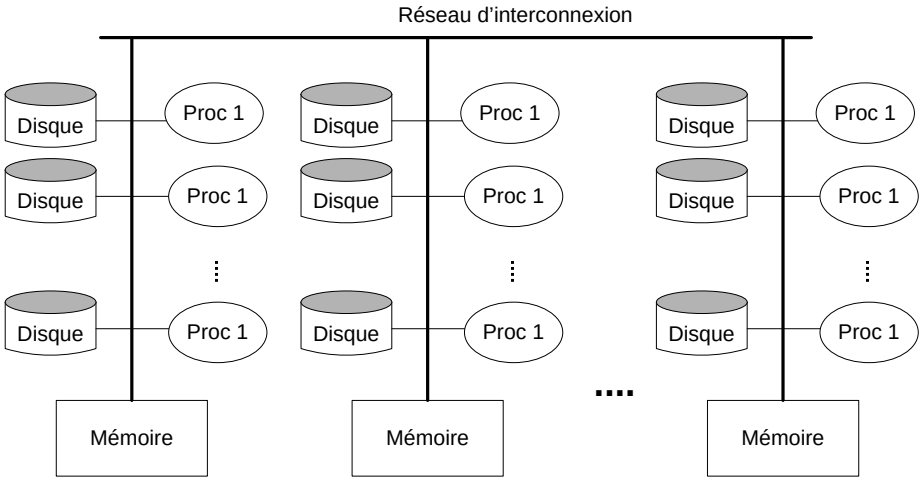

Figure 2-4 : Architecture Hybride

2.3 Partitionnement des Données

Après avoir présenté les différentes architectures matérielles des systèmes de bases de données parallèles, nous présentons, dans cette section, le partitionnement de leurs données. Le partitionnement des données, appelé aussi fragmentation, est l'un des facteurs critiques qui font d'un SGBD un système performant ou non. La technique de placement des données utilisée permet de décider de la qualité de la charge assurée par le système. Dans cette section, nous décrivons les différents schémas de partitionnement ainsi que les stratégies de partitionnement de données.

2.3.1 Schémas de Partitionnement

Le *partitionnement* d'une table permet à des transactions portant sur des tuples différents de cette table de s'exécuter de façon concurrente. Ainsi, les applications n'accèdent qu'à des sous ensembles de tables et non a des tables entières. Il existe deux schémas fondamentaux du partitionnement : le partitionnement horizontal et le partitionnement vertical [OV99].

2.3.1.1 Partitionnement Horizontal

Le partitionnement horizontal fragmente une table selon ses tuples. Chaque fragment a alors un sous ensemble de tuples de la table. Il existe deux manières de partitionner une table horizontalement : le *partitionnement primaire* et le *partitionnement dérivé*. Le premier partitionnement est effectué en utilisant des prédicats définis sur la table elle

même. Quant au deuxième partitionnement, il est effectué en utilisant des prédicats définis sur une autre table.

2.3.1.2 Partitionnement Vertical

Le partitionnement vertical d'une table *T* produit un ensemble de fragments *(T1,..., Tn)* contenant chacun un sous ensemble des attributs de *T* en plus de la clé primaire de la table *T*. Le partitionnement vertical est beaucoup plus complexe que le partitionnement horizontal. Ceci est essentiellement dû au grand nombre d'alternatives possibles.

2.3.1.3 Partitionnement Hybride

Dans la plupart des cas, un partitionnement horizontal ou vertical simple d'une base de données ne satisfait pas les demandes des applications. Dans ce cas, un partitionnement horizontal peut être suivi par un partitionnement vertical ou vice versa.

2.3.2 Stratégies de Partitionnement de Données

Après avoir présenté les schémas de partitionnement des données dans un SGBD parallèle, nous présentons dans ce qui suit les différentes méthodes ou stratégies suivies pour obtenir ces schémas. Plusieurs stratégies de partitionnement ont été proposées : le *partitionnement circulaire* [T88], le *partitionnement par hachage* [KTM88] et le *partitionnement par intervalle* [DGG+86].

2.3.2.1 Partitionnement Circulaire

Le partitionnement circulaire (ang. *Round Robin*) est implanté dans les systèmes RAID (*Redundant Arrays of Independant Disks*) [PGK88]. Il est considéré comme la stratégie la plus simple pour le partitionnement des données. Cette stratégie garantit une distribution uniforme des données. Elle permet de partitionner une table en fragments de même taille entre des nœuds distribués. Si *N* est le nombre de partitions d'une table, alors le *i*-ème tuple de cette table est affecté à la partition de numéro (*i mod N*). Les tuples sont placés en fonction de leur numéro d'ordre. La première donnée sera placée sur le premier processeur. La seconde sur le deuxième et ainsi de suite jusqu'à placer la nouvelle donnée sur le dernier processeur. La Figure 2-5 illustre la stratégie du partitionnement circulaire.

L'avantage majeur du partitionnement circulaire est l'optimisation du temps de réponse des requêtes tout en permettant un accès séquentiel et parallèle aux données. Cependant, cet accès nécessite le parcours de toutes les partitions d'une table, ce qui entraîne inévitablement une dégradation des performances. De plus, le problème avec cette

technique est que les applications désirent souvent accéder aux tuples de façon associative, ce qui n'est pas possible avec cette technique.

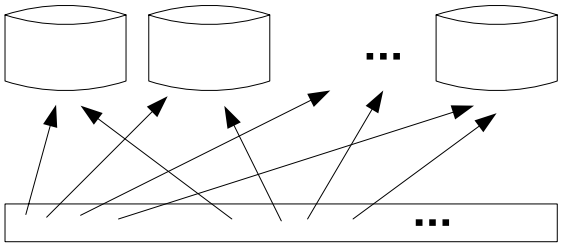

Figure 2-5 : Partitionnement circulaire

2.3.2.2 Partitionnement par Hachage

La stratégie de partitionnement par hachage (ang. *Hashage Partitioning*), illustrée dans la Figure 2-6, distribue l'ensemble des tuples en utilisant une fonction de hachage *h* sur un ensemble d'attributs. La fonction de hachage retourne le numéro du serveur dans lequel le tuple sera rangé. Elle convient aux applications qui veulent accéder aux données de façon séquentielle et associative. L'accès associatif aux tuples ayant une valeur d'attribut spécifique peut être dirigé vers un seul disque, évitant ainsi le surcoût dû à un lancement de requêtes sur plusieurs disques.

Il existe deux méthodes de hachage : *hachage statique* et *hachage dynamique*. Tant qu'il n'y a pas de débordements, le hachage statique est très simple et donne d'excellentes performances. Cependant, les débordements dégradent rapidement les performances. Le hachage dynamique permet de faire grandir progressivement un fichier haché saturé en distribuant les tuples dans de nouvelles régions allouées à une table. Deux méthodes du hachage dynamique s'avèrent importantes : le *hachage extensible* et le *hachage linéaire*.

- **Hachage Extensible**

Le hachage extensible [FNPS79] est une méthode de hachage dynamique qui associe à chaque fichier un répertoire des adresses de paquets. Au départ, *M* bits de la fonction de hachage sont utilisés pour adresser le répertoire. A la première saturation d'un paquet, le répertoire est doublé, et un nouveau paquet est alloué au fichier. Le paquet saturé est distribué entre l'ancien et le nouveau paquet, selon le bit suivant *(M+1)* de la fonction de hachage. Ensuite, tout paquet plein est éclaté en deux paquets, lui-même est un nouveau paquet alloué au fichier. L'entrée du répertoire correspondant au nouveau paquet est mise

à jour avec l'adresse de ce nouveau paquet si elle pointait encore sur le paquet plein. Sinon, le répertoire est à nouveau doublé.

Le hachage extensible consiste alors à éclater un paquet plein et à mémoriser l'adresse des paquets dans un répertoire adressé par les *(M+P)* premiers bits de la fonction de hachage, où *P* est le nombre d'éclatements maximal subi par les paquets. Le hachage extensible est plus robuste face aux mauvaises distributions de clés que le hachage linéaire comme nous le présenterons dans la section suivante. Par contre, la gestion d'un répertoire est plus lourde que celle d'un pointeur courant.

↻ Hachage Linéaire

Le hachage linéaire [L80] est une méthode de hachage dynamique nécessitant la gestion du débordement et consistant à :

- éclater le paquet pointé par un pointeur courant quand un paquet est plein ;
- mémoriser le niveau d'éclatement du fichier afin de déterminer le nombre de bits de la fonction de hachage à appliquer avant et après le pointeur courant.

Le hachage linéaire peut aussi s'implémenter avec un répertoire. Dans ce cas, le pointeur courant est un pointeur sur le répertoire : il référencie l'adresse de paquet suivant à éclater.

L'avantage du hachage linéaire est alors la simplicité de l'algorithme d'adressage du répertoire : on utilise *(M+P)* bits de la fonction de hachage. Si on est positionné avant le pointeur courant, on utilise un bit de plus, sinon on lit l'adresse du paquet dans le répertoire. A chaque éclatement, le répertoire s'accroît d'une seule entrée.

Le hachage linéaire est utilisé par plusieurs SGBDs. Notamment SQL Server, Postgres, DB Library (Oracle), et MySQL. Les manuels, en ligne de chaque système, détaillent l'usage de l'algorithme.

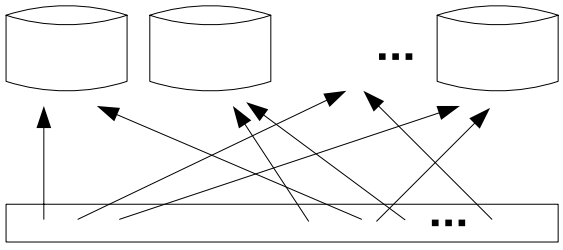

Figure 2-6 : Partitionnement par hachage

2.3.2.3 Partitionnement par Intervalle

La stratégie de distribution par intervalle (ang. *Range Partitioning*), illustrée dans la Figure 2-7, distribue les tuples d'une table en fonction de la valeur d'un ou de plusieurs attributs, formant alors une valeur unique dite *clé*, par rapport à un ordre total de l'espace des clés. Les fonctions de hachage en général n'offrent pas ce type de partition. Les attributs formant la clé sont appelés alors aussi attributs ou *clés de partitionnement*. La table ou le fichier partagé sont dits *ordonnés*. La méthode générale du partitionnement par intervalle consiste en deux phases. La première phase permet de diviser l'espace des valeurs des attributs sélectionnés en intervalles puis chaque intervalle est affecté à un nœud. Quant à la deuxième phase, elle assigne chaque tuple *t* au nœud *n* si les valeurs d'attributs spécifiés du tuple *t* appartiennent à l'intervalle de *n*.

La stratégie de partitionnement par intervalle convient aux requêtes dont le prédicat implique les attributs de partitionnement et donc les requêtes par intervalle. Cependant, elle ne garantit pas un équilibre de charge entre les nœuds.

Dans ce qui suit, nous décrivons une des méthodes de partitionnement par intervalle les plus utilisées, qui est la méthode des arbres-B [BM72].

- **Arbres-B**

Les arbres-B ont été introduits afin de corriger le principal défaut des méthodes de partitionnement ordonné connu avant sous le nom *organisation séquentielle indexée*, d'IBM (ang. *Indexed Sequential Access Method, ISAM*) [IBM78]. Cette méthode générait une partition statique. Un fichier grandissant nécessitait périodiquement une réorganisation globale. Cette contrainte posait des problèmes évidents aux applications et usagers. Les arbres-B ainsi que l'organisation largement similaire introduite par IBM à l'époque dite *organisation séquentielle indexée régulière* (ang. *Virtual Sequential Access Method*, *VSAM*) [IBM78], évitaient cet avatar. L'idée bien connue depuis était celle de l'éclatement de toute page (case) du fichier organisé en arbre m-aire, où une insertion créait un débordement.

En quelques mots, puisque les arbres-B sont très connus, les clés d'un arbre-B sont triées selon l'ordre post-fixé induit par l'arbre, afin de permettre les recherches en un nombre d'accès n'excédant pas le niveau de l'arbre-B. L'utilisation des arbres-B « originaux » pour réaliser des tables indexées conduit néanmoins à un taux d'accès aux disque durant le parcours des noeuds internes qui peut être aisément amélioré. Les arbres dit B+ atteignent cet objectif, en ne mettant dans les nœuds internes que les clés et les pointeurs. C'est cette version qui est actuellement la plus utilisée.

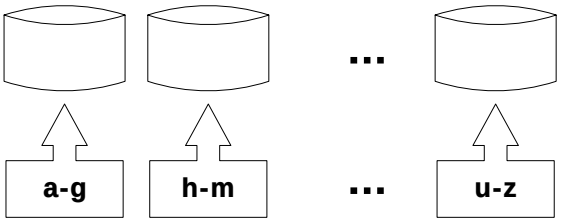

Figure 2-7 : Partitionnement par intervalle

2.3.3 Traitement Parallèle des Données

Pour atteindre de hautes performances, un SGBD exploite le parallélisme pour les requêtes exprimées dans un langage d'assertions [K95, ACP+99, ÖV99]. Le traitement parallèle est utilisé afin de maximiser le débit d'un système distribué en réduisant le temps total des requêtes d'une part, et de minimiser le temps de réponse des requêtes d'autre part. Le partitionnement de données fournit des opportunités de traitement parallèle. Plus précisément, deux formes de parallélisme sont obtenues directement à partir du partitionnement des données. Le *parallélisme inter-requête* permet l'exécution parallèle de plusieurs requêtes, chacune agissant sur des partitions différentes. Le *parallélisme intra-requête* permet l'exécution parallèle de plusieurs opérations relationnelles à l'intérieur de la requête même. Une même opération peut être traitée par plusieurs processeurs. Il s'agit du *parallélisme intra-opération*.

Il existe trois mécanismes de base pour réaliser le parallélisme des traitements : le *parallélisme indépendant*, le *parallélisme en tuyau* et le *parallélisme par fragmentation*.

2.3.3.1 Parallélisme Indépendant

Le parallélisme indépendant (ang. *Independent Parallelism*), illustré dans la Figure 2-8, permet d'exécuter en parallèle plusieurs opérations indépendantes d'une même tâche ou requête.

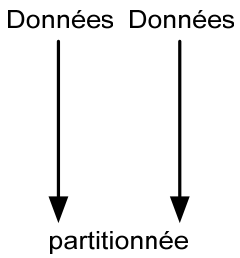

Figure 2-8 : Parallélisme indépendant

2.3.3.2 Parallélisme en Tuyau

Le parallélisme en tuyau (ang. *Pipelined Parallelism*), illustré dans la Figure 2-9, est utilisé dans le cas où le résultat d'une opération constitue les données d'entrée pour l'opération suivante.

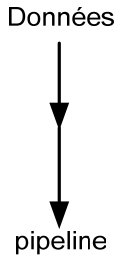

Figure 2-9 : Parallélisme en tuyau

2.3.3.3 Parallélisme par Fragmentation

Le parallélisme par fragmentation (ang. *Partitioned Parallelism*), illustré da la Figure 2-10, permet de fragmenter la charge de travail entre plusieurs processus. Cela nécessite à la fin une fusion des résultats partiels pour produire le résultat final.

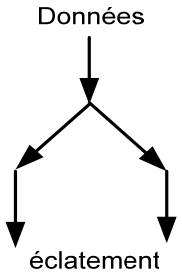

Figure 2-10 : Parallélisme par fragmentation

2.4 Administration de Partitions des Principaux SGBDs

Dans ce qui suit, nous présentons les principaux concepts de quelques prototypes de SGBDs parallèles issus de la recherche et commercialisés. Nous décrivons en particulier leur partitionnement des données auquel nous nous intéressons dans notre travail.

2.4.1 Prototypes de SGBDs Parallèles issus de la Recherche

2.4.1.1 DBC/1012 de Teradata

Teradata est un SGBD développé pour la gestion des entrpôts des données [T85]. Il a été construit pour fonctionner sur plusieurs serveurs. Les serveurs *Teradata* travaillent comme serveurs SQL pour des programmes clients situés sur des ordinateurs conventionnels. Les systèmes *Teradata* peuvent inclure plus d'un millier de processeurs et plusieurs milliers de disques. Les processeurs *Teradata* sont fonctionnellement divisés en deux groupes : les processeurs interfaces (IFPs) et les processeurs d'accès (APs) [T88].

Les IFPs gèrent la communication avec l'hôte, la compilation, l'exécution des requêtes et la coordination des APs durant l'exécution des requêtes. Chaque AP dispose de plusieurs disques et d'une grande mémoire cache. Chaque table est partitionnée sur un ensemble d'APs par adressage dispersé. Lorsqu'un tuple est inséré dans une relation, une fonction de hachage est appliquée à la clé primaire du tuple afin de sélectionner l'AP sur lequel il va être stocké. Lorsqu'un tuple arrive à un AP, une autre fonction de hachage lui est appliquée afin de déterminer sa position dans son fragment. Les tuples de chaque fragment sont ordonnés.

Teradata a installé plusieurs systèmes contenant plus de cent processeurs et des centaines de disques. Ces systèmes ont démontré une accélération (ang. *speed-up*) et une possibilité d'extension (ang. *scale-up*) presque linéaires, dépassant de loin les vitesses des machines dédiées traditionnelles dans le traitement de grandes bases de données (Terabytes).

2.4.1.2 Gamma

Gamma, développé à l'université du Wisconsin, est l'un des systèmes pionniers pour les architectures parallèles à mémoire distribuée. Il en est aussi l'un des plus représentatifs. La version actuelle de *Gamma* est implantée sur un hyper-cube de 32 nœuds iPSC/2 avec un disque attaché à chaque nœud [DG86]. Dans *Gamma*, toutes les relations sont fragmentées horizontalement sur l'ensemble des noeuds du système, permettant ainsi de profiter pour chaque relation de toute la bande passante I/O du système matériel. Trois méthodes de fragmentation des données sont disponibles : *Round-Robin,* par hachage et par intervalle de valeurs [DGG+86].

Le modèle d'exécution des requêtes introduit par *Gamma* est le modèle *bracket.* C'est un modèle générique de processus utilisé pour chaque type d'opérateurs (sélection, jointure...). Chaque processus peut recevoir et envoyer des données pour l'opérateur qu'il contient et ne peut exécuter plus d'un opérateur à la fois.

L'entrée et la sortie d'un processus opérateur sont un flot de tuples. Le flot en sortie est dé-multiplexé grâce à une structure particulière appelée *split table.* Cette structure (ou

table) définit une correspondance entre un ensemble de valeurs et un ensemble de processus destination. Lorsqu'un processus produit un tuple, il utilise cette table comme un index afin d'obtenir l'adresse (numéro) du processus destination à même de consommer le tuple produit. Trois types de *split table* sont utilisés suivant que les processus destination utilisent une relation stockée en *Round-Robin*, par hachage ou par intervalle de valeurs.

2.4.2 SGBDs Commercialisés

2.4.2.1 Microsoft SQL Server

SQL Server repose sur une architecture à mémoires distribuées. Le concept de partitionnement sur SQL Server n'est pas nouveau. Chaque version du SGBD (version 6, 7, 2000 et 2005) autorise des formes de partitionnement. Microsoft SQL Server prend en charge le partitionnement des données par l'intermédiaire de vues partitionnées dans SQL Server 7.0/2000. Dans SQL Server 2000, la fonctionnalité a été améliorée afin de prendre en charge les vues partitionnées pouvant être mises à jour. Une vue partitionnée est particulièrement adaptée lorsqu'une table peut être naturellement partitionnée ou divisée en tables distinctes par plages de données. Les tables sous-jacentes de la vue partitionnée font l'objet d'une *union* afin de présenter un ensemble de données unifié. Les vues partitionnées réduisent considérablement la complexité des applications, car l'implémentation physique est déduite des méthodes d'accès aux données par l'application.

Dans SQL Server 2000, les vues partitionnées peuvent être étendues afin d'inclure des vues partitionnées distribuées, ce qui permet la fédération des bases de données sur plusieurs serveurs/instances [BM00].

Dans SQL Server 2005, les fonctions de partitionnement des tables et index offrent flexibilité et performances et simplifient la création et la maintenance de telles tables. SQL Server 2005 permet le partitionnement horizontal par plage avec la ligne de données comme plus petite unité de partitionnement [T05].

Les partitions par plage sont des partitions de table définies par des plages personnalisables de données. L'utilisateur définit la fonction de partition avec des valeurs limites, un schéma de partition avec mise en correspondance des groupes de fichiers, ainsi que des tables mises en correspondance avec le schéma de partition. Une fonction de partition détermine à quelle partition appartient une ligne particulière d'une table ou d'un index. Chaque partition définie par une fonction de partition est mise en correspondance avec un emplacement de stockage (groupe de fichiers) via un schéma de partition.

2.4.2.2 Oracle

La version parallèle d'Oracle (*Oracle Parallel Server*) peut fonctionner sur des machines de type «réseau de station ». Toutefois, le modèle d'exécution retenu est le mode à disque partagé, et chaque processeur (ou plus exactement, l'instance d'Oracle fonctionnant sur chaque processeur) dispose donc d'un accès direct à tous les disques. Une telle approche suppose que l'accès aux disques soit particulièrement rapide et efficace, aussi l'implantation se fait-elle de préférence sur des systèmes possédant un réseau d'interconnexion rapide. De plus, les entrées/sorties sont limitées grâce à un gestionnaire de verrous distribués qui permet aux différents nœuds de conserver un verrou sur une ou plusieurs pages de données entre deux transactions [D92].

Le partitionnement permet de découper une table et/ou un index sur un ou plusieurs critères logiques. La table se comporte alors comme plusieurs tables de dimensions plus petites. La distribution des données supporte la mise en place d'un index sur chaque partition, ainsi que la possibilité d'un index global correspondant à l'assemblage de l'index de chaque partition. L'exécution des requêtes est parallélisée grâce à un coordinateur de requêtes, qui se charge de diffuser les sous tâches parallèles aux différents nœuds concernés puis de récupérer les résultats produits. Ceci est présenté sur la version 9i d'Oracle [D00].

Après sa version 9i, Oracle a crée une nouvelle version 10g (*g* pour *Grid Computing*). La principale nouveauté de cette dernière version concerne l'administration de grappes de serveurs via un mécanisme de distribution des programmes sur plusieurs machines. Ceci, en vue de mieux répartir la charge lors du passage en production, [S05].

Quant au partitionnement des tables sur Oracle 10g, une nouvelle méthode a été introduite en plus de méthodes de la version 9i. A savoir, de partitionnement par hachage, par intervalle et par liste (le partitionnement par liste définit une valeur par partition.). Il s'agit de la méthode des tables organisées en index (ang. *Index Organized Tables, IOTs*). Cette méthode combine les caractéristiques d'une table traditionnelle avec un accès rapide à un index dans un seul segment. Cependant, quelque soit la méthode choisie, toute partition d'une table reste statique. L'utilisateur (ou l'administrateur) doit exécuter manuellement plusieurs opérations afin de réorganiser une partition. Dans [LB05], on documente quatorze opérations de maintenance correspondates.

2.4.2.3 DB2

L'édition parallèle de DB2 (*PE DB2*) est une solution logicielle de base de données qui peut s'exécuter sur n'importe quelle plate-forme parallèle. Son modèle d'exécution est basé sur une architecture multi-ordinateur sans mémoire partagée dans laquelle le système de bases de données se compose de plusieurs nœuds logiques indépendants. Chaque nœud

logique représente une collection de ressources systèmes comprenant, un processeur, une mémoire centrale, un disque et une carte réseau. L'ensemble est contrôlé par un gestionnaire de bases de données autonome. La communication entre les nœuds logiques est basée sur des échanges de messages [B95].

Les tables sont fragmentées à travers les nœuds en utilisant une stratégie par hachage. Un utilitaire de maintenance permet de redistribuer les données entre les différents nœuds pour équilibrer la charge du système. L'optimiseur de données tient compte de l'information de hachage des tables pour produire les plans d'exécution des requêtes.

Sous DB2, une base de données peut être également fragmentée en plusieurs parties distinctes appelées partitions [M93]. Les enregistrements d'une table de la base de données peuvent être répartis sur plusieurs partitions. Chaque partition de bases de données a son propre journal de transaction et ses propres tables d'index.

2.4.3 Bilan

Avoir un partitionnement efficace dans les systèmes de bases de données parallèles a été l'objectif de nombreuses études depuis plusieurs années. Des travaux ont été lancés sur la réorganisation dans le partitionnement avec notamment quelques résultats comme nous venons de le présenter dans notre étude des principaux SGBDs. Toutefois, dans tous les SGBDs parallèles, le partitionnement reste statique que ce soit pour les SGBDs commercialisés ou ceux issus de la recherche. Si une table grandit, une intervention manuelle est nécessaire pour la partitionner.

Une solution intéressante semble émerger : rendre le partitionnement des données dynamique, ce qui est appelé aussi la *réorganisation on-line*. Plusieurs travaux de recherche ont été proposés pour remédier aux problèmes liés au partitionnement statique des SGBDs parallèles actuels. Il y a la contribution des éditeurs dans [SL96], à la conception de deux méthodes de réorganisation *on-line* qui sont respectivement la *réorganisation en-place* (*ang. in-place*) et la *réorganisation nouvelle-place* (*ang. new-place*).

- La *réorganisation en-place* crée une nouvelle structure de disque pour y basculer les traitements.
- La *réorganisation nouvelle-place* place les données parmi les pages de disque aussi longtemps qu'il y a des places pour les données.

Parmi les contributions dans [SL96], il y a aussi la proposition de la commande de *Placement de la limite de Partition* (ang. *Move Partition Boundary*) dans le système Tandem Non Stop SQL/MP. Cette commande concerne les changements *on-line* des partitions de bases de données adjacentes. La nouvelle limite ou frontière diminue la charge de n'importe quelle partition proche qui est pleine en assignant quelques tuples

dans une partition moins chargée. Cependant, la commande *Move Partition Boundary* reste une opération manuelle. De plus, cette commande est seulement décrite, aucune réalisation n'a été faite.

Les plus récentes stratégies de réorganisations *on-line* sont celles proposées dans [RZLM02]. Il s'agit d'utiliser un conseiller automatique (ang. *automatic advisor*) qui équilibre la charge de bases de données DB2 selon des réorganisations périodiques. Cependant, ce conseiller n'est juste qu'un utilitaire *offline* de DB2. Autrement dit, il ne s'exécute pas réellement d'une façon automatique et son utilisation reste manuelle.

Une autre proposition dans [GG05] décrit une autre technique sophistiquée de réorganisation basée sur le regroupement de bases de données (*ang. clustering*). Cette technique est appelée *Autoclust*. Elle extrait des ensembles fermés puis regroupe les enregistrements selon les clusters d'attribut. Le traitement *AutoClust* commence lorsque la moyenne du temps de réponse d'une requête de suppression par exemple suit le seuil défini par l'utilisateur. Cependant, cette technique n'est toujours pas mise en pratique.

2.5 Conclusion

Tout au long de ce chapitre, nous avons évoqué des éléments fondamentaux pour la compréhension des systèmes de bases de données parallèles auxquels nous allons faire référence dans les chapitres suivants. Nous pouvons conclure que le succès des produits commerciaux et des prototypes issus de la recherche a démontré la viabilité des machines de bases de données parallèles. Cependant, plusieurs problèmes restent non résolus dont principalement les algorithmes de partitionnement supportant les relations à distribution non uniforme. Lorsque le critère utilisé pour le placement change au point de dégrader l'équilibrage de la charge, une réorganisation dynamique devrait être faite. Une solution intéressante est d'effectuer une telle réorganisation « on-line » de manière efficace (à travers le parallélisme) sans interrompre les transactions qui arrivent. De plus cette réorganisation doit être transparente aux programmes qui s'exécutent sur le système parallèle.

Dans le chapitre suivant, nous présenterons une classe de techniques qui permet le partitionnement dynamique. Il s'agit des structures de données distribuées et scalables (SDDSs).

3 LES STRUCTURES DE DONNEES DISTRIBUEES ET SCALABLES

3.1 Introduction

L'explosion et la complexité des données rendent les bases de données de plus en plus volumineuses. Pour faire face à cela, les noyaux des SGBDs doivent alors tirer le meilleur parti des nouvelles architectures de machines. Les structures de bases de données distribuées et scalables - SDDSs - ont été proposées pour constituer de telles architectures [LNS93a, LN93b]. Elles permettent de fournir un mécanisme général d'accès à des données réparties dynamiquement. Ces structures assurent des temps d'accès beaucoup plus courts que les temps d'accès aux données stockées sur les disques.

Ce chapitre présente les principes de base des SDDSs. Nous commençons par présenter les principales architectures des systèmes distribués, en particulier celles utilisées par les SDDSs. Ensuite, nous définissons la scalabilité et nous enchaînons avec la présentation des principes des SDDSs. Nous insistons en particulier sur la famille RP* des SDDSs que nous utiliserons par la suite dans nos travaux.

3.2 Architecture des Systèmes

Dans ce qui suit, nous décrivons les différentes architectures distribuées appliquées aux systèmes de gestions de bases de données [G94, GG96, P04].

3.2.1 Architecture Client/Serveur

L'architecture Client/Serveur est un modèle d'architecture qui met en jeu une répartition entre des serveurs et des clients. Les serveurs gèrent tout ce qui concerne les données : traitement de requêtes, gestion de transactions, optimisation, etc. Les clients constituent un support pour les applications et les interfaces utilisateurs. Ils envoient leurs requêtes aux serveurs sans les optimiser et les serveurs effectuent le traitement et retournent les réponses des requêtes aux clients. La Figure 3-1 illustre cette architecture.

Il existe plusieurs variantes de l'architecture Client/Serveur :

- *Multiple-client-single-server*, appelée aussi architecture mono-tâche, où un serveur est associé à chaque utilisateur. L'inconvénient majeur de cette architecture ne diffère pas trop de celui des bases de données centralisées puisque la base est stockée sur seulement une machine (le serveur).
- *Multiple-client-Multiple-server*, appelée aussi architecture multi-tâches, où un serveur est capable de traiter plusieurs requêtes clients en parallèle. Une telle architecture permet de meilleures performances en présence d'un nombre important d'utilisateurs. Cependant, cette architecture peut conduire à un client lourd si chaque client gère ses propres connexions à son serveur, ou un client maigre si les fonctionnalités de gestion sont concentrées sur les serveurs.

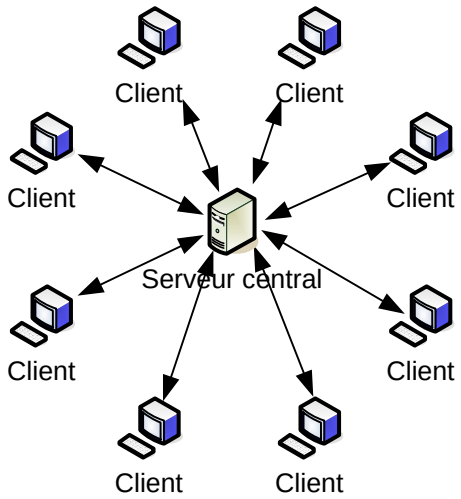

Figure 3-1 : Architecture Client/Serveur

3.2.2 Architecture Pair à Pair

Le principe de l'architecture *Pair à Pair* (ang. *Peer to Peer*, *P2P*) est la mise en commun de leurs ressources par un grand nombre de participants afin de fournir un service

commun. Par exemple, un tel service commun peut correspondre à un service de stockage de données ou à un système de calcul. Cette architecture fortement distribuée, comme illustré dans la Figure 3-2, est souvent obtenue en exécutant des programmes identiques sur un grand nombre de machines. Cela permet d'obtenir un certain nombre de bonnes propriétés comparativement aux systèmes basés sur l'architecture client/serveur. En effet, les systèmes pair à pair sont souvent plus tolérants aux pannes, passent plus facilement à l'échelle, et sont plus adaptatifs que leurs contreparties client/serveur. Parmi les sytèmes à architectures pair à pair, nous citons *Napster, Gnutella* et *SETI@home* (acronyme de *Search for Extraterristrial Intelligence at Home*) [S01].

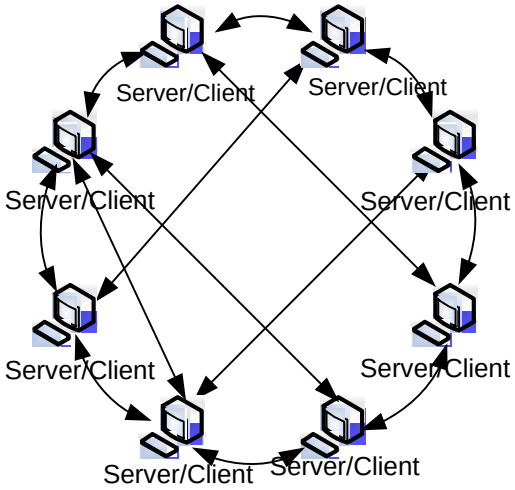

Figure 3-2 Architecture Pair à Pair

3.3 Les SDDSs

Avant de présenter le concept de la scalabilié et les principes des SDDSs, nous commençons par définir les multi-ordinateurs, le réseau pour lequel les SDDSs ont été introduites.

3.3.1 Les Multi-ordinateurs

Des recherches avancées sont menées pour mieux exploiter la puissance de calcul d'un ensemble d'ordinateurs interconnectés à travers des réseaux à haut débit (>100bits/s), [G96, CACM97, GW97, MC99]. De telles configurations existent déjà dans plusieurs organisations. Des termes sont apparus pour désigner les machines organisées de la sorte : *multi-ordinateurs*, *réseau de stations de travail* ou plus récemment de *Grid computing*. Les capacités cumulées de traitement parallèle et de stockage d'un multi-ordinateur sont impressionnantes et même supérieures aux performances des gros systèmes. De telles

configurations sont évolutives et exploitent au mieux les progrès constants au niveau du matériel.

Les multi-ordinateurs se caractérisent par la manière dont leurs composants de base (la mémoire principale, le processeur et les mémoires secondaires) sont interconnectés.

3.3.2 Concept de Scalabilité

La *scalabilité* (ang. *scalabilty*) est une caractéristique des architectures multiprocesseur. Elle permet à une base de données d'utiliser des ressources additionnelles de manière optimale. La scalabilité est ainsi définie comme étant la capacité d'une application à maintenir le même niveau de performance lorsque la charge augmente [G93, G99]. Deux paramètres principaux mesurent la scalabilité d'un système, il s'agit de : *speed-up* et *scale-up* :

1. *Speed-up*, appelé aussi le *facteur de rapidité*, mesure la diminution du temps de réponse d'une requête pour une taille de base de données constante et une augmentation des capacités de la configuration (nombre de noeuds). Autrement dit, si la capacité augmente d'un facteur de *n*, alors dans un système scalable, le temps de réponse d'une requête diminue d'un facteur de *n*. La Figure 3-3 illustre le facteur de la rapidité.

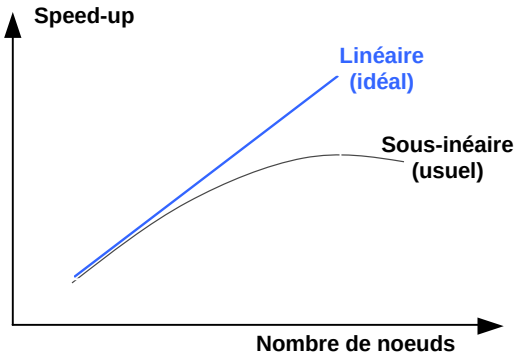

Figure 3-3 : Courbe idéale du facteur de rapidité

2. *Scale-up*, appelé aussi le *facteur d'échelle*, mesure la conservation du temps de réponse d'une requête de réponse pour une augmentation proportionnelle de la taille de la base de données et des capacités de la configuration. Autrement dit, si la taille d'une base de données augmente d'un facteur de *n*, alors il suffit d'augmenter la capacité de la configuration. La Figure 3-4 illustre le facteur de d'échelle.

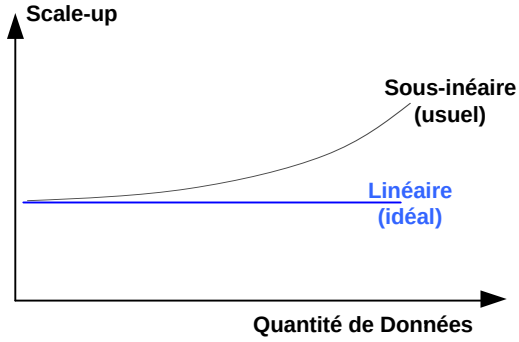

Figure 3-4 : Courbe idéale du facteur d'échelle

3.3.3 Principes des SDDSs

Les SDDSs présentent une nouvelle classe d'organisation de données définies spécifiquement pour les multi-ordinateurs. Elles sont proposées pour contourner les insuffisances des structures de données distribuées classiques qui se résument en particulier dans l'accès centralisé et la fragmentation statique des données [WBW94, KW94, LNS94, KLR94].

Les SDDSs sont conçues principalement selon l'architecture Client/Serveur. Elles diffèrent des autres schémas de gestion de données distribuées dans le fait qu'elles permettent une extension du nombre de serveurs tout en maintenant un coût minimal. Dans les schémas classiques, l'ajout d'un nouveau serveur nécessitera une réorganisation totale des données, ce qui est très coûteux. Tandis que les SDDSs gèrent l'extension du nombre de serveurs d'une manière transparente et dynamique en éclatant les serveurs débordés.

L'éclatement est basé sur un paramètre important d'un serveur SDDS, qui est sa capacité maximale en nombre d'enregistrements. Ainsi, pour toute insertion d'enregistrements, le serveur SDDS vérifie si la taille maximale est atteinte. Dans le cas où la capacité est atteinte, il lance l'éclatement de ce serveur en transférant la moitié de ses enregistrements sur un nouveau serveur. L'éclatement est basé sur les champs clés des enregistrements.

Quant au client SDDS, il dispose d'une image de la structure du fichier. Celle-ci représente les adresses des serveurs où se trouvent les enregistrements. Afin de minimiser les coûts de communication entre clients et serveurs et éviter tout goulot d'étranglement, les mises à jour de la structure SDDS ne sont pas envoyées au client d'une manière synchrone. Un client peut alors faire une erreur d'adressage de requête. Si une erreur est détectée, le serveur cible vérifie l'adresse de requête et l'envoie au serveur adéquat. Ce dernier envoie alors un message correctif au client ayant fait l'erreur d'adressage. Ce

message correctif est appelé *message d'ajustement d'image* (ang. *Image Adjustment Message, IAM).* Enfin, le client ajuste son image.

La distribution des enregistrements sur l'ensemble des serveurs est réalisée essentiellement selon deux stratégies : par hachage linéaire (LH*) ou par intervalle (RP*). D'où les deux familles principales des SDDS : *SDDS LH** et *SDDS RP**.

3.3.3.1 Les SDDS LH*

La famille des SDDSs LH* est une version distribuée du hachage linéaire [LNS96]. Elle est basée sur un algorithme de hachage extensible qui étend progressivement l'espace d'adressage primaire d'un fichier disque [LNS93a]. Le fichier s'étend alors et se rétrécit de manière dynamique afin d'éviter les débordements et donc la détérioration des performances d'accès. Plusieurs variantes de LH* ont été proposées notamment LH*$_{LH}$ et LH*$_{RS}$, comme illustré dans la Figure 3-5.

- LH*$_{LH}$ [KLR94] utilise deux niveaux d'indexation. Le premier niveau correspond à LH* et permet aux clients d'accéder aux serveurs pour exécuter une requête. Le deuxième niveau permet l'indexation interne des données suivant l'algorithme LH.
- LH*$_{RS}$ [LMRS99, LS00] supporte la haute disponibilité. Elle permet la reconstitution des données perdues suite à une panne d'un ou de plusieurs serveurs. Elle utilise pour cela les codes de *Reed Solomon* qui mettent en œuvre un calcul de parité [S02].

3.3.3.2 Les SDDS RP*

Cette famille des SDDSs réalise la fragmentation dynamique par intervalle [LNS94]. Elle est basée sur la technique des arbres-B+ distribués [VBWS98].

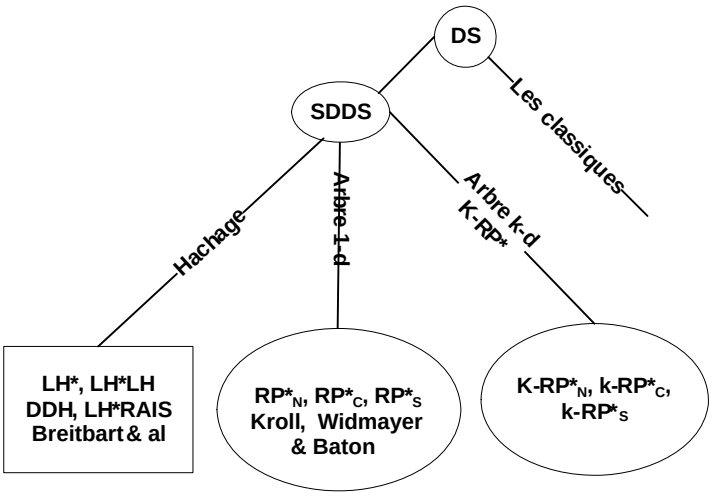

Figure 3-5 : Les familles des SDDSs

Dans ce qui suit nous insistons sur les SDDS RP* sur lesquelles repose notre travail.

3.4 Les SDDSs RP*

Les SDDSs RP* permettent de préserver l'ordre des données [LNS94]. Elles possèdent trois variantes : RP*n, RP*c et RP*s [K73, C79, KS86].

- La première variante RP*n définit un partitionnement ordonné comme un arbre-B+ mais sans aucun index. Elles utilisent des messages *multipoints* (ang. *multicast*) envoyés à tous les serveurs et seul le serveur concerné répond par un message *point à point* (ang. *unicast*).
- La seconde, RP*c, est un fichier RP*n muni d'index construits au niveau des clients à travers des messages d'ajustement d'images (IAMs). Chaque client a une image du fichier réparti sur l'ensemble des serveurs et en cas d'erreur d'adressage, le serveur ayant reçu la requête fait une redirection par *multipoints* vers les autres serveurs et le serveur concerné répond en envoyant un message correctif d'ajustement d'image à l'initiateur de la requête.
- Et enfin la troisième variante RP*s, elle ajoute à RP*c les index au niveau des serveurs, ce qui procure l'avantage d'utiliser des messages *point à point* en cas de redirection de la requête.

Après avoir présenté les différentes variantes d'une SDDS RP*, nous décrivons dans ce qui suit la structure et la manipulation d'un fichier SDDS RP* ainsi que la structure d'une image RP*.

3.4.1 Structure d'un fichier SDDS RP*

La structure d'un fichier RP* est similaire à celle d'un arbre-B+ (variante d'arbres-B implémentée en mémoire centrale). Chaque fragment d'un fichier correspond à une case pouvant contenir un maximum de b enregistrements. Ces fragments sont stockés au niveau de la mémoire centrale des serveurs. A chacune des cases, est associé un intervalle de clés borné par une clé minimale et une clé maximale. Ainsi, un enregistrement de clé C appartiendra à la case d'intervalle $[C_{Min}, C_{Max}]$ si $C_{Min} < c < C_{Max}$. L'ensemble des cases constituera un ensemble de clés ordonnées.

Initialement, un fichier RP* est composé d'une seule case avec un intervalle initial $[-\infty, +\infty]$. Toutes les insertions se font dans cette case jusqu'au dépassement de sa capacité maximale. Un éclatement est alors provoqué transférant vers un nouveau serveur tous les enregistrements (au nombre de $b/2$) dont la clé est supérieure à la clé médiane de la case.

Un gestionnaire de fichiers RP* a été proposé et implanté [DL00, DL01, D01]. Son prototype, appelé SDDS 2000, est téléchargeable du site web du CERIA [D01-p].

3.4.2 Manipulation d'un fichier SDDS RP*

Les clients effectuent diverses opérations sur les fichiers présents dans les mémoires des différents serveurs en envoyant des requêtes aux serveurs concernés. Ces opérations concernent la mise à jour, la suppression et l'insertion de nouveaux enregistrements, etc. On distingue deux types de requêtes :

3.4.2.1 Requête Simple

Elle correspond à la recherche, à l'insertion, à la suppression ou à la mise à jour d'un enregistrement de clé *C* ou encore au stockage et chargement des enregistrements constituant une case d'un fichier se trouvant sur différents serveurs.

3.4.2.2 Requête Parallèle

Ces requêtes concernent la recherche ou l'insertion d'un ensemble d'enregistrements appartenant à un intervalle de clés appelé l'intervalle de la requête.

3.4.3 Structure d'une Image SDDS RP*

Une image RP* est une collection d'intervalles et d'adresses des sites qui traduit la répartition des enregistrements sur les cases et les serveurs qui les hébergent. Elle est représentée par une table dynamique *T[0,1,…]*. Chaque élément *T[i]* de cette table, contient l'adresse d'une case et son intervalle. Logiquement, la table *T* est une liste ordonnée de couples *T[i] = (A, C)*, où *A* est l'adresse d'une case du fichier SDDS et *C* est la clé maximale que la case *A* peut contenir. Initialement $T = [(0,\infty)]$, elle évolue en fonction des messages correctifs d'ajustement d'image reçus qui entraînent l'insertion ou la mise à jour de couples.

3.4.3.1 Ajustement d'une Image SDDS RP*

La réponse à une requête contient un champ IAM (message d'ajustement d'image) qui permet au client de corriger son image du fichier. L'IAM se présente sous forme d'un ou deux triplets (λ, a, Λ) où $[\lambda, \Lambda]$ est l'intervalle de la case serveur ayant traité la requête et *a* son adresse.

L'ajustement de l'image du client se fait donc de manière asynchrone suivant l'algorithme ci-dessous :

(1). S'il n'existe pas un élément t appartenant à T avec $C(t) = \lambda$ et $\lambda \neq -\infty$ alors insérer $(*,\lambda)$ dans T.

(2). S'il existe un élément t appartenant à T avec $C(t) > \Lambda$, alors :
- si $C(t) = +\infty$ alors $t = (a, \Lambda)$ et ajouter $(*, +\infty)$ dans T.
- si $C(t) < +\infty$ alors $t = (a, \Lambda)$.

(3). S'il existe un élément t appartenant à T avec $t = (*,\Lambda)$, alors $t = (a, \Lambda)$.

(4). S'il n'existe pas d'élément $t = (a, \Lambda)$ appartenant à T, alors insérer (a, Λ) dans T.

3.5 Conclusion

Nous avons présenté dans ce chapitre les principes de base des SDDSs. Nous avons aussi montré les avantages de ces structures par rapport aux structures de données classiques en particulier dans leur partitionnement dynamique des données. Nous nous sommes intéressés dans la description des SDDSs à la famille des SDDSs RP* sur laquelle notre travail repose.

4 ARCHITECTURE DE SD-SQL SERVER

4.1 Introduction

Ce chapitre décrit l'architecture de SD-SQL Server[1], le système de gestion de bases de données distribuées et scalables que nous proposons. Nous commençons par détailler les différents objectifs qui nous ont motivés à cette proposition. Ensuite, nous décrivons l'organisation générale du système. Nous présentons ses éléments de base et les règles structurelles nécessaires à sa conception. Ensuite, nous décrivons son architecture avec tous ses composants. Nous présentons aussi ses fonctionnalités principales qui se résument dans le partitionnement des tables scalables.

4.2 Vue Générale d'un Système de Gestion de Bases de Données Distribuées et Scalables

L'architecture de référence d'un système de gestion de bases de données distribuées et scalables a été introduite la première fois dans [LRS02]. Dans le cadre de cette thèse, nous avons fait évoluer cette architecture pour mieux répondre aux besoins d'un système capable de gérer des données distribuées et scalables tant au niveau architecture qu'au niveau fonctionnalités.

Un système de gestion de bases de données distribuées et scalables - SD-DBS - est une généralisation d'un système de gestion de bases de données parallèles. Il est représenté par un ensemble de SGBDs distribués et liés entre eux comme illustré dan la Figure 4-1. Contrairement aux SGBDs parallèles où le nombre de nœuds de stockage est constant, dans un SD-DBS le nombre de nœuds est variable. Il varie selon l'augmentation ou la diminution du nombre de bases de données qu'il contient.

[1] Le nom SD-SQL Server est issu du SGBD SQL Server sur lequel nous avons réalisé notre système.

Nous avons défini les notions de base d'un SD-DBS à partir des notions de base des SDDSs puisque celles-ci s'exécutent dans un environnement distribué et scalable. En effet, un SD-DBS constitue une couche supplémentaire sur une collection d'instances de SGBDs liés entre eux, comme illustrée dans la Figure 4-1. Cette couche est basée sur les principes des SDDSs.

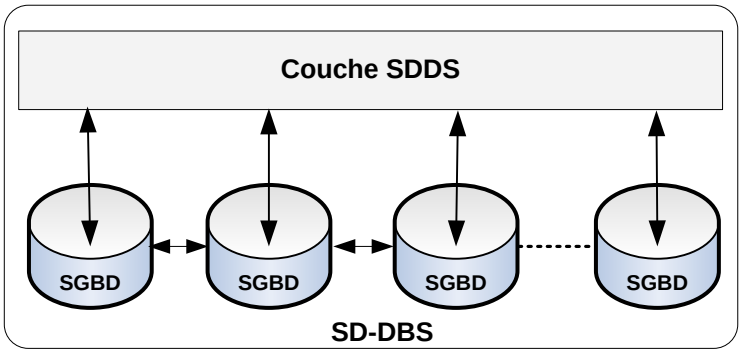

Figure 4-1 : Présentation générale d'un SDB-DBS

Nous avons introduit trois notions pour concevoir un SD-DBS : les bases de données scalables, les tables scalables et les images. Ces éléments constituent les données de base pour un SD-DBS. Nous décrivons brièvement ces éléments avant de les détailler par la suite.

- **Les bases de données scalables**

Nous avons appelé les bases d'un SD-DBS des *bases de données scalables* (ang. *scalable distributed databases, SDBs*). Une SDB est donc constituée d'un ensemble de bases distribuées sur plusieurs nœuds représentant des SGBDs. Nous avons appelé ces bases des *bases de nœuds* (ang. *node databases, NDBs*). Les NDBs sont perçues par les applications comme une seule base de données qui est la base scalable. Leur nombre est variable dans une SDB, autrement dit, on peut étendre (ou rétrécir) une SDB avec plus (ou moins) de NDBs.

- **Les tables scalables**

L'unité de stockage dans une SDB est une *table scalable*. De même, une table scalable est constituée d'un ensemble de tables distribuées sur plusieurs nœuds d'une base scalable. Chaque table distribuée composant une table scalable est appelée *segment*.

- **Les images**

Une image représente la définition du partitionnement d'une table scalable. Les applications accèdent aux tables scalables à travers leurs images. Ainsi, les segments qui constituent les tables scalables restent transparents vis-à-vis des applications.

Toutes ces notions sont similaires aux notions de bases des SDDSs RP*. Tout comme les SDDSs RP*, un SD-DBS permet de gérer l'extension du nombre de segments d'une table scalable en éclatant les segments débordés d'une manière transparente et dynamique. L'éclatement est basé sur un paramètre important d'une table scalable : il s'agit de sa capacité maximale en nombre de tuples, soit *b* cette capacité. Ainsi, pour toute insertion de tuples, le SD-DBS vérifie si la taille maximale de la table scalable est atteinte. Dans le cas affirmatif, il lance l'éclatement du segment débordant de cette table en transférant la moitié de sa capacité (*b/2* tuples) vers un nouveau nœud. Ce nœud est une partition de la SDB qui détient la table scalable qui éclate. S'il n'y a pas de nœuds disponibles de la même SDB pour héberger le nouveau segment (issu de l'éclatement du segment débordé), cette SDB sera ainsi étendue. De nouveaux nœuds de stockage seront ainsi ajoutés à la SDB et seront liés à ses nœuds.

De même que les SDDSs, chaque table scalable a une image qui représente son partitionnement actuel. Celle-ci représente alors les nœuds de la SDB où sont localisés les segments d'une table scalable.

Enfin, pour mieux voir la différence entre un SD-DBS et un SGBD parallèle, la Table 4-1 suivante compare ces deux types de système.

	SGBD parallèle	**SD-DBS**
Unité de stockage	table	table scalable
Nombre de nœuds de stockage	fixe	variable
partitionnement	statique	dynamique
Instance de stockage	BD	SDB

Table 4-1 : Comparaison entre les caractéristiques d'un SGBD parallèle et d'un SD-DBS

4.2.1 Objectifs d'un SD-DBS

Idéalement, un SD-DBS doit présenter les avantages suivants :

4.2.1.1 Partitionnement Dynamique

Comme nous l'avons déjà mentionné, le partitionnement dynamique est le principal objectif d'un SD-DBS. Si une table scalable monte en échelle, elle doit être partitionnée. Dans les SGBDs traditionnels, les administrateurs sont obligés d'intervenir pour partitionner une table manuellement quand elle est surchargée (ce qui est connu sous le nom de partitionnement statique).

En quoi le partitionnement dynamique peut-il être utile ? Lorsque les tables scalables deviennent très volumineuses, le partitionnement dynamique peut permettre de partitionner les données en sections plus petites et plus faciles à gérer sans faire appel aux administrateurs. Le partitionnement dynamique est une réorganisation *on-line* des données sans interrompre les transactions qui arrivent et se réalise d'une manière efficace à travers le parallélisme.

Le partitionnement dans un SD-DBS est horizontal. Il est effectué par intervalle tout comme le partitionnement dans les SDDSs RP*. Il permet de créer un ou plusieurs nouveaux segments pour chaque table scalable ayant un segment à éclater. Il transfert ainsi les tuples qui surchargent le segment éclaté vers ces nouveaux segments tout en laissant tous les segments (d'une table scalable) à moitié pleins. La table scalable aura alors de nouvelles partitions sur de nouveaux nœuds autres que ceux qui hébergent ses premiers segments. Toutes ces manipulations sont réalisées d'une manière transparente.

4.2.1.2 Possibilité d'Extension (Scalabilité)

Un SD-DBS doit avoir deux caractéristiques clés : une capacité d'accroissement (scale-up) linéaire et une accélération (speed up) linéaire. L'accélération correspond au gain de performance obtenu en augmentant le nombre de nœuds de stockage d'une SDB dans un SD-DBS et en laissant la taille des tables scalables utilisant ces nœuds (pour ses segments) telle qu'elle. La capacité d'accroissement est mesurée en augmentant proportionnellement la taille d'une table scalable et le nombre de ses segments. L'idéal est que l'accélération augmente proportionnellement au nombre de segments et que la capacité d'accroissement reste constante.

4.3 SD-SQL Server

SD-SQL Server est le SD-DBS que nous avons proposé. L'ensemble des nœuds qui composent SD-SQL Server est donc constitué des instances du SGBD SQL Server. Comme les nœuds du SD-SQL Server communiquent entre eux, les instances SQL Server sont donc les serveurs liés SQL Server [LS04].

Les NDBs qui constituent une SDB du système SD-SQL Server sont représentées par les bases de données se trouvant sur des nœuds distribuées SQL Server (serveur liés). Quant aux segments composant une table scalable, ce sont des tables du SGBD SQL Server [LS04].

Avant de détailler l'architecture du système SD-SQL Server, nous présentons tout d'abord ses règles structurelles.

4.3.1 Les Règles Structurelles

Nous avons conçu SD-SQL Server en tenant compte de certaines caractéristiques qui représentent les règles structurelles nécessaires à sa conception. Ces règles peuvent être différentes selon le SGBD utilisé pour concevoir un SD-DBS. Les règles que nous présentons ci-dessous concernent le système SD-SQL Server.

4.3.1.1 Vue Partitionnée Distribuée

Nous avons utilisé les vues partitionnées afin de présenter les images des tables scalables sur SD-SQL Server [LS04]. Une vue partitionnée distribuée est une vue qui lie horizontalement des données partitionnées à partir d'un ensemble de tables membres à travers un ou plusieurs serveurs. Cette liaison horizontale présente ces données partitionnées comme si elles forment une table unique [T05]. La Figure 4-2 illustre une vue partitionnée distribuée.

Il y a deux types de vues partitionnées : les vues partitionnées locales et les vues partitionnées distribuées. Dans le premier type, toutes les tables participant à la vue résident sur différentes bases de données de la même instance du SGBD. Dans le deuxième type de vues, il y a au moins une table parmi les tables participantes, qui réside sur un autre serveur distant. Dans notre travail, nous nous intéressons au deuxième type de vues partitionnées puisque les segments d'une table scalable sont distribués.

Lorsqu'une table scalable éclate, un ou plusieurs nouveaux segments sont créés et le partitionnement de cette table est ainsi modifié pour inclure les nouveaux segments. Comme les images définissent le partitionnement actuel d'une table scalable, alors dès que le schéma d'une table scalable est modifié son image doit l'être aussi. Nous entendons par la modification du schéma d'une table scalable, l'ajout d'un ou de plusieurs nouveaux segments résultant d'un éclatement. Puisque les images sont représentées par des vues partitionnées, la définition de ces vues doit être donc modifiée pour inclure les nouveaux segments.

Nous rappelons que les vues partitionnées de la plupart des SGBDs ne peuvent être modifiées. Seul le SGBD SQL Server permet leur mise à jour [BM00]. D'où d'ailleurs notre utilisation de ce SGBD pour la conception de notre SD-DBS.

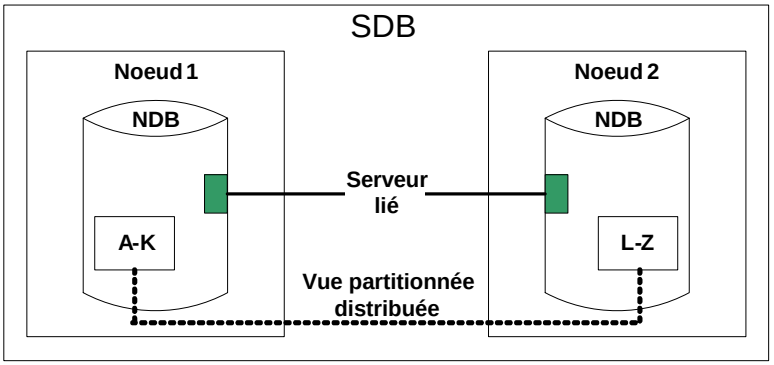

Figure 4-2 : Description d'une vue partitionnée

4.3.1.2 Clé de Partitionnement

La première étape de partitionnement des tables scalables consiste à définir les données à partir desquelles la clé de partitionnement est définie. La clé de partitionnement doit exister sous forme d'un attribut unique dans chaque segment de la table scalable et doit satisfaire un certain nombre de critères. Cette clé représente la clé primaire de la table si la clé primaire est construite sur un seul attribut de la table. Si la clé primaire est composée de plusieurs attributs, un seul attribut sera pris comme clé de partitionnement. C'est l'utilisateur du système SD-SQL Server qui choisit la clé de partitionnement. Sinon, SD-SQL Server choisit aléatoirement un attribut de la clé primaire [SLS05]. Ce mécanisme se décrit en détail ultérieurement.

La fonction de partitionnement définit le type de données sur lesquelles la clé (également appelée séparation logique des données) est basée. La fonction définit cette clé mais pas le positionnement physique des données sur le disque. Le positionnement des données est déterminé par le schéma de partitionnement.

Pour les partitions par intervalles de valeurs, l'ensemble des données est divisé par une limite logique liée aux données. L'utilisation des données dicte une partition par plages de valeurs lorsque la table est utilisée dans un schéma qui définit des limites logiques d'analyse (également appelées plages de valeurs). La clé de partitionnement pour une fonction de plage de valeurs peut comprendre une seule colonne. La fonction de partition inclut l'intégralité du domaine, même lorsque les données n'existent pas dans la table (en raison des contraintes d'intégrité sur les données). En d'autres termes, les limites sont définies pour chaque partition, mais la première et la dernière partitions incluent,

potentiellement, des lignes pour les valeurs les plus à gauche (valeurs inférieures à la condition de limite la moins élevée) et pour les valeurs les plus à droite (valeurs supérieures à la condition de limite la plus élevée). Ainsi, pour restreindre le domaine de valeurs à un ensemble de données spécifique, les partitions doivent être combinées à l'aide de contraintes d'intégrité *CHECK* [M00]. Nous décrivons ces contraintes dans la section suivante.

4.3.1.3 Contraintes d'intégrité

Les contraintes d'intégrité sont un moyen offert par certains SGBDs pour permettre de décrire les règles logiques que doivent respecter les données pour assurer la cohérence d'une base [Gar99]. Une fois cette description fournie au système, le SGBD fait en sorte que ces contraintes soient toujours vérifiées, en exécutant que les requêtes de mise à jour qui respectent toutes les contraintes. L'utilisation de contraintes d'intégrité permet donc de restreindre l'ensemble de données pour former une plage de valeurs finie plutôt qu'une plage de valeurs infinie.

Nous utilisons les contraintes d'intégrité dans notre système SD-SQL Server pour définir les limites des valeurs dans chaque segment d'une table scalable. Ces contraintes sont appliquées sur la clé de partitionnement de chaque segment. Elles servent à limiter les plages de valeurs dans les segments composant une table scalable. Ainsi, lors de chaque requête de mise à jour d'une table scalable, SD-SQL Server vérifie s'il y a des contraintes. S'il en trouve, il les teste et effectue l'action appropriée. Les contraintes d'intégrité sur la clé de partitionnement, d'une table scalable, permettent la mise à jour de leurs images représentées par les vues partitionnées et distribuées.

4.3.2 Architecture de Référence

Nous avons adapté l'architecture de notre système SD-SQL Server à partir de l'architecture de référence dans [LRS02]. Nous avons développé cette architecture et nous lui avons introduit d'autres concepts qui lui permettent de constituer un SD-DBS [LS04, LSS06].

L'architecture du système SD-SQL Server est basée sur les deux architectures logicielles client/serveur et pair à pair. Autrement dit, un nœud de stockage SD-SQL Server peut jouer le rôle d'un nœud serveur, d'un nœud client ou d'un nœud pair.

Avec la première architecture (client/serveur), SD-SQL Server repose sur le principe simple de la séparation d'une tâche de traitement en deux parties : l'une proche de l'utilisateur (le client), et l'autre proche de la base de données distribuées et scalables (le serveur). Les clients envoient leurs requêtes au serveur à partir de l'interface d'application

(au niveau client). Les serveurs sont responsables de la gestion des données et de l'exécution des tâches.

Avec la deuxième architecture pair à pair, SD-SQL Server regroupe les tâches du client et celles du serveur dans un seul composant appelé pair. Ce dernier joue alors le rôle d'un client et d'un serveur en même temps.

SD-SQL Server est une collection de nœuds SD-SQL Server. Un nœud SD-SQL Server représente une instance du SGBD SQL Server. Le nombre de nœuds dans l'architecture SD-SQL Server est variable. Autrement dit, nous pouvons ajouter ou supprimer un nœud de l'architecture SD-SQL Server d'une manière dynamique.

Parmi les nœuds SD-SQL Server, il y a un *nœud primaire*. Il s'agit du premier nœud créé pour l'architecture SD-SQL Server. Ce nœud détient la méta-base MDB. Celle-ci garde trace de tous les nœuds du système SD-SQL Server ainsi que de ses bases scalables. Elle sert comme dictionnaire du SD-DBS SD-SQL Server.

Chaque nœud SD-SQL Server détient une à plusieurs bases de nœuds (NDBs). Ces NDBs représentent les bases de données habituelles des SGBDs. Les NDBs peuvent partager le même nom sur différents nœuds SD-SQL Server. Ils forment dans ce cas ce que nous avons appelé une base de données scalable SDB [SLS05]. Ainsi, le nom commun entre les NDBs distribuées sur différents nœuds SD-SQL Server est le nom de leur SDB.

Selon son type (client, serveur ou pair), une NDB gère les tables scalables ou leurs images à l'aide de son gestionnaire. Au niveau de toute NDB, il y a des méta-tables décrivant les métadonnées gérées par la NDB.

Les NDBs de type serveur gèrent les tables scalables. Ainsi, une NDB serveur détient les tables scalables et les métadonnées les décrivant dans des méta-tables. Une NDB serveur se charge en particulier du partitionnement dynamique d'une table scalable (son éclatement).

Les NDBs de type client gèrent les images des tables scalables et l'interface d'application. Elles distinguent les images primaires des images secondaires comme nous le présenterons plus loin dans ce mémoire. En plus des images, une NDB de type client détient une méta-table décrivant ses images. Cette méta-table aide la NDB client pour accomplir sa tâche principale qui est l'ajustement des images lorsqu'elles sont incorrectes. La NDB client découvre qu'une image n'est pas correcte s'il y a une requête qui l'adresse. Les requêtes sont formulées à travers l'interface application/utilisateur qui est gérée également par la NDB client.

Les NDBs de type pair jouent les rôles de serveur et de client en même temps. Ainsi, elles accumulent toutes leurs tâches. La Figure 4-3 illustre l'architecture du système SD-SQL Server. Elle montre quelques SDBs et leurs NDBs localisées sur les nœuds *D1*…*Di+1*.

La NDB localisée sur le nœud *D1* est une NDB de type client puisque elle fait appel à des images et elle est liée à une interface d'application. Les méta-tables de cette NDB sont présentées par le catalogue *C* dans la Figure 4-3.

Les NDBs sur tous les nœuds (*D2*…..*Di*) sont de type serveur. Elles ne sont connectées à aucune interface d'application. Elles détiennent seulement les segments des tables scalables et les méta-tables qui les décrivent. Celles-ci sont présentées par le catalogue *S* (pour désigner *Server*).

Enfin, la NDB *Di+1* est de type pair. Comme le montre la Figure 4-3, la NDB du nœud *Di+1* est liée à une interface d'application. Elle détient les images ainsi que les segments des tables scalables. Le catalogue *P* désigne les méta-tables de la NDB pair. Le catalogue *P* est en réalité l'union des deux catalogues *C* et *S*.

Les NDBs de la Figure 4-3 contiennent une table scalable *T*. Cette table possède un index *I*. Nous supposons que la NDB du nœud *D1* détient l'image primaire de *T*. Cette image est donc l'union de tous les segments de *T* localisés sur les NDBs des nœuds *D2*…*Di*. Le segment primaire de *T* est localisé sur la NDB du nœud *D2*.

La NDB pair du nœud *Di+1* détient l'image secondaire de *T* qui est nommée *D1_T*. Le nom de l'image secondaire est différent de celui de l'image primaire (*T*) comme le montre la figure. Nous discutons cette différence des noms plus loin dans la description de ces composants.

Notons enfin que les segments de *T* ont tous le nom *_D1_T*. Ce nom est présenté par le couple *(nœud de création, nom de la table)*. Nous l'avons formulé ainsi afin d'éviter les conflits des noms sur les différentes NDBs de la même SDB. Ceci est le cas par exemple de plusieurs tables scalables nommées *T* mais créées sur différentes NDBs client. Nous discutons ces détails plus loin dans ce mémoire.

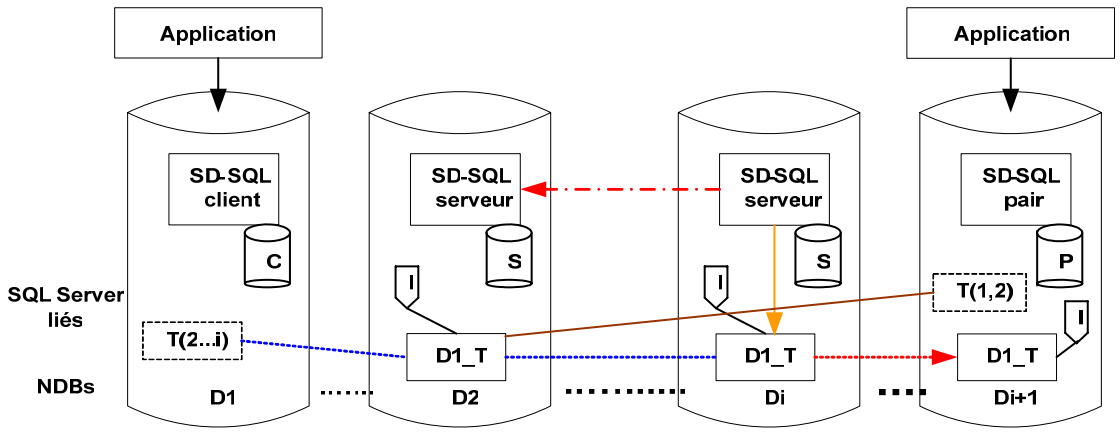

Figure 4-3 : Architecture de SD-SQL Server

4.4 Description des Composants SD-SQL Server

Après avoir décrit l'architecture générale de SD-SQL Server, nous passons maintenant à la description détaillée des différents composants de cette architecture [SLS05].

4.4.1 La Méta-base (MDB)

Les informations sur les différents nœuds ainsi que sur les SDBs de l'architecture SD-SQL Server, sont stockées dans la méta-base, que nous avons appelée MDB (ang. *Meta Database*). MDB est une base de données de type système dans SD-SQL Server. Elle permet de décrire ses nœuds et ses SDBs.

La MDB est gérée uniquement par l'administrateur du système (ang. *SD-DBA*, *Scalable Distributed Database Administrator*). Elle est créée lors de l'initialisation du système SD-SQL Server. Le premier nœud créé dans l'architecture SD-SQL Server, que nous avons appelé *nœud primaire*, détient la MDB. Les données qu'elle décrit sont organisées dans des méta-tables que nous décrivons dans la section suivante.

4.4.1.1 Les Méta-tables

Nous avons distingué les méta-tables décrivant les nœuds SD-SQL Server de celles décrivant les SDBs. Ces méta-tables ne se trouvent que sur la MDB.

Pour la gestion des nœuds SD-SQL Server, il y a la méta-table appelée *Nodes* décrite ainsi :

- ***Nodes (Node, Type)***

Chaque tuple de cette méta-table enregistre un nœud SD-SQL Server qui fait partie de l'architecture du système SD-SQL Server. Les attributs de la table *Nodes* sont comme suit :

- *Node* affecte le nom d'un nœud SD-SQL Server.
- *Type* indique le type du nœud SD-SQL Server. Un nœud peut être de type *serveur*, *client* ou *pair*. Ainsi le champ *Type* peut avoir les valeurs *'server'*, *'client'* ou *'peer'*.

- ***SDB (SDB_Name, Node, NDBType)***

Chaque tuple de cette méta-table enregistre une SDB. Les colonnes de la table SDB sont présentées comme suit :

- *SDB_Name* affecte le nom de la SDB.

- *Node* indique le nœud SD-SQL Server où la NDB primaire de la SDB a été créée. Ce nœud est de type serveur ou pair.
- *NDBType* indique le type de la NDB primaire de la SDB. Ce champ affecte alors les valeurs *'server'* ou *'peer'* puisque la NDB primaire d'une SDB est de type serveur ou pair.

4.4.2 Les Serveurs SD-SQL Server

Les serveurs SD-SQL Server sont les nœuds SD-SQL Server de type serveur. Ces nœuds détiennent les nœuds de bases de données (NDBs) de type serveur. Chaque NDB au niveau d'un nœud serveur présente un composant parmi les autres NDBs d'une SDB. Les NDBs détiennent les tables scalables et les méta-tables qui les décrivent. Les sections suivantes décrivent les différents composants d'une NDB serveur.

4.4.2.1 Les Tables Scalables

Une table scalable *T* est formellement un tuple *(T, S)*, où *T* est l'image primaire de la table scalable *T* et *S* est l'ensemble de ses segments. Les segments d'une table scalable sont stockés sur les NDBs des nœuds serveurs. Quant à l'image primaire *T* de la table, elle est stockée sur sa NDB client comme nous le décrivons plus tard. Une table scalable *T* monte en échelle à travers les éclatements de ses segments qui excèdent leur capacité.

Un segment d'une table scalable est présenté comme une table du SGBD SQL Server, que nous avons désignée dans notre système par une *table statique* munie de paramètres spécifiques sous SD-SQL Server. Nous distinguons, dans les segments d'une table scalable, le *segment primaire* et les *segments secondaires*.

- ***Segment Primaire***

Le segment primaire est le premier segment créé pour une table scalable. Il est créé lors de la création initiale de sa table. Il est localisé sur la NDB de type serveur de la SDB qui détient cette table. Cette NDB est considérée comme la NDB primaire de la table scalable en question.

- ***Segments Secondaires***

Les segments secondaires sont tous les autres segments qui composent une table scalable. Ils résultent de l'éclatement du segment primaire ou des autres segments secondaires. Ils sont localisés sur les autres NDBs (appelées *NDBs secondaires)* de la SDB qui détient leur table.

Le segment primaire et les segments secondaires de la même table scalable ont les mêmes caractéristiques. Chaque segment possède une *capacité*. Il s'agit de sa taille maximale calculée en nombre de tuples. Lorsqu'un segment excède cette capacité suite à une insertion, il deviendra surchargé et éclatera ainsi. L'éclatement est lancé par un *déclencheur* attaché au segment. Ce déclencheur se trouve sur chaque segment d'une table scalable. Le segment éclate en une ou plusieurs partitions horizontales selon sa taille. Les nouvelles partitions résultant de l'éclatement constituent les nouveaux segments de la table scalable correspondante. Ils sont distribués sur les nœuds de stockage (les NDBs) de la SDB courante (celle qui détient la table scalable).

Parmi les propriétés spécifiques à une table scalable, il y a aussi les contraintes d'intégrité. Chaque segment possède une contrainte d'intégrité (ang. *check constraint*). Celle-ci définit les intervalles de valeurs de partitionnement de la clé d'une table scalable. Ces intervalles partitionnent donc la clé de partitionnement d'une table scalable. Les conditions sur les contraintes d'intégrité rendent possible les mises à jour des vues distribuées et partitionnées comme nous l'avons déjà mentionné dans la Section 4.3.1. Elles constituent donc une condition nécessaire et suffisante pour qu'une table scalable sous SD-SQL Server soit mise à jour.

SD-SQL Server sauvegarde toutes ces propriétés des tables scalables dans les méta-tables que nous décrivons dans la section suivante.

4.4.2.2 Les Méta-tables

Les méta-tables sont des tables qui se trouvent sur chaque NDB du système SD-SQL Server. Au niveau des NDBs de type serveur, elles permettent le stockage des métadonnées décrivant ces NDBs et les tables scalables qu'elles détiennent. Les fonctionnalités principales de ces méta-tables se résument en ce qui suit :

- Définition des segments qui composent chaque table scalable en les référant aux NDBs qui les hébergent.
- Définition de l'état actuel du partitionnement des tables scalables, autrement dit, la définition de la NDB qui localise chaque segment d'une table.
- Sauvegarde des noms des NDBs de type serveur disponibles dans la SDB courante afin de les utiliser lors de l'éclatement d'un segment. Les NDBs disponibles sont celles qui n'ont pas encore été utilisées pour héberger de nouveaux segments de tables scalables.
- Définition de la NDB qui détient le segment primaire pour chaque segment composant une table scalable.
- Stockage des informations sur les capacités (taille maximale) des tables scalables.

Les méta-tables des NDBs de type serveur constituent un catalogue logique que nous avons appelé *S-Catalog*. *S-Catalog* se trouve donc sur chaque NDB de type serveur, comme illustré dans la Figure 4-3. Il est composé des méta-tables suivantes :

- ***RP (SgmNd, CreatNd, Table)***

La méta-table *RP* permet de définir le partitionnement distribué actuel de chaque table scalable *Table*. Elle sauvegarde alors les tuples décrivant tous les segments des tables scalables ainsi que les nœuds de stockage qui les hébergent. La table *RP* est définie par les attributs suivants :

- *SgmNd* indique les noms des NDBs serveurs qui détiennent les segments d'une table scalable.
- *CreatNd* indique le nom de la NDB de type client qui lance initialement la création d'une table scalable.
- *Table* est la colonne qui affecte le nom de la table scalable.

Le tuple (*SgmNd, CreatNd, Table*) est inséré dans la méta-table *RP* si la table scalable *Table* aura un nouveau segment créé sur la NDB *SgmNd* suite à un éclatement de l'un de ses segments. Nous précisons que la méta-table *RP* de la NDB primaire, d'une table scalable, est la seule qui sauvegarde les tuples décrivant les segments d'une table. Les autres méta-tables *RP* resteront vides tant que leurs NDBs ne détiennent pas un segment primaire d'une table scalable.

Soit la table scalable *T* initialement créée sur la NDB client *D* localisée sur le nœud client *C* (notée *C.D*). Nous supposons que *T* est partitionnée en deux segments : un segment primaire sur la NDB *S1.D* et un segment secondaire sur la NDB *S2.D*. Ainsi pour garder trace du partitionnement de la table scalable *T*, les tuples suivants seront insérés dans la méta-table *RP* de la NDB primaire : *(S1, C, T)* et *(S2, C, T)*. Nous gardons trace seulement des nœuds SD-SQL Server sans leur NDBs puisque celles-ci (les NDBs) sont les bases courantes.

- ***Size (CreatNd, Table, Size)***

Cette méta-table permet de définir la capacité d'une table scalable et donc la capacité de ses segments. Elle est définie par les attributs suivants :

- *CreatNd* garde trace de la NDB de type client qui lance la création d'une table scalable.
- *Table* affecte le nom d'une table scalable.
- *Size* indique la taille maximale calculée en nombre de tuples pour chaque table scalable et donc pour chacun de ses segments.

Un tuple *(CreatNd, Table, Size)* de la méta-table *Size* explique que la capacité de la table scalable, initialement créée sur la NDB client *CreatNd*, est de *Size* tuples. Le tuple *(CreatNd, Table, Size)* est inséré dans la méta-table *Size* de la NDB primaire qui détient la table *Table*.

- ***Primary (PrimNd, CreatNd, Table)***

La méta-table *Primary* permet de garder trace du noeud de la NDB primaire où est localisée chaque table scalable. Nous rappelons que la NDB primaire d'une table scalable est la NDB qui détient le segment primaire de cette table. La méta-table est définie par les attributs suivants :

- *PrimNd* indique les NDBs (serveurs) primaires des tables scalables. En d'autres termes, *PrimNd* pointe vers le nœud de la NDB qui contient le segment primaire d'une table scalable.
- *CreatNd* est la NDB client qui lance la création d'une table scalable.
- *Table* est le nom de la table scalable.

Nous supposons la table scalable *T* initialement créée par la NDB client *C.D* et qui a son segment primaire sur la NDB *S1.D* et son segment secondaire sur *S2.D*. Ainsi, dans chaque NDB *D* des nœuds *S1* et *S2*, la méta-table *Primary* aura le tuple *(S1, C, T)* où *S1* indique le nœud primaire de la table *T* et *C* le nœud de la NDB client où *T* a été créée.

- ***SDBNode (Node)***

La méta-table *SDBNode* pointe vers la NDB primaire qui compose sa SDB. Elle est définie par l'attribut *Node* :

- *Node* affecte le nœud SD-SQL Server qui détient la NDB primaire de la SDB courante.

Soit *D*, la NDB courante où se trouve la méta-table *SDBNode*. *D* est donc une NDB parmi les NDBs qui constituent la SDB *D*. Nous insistons qu'il ne s'agit que des NDBs de type serveur puisque ce sont celles-ci qui détiennent les segments des tables scalables. Comme chaque SDB a une NDB primaire, donc chaque NDB de type serveur devrait garder trace de sa NDB primaire. Nous avons proposé l'utilisation de la méta-table *SDBNode* pour faciliter la suppression d'une SDB, la gestion de pannes, etc.

Dans le cas de la table scalable *T* utilisée dans les exemples précédents, la méta-table *SDBNodes* est alors localisée sur la NDB *S2.D* (qui n'est pas primaire) et elle sauvegarde le tuple *(S1)* qui pointe vers le nœud de la NDB primaire de la SDB *D*.

↻ ***MDBNode (Node)***

Cette méta-table pointe vers le nœud primaire qui détient la MDB. Elle est donc définie par l'attribut *Node* :

- *Node* qui affecte le nom du nœud primaire du système SD-SQL Server.

Cette table est utile en cas de panne d'un nœud ou d'une NDB du système SD-SQL Server.

4.4.3 Les Clients SD-SQL Server

Les clients SD-SQL Server présentent tous les nœuds de type client de l'architecture SD-SQL Server. Ils détiennent les NDBs de type client de chaque SDB. Ces NDBs permettent de stocker les images des tables scalables et les méta-tables qui les décrivent. Les NDBs client sont connectées aux interfaces d'application/utilisateurs. Celles-ci envoient leurs requêtes aux NDBs client. Les requêtes adressent les images des tables scalables ou leur vues scalables. Nous décrivons dans ce qui suit les images et les vues scalables ainsi que les méta-tables sauvegardées dans les NDBs de type client.

4.4.3.1 Les Images

SD-SQL Server crée pour chaque table scalable son image. Les applications/utilisateurs interrogent les tables scalables à travers leurs images. Les requêtes des clients n'adressent pas alors les tables scalables mais plutôt leurs images. Les segments d'une table scalable restent transparents vis-à-vis des utilisateurs.

Une image représente l'ensemble des segments d'une table scalable ainsi que leurs emplacements (les NDBs serveurs qui les détiennent). Les images définissent alors le partitionnement des tables scalables. Cette définition ne correspond pas nécessairement au partitionnement exact d'une table. Autrement dit, une image ne présente pas forcément tous les segments qui composent une table scalable. Dans ce cas, on dit que l'image est non ajustée ou incorrecte. Elle doit donc être ajustée.

L'ajustement d'une image s'effectue uniquement lorsqu'il y a des requêtes qui l'interrogent. Nous avons évité d'effectuer l'ajustement des images au moment du partitionnement des tables scalables qu'elles représentent afin de ne pas encombrer le système. Ainsi, si une table scalable a un nouveau segment suite à un éclatement, ce segment ne sera représenté par l'image de la table que si cette image est interrogée. Nous décrivons ce point en détail plus loin dans ce rapport. La définition interne d'une image se présente sous forme d'une vue partitionnée distribuée. Celle-ci définit l'union des segments d'une table scalable comme nous l'avons décrit précédemment dans la Section 4.3.1.

Nous avons défini deux catégories d'images : *les images primaires* et *les images secondaires* :

↻ Image Primaire

Une image primaire est créée au moment de la création de sa table scalable. Chaque table scalable n'a qu'une seule image primaire. Celle-ci partage le même nom que sa table. Elle est créée sur la NDB de type client qui lance la création de la table. Initialement, l'image primaire représente seulement le segment primaire de la table scalable. Dans la vue partitionnée distribuée qui la définit, il n'y a que le segment primaire et sa localisation (la NDB qui le détient). C'est seulement lorsqu'il y a des requêtes qui adressent l'image primaire qu'elle est ajustée pour contenir tous les autres segments de sa table scalable.

Exemple

Soit la table scalable *T* créée à partir la NDB client *D* du nœud *C* de l'architecture SD-SQL Server. Nous supposons que le segment primaire de *T* est créé sur la NDB serveur *D* du nœud *S1*, c'est-à-dire *S1.D*. Ainsi, l'image primaire, correspondant à la table scalable *T*, sera créée sur la NDB client *C.D* lors de la création de *T* et elle sera définie comme suit[2] :

```
CREATE VIEW T AS
SELECT * FROM S1.D.T
```

↻ Image Secondaire

Une image secondaire est créée en différé de la création de sa table scalable. Une table scalable peut avoir plusieurs images secondaires. Celles-ci sont créées sur des NDBs de type client du système SD-SQL Server. Ces NDBs sont différentes de celle qui détient l'image primaire de la même table scalable.

Les images secondaires ont la même définition que l'image primaire sauf que celle-ci est créée sur la NDB client où est créée sa table scalable. De plus, le nom d'une image secondaire est différent de celui d'une image primaire. Contrairement à cette dernière qui garde le même nom que sa table, les noms des images secondaires font référence à la NDB client qui lance la création de leur table scalable.

[2] Nous utilisons les noms logiques (sans référence au propriétaire de la base) des segments ainsi que ceux des images. Nous décrivons leurs noms physiques plus loin dans ce rapport.

4.4.3.2 Les Vues Scalables

Les applications peuvent adresser des vues scalables dans leurs requêtes. Une vue scalable est présentée comme une vue SQL d'un SGBD. Ce qui la différencie des autres vues est le fait qu'elle fait appel à des images de tables scalables d'une manière directe ou indirecte. Dans le premier cas, une image est adressée dans la définition même de la vue scalable. Dans le deuxième cas, la vue scalable adresse d'autres vues, qui elles mêmes adressent des images. De ce fait, nous avons défini les vues scalable à plusieurs niveaux :

- Une vue scalable est de niveau 1 si elle contient dans sa définition même une référence à une image. La vue *V* ci-dessous est une vue scalable de niveau 1 puisqu'elle fait appel à l'image *T* directement dans sa définition :

```
CREATE VIEW V AS
SELECT * FROM T
```

- Une vue scalable est de niveau *i* si elle fait référence à une autre vue scalable de niveau *i-1*. La vue scalable *V1* suivante est de niveau 2 puisqu'elle fait référence à la vue scalable *V* de niveau 1.

```
CREATE VIEW V1 AS
SELECT * FROM V
```

4.4.3.3 Les Méta-tables

Les méta-tables des NDBs client permettent le stockage des métadonnées décrivant les images des tables scalables. Les fonctionnalités principales de ces méta-tables se résument en ce qui suit :

- Définition des tables scalables représentées dans des images primaires ou secondaires.
- Définition du nombre de segments, d'une table scalable, représentés dans chaque image (ceci aide dans l'ajustement de l'image).
- Sauvegarde des nœuds de type serveur qui servent de nœud primaire pour la création des tables scalables.

Notons que les vues scalables ne sont pas présentées dans les méta-tables. Nous rappelons que les vues scalables servent à adresser des images à plusieurs niveaux.

Les méta-tables qui se trouvent sur les NDBs de type client constituent un catalogue logique que nous avons appelé *C-Catalog*. *C-Catalog* se trouve donc sur chaque NDB

client, comme il a été illustré dans la Figure 4-3. Il est composé des méta-tables suivantes :

- ***Image (Name, Type, PrimNd,Size)***

La méta-table *Image* permet de définir chaque image de la NDB courante (où la table *Image* se trouve). Elle sauvegarde toutes les informations décrivant les images primaires et secondaires. La table *Image* est définie par les attributs suivants :

- *Name* indique les noms des images.
- *Type* précise si une image est de type primaire ou secondaire. Ainsi, ce champ peut avoir la valeur *'Primary'* ou *'Secondary'*.
- *PrimNd* indique pour chaque image le nœud de la NDB où se trouve le segment primaire de la table scalable représentée par cette image. Ceci aide à l'ajustement de l'image plus tard.
- *Size* indique pour chaque image le nombre de segments de la table scalable qu'elle référencie. De même, ce nombre aide dans l'ajustement de l'image.

Par exemple, le tuple *(T, Primary, S1, 1)* de la méta-table *Image* indique qu'il existe sur la NDB client courante une image primaire représentant un seul segment (*Size=1*) de la table scalable *T*. Cette table a son segment primaire sur la NDB du nœud *S1*.

- ***Server (Node)***

La méta-table *Server* met un certain nombre de nœuds serveur (ou pair) à la disposition des NDBs client. Celles-ci utilisent la table *Server* pour pouvoir choisir un nœud pour héberger les segments primaires des tables scalables à créer. L'enregistrement des nœuds de type serveur dans la méta-table *Server* se fait lors de l'initialisation (ou la création) de ces nœuds.

4.4.4 Les Nœuds Pairs

Un nœud SD-SQL Server de type pair est un nœud qui joue le rôle d'un nœud serveur et d'un nœud client en même temps. Les NDBs qu'il détient sont alors de type pair, client ou serveur. Une NDB hérite le type du nœud qui l'héberge. Ainsi, un nœud de type pair, englobant toutes les fonctionnalités d'un nœud serveur et d'un nœud client, peut héberger des NDBs de type pair, de type client ou de type serveur. Par contre, un nœud client ne peut pas héberger une NDB de type serveur ou pair. Il en est de même pour les nœuds de type serveur.

Les NDBs de type pair sauvegardent toutes les méta-tables décrivant les tables scalables et leurs images. Ces méta-tables, représentées par *P-Catalog* dans la Figure 4-3, sont donc l'union des catalogues *S* et *C* décrits précédemment.

4.5 Fonctions de SD-SQL Server

Les fonctionnalités du système SD-SQL Server sont partagées entre ses clients et ses serveurs. Au niveau de chaque NDB, il y a un gestionnaire qui assure ces fonctionnalités. Sur les NDBs serveurs, il y a les gestionnaires de type serveur qui permettent de compléter les éclatements des segments débordants. Sur les NDBs client, leurs gestionnaires ajustent les images non correctes et permettent de traiter toutes les requêtes qui passent par l'interface d'application du système SD-SQL Server [SLS05]. Dans ce qui suit, nous décrivons les tâches principales du système qui sont l'éclatement des tables scalables et l'ajustement des images. Quant aux commandes de l'interface d'application, nous les développerons dans le chapitre suivant.

4.5.1 L'Eclatement

L'une des caractéristiques les plus importantes d'une table scalable est de pouvoir s'étendre dynamiquement sur différentes NDBs d'une SDB. Cette extension est possible grâce aux éclatements des segments d'une table scalable. Pour réaliser cette opération, il est nécessaire d'avoir un algorithme pour déterminer quelle NDB doit recevoir les nouveaux segments et quelle stratégie doit être adoptée pour réaliser un transfert efficace des données.

Avant de décrire les étapes de l'algorithme d'éclatement, nous énumérons d'abord ses objectifs qui se résument en ce qui suit :

- L'éclatement supprime les tuples débordant du segment qui éclate tout en laissant ce segment à moitié plein. Afin de supprimer la surcharge de tuples du segment éclatant, l'éclatement transfère quelques tuples vers au moins un nouveau segment. Chaque nouveau segment finit à moitié plein aussi. Le résultat final d'un éclatement est alors des segments pleins à un facteur d'environ 70%.

- Le temps d'un éclatement, déclenché par une insertion, peut être plus long que le temps de l'insertion elle même. Ainsi, pour ne pas pénaliser l'insertion, le traitement de l'éclatement ne doit pas conduire l'insertion qui le déclenche en temps mort (ang. *timeout*).

- ↻ L'allocation des NDBs pour les nouveaux segments issus d'un éclatement doit permettre d'équilibrer la charge entre les différents nœuds du système SD-SQL Server. L'algorithme d'éclatement alloue les mêmes nœuds aux segments successifs de différentes tables scalables créées sur la même NDB client. Ce principe d'éclatement permet de réduire les temps d'exécution des requêtes puisque celles-ci interrogent généralement les tables du même client.

- ↻ L'exécution concurrente de l'éclatement et des requêtes doit être cohérente. Autrement dit, s'il y a une requête scalable concurrente qui utilise le même segment que l'éclatement utilise, cette requête sera exécutée soit avant ou après l'éclatement.

L'algorithme d'éclatement décrit le mécanisme d'éclatement d'une table scalable tout en respectant les objectifs cités ci-dessus. Nous avons organisé les différentes étapes de cet algorithme dans un agent asynchrone que nous avons appelé *éclateur* (ang. *splitter*). L'éclateur est lié au déclencheur du segment qui éclate. Ainsi, si un segment excède sa capacité, suite à une insertion, son déclencheur lance l'éclateur. L'éclatement est effectué d'une façon asynchrone, par rapport à l'insertion (qui le déclenche) ou toute autre requête, afin de ne pas pénaliser ces requêtes.

Nous passons dans ce qui suit à la description des étapes de l'algorithme d'éclatement :

4.5.1.1 Allocation d'une NDB

Le choix d'une nouvelle NDB, pour héberger un nouveau segment résultant de l'éclatement, est à la charge de la NDB qui détient le segment en débordement. La liste des NDBs disponibles pour l'allocation de nouveaux segments est connue à l'avance par toutes les NDBs de type serveur. Elle est définie à l'aide de la méta-table *NDB* localisée au niveau de chaque NDB du segment primaire. La méta-table *NDB* contient tous les nœuds qui détiennent les NDBs de la SDB courante, ainsi que les types de ces nœuds. Pour avoir une NDB disponible, il suffit que l'éclateur sélectionne un nœud de type serveur (ou pair) de la méta-table *NDB*. Ensuite, il utilise la NDB en question du nœud sélectionné pour héberger un nouveau segment d'une table scalable. Il y a deux cas qui se présentent pour accéder à la méta-table *NDB* afin de sélectionner une NDB disponible :

- o Si le segment qui éclate est un segment primaire, alors l'éclateur accède directement à la méta-table *NDB* qui se trouve sur sa NDB même. Nous rappelons que toutes les informations concernant une table scalable sont principalement stockées dans les méta-tables de la NDB qui détient le segment primaire.

- Si le segment qui éclate est un segment secondaire, alors son éclateur cherche la méta-table *NDB* dans la NDB du segment primaire de la même table scalable. Comme nous l'avons déjà présenté, dans chaque NDB il y a la méta-table *Primary* qui sauvegarde les noms des NDBs disposant des segments primaires. Ainsi, un segment secondaire récupère le nom de sa NDB primaire à partir de la méta-table *Primary*. Ensuite, il cherche la liste des NDBs disponibles dans la méta-table *NDB* de la NDB trouvée.

Une fois l'éclateur accède à la méta-table *NDB*, il sélectionne un nœud disponible pour la SDB courante. Ce nœud est choisi de telle façon qu'il n'y ait pas une NDB hébergeant déjà un segment de la même table scalable qui éclate. Nous désignons cette table par *T*. Pour vérifier cette condition, l'éclateur utilise aussi la méta-table *RP* localisée sur la même NDB du segment éclatant. En effet, il sélectionne de cette méta-table, les nœuds (désignés par l'attribut *SgmNd*) des tuples décrivant la table scalable *T*. Ensuite, il sélectionne les nœuds *Nodes* de type serveur (ou pair) de la méta-table *NDB*. Enfin, il choisit les nœuds sélectionnés de la méta-table *NDB* qui n'existent pas dans *RP*. Le résultat sera donc un nœud qui détient une NDB, de la SDB courante, qui n'a pas encore hébergé un segment de la table *T*. Si le résultat correspond à plusieurs nœuds disponibles, l'éclateur choisit alors le premier nœud dans la liste. La requête SQL suivante permet de donner ces résultats :

```
SELECT TOP 1 Node FROM NDB WHERE NOT EXISTS
(SELECT SgmNd FROM RP WHERE tab= 'T' AND SgmNd = Node)
AND (NdbType='server' OR NdbType ='peer')
```

Dans le cas où la requête ci-dessus ne retourne pas de résultat, ceci signifie qu'il n'existe pas des NDBs disponibles, dans la SDB courante, pour héberger des nouveaux segments. Ainsi, il faut étendre cette SDB pour contenir de nouvelles NDBs. Nous supposons son extension par une seule nouvelle NDB. Pour cela, il faut avoir un nœud SD-SQL Server disponible qui ne détient pas encore une NDB de la SDB courante. Pour cela, l'éclateur suit les étapes suivantes :

- Il cherche dans la méta-table *Nodes* de la MDB un nœud qui vérifie les conditions ci-dessous :
    - Ce nœud ne doit pas être le nœud primaire de la SDB courante, c'est-à-dire il n'existe pas dans la méta-table *SDB* de la MDB.
    - De plus, il faut que ce nœud n'existe pas dans la table *NDB* qui compose les nœuds des NDBs de la base courante.
- Une fois un nœud est trouvé, SD-SQL Server crée donc une nouvelle NDB sur ce nœud. Cette NDB sera la localisation du nouveau segment qui résulte de l'éclatement de la table scalable *T*.

Si l'éclateur ne trouve pas un nœud SD-SQL Server pour en créer une nouvelle NDB de la SDB courante, donc les étapes ci-dessus ne peuvent être accomplies. Il faut étendre ainsi le nombre de nœuds dans le système SD-SQL Server. Pour cela, il faut chercher un nœud SQL Server libre qui soit un serveur lié et qu'il n'est pas encore utilisé dans l'architecture SD-SQL Server.

Après avoir présenté les détails, avec tous les cas critiques qui peuvent exister, sur l'allocation d'une NDB pour héberger un nouveau segment d'une table scalable, nous passons maintenant à la description du transfert des données débordant vers la nouvelle NDB sélectionnée.

4.5.1.2 Transfert des données

Une fois la phase d'allocation de NDB terminée, l'éclateur commence la phase de transfert des données. Cette phase consiste à faire migré les tuples qui surchargent un segment de sa capacité. Notons *b* la capacité du segment éclatant. Celle-ci est sauvegardée dans la méta-table *Size*.

L'éclatement est effectué selon le schéma de partitionnement du segment à éclater. Ce schéma consiste à ajouter *N (N>=1)* segments à sa table scalable. Soit *T* cette table, et soit *P* l'ensemble des tuples qui déclenchent l'éclatement de l'un de ses segments. Chaque NDB du segment éclaté coupe *P* en portions successives *P1, P2…, Pn*. Chaque portion *Pi* représente la moitié de la capacité *b* de la table *T*, c'est-à-dire *Pi* reçoit la partie entière de *b/2* tuples (*INT(b/2)*). Au moment de l'éclatement d'un segment, SD-SQL Server transfère la moitié supérieure *b/2* de ses tuples vers un nouveau segment sur la NDB sélectionnée dans l'étape précédente. Pour cela il suit les étapes suivantes :

- **Création de nouveaux segments**

Avant de transférer les tuples débordant du segment qui éclate, SD-SQL Server crée d'abord le(s) nouveau(x) segment(s) sur les NDBs sélectionnées. Chaque nouveau segment aura le même schéma que le segment éclatant. Ce schéma inclut la même définition du segment, son nom, sa clé primaire et ses index. Seulement les contraintes d'intégrités seront modifiées comme nous l'expliquerons plus tard. Nous supposons les hypothèses suivantes :

- *S* est le segment à éclater ;
- *Pi* est une portion des tuples à transférer tel que *Pi=INT (b/2)* ;
- *C* est la clé primaire du segment *S* ;
- *Si* est le nouveau segment, de la NDB *Ni*, qui recevra la portion *Pi*.

La création d'un nouveau segment exécute la requête SQL suivante pour tout *i=1 à N* :

SELECT TOP Pi WITH TIES * INTO Ni.Si FROM S ORDER BY C ASC

Cette requête permet de créer un segment *Si* sur la NDB *Ni* (sélectionnée dans l'étape précédente) avec les mêmes attributs du segment éclatant *S*. De plus, elle transfert une portion *Pi* du segment *S* dans le nouveau segment *Ni.Si*. Par contre, la clé primaire, les index et les contraintes d'intégrités ne sont pas encore établis.

Notons aussi l'utilisation du mot clé *WITH TIES* dans la requête. Celui-ci permet de transférer les tuples ayant des duplicatas dans leur clé de partitionnement. Ce cas se présente tout particulièrement dans le cas d'une clé primaire composée de plusieurs attributs clé. Le fait de transférer tous les duplicatas de la clé de partitionnement nous évite des problèmes dans la définition des contraintes d'intégrité.

- **Etablissement du schéma de partitionnement**

Une fois le nouveau segment est créé, l'éclateur complète son schéma de partitionnement. Tout d'abord, il modifie le schéma du nouveau segment en lui appliquant une clé primaire. Il s'agit de la même clé primaire du segment original *S*. L'éclateur cherche l'attribut doté d'une clé dans le segment *S* éclatant et l'applique ensuite sur le même attribut pour chaque nouveau segment *Si*. Ensuite, l'éclateur détermine les index du segment original *S* et les applique sur chaque nouveau segment *Ni.Si*.

Pour définir chaque segment *Si* comme une partition de la table scalable *T*, SD-SQL Server lui définit une contrainte d'intégrité. Nous rappelons qu'une contrainte d'intégrité est une clause permettant de contraindre la modification de tables. Soit *C(S)* la contrainte d'intégrité qui définit les limites inférieure *L* et supérieure *H* d'une clé de partitionnement d'un segment *Si*. Ainsi, *C(Si)* est définie comme suit :

$$C(Si) = \{C : L \leq C < H\}$$

Une contrainte d'intégrité n'est pas forcément créée pour un segment primaire lors de la création initiale de sa table scalable. Cependant, lors de la création de nouveaux segments pour une table scalable suite à un éclatement, les contraintes d'intégrités doivent être calculées pour chaque segment de la table scalable.

Soit *Hi* la plus grande valeur de la clé de partitionnement dans la portion *Pi (i>1)* transférée. Soit H_{N+1} la plus grande valeur de la clé dans le segment éclatant. Ainsi, les limites inférieure et supérieure du nouveau segment *Si* qui reçoit la portion *Pi* sont $L=H_{i+1}$ et $H=H_i$. Quant au segment éclatant, il garde sa valeur inférieure *L* qu'il avait déjà lors de la création. Par contre, il aura comme valeur supérieure la nouvelle valeur $H'=H_{N+1}$ où $H'<H$. En appliquant ces contraintes d'intégrité, la table scalable *T* sera partitionnée par intervalle.

Une fois, les nouveaux segments résultant de l'éclatement sont créés, l'éclateur effectue les étapes suivantes :

- Il enregistre les informations sur le(s) nouveau(x) segment(s) dans les méta-tables correspondantes. Il insère les tuples décrivant le segment *Si* dans la méta-table *Primary* de la NDB courante (du segment *Si*), et la méta-table *RP* de la NDB du segment primaire de la table scalable *T*. Il insère aussi le tuple décrivant la capacité du segment *Si* dans la méta-table *Size* de sa NDB.
- Enfin, il supprime les tuples transférés vers les nouveaux segments du segment *S* qui a éclaté.

4.5.1.3 Types d'Eclatement

L'éclatement d'une table scalable varie selon les paramètres suivants :

- Le nombre de segments à éclater dans une table scalable.
- Le nombre de tuples insérés dans une table scalable.

Selon ces paramètres, nous avons défini les trois types d'éclatement suivants [SLS05, LSS06b] :

↻ **Eclatement dû à une insertion d'un tuple dans un segment primaire**

Soit *N* le nombre de segments issus d'un éclatement. Si l'insertion d'un tuple déclenche l'éclatement d'un segment *Si*, donc un seul nouveau segment résultera de cet éclatement comme illustré dans la Figure 4-4. Dans ce cas là, *N* sera égal à 1.

↻ **Eclatement dû à une insertion d'un bloc de tuples dans un segment**

Si une insertion d'un bloc de tuples dans un segment *Si* déclenche son éclatement, alors plusieurs nouveaux segments *(N>1)* résulteront de cet éclatement, comme l'illustre la Figure 4-5.

↻ **Eclatement dû à une insertion multiple dans plusieurs segments**

Ce type d'éclatement, illustré dans la Figure 4-6, regroupe les deux types précédents. Ainsi, une insertion d'un bloc de tuples, qui déclenche l'éclatement, concerne plusieurs segment de cette table.

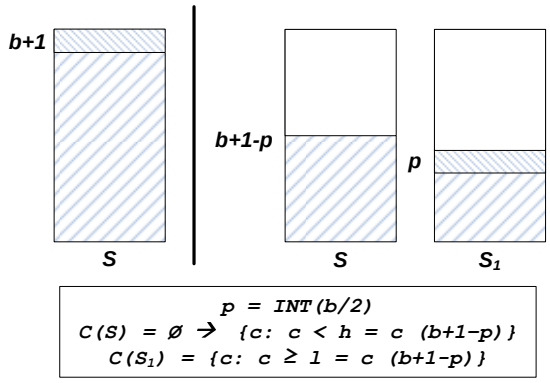

Figure 4-4 : Eclatement suite à une insertion d'un tuple dans un segment primaire

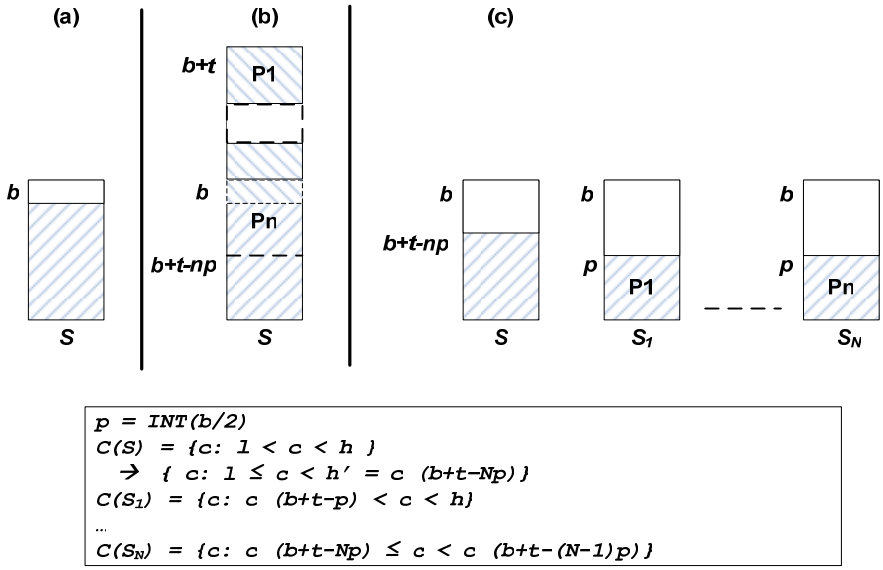

Figure 4-5 : Eclatement suite à l'insertion d'un bloc de tuples dans un segment

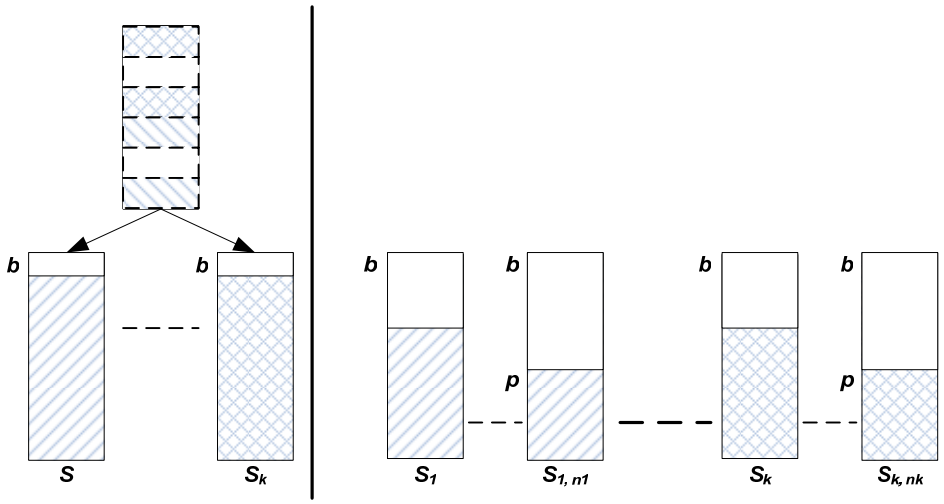

Figure 4-6 : Eclatement suite à l'insertion d'un bloc de tuples dans plusieurs segments

4.5.2 Ajustement des images

L'ajustement d'une image est effectué lorsqu'il y a une requête qui interroge cette image. Nous avons appelé de telles requêtes, adressant des images de tables scalables, des *requêtes scalable.* L'image adressée dans une requête scalable peut être une image primaire ou secondaire. Elle peut être invoquée directement dans une requête ou par le biais d'une vue scalable.

L'ajustement d'image est effectué sur une NDB de type client (ou pair). Le gestionnaire de cette NDB formule à partir d'une requête scalable *Q* une requête SQL *Q'* qui peut être exécutée sur le SGBD SQL Server. *Q'* adresse la vue partitionnée distribuée présentant une image, soit *T* cette image. Avant d'exécuter la requête *Q'*, SD-SQL Server vérifie si l'image *T* est correcte, c'est-à-dire si elle définit tous les segments de sa table scalable *T*. Pour cela, le gestionnaire SD-SQL Server suit les étapes suivantes :

- Il cherche le nombre de segments définis dans l'image *T* en utilisant la méta-table *Image*. Ce nombre est sauvegardé dans la colonne *Size* du tuple décrivant l'image *T*, soit *N* ce nombre.
- Ensuite, il récupère le nombre de segments (qui composent réellement la table scalable *T)* en utilisant la méta-table *RP*, soit *N'* ce nombre. Pour cela, il exécute la requête SQL suivante qui retourne le nombre de ces segments[3] :

[3] Notons que nous avons juste donné la requête SQL sans préciser la syntaxe réelle concernant le chemin global de la méta-table *RP*. Tous ces détails seront décrits par la suite.

SELECT COUNT(*) FROM RP WHERE TAB='T'

- Il compare *N* et *N'*.
  - Si *N*<*N'*, l'image *T* ne correspond pas au partitionnement actuel de sa table scalable *T*. L'image *T* doit être alors ajustée en rajoutant les segments qui manquent dans sa définition. Il récupère ces segments à partir de la méta-table *RP* aussi. Ensuite, il modifie la définition de la vue partitionnée représentant l'image *T* en exécutant la requête SQL *ALTER VIEW*.
  - Si *N*=*N'*, l'image *T* correspond bien au partitionnement de sa table sclabale *T*. Autrement dit, elle représente dans sa définition tous les segments de la table scalable *T*.
- Une fois l'image *T* ajustée, SD-SQL Server exécute la requête qui l'interroge.

4.6 Conclusion

Tout au long de ce chapitre nous avons décrit l'architecture du système de gestion de bases de données distribuées et scalable SD-SQL Server que nous avons proposé. Nous avons présenté les règles structurelles qui permettent de concevoir une telle architecture. Nous avons ensuite décrit en détail les différents composants du système SD-SQL Server. Le chapitre suivant décrit l'interface qui permet d'accéder au système SD-SQL Server.

5 INTERFACE D'APPLICATION SD-SQL SERVER

5.1 Introduction

Comme tout SGBD, SD-SQL Server permet de décrire ses données, de les interroger, de les mettre à jour, de transformer des représentations de données d'assurer les contrôles d'intégrité et de concurrence. Toutes ces fonctionnalités sont illustrées à travers son interface d'application. Dans ce chapitre, nous décrivons les différentes opérations qu'offre l'interface d'application SD-SQL Server. Nous décrivons en détail les commandes SD-SQL Server tout en les illustrant par des exemples. Nous les avons organisées selon leurs fonctionnalités qui se résument dans la gestion des nœuds SD-SQL Server, des SDBs, des tables scalables, des images et enfin des requêtes scalables.

5.2 Préliminaires

Les applications/utilisateurs peuvent manipuler les tables scalables en utilisant des commandes spécifiques au système SD-SQL Server que nous avons appelées *commandes SD-SQL Server* ou *requêtes (commande) scalables*. Comme nous l'avons déjà mentionné, les commandes SD-SQL Server interrogent les tables scalables à travers leurs images. La plupart de ces commandes que nous avons définies peuvent être exécutées aussi sur des tables ou vues statiques. Ces dernières représentent les tables et les vues habituelles de tout SGBD.

Les commandes SD-SQL Server exécutent des manipulations usuelles du langage SQL. Ce qui les différencie des commandes SQL habituelles est la manipulation des tables scalables. De plus, une commande scalable peut inclure des paramètres additionnels spécifiques à l'environnement scalable de l'architecture SD-SQL Server (capacité d'un segment, clé de partitionnement, etc.) [SLS05].

La règle que nous avons appliquée pour nommer une commande SD-SQL Server est basée sur l'utilisation des noms des commandes SQL traditionnelles. Nous préfixons celles-ci par le mot *'sd_'* (*scalable distributed)* pour désigner la scalabilité et la distribution de nos commandes. S'il y a des blancs dans une requête SQL, nous remplaçons ces blancs par le caractère *'_'*. Ainsi, la requête SQL *SELECT,* par exemple, sera représentée sur notre système par la commande *sd_select*. Quant à la requête SQL *CREATE TABLE*, par exemple, elle sera représentée par la commande *sd_create_table*.

Les informations spécifiques aux données scalables ont été présentées comme des paramètres d'entrée dans les commandes SD-SQL Server. Ces paramètres représentent généralement les clauses SQL qui correspondent aux commandes SD-SQL Server. Par exemple, la commande SD-SQL Server *sd_select* doit avoir comme paramètre la suite de la clause SQL *SELECT*. En effet, *sd_select* interroge une table scalable, mais comme une table scalable est constituée de segments qui sont des tables du SGBD SQL Server, le résultat de l'exécution de la commande *sd_select* est donc celui de l'exécution de la requête *SELECT*. Ainsi, c'est cette dernière qui sera exécutée par le SGBD qui supporte notre système. Ces paramètres de *requêtes scalables* ne contiennent pas tous la syntaxe d'une requête SQL. Nous présentons les détails sur ces paramètres par la suite.

Nous avons réparti les commandes SD-SQL Server selon leur usage. Nous avons défini les commandes concernant les manipulations suivantes [SLS06]:

- La gestion des nœuds de stockage SD-SQL Server.
- La gestion des bases de données scalables.
- La gestion des tables scalables.
- La gestion des requêtes scalables.

Dans ce qui suit, nous décrivons en détail chacune de ces catégories de commandes scalables tout en discutant leur syntaxe et leur sémantique. Nous illustrons ces commandes par des exemples utilisant la base de test (ou *benchmark)* de la base de données *SkyServer*. Nous présentons d'abord ce *benchmark*.

5.3 Description de la Base de Test *SkyServer*

SkyServer est une base qui permet un accès aux données du *Sloan Digital Sky Survey* (SDSS) pour les astronautes et l'éducation scientifique. SDSS est une vue générale du ciel du nord (*Northern*) à une résolution d'environ ½ arc-second en utilisant un télescope moderne basé sur la terre. Il caractérise environ 200 Méga d'objets dans cinq bandes optimales et mesure le spectre d'un million d'objets [ASG+02].

Le traitement SDSS est en cours de réalisation dans le laboratoire *Fermilab* [ASG+02]. Il examine les images à partir du télescope et identifie les objets photo comme étant des étoiles, des galaxies, etc. La taille de ces données est de 80GB contenant environ 14 millions objets et cinquante milles spectres [G02]. Ces données sont accessibles via *SkyServer* (http://skyserver.sdss.org/). SDSS a initialement développé la base de données *SkyServer* sur l'outil *ObjectivityDB*. Ensuite, cette conception sur *ObjectivityDB* a été transformée en un schéma relationnel. La Figure 5-1 suivante présente ce schéma.

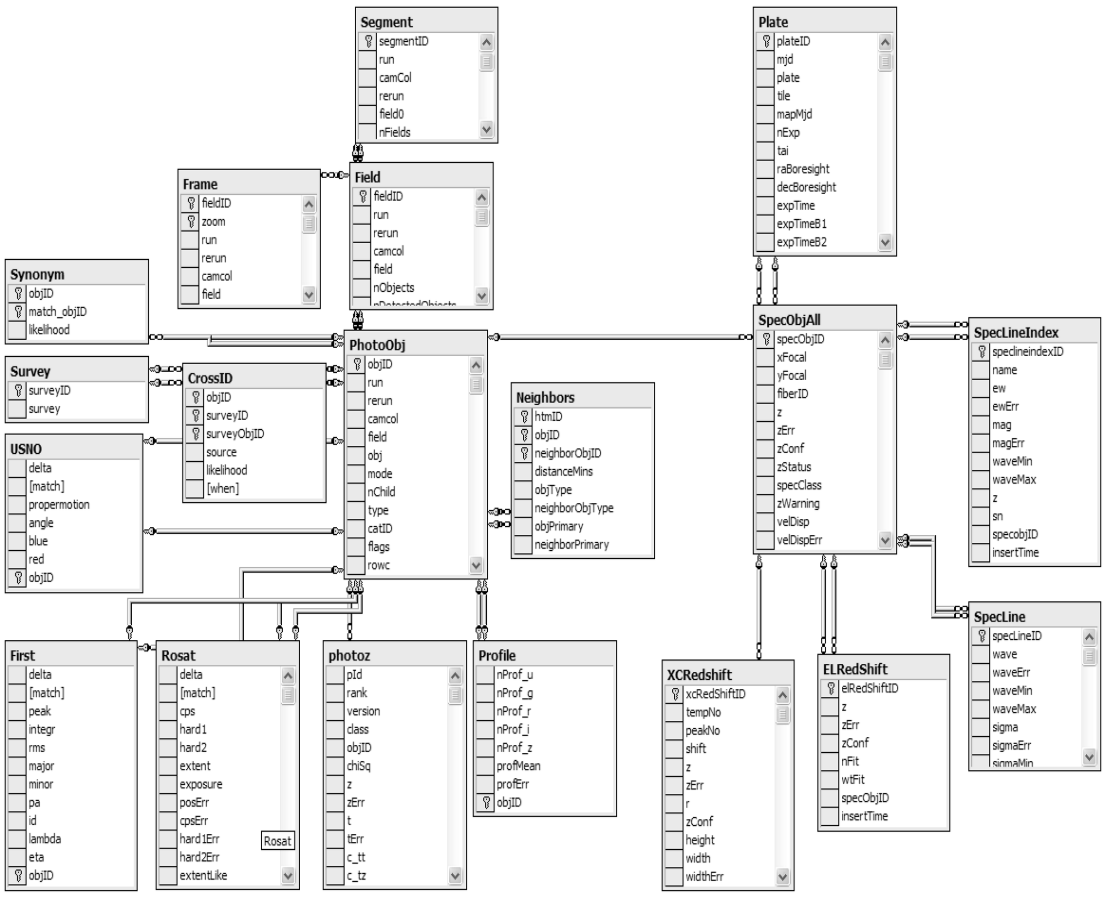

Figure 5-1 : Schéma logique de la base de données *SkyServer*

5.4 Gestion des Nœuds SD-SQL Server

Pour la gestion des différents nœuds serveur, client et pair, SD-SQL Server permet les manipulations suivantes :

- Création d'un nœud.
- Modification du type d'un nœud.

- Suppression d'un nœud.

Ces manipulations sont propres au système SD-SQL Server uniquement. La description de chacune d'elles est comme suit :

5.4.1 Création d'un Nœud SD-SQL Server

Une application peut créer des nœuds de type serveur, client ou pair, dans un système SD-SQL Server, en exécutant la commande suivante :

sd_create_node 'new_node' [, node_type]'

Cette commande a deux paramètres en entrée :

- Le paramètre *'new_node'* correspond au nom du nœud que l'application veut créer. Les noms des nœuds doivent représenter des serveurs SQL liés pour pouvoir les utiliser en tant que nœuds SD-SQL Server.
- Le paramètre *'node_type'* correspond au type du nœud à créer. Ce paramètre peut avoir la valeur *'client'*, *'serveur'* ou *'pair'*. Il peut être aussi optionnel. Si aucune valeur n'est entrée dans le paramètre *'node_type'*, le type qui sera pris par défaut sera serveur.

Remarque

Notons que pour pouvoir utiliser les commandes SD-SQL Server, celles-ci doivent exister déjà sur la MDB. Nous rappelons que la MDB est une base système qui permet d'initialiser tous les nœuds du système SD-SQL Server et ses SDBs. Elle est créée sur le nœud primaire du système en exécutant un script SQL. Le nœud primaire est de type serveur ou pair. Il est initialisé par défaut par l'administrateur du système. Nous supposons que ce nœud est *Node1,* de type pair, dans tous les exemples qui illustrent les commandes SD-SQL Server. Les détails sur l'initialisation du système SD-SQL Server seront décrits dans le chapitre suivant.

Exemple

Soit *Node1* un nœud primaire (de type pair) du système SD-SQL Server. Nous utilisons ce nœud pour créer de nouveaux nœuds *Node2*, *Node3* et *Node4*, en exécutant respectivement les commandes suivantes :

sd_create_node 'Node2'

sd_create_node 'Node3', 'client'

sd_create_node 'Node4', 'peer'

Le nœud SD-SQL Server *Node2* sera pris comme un nœud de type serveur par défaut puisque aucun type n'a été indiqué pour sa création. Les nœuds *Node3* et *Node4* sont respectivement de type client et pair comme les commandes l'indiquent.

5.4.2 Modification d'un Nœud

Une application peut modifier un nœud SD-SQL Server en modifiant son type. On peut transformer un nœud de type client (ou serveur) en un nœud de type pair. Elle peut aussi réduire les fonctionnalités d'un nœud pair en le transformant en un nœud client (ou serveur). La modification des types des nœuds SD-SQL Server est réalisée par la commande suivante :

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

La commande *sd_alter_node* a deux paramètres en entrée :

- Le paramètre *'node_name'* indique le nom du nœud dont l'application veut modifier le type.
- Le paramètre *'ADD/DROP client/server'* indique en quel type le nœud en question sera transformé. En combinant *ADD/DROP* et *client/server*, ce paramètre peut avoir l'une des valeurs suivantes :
    - *'ADD server'* si une application veut rajouter à un nœud client les fonctionnalités d'un nœud serveur, c'est-à-dire transformer ce nœud client en un nœud pair.
    - *'DROP server'* si une application veut supprimer à un nœud pair ses fonctionnalités de serveur, c'est-à-dire le transformer en un nœud client.
    - *'ADD client'* si une application veut rajouter à un nœud serveur les fonctionnalités d'un nœud client, c'est-à-dire le transformer en un nœud pair.
    - *'DROP client'* si une application veut supprimer à un nœud pair ses fonctionnalités de client, c'est-à-dire le transformer en un nœud serveur.

Toute autre combinaison entre *ADD/DROP* et *client/server* pour transformer d'autres types de nœuds que ceux que nous venons de décrire n'est pas possible. Par exemple, une application ne peut entrer la valeur *'ADD server'* pour un nœud de type pair parce que ce dernier a déjà les fonctionnalités d'un serveur, etc.

Exemple

Soit le nœud *Node200* de type client. La commande ci-dessous transforme le nœud client *Node200* en un nœud pair. Le nœud *Node200* peut alors se comporter comme un nœud client et serveur en même temps.

sd_alter_node 'Node200', 'ADD server'

Quant à la commande suivante, elle transforme le nouveau nœud pair *Node200* en un nœud serveur.

sd_alter_node 'Node200','DROP client'

5.4.3 Suppression d'un Nœud

Une application peut supprimer un nœud SD-SQL Server en exécutant la commande scalable *sd_drop_node* comme suit :

sd_drop_ node 'node_name'

Le résultat de cette commande est un système SD-SQL Server sans le nœud *'node_name'*. La suppression d'un nœud de type client ou pair supprime toutes les tables scalables qu'il a créé. Les nœuds supprimés de l'architecture SD-SQL Server deviennent en effet des nœuds libres représentant leur SGBD comme à l'origine.

Exemple

La commande suivante supprime le nœud *Node200* de l'architecture SD-SQL Server.

sd_drop_node 'Node200'

5.5 Gestion des Bases de Données Scalables

La gestion des SDBs consiste à créer une nouvelle SDB, ajouter ou supprimer des NDBs dans une SDB et enfin supprimer une SDB. Nous rappelons qu'une SDB est une collection de NDBs dont une NDB primaire. Le nombre de NDBs dans une SDB peut s'étendre quand ses tables scalables se partitionnent dynamiquement.

Les commandes sur la gestion des SDBs ne concernent que le système SD-SQL Server. Nous décrivons leur syntaxe et leur sémantique dans ce qui suit.

5.5.1 Création d'une Base de Données Scalable

La création d'une SDB est réalisée par l'administrateur du système SD-SQL Server (SD-DBA). Cette opération permet à un SD-DBA de créer la NDB primaire d'une SDB. La commande qui réalise cette opération est décrite par la syntaxe suivante :

sd_create_scalable_database 'sdb_name' ['node_name'] ['type'] ['extent']

Cette commande dispose des paramètres suivants en entrée :

- Le paramètre *'sdb_name'* indique le nom de la SDB à créer. Il s'agit du nom de toutes ses NDBs aussi y compris la NDB primaire.
- Le paramètre *'node_name'* correspond au nom du nœud sur lequel la NDB primaire de la SDB sera créée. Ce paramètre est optionnel. S'il n'est pas entré dans la commande, la SDB sera alors créée sur le nœud de l'exécution de la commande.
- Le paramètre *'extent'* est optionnel. Il indique le nombre d'extensions (NDBs) qu'une application veut créer pour la SDB. Il permet donc la création simultanée de *n* (n>1) NDBs dont principalement la NDB primaire. Si le paramètre *'extent'* n'est pas précisé, une seule extension de la SDB sera créée (sa NDB primaire).
- Le paramètre *'type'* est un paramètre optionnel. Il indique si la NDB primaire de la SDB à créer est de type client, serveur ou pair. Par défaut, la NDB primaire hérite le même type du nœud SD-SQL Server sur lequel elle sera créée, c'est-à-dire le type du nœud *'node_name'*. Par exemple, si le de la SDB est de type serveur, donc la NDB primaire créée sera de type serveur, etc. En conséquence, la commande *sd_create_scalable_database* ne peut pas créer une NDB de type client sur un nœud serveur ou vice versa.

Exemple

La commande suivante crée une NDB primaire pour une nouvelle SDB nommée *SkyServer* sur le nœud *Node1* qui est de type pair. Comme le type de la NDB primaire *SkyServer* n'est pas mentionné dans les paramètres de la commande, donc cette NDB aura le même type que son nœud *Node1*, c'est-à-dire pair.

sd_create_scalable_database 'SkyServer', 'Node1'

La commande qui suit crée la NDB primaire d'une nouvelle SDB nommée *SkyServer1* sur le nœud *Node1*. Comme *Node1* est de type pair, et la NDB primaire *SkyServer1* est déclarée de type client dans la commande, donc *SkyServer1* peut être créée sur *Node1* et elle jouera le rôle d'un client.

sd_create_scalable_database 'SkyServer1', 'Node1', 'Client'

Pour la commande ci-dessous, la création de la SDB *SkyServer2* sera annulée. Ceci est dû au fait que sa NDB primaire est déclarée de type client et le nœud *Node2* où elle sera créée est de type serveur par exemple. Un conflit est alors généré au niveau de la hiérarchie des clients, serveurs et pairs.

sd_create_scalable_database 'SkyServer2', 'Node2', 'Client'

5.5.2 Modification d'une Base de Données Scalable

La modification d'une SDB consiste à modifier le nombre de ses NDBs. Il s'agit de créer de nouvelles NDBs pour cette SDB ou de supprimer des NDBs qui existent déjà. Nous décrivons ces deux opérations dans ce qui suit.

5.5.2.1 Création d'une NDB

Une application peut créer une nouvelle NDB dans une SDB en exécutant la commande suivante :

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

Le résultat de l'exécution de cette commande est une nouvelle NDB dans la base SDB nommée *'sdb_name'*. Cette NDB aura bien sûr le même nom que la SDB, c'est-à-dire *'sdb_name'*. Elle est créée sur le nœud *'node_name'*. Les paramètres de la commande *sd_create_node_database* sont :

- Le paramètre *'sdb_name'* indique le nom de la SDB pour laquelle une nouvelle NDB sera créée.
- Le paramètre *'node_name'* est optionnel. S'il n'est pas indiqué dans la commande, la nouvelle NDB sera créée sur le nœud courant de l'exécution de la commande, c'est-à-dire cette NDB sera de type client ou pair. Dans le cas où ce nœud ne correspond pas au type de la NDB à créer, SD-SQL Server sélectionnera alors un nœud disponible de la méta-base MDB.
- Le paramètre *'type'* indique le type de la NDB à créer. C'est un paramètre optionnel. S'il n'est pas indiqué dans la commande, le NDB en question hérite le type du nœud qui va la détenir.

Exemple

Le résultat de la commande ci-dessous sera la création d'une nouvelle NDB sur le nœud *Node2* pour la SDB nommée *SkyServer*. Puisque le type de la NDB *SkyServer* n'est pas précisé dans la commande, elle aura alors le type du nœud *Node2* sur lequel elle sera créée. Le type du nœud *Node2* est serveur d'après nos exemples précédent. Ainsi, la NDB *SkyServer* sera de type serveur.

sd_create_node_database 'SkyServer', 'Node2'

Soit le nœud *Node4* de type pair. La commande qui suit, exécutée sur le noeud *Node4*, permet de créer une nouvelle NDB pour la SDB *SkyServer*. Comme le nœud, sur lequel cette NDB sera créée, n'est pas mentionné dans la commande, la NDB *SkyServer* sera alors créée sur le nœud courant de l'exécution de la commande, c'est-à-dire *Node4*. La NDB *SkyServer* sera donc de type pair tout comme le type du nœud *Node4*.

sd_create_node_database 'SkyServer', 'client'

La commande suivante annule la création d'une nouvelle NDB pour la SDB *SkyServer* sur le noeud *Node1*. Cette création est annulée parce qu'il existe déjà une NDB de la SDB *SkyServer* (la NDB primaire) sur ce nœud *Node1*. Celle-ci constitue la NDB primaire de la SDB *SkyServer*, elle a été créée dans l'exemple précédent (cf. Section 5.5.1).

sd_create_node_database 'SkyServer', 'Node1'

5.5.2.2 Suppression d'une NDB

Une application peut supprimer une NDB d'une SDB en exécutant la commande *sd_drop_node_database* selon la syntaxe suivante :

sd_drop_node_database 'sdb_name', 'node_name'

Cette commande possède deux paramètres en entrée. Il s'agit de :

- *'sdb_name'* qui indique le nom de la SDB de laquelle une NDB sera supprimée ;
- *'node_name'* qui indique le nom du nœud où se trouve la NDB à supprimer.

Avant de supprimer la NDB *'sdb_name'* (elle a le même nom que sa SDB) du nœud *'node_name'*, SD-SQL Server s'assure que toutes les données et métadonnées ont été transférées vers d'autres NDBs de la même SDB *'sdb_name'*. Ceci évite la perte des données dépendant des autres nœuds.

Exemple

La commande suivante supprime la NDB localisée sur le nœud *Node2* de sa SDB nommée *SkyServer*.

sd_drop_node_database 'SkyServer', 'Node2'

5.5.3 Suppression d'une Base de Données Scalable

La suppression d'une SDB est assurée par le SD-DBA. Cette opération permet de supprimer toutes les NDBs d'une base scalable avec toutes les métadonnées correspondantes. Elle est réalisée en exécutant la commande suivante :

sd_drop_scalable_database 'sdb_name'

Exemple

Soit la SDB *SkyServer* composée de trois NDBs localisées sur *Node1*, *Node2* et *Node3* respectivement. La commande suivante supprime la SDB *SkyServer*. Ainsi toutes ses NDBs sur *Node1*, *Node2* et *Node3* seront supprimées avec toutes leurs données.

sd_drop_scalable_database 'SkyServer'

Après avoir décrit les commandes concernant la gestion des nœuds, des SDBs et des NDBs dans un système SD-SQL Server, nous passons maintenant à la description des tables scalables, des images et des requêtes. Avant cela, notons que les commandes que nous venons de voir ne peuvent être exécutées que sur notre système SD-SQL Server. Elles ne peuvent concerner d'autres architectures de SGBDs. Par contre, les commandes des sections suivantes peuvent concerner des objets du SGBD SQL Server comme nous allons le voir. En effet, ces commandes sont les requêtes SQL habituelles adaptées à un environnement distribué et scalable.

5.6 Gestion des Tables Scalables

La gestion des tables scalables inclut toute opération effectuée sur le schéma d'une table scalable. Ces opérations se résument en :

- la création d'une table scalable ;
- la modification d'une table scalable ;
- la gestion de ses index ;
- et la suppression des tables scalables.

5.6.1 Création d'une Table Scalable

Soit une application s'exécutant sur une NDB de type client (ou pair) du système SD-SQL Server. Cette application peut créer une table scalable *T* en exécutant la commande scalable *sd_create_table* selon la syntaxe suivante :

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

Cette commande est définie par trois paramètres :

- Le paramètre *'SQL: Create Table T clauses'* affecte le texte de la requête SQL *CREATE TABLE*. Le nom de la table scalable à créer peut être un nom global dans ce paramètre. Autrement dit, le nom de la table peut être préfixé par le nom du nœud et celui de la NDB où la table sera créé (son chemin). Ce paramètre possède les mêmes propriétés que la requête SQL traditionnelle *CREATE TABLE*. Cependant, il y a quelques propriétés qui sont propres à SD-SQL Server et qui doivent être vérifiées. Il s'agit de la définition d'une clé primaire pour chaque table scalable créée. La clé primaire peut être définie sur un ou plusieurs attributs de la table scalable. Elle servira après pour la définition de la clé de partitionnement et des contraintes d'intégrité lors du partitionnement de la table scalable.

- Le paramètre *'Segment_size'* définit la capacité de la table scalable à créer. Il s'agit de sa taille maximale en nombre de tuples. Cette taille sera sauvegardée dans la méta-table *Size*, (cf. Section 4.4.2.2). Elle sera vérifiée lors de chaque insertion dans une table scalable. Si une table scalable excède cette capacité, elle éclatera.
- Le paramètre *'Partitioning_key'* est un paramètre optionnel. Il indique un attribut clé parmi les attributs constituant la clé primaire de la table scalable *T*. En effet, c'est sur cet attribut clé que la table *T* sera partitionnée par la suite. S'il n'est pas entré dans la commande *sd_create_table*, SD-SQL Server partitionne la table *T* sur un de ses attributs clé. Nous décrirons le choix de cet attribut dans le prochain chapitre.

Exemple

Nous supposons que la SDB *SkyServer* créée dans les exemples précédents n'a pas été supprimée. Soit sa NDB *SkyServer* du nœud *Node1* sur laquelle nous créons la table scalable *PhotoObj*. Le nom de cette table est tiré du *benchmark SkyServer*. Supposons que la table scalable *PhotoObj* a une capacité de 10000 tuples, la requête scalable de sa création est donc exécutée comme suit:

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

Le résultat de cette commande sera alors la table *PhotoObj* ayant la clé primaire *Objid*, et la capacité de 10000 tuples.

Notons que le paramètre correspondant à la clause SQL *CREATE TABLE* ne comporte pas le verbe (ou mot clé) *CREATE TABLE,* puisque celui-ci apparaît dans le nom de la commande *sd_create_table*. Nous utilisons le même principe pour toutes les requêtes scalables suivantes.

5.6.2 Modification d'une Table Scalable

Pour de modifier le schéma d'une table scalable *T*, l'application exécute la commande *sd_alter_table* selon la syntaxe suivante :

```
sd_alter_table 'SQL : ALTER TABLE T clauses' ['Segment_Size']
```

La commande *sd_alter_table* joue le même rôle que la requête SQL *ALTER TABLE* mais sur les tables scalables. De plus, la commande *sd_alter_table* peut modifier la capacité d'une table scalable qui était déjà définie lors de sa création. Pour cela, cette commande dispose de deux paramètres en entrée. Il s'agit de :

- '*SQL : ALTER TABLE T clauses'* qui correspond à la clause SQL usuelle *ALTER TABLE*. Toutes les modifications que permet *ALTER TABLE* sont possibles sur les tables scalables sauf pour les contraintes d'intégrité. Ces dernières définissent le

partitionnement d'une table scalable donc elles ne peuvent être modifiées que par le système lui même lors d'un éventuel éclatement d'un segment d'une table scalable.

- Le paramètre *'Segment_Size'* est optionnel. Il correspond à la nouvelle capacité d'une table scalable. *'Segment_Size'* remplace l'ancienne taille maximale de cette table scalable sauvegardée dans la méta-table *Size*.

Exemple

Soit la table scalable *PhotoObj* que nous avons créée dans l'exemple précédent. Si un utilisateur veut modifier cette table scalable en lui rajoutant une nouvelle colonne *t* de type entier, il exécute alors la commande *sd_alter_table* comme suit :

```
sd_alter_table 'PhotoObj ADD t INT'
```

Le résultat de l'exécution de cette commande est donc la table scalable *PhotoObj* avec une nouvelle colonne *t* de type entier au niveau de chacun de ses segments distribués.

Si cet utilisateur veut modifier aussi la capacité de la table scalable *PhotoObj* pour la rendre à 20000 tuples, il exécute alors la commande s*d_alter_table* comme suit :

```
sd_alter_table 'PhotoObj', 20000
```

Le résultat de l'exécution de cette commande est la modification de la capacité de *PhotoObj* à 20000 tuples dans la méta-table *Size*.

5.6.3 Suppression d'une Table Scalable

Pour supprimer une table scalable *T*, l'application utilise la commande *sd_drop_table* selon la syntaxe suivante :

```
sd_drop_table 'SQL : DROP TABLE T clauses'
```

Cette commande supprime tout ce qui est lié à la table scalable *T*. Elle supprime ses segments, ses images et toutes ses métadonnées. Son paramètre d'entrée est défini par la clause SQL *DROP TABLE*.

Exemple

La commande suivante supprime la table scalable *PhotoObj* créée précédemment.

```
sd_drop_table 'PhotoObj'
```

Le résultat de cette commande est un système SD-SQL Server sans aucune trace de la table scalable *PhotoObj*.

5.6.4 Les Index

Une application/utilisateur peut manipuler des index sur une table scalable. Nous avons appelé un index d'une table scalable *'un index scalable'*. Un index scalable se propage alors sur tous les segments d'une table scalable. L'interface d'application SD-SQL Server permet d'ajouter ou de supprimer des index pour une table scalable tout comme n'importe quelle autre table statique. Nous décrivons dans ce qui suit les commandes SD-SQL Server gérant les index scalables.

- **Création d'un index**

Une application peut créer un index distribué et scalable *I* sur une colonne d'une table scalable *T*, en exécutant la commande *sd_create_index* selon la syntaxe suivante :

sd_create_index 'SQL: CREATE INDEX I ON T clauses'

Le résultat de cette commande est la création de l'index *I* sur tous les segments distribués de la table scalable *T*. Cette commande crée un index scalable en utilisant son paramètre en entrée défini comme suit :

- Le paramètre '*SQL : CREATE INDEX I ON T clauses'* correspond à la clause SQL *CREATE INDEX*.

Exemple

La commande suivante, exécutée sur le nœud pair *Node1*, crée l'index scalable *run_index* sur la colonne *run* de la table scalable *PhotoObj* :

sd_create_index 'run_index ON Photoobj (run)'

- **Suppression d'un index**

Une application supprime un index scalable *I* de la table *T* en exécutant la commande *sd_drop_index* selon la syntaxe suivante :

sd_drop_index 'SQL : DROP INDEX T.I clauses'

Le paramètre en entrée de la commande est la clause SQL *DROP INDEX*. De même que la création d'un index scalable, la suppression d'un index est propagée sur tous les segments qui composent la table scalable.

Exemple

La commande suivante, exécutée sur la NDB *SkyServer* du nœud *Node1*, supprime l'index *run_index* de la table scalable *PhotoObj* :

sd_drop_index 'PhotoObj.run_index'

5.7 Gestion des Images Secondaires

Comme nous l'avons déjà présenté, une application/utilisateur accède à une table scalable à travers ses images (primaire ou secondaires). Pour cela, la gestion des images est nécessaire. L'interface d'application de SD-SQL Server permet la création et la suppression des images secondaires seulement. Les images primaires ne peuvent être manipulées par les applications. Elles sont créées et supprimées avec leurs tables scalables. Nous décrivons ces deux opérations dans les sections suivantes.

5.7.1 Création d'une Image

Une application peut créer des images secondaires sur des NDBs de type client (pair) en exécutant la commande *sd_create_image* selon la syntaxe suivante :

```
sd_create_image 'Primary_node', 'Table'
```

Cette commande possède deux paramètres :

- *'[Primary_node]'* indique le nœud de la NDB client qui détient l'image primaire correspondant à l'image secondaire à créer. Ce paramètre nous aide à composer le nom de l'image secondaire en question.
- *'Table'* indique le nom de la table scalable pour laquelle on veut créer une image secondaire.

Sur chaque NDB client, il n'existe qu'une seule image (primaire ou secondaire) d'une table scalable donnée. Si une application veut utiliser un autre nom pour les images secondaires, elle utilise alors la requête SQL *CREATE VIEW*.

Exemple

Nous supposons que la table scalable *PhotoObj* n'est pas encore supprimée. Une application peut créer une image secondaire de la table *PhotoObj* en exécutant la commande suivante sur la NDB *Node4.SkyServer* par exemple :

```
sd_create_image 'Node1', 'PhotoObj'
```

Node1 est le nom du nœud de la NDB *SkyServer* qui détient l'image primaire. Le résultat de la commande est une image secondaire sur la NDB de l'exécution de la commande.

5.7.2 Suppression d'une Image Secondaire

La suppression des images concerne seulement les images secondaires. Les images primaires ne peuvent pas être supprimées sinon l'interrogation des tables scalables

devient impossible. Une application peut supprimer une image secondaire en exécutant la commande suivante :

sd_drop_image 'image_name'

Le paramètre *'image_name'* peut correspondre à une image locale (sur la même NDB de l'exécution de la commande), comme il peut correspondre à une image sur une autre NDB. Dans ce dernier cas, *'image_name'* contiendra le nom global de l'image pour indiquer la NDB où elle est localisée.

Exemple

La commande suivante, exécutée sur la NDB *Node4.SkyServer,* supprime l'image secondaire de la table scalable *PhotoObj* se trouvant sur la NDB courante de l'exécution de la commande.

sd_drop_image 'PhototObj'

Si cette commande n'est pas exécutée sur la NDB *Node4.SkyServer* (celle qui détient l'image secondaire à supprimer), il faut donc la préfixer par sa localisation complète (nœud et NDB).

5.8 Gestion des Requêtes Scalables

La gestion des requêtes d'accès aux tables scalables par SD-SQL Server concerne toutes les requêtes de sélection, d'insertion, de mise à jour et de suppression. Ces requêtes jouent le même rôle que les requêtes SQL. Elles s'exécutent sur des tables scalables à travers leurs images. Elles peuvent aussi être exécutées sur des tables statiques. Nous décrivons dans les sections suivantes les commandes SD-SQL Server qui servent de requêtes de recherche et de mise à jour (insertion, modification et suppression) des tables scalables.

5.8.1 La Recherche

Une application peut exécuter une requête de sélection sur des tables scalables en utilisant la commande *sd_select*, sur sa NDB client, selon la syntaxe suivante :

sd_select 'SQL: SELECT clauses', 'Segment_size', 'Primary_key'

Cette commande joue le même rôle que la requête SQL *SELECT*. Elle dispose des paramètres en entrée suivants :

- Le paramètre *'SQL: SELECT clauses'* est la requête SQL *SELECT* avec sa syntaxe usuelle (mais sans le mot clé *SELECT*). Ce paramètre peut contenir des requêtes avec des agrégations, des jointures, des alias, des sous requêtes, etc.

- Les paramètres *'Segment_size'* et *'Primary_key',* sont des paramètres optionnels. Ils sont utilisés dans la commande *sd_select* lorsqu'elle exécute une requête *SELECT INTO*. L'utilisation d'une requête *SELECT INTO* sur SD-SQL Server permet de créer une table scalable à partir d'une table statique.
    - Le paramètre *Segment_size'* indique la capacité de la table statique que l'application souhaite transformer en une table scalable.
    - Le paramètre *'Primary_key'* indique la colonne qui sera définie comme clé primaire et qui servira par la suite dans le partitionnement de la table.

Exemple

L'utilisateur peut exécuter une requête de sélection sur une table scalable comme suit :

sd_select '* FROM PhototObj'

Le résultat de cette commande est l'exécution de la requête SQL :

'SELECT * FROM PhototObj'

La commande suivante crée une nouvelle table scalable, *S_PhotoObj*, à partir de la table statique *PhotoObj*. La capacité de la nouvelle table scalable *S_PhotoObj* est 500 tuples. Quant à sa clé primaire, elle est définie sur la colonne *Objid.*

sd_select 'INTO S_PhotoObj * FROM PhototObj', 500, 'Objid'

5.8.2 L'Insertion

Une application peut effectuer une insertion dans une table scalable en exécutant la commande *sd_insert* selon la syntaxe suivante :

sd_insert 'SQL: INSERT clauses'

Cette commande peut être exécutée sur une table statique ou une table scalable. Son paramètre d'entrée '*SQL : INSERT clauses'* correspond à la commande SQL usuelle *INSERT*. Une application peut utiliser les clauses *SELECT* dans la requête *INSERT*.

Exemple

La commande suivante permet d'insérer le tuple *(2255031257923860)* dans la colonne *Objid* de la table scalable *PhotoObj* :

sd_insert 'INTO PhotoObj (objid) values (2255031257923860)'

La commande *sd_insert* qui suit permet d'insérer des tuples dans la table scalable *PhotoObj*, se trouvant sur le nœud *Peer1*, à partir de la table statique *PhotoObj* se trouvant sur le nœud *Peer5*.

```
sd_insert 'INTO PhotoObj SELECT * FROM Peer5.PhotoObj
        WHERE objid NOT EXISTS IN (SELECT objid FROM PhotoObj)'
```

5.8.3 La Mise à Jour

Une application met à jour une table scalable en exécutant la commande *sd_update* comme suit :

```
sd_update ' SQL: Update clauses'
```

Le paramètre d'entrée est la commande SQL usuelle *UPDATE* qui peut contenir, éventuellement, la clause *SELECT*. La mise à jour des données d'une table scalable doit respecter les contraintes d'intégrité précédemment définies sur cette table.

Exemple

La commande suivante met à jour la valeur de la colonne *run* de la table scalable *PhotoObj* pour le tuple dont la clé *Objid* est égale à *2255031257923860.*

```
sd_update 'PhotoObj set run= 752 where objid=2255031257923860'
```

Quant à la commande ci-dessous, elle effectue une mise à jour de la table scalable *PhotoObj* en utilisant la clause *SELECT*. Elle modifie la valeur de la colonne *run* à *752* pour les dix premiers tuples de la table.

```
sd_update 'PhotoObj set run= 752 where Objid in
        (SELECT TOP 10 Objid FROM PhotoObj)'
```

5.8.4 La Suppression

Une application peut supprimer des tuples d'une table scalable en utilisant la commande suivante :

```
sd_delete 'SQL: Delete clauses'
```

Le paramètre en entrée de cette commande est la clause SQL *DELETE*. L'utilisation de cette commande doit respecter les contraintes d'intégrité déjà définies.

Exemple

La commande suivante supprime de la table scalable *PhotoObj* le tuple dont la clé *Objid* est égale à *2255031257923860*.

```
sd_delete 'FROM PhotoObj WHERE objid=2255031257923860'
```

5.9 Conclusion

Dans ce chapitre, nous avons décrit toutes les commandes de l'interface d'application du système SD-SQL Server, tout en illustrant leur utilisation par des exemples. Le chapitre suivant présentera l'implantation du prototype SD-SQL Server. Nous détaillerons aussi, dans le chapitre suivant, les traitements internes de chacune des commandes SD-SQL Server. Nous discuterons également la gestion de concurrence de ces commandes.

6 LE PROTOTYPE SD-SQL SERVER

6.1 Introduction

Ce chapitre décrit la mise en œuvre de notre prototype SD-SQL Server. Il décrit la structure et le traitement internes des différents composants fonctionnels de l'architecture. Nous commençons par justifier nos choix techniques dans l'implémentation de SD-SQL Server. Ensuite, nous décrivons l'architecture fonctionnelle de SD-SQL Server. Par la suite, nous décrivons en détail la structure et le traitement internes de chaque composant de l'architecture ainsi que toutes ses fonctionnalités. Nous illustrons ces traitements en utilisant la base de test *SkyServer* déjà présentée dans le chapitre précédent.

6.2 Choix Techniques

Nous présentons dans cette section nos choix techniques dans l'implantation de SD-SQL Server. Nous justifions notre choix pour l'utilisation du SGBD SQL Server dans l'implantation de notre système tout en présentant une correspondance entre les composants des deux systèmes. Afin d'éviter quelques restrictions posées par SQL Server, nous avons proposé des mécanismes propres à SD-SQL Server. Il s'agit en particulier de la gestion des rôles des utilisateurs du système SD-SQL Server ainsi que les conventions sur les noms clés des objets utilisés.

6.2.1 Correspondance entre SD-SQL Server et SQL Server

Comme nous l'avons déjà mentionné, nous avons utilisé le SGBD Microsoft SQL Server 2000 pour l'implantation de notre SD-DBS. Nous avons superposé l'architecture du système SD-SQL Server sur l'architecture d'un ensemble d'instances SQL Server liées. SD-SQL Server constitue alors une couche scalable et distribuée sur le SGBD SQL Server.

Afin de mieux illustrer les différents composants de notre système SD-SQL Server, nous avons proposé de faire correspondre ses composants avec les composants classiques du SGBD SQL Server. Cette correspondance est valable avec tout autre SGBD étant donné que ce sont les mêmes éléments de base sur tous les SGBDs. La table suivante résume la présentation des objets SD-SQL Server sur SQL Server.

SD-SQL Server	SQL Server
Serveur lié	Nœud SD-SQL Server
Base de données	NDB
Procédure Stockée	Commande SD-SQL Server
Tables Système	Méta-tables
Table	Segment
Vue Partitionné	Image

Table 6-1 : Correspondance entre SD-SQL Server et SQL Server

Nous avons choisi SQL Server parmi tous les autres SGBDs actuels pour la raison principale suivante : SQL Server est le seul SGBD qui permet la modification des vues partitionnées et distribuées. En effet, nous utilisons les vues partitionnées pour représenter les images des tables scalables. Comme une table scalable peut s'étendre en nombre de segments, son image doit alors correspondre à son partitionnement actuel. Par conséquent, il faut pouvoir modifier la définition de la vue partitionnée représentant cette image.

6.2.2 Gestion des Utilisateurs

Tout SGBD définit des contextes de sécurité de ses services afin de bien gérer les autorisations d'accès à ses bases de données. Nous présentons, dans ce qui suit, le mécanisme de gestion des utilisateurs sur SQL Server. Ensuite, nous décrivons le mécanisme que nous avons proposé pour la gestion des utilisateurs sur SD-SQL Server.

6.2.2.1 Gestion des Utilisateurs sur SQL Server

Sur SQL Server, l'accès des utilisateurs et la sécurité sont basés sur les utilisateurs et les rôles des bases de données SQL Server. Un *rôle* est en fait un groupe d'utilisateurs possédant des droits communs. Tout utilisateur peut accéder à la base et doit appartenir au moins à un rôle [M00].

Pour gérer facilement les autorisations dans les bases de données sur SQL Server, il est recommandé de définir un ensemble de rôles basés sur des fonctions et d'attribuer à chaque rôle les autorisations qui s'appliquent à ces fonctions au lieu d'attribuer des autorisations à chaque utilisateur individuellement.

SQL Server s'appuie par défaut sur le système d'utilisateurs de *Windows* [M00]. Des droits sont donnés sur les différents éléments de SQL Server à un groupe ou à un utilisateur *Windows*. Lors de la connexion à la base de données, l'utilisateur est identifié grâce à son *login Windows*. Il accède aux ressources de la base de données auxquelles l'administrateur lui a donné droit par l'intermédiaire de son groupe Windows ou directement à partir de son identifiant. Lors de l'accès à une ressource extérieure, le processus SQL Server agit par emprunt d'identité : l'utilisateur ne peut accéder par l'intermédiaire de SQL Server qu'aux ressources auxquelles il aurait droit s'il y accédait directement.

Lorsqu'un utilisateur est connecté à une instance, il peut disposer de droits sur l'instance elle-même et/ou sur chacune des bases gérées par l'instance. Les droits sur l'instance sont donnés par l'intermédiaire de rôles d'instance prédéfinis. Les droits sur les bases de données sont donnés par l'intermédiaire de rôles de base de données, de groupes Windows ou directement à l'utilisateur. Il existe des rôles de bases de données de type système qui donnent des droits spécifiques sur la base de données et d'autres définis par l'administrateur qui donnent des droits sur les objets. Lorsque l'accès à une instance est donné à un utilisateur ou à un groupe Windows, celui-ci dispose d'un espace utilisateur dans lequel peuvent être enregistrés des objets ayant un nom identique à celui d'un autre espace utilisateur. L'espace utilisateur par défaut des administrateurs d'une base de donnée est *dbo* (ang. *database owner*).

Si aucun utilisateur ni rôle n'ont été créé pour l'exploitation des données, c'est le propriétaire des bases par défaut (*dbo*) qui exerce ses droits. Ceux-ci étant illimités il est possible pour les utilisateurs finaux de modifier, supprimer ou insérer dans toutes les bases y compris les bases systèmes, les schémas, les données comme le code (procédures stockées et triggers notamment). Bien évidemment, cet état de fait laisse la porte grande ouverte aux attaques de serveurs SQL Server.

6.2.2.2 Gestion des Utilisateurs sur SD-SQL Server

Un utilisateur du système SD-SQL Server doit aussi disposer d'autorisations appropriées pour accéder aux données du système SD-SQL Server. Comme nous avons utilisé le SGBD SQL Server pour son implantation, nous nous basons alors sur les mêmes principes de gestion d'accès cités ci-dessus.

Nous avons défini un nouveau rôle que nous avons appelé *SD* (*Scalable Distributed*). Ce rôle est appliqué à tous les objets du système SD-SQL Server qui sont transparents pour les utilisateurs. Ces objets sont :

- Les méta-tables localisées sur les NDBs de type client, serveur et pair ainsi que sur la MDB, autrement dit, toutes les méta-tables du système SD-SQL Server.

- Les segments qui composent les tables scalables.
- Les images secondaires des tables scalables.

Quant aux autres objets du système SD-SQL Server, nous avons utilisé le rôle *dbo* pour les accéder. Ces objets sont les images primaires et toutes les procédures stockées qui implémentent les commandes SD-SQL Server.

6.2.3 Convention des Noms

Nous avons conçu SD-SQL Server de telle manière qu'il ait ses propres objets internes [SLS05]. Ces objets consistent en : les bases de nœud (NDBs), les méta-tables, les procédures stockées ainsi que les segments, les index et les images des tables scalables. Tous les objets de type systèmes sont implémentés comme des objets du système SQL Server.

Afin d'éviter les conflits des noms, en particulier entre les noms des objets sur SQL Server créés par une application SD-SQL Server, nous avons défini ces règles de nomination :

- Chaque NDB doit être créée avec le rôle *'SD'* dédié à SD-SQL Server.
- Le nom d'une base de données, d'une table, d'une vue ou d'une procédure stockée, ne doit pas être le nom d'une commande SD-SQL Server. Les noms des commandes SD-SQL Server sont considérés comme des mots clé réservés à SD-SQL Server, tout comme les commandes SQL sur SQL Server. Les raisons techniques de cette convention sont le fait que les procédures stockées qui implémentent les commandes SD-SQL Server sont de type *'public'*, c'est-à-dire elles sont implémentées sur le compte de l'utilisateur '*dbo'*. Les noms des commandes de SD-SQL Server sont composés des noms des commandes correspondantes sur SQL Server préfixés par *'sd_'* et remplaçant tout caractère blanc par le caractère *'_'* comme nous l'avons déjà présenté.
- Une table scalable *T* représente un objet *public*, c'est-à-dire, son nom global sur SQL Server est *dbo.T*. Son nom doit donc être unique sur chaque NDB. En d'autres termes, deux usagers différents d'une même NDB ne peuvent pas créer tous les deux une table scalable ayant le nom *T*. En outre, deux utilisateurs de NDBs différentes peuvent chacun créer une table scalable avec le même nom *T*.
- Un segment d'une table scalable *T* créé par l'utilisateur du nœud SQL Server *N*, aura le nom *SD._N_T* sur SD-SQL Server. Autrement dit, le nom d'un segment fait référence au nom du nœud où sa table scalable a été créée.

- Une image primaire d'une table scalable *T* a le même nom (*T)* que sa table. Son nom global sur un nœud est *dbo.T*. *T* est, en fait, le nom de la vue partitionnée et distribuée SQL Server qui implémente l'image primaire de cette table.
- Chaque image secondaire, d'une table scalable *T* créée sur une NDB de type client (ou pair) *N*, aura le nom *SD.N_T*. Ce nom est aussi celui de la vue partitionnée et distribuée implémentant l'image secondaire.

Puisque toutes les commandes SD-SQL Server sont implémentées comme des procédures stockées, SQL Server préfixe alors toutes ces procédures qui doivent être à accès *public* sur SD-SQL Server avec le rôle *'dbo.'*. Ceci permet d'éviter les conflits des noms entre les différents utilisateurs des NDBs du système SD-SQL Server.

Les règles (ou conventions) de noms, que nous avons définies, permettent d'éviter les conflits de noms entre les objets système qui existent déjà sur SD-SQL Server et ceux créés par de nouvelles applications. En effet :

- Un nom privé d'un objet créé par une application ne peut pas être en conflit avec le nom d'une autre image ou d'un segment de sa table scalable existant sur la même NDB.
- Un nom d'une image primaire ne peut pas être en conflit avec le nom d'un segment de la même NDB.
- Deux segments d'une même NDB appartenant à deux tables scalables ayant le même nom, ont des noms différents. Ainsi, ces deux segments peuvent partager la même NDB sans créer un conflit de noms.
- Les images primaires et secondaires de tables scalables différentes peuvent être créées sur la même NDB pair (ou client) sans conflit dans leurs noms.

Exemple

Soient *Dell1*, *Dell2* et *Dell3* des nœuds de type client, serveur et pair respectivement. Soit *PhotoObj* le nom de deux tables scalables différentes. Nous supposons les cas suivants :

- La première table *PhotoObj* est créée sur la NDB client *Dell1.SkyServer*. Son segment primaire est donc appelé *_Dell1_PhotoObj*, il est localisé sur *Dell2.SkyServer*. S'il éclate, son nouveau segment résultant sera localisé sur la NDB *Dell3.SkyServer*. Nous supposons que *PhotoObj* a aussi une image secondaire *Dell1_PhotoObj* localisée sur *Dell3.SkyServer*.
- La deuxième table *PhotoObj* est créée sur la NDB pair *Dell3.SkyServer*. Son segment primaire est donc appelé *_Dell3_PhotoObj*, il est localisé sur *Dell3.SkyServer*. S'il éclate, son nouveau segment résultant sera localisé sur la NDB *Dell2.SkyServer*.

Malgré que ces deux tables scalables aient le même nom *PhotoObj* sur la même SDB *SkyServer*, il n'y aura aucun conflit de noms lors de leur création et leur éclatement. En effet, avec notre convention de noms, on aura la situation suivante :

- La base de données *Dell1.SkyServer* contient l'image primaire *dbo.PhotoObj* qui constitue la vue partitionnée et distribuée de la première table *PhotoObj*, elle est nommée *SkyServer.dbo.PhotoObj* sur *Dell1*.
- La base de données *Dell2.SkyServer* contient les segments *SD._Dell1_PhotoObj* et *SD._Dell3_PhotoObj* des deux tables scalable *PhotoObj*.
- La base de données *Dell3.SkyServer* contient les segments *SD._Dell1_PhotoObj* et *SD._Dell3_PhotoObj*. Elle contient aussi les vues partitionnées et distribuées de la première table *PhotoObj* (nommée *SD.Dell1_PhotoObj*) ainsi que celle de la deuxième table correspondant à son image primaire *dbo.PhotoObj*.

6.2.3.1 Les Méta-tables

Les méta-tables SD-SQL Server sont implémentées comme des tables statiques sur SQL Server. Ces tables sont consultées et mises à jour en utilisant uniquement des procédures stockées et des requêtes SQL. Autrement dit, elles sont transparentes aux utilisateurs. C'est pour cela d'ailleurs, que nous avons proposé de créer ces tables sous le rôle *SD*. Ainsi, afin d'accéder à une méta-table dans une procédure stockée, il faut la préfixer par le rôle *'SD.'*.

6.3 Etapes Suivies dans la Configuration et l'Utilisation de SD-SQL Server

L'implémentation du système SD-SQL Server nécessite la disposition d'une configuration initiale. Cette configuration consiste à avoir au moins deux instances, du SGBD SQL Server, distribuées et liées entre elles. En effet, c'est l'administrateur du système qui s'occupe de la configuration initiale de SD-SQL Server.

Nous prenons le cas de figure de notre laboratoire. Nous avons sept machines liées entre elles dans un réseau local. Sur chaque machine, nous avons installé le logiciel SQL Server 2000. Les noms des serveurs SQL sont ceux des machines : *Ceria*, *Dell1*, *Dell2*, *Dell7*, *Dell8*, *Dell10* et *Dell11*.

Afin d'implémenter SD-SQL Server, ces SGBDs SQL Server doivent être liés entre eux (ang. *linked servers*). Nous lions toutes les instances SQL Server entre elles en exécutant

des procédures stockées du système SQL Server. Nous exécutons ces commandes sur *SQL Analyser*, l'éditeur de requête SQL [M00].

Nous prenons l'exemple des deux serveurs *Dell1* et *Dell7*. Afin de lier ces deux SGBDs entre eux, nous procédons comme suit :

1. Exécuter le code suivant sur la première instance SQL Server *Dell1* :

```
EXEC sp_addlinkedserver  'Dell1', N'SQL Server'
EXEC sp_addlinkedserver  'Dell7'
EXEC sp_configure 'remote access', 1
RECONFIGURE
```

2. Arrêter et redémarrer l'instance *Dell1*.
3. Exécuter le code suivant sur la seconde instance *Dell7* :

```
EXEC sp_addlinkedserver 'Dell7', local
EXEC sp_addlinkedserver 'Dell1'
EXEC sp_configure 'remote access', 1
EXEC sp_addremotelogin 'Dell1', sa, sa
RECONFIGURE
```

4. Arrêter et redémarrer l'instance *Dell7*.

Nous exécutons cette liste de commandes pour chaque couple d'instances de SQL Server. Ainsi, nous aurons les instances liées suivantes : *Ceria*, *Dell1*, *Dell2*, *Dell7*, *Dell8*, *Dell10* et *Dell11*.

Ayant des serveurs liés, nous pouvons les utiliser pour constituer les nœuds SD-SQL Server. Pour cela, nous exécutons les étapes décrites dans les sections suivantes.

6.3.1 Création du Nœud Primaire SD-SQL Server

La première étape dans la configuration d'un système SD-SQL Server est la création de son nœud primaire à partir d'un serveur lié SQL Server. Nous choisissons un serveur lié parmi la collection des serveurs liés que nous avons déjà établie dans la section précédente. Nous supposons que *Dell1* est un serveur lié. Le premier nœud créé pour le système SD-SQL Server doit être de type pair pour pouvoir contenir la méta-base MDB avec tous les objets (méta-tables et procédures stockées) qui permettent la gestion des nœuds, des SDBs et des NDBs.

Nous créons alors la méta-base MDB sur le nœud primaire *Dell1*. MDB est une base de données SQL Server simple pour le moment. Afin de la rendre comme étant une méta-base servant de dictionnaire dans le système SD-SQL Server, nous exécutons un script SQL sur cette base. Nous avons écrit ce script pour contenir l'ensemble de transact-SQL permettant de créer toutes les méta-tables et les procédures stockées qu'une NDB de type pair peut contenir. L'exécution de ce script crée alors l'ensemble de ces objets sur la MDB. Nous avons appelé ce script *Install.sql*. Lors de l'initialisation du nœud SD-SQL Server primaire, et avant la création de la MDB, nous sauvegardons ce fichier dans un répertoire appelé *script*, dédié aux scripts SQL du système SD-SQL Server. Ce répertoire se trouve sur le disque de la machine *Dell1* qui représente ce nœud primaire. Nous utilisons la commande suivante pour l'exécution du script *install* :

```
EXEC master..xp_cmdshell 'osql /U sa /P /d MDB < c:\script\install.sql'
```

L'exécution du script *install.sql* sur la MDB donne les résultats suivants :

- La création des méta-tables *Nodes* et *SDB*.
- L'insertion du tuple *(Dell1, peer)* dans la méta-table *Nodes* pour indiquer l'existence du nœud primaire *Dell1*.
- Création de toutes les procédures stockées qui permettent la création des nœuds, des NDBs et des SDBs afin de les utiliser par la suite dans la création des premiers nœuds et SDBs de la configuration. Ces procédures stockées sont :
    - sd_create_node
    - sd_alter_node
    - sd_drop_node
    - sd_create_scalable_database
    - sd_create_node_database
    - sd_drop_node_database
    - sd_drop_scalable_database

6.3.2 Création des autres Nœuds SD-SQL Server

Après avoir créé les procédures stockées qui implémentent les commandes SD-SQL Server permettant la création des nœuds et des SDBs, nous pouvons alors créer d'autres nœuds SD-SQL Server avec la commande *sd_create_node* à partir de la base MDB du nœud *Dell1*.

Nous supposons la création des nœuds *Dell11*, *Dell0, Dell7, Dell8* et *Ceria.* Nous considérons les trois premiers nœuds (*Dell11*, *Dell0* et *Dell7*) de type serveur, *Dell8* de type pair et *Ceria* de type client.

L'exécution des commandes suivantes (de la MDB) sur l'analyseur SQL, permet la création des nœuds *Dell11*, *Dell0, Dell7, Dell8* et *Ceria* :

sd_create_node 'Dell11'

sd_create_node 'Dell10, 'server'

sd_create_node 'Dell7, 'server'

sd_create_node 'Dell8, 'peer'

sd_create_node 'Ceria', 'Client'

En plus de la création des nœuds, les commandes ci-dessus permettent d'insérer les tuples décrivant ces nœuds dans les méta-tables de la MDB. Les tuples *(Dell11, server), (Dell10, server), (Dell7, server), (Dell8, peer)* et *(Ceria, client)* seront insérés dans la méta-table *Nodes* de la MDB.

Si l'administrateur du système veut modifier le type d'un nœud existant, il exécute alors la commande *sd_alter_node*. Nous supposons par exemple la modification du nœud *Ceria* en lui rajoutant les capacités d'un nœud de type serveur. Nous exécutons ainsi la commande *sd_alter_node* comme suit :

sd_alter_ node 'Ceria, 'ADD Server'

Pour résumer, la configuration de notre système possède jusqu'à maintenant six nœuds :

- Un nœud primaire de type pair, *Dell1* contenant la MDB.
- Un nœud de type serveur, *Dell11.*
- Un nœud de type serveur, *Dell10.*
- Un nœud de type serveur, *Dell7.*
- Un nœud de type pair, *Dell8.*
- Un nœud de type client, *Ceria*.

6.3.3 Création d'une SDB

Comme nous utilisons la base de test *SkyServer*, nous allons alors garder les mêmes noms de base de données et de tables de cette base. Nous supposons la création de la SDB *SkyServer* avec une seule extension qui est sa NDB primaire. Nous choisissons le nœud *Dell11* pour héberger la NDB primaire *SkyServer*. L'exécution de la commande suivante sur la base *Dell1.MDB* est comme suit :

sd_create_scalable_database 'SkyServer, 'Dell11'

Le résultat de l'exécution de cette commande est la SDB *SkyServer* qui a pour l'instant une NDB primaire. Cette NDB est de type serveur par défaut puisque aucun type n'a été indiqué dans la commande. Cette commande permet aussi la création et l'initialisation de toutes les méta-tables qu'une NDB serveur peut contenir, c'est-à-dire *RP, Primary, Size, MDBNode* et *SDBNodes*. Les méta-tables *RP, Primary* et *Size* ne seront pas encore initialisées à ce niveau de la configuration. Quant à la table *MDBNode*, elle aura comme entrée le tuple *(Dell1)* qui indique la localisation (le nœud) de la MDB. La méta-table *SDBNode* sera initialisée par le tuple *(Dell11)* qui indique la localisation de la NDB primaire de la SDB *SkyServer*.

Pour pouvoir accéder à la SDB *SkyServer*, il faudrait qu'elle ait au moins une NDB de type client (ou pair). Nous créons ainsi une NDB *SkyServer* de type client en exécutant la commande ci-dessous. Nous créons cette NDB sur le nœud *Ceria* qui est de type client :

sd_create_node_database 'SkyServer', 'Ceria', 'client'

Le résultat de cette commande est alors une nouvelle NDB (de type client), dans la SDB *SkyServer*, qui sera localisée sur le nœud *Ceria*. En plus de la création d'une NDB, cette commande permet de créer toutes les procédures stockées correspondant aux commandes SD-SQL Server ainsi que les méta-tables *Image* et *Server*. La méta-table *Image* reste vide, quant à la méta-table *Server*, elle aura comme entrée les tuples indiquant les nœuds SD-SQL Server de type serveur ou pair qui détiennent des NDBs de la SDB courante *SkyServer*. Puisque la SDB *SkyServer* n'a qu'une NDB serveur *Dell11*, le tuple *(Dell11)* est donc inséré dans la méta-table *Server*. Ces NDBs vont servir dans la création de tables scalables. Dès lors, les utilisateurs de SD-SQL Server peuvent accéder à la SDB *SkyServer* et créer ainsi des tables scalables.

6.3.4 Création d'une Table Scalable

Nous utilisons aussi la base de test *SkyServer* pour la création des tables scalables. Nous exploitons les schémas de ses tables (qui sont statiques) pour les créer comme des tables scalables sur SD-SQL Server. Nous utilisons tout particulièrement la table *PhotoObj* [G02].

Soit un utilisateur de la NDB client *Ceria.SkyServer*. Nous supposons que cet utilisateur veut créer la table scalable *PhotoObj* avec la capacité de 150000 tuples. Comme clé de partitionnement, il utilise sa clé primaire *Objid*. Il exécute alors la commande *sd_create_table* comme suit :

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

Le résultat de cette commande sera alors la création du segment primaire et de l'image primaire de la table scalable *PhotoObj*. Le segment sera nommé *_Ceria_PhotoObj*. Il sera localisé sur la NDB *SkyServer* du nœud *Dell11*. Ce nœud est sélectionné à partir de la méta-table *Server*. Nous rappelons que la référence au nœud *Ceria* dans le nom du segment désigne le nœud client (ou pair) sur lequel a été lancé la création de la table scalable. Quant à l'image primaire, elle sera nommée *PhototObj* et elle sera localisée sur *Ceria.SkyServer*, la NDB de l'exécution de la commande.

Les méta-tables *RP, Primary* et *Size* auront les entrées suivantes respectivement : *(Dell11, Ceria, PhotoObj)*, *(Dell11, Ceria, PhotoObj)* et *(Ceria, PhotoObj, 150000)*. Quant à la clé de partitionnement, puisqu'elle n'a pas été indiquée dans la commande (il n' y a pas de troisième paramètre), elle sera donc implicitement l'attribut clé *Objid* comme déjà mentionné.

Nous donnons maintenant l'exemple de la création de la table scalable *Neighbors*. Cette table a une clé primaire composée de trois attributs clé (*objid*, *htmid* et *Neighborobjid*), comme modélisés sur la base du benchmark *SkyServer* [G02]. Le même utilisateur (du nœud *Ceria*) choisit, par exemple, l'attribut clé *Objid* comme clé de partitionnement de la table *Neighbors*. Pour la capacité de la table, nous supposons qu'il choisit une capacité de 500 tuples. Ainsi, il exécute la commande *sd_create_table* comme suit :

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

Le résultat de cette commande est alors la table scalable *Neighbors* avec son segment primaire *_Ceria_Neighbors* localisé sur la NDB *SkyServer* du nœud *Dell11*, et une image primaire sur la NDB *Ceria.SkyServer.*

6.3.5 Modification d'une Table Scalable

Le même utilisateur de la NDB client *Ceria.SkyServer* peut modifier aussi la table scalable *PhotoObj* qu'il a créée. Nous supposons qu'il veut rajouter l'attribut *t* de type *INT* à l'ensemble des attributs de la table *PhotoObj,* et modifier aussi sa capacité de 150000 tuples à 10000 tuples. Il exécute alors la commande *sd_alter_table* comme suit :

```
sd_alter_table 'PhotoObj ADD t INT', 10000
```

Le résultat de l'exécution de cette commande sera alors le segment primaire de la table scalable *PhotoObj* avec une nouvelle colonne *'t'* et une nouvelle capacité (10000 tuples). Le tuple décrivant la table scalable *PhotoObj* dans la méta-table *Size* sera donc mis à jour. Sa colonne *Size* qui était égale à 150000 aura la nouvelle valeur 10000.

L'utilisateur peut aussi créer un index sur un attribut de la table scalable *PhotoObj*. Il exécute alors la commande *sd_create_index* comme suit :

sd_create_index ‘run_index ON Photoobj (run)'

Si la table scalable *PhotoObj* éclatera, la nouvelle colonne *t* et le nouvel index seront propagés sur ses nouveaux segments.

6.3.6 Création des Images Secondaires

Nous avons vu lors de la création de la table scalable *PhotoObj* que son image primaire est créée sur la NDB client *Ceria.SkyServer*. L’utilisateur peut créer une image secondaire pour la table scalable *PhotoObj* sur la NDB *SkyServer* du nœud pair *Dell8* par exemple. Il exécute alors la commande *sd_create_image* sur cette NDB tout en indiquant le nœud *Ceria* qui détient l’image primaire *PhotoObj* :

sd_create_image ‘Ceria’, ‘PhotoObj’

Le résultat de l’exécution de cette commande sera alors l’image secondaire nommée *Ceria_PhotoObj* localisée sur la NDB *Dell8.SkyServer*. Si l’utilisateur veut créer l’image secondaire, sous un autre nom, il exécute alors la commande SQL habituelle *CREATE VIEW*.

6.3.7 Suppression d’une Image Secondaire

Si l’utilisateur n’a plus besoin de l’image secondaire *Ceria_PhotoObj* créée précédemment, il peut la supprimer en exécutant la commande *sd_drop_image* comme suit :

sd_drop_image 'SD.Ceria_Photoobj'

6.3.8 Accès à une Table Scalable

L’accès à une table scalable consiste à chercher, insérer, mettre à jour, ou supprimer des données dans cette table. Toutes les requêtes d’accès interrogent une table scalable à travers son image. Afin d’illustrer l’utilisation de ces commandes sur SD-SQL Server, nous donnons l’exemple de la commande d’une insertion. Nous donnons d’autres exemples pour toutes les commandes, dans les sections suivantes lors de l’explication de leur traitement interne.

Soit un utilisateur de la NDB *Ceria.SkyServer* qui effectue une insertion dans la table *PhotoObj* (à travers son image primaire). Nous supposons qu’il exécute la commande d’insertion comme suit :

sd_insert ‘INTO PhotoObj SELECT * FROM Ceria5.Skyserver-S.PhotoObj’

Le résultat d'exécution de cette commande sera le transfert de tous les tuples de la table statique *PhotoObj,* localisée sur la base statique *Ceria5.SkyServer-S,* vers la nouvelle table scalable *PhotoObj*. Si le nombre de tuples transférés dépasse la capacité qui a été fixée pour la table *PhotoObj* (lors de sa création), elle éclatera en conséquence.

6.3.9 Suppression d'une Table Scalable

Pour la suppression de la table scalable *PhotoObj*, l'utilisateur exécute la commande *sd_drop_table* comme suit :

```
sd_drop_table 'PhotoObj'
```

Le résultat d'exécution de cette commande sera alors la suppression de l'image primaire et de tous les segments de la table scalable *PhotoObj*. Toutes les métadonnées décrivant la table scalable et son image primaire seront supprimées aussi.

6.4 Traitement Interne des Commandes SD-SQL Server

Nous présentons, dans cette section, les détails sur la structure et le traitement interne de chaque commande SD-SQL Server. Nous implémentons chaque commande par une procédure stockée sur SQL Server. Les paramètres spécifiques (capacité, clé de partitionnement, etc) de chaque commande représentent les arguments d'entrée de la procédure stockée l'implémentant. Par exemple, la commande SD-SQL Server *sd_select* qui a comme paramètre la clause SQL *SELECT* sera implémentée par une procédure stockée nommée *sd_select* et ayant comme argument la chaîne de caractère correspondant à la clause SQL.

Quant au traitement interne de chaque commande, il sera représenté par un code dans le corps de la procédure stockée. Ce code utilise Transact-SQL ainsi que des instructions dynamiques. Nous détaillons, dans les sections suivantes, les traitements sur l'évolution des tables scalables, des images et des requêtes. Nous présentons aussi les traitements sur les nœuds, les SDBs et les NDBs. Pour plus de clarté sur le déroulement de ces traitements, nous abordons les traitements sur les tables scalables avant les traitements sur leurs nœuds. Nous reprenons pour cela la configuration utilisée dans la section précédente.

6.4.1 Gestion des Tables Scalables

6.4.1.1 Création d'une Table Scalable

Nous rappelons qu'une table scalable est formellement définie par un tuple *(T, S)* où *T* présente son image primaire et *S* l'ensemble de ses segments. La création d'une table scalable est réalisée par l'exécution de la commande *sd_create_table* comme nous l'avons déjà mentionné.

Nous reprenons l'exemple de la création de la table scalable *PhotoObj* de la section 6.3.4 afin d'expliquer les étapes du traitement interne de la commande *sd_create_table.* Nous rappelons que nous avons exécuté cette commande sur la NDB *Ceria.SkyServer* comme suit :

sd_create_table 'PhotoObj (Objid Bigint NOT NULL Primary_Key)…', 150000

Le traitement interne de cette commande consiste à suivre les étapes suivantes :

- Le gestionnaire SD-SQL Server extrait le nom de la table scalable à créer du premier paramètre de la commande. Pour cela, il utilise les fonctions sur les chaînes de caractères *SUBSTRING* et *CHARINDEX*. Il sauvegarde le nom de la table scalable (qui est dans notre exemple *PhotoObj)* dans une variable.
- Ensuite, à partir du nom de la table scalable *PhotoObj* (récupéré du premier paramètre) et à partir du nœud de l'exécution de la commande *Ceria*, il formule le nom de son segment primaire comme nous l'avons déjà décrit dans la section précédente. Donc le segment primaire de la table scalable *PhotoObj* aura le nom *_Ceria_PhotoObj* et il sera préfixé du rôle *SD*, c'est-à-dire le nom global d'un segment est *SkyServer.SD._Ceria_PhotoObj*.
- Puisque SD-SQL Server commence par la création du segment primaire d'une table scalable, il reformule donc la clause entrée dans la commande en précisant le nom du segment *SD._Ceria_PhotoObj*. Il complète aussi cette clause en la transformant en une requête SQL compréhensible par SQL Server. Ainsi, la requête devient :

  CREATE TABLE SD._Ceria_PhotoObj (Objid bigint NOT NULL Primary)

- Avant d'exécuter cette requête, il sélectionne d'abord un noeud serveur où créer ce segment primaire. Il sélectionne ce nœud à partir de la méta-table *SD.Server* qui se trouve sur la NDB courante de l'exécution *(Ceria.SkyServer)*. Soit *Dell11*, le nœud serveur qui détient la NDB *SkyServer*. Il exécute alors la requête précédente sur *Dell11.SkyServer*. Le résultat de l'exécution de cette requête est le segment primaire *SD._Ceria_PhotoObj* sur la NDB *Dell11.SkyServer*.
- Une fois le segment primaire de la table scalable *PhotoObj* est créé, SD-SQL Server lui associe un déclencheur de type *AFTER INSERT*. Ce déclencheur fait appel à un

job (ou agent) asynchrone que nous avons appelé *éclatur* (ang. *splitter*). Ce dernier vérifie la taille du segment correspondant et lance son éclatement en cas de débordement. Nous décrivons sa structure interne dans la Section 6.4.1.2.1.

- SD-SQL Server crée aussi l'image primaire qui représente le partitionnement actuel de la table *PhotoObj*. Cette image sera créée sur le compte utilisateur courant de la SDB *SkyServer*, c'est-à-dire *'dbo'*. Elle sera créée sur la NDB client qui lance la commande de création, c'est-à-dire *Ceria.SkyServer*. La définition de l'image primaire *dbo.PhotoObj* est comme suit :

```
CREATE view PhotoObj AS
SELECT * FROM dell11.SkyServer.SD._Ceria_PhotoObj
```

- Une fois, le segment primaire et l'image primaire de la table scalable *PhotoObj* sont créés, SD-SQL Server insère de nouvelles entrées décrivant *PhotoObj* dans les méta-tables :
  - Il insère le tuple *(Dell11, Ceria, PhotoObj)* dans la méta-table *Dell11.SkyServer.SD.Prim*.
  - Il insère le tuple *(Ceria, PhotoObj, 150000)* dans la méta-table *Dell11.SkyServer.SD.Size*. La troisième colonne *Size* de la méta-table *SD.Size* est affectée par la capacité entrée dans la commande *sd_create_table*.
  - Il insère le tuple *(Dell11, Ceria, PhotoObj)* dans la méta-table *Dell11.SkyServer.SD.RP*.
  - Il insère le tuple *(PhotoObj, Primary, Dell11, 1)* dans la méta-table *Ceria.SkyServer.SD.Image.*

La figure suivante montre les résultats de la création de la table scalable *PhotObj*. Les détails sur le traitement interne de la procédure stockée correspondant à la commande *sd_create_table*, sont dans l'Annexe A.

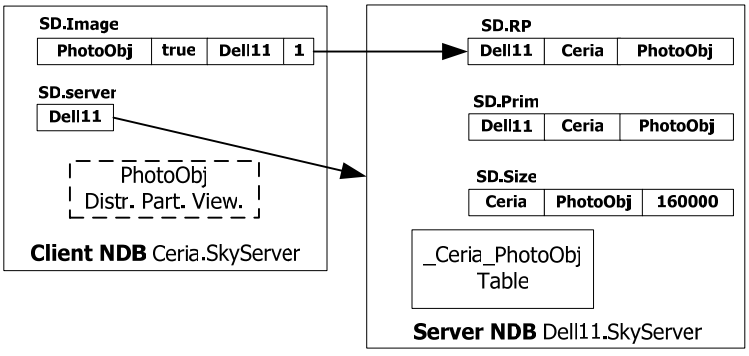

Figure 6-1 : Résultats de la création de la table scalable *PhotoObj*

6.4.1.2 Evolution d'une table scalable

L'évolution d'une table scalable consiste à rajouter de nouveaux segments dans la table scalable (son éclatement), modifier son schéma, ajuster ses images et la supprimer. Nous détaillons dans ce qui suit les traitements internes de chaque commande qui réalise ces manipulations.

6.4.1.2.1. Eclatement

Une table scalable est étendue sur plusieurs NDBs si elle éclate en plusieurs segments. Son éclatement résulte d'une insertion qui fait déborder un (ou plusieurs) de ses segments. Nous expliquons les traitements internes d'un éclatement à travers l'exemple de la table *PhotoObj* créée précédemment.

Nous supposons l'insertion de *160000* tuples dans la table scalable *PhotoObj* composée d'un segment primaire jusqu'à présent. Nous utilisons la capacité *b=150000* tuples déjà associée à la table *PhototObj*. Il est clair que les *160000* tuples font déborder *PhotoObj* de sa capacité et ainsi l'éclater. Voyons comment procède le gestionnaire SD-SQL Server dans ce cas.

Lors de toute insertion dans une table scalable, le gestionnaire SD-SQL Server lance le déclencheur du segment (des segments) qui est (sont) impliqué(s) dans cette insertion. En effet, les tuples seront insérés dans les segments tout en respectant leurs contraintes d'intégrité. Dans notre exemple, un seul segment est concerné par l'insertion des 160000 tuples. Suite à cette insertion, le déclencheur du segment sera lancé. Ce dernier fait appel au *splitter*, qui est un programme exécuté en différé avec l'insertion afin de ne pas la pénaliser (en la mettant en attente). L'éclateur est implémenté sur SD-SQL Server comme étant un *job* asynchrone du système SQL Server. Il est invoqué dans le corps du déclencheur de chaque segment d'une table scalable par la commande suivante :

```
EXEC msdb..sp_start_job 'splitter'
```

Le rôle du splitter est de vérifier si le segment visé par l'insertion n'a pas débordé et dans le cas affirmatif il lance son éclatement. Pour cela, il fait appel à une procédure stockée, que nous avons appelée *split_table.* Cette procédure stockée a un paramètre en entrée. Il s'agit du nom du segment à éclater. Dans notre exemple, ce paramètre est affecté par le nom *SD._Ceria_PhotoObj* du segment primaire de la table *PhotoObj*. Le traitement interne de cette procédure consiste à suivre les étapes ci-dessous :

- Tout d'abord, le gestionnaire SD-SQL Server vérifie si la taille du segment *_Ceria_PhotoObj* lançant l'éclateur n'a pas dépassé sa capacité.
    - Si le nombre de tuples dans ce segment est inférieur à sa capacité *b,* l'éclateur termine ses traitements.

- Si *b* est dépassée, l'éclateur poursuit son traitement. Et c'est le cas de notre exemple.

o En cas de débordement, le gestionnaire crée, selon la taille du segment, un (ou plusieurs) nouveau(x) segment(s) ayant le même nom que le segment éclatant (*_Ceria_PhotoObj*). En effet, il soustrait la taille *INT (b/2)* (la moitié de la capacité) de la taille du segment *_Ceria_PhotoObj*. La taille qui est en résulte est *b'= 85000* tuples. Puisque *b'* est inférieure à la capacité *b* de *PhotoObj*, donc un seul nouveau segment suffira pour y transférer les tuples en excès. Un nouveau segment *_Ceria_PhotoObj* sera donc créé. Mais pour cela, il faudrait une NDB serveur disponible de la SDB *SkyServer* afin d'héberger ce nouveau segment. Le gestionnaire procède alors comme suit :

- Il sélectionne une localisation (un noeud) d'une NDB *SkyServer* à partir de la méta-table *SD.NDB* (localisée sur *Dell11.SkyServer*). La sélection porte sur les NDBs de type serveur (ou pair) qui n'ont pas encore hébergé un segment de la table *PhotoObj*. S'il y a plusieurs résultats, le choix se porte sur la première NDB sélectionnée. Nous supposons que *Dell10* est le nœud qui détient une NDB serveur *SkyServer*.
- Une fois cette NDB est sélectionnée, le nouveau segment *_Ceria_PhotoObj* de la table *PhotoObj* est alors créé sur la NDB *Dell10.SkyServer* en exécutant la requête suivante :

```
SELECT TOP 85000 WITH TIES *
  INTO Dell10.SkyServer.SD._Ceria_PhotoObj
    FROM Dell11.SkyServer.SD._Ceria_PhotoObj
      ORDER BY Objid ASC
```

Ainsi, la table scalable *PhotoObj* possède deux segments distribués sur les NDBs *SkyServer* des noeuds *Dell11* et *Dell10* respectivement.

o Chaque nouveau segment résultant de l'éclatement aura le même schéma (les mêmes attributs) que son segment éclatant. Or, la requête *SELECT INTO* exécutée précédemment n'applique ni les contraintes d'intégrité ni la clé primaire sur le nouveau segment. Ainsi, le gestionnaire doit compléter ces traitements :

- Tout d'abord, il crée une clé primaire sur le nouveau segment. Il la récupère à partir du segment éclatant en utilisant la commande *sp_helpindex* du système SQL Server ; et la sauvegarde dans une variable. Ensuite, il utilise la requête SQL habituelle *ALTER TABLE* combinée avec le langage SQL dynamique. La clé primaire concerne l'attribut *Objid* de la table *PhotoObj* dans notre exemple.

- Maintenant que tous les segments distribués ont leur propre clé primaire et ainsi leur clé de partitionnement, nous pouvons définir leur contrainte d'intégrité. L'intervalle de valeurs de la contrainte d'intégrité pour le segment *SD._Ceria_PhotoObj* de la NDB *Dell10.SkyServer* est défini entre 85000 et 160000. Quant au premier segment *Dell7.SkyServer.dbo._Ceria_PhotoObj,* il aura le reste des tuples, et sera défini dans l'intervalle des valeurs entre 1 et 84499.

o Ensuite, les informations sur le nouveau segment sont enregistrées dans les méta-tables comme suit :

- Insertion du tuple *(Dell11, Ceria, PhotoObj)* dans la méta-table *Dell10.SkyServer.SD.Primary*. Celui-ci pointe sur le nœud *Dell11* qui détient le segment primaire de la table *PhotoObj*.
- Insertion du tuple *(Ceria, PhotoObj, 150000)* dans la méta-table *Dell10.SkyServer.SD.Size*.
- Insertion du tuple *(Dell10, Ceria, PhotoObj)* dans la méta-table *SD.RP* de la NDB *Dell11.SkyServer* qui héberge le segment primaire.

o Suppression des tuples transférés du segment éclatant *Dell11.SkyServer.SD._Ceria_PhotoObj.*

La figure suivante montre le résultat de l'éclatement du segment primaire *Dell11.SkyServer.SD._Ceria_PhotoObj.*

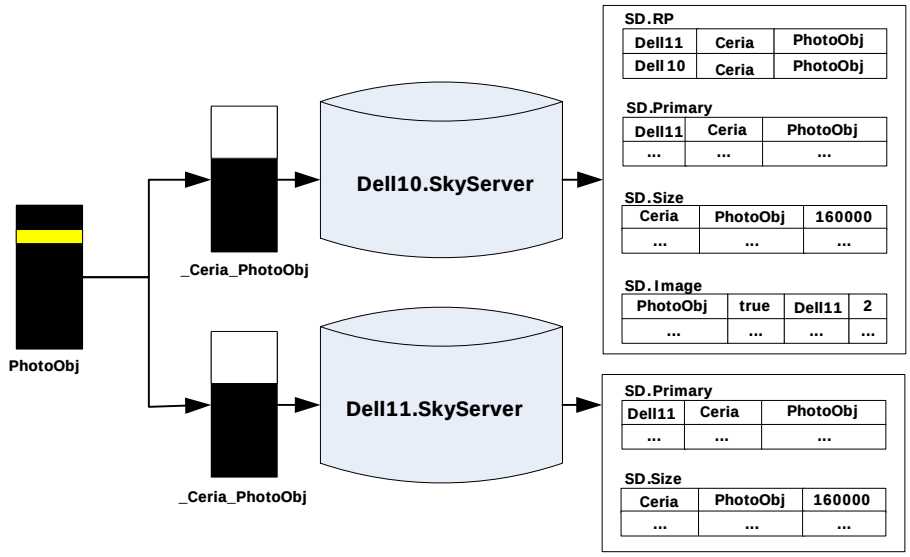

Figure 6-2 : Résultat de l'éclatement de la table scalable *PhotoObj*

Considérons maintenant la même insertion précédente mais avec une nouvelle capacité *b=100000*. L'insertion dans la table *PhotoObj* va donc l'éclater en trois segments. Soit *Dell10* et *Dell8*, muni chacun de la NDB *SkyServer*, les nœuds disponibles pour l'éclatement de *PhotoObj*. La première portion *b/2=50000* tuples sera transférée vers la NDB *Dell10.SkyServer,* par exemple. Elle remplira la moitié de son nouveau segment *_Ceria_PhotoObj*. Le segment primaire éclatant lui restera alors *110000* tuples, ce qui est toujours supérieur à sa capacité *b*. Ainsi, l'éclatement se poursuivra et une autre portion *b/2=50000* tuples sera transférée vers l'autre nouveau segment *Ceria_PhotoObj* de la NDB *Dell8.SkyServer* comme nous l'avons déjà mentionné. Le segment primaire de la table *PhotoObj* aura alors *60000* tuples, ce qui est inférieur à sa capacité *b*, donc l'éclatement s'arrêtera. Enfin, le segment primaire supprime tous les tuples qui ont été transférés vers les nouveaux segments. Quant aux traitements sur la définition des contraintes d'intégrité…etc, se sont les même décrites précédemment.

Dans ce dernier cas, la table *PhotoObj* aura trois segments distribués. Ainsi, une table scalable s'étend selon sa taille.

6.4.2 Ajustement des Images

L'ajustement des images est effectué lorsqu'il y a une requête scalable qui interroge cette image. L'image peut être invoquée directement dans la requête scalable, c'est-à-dire c'est une image primaire ou secondaire. Elle peut être aussi invoquée à travers une vue scalable. Dans chaque commande SD-SQL Server qui fait appel à une image, la procédure d'ajustement de cette image est invoquée. En effet, cet ajustement permet de vérifier si une image correspond au partitionnement actuelle de sa table scalable. Nous rappelons que si une table scalable a de nouveaux segments suite à un éclatement, ces segments ne seront pas encore présentés dans leur image que si celle-ci est interrogée. C'est ainsi que SD-SQL Server effectue l'ajustement des images dans toute commande qui les interroge. Les commandes SD-SQL Server concernées sont celles qui ont comme paramètre d'entrée des clauses SQL. Il s'agit en particulier des commandes suivantes :

- sd_select
- sd_insert
- sd_update
- sd_delete

Nous détaillons le traitement interne de l'ajustement des images à travers la commande *sd_select* suivante :

```
sd_select '* FROM PhotoObj'
```

La clause de cette commande est *'* FROM PhotoObj'*. Nous désignons par *Q* cette commande. Avant de faire appel à la procédure d'ajustement des images, SD-SQL Server vérifie d'abord s'il y a effectivement une image qui est utilisée dans la clause entrée. Les détails sur ce point sont traités dans la Section 6.4.6.1. Nous nous intéressons dans cette section à l'ajustement des images. Nous supposons donc que cette étape de vérification est déjà faite et que son résultat sur la requête *Q* est l'image *PhotoObj*. Cette dernière doit être donc ajustée avant l'exécution de *Q*. Le traitement interne de la procédure d'ajustement des images suit les étapes suivantes :

- Tout d'abord, SD-SQL Server génère une requête SQL complète *Q'* à partir de la requête *Q*. Nous voulons dire par le mot *'complète'*, le fait qu'elle soit compréhensible par le SGBD SQL Server. C'est donc *Q'* qui sera exécutée par la suite. Ce traitement est dû au fait que les clauses des commandes SD-SQL Server n'ont pas le mot clé désignant la clause SQL. Dans la commande *sd_select*, par exemple, la clause *Q* en paramètre n'a pas le mot clé *SELECT* puisque celui-ci apparaît dans le verbe (nom de la commande) *sd_select*. C'est ainsi que SD-SQL Server génère la requête *Q'* définie comme suit :

  'SELECT * FROM PhotoObj'

- Avant de passer *Q'* à l'exécution sur SQL Server, le gestionnaire client vérifie si l'image *PhotoObj* est correcte. Pour cela, il récupère à partir de la méta-table *Image* localisée sur la NDB *Ceria.SkyServer*, le champ *Size* indiquant le nombre de segments défini dans l'image *PhotoObj*. Soit S_I ce nombre. Dans notre exemple $S_I = 1$, parce que l'image *PhotoObj* n'a pas encore été ajustée, donc elle représente uniquement le segment primaire de sa table *PhotoObj*.

- Il récupère ensuite la localisation de la NDB serveur *SkyServer* qui détient le segment primaire de la table scalable *PhotoObj*. Il le trouve dans le champ *PrimNd* du tuple décrivant *PhotoObj* dans la méta-table *Image.* Il s'agit du nœud *Dell11* dans notre exemple. Si entre temps, la NDB primaire a été supprimée par la commande *sd_drop_node_database,* ou elle a été déplacé suite à une exécution de la commande *sd_drop_node*, le client cherche la localisation de la nouvelle NDB primaire de l'image *PhotoObj* dans la méta-table *SD.SDB*.

- Une fois la localisation de la NDB primaire est déterminée (*Dell11.SkyServer*), SD-SQL Server accède à sa méta-table *RP* et détermine le partitionnement actuel de *PhotoObj*. Nous supposons que *PhotoObj* est partitionnée en deux segments localisés sur les NDBs *SkyServer* des nœuds *Dell11* et *Dell10*. Une fois sur *RP*, SD-SQL Server exécute une requête qui calcule le nombre de segments de la table scalable *PhotoObj*, soit S_A ce nombre. Ayant ce nombre, il procède comme suit :

- Si $S_A = S_I$, l'image *PhotoObj* définit le même nombre de segments qui existent actuellement dans sa table scalable. L'image *PhotoObj* est donc correcte et elle ne sera pas ajustée.
- Si $S_A > S_I$, le gestionnaire ajuste alors S_I à la valeur de S_A dans le champ *Size* du tuple correspondant à l'image *PhotoObj* dans la méta-table *Image*. De plus, il modifie la définition de l'image *PhotoObj* en lui rajoutant le(s) nom(s) du segment(s) manquant(s). Il récupère tout ça de la méta-table *Dell11.SkyServer.SD.RP*. Ce cas correspond à notre exemple où (S_I=1) < (S_A=2).

o Enfin, l'image *PhotoObj* est ajustée, SD-SQL Server passe alors la requête *Q'* à l'exécution sur SQL Server.

6.4.3 Modification d'une Table Scalable

La modification d'une table scalable est réalisée par la commande *sd_alter_table*. Cette commande offre les fonctionnalités traditionnelles de la requête SQL *ALTER TABLE* qui se résument en : ajouter, modifier ou supprimer un attribut de la table scalable. De plus de ces fonctionnalités, la commande *sd_alter_table* permet aussi de modifier la capacité d'une table scalable. Nous décrivons dans ce qui suit les traitements internes de chacune de ces fonctionnalités.

6.4.3.1 Modification du Schéma d'une Table Scalable

Nous reprenons l'exemple de la modification de la table scalable *PhotoObj* de la section 6.3.5 présenté comme suit :

sd_alter_table 'PhotoObj ADD test varchar (50)', 100000

Nous supposons que la table scalable *PhotoObj* est partitionnée en deux segments. Son segment primaire est localisé sur *Dell11.SkyServer* et l'autre segment sur *Dell10.SkyServer*. Les traitements internes de la commande *sd_alter_table* consistent à suivre les étapes suivantes :

o Le gestionnaire SD-SQL Server extrait le nom de la table scalable *PhotoObj* à partir de la clause *'PhotoObj add test varchar (50)'* que nous désignons par *Q*. Il utilise pour cela les fonctions *SUBSTRING* et *CHARINDEX*.

o Ensuite, il récupère le nom du noeud primaire de la table scalabe à modifier. Pour cela, il accède à la méta-table *Image* localisée sur la NDB de l'exécution de la commande. Il s'agit de la NDB *Ceria.SkyServer* dans notre exemple. Il cherche le nom de l'image *PhotoObj* et récupère son tuple. A partir du champ *PrimNd* de ce tuple, il récupère le nom de la NDB cherché, qui est dans notre exemple le nœud

Dell11. Ayant le nœud primaire de la table *PhotoObj*, il accède à sa NDB *SkyServer*. Il trouve le nom de cette NDB à partir de la NDB client courante où la commande *sd_alter_table* est exécutée. Nous rappelons que ces NDBs constituent la même SDB *SkyServer*.

- Il accède alors à la méta-table *SD.Prim* de la NDB primaire *Dell11.SkyServer*. Il extrait ensuite le nom du nœud client sur lequel la création de la table scalable a été lancée. Pour cela, il cherche le tuple dont le champ *Tab* est égal à '*PhotoObj'*. Une fois ce tuple est trouvé, il extrait le nom du nœud client à partir de son champ *CreatNd*. Dans notre exemple, ce nœud est *Ceria*. Ce nœud sert à formuler le nom des segments de la table scalable *PhotoObj* qui est *_Ceria_PhotoObj*.
- Après avoir constitué le nom de segment de la table *PhotoObj*, SD-SQL Server reformule la clause *Q* afin qu'elle soit compréhensible par SQL Server. De plus, il remplace le nom de l'image *PhotoObj* dans *Q* par le nom de ses segments parce que c'est le schéma des segments qui sera modifié. La nouvelle requête sera alors *Q'*. Elle est définie comme suit :

  'ALTER TABLE _Ceria_PhotoObj ADD test Varchar(50)'

- Ensuite, Il accède à la méta-table *RP,* localisée sur la NDB primaire *Dell11.SkyServer,* pour chercher les tuples décrivant la table *PhotoObj*. Pour chaque tuple trouvé, il récupère la localisation de chacun des segments de la table scalable. Il utilise pour cela le champ *SgmNd*. Dans le cas de notre exemple, les localisations des segments de *PhotoObj* sont les nœuds *Dell11* et *Dell10*.
- Pour chaque nœud trouvé, il accède à sa NDB *SkyServer* et exécute la nouvelle requête scalable reformulée *Q'*. Il sauvegarde aussi la nouvelle capacité entrée dans la commande *sd_alter_table* (qui est 10000 tuples) pour chaque segment *_Ceria_PhotoObj*. En fait, il remplace par la nouvelle capacité (100.000 tuples) l'ancienne capacité de *PhotoObj* qui est sauvegardée dans la méta-table *SD.Size* de chaque NDB *Dell11.SkyServer* et *Dell10.SkyServer*.

Ainsi, le résultat de l'exécution de la commande *sd_alter_table* sera les deux segments de la table *PhotoObj Dell11.SkyServer.SD._Ceria_PhotoObj* et *Dell10.SkyServer.SD._Ceria_PhotoObj* ayant chacun une nouvelle colonne *'test'* et une nouvelle capacité, *10000* tuples.

6.4.3.2 Les Index

Les manipulations des index sur une table scalable se résument dans la création ou la suppression d'index. Ces deux manipulations suivent les mêmes traitements internes dans l'exécution de leurs commandes *sd_create_index* et *sd_alter_index*. Nous détaillons alors ces traitements pour la création uniquement.

La création d'un index pour une table scalable consiste à créer un index sur chacun de ses segments. Nous avons vu dans la section précédente que la création des index est réalisée par la commande *sd_create_index*. Nous supposons que cette commande concerne la création de l'index scalable *test_index* sur la colonne *test* crée précédemment sur chaque segment de la table *PhotoObj*. Ainsi la commande sera exécutée comme suit :

sd_create_index 'test_index ON PhotoObj (test)'

Le gestionnaire SD-SQL Server suit les étapes ci-dessous pour effectuer les traitements internes de cette commande :

- Il suit les mêmes premières étapes que la commande *sd_alter_table* pour récupérer le nom de la table scalable *PhotoObj*, sa NDB primaire, sa NDB client initial ainsi que pour reformuler la clause *'test_index ON PhotoObj(test)'*. Après toutes ces étapes, la nouvelle requête reformulée *Q'* sera comme suit :

  'CREATE INDEX test_index ON SD._Ceria_PhotoObj (test)'

- Ensuite, il boucle sur la méta-table *SD.RP*, localisée sur la NDB primaire *Dell11.SkyServer*, pour chercher les localisations des segments qui constituent la table scalable *PhotoObj*. C'est aussi le même traitement effectué dans la section précédente. Pour chaque nœud trouvé (*Dell11* et *Dell10*), il accède à sa NDB *SkyServer* et exécute la requête *Q'*.

Le résultat de la commande *sd_create_index* est donc la création de l'index *'test_index'* sur la colonne *'test'* de chacun des segments de la table *PhotoObj* : *Dell11.SkyServer.SD._Ceria_PhotoObj* et *Dell10.SkyServer.SD._Ceria_PhotoObj*.

6.4.4 Suppression d'une Table Scalable

La suppression d'une table scalable est réalisée par la commande *sd_drop_table*. Cette commande permet d'exécuter la requête SQL *DROP TABLE* sur chaque segment composant la table scalable à supprimer. Elle permet aussi la suppression de toutes les métadonnées décrivant cette table. Nous reprenons l'exemple de la table scalable *PhotoObj* partitionnée en deux segments. Nous exécutons alors la commande *sd_drop_table* sur la NDB *Ceria.SkyServer* comme suit :

sd_drop_table 'PhotoObj'

Le gestionnaire suit les étapes suivantes afin d'effectuer les traitements internes de la commande de suppression :

- Il récupère le nom de la NDB qui détient le segment primaire de la table *PhotoObj*. Pour cela, il suit le traitement que nous avons déjà décrit précédemment. La NDB en question est *Dell11.SkServer*.

- Ensuite, le gestionnaire se place sur la méta-table *Dell11.SkyServer.SD.RP* et récupère toutes les NDBs des segments de la table *PhotoObj* comme nous l'avons décrit précédemment aussi. Ces NDBs sont *Dell11.SkyServer* et *Dell10.SkyServer*.
- Il commence par la suppression des segments *_Ceria_PhotoObj* des NDBs *Dell11.SkyServer* et *Dell10.SkyServer* ainsi que leur image primaire *PhotoObj* localisée sur la NDB courante de l'exécution de la commande, *Ceria.SkyServer*.
- Il supprime ensuite tous les tuples décrivant la table scalable *PhotoObj* dans les méta-tables. Il supprime les tuples suivants :
    - *(Dell11, Ceria, PhotoObj)* et *(Dell10, Ceria, PhotoObj)* de la méta-table *Dell11.SkyServer.SD.RP*.
    - *(Dell11, Ceria, PhotoObj)* des méta-tables *Dell11.SkyServer.SD.Prim* et *Dell10.SkyServer.SD.Prim*.
    - *(Ceria, PhotoObj, 10000)* des méta-tables *Dell11.SkyServer.SD.Size* et *Dell10.SkyServer.SD.Size*.
    - *(PhotoObj, Primary,Dell11,1)* de la méta-table *Ceria.SkyServer.SD.Image*.

6.4.5 Gestion des Images Secondaires

6.4.5.1 Création d'une Image Secondaire

La commande *sd_create_image* permet de créer une image secondaire pour une table scalable comme nous l'avons déjà présenté dans le chapitre précédent. Nous reprenons l'exemple de la Section 6.3.6 qui permet la création d'une image secondaire pour la table *PhotoObj* sur la NDB client *Dell8.SkyServer* :

```
sd_create_image 'Ceria', 'PhotoObj'
```

Nous rappelons que *PhotoObj* est partitionnée en deux segments localisés sur les NDBs *SkyServer* des nœuds *Dell10* et *Dell11* respectivement. Le gestionnaire SD-SQL Server suit les étapes ci-dessous pour effectuer le traitement interne de la commande *sd_create_image*.

- Tout d'abord, SD-SQL Server détermine la NDB qui détient le segment primaire de la table *PhotoObj*. En fait, cette NDB sert à définir la nouvelle image secondaire. Pour cela, il utilise le paramètre d'entrée *'Ceria'*. Ce dernier représente le nœud de la NDB client *SkyServer* qui détient l'image primaire *PhotoObj* ainsi que la méta-table *SD.Image*. En effet, le gestionnaire accède à cette méta-table et récupère la valeur du champ *PrimNd* du tuple correspondant à la table *PhotoObj*. Cette valeur sera, d'après notre exemple, le nœud *Dell11*.

- Ensuite, il formule le nom de l'image secondaire à partir du paramètre *Ceria*. Son nom sera alors *SD.Ceria_PhotoObj* comme nous l'avons déjà décrit.
- Ayant ces éléments, SD-SQL Server crée la vue partitionnée qui implémente l'image secondaire *Ceria_PhotoObj* sur la NDB client *Dell8.SkyServer* et définit son contenu comme suit :

```
CREATE VIEW SD.Ceria_PhotoObj AS
 SELECT * FROM Dell11.SkyServer.SD._Ceria_PhotoObj
 UNION ALL SELECT * FROM Dell10.SkyServer.SD._Ceria_PhotoObj
```

- Enfin, il insère le tuple *(PhotoObj, Secondary, Dell11, 2)* qui décrit l'image *Ceria_PhotoObj* dans la méta-table *Dell8.SkyServer.SD.Image*.

Notons que l'image secondaire correspond au partitionnement actuel de sa table scalable.

6.4.5.2 Suppression d'une Image

La commande *sd_drop_image* permet de supprimer une image secondaire d'une table scalable. Elle supprime aussi les méta-données qui la décrivent. Nous rappelons que cette commande ne supprime que les images secondaires car les images primaires ne sont supprimées que lors de la suppression de leurs tables scalables. Nous reprenons l'image secondaire *Ceria_PhotoObj* que nous avons créée dans la section précédente. La suppression de cette image est réalisée par la commande *sd_drop_image* exécutée sur la NDB de l'image elle même, *Dell8.SkyServer,* selon la syntaxe suivante :

```
sd_drop_image 'Ceria_PhotoObj'
```

Le gestionnaire SD-SQL Server effectue les traitements internes ci-dessous pour exécuter cette commande :

- Il récupère le nom de la table scalable à partir du nom de l'image secondaire à supprimer, puisque celui-ci est composé du nom de la table. Ce nom sera alors *PhotoObj*.
- Ensuite, Il supprime la vue partitionnée distribuée qui implémente l'image secondaire *SD.Ceria_PhotoObj* de la NDB courante *Dell8.SkyServer* de l'exécution de la commande.
- Enfin, il supprime le tuple *(PhotoObj, Secondary, Dell11,2)* décrivant l'image secondaire de la méta-table *Dell8.SkyServer.SD.Image*. Il utilise pour cela la requête SQL *DELETE* avec une clause *WHERE* sur le nom de la table scalable *PhotoObj* extrait dans la première étape.

6.4.6 Gestion des Requêtes Scalables

Dans ce qui suit, nous traitons les requêtes scalables concernant une recherche, une insertion, une mise à jour ou une suppression sur une table scalable. Les commandes SD-SQL Server qui correspondent à ces manipulations sont respectivement : *sd_select*, *sd_insert*, *sd_update* et *sd_delete*.

Le traitement interne de chaque requête scalable commence par la phase que nous avons appelé *phase d'analyse* ou *image binding*. Cette phase permet d'extraire les images interrogées dans une requête et de vérifier ensuite l'ajustement de chaque image extraite avant de passer la requête au SGBD SQL Server pour l'exécution.

Dans ce qui suit, nous détaillons le traitement interne des commandes d'accès aux tables scalables. Nous commençons par décrire la phase d'analyse des requêtes scalables que nous désignons dans tout ce qui suit par *image binding*. Ensuite, nous discutons les spécifications de chaque commande.

6.4.6.1 Image Binding

La phase d'*image binding* consiste à analyser le contenu d'une requête scalable et extraire les noms des tables ou des vues qu'elles interrogent. Chaque nom extrait peut être celui d'une image primaire, d'une image secondaire, d'une vue scalable (ou statique) ou d'une table statique. S'il s'agit du nom d'une image (ou d'une vue scalable), SD-SQL Server doit vérifier si cette image est correcte avant l'exécution de la requête qui l'interroge.

Dans cette phase, SD-SQL Server cherche les noms des tables et vues dans la clause *FROM* principalement, ou dans toute autre clause SQL (*INSERT*, *DELETE*, *UPDATE*) qui apparaît dans les paramètres des requêtes scalables. Il utilise pour cela la fonction *xp_sscanf* du système SQL Server ainsi que d'autres traitements correspondants. En lui indiquant la structure de la requête (avec ses clauses *SELECT*, *FROM*…etc.) comme argument, la fonction *xp_sscanf* permet de retourner tout le contenu de la clause utilisée comme il apparaît dans la requête. Nous expliquons ces traitements à travers la clause *FROM* (il s'agit des mêmes traitements pour les autres clauses). Le contenu de cette clause représente des objets séparés par des ',', avec des alias, etc.

SD-SQL Server analyse la chaîne de caractères retournée par la fonction *xp_sscanf* et récupère chaque nom d'objet utilisé dans la clause *FROM*. Pour chaque nom *X* trouvé, il procède comme suit :

- Il cherche le nom *X* dans la méta-table *Image* de la NDB courante (celle de l'exécution de la requête). S'il trouve un tuple dont l'attribut *Name* est égal à *X*, *X* correspond donc à une image.

- Si le gestionnaire ne trouve pas le nom *X* dans la table *Image*, il parcourt alors les tables systèmes du SGBD SQL Server *sysobjects* et s*ysdepends* pour vérifier si *X* est une vue scalable.
    - La table *sysobjects* fournit pour chaque objet son type et son identifiant interne *Id*. Le type d'un objet est soit une *vue* soit une *table*. Dans le premier cas, l'attribut *type* de la table *sysobjects* aura la valeur *'V'* et dans le deuxième cas, il aura la valeur *'T'*.
    - La table *sysdepends* fournit pour chaque objet *X* de type *'vue'*, selon son *Id*, ses objets dépendants. Ces objets peuvent être des tables ou des vues. La table *sysdepends* fournit les objets dépendants locaux uniquement. Autrement dit, les objets dépendants de *X* qui se trouvent sur d'autres serveurs distants ne seront pas fournis.

  Nous supposons que l'objet *X* est de type *vue*. En faisant la jointure des deux tables *sysobjects* et s*ysdepends*, SD-SQL Server peut avoir tous les objets qui dépendent de l'objet *X* Ensuite, pour chaque objet dépendant, il refait les traitements dès le début : vérifier s'il existe dans la méta-table *Image,* etc.

Une fois toutes les images utilisées dans la clause *FROM* sont extraites, le gestionnaire vérifie alors l'ajustement de chacune d'elles comme nous l'avons présenté dans la section 6.4.2.

6.4.6.2 La Recherche Scalable

La recherche dans une table scalable est réalisée par la commande *sd_select*. Cette commande invoque dans son paramètre d'entrée la clause SQL *SELECT* ou *SELECT INTO*. Dans le cas d'une requête *SELECT INTO*, SD-SQL Server ne permet pas le transfert de données d'une table vers une nouvelle table uniquement (comme le fait la clause habituelle *SELECT INTO)*. De plus, il permet la transformation de la nouvelle table créée en une table scalable. Pour cela, sd_select nécessite l'entrée des autres paramètres spécifiques à une table scalable. Il s'agit de la capacité de la nouvelle table scalable et de sa clé de partitionnement.

Nous appliquons la commande *sd_select* sur la table scalable *PhotoObj*. Nous supposons que cette table est partitionnée en deux segments sur *Dell11.SkyServer* et *Dell10.SkyServer* respectivement. Son image primaire est localisée sur la NDB client *Ceria.SkyServer*. Nous supposons que son image n'est pas ajustée, c'est-à-dire elle définit un seul segment de la table *PhotoObj*. Sa définition est comme suit :

```
CREATE VIEW PhotoObj AS
  SELECT * FROM Dell11.SkyServer.SD._Ceria_PhotoObj
```

Si nous voulons sélectionner les dix premiers tuples de la table scalable *PhotoObj*, nous exécutons alors la commande *sd_select* comme suit :

sd_select 'TOP 10 * FROM PhotoObj'

Les traitements internes de cette commande consistent à suivre les étapes suivantes :

- Le gestionnaire SD-SQL Server récupère le paramètre *'top 10 * from PhotoObj'* que nous désignons par *Q*. Il applique ensuite la phase de l'*image binding* sur cette requête.
- Lors de la phase d'*image binding*, il récupère l'image *PhotoObj* de la clause *FROM*. Il vérifie si cette image est ajustée comme nous l'avons déjà décrit. Puisque *PhotoObj* n'est pas ajustée d'après notre exemple, alors il l'ajuste.
- Une fois l'image est ajustée, SD-SQL Server reformule la clause *Q* avant de l'envoyer à SQL Server pour l'exécution. Il lui rajoute la verbe *SELECT* et elle devient alors :

  SELECT TOP 10 * FROM PhotoObj

Si la commande *sd_select* a un paramètre correspondant à la clause SQL *SELECT INTO*, le traitement interne de la commande changera. Pour expliquer le traitement interne dans ce cas, nous exécutons la commande *sd_select* sur la table scalable *PhotoObj* comme suit :

sd_select '* INTO Star FROM Dell1.SkyServerDB.dbo.Star', 500, 'Objid'

Cette commande permet de transférer toutes les données de la table statique *Star*, localisée sur la base de données *Dell1.SkyServerDB.dbo* (qui n'est pas une SDB), vers une nouvelle table scalable appelée *Star*. Le gestionnaire SD-SQL Server suit les étapes suivantes pour effectuer les traitements internes concernant cette commande :

- Tout d'abord, il applique la phase de l'*image binding* sur la requête scalable. Il vérifie s'il y a des images utilisées dans la requête. Si oui, il ajuste ces images en cas où elles ne sont pas ajustées. Ensuite, il enregistre la table *Star* comme une table scalable. Ainsi, *Star* aura un segment primaire et des entrées dans les méta-tables.
- Ensuite, il reformule la requête scalable sous forme de requête SQL compréhensible par SQL Server comme suit :

  SELECT * INTO Star FROM Dell1.SkyServerDB.dbo.Star

- Il passe cette requête à SQL Server pour l'exécution. Le résultat de l'exécution sera une nouvelle table *Star* créée sur une NDB *SkyServer* de type serveur. Le nœud de cette NDB est sélectionné à partir de la méta-table *SD.Server* de la NDB courante de l'exécution de la commande (*Ceria.SkyServer*). Nous supposons que le nœud de type serveur sélectionné est *Dell7*. Le segment primaire *_Ceria_Star* de la table

scalable *Star* sera alors créé sur la NDB *Dell7.SkysServer*. Les tuples de la table statique *Dell1.SkyServerDB.dbo.Star* seront alors transférés vers ce nouveau segment.

- Puisque la clause SQL *SELECT INTO* n'applique pas les clés primaires de la table initiale sur la nouvelle table crée, SD-SQL Server se charge alors de ce traitement. Une fois le segment primaire *Dell7.SkysServer.SD._Ceria_Star* est créé, le gestionnaire utilise le paramètre entré *'Objid'* qui désigne l'attribut clé de la nouvelle table. Cet attribut sera considéré comme une clé de partitionnement de la nouvelle table *Star*. Si la commande a en entrée plusieurs attributs clé, donc tous ces attributs seront pris en compte comme une clé primaire.
- Le gestionnaire crée aussi un déclencheur sur le segment *Dell7.SkysServer.SD._Ceria_Star*. Ce déclencheur fera appel à un éclateur comme nous l'avons déjà décrit. Il est défini comme suit :

```
CREATE TRIGGER split_trigger_Ceria_star ON SD._Ceria_star
  AFTER INSERT AS
  BEGIN
    EXEC msdb..sp_start_job 'splitter'
  END
```

- Une fois la table scalable *Star* est créée avec son segment primaire, SD-SQL Server enregistre les métadonnées qui la décrivent :
  - Il insère le tuple *(Dell7, Ceria, Star)* dans la méta-table *Dell7.SkyServer.SD.Prim*.
  - Il insère le tuple *(Ceria, Star, 500)* dans la méta-table *Dell7.SkyServer.SD.Size*. La troisième colonne *Size* égale à 500 correspond à la capacité entrée comme paramètre dans la commande *sd_select*.
  - Il insère le tuple *(Dell7, Ceria, Star)* dans la méta-table *Dell7.SkyServer.SD.RP*.
  - Il insère le tuple *(Star, Primary, Dell7, 1)* dans la méta-table *Ceria.SkyServer.SD.Image*.
- Enfin, il exécute le déclencheur pour vérifier si le segment créé n'excède pas sa capacité suite au transfert des tuples. Dans le cas affirmatif, le segment éclate et le résultat de la commande *sd_select* sera un ou plusieurs nouveaux segments de la table scalable *Star* (selon sa capacité).

6.4.6.2.1. Exemples

Nous présentons une liste d'exemples sur des requêtes de recherche scalable. Nous donnons les différents cas qui peuvent se présenter pour un utilisateur. Nous utilisons pour tous les exemples la table scalable *PhotoObj* initialement créée sur la NDB client *Ceria.SkyServer*. Nous supposons que cette table est partitionnée en deux segments sur les NDBs *Dell11.SkyServer* et *Dell10.SkyServer* respectivement. Comme dans nos exemples précédents, *Dell11.SkyServer* est la NDB qui détient le segment primaire de la table *PhotoObj*. Nous exécutons les commandes suivantes sur la NDB client *Ceria.SkyServer*.

- **Recherche scalable avec plusieurs objets dans la clause *FROM***

Soit la commande suivante :

```
sd_select '* FROM PhotoObj, T1'
```

D'après sa syntaxe, cette commande exécute une requête de sélection sur les tables *PhotoObj* et *T1*. Nous supposons que *PhotoObj* est une image primaire et *T1* est une table statique. La définition de l'image *PhotoObj* est comme suit :

```
CREATE VIEW PhotoObj AS
  SELECT * FROM Dell11.SkyServer.SD._Ceria_PhotoObj
```

En appliquant le traitement interne d'une requête de recherche scalable sur la commande *sd_select* ci-dessus, SD-SQL Server va tout d'abord récupérer les objets de la clause *FROM* à partir du paramètre d'entrée de la commande *sd_select*. Il s'agit de *PhotoObj* et *T1*. Il trouve que *T1* est une table statique et *PhotoObj* est une image d'une table scalable. Ainsi, il vérifie l'ajustement de *PhotoObj*. D'après sa définition, l'image *PhotoObj* ne définit pas réellement le partitionnement actuel de sa table scalable *PhotoObj*. Elle définit uniquement son segment primaire. Or, *PhotoObj* possède deux segments d'après notre hypothèse. SD-SQL Server ajuste alors l'image *PhotoObj* avant d'exécuter la requête l'interrogeant. Enfin, il exécute la requête.

- **Recherche scalable avec une fonction d'agrégation**

Soit la commande suivante :

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

Nous supposons que *T2*, utilisée dans cette commande, est une vue définie comme suit :

```
CREATE VIEW T2 AS SELECT * FROM PhotoObj
```

Lors du traitement interne effectué sur la commande *sd_select* ci-dessus, SD-SQL Server récupère l'objet *T2* de la clause entrée dans la commande *sd_select*. En trouvant que *T2* est une vue, alors il cherche si *T2* dépend d'une table scalable directement ou indirectement (à plusieurs niveaux). Comme *T2* dépend directement de l'image *PhotoObj*,

T2 sera alors considérée comme une vue scalable de niveau 1. Ainsi, SD-SQL Server vérifie si *PhotoObj* est ajustée ou non et ensuite il exécutera la commande.

- **Recherche scalable avec le mot clé *TOP* et des alias**

Nous supposons la commande *sd_select* suivante :

sd_select 'TOP 5 P.objid FROM PhotoObj AS P'

Cette commande correspond à une requête *SELECT* utilisant le mot clé *TOP* et des alias dans la clause *FROM*. Le résultat d'exécution de cette commande sera les cinq premiers tuples de la table scalable *PhotoObj*. Pour son traitement interne, SD-SQL Server suit les mêmes étapes précédentes.

6.4.6.3 Les Mises à Jour Scalables

Les mises à jour sur une table scalable sont réalisées par la commande *sd_update*. Nous supposons la mise à jour d'un tuple de la table *PhotoObj*. Nous exécutons alors la commande *sd_update* sur la NDB client *Ceria.SkyServer* comme suit :

sd_update 'PhotoObj set run=123 where Objid=2214566'

Le gestionnaire SD-SQL Server effectue les traitements internes de la commande *sd_update* en suivant les étapes ci-dessous :

- Tout d'abord, il récupère La clause entrée de la commande, que nous désignons par *Q*, et lui applique la phase de l'*image binding*. Il vérifie s'il y a des images utilisées dans *Q*. Il trouve l'image *PhotoObj*, il vérifie alors si elle est ajustée ou non. Nous supposons que l'image *PhotoObj* est ajustée depuis la commande précédente (Section 6.4.6.2).
- Ensuite, il génère une requête SQL *Q'* à partir de la clause *Q* comme suit :

  'UPDATE PhotoObj set run=123 where Objid=2214566'

  Il envoie *Q'* à SQL Server pour l'exécution. Le résultat de l'exécution sera la modification du tuple du segment de la table scalable qui possède la clé *Objid* égale à la valeur *2214566*. Le choix du segment qui possède cette clé sera selon la contrainte d'intégrité de chaque segment de la table scalable.

6.4.6.4 L'Insertion Scalable

L'insertion de nouveaux tuples dans une table scalable est réalisée par la commande *sd_insert*. Le traitement interne de cette commande suit les mêmes étapes que celles du traitement interne de la commande *sd_update*. Ce que nous pouvons rajouter dans cette section, est le fait qu'une insertion peut déclencher l'éclatement de la table qu'elle utilise.

6.4.7 Gestion des Nœuds, des SDBs et des NDBs

Nous traitons dans cette section les principales commandes de la gestion des nœuds, des SDBs et des NDBs. Notons l'utilisation de scripts SQL dans le traitement de ces commandes. En effet, nous avons écrit quatre scripts SQL : *msdb.sql*, *server.sql*, *client.sql* et *peer.sql*. Ces scripts contiennent les commandes SQL permettant la création des tables et des procédures stockées utilisées pour le traitement des commandes SD-SQL Server. Leur description est comme suit :

- **Le script *msdb.sql***

Ce script contient les procédures stockées qui permettent principalement la création et la suppression de bases de données sur un nœud SD-SQL Server distribué. En effet, la commande SQL *CREATE DATABASE* ne permet pas la création d'une base sur un autre nœud que le nœud courant de la création. Il en est de même pour la requête *DROP DATABASE*. Autrement dit, nous ne pouvons pas exécuter la requête *CREATE DATABASE*, par exemple, comme suit :

```
CREATE DATABASE Dell11.SkyServer
```

Nous avons alors utilisé des procédures stockées que nous pouvons préfixer et ainsi exécuté à distance. Ces procédures utilisent les requêtes SQL de création et de suppression de bases qui seront exécutées localement. Nous exécutons le script *msdb.sql* sur la base de données SQL Server *MSDB*. Nous avons choisi cette base parce qu'elle est créée par défaut sur chaque instance SQL Server lors de son installation. Ainsi, si un nouveau nœud SD-SQL Server est créé, pour pouvoir y créer des NDBs, il faudra utiliser les procédures stockées créées sur sa base *MSDB*.

- **Le script *server.sql***

Ce script contient les requêtes SQL *CREATE TABLE* permettant de créer les méta-tables d'une NDB de type serveur. Il contient aussi toutes les procédures stockées qui implémentent les traitements effectués sur une NDB serveur (éclatement, etc.). Nous exécutons ce script lors de la création de toute NDB de type serveur.

- **Le script *client.sql***

Ce script contient les requêtes SQL *CREATE TABLE* permettant de créer les méta-tables d'une NDB de type client. Il contient aussi toutes les procédures stockées qui implémentent les commandes SD-SQL Server. Nous exécutons alors ce script lors de la création de toute NDB de type client.

↻ **Le script *peer.sql***

Le contenu de script combine le contenu des deux scripts *server.sql* et *client.sql*.

Après avoir présenté les scripts SQL, nous passons à la description des traitements internes des commandes concernant la gestion des nœuds, des SDBs et des SDBs.

6.4.7.1 Création d'un Nœud

La création d'un nœud SD-SQL Server est réalisée par la commande *sd_create_node*. Cette commande est localisée sur la méta-base MDB. Elle est exécutée sur un serveur lié de SQL Server pour le transformer en un nœud SD-SQL Server. Nous supposons la création du nœud SD-SQL Server *Dell11* de type pair. Nous exécutons alors la commande *sd_create_node* comme suit :

```
sd_create_node 'Dell11', 'Server'
```

Le gestionnaire SD-SQL Server effectue les traitements internes suivants lors de l'exécution de la commande ci-dessus :

- Tout d'abord, il vérifie si le premier paramètre entré *Dell11* est un serveur SQL Server lié sinon la commande ne peut pas être exécutée.
- Ensuite, il vérifie si *Dell11* n'existe pas dans le système SD-SQL Server. Pour cela, il utilise la méta-table *SD.Nodes* de la MDB. Si *Dell11* n'existe pas, le gestionnaire insère alors le tuple *(Dell11,Server)* dans la méta-table *SD.Nodes*.
- Enfin, SD-SQL Server exécute à partir de la MDB le script SQL *msdb.sql*. Nous exécutons ce script sur la base de données *MSDB*, du nœud *Dell11*. L'exécution de ce script permettra la création de procédures stockées qui permettent la création et la suppression de NDBs sur le nouveau nœud *Dell11*. Ce script est exécuté en utilisant la commande suivante du système SQL Server:

```
EXEC master..xp_cmdshell 'osql /S dell11 /U sa /P
    /d msdb <\\Dell11\script\msdb.sql
```

Nous ne détaillons pas les traitements de la commande *sd_alter_node* puisque il s'agit juste de modifier le type du nœud dans la méta-table *SD.Nodes*.

6.4.7.2 Création d'une SDB

La création d'une SDB est réalisée par la commande *sd_create_scalable_database*. Nous supposons la création de la SDB *SkyServer* avec deux NDBs dont la NDB primaire est une NDB de type serveur localisée sur le nœud *Del117*. Ainsi, nous exécutons la commande *sd_create_scalable_database* comme suit :

```
sd_create_scalable_database 'SkyServer','Dell11', 'Server', 2
```

Le gestionnaire SD-SQL Server effectue les traitements internes de cette commande en suivant les étapes ci-dessous :

- Il exécute la procédure stockée *create_sdb* localisée sur la base système MSDB de SQL Server. Nous rappelons que cette procédure stockée a été créée sur MSDB lors de l'exécution du script SQL *msdb.sql* dans la création du nœud *Dell11*. La procédure *create_sdb* permet de créer une nouvelle base SQL Server sur le nœud *Dell11*. Cette base est la NDB primaire de la SDB *SkyServer*. La procédure stockée *create_sdb* est exécutée comme suit :

  EXEC Dell11.msdb.dbo.create_sdb 'SkyServer', 'Server'

- Selon le type de la NDB, la procédure *create_sdb* exécute un script SQL permettant de créer les méta-tables ainsi que les procédures implémentant les commandes liées à ce type de NDB. Dans notre exemple, la NDB créée (*Dell11)* est de type serveur, ainsi le gestionnaire exécutera le script *server.sql*. Ce dernier crée les procédures stockées ainsi que les méta-tables correspondant à une NDB serveur.
- Une fois la NDB primaire est créée avec tous ses composants, le gestionnaire insère alors les tuples suivants dans ses méta-tables :
  - Le tuple *(Dell11)* dans la méta-table *SD.NDB*. Ce tuple indique que le nœud *Dell11* détient la NDB *SkyServer* courante.
  - Le tuple *(Dell1)* dans la méta-table *SD.MDBNode* pour indiquer la localisation de la MDB.
- Ensuite, il garde trace de la NDB primaire *Dell11.SkyServer* dans la méta-base MDB. Pour cela, il insère le tuple *(SkyServer, Dell11, Server)* dans la méta-table *SD.SDB* de la MDB.
- Enfin, il vérifie s'il y a un nombre d'extension de la SDB *SkyServer* à créer. Dans notre exemple, nous avons entré deux extensions dans la commande *sd_create_scalable_database*. Ainsi, SD-SQL Server crée en plus de la NDB primaire, une autre NDB de la SDB *SkyServer*. Cette NDB aura le même type que la NDB primaire (c'est-à-dire de type serveur). Quant à sa localisation, elle sera déterminée à partir de la méta-table *SD.Nodes* de la MDB. Nous supposons que *Dell10* est un noeud disponible pour la création de l'autre extension (NDB) *SkyServer*. Les mêmes étapes décrites ci-dessus seront suivies pour sa création. De plus, il insère le tuple *(Dell10)* dans la méta-table *SD.NDB* de la NDB primaire *Dell11.SkyServer* pour indiquer le nouveau nœud hébergeant une NDB de la SDB *SkyServer*.

Le résultat de l'exécution de la commande *sd_create_scalable_database* est la SDB *SkyServer* avec une NDB primaire sur le nœud *Dell11* et une autre NDB serveur sur le nœud *Dell10*.

Si nous voulons créer d'autres NDBs pour la SDB *SkyServer*, nous exécutons la commande *sd_create_node_database*. Nous ne décrivons pas le traitement interne de cette dernière commande puisque ce sont les mêmes étapes suivies dans la création de la NDB primaire et ses extensions. Par contre, la suppression d'une NDB nécessite d'être détaillée et c'est ce que nous présentons dans la section suivante.

6.4.7.3 Suppression d'une NDB

La suppression d'une NDB est plus complexe que sa création. Trois cas se présentent dans la suppression des NDBs :

- Si la suppression concerne une NDB de type client, le gestionnaire doit alors supprimer toutes les tables scalables créées par les utilisateurs de cette NDB.
- Si la NDB à supprimer est de type serveur, le gestionnaire doit transférer tous ses segments et leurs métadonnées vers d'autres NDBs (de type serveur) de la même SDB avant la suppression de la NDB.
- Si la NDB à supprimer est de type pair, le gestionnaire agit comme dans le cas d'une NDB client et d'une NDB serveur en même temps.

Nous rappelons que la suppression d'une NDB est réalisée par la commande *sd_drop_node_database*. Nous supposons la suppression de la NDB *SkyServer* du nœud *Ceria* qui est de type client. Nous exécutons alors la commande *sd_drop_node_database* comme suit :

```
sd_drop_node_database 'SkyServer', 'Ceria'
```

Afin d'effectuer le traitement interne de cette commande, le gestionnaire SD-SQL Server suit les étapes suivantes :

- Tout d'abord, il récupère le nœud qui localise la NDB primaire *SkyServer*. Le nœud sélectionné est donc *Dell11* d'après nos exemples précédents. Il récupère ce nœud à partir de la méta-table *Ceria.SkyServer.SD.SDBNode*. Nous rappelons que cette méta-table insère le nom du nœud de la NDB primaire pour chaque NDB de la SDB courante.
- Il accède ensuite à la méta-table *SkyServer.SD.NDB* du nœud *Dell11*. Cette table contient tous les nœuds ayant les NDBs *SkyServer*. Il cherche dans cette table le nom du nœud de la NDB à supprimer, c'est-à-dire *Ceria*.
- Selon le type de la NDB à supprimer, SD-SQL Server effectue des traitements différents. Dans notre exemple, la NDB *Ceria.SkyServer* est de type client. Le gestionnaire supprime alors toutes les tables scalables créées par cette NDB.

- Pour cela, il accède à la méta-table *SD.Image* de la NDB *Ceria.SkyServer* pour récupérer les images primaires des tables scalables créées par cette NDB. Puisque cette NDB a lancé la création de la table *PhotoObj* d'après nos exemples précédents, donc nous trouvons le tuple décrivant l'image primaire *PhotoObj* dans la méta-table *SD.Image*.
- Ayant les images primaires de la méta-table *SD.Image*, SD-SQL Server commence la suppression de leurs tables scalables. Puisqu'il y a une seule image primaire *PhotoObj*. SD-SQL Server lance alors la suppression de la table scalable *PhotoObj* en faisant appel à la procédure stockée qui implémente la commande *sd_drop_table*. Les traitements de cette commande sont dans la Section 6.4.4.
- Ensuite, il supprime le tuple décrivant la NDB *Ceria.SkyServer* de la méta-table *Dell11.SkyServer.SD.NDB*.
- Enfin, il supprime la base de données *SkyServer* correspondant à la NDB du nœud *Ceria*. Pour cela, il exécute la procédure stockée *sd_drop_ndb* comme suit :

  msdb.dbo.sd_drop_ndb 'SkyServer', 'Ceria'

  Cette procédure a été créée sur la base *MSDB* de chaque nœud SD-SQL Server en exécutant le script *msdb.sql* comme nous l'avons déjà présenté. Elle fera appel à la requête SQL *DROP DATABASE* pour supprimer la NDB *SkyServer* en question.

Ces traitements sont effectués si la NDB à supprimer est de type client. Maintenant, si la NDB à supprimer est de type serveur, d'autres traitements sont effectués. Nous supposons la suppression de la NDB primaire *Dell10.SkyServer* par exemple.

- Tout d'abord, le gestionnaire cherche un nœud SD-SQL Server pour allouer une nouvelle NDB de la SDB *SkyServer*. Toutes les tables scalables de la NDB *Dell10.SkyServer* (qui sera supprimée) seront transférées vers cette nouvelle NDB. SD-SQL Server commence par chercher dans la méta-table *SD.Nodes* de la MDB s'il y a un nœud qui n'a pas encore hébergé une NDB de la SDB *SkyServer*. Pour cela, il exécute une jointure avec les méta-tables *SD.SDB* (de la méta-base MDB) et *SD.NDB* (de la NDB primaire *SkyServer)*. La requête SQL de cette jointure est la suivante :

  SELECT Node FROM Dell1.MDB.SD.Nodes WHERE Node NOT IN
  (SELECT Node FROM Dell11.SkyServer.SD.NDB)

- S'il existe un nœud serveur (ou pair) qui ne détient pas la NDB *SkyServer*, le gestionnaire utilise ce nœud pour y créer une nouvelle NDB *SkyServer* et lui transférer les données de la NDB *Dell10.SkyServer*. Sinon, il doit créer un nouveau noeud SD-SQL Server pour héberger la nouvelle NDB. Nous supposons qu'il existe déjà un nœud SD-SQL Server, soit *Dell7* ce noeud. Ainsi SD-SQL Server crée une nouvelle NDB *SkyServer* de type serveur sur ce nœud avec la commande

sd_create_node_database. Un nouveau tuple correspondant à la nouvelle NDB *Dell7.SkyServer* sera inséré dans la méta-table *SD.NDB* de la NDB primaire *Dell11.SkyServer*.

- Une fois la nouvelle NDB *SkyServer* est créée sur le nœud *Dell7*, le gestionnaire commence le transfert des données. Il transfère tous les tuples des méta-tabes *SD.Prim* et *SD.Size* de la NDB *Dell10.SkyServer* à supprimer vers les méta-tables correspondantes de la nouvelle NDB *Dell7.SkyServer*. Il transfère aussi tous les segments vers la nouvelle NDB.
- Ensuite, le gestionnaire supprime la NDB *SkyServer* du nœud *Dell10*. Il supprime aussi les tuples qui la décrivent dans les méta-tables *SD.NDB* de la NDB primaire *Dell11.SkyServer*.

Si la NDB à supprimer est une NDB primaire, d'autres traitements sont effectués. Nous supposons par exemple la suppression de la NDB primaire *Dell11.SkyServer*. SD-SQL Server suit le même traitement interne que celui de la suppression d'une NDB serveur non primaire. Les tuples des méta-tables *NDB*, *Prim* et *Size* sont donc transférés vers les mêmes méta-tables de la nouvelle NDB (*Dell7.SkyServer* par exemple). La seule différence est que tous les tuples de la méta-table *Dell11.SkyServer.SD.RP* seront transférés vers la méta-table *SD.RP* de la nouvelle NDB *Dell7.SkyServer*. De plus, tous les tuples qui ont dans leur champ *SgmNd* la valeur *Dell11* (l'ancien nœud primaire), ils auront à la place le nœud de la nouvelle NDB (*Dell7*).

Une fois toutes les données sont transférées vers la nouvelle NDB, le tuple décrivant la NDB *Dell11* dans la méta-table *MDB.SD.SDB* sera mis à jour. Il aura la valeur *Dell7* au lieu de l'ancienne valeur '*Dell11'* dans le champ *Node*. Ceci indique une nouvelle localisation pour la NDB primaire de la SDB *SkyServer*. Enfin, la NDB *SkyServer* sera supprimée du nœud *Dell11*.

6.4.7.4 Suppression d'une SDB

La suppression d'une SDB est réalisée par la commande *sd_drop_scalable_database*. Nous supposons la suppression de la SDB *SkyServer*. Nous rappelons que cette SDB est composée des NDBs *SkyServer* localisées sur les nœuds *Dell11*, *Dell7*, *Dell10* et *Ceria*. La commande de suppression est exécutée comme suit :

```
sd_drop_scalable_database 'SkyServer'
```

Le gestionnaire SD-SQL Server supprime uniquement les NDBs serveurs et pairs de la SDB *SkyServer*. Les NDBs de type client peuvent être éventuellement indisponibles lors de la suppression de leur SDB (en raison d'une panne, etc.). Ainsi, elles ne seront supprimées que lors de leurs accès. Le gestionnaire effectue les traitements internes de la commande *sd_drop_scalable_database* en suivant les étapes ci-dessous :

- Tout d'abord, il récupère le nœud qui détient la NDB primaire de la SDB *SkyServer*. Pour cela, il accède à la méta-table *SD.SDB* de la méta-base, il cherche le tuple où le champ *SDB_name* est égal à *'SkyServer'* et il récupère la valeur de son champ *Node* qui est dans notre exemple *'Dell11'*.
- Ensuite, il accède à la méta-table *SD.NDB* de la NDB primaire *Dell11.SkyServer*. Pour chaque tuple de cette table, il sélectionne son nœud. Chaque nœud sélectionné représente la localisation de chaque NDB de la SDB *SkyServer*.
- Pour chaque nœud trouvé dans la méta-table *SD.NDB*, le gestionnaire supprime sa NDB *SkyServer* en faisant appel à la procédure qui implémente la commande *sd_drop_node_database*.
- Enfin, il supprime le tuple décrivant la SDB *SkyServer* de la méta-table SDB de la métabase.

Ainsi, la suppression d'une SDB entraîne la suppression de toutes ses NDBs de type pair et serveur. Pour les NDBs de type client, nous rappelons qu'elles ne sont pas supprimées au moment de la suppression de leur SDB. Ceci est à cause de leur éventuelle indisponibilité. Ainsi la NDB client sera supprimée au moment de son utilisation plus tard lorsqu'il il s'avère que sa SDB n'existe plus.

6.4.7.5 Suppression d'un Nœud

La suppression d'un nœud est réalisée par la commande *sd_drop_node*. Nous supposons la suppression du noeud *Dell8* par exemple. Le gestionnaire suit les étapes suivantes pour effectuer le traitement interne de la commande *sd_drop_node* :

- Il sélectionne tout d'abord un nœud libre de la méta-table *Nodes* de la MDB. Soit *Dell5* ce nœud. Ensuite, il supprime le tuple qui enregistre le nœud *Dell8* de la même méta-table *Nodes*.
- Le gestionnaire déplace ensuite chaque NDB du nœud *Dell8* avec toutes ses méta-tables vers le nouveau nœud *Dell5*. Ceci entraîne aussi la modification des données de quelques tuples des méta-tables. En effet, le gestionnaire boucle sur la méta-table *Dell8.SkyServer.SD.NDB* et récupère chaque NDB trouvée. Ensuite pour chaque NDB trouvée (à l'exception de la NDB *SkyServer* courante du nœud *Dell8*), il récupère ses données et les transfère vers le nouveau nœud *Dell5*.
- Enfin, le gestionnaire supprime le nœud *Dell8*. Il modifie ensuite *Dell8* par *Dell5* dans la méta-table *Nodes* de la MDB. Il modifie aussi la SDB qui a une NDB localisée sur le nœud *Dell8*.

6.5 Gestion des Concurrences

Une grande partie des applications sur les SGBDs ne peuvent se permettre d'exécuter les programmes et requêtes de leurs utilisateurs les uns après les autres, car cela impliquerait des temps d'attente trop longs. Les SGBDs doivent donc exécuter simultanément, autant que possible, les programmes et requêtes des utilisateurs. Cette simultanéité d'exécution est appelée *concurrence.*

Comme tous les SGBDs, SD-SQL Server a aussi un environnement concurrentiel. La perte d'une requête scalable et l'écrasement de celle-ci par une autre requête concurrente peut être très fréquent. Une mise à jour d'une table scalable, par exemple, peut intervenir au milieu d'un éclatement de la même table. Ces deux traitements (la mise à jour et l'éclatement) sont donc en concurrence parce qu'elles manipulent des données communes (la même table scalable) avec des opérations incompatibles.

Dans ce qui suit, nous décrivons le comportement du système SD-SQL Server lorsque deux traitements, ou plus, essaient d'accéder aux mêmes données au même moment. Le but dans cette situation est de permettre un accès efficace pour toutes les sessions tout en maintenant une intégrité stricte des données.

Avant de voir comment nous avons procédé pour la gestion de concurrence sur SD-SQL Server, nous présentons d'abord quelques notions sur les techniques de verrouillage sur lesquelles est basée la gestion de concurrence.

6.5.1 Techniques de Verrouillage

Toute unité de traitement (programme ou requête utilisateur), exécutée sur un SGBD, est appelée *transaction.* Une transaction est dite *correcte* si en s'exécutant seule (sans concurrence) sur une base de données cohérente (dont toutes les contraintes d'intégrité sont vérifiées), fournit en résultat un état cohérent de la base de données. Afin de conserver la cohérence de la base lors de l'exécution simultanée des transactions, des techniques de gestion de concurrence sont utilisées. Il s'agit tout particulièrement de la technique de verrouillage, celle que nous utiliserons par la suite dans notre travail. Cette technique est la plus ancienne et la plus couramment utilisée pour contrôler la concurrence des accès à des objets partagés [Gar99].

Les techniques de verrouillage ont pour principe que les transactions voulant travailler sur un élément de la base, doivent auparavant demander à obtenir le droit d'utiliser cet élément. Ce droit est matérialisé par l'obtention d'un verrou. Si l'élément n'est pas disponible pour ce type d'usage, alors la transaction est mise en attente. Une fois le travail effectué, la transaction libère le verrou sur l'élément, qui devient disponible. L'élément,

unité sur laquelle on pose un verrou, est en général un tuple. Deux types de verrous sont à considérer généralement :

- o les *verrous en lecture* ou *verrous partagés* qui sont destinés à protéger les opérations de lecture sur l'objet verrouillé ;
- o et les *verrous en écriture* ou *verrous exclusifs* qui sont destinés à protéger les opérations d'écriture.

L'existence d'un verrou en écriture est par définition incompatible avec celle d'un autre verrou de type quelconque sur le même objet. Les règles suivantes, illustrées dans la Table 6-2, sont ainsi appliquées pour les demandes d'acquisition de verrous sur une donnée en fonction des verrous qui y sont actuellement posés. La mention impossible dans la table signifie que le demandeur sera bloqué jusqu'à ce que le verrou conflictuel soit levé.

Demande de verrou(s) déjà posé(s)	**En lecture**	**En écriture**
Aucun	Possible	possible
En lecture	Possible	Impossible
En écriture	impossible	Impossible

Table 6-2 : Règles d'utilisation des verrous

6.5.1.1 Isolation des Transactions

Les transactions spécifient un niveau d'isolement. Ce niveau définit le degré d'isolement d'une transaction par rapport aux modifications de ressources ou de données apportées par d'autres transactions. Les niveaux d'isolement déterminent les effets secondaires de la concurrence (lectures incorrectes, lectures fantômes) qui sont autorisés. Le standard ANSI/ISO SQL définit quatre niveaux d'isolation des transactions en termes de trois anomalies qui doivent être évités entre les transactions concurrentes. Ces phénomènes indésirables sont :

- o *La lecture impropre* (ang. *dirty read*) où une transaction lit des données écrites par une transaction concurrente libre.
- o *La lecture non répétable* (ang. *non-repeatable read*) où une transaction relit des données qu'elle a précédemment lues et trouve que les données ont été modifiées par une autre transaction non libre.
- o *La lecture fantôme* (ang. *phantom reads*) où une transaction ré-exécute une requête en renvoyant un ensemble de lignes qui satisfont une condition de recherche et

trouve que les lignes additionnelles satisfaisant la condition ont été insérées par une autre transaction non libre.

Les quatre niveaux d'isolation et leur comportement correspondant sont décrits dans la table ci-dessous [4]:

Niveau d'isolement	Lecture incorrecte	Lecture non renouvelable	Fantôme
Lecture non validée	Oui	Oui	Oui
Lecture validée	Non	Oui	Oui
Lecture renouvelable	Non	Non	Oui
Sérialisable	Non	Non	Non

Table 6-3 : Niveaux d'isolation

Après avoir présenté les notions de base dans la gestion de concurrence, nous présentons dans ce qui suit la gestion de concurrence sur SD-SQL Server.

6.5.2 Gestion des concurrences sur SD-SQL Server

Afin de gérer les différents conflits entre les opérations concurrentes sur SD-SQL Server, nous avons défini une matrice qui détermine les conflits entre les différentes commandes. Nous avons appelé cette matrice, « *matrice des conflits* ». En effet, nous avons proposé des schémas de concurrence entre toutes les transactions qui peuvent créer des conflits en s'exécutant simultanément. Ces schémas permettent l'exécution des transactions concurrentes tout en préservant leur sérialisabilité.

Tout d'abord, nous présentons la matrice des conflits entre les différentes opérations SD-SQL Server. Ensuite, nous décrivons les schémas de concurrence que nous avons proposés.

6.5.2.1 Matrice des Conflits

La matrice des conflits, que nous avons appelée *C*, est une matrice carrée de dimension *m* x *m* où *m* est le nombre d'opérations concurrentes sur SD-SQL Server. Les colonnes et les lignes de la matrice *C* correspondent à toutes les commandes SD-SQL Server. Notons qu'en plus des commandes SD-SQL Server, nous avons aussi l'éclateur (*splitter*) comme

[4] Ces niveaux d'isolation sont tous pris en charge par le SGBD SQL Server 2000.

élément de la matrice *C*. En effet, l'éclateur est déclenché par une insertion réalisée par la commande *sd_insert,* mais comme il est exécuté dans un agent asynchrone, il risque donc d'être en conflit avec d'autres commandes ou éclateurs. Nous énumérons les opérations qui composent la matrice comme suit :

(1) sd_create_node

(2) sd_drop_node

(3) sd_alter_node

(4) sd_create_scalable_database

(5) sd_create_node_database

(6) sd_drop_node_database

(7) sd_drop_scalable_database

(8) sd_create_table

(9) sd_alter_table

(10) sd_create_index

(11) sd_drop_index

(12) sd_drop_table

(13) sd_create_image

(14) sd_drop_image

(15) sd_select

(16) sd_insert

(17) sd_update

(18) sd_delete

(19) Splitter

Un élément C_{ij} de la matrice *C* qui lie la ligne *i* à la colonne *j* indique si la commande *i* peut être en conflit avec la commande *j*. Cet élément peut avoir la valeur *'Oui'* (désignée par *'O'*) pour dire que la commande *i* peut être en conflit avec la commande *j*. S'il n'y a pas de conflits entre deux éléments *i* et *j*, la case de l'élément C_{ij} reste vide. Ainsi, la matrice se présente comme suit :

$$C = \begin{array}{c|ccccccccccccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 \\ \hline 1 & & & & & & & & & & & & & & & & & & & \\ 2 & & & & & & & & & & & & & & & & & & & \\ 3 & & & & & & & & & & & & & & & & & & & \\ 4 & & & & & & & & & & & & & & & & & & & \\ 5 & & & & & & & & & & & & & & & & & & & \\ 6 & & & & & & O & O & & O & & & O & & & & & & & O \\ 7 & & & & & & O & O & & O & & & O & & & & & & & O \\ 8 & & & & & & & & & & & & & & & & & & & \\ 9 & & & & & & O & O & & O & O & O & O & & & & O & O & O & O \\ 10 & & & & & & O & O & & O & O & O & O & & & & O & O & O & O \\ 11 & & & & & & O & O & & O & O & O & O & & & & O & O & O & O \\ 12 & & & & & & O & O & & O & O & O & O & & & & O & O & O & O \\ 13 & & & & & & & & & O & O & O & O & O & O & O & O & O & O & O \\ 14 & & & & & & & & & O & O & O & O & O & O & O & O & O & O & O \\ 15 & & & & & & & & & O & O & O & O & O & O & O & O & O & O & O \\ 16 & & & & & & & & & O & O & O & O & O & O & O & O & O & O & O \\ 17 & & & & & & & & & O & O & O & O & O & O & O & O & O & O & O \\ 18 & & & & & & & & & O & O & O & O & O & O & O & O & O & O & O \\ 19 & & & & & & O & O & & O & O & O & O & O & O & O & O & O & O & O \end{array}$$

Figure 6-3 : Matrice des conflits

Dans le cas où l'élément C_{ij} a la valeur *'O'*, ceci indique qu'il y a une ressource commune qui pourrait générer un conflit entre deux commandes *i* et *j* se terminant par une perte d'intégrité de la table, si la concurrence n'était pas bien gérée. Cette ressource commune est un composant de SD-SQL Server, une méta-table par exemple. Pour gérer la concurrence sous SD-SQL Server, nous avons implémenté les opérations potentiellement concurrentes sous forme de transactions distribuées du standard SQL. La première approche pourrait être l'emploi du niveau *SERIALIZABLE*. En effet, le niveau d'isolation par défaut sur SQL Server 2000 est la lecture validée (*READ COMMITTED)*. Ainsi, chaque transaction sous SD-SQL Server peut garantir au moins que le cas de lecture impropre ne peut se produire. Or, ce niveau n'est pas très sûr pour assurer la cohérence. Le deuxième niveau fourni par SQL Server évite les trois types d'incohérence présentés précédemment. Il s'agit du niveau *SERIALIZABLE* qui signifie que les requêtes sont exécutées comme si elles étaient lancées les unes après les autres. Le premier inconvénient avec ces transactions très sûres est qu'elles font baisser les performances du système. Des transactions commencées, bloquant l'accès aux données, devront être terminées avant que la transaction *"sérialisable"* ne se poursuive.

Dans notre implantation, nous avons évité l'utilisation du niveau d'isolation *SERIALIZABLE* sur SD-SQL Server, ceci est à cause du cas suivant qui peut se présenter:

Si un éclateur aurait accédé à la table *SD.RP* pour effectuer une écriture (il y insère alors toujours un tuple d'un nouveau segment d'une table scalable), alors il poserait un verrou exclusif sur la totalité de la table *RP*. Si entre temps, une commande *sd_select* par exemple, veut accéder à la même table *RP* pour une lecture (récupérer le partitionnement actuel d'une table scalable pour l'ajustement de son image par la suite), cette commande

pose alors un verrou partagé sur *RP*. Cependant, elle trouve le verrou exclusif de l'éclateur sur *RP*, donc la commande *sd_select* sera mise en attente jusqu'à ce que l'éclateur termine, même si la lecture concerne des tuples non utilisé par l'éclateur (des tuples décrivant une autre table scalable).

Afin d'éviter cet inconvénient, nous gérons la concurrence sur SD-SQL Server en traitant ses commandes comme des transactions distribuées au niveau d'isolation *REPEATABLE READ*. Avec ce niveau d'isolation, le conflit avec l'éclateur, présenté ci-dessus, sera évité. En effet, l'éclateur pose son verrou exclusif sur un tuple uniquement (celui décrivant le segment qui éclate) de la table *RP*. Ainsi, une commande *sd_select* pourra poser son verrou partagé sur d'autres tuples de la même table *RP* sans aucune attente inutile. Les autres conflits sont aussi évités avec le niveau d'isolation *REPEATABLE READ* comme nous le montrerons par la suite.

La modification d'un niveau d'isolation est réalisée par l'instruction *SET TRANSACTION ISOLATION LEVEL*. Au niveau de chaque procédure stockée qui implémente une commande SD-SQL Server, nous avons limité son code par les instructions *BEGIN DISTRIBUTED TRANSACTION* et *COMMIT TRANSACTION*.

Après avoir présenté comment nous gérons la concurrence sur SD-SQL Server d'une manière générale, nous passons dans ce qui suit aux détails sur les différents conflits qui peuvent exister et comment notre solution les résout. Nous présentons aussi les principaux composants du système SD-SQL Server qui peuvent être des ressources partagées entre des transactions concurrentes.

6.5.2.2 Accès à la Méta-table *RP*

La méta-table *RP* est souvent une source de conflit entre des transactions distribuées sur SD-SQL Server. En effet, elle peut être une ressource partagée entre les transactions suivantes :

- Un éclateur d'une table scalable. Celui-ci accède à la méta-table *RP* pour insérer des tuples décrivant les nouveaux segments résultant de l'éclatement d'une table. Ainsi, tout éclateur effectue une écriture sur *RP*, et ceci nécessite un verrou exclusif. Comme nous l'avons déjà mentionné, ce verrou exclusif est généré seulement sur les tuples décrivant la table scalable qui éclate afin que les autres tuples utilisées par d'autres transactions ne soient pas bloqués.
- Une commande sur une table scalable. Cette commande peut être une commande de modification du schéma d'une table scalable, de recherche, de mise à jour ou une commande sur les images secondaires. En effet, les commandes *sd_alter_table, sd_create_index, sd_drop_index* et *sd_drop_table* accèdent toutes à la méta-table *RP* pour récupérer les localisations des segments de la table scalable utilisée. Ainsi,

ces commandes effectuent une lecture sur *RP*. De même, les commandes *sd_select, sd_insert, sd_update* et *sd_delete* accèdent à la table *RP* pour vérifier le partitionnement actuel des tables scalables correspondant aux images qu'elles adressent. Ainsi, toutes ces commandes génèrent alors un verrou partagé sur *RP*.

Dans ce qui suit, nous présentons les conflits qui peuvent être produits entre deux transactions (de celles présentées ci-dessus), et comment nous avons procédé pour garder leur séraialisabilité :

- **Concurrence entre un éclateur et une commande**

Si une transaction représentant un éclateur et une autre transaction, représentant une des commandes ci-dessus, essaient d'accéder en même temps à la même table *RP*, nous voulons clairement que la deuxième transaction commence à partir de la mise à jour de la table *RP* par l'éclateur. En effet, ce dernier insère de nouveaux tuples dans *RP* pour représenter de nouveaux segments dans la table scalable qui éclate. Nous voulons que ces segments (représentés par les tuples dans *RP*) soient pris en compte par une autre commande utilisant la même table scalable. Comme nous utilisons le niveau d'isolation *REPEATABLE READ*, ces transactions distribuées peuvent s'exécuter sans aucun conflit. Par contre, si une commande accède à *RP* avant l'éclateur, elle s'exécutera sans prendre en compte les nouveaux segments de l'éclatement. Si l'éclateur trouve le verrou partagé d'une commande, il attendra que la commande termine avec *RP*, ensuite il génère son verrou exclusif sur le tuple en question.

- **Concurrence entre deux éclateurs**

Si deux éclateurs de deux tables scalables différentes accèdent en même temps à la méta-table *RP*, le niveau d'isolation *REPEATABLE READ* évitera tout conflit entre ces deux transactions. En effet, si le premier éclateur accède à *RP*, donc il pose son verrou exclusif uniquement sur les tuples de la table qu'il utilise. Ainsi, si le deuxième éclateur arrive pour utiliser *RP*, il pose son verrou exclusif sur d'autres tuples de la table *RP*, sans aucun problème.

Deux éclateurs de la même table scalable peuvent utiliser aussi la même table *RP*. Il s'agit, des éclatements de deux segments de la même table scalable. Dans ce cas, si un éclateur trouve un verrou exclusif sur les tuples qu'il va utiliser, donc il va attendre que l'autre éclateur termine son traitement avec *RP*, pour qu'il puisse poser à son tour son verrou exclusif. Ainsi, nous éviterons tout problème sur la définition des contraintes d'intégrité sur les segments de la même table scalable.

- **Concurrence entre deux commandes**

Deux commandes SD-SQL Server sur une image de la même table scalable ne peuvent pas être en conflit puisque chaque commande génère un verrou partagé sur les tuples de *RP* qu'elle utilise. De plus, même si ces commandes concernent la même table scalable, aucun conflit ne se produira. Nous supposons une commande *sd_select* par exemple, qui génère un verrou partagé sur des tuples de *RP*. Si une autre commande sur la même table, *sd_update* par exemple, veut sélectionner les mêmes tuples sur *RP*, elle trouve le verrou partagé de *sd_select*, donc elle pourra déposer elle aussi son verrou partagé sans aucun problème.

6.5.2.3 Accès à la méta-table *Image*

La méta-table *Image* peut être une source partagée entre toutes les commandes qui effectuent un ajustement d'image. Ces commandes sont : *sd_select, sd_insert, sd_update* et *sd_delete*. Chacune de ces commandes utilise la méta-table *Image* pour récupérer le nombre de ses segments et si nécessaire ajuster ce nombre. Pour cela, un verrou partagé puis un autre exclusif sont nécessaires sur le tuple qui décrit l'image interrogée dans ces commandes.

Ces commandes sont exécutées comme des transactions distribuées, au niveau d'isolation *REPEATABLE READ*, afin d'éviter tout conflit qui peut se produire en particulier les conflits suivants :

- **Concurrence entre deux commandes exécutées sur différentes images**

Nous supposons une commande, *sd_select* par exemple, qui génère un verrou exclusif sur le tuple de l'image qu'elle interroge. Si une autre commande *sd_select* sur une image différente veut accéder à la table *Image*, elle dépose donc son verrou exclusif (ou partagé) sur le tuple décrivant l'image qu'elle utilise sans aucun conflit.

- **Concurrence entre deux commandes exécutées sur la même image**

Si deux commandes utilisent le même tuple de la table *Image*, donc la première commande qui arrive génère son verrou exclusif sur ce tuple. Quant à la deuxième commande, elle attend que la première commande termine son traitement pour qu'elle puisse à son tour poser son verrou exclusif sur le même tuple. Lorsqu'il s'agit d'une lecture du même tuple dans *Image*, chaque commande génère donc son verrou partagé sur le même tuple sans aucun problème. Cependant, deux transactions sur la même image, *sd_select* par exemple, peuvent être mises en attente mutuellement quand elles ne peuvent obtenir un verrou exclusif sur le tuple utilisé dans *Image*. Ceci les entraîne à un interblocage (ang. *deadlock*). Dans un tel cas, SQL Server interviendra pour avorter ce deadlock. L'exemple ci-dessous montre cet interblocage.

Exemple

Soient A et B les deux commandes suivantes :

A : sd_select '* FROM PhotoObj'

B : sd_select 'COUNT(*) FROM PhotoObj'

Nous présentons les traitements effectués par ces deux transactions comme suit :

- Récupérer le nom de l'image *PhotoObj* dans la phase de l'*image binding* → Lire (Image).
- Récupérer le nombre de segments dans *RP* → Lire (RP).
- Récupérer le nombre de segments *PhotoObj* dans *Image* → Lire (RP).
- Modifier le nombre de segments *PhotoObj* dans *Image* → Ecrire (Image).

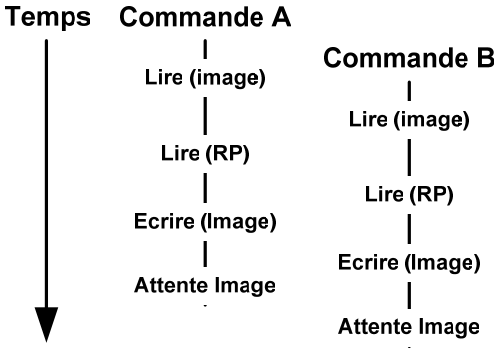

Figure 6-4 : Exécution de deux commandes utilisant le même tuple dans *Image*

Dans la figure ci-dessus, la transaction A peut être dans le conflit suivant : elle demande un verrou exclusif pour modifier le tuple dans *Image*, mais elle trouve le verrou partagé de la transaction B.

6.5.2.4 Les Segments d'une Table Scalable

Un segment d'une table scalable peut être une ressource partagée entre plusieurs transactions. Il s'agit en particulier des transactions suivantes :

- Un éclateur peut bloquer un segment qui éclate afin d'utiliser sa clé de partitionnement et lui définir une contrainte d'intégrité par la suite.
- Une commande de modification du schéma d'une table scalable. Il s'agit des commandes sd_*alter_table, sd_create_index, sd_drop_index* et *sd_drop_table*.

Chacune de ces commandes peut bloquer un segment pour lui rajouter (ou supprimer) une colonne, ajouter (ou supprimer) un index, etc.

Pour gérer la concurrence entre ces transactions, nous les exécutons au niveau d'isolation *REPEATABLE READ*. Chaque transaction génère un verrou exclusif sur le segment utilisé. Ceci est afin d'éviter les conflits des concurrences suivantes :

- **Concurrence entre deux éclateurs**

La concurrence entre deux éclateurs de deux tables scalables différentes ne pose aucun conflit puisque chaque table scalable a ses propres segments. Ainsi, le verrou exclusif d'un éclateur sur son segment n'empêche pas un autre éclateur de générer son verrou exclusif sur son propre segment. Cependant, deux éclatements de deux segments de la même table scalable peuvent produire un conflit. Pour éviter ce conflit, le premier éclateur bloque son segment avec un verrou exclusif. Ainsi, le deuxième éclateur sera mis en attente jusqu'à ce que le premier termine sa tâche.

- **Concurrence entre un éclateur et une commande**

Si un éclateur commence par générer son verrou exclusif sur un segment éclatant, toute la table scalable correspondante sera bloquée. En effet, si une commande *sd_alter_table* par exemple, s'exécutant sur la même table scalable veut générer son verrou exclusif sur le même segment afin de le modifier, elle sera mise en attente jusqu'à ce que l'éclateur libère ce segment. Ainsi, s'il y a des nouveaux segments résultant de l'éclatement, ils seront pris en compte par la commande *sd_alter_table*. De même, si cette commande commence sa tâche avant l'éclateur, c'est donc l'éclateur qui sera mis en attente. Dans les deux cas, ces deux transactions s'exécutent d'une manière sérialisable donc cohérente.

- **Concurrence entre deux commandes**

Nous supposons que la commande *sd_alter_table* est exécutée pour modifier le type d'un attribut d'une table scalable. Si elle commence en premier, elle génère donc un verrou exclusif sur chaque segment de la table scalable qu'elle utilise. Si une autre commande *sd_create_index* par exemple, s'exécute sur la même table scalable utilisée par la commande *sd_alter_table*, elle doit donc attendre que cette commande termine son traitement.

6.5.2.5 Accès à la méta-table *Primary*

La méta-table peut être une ressource partagée par les transactions suivantes :

- Un éclateur peut utiliser la méta-table *Primary* si le segment qui éclate est un segment secondaire. Il accède à cette méta-table pour récupérer la NDB du segment primaire correspondant et il accède ensuite à sa méta-table *RP*. Ainsi, un éclateur

peut effectuer une lecture sur la méta-table *Primary*. Pour cela, il génère un verrou partagé sur le tuple qu'il utilise dans cette méta-table.

- Une commande de suppression d'une table scalable *sd_drop_table*, ou d'une NDB *sd_drop_node_database*, utilisent la méta-table *Primary* pour leur traitement. La première commande génère un verrou exclusif sur le tuple décrivant la table scalable à supprimer et elle supprime bien sûr ce tuple par la suite. La deuxième commande génère aussi un verrou exclusif sur les tuples qu'elle utilise dans *Primary*. En effet, la commande *sd_drop_node_database* utilise *Primary* si elle supprime une NDB de type serveur. Elle modifie le champ *PimNd* des tuples de *Primary* qui pointent sur la NDB (à supprimer) par une nouvelle valeur affectant la nouvelle NDB qui remplace celle à supprimer.

↻ Concurrence entre un éclateur et une commande

Un éclateur peut utiliser la méta-table *Primary* simultanément avec l'une des commandes citées ci-dessus. Si l'éclateur commence en premier en générant un verrou partagé sur le tuple décrivant la table scalable qu'il utilise, alors la commande *sd_drop_table* par exemple, sera mise en attente. Une fois l'éclateur termine, la commande *sd_drop_table* peut générer son verrou exclusif sur le même tuple. Ainsi, s'il y a un nouveau segment résultant de l'éclatement, il sera pris en compte lors la suppression de sa table. Le niveau *Repeatable Read* gère donc la sérialisabilité de ces transactions concurrentes.

↻ Concurrence entre deux éclateurs

Si deux éclateurs de la même table scalable s'exécutent simultanément sur la méta-table *Primary* donc tous les deux effectuent une lecture simultanée du tuple qu'ils utilisent. En effet, ils génèrent chacun un verrou partagé sur le même tuple sans aucun conflit.

↻ Concurrence entre deux commandes

Les deux commandes utilisant la méta-table *Primary* peuvent s'exécuter simultanément sans interblocage. Si une commande, *sd_drop_node_database* par exemple, veut accéder au même tuple de *Primary* utilisé par une autre commande (*sd_drop_table* en particulier), elle attendra donc que cette dernière commande termine son traitement et récupère ensuite son verrou exclusif.

6.5.2.6 Accès à la Méta-table *NDB*

La méta-table NDB peut être utilisée simultanément par les transactions suivantes :

- Un éclateur accède à la méta-table NDB pour récupérer une NDB de type serveur disponible qui héberge le nouveau segment résultant de l'éclatement. Il génère alors un verrou partagé sur le(s) tuple(s) utilisé(s).

- Les commandes de suppression de NDB et de SDB utilisent la méta-table NDB. La première commande (*sd_drop_node_database*) récupère le type de la NDB à supprimer à partir de cette méta-table, donc elle utilise un verrou partagé au début. Ensuite, lors de la suppression du tuple décrivant la NDB à supprimer, elle génère un verrou exclusif. La commande *sd_drop_scalable_database* génère aussi un verrou exclusif sur les tuples décrivant les NDBs qui composent la SDB à supprimer.

Ainsi, nous pouvons avoir les exécutions simultanées suivantes :

↻ **Concurrence entre un éclateur et une commande**

Un éclateur peut être mis en attente pour accéder à la méta-table *NDB*. En effet, il peut trouver le tuple, qu'il veut utiliser, bloqué par un verrou exclusif d'une commande *sd_drop_node_database*. De même, cette dernière commande sera mise en attente si elle trouve un verrou partagé d'un éclateur.

↻ **Concurrence entre deux commandes**

Deux commandes *sd_drop_node_database* et *sd_drop_scalable_database* tout particulièrement, peuvent s'exécuter simultanément mais sans finir dans un interblocage. En effet, si une commande *sd_drop_scalable_database* trouve verrou partagé (ou exclusif) de la commande *sd_drop_node_database,* elle sera mise en attente. De même, cette dernière commande sera mise en attente si elle trouve le verrou exclusif de l'autre commande.

6.5.2.7 Accès à la méta-table *SDBNode*

La méta-table *SDBNode* est une ressource du système SD-SQL Server qui peut être partagée par les transactions de suppression de NDBs, *sd_drop_node_database*. En effet, cette commande peut supprimer le tuple décrivant la NDB à supprimer lors de la suppression de sa SDB. Elle peut aussi modifier un tuple décrivant une NDB primaire à supprimer. Dans le deuxième cas, il s'agit de modifier le champ *Node* qui hébergeait la NDB à supprimer. Dans les deux cas, la commande *sd_drop_node_database* génère un verrou exclusif sur le tuples utilisé. La commande *sd_drop_scalable_database* peut aussi utiliser la méta-table *SDBNode* lors de la suppression d'une SDB. Elle génère un verrou exclusif sur les tuples à supprimer. Ainsi, nous pouvons avoir l'exécution simultanée suivante :

↻ **Concurrence entre deux commandes**

Le conflit qu'il peut y avoir entre deux commandes s'exécutant simultanément sur *SDBNode* est le suivant : soit une commande *sd_drop_node_database* par exemple, qui

génère un verrou exclusif sur le tuple décrivant la NDB (à supprimer) afin de le modifier. Si une autre commande *sd_drop_node_database* veut utiliser le même tuple pour le supprimer, elle sera donc mise en attente pour générer son verrou exclusif sur ce tuple.

6.5.2.8 Gestion des Erreurs

Toute application qui utilise le système SD-SQL Server doit explicitement intercepter et gérer les erreurs. L'interception et la gestion des erreurs, à mesure qu'elles se produisent dans l'application, permettent d'effectuer une récupération d'erreurs élaborée et d'afficher des messages d'erreur pertinents. SQL Server permet une prise en charge complète des erreurs d'exécution générées. Nous nous basons sur la gestion des erreurs du système SQL Server pour gérer les erreurs sur SD-SQL Server.

Un grand nombre des erreurs générées peuvent être capturées et résolues. Les capacités de gestion des erreurs sont fournies par le langage Transact-SQL ainsi que les API (interfaces de programmation d'application) que les applications utilisent pour accéder aux données stockées dans une base de données. Les erreurs peuvent être gérées à deux niveaux [M00] :

- Les erreurs peuvent, d'une part, être gérées grâce à l'ajout de code de gestion d'erreurs aux lots Transact-SQL, aux procédures stockées, aux déclencheurs ou aux fonctions définies par l'utilisateur. Les mécanismes de gestion d'erreurs Transact-SQL contiennent la construction *TRY...CATCH*, l'instruction *RAISERROR* et la fonction *@@ERROR* :
    - Les erreurs dans le code Transact-SQL peuvent être traitées à l'aide d'une construction *TRY...CATCH* similaire aux fonctionnalités de gestion des exceptions des langages de programmation (C++, Java...). Une construction *TRY...CATCH* comprend deux parties : un bloc *TRY* et un bloc *CATCH*. Lorsqu'une condition d'erreur est détectée dans une instruction Transact-SQL contenue dans un bloc *TRY*, le contrôle est transmis à un bloc *CATCH* où elle peut être traitée. Cette construction a été introduite dans la version 2005 de SQL Server [T05].
    - *RAISERROR* permet de retourner des messages aux applications dans le même format qu'une erreur système ou qu'un message d'avertissement généré par SQL Server.
    - la fonction *@@ERROR* permet de détecter des erreurs dans les instructions Transact-SQL. Elle renvoie 0 si la dernière instruction Transact-SQL a été exécutée avec succès. Dans le cas contraire, *@@ERROR* renvoie le numéro de l'erreur. La valeur de *@@ERROR* change à la fin de l'exécution de chaque instruction Transact-SQL.

- Les erreurs peuvent, d'autre part, être retournées à l'application appelante pour être gérées dans le code de l'application. Chacune des API qu'utilisent les applications pour accéder à une base de données offre des mécanismes de transfert des informations relatives aux erreurs vers l'application.

Nous proposons de gérer les erreurs sur SD-SQL Server en utilisant l'instruction de SQL Server, *RAISERROR*. Au niveau de chaque procédure stockée, qui implémente une commande SD-SQL Server, nous introduisons l'instruction *RAISERROR*. Par contre l'implémentation de cette partie du prototype reste dans les perspectives puisque ça ne constitue pas l'objectif de notre travail. Nous comptons utiliser par la suite la construction *TRY…CATCH* puisqu'elle est plus améliorée dans la gestion d'erreurs. Mais nous ferons ça une fois que notre prototype sera exécuté sur la version 2005 de SQL Server.

6.6 Conclusion

Dans ce chapitre, nous avons présenté notre prototype SD-SQL Server. Nous avons d'abord présenté nos choix techniques pour son implantation tout en les justifiant. Ensuite, nous avons présenté les éventuelles étapes suivies dans l'utilisation du système SD-SQL Server. Nous avons aussi détaillé le traitement interne de chaque commande. Enfin, nous avons présenté et résolu les différents conflits entre les commandes concurrentes. Le chapitre suivant sera consacré à l'évaluation des performances de ces commandes.

7 MESURES DE PERFORMANCES

7.1 Introduction

Les performances en termes de débit (nombre de transactions exécutées par seconde) et de temps de réponse (temps d'attente moyen pour une requête type) sont un problème clé dans un SGBD. L'objectif de débit élevé nécessite un surcoût minimal dans la gestion des tâches accomplies par le système. L'objectif de bons temps de réponse implique qu'une requête courte d'un utilisateur n'attende pas une requête longue d'un autre utilisateur.

Nous montrons tout au long de ce chapitre que le système SD-SQL Server atteint les objectifs d'un SGBD classique. Nous décrivons les expérimentations que nous avons effectuées afin d'évaluer ses performances et étudier sa scalabilité. Nous comparons aussi ces performances avec celles du SGBD SQL Server afin de montrer l'efficacité de notre système par rapport à SQL Server.

7.2 Environnement Expérimental

Notre prototype est réalisé dans un environnement *Windows 2000 Server*. Ce prototype est déployé sur un réseau local *Ethernet* de 1 G bits/s reliant six machines *Pentium IV* ayant entre *785* MO à *1* Gb de RAM. La Table 7-1 suivante décrit les caractéristiques de chacune de ces machines.

nom de la machine	RAM (GB)	horloge (GHtz)
Ceria	1	1,7
Dell1	0,780	1,7
Dell7	0,780	1,7
Dell8	1	1,7
Dell10	0,780	1,7
Dell11	0,780	2,3

Table 7-1 : Configuration expérimentale

Sur chacune de ces machines, nous avons installé le SGBD *Microsoft SQL Server 2000*. Les noms de chaque instance SQL Server sont ceux de leurs machines. Nous avons mesuré le temps de nos différentes expériences à l'aide de l'outil *SQL Profiler* qui est intégré dans SQL Server. C'est un outil graphique installé avec le SGBD SQL Server. Il permet de tracer les performances des requêtes exécutées sur une instance SQL Server. Nous exécutons nos requêtes sur l'éditeur de requêtes (ou commandes) *SQL Analyzer*, installé aussi avec SQL Server [M00].

Notons que les résultats des expérimentations dépendent fortement de la puissance des machines utilisées (vitesse du processeur, mémoire disponible, etc.) et de la nature du réseau. Le temps de réponse d'une requête comprend trois composantes : (*1*) le temps d'envoi des paramètres de la requête, (*2*) le temps de traitement de la requête et (*3*) le temps d'envoi de la réponse. (*1*) et (*3*) dépendent de la vitesse du réseau et de son niveau d'encombrement, (*2*) dépend de la vitesse de traitement du serveur, variable en fonction de sa charge.

7.2.1 Description des Expérimentations

Afin de valider l'architecture du système SD-SQL Server, nous avons effectué plusieurs mesures qui prouvent sa scalabilité et son efficacité. Ces mesures ciblent la détermination d'un surcoût au niveau des nœuds SD-SQL Server :

- Au niveau des nœuds de type serveur, nous effectuons les mesures de temps sur l'éclatement des segments des tables scalables.
- Au niveau des clients SD-SQL Server, nous mesurons le surcoût du traitement des commandes SD-SQL Server sur les tables scalables.

Nous avons effectué nos expérimentations sur le fragment du benchmark *SkyServer* [G02]. Nous avons tout particulièrement concentré nos expériences sur la table *PhotoObj* étant donné que c'est la table la plus volumineuse du benchmark *SkyServer*. Cette table contient environ 160.000 tuples et 400 colonnes, ce qui lui fait une taille d'environ 260 MB.

La liste des expériences que nous avons effectuées et que nous détaillerons par la suite se résume en :

- Eclatement d'une table scalable *PhotoObj* en fonction de sa capacité et du nombre de segments qui résultent de l'éclatement.
- Exécution d'une commande SD-SQL Server. Nous prenons en compte, dans le temps de l'exécution, le temps effectué pour l'*image binding* et l'ajustement des images.

Pour chaque série d'expérimentation sur SD-SQL Server, nous effectuons l'expérimentation équivalente sur SQL Server. Ensuite, nous comparons les temps obtenus entre une exécution sur SD-SQL Server et sur SQL Server.

7.3 Eclatement

L'étude de l'éclatement d'une table scalable nous a amené à étudier deux cas de la migration des données :

- Eclatement d'une table scalable ayant un segment primaire en deux, trois, quatre puis cinq segments respectivement. Nous effectuons ces éclatements à plusieurs reprises selon différentes capacités de la table scalable qui éclate [LS04].
- Eclatement d'une table scalable contenant des index. Dans cette expérience, nous prenons en compte, dans le temps d'éclatement, le temps de transfert des index d'une table scalable vers ses nouveaux segments résultant [SLS05].

Afin d'effectuer ces traitements, nous utilisons la table *PhotoObj* du benchmark *SkyServer* comme une table scalable sur notre système SD-SQL Server. Nous lui avons attribuée une capacité *b.* Nous avons varié sa capacité pour chaque expérience. Les différentes capacités que nous avons utilisé pour *PhotoObj* sont *b=1000, 2000…160000 tuples*. *PhotoObj* contient initialement un seul segment (son segment primaire). Ensuite, nous effectuons des insertions dans cette table afin de la faire éclaté en deux, trois, quatre puis cinq segments pour chacune des capacités citées.

Dans chaque traitement, nous effectuons une insertion pour déclencher un éclatement. Pour l'éclatement en deux segments, nous exécutons la commande *sd_insert* pour insérer un tuple, comme suit [5]:

```
sd_insert 'PhotoObj (objid) values (9999999)'
```

Pour les autres éclatements (en trois, quatre et cinq segments), nous exécutons la commande *sd_insert* pour insérer un bloc de tuples. Nous récupérons ces tuples à partir d'une table statique *PhotoObj* d'une base statique *SkyServer* (localisé sur l'instance SQL Server local de l'exécution de la commande, *Ceria* par exemple) que nous avons importée à partir du benhmark *SkyServer* [G02]. Nous changeons le nombre de tuples insérés selon le nombre de segments que nous voulions avoir en résultat. Nous exécutons la commande *sd_insert* comme suit :

```
sd_insert 'PhotoObj SELECT TOP * FROM Ceria.SkyServer.dbo.PhotoObj'
```

[5] Nous avons mis affecté à tous les autres attributs,de *PhotoObj,* la valeur *NULL* avant l'exécution de cette commande.

7.3.1 Cas-1 : Table Scalable sans Index

Notre expérience sur l'éclatement étudie le temps des différents traitements effectués lors de l'éclatement. Nous avons réalisé quatre expériences d'éclatement. La première expérience déclenche un éclatement lors d'une insertion de tuples qui font éclater le segment primaire de la table scalable *PhotoObj* en deux segments. Nous avons refait cette expérience pour les différentes capacités *b=1000, 2000…160000.* A chaque nouvelle capacité *b*, nous recréons la table scalable *PhotoObj* avec un segment primaire. Les trois autres expériences font la même étude mais avec des éclatements en trois, quatre puis cinq segments respectivement selon les différentes capacités *b*.

La Table 7-2 donne les résultats numériques obtenus de ces expériences. La Figure 7-1 présente les courbes correspondantes. Le temps d'éclatement, calculé en secondes, est d'une manière prévisible meilleur pour l'éclatement en deux segments ou pour les segments qui ont des tailles plus petites. En outre, le temps reste relativement rapide (quelques minutes) tout en respectant la réorganisation globale pour chaque capacité de *PhotoObj*.

capacité d'un segment (en tuples)	**temps d'eclatement (sec) en :**			
	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 7-2 : Résultats numériques du temps d'éclatement de *PhotoObj* en fonction du nombre de segments qui résultent

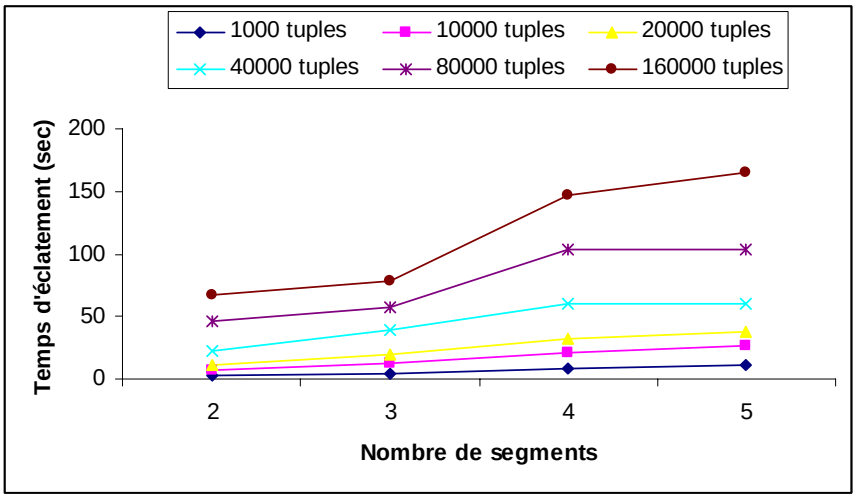

Figure 7-1 : Evolution du temps d'éclatement en fonction du nombre de segments résultant

7.3.2 Cas-2 : Table Scalable avec des Index

Dans cette section, nous évaluons le temps d'éclatement d'une table scalable contenant des index. Ainsi, le temps d'éclatement inclut aussi le temps du transfert des index vers les nouveaux segments qui résultent de l'éclatement. Nous reprenons les mêmes expériences de la section précédente mais sur une table *PhotoObj* de 160k tuples et contenant zéro (sans index), un, deux et trois index. La Figure 7-2 et la Table 7-3 montrent les résultats.

La conclusion de cette expérience est que le temps d'éclatement augmente naturellement avec l'augmentation du nombre d'index existants sur le segment qui éclate ainsi que le nombre de segments résultant de l'éclatement. Ceci crée un surcoût additionnel pour la gestion d'index. Ce surcoût reste néanmoins négligeable. Il est à moins de 10% pour un segment qui éclate en quatre nouveaux segments. L'augmentation du temps pour un éclatement qui donne cinq segments est un peu plus importante mais reste modérée à environ 22%.

nombre d'index à transférer	**temps d'Eclatement (sec) en :**			
	2 seg	**3 seg**	**4 seg**	**5 seg**
0 index	54,656	77,86	130,89	165,11
1 index	52,33	77	133,86	187,813
2 index	54,816	82,596	137,626	196,783
3 index	56,486	83,61	139	202,65

Table 7-3 : Résultats numériques des temps d'éclatement de *PhotoObj* avec index

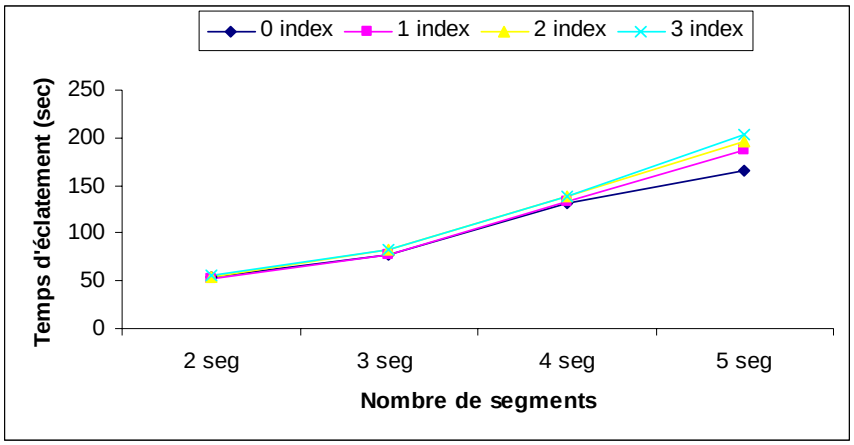

Figure 7-2 Temps d'éclatement des segments avec index

7.3.3 Comparaison entre un Eclatement sur SD-SQL Server et un Eclatement sur SQL Server

Dans cette section, nous déterminons le surcoût de l'éclatement tout en respectant les mêmes traitements effectués pour l'éclatement d'une table scalable. Nous avons comparé le temps d'éclatement sur SD-SQL Server avec le temps d'éclatement effectué directement sur SQL Server. L'éclatement sur SQL Server n'est pas dynamique, il consiste à transférer manuellement les tuples qui surchargent une table. Le temps d'éclatement sur SQL Server inclut alors :

- le temps du transfert de la moitié de la capacité d'un segment vers un nouveau segment ;
- le temps de suppression des tuples transférés du segment qui éclate ;
- et le temps des autres opérations comme la mise à jour des méta-tables...

Nous avons exécuté des requêtes SQL (*SELECT INTO, DELETE*…) qui permettent d'effectuer les traitements ci-dessus. Ensuite, nous avons mesuré le temps d'exécution de ces requêtes. Le temps du transfert des tuples vers un nouveau segment et la suppression de ces tuples du segment éclatant est d'environs 40 secondes sur SQL Server pour un segment de capacité 80.000 tuples. Le temps total d'éclatement de ce segment en deux segments sur SD-SQL Server est d'environ 46 secondes. Ainsi, la différence entre le temps d'éclatement sur SQL Server et celui sur SD-SQL Server est 6 secondes seulement. Autrement dit, la différence est d'environ 15% du temps qui inclut les autres requêtes auxiliaires déjà citées.

7.4 Exécution des Commandes SD-SQL Server

Afin d'étudier le surcoût dans l'exécution des commandes SD-SQL Server, nous avons exécuté des commandes selon différents critères. Nous avons pris en compte :

- Les commandes qui retournent des résultats coûteux dans l'évaluation des requêtes. Ceci correspond aussi aux clauses SQL complexes.
- Les commandes qui permettent un traitement distribué développé.
- Les commandes qui interrogent des images à plusieurs niveaux, ce que nous avons appelé les vues scalables.

Nous avons utilisé, en particulier, des commandes de recherche (*sd_select*) tout en les adaptons pour chaque cas cité ci-dessus. Le temps d'exécution d'une commande *sd_select* inclut le temps des traitements suivant :

- Le temps de la phase *image binding*. Nous rappelons que dans cette phase, SD-SQL Server analyse la requête scalable *sd_select*. Cette phase analyse principalement la clause *FROM* et récupère les images (ou les vues scalables) utilisées si elles existent. Ainsi le temps de l'*image binding* inclut le temps du traitement sur la chaîne de caractère qui représente la clause *FROM*, le temps d'accès à la méta-table *Image* (pour vérifier si un objet correspond à une image) et le temps d'accès aux tables systèmes *sysobjects* et *sysdepends* dans le cas de vues scalable (cf. Section 6.4.6.1).
- Le temps de l'ajustement des images. S'il résulte des images de la phase *image binding*, SD-SQL Server vérifie alors si ces images sont correctes. Pour cela, il accède à la méta-table *Image* pour récupérer le nombre de segments définis dans une image. Il effectue aussi un accès à la méta-table *RP* pour récupérer le nombre réel de segments dans la table scalable représentée par cette image. Ensuite, il compare les deux nombres obtenus et, si nécessaire, il modifie le nombre trouvé dans la table *Image*. Ainsi le temps de vérification de l'ajustement d'une image correspond particulièrement au temps d'un accès à la table *Image*. Quant au temps de l'ajustement, il inclut en plus le temps d'un accès à *RP* ainsi que le temps des autres traitements (comparaison des nombres, modification de la définition de l'image, etc.).
- Le temps de l'exécution de la requête scalable. Ce temps correspond au temps habituel de l'exécution d'une requête SQL correspondant à la requête scalable *sd_select*.

Ainsi, le temps d'exécution de toute commande *sd_select* inclut les temps des traitements cités ci-dessus. Nous prenons en compte tous les cas possibles dans l'exécution d'une commande *sd_select*, dans les expérimentations que nous présentons dans ce qui suit.

7.4.1 Requête Coûteuse

Nous avons exécuté la commande suivante, obtenue à partir du benchmark *SkyServer* et transformée pour représenter une commande SD-SQL Server :

(Q1) sd_select '* FROM PhotoObj
WHERE (status &0x00002000 > 0) AND (status &0x0010 > 0)'

L'image *PhotoObj* interrogée dans cette commande représente la table scalable *PhotoObj* partitionnée en deux segments et de taille 158.426 tuples. Nous supposons que l'image *PhotoObj* n'est pas ajustée.

La requête (Q1) montre le côté le plus coûteux dans l'évaluation des requêtes. En effet, elle donne en résultat plusieurs milliers de tuples (129.470 tuples). De plus, ces tuples sont sélectionnés à partir de segments (de la table scalable *PhotoObj)* se trouvant sur des NDBs distribuées de la SDB *SkyServer*. L'exécution de la requête (Q1) sur SD-SQL Server prend 45 secondes en prenant en compte le temps de l'*image binding* et de l'ajustement de l'image *PhotoObj*.

Pour évaluer le temps d'exécution de (Q1) sur SQL Server, nous avons exécuté la même requête SQL correspondante, il s'agit de :

SELECT * FROM PhotoObj
WHERE (status &0x00002000 > 0) AND (status &0x0010 > 0)

PhotoObj représente dans cette requête, une vue partitionnée avec les deux segments de *PhotoObj*. Le temps d'exécution de cette requête prend 44 secondes sur SQL Server. Nous rappelons que l'exécution directe sur SQL Server n'inclut pas les traitements liés à l'*image binding* et l'ajustement de *PhotoObj*. Ainsi, l'ajustement de l'image prend seulement une seconde dans l'exécution de (Q1) sur SD-SQL Server. Il présente uniquement 2 % du temps de l'exécution de la requête. Nous pouvons conclure que L'analyse théorique indique que le surcoût de l'*image binding* et l'ajustement est négligeable dans le cas de requêtes coûteuses.

Afin d'évaluer le surcoût dans l'ajustement des images dans le cas de requêtes plus complexes, nous avons exécuté la requête suivante :

(Q1-a) sd_select "TOP 10000 x.objid FROM PhotoObj x, PhotoObj y
WHERE x.obj=y.obj AND x.objid>y.objid

La requête (Q1-a) représente une requête complexe avec des jointures. Elle est exécutée sur l'image *PhotoObj* ayant 158.426 tuples. Nous avons partitionné sa table scalable *PhotoObj* en deux, trois puis quatre segments avec la même taille de la table (environ 160k tuple) pour chaque partitionnement. Le temps d'exécution de (Q1-a) avec l'*image binding* uniquement (sans l'ajustement de l'image *PhotoObj*), est entre 10 et 12 secondes.

Le surcoût est d'environ une seconde. Il augmente un peu tant que la requête SQL *ALTER VIEW* (pour modifier la définition de l'image) traite plus de segments distribués.

Le surcoût de l'ajustement d'image devient relativement négligeable. Il est d'environ 10% du coût de la requête. Nous rappelons que l'ajustement est généralement une opération très rare.

Exécution de (Q1-a)	***PhotoObj* partitionnée en :**		
	2 seg	**3 seg**	**4 seg**
avec ajustement de *PhotoObj*	10.898	13.036	14.071
sans ajustement de *PhotoObj*	9.862	11.578	11.712

Table 7-4 : Mesures Numériques de l'exécution de (Q1-a)

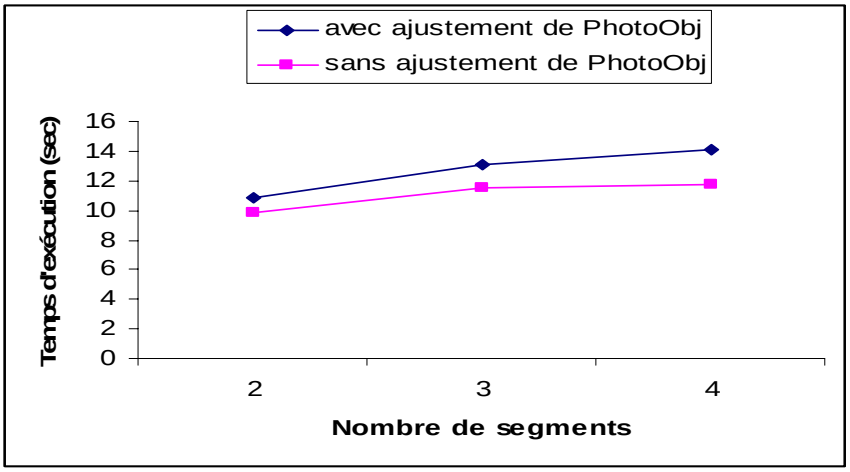

Figure 7-3 : Résultats graphique des temps d'exécution de (Q1-a)

7.4.2 Requête Rapide

Soit la requête suivante :

(Q2) sd_select 'TOP 10 objid FROM PhotoObj WHERE objid not in
(SELECT objid FROM PhotoObj WHERE objid <= @objidMax)'

Nous avons exécuté (Q2) sur la NDB client *Ceria.SkyServer* où l'image primaire est localisée. La NDB *Dell11.SkyServer* détient le segment primaire de la table *PhotoObj*. Cette dernière a deux segments. Nous varions sa capacité *b* pour chaque expérience, *b=39500, 79000, 158000 tuples,* comme le montre la Figure 7-4.

La requête (Q2) montre le côté rapide dans l'évaluation des requêtes. Elle évalue le traitement distribué avec un temps de réponse qui fait monter en échelle la table *PhotoObj*

comme nous le montrerons par la suite. Cependant, elle donne en résultat dix tuples seulement, ce qui n'est pas très coûteux comme résultat. Ces tuples sont sélectionnés à partir des segments se trouvant sur des NDBs distribuées de la SDB *SkyServer*. Le surcoût de l'*image binding* devrait affecter plus de telles requêtes puisqu'il y a deux clauses *FROM* à analyser. L'analyse expérimentale semble le moyen le plus facile pour trouver ce coût, elle montre, tout particulièrement, si le surcoût de l'*image binding* s'avère négligeable.

Le temps d'exécution de (Q2) dépend de la taille de *PhotoObj* et ceci à cause de sa sous-requête :

SELECT objid FROM PhotoObj WHERE objid <= @objidMax'

Le paramètre *@objidMax,* qui apparaît dans la sous-requête de la commande (Q2), représente la clé maximale du premier segment. Nous l'avons choisi ainsi afin que la sous-requête ci-dessus interroge uniquement le premier segment de la table *PhotoObj*. SQL Server évalue (Q2) très probablement en utilisant les index (automatique) sur la clé *objid* de *PhoroObj*.

Les mesures apportées sur la requête (Q2) montrent le temps de réponse en prenant en compte : (1) la vérification de l'image *PhotoObj* (l'image binding) et (2) l'ajustement de l'image. Nous avons ensuite comparé le temps de réponse de (Q2) présenté dans la Figure 7-4 à celui de la requête (Q2) en prenant en compte (3) l'exécution directe sur SQL Server. Cette exécution est générée par la requête SQL *SELECT* qui correspond à la commande *sd_select*.

La différence entre le temps d'exécution des traitements (1) et (3) apparaît négligeable. Dans les deux cas, le temps d'exécution est d'environ 300 ms. Ceci explique que le temps de l'*image binding* est presque négligeable. Nous constatons que le surcoût est constant puisqu'il correspond aux mêmes opérations qui sont indépendantes de la sémantique de la requête. Les courbes montrent ainsi que le surcoût des traitements de requêtes par SD-SQL Server est négligeable.

Le surcoût de l'ajustement de l'image *PhotoObj* dans le cas (2) domine le temps de réponse de la requête (Q2) qui devient constant avec l'augmentation de la capacité de *PhotoObj*. Le temps total est d'environ 700 ms. Le temps de réponse de la requête devient considérablement plus long. Il reste cependant encore largement négligeable en pratique sachant que l'ajustement de l'image est une opération rare. Le temps du surcoût dans l'ajustement de l'image est d'environ 500 ms. Il est presque constant pour les différentes capacités de la table scalable *PhotoObj*. Nous rappelons que ce temps est dû au traitement distribué sur la méta-table *RP* afin de récupérer les nœuds qui détiennent les segments de *PhotoObj*. De plus, il y a le traitement qui redéfinit l'image *PhotoObj* en modifiant l'ancienne vue partitionnée distribuée qui la représente.

capacité de *PhotoObj* (en tuples)	Exécution de (Q2)		
	avec ajustement de *PhotoObj*	sans ajustement de *PhotoObj*	sur SQL Server
39500	0.7796	0.11	0.096
79000	0.812	0.148	0.176
158000	0.8515	0.32	0.281

Table 7-5 : Résultats Numériques du temps d'exécution de (Q2)

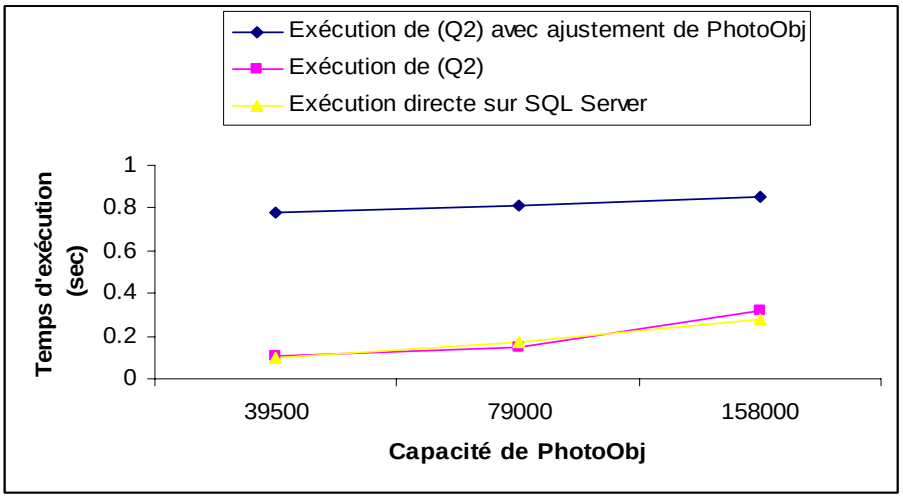

Figure 7-4 : Temps d'exécution de (Q2)

Après cette expérience, nous comparons, dans ce qui suit, les temps d'exécution de (Q2) sur une NDB de type client et ensuite sur une NDB de type pair. Nous rappelons que si une NDB est de type client, elle ne contient donc que la méta-table *Image* et les images des tables scalables notamment l'image *PhotoObj*. Ainsi, l'accès à la méta-table *RP,* pour l'*image binding* et l'ajustement d'image, nécessite d'accéder deux fois à un serveur lié (la NDB serveur qui héberge la méta-table *RP*). Cependant, si la NDB de l'exécution d'une commande est de type pair, donc les méta-tables *Image* et *RP* se trouvent sur la même NDB. Ainsi, l'accès à *RP* est fait localement sur la même NDB sur laquelle s'exécute la commande. Nous avons pris le cas où la NDB pair détient le segment primaire de la table scalable *PhotoObj*, ainsi la méta-table *RP* est sur la même NDB pair. *PhotoObj* est partitionnée en deux segments. Nous mesurons l'exécution de (Q2) sur *PhotoObj* avec les capacités respectives *b= 39500, 79000* et *158000 tuples*.

La Figure 7-5 rajoute à la figure précédente trois courbes :

- La courbe *Pair avec ajustement* représente le temps d'exécution de la requête (Q2) sur une NDB de type pair. Le temps d'exécution est clairement inférieur à celui sur une NDB de type client uniquement. Il est d'environ 0.8 secondes sur une NDB pair et 1.5 secondes sur une NDB client. Cette différence est visiblement due à l'accès à la méta-table *RP* qui est soit local (dans le cas d'une NDB pair) ou distant (dans le cas d'une NDB client).
- La courbe *Pair sans ajustement* représente le temps d'exécution de (Q2) en prenant en compte uniquement la vérification de l'ajustement de l'image sans l'ajuster. Le temps d'exécution est proche de celui de (Q2) exécutée sur une NDB client. En effet, la différence est uniquement dans le temps d'accès à *RP* pour vérifier le nombre de segments dans l'image *PhotoObj*.
- La courbe *Exécution directe sur un SQL Server pair* représente le temps d'exécution de la requête SQL *SELECT* (correspondant à (Q2)) sur SQL Server. Nous voulons dire par *SQL Server pair*, le fait que cette instance contient l'image interrogée *PhotoObj* ainsi que le segment primaire de sa table scalable.

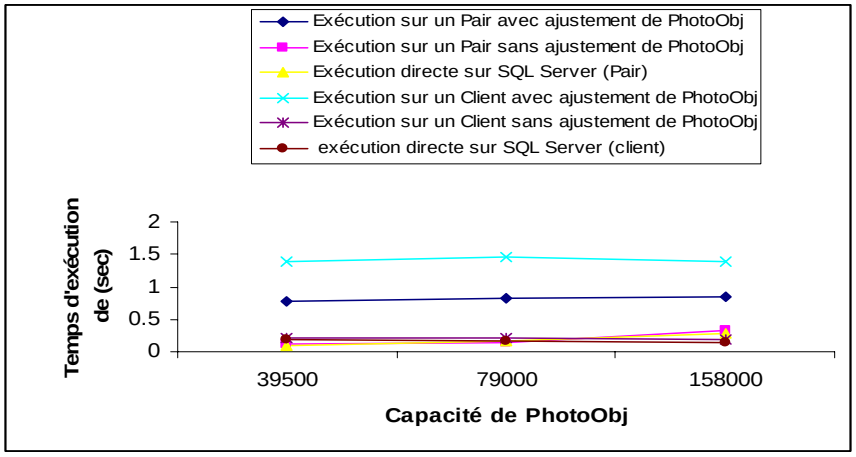

Figure 7-5 : Comparaison des temps d'exécution de (Q2) sur une NDB client et une NDB pair

7.4.3 Requête adressant des images à Plusieurs Niveaux

Nous avons étudié aussi le temps de l'*image binding* pour des requêtes qui interrogent des images à différents niveaux, ce que nous avons appelé des *vues scalables*. Nous nous sommes intéressés en particulier à déterminer le surcoût d'une requête qui adresse une vue scalable qui fait appel à une image, ensuite une vue d'une vue d'une image, etc. Nous rappelons que ces vues sont appelées vues de niveau *i = 1, 2*... L'image binding, dans

cette expérience, boucle d'une façon récursive sur les méta-tables SD-SQL Server quand *i* s'incrémente. Pour cela, des jointures coûteuses sont utilisées comme nous l'avons déjà mentionné précédemment. Pour effectuer cette expérience, nous avons alors créé les vues scalables suivantes *T1*, *T2* et *T3*:

CREATE VIEW T1 AS SELECT * FROM PhotoObj

CREATE VIEW T2 AS SELECT * FROM T1

CREATE VIEW T3 AS SELECT * FROM T2

Ensuite, nous avons exécuté les requêtes suivantes qui adressent ces vues scalables :

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

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

(Q5) 'sd_select COUNT (*) FROM T2'

(Q6) 'sd_select COUNT (*) FROM T3'

Un segment de la table scalable *PhotoObj*, dans cette expérience, contient 39.500 tuples. La Figure 7-6 montre le résultat d'exécution des requêtes (Qi) *(i=3,4,5,6)* avec et sans ajustement de l'image *PhotoObj* qu'elles interrogent directement (elle est donc de niveau 0) ou indirectement (à plusieurs niveaux). Les courbes dans la figure montent légèrement, mais restent pratiquement plates. L'incidence du niveau de l'image sur le temps de l'*image binding* est donc négligeable.

	***PhotoObj* est de :**			
exécution de (Qi) :	**Niveau 0**	**Niveau 1**	**Niveau 2**	**Niveau 3**
avec ajustement de *PhotoObj* (temps en sec)	0.811	0.826	0.846	0.853
sans ajustement de *PhotoObj*	0.11	0.126	0.14	0.153

Table 7-6 : Résultats Numériques du temps d'exécution de (Qi) (i=3,4,5,6)

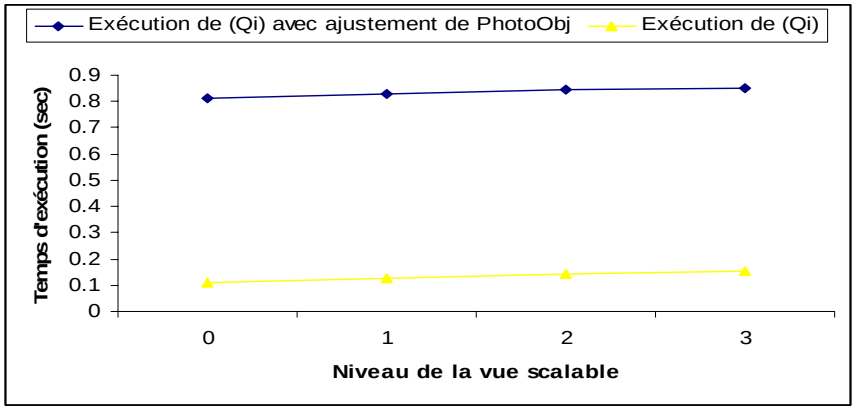

Figure 7-6 : Temps d'exécution de (Qi) pour une image à plusieurs niveaux

7.4.4 Comparaison entre SD-SQL Server et SQL Server

Dans cette section, nous visons la comparaison du temps de réponse de l'exécution d'une requête sur SD-SQL Server et ensuite sur SQL Server. Nous avons utilisé la requête (Q3), de la section précédente. Pour l'exécution de (Q3) sur SD-SQL Server, nous procédons comme dans les sections précédentes, c'est-à-dire il suffit d'exécuter *sd_select* comme elle se présente. Quant à l'exécution sur SQL Server, nous utilisons la requête SQL suivante qui correspond à (Q3) :

```
SELECT COUNT(*) FROM PhotoObj
```

Pour aboutir à une bonne comparaison entre l'exécution sur SD-SQL et sur SQL Server, nous évitons l'ajustement de l'image *PhotoObj* lors de l'exécution de (Q3) sur SD-SQL Server. En effet, l'exécution sur SQL Server retourne le résultat de la requête *SELECT* uniquement. Par contre, sur SD-SQL Server, en plus de l'exécution, il y a aussi les traitements liés à l'*image binding* et l'ajustement de l'image (si nécessaire). Pour éviter l'ajustement de l'image qui peut alourdir le temps d'exécution de (Q3), nous définissons la vue partitionnée distribuée de l'image *PhotoObj* avec le même nombre (réel) des segments de sa table scalable.

Les sections suivantes montrent les mesures de performances qui comparent SD-SQL Server à SQL Server selon la variation du nombre de segments, ensuite selon la taille d'un segment dans une table scalable

7.4.4.1 Variation du Nombre de Segments

Nous avons expérimenté sur la table scalable *PhotoObj* partitionnée en deux, trois, quatre puis cinq segments respectivement. La requête (Q3) effectue le compte de tuples (*COUNT)* sur l'image *PhotoObj* contenant un, deux, trois, quatre puis cinq segments.

Chaque segment a une capacité de 30k tuples. La table 7-7 montre les résultats numériques de cette expérience. La Figure 7-7 montre la représentation graphique de cette expérience :

- La courbe nommée « SQL Server Centr. » montre le cas d'une table *PhotoObj* centralisée, c'est-à-dire il s'agit d'une seule table statique *PhotoObj* localisée sur une base locale.
- La courbe nommée «SQL Server Distr. » reflète la réorganisation manuelle de *PhotoObj* sur SQL Server. Dans ce cas, nous modifions manuellement la vue partitionnée de la table *PhotoObj* en lui rajoutant à chaque expérience une table membre (segment distribué) dans sa définition.
- La courbe nommée « SD-SQL Server » montre l'exécution de (Q3) en laissant l'option *Lazy Scema Validation*[6] désactivée.
- La courbe nommée « SD-SQL Server avec LSV» montre l'exécution de (Q3) en activant l'option *Lazy Scema Validation* (LSV).

Nous pouvons voir que le traitement sur SD-SQL Server est toujours proche du traitement sur SQL Server. Le surcoût du traitement de la requête (Q3) est d'environ 5%. Nous pouvons voir aussi qu'avec les mêmes conditions d'utilisation, SD-SQL Server accélère l'exécution de la requête (Q3) d'environ 30%. Pour la table la plus volumineuse, le temps est d'environ 100 msec. Cette accélération est réalisée en désactivant l'option *lazy schema validation*. Si cette option est activée, le temps baisse à 220 msec. Il est ainsi amélioré d'environ 50%. Ce facteur caractérise toutes les autres capacités utilisées. Tous ces résultats prouvent l'utilité immédiate de notre système.

Notons qu'en théorie, le temps d'exécution sur SD-SQL Server reste constant et proche de celui d'une requête sur un seul segment de 30 k tuples. Il est de 93 msec dans notre cas. Le résultat semble indiquer que le traitement parallèle d'une fonction d'agrégation par SQL Server est encore à désirer. Ceci augmentera la supériorité de SD-SQL Server pour le même confort d'utilisateur.

[6] L'option *Lazy Scema Validation* est définie pour un serveur SQL Server. Elle détermine si le schéma des tables distantes doit être vérifié. Si cette option a la valeur *true*, la vérification du schéma des tables distantes est ignorée au début de la requête et ainsi le temps d'exécution de la requête est optimisé.

Exécution sur	Exécution de (Q3) *PhotoObj* contenant (temps en sec)				
	1 seg	2 seg	3 seg	4 seg	5 seg
SQL Server Distr	0.093	0.156	0.220	0.250	0.326
SD-SQL Server	0.106	0.164	0.226	0.256	0.343
SD-SQL Server LSV	0.016	0.076	0.123	0.203	0.220
SQL Server Centr	0.093	0.203	0.283	0.356	0.436

Table 7-7 : Résultats numériques de l'exécution de la requête (Q3)

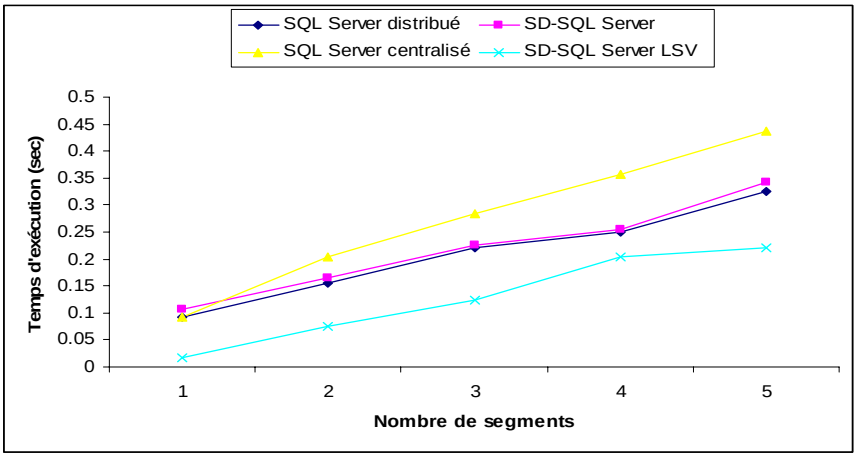

Figure 7-7 : Comparaison des temps d'exécution de (Q3) sur SQL Server et SD-SQL Server

7.4.4.2 Variation de la Taille d'un Segment

Afin de mieux confirmer nos résultats dans la comparaison entre SD-SQL Server et SQL Server, nous avons réalisé l'expérience suivante. Nous avons exécuté la requête (Q3) sur une image *PhotoObj* contenant un seul segment. Nous avons ensuite réalisé une série d'expérimentations en variant la taille *b* du segment de la table scalable *PhotoObj*. Les différentes tailles que nous avons utilisé sont *b*= 1000, 4000, 8000, 16000, 32000, 64000, 96000, 128000 et 164000 tuples.

La Table 6-1 et la Figure 7-8 montrent respectivement les résultats numériques et la représentation graphique de cette expérience. Notons que le temps d'exécution sur SD-SQL Server et presque le même que celui sur SQL Server. Ceci montre aussi un surcoût négligeable sur SD-SQL Server.

A partir de cette expérience, nous pouvons conclure que si nous avons un segment de 1000 tuples et 164 serveurs, SD-SQL Server pourra atteindre l'accélération de 453/20

pour la table *PhotoObj* avec une taille de 164k tuples. Le premier temps dans le quotient (453ms) représente le temps du traitement centralisé effectué sur SQL Server comme le montre nos expériences. Le deuxième temps dans le quotient (20ms) représente le temps d'exécution de (Q3) sur *PhotoObj* à 1000 tuples, qui est 16 ms, en lui rajoutant le temps d'échange des messages [G89]. Nous pouvons assumer que le temps total est celui d'une exécution parallèle et idéale de la requête (Q3).

Capacité d'un Segment *PhotoObj* (tuples)									
temps d'exécution sur :	**1000**	**4000**	**8000**	**16000**	**32000**	**64000**	**96000**	**128000**	**164000**
SQL Server	0.013	0.016	0.033	0.046	0.080	0.173	0.266	0.343	0.436
SD-SQL Server	0.016	0.023	0.038	0.063	0.093	0.180	0.280	0.376	0.453

Table 7-8 : Résultats Numériques de l'exécution de la requête (Q3)

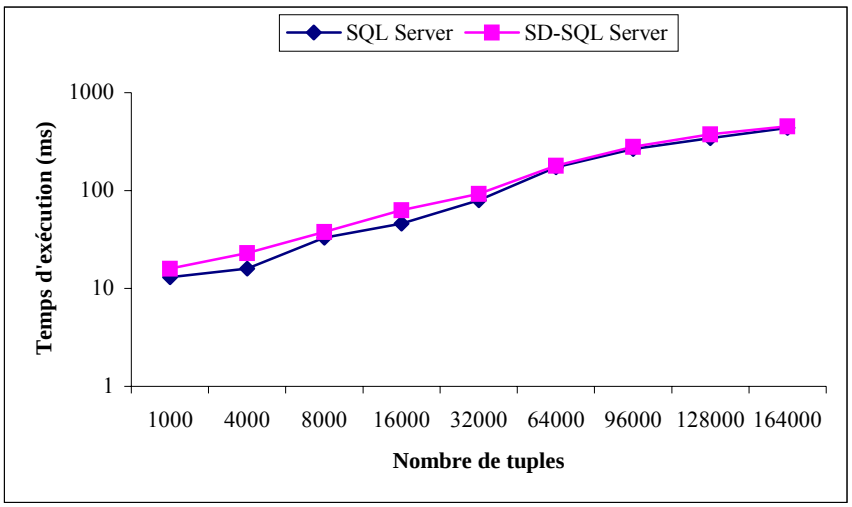

Figure 7-8 : Comparaison des temps d'exécution de (Q3) sur SQL Server et SD-SQL Server

7.5 Conclusion

Tout au long du présent chapitre, nous avons présenté et discuté les mesures de performances que nous avons mené pour valider l'architecture du système SD-SQL Server que nous avons proposé ainsi que les choix techniques que nous avons effectué

pour son implantation. Les résultats obtenus sont prometteurs dans le domaine, et prouvent l'efficacité du système SD-SQL Server. Néanmoins, dans le but d'avoir de meilleures performances, des améliorations peuvent être apportées, et c'est ce que le chapitre suivant évoque.

8 CONCLUSION & PERSPECTIVES

Ce chapitre résume les principaux apports de nos travaux, et s'achève sur diverses perspectives ouvertes.

8.1 Conclusion

Notre objectif a été la conception d'un système de gestion de bases de données distribuées et scalables. Le besoin de scalabilité a été ressenti ces dernières années à la suite de l'augmentation considérable de volumes des données stockées et l'apparition de l'utilité de leur réorganisation scalable et distribuée. Notre système est le premier à notre connaissance à mettre en pratique cette possibilité, sur la base des collections des SQL Servers liés.

Dans un premier lieu, nous avons présenté une étude des systèmes de gestion de bases de données parallèles en précisant leurs architectures et certains aspects de leurs qualités de services. Nous avons décrit les méthodes de partitionnement de données utilisées. Ces méthodes étant statiques, elles demandent une intervention manuelle dès que les données montent en échelle. Nous avons discuté les inconvénients de cet état de l'art, notamment pour SQL Server et Oracle.

Pour permettre un partitionnement dynamique, scalable et distribué, notre prototype SD-SQL Server applique les principes des SDDSs et l'architecture de référence dans [LRS02]. Notre architecture a introduit alors plusieurs nouvelles notions : celles de bases de données scalables, de vues scalables, etc. Nous avons implémenté toutes ces nouvelles notions.

Dans ce cadre, nous avons défini et réalisé les commandes de l'interface d'application leur correspondantes. Puis, nous avons conçu et implémenté les fonctions internes de traitement efficace de ces commandes. Notre structure interne de traitement repose sur les services du SQL Server. Nous avons tout particulièrement étudié la minimisation de surcoût de l'exécution de nos traitements, notamment des exécutions concurrentes. C'est

le souci d'optimisation des exécutions concurrentes qui nous a amené rendre les éclatements asynchrones et à utiliser le niveau d'isolation *Repeatable Read.*

Pour valider notre prototype, nous avons réalisé des mesures de performance, tout particulièrement sur les données de la base *SkyServer*. Nous avons montré que les temps d'exécution de requêtes sont peu affectés par nos traitements supplémentaires, tels que le test de l'image, ou le lancement d'un éclatement à la suite d'une insertion. Ceci, par rapport au temps d'exécution de requêtes similaires sous SQL Server. Nos mesures ont montré que le surcoût de notre prototype doit être en pratique négligeable. Elles confirment nos prévisions théoriques.

Notre travail a fait l'objet de quatre publications internationales. Nous avons été aussi invité à présenter notre prototype aux *Journées Académiques Microsoft Research* 2006. Les diapos de cette présentation sont affichées sur le site de Microsoft. Enfin, nos travaux ont été financés notamment par *Microsoft Research* et la *CEE*. L'ensemble semble confirmer l'intérêt dans les contributions de notre tèse à la théorie et la pratique de bases de données.

8.2 Perspectives

Nos travaux ont été du type de celles dites « la preuve de concept ». Elles sont limitées aux buts essentiels dans ce sens. Beaucoup d'extensions et de nouvelles applications sont possibles.

La première amélioration pourrait concerner notre gestion des erreurs de SD-SQL Server. Il serait intéressant d'avoir des instructions de gestion d'erreurs dédiées. On pourrait baser celle-ci sur les instructions habituelles (*RAISERROR*, etc*)* de SQL Server. Ou mieux, comme on a dit sur les nouvelles possibilités dites *Try…Catch* de SQL Server 2005.

On pourrait aussi étendre la tolérance aux pannes. Nous avons signalé les limitations correspondantes de nos méta-tables. Il faudrait peut-être répliquer certaines tables primaires. On pourrait aussi tenter les techniques de données de parité développée pour la SDDS de haute disponibilité *LH* *$_{RS}$ à [LS00, S02, LMS05].

Nos expériences ont été réalisées sur un réseau 1 Gbit/s avec quelques nœuds SD-SQL Server seulement. Nous souhaiterions montrer la validité de notre prototype par une expérimentation à plus grande échelle. Si possible de tester comme ça semble possible la configuration de 250 nœuds ou plus. Ces résultats pourraient intéresser notamment les développeurs de systèmes en grille ou P2P. Nous rappelons en effet que nos nœuds peuvent être tout particulièrement de type pair.

Ensuite nos principes pourraient être portés vers d'autres SD-DBS, basés sur Oracle, DB2, MySQL, etc. On pourrait envisager ensuite une configuration hétérogène. Ceci

donnerait encore plus de possibilités à l'utilisation de nos techniques sur les systèmes en grille ou P2P mentionés.

Enfin, nous pensons que notre travail ouvre des possibilités de nombreuses applications nouvelles. Nous pensons tout particulièrement appliquer notre système au sein d'un *repositoire virtuel*, destiné à la gestion de futurs *e-documents* de la vie courante des citoyens de la CEE. Cette application fait l'objet du projet *eGov* de la CEE dans lequel nous participons [LMS06].

BIBLIOGRAPHIE

[A01] Aberer, K. *P-Grid: A Self-organizing Access Structure for P2P Information Systems*. COOPIS, Trento, Italy, 2001.

[AVFG+92] Apers, P. M. G., Van Den Berg, C. A., Flokstra, J. P., Grefen, W. P. J., Kersten, M. L. Wilschut, A. N. "*PRISMA/DB: A Parallel Main Memory Relational DBMS*". IEEE Transaction on Knowledge and Data Engineering, 4(6), 1992.

[ACP+99] Atzeni, P., Ceri, Stefano, Paraboschi, S. & Torlone, R. *Database Systems: Concepts, Language and Architectures*. McGraw-Hill, 1999.

[AD01] Aberer, K., Despotovic, Z. *Managing Trust in a Peer-2-Peer Information System*. To appear in *the Proceedings of the Ninth International Conference on Information and Knowledge Management* (CIKM 2001) 2001.

[ASG+02] Alexander, S., Szalay, Gray, J., Ani, R., Thakar, Peter, Z., Kunszt, Tanu, M., Raddick, J., Stoughton, C. & VandenBerg, J. *The SDSS SkyServer – Public Access to the Sloan Digital Sky Server Data*, Technical Report, MSR-TR-2001-104, February 2002.

[BACC+90] Boral, H., Alexander, W., Clay, L., Copeland, G., Danforth, S., Franklin, M., Hart, B., Smith, M. & Valduriez, P. *Prototyping bubba, a highly parallel database system*. IEEE Knowledge and Data Engineering, March 1990.

[BM00] Ben-Gan, I. & Moreau, T. *Advanced Transact SQL for SQL Server 2000*. Apress Editors, 2000.

[BCV91] Bergsten, B., Couprie, M. & Valduriez P. *Prototyping dbs3 A Shared Memory Parallel Database System*. In Fist International Conference on Parallel and Distributed Information Systems. Décembre 1991.

[B95] C. Baru, & al. *DB2 Parallel Edition*. IBM Syst. Journal, 34(2), 1995, 292-322.

[BM72] Bayer, R. & McCreight, E. *Organization and Maintenance of large ordered indexs*. Acta Informatica, 1:173-189, 1972.

[CACM97] Comm. of ACM. *Special Issue on high-performance Computing.* October, 1997.

[C79] Comer, D. *The Ubiquitous B-tree.* Computing Surveys, 11(2):121-137, 1979.

[D01] Diène, A.W. *Contribution à la Gestion de Structures de Données Distribuées et Scalables,* Thèse de doctorat, Nov. 2001, Université Paris Dauphine. http://ceria.dauphine.fr/aly/aly.html.

[D01-p] Diène, A.W. *Prototype de la SDDS RP**, CERIA Lab., Université Paris Dauphine. http://ceria.dauphine.fr/aly/aly.html.

[D00] Demel, S. *Oracle 9i Parllel Server – Cache Fusion Delivers Scalability*, An Oracle White Paper. October 2000.

[DL01] Diène, A.W, Litwin, W. *Performance Measurements of RP*: Scalable and Distributed Data Structure for Range Partitioning*, Intl. Conf. on Information Society in the 21st Century: Emerging Tech. And New Challenges, Japan 2000.

[DL00] Diène, A.W, Litwin, W. *Implementation and Performance Measurements of the RP* Scalable and Distributed Data Structure for Windows Multicomputers*, Intl. Workshop on Performance-Oriented Program Devpt for Distributed Architectures, PADDA 2001.

[D92] Davis, D. *Oracle's Parallel Punch for OLTP*. Datamation, 1992.

[Da92] Davison, W. *Parallel Index Building in Informix OnLine 6.0.* ACM-SIGMOD International Conference, 1992.

[DG92] DeWitt, D. & Gray, J. *Parallel Database Systems: The Future of High Performance Database Systems*, Communications of The ACM, June 1992, Vol.35 No.6, pp.85-97.

[DG86] DeWitt, D. & Gerber, R.H. *Gamma, a high performance dataflow machine*. In 12th International Conference on Very Large Databases, Kyoto, August 1986.

[DGG+86] DeWitt, D., Gerber, R.H., Graefe, G., Heytens, M. L., Kumar, K. B., Muralikrishna, M. *GAMMA - A High Performance Dataflow Database Machine*. International Conference on Very Large Data Bases, 1986.

[G02] Gray, J. & al. *Data Mining of SDDS SkyServer Database*. WDAS 2002, Paris, Carleton Scientific.

[F01] Foster, I. *The Anatomy of the Grid: Enabling Scalable Virtual Organizations*, IJSA, 2001.

[FNPS79] Fagin, R., Nivergelt, J., Pippengar, N. & Strong, H.R. *Extendible Hashing – A Fast Access Method for Dynamic Files*, ACM TODS, vol.4, n° 3, septembre 1779, p. 315-344.

[G99] Gray, J. *Turing Award Lecture: What Next?* ACMComputer Conference, Atlanta, Georgia, 4 May 1999.

[Gar99] Gardarin, G. Bases de Données. Eyrolles, ISBN 2-212-11281-5. 1ère édition. 1999.

[GW97] Grimshaw, A. & Wulf, W. *The Legion Vision of a WorldWide Virtual Computer*. Comm. Of ACM, January, 1997.

[G96] Gray, J. S*uper-Servers: Commodity Computer Clusters Pose a Software Challenge*. Microsoft, 1996.

[G94] Ghernaouti-Hélie, S. *Client/Serveur les outils du Traitement Réparti Coopératif*, Editions Masson 1994.

[Gar93] Gardarin, G. *Maîtriser les Bases de Données*. Eyrolles Edition 1993.

[G93] Gray, J. *Super-Servers: Commodity Computer Clusters Pose a Software Challenge*. http://131.107.1.182:80/research/barc/gray/default.htm.

[G90] Graefe, G. *Encapsulation of parallelism in the volcano query processing system.* In ACM SIGMOD International Conference. Atlantic City, 1990.

[G89] Gray, J. *The Cost of Messages*. Proceeding of Principles Of Distributed Systems, Toronto, Canada, 1989.

[GG05] Guinepain, S. & Gruenwald, L. *Research Issues in Automatic Database Clustering*. ACM-SIGMOD, Mars 2005.

[GG96] Gardarin, G. & Gardarin, O. *Le Client-Serveur*, Editions Eyrolles 1996.

[GV91] Gardarin, G. & Valduriez, P. *SGBD avancées*. Eyrolles edition. 1991.

[IBM87] IBM Corporation, *Introduction to IBM Direct Access Storage Devices and Organization Methods*, Student text, Manual form GC20-1649-10.

[k95] Kim, W. *Modern Database Systems: the Object Model, Interoperability and Beyond*, ACM Press and Addison-Wesley, New York, 1995.

[KLR94] Karlson, J. S., Litwin, W. & Risch, T. *LH*LH: A Scalable High Performance Data Structure for Switched Multicomputers*. In Advances in Database Technology - EDBT'96, pages 573-591, Avignon, France, March 1996. Springer.

[KW94] Kroll, B., Vidmayer, P. *Distributing a Search Tree among a Growing Number of Processors*, ACM Intl. Conf. on Management of Data -SIGMOD 1994.

[KTM88] Kitsuregawa, M., Tanaka, H. & Moto-Oka, T. *Architecture and performance of relational algebra machine GRACE*. In Proc. of the Intl. On Prallel Processing, Chicago, 1984.

[KS86] Korth, H. F. & Silberchatz, A. *Database System Concepts*. Mc Graw Hill Inc., New York, 1986.

[K73] Knuth, D. E. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley Publishing Company, Reading, Massachussetts, 1973.

[L80] Litwin, W. *Linear Hashing: a new tool for file and tables addressing*. Reprint from VLDB-80.

[LB05] Loney. K & Bryla. B. *Oracle Database 10g, DBA Handbook: Manage a Robust, Scalable and Highly Available Oracle Database*. Oracle Press. ISBN 0-07-223145-9. 2005.

[L03] Lejeune, H. *Technical Comparison of Oracle vs. SQL Server 2000: Focus on Performance*, December 2003.

[LMS06] Litwin, W., Mokadem, R. & Sahri, S. *Virtual Repository for eGov Life Event Documents*. 2006.

[LMS05] Litwin, W., Moussa, R. & Schwarz, Th. *LH*$_{RS}$, a highly-available scalable distributed data structure.* ACM Transactions on Database Systems (TODS). Volume 30, Issue 3, September 2005. Pages 769-811.

[LMRS99] Litwin, W., Menon, J., Risch, T. & Schwarz, Th. *Design Issues For Scalable Availability LH* Schemes with Record Grouping*. DIMACS Workshop on Distributed Data and Structures, Princeton U. Carleton Scientific, (publ.), 1999.

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

[LNS94] Litwin, W., Neimat, M.-A. & Schneider, D. *RP*: A Family of Order-preserving Scalable Distributed Data Structures*, Proceedings of the 20th VLDB Conference, Satiago, Chili, 1994.

[LNS93a] Litwin, W., Neimat, M.-A. & Schneider, D. *Linear Hashing for Distributed Files*. ACM-SIGMOD International Conference on Management of Data, 1993.

[LNS93b] Litwin, W., Neimat, M-A. & Schneider, D. *LH*: A Scalable Distributed Data Structure.* Submitted for journal publ. *Nov. 1993.*

[LRS02] 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.).

[LSS06a] Litwin, W., Sahri, S. & Schwarz, Th. *Scalable Command Processing in SD-SQL Server: a Scalable Distributed Database System*. 7th Intl. Workshop on Distributed Data and Structures (WDAS-7) Santa Clara, CA, 2006.

[LSS06b] Litwin, W., Sahri, S. & Schwarz, Th. S, *Prototyping a Scalable Distributed Database System SD-SQL Server*. The British National Conference on Databases, BNCOD, July 2006 (to appear).

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

[LS04] 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.

[LS90] Levy, E. & Silberschatz, A. *Distributed file systems: Concepts examples.* ACM Computing Surveys, 22(4), December 1990.

[M00] Microsoft SQL Server 2000: SQL Server Books Online.

[MC99] Musick, R. & Critchlow, T. *Practical Lessons in supporting Large-Scale Computational Science*, ACM SIGMOD Record, December 1999.

[M93] Mohan, C. *IBM's Relational DBMS Products: Features and Technologies*. ACM SIGMOD, May 1993.

[NZT96] Norman, M.G., Zurek, T. & Thanisch, P. *Much Ado About Shared-Nothing*, SIGMOD Record Vol 25 N°3, p.16-21.

[OV99] Özsu, T. & Valduriez, P. *Principles of Distributed Database Systems*, 2ème édition, Prentice Hall, 1999.

[P04] Pillou, J.F. *Le client Serveur*, http://www.commentcamarche.net/cs/csintro.php3.

[PGK88] Patterson, D. A., Gibson, G. & Katz, R. H. *A Case for Redundant Arrays of Inexpensive Disks*, Proc. of ACM SIGMOD Conf, pp.109-106, June 1988.

[RZLM02] Rao, J., Zhang, C., Lohman, G. & Megiddo, N. *Automating Physical Database Design in a Parallel Database*, ACM SIGMOD '2002 June 4-6, USA.

[SLS06] Sahri, S. Litwin, W. & T.Schwartz. *Architecture and Interface of Scalable Distributed Database System SD-SQL Server.* The Intl. Ass. of Science and Technology for Development Conf. on Databases and Applications, IASTED-DBA 2006, to appear.

[S05] Shimp, R. *Oracle Grid Computing*, An Oracle Business White Paper, February 2005.

[SLS05] Sahri, S. Litwin, W. & T.Schwartz. *SD-SQL Server: a Scalable Distributed Database System*, Rapport de Recherche, December 2005.

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

[SL96] Salzberg, B. & Lomet, D. *Special Issue on Online Reorganization*, Bulletin of the

IEEE Computer Society Technical Committee on Data Engineering, 1996.

[SMKB01] Stoica, I., Morris, R., Karger, D., Kaashoek, F. & Balakrishnan, Hari. *Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications*. Proceedings of the *ACM SIGCOMM*, 2001.

[T05] Tripp, K. *Tables et index partitionnés dans SQL Server 2005.* Janvier 2005. SQLskills.com.

[T88] Teradata Corporation. *DBC/ 1012 data base computer concepts and facilities*. Technical Report Teradata Document C02-001-05, Teradata corporation, 1988.

[T85] Teradata Corporation. *DBC/ 1012 data base computer concepts and facilities*. Technical Report Teradata Document C10-0001-02, Release 2.0, November, 1985.

[VBWS98] Vingralek, R., Breitbart, Y., Weikum, G. & Snowball. *Scalable Storage on Networks of Workstations with Balanced Load*. Distributed and Parallel Databases 6(2): 117-156 (1998).

[V93] Valduriez, P. *Parallel Database Systems: Open Problems and New Issues*. In Distributed and Parallel Databases. Pages 137-165. Kluwer Academic Publishers, 1993.

[WBW94] Wingralek, R., Breitbart, Y. & Weikum, G. *Distributed file organization with scalable cost/performance.* In Proc of ACM-SIGMOD, May 1994.

ANNEXE A : TRAITEMENTS INTERNES DES COMMANDES SD-SQL SERVER

Dans cette annexe, nous présentons le code des procédures stockées qui implémentent les commandes SD-SQL Server.

```
CREATE PROCEDURE sd_create_table  @q varchar(5000), @size int, @key_partitioning varchar(50)  AS
DECLARE@table varchar(50), @q1 varchar(5000), @local varchar(50), @db varchar(50), @query varchar(5000), @taille_table
int, @view varchar (50), @trigger varchar(5000), @insert varchar(5000), @exec_query varchar(5000),@ndb varchar (50),@type
varchar(50),@db_t varchar(50), @server varchar(50), @trigger_tab varchar(5000), @dbo varchar(50), @old_table varchar(50),
@job varchar(50)
--------------sélectionner la NDB courante-------------------------------------------------------
select @local=@@servername, @db=db_name()
set @db_t=@local+'.'+@db
set @server=@local+'.'+@db
set @dbo=@local+'.'+@db+'.sd'
------------extraire le nom de la table de la requete----------------------------------------------
select @taille_table=len(SUBSTRING ( @q , CHARINDEX(' ', @q) , 1000 ))
select @table=substring(@q, 1, len(@q)-@taille_table)
-------------------------réecrire et exécuter le nouveau texte de la requête-----------------------------------
set @ndb=(select top 1 node from sd.server)
set @q1= 'create table SD.'+@q
set @old_table='sd.'+@table
set @view=@table
set @table='_'+@local+'_'+@table
set @exec_query='Execute ' +@ndb+'.'+@db+'.dbo.sp_executesql N'" +@q1+  '"
exec dell1.db_1.dbo.sp_rename  '"+@old_table+"' ,  '"+@table+""
exec (@exec_query)
set @query='create view ' +@view+ ' as select * from ' +@local+ '.' +@db+ '.sd.' +@table+ ''
execute (@query)
set @type='true'
set @insert='
insert into ' +@ndb+'.'+@db+'.sd.rp values ('"+@ndb+ "','"+@local+ "','"+@view+"')
insert into ' +@ndb+'.'+@db+'.sd.prim values ('"+@ndb+ "','"+@local+ "','"+@view+"')
insert into  ' +@ndb+'.'+@db+'.sd.size values ('"+@local+ "','"+@view+ "','" +cast(@size as varchar(10))+"')
insert into  ' +@ndb+'.'+@db+'.sd.image values ('"+@view+ "','"+@type+ "', '"+@ndb+ "',1)'
execute (@insert)
set @trigger_tab='CREATE TRIGGER split_trigger' +@table+ ' ON sd.'+@table+'after INSERT AS
EXEC msdb..sp_start_job  '"+@job+"' '
exec (@trigger_tab)
```

Figure A-1 : Structure interne du code de la commande *sd_create_table*

```
CREATE PROCEDURE sd_alter_table @query varchar(5000),@size int AS
declare  @db varchar(50), @db_rp varchar(5000), @q varchar(5000), @alter varchar(5000),@primnd varchar(50), @sgmnd
varchar(50),
@new_table varchar(50), @drop varchar (5000), @local varchar(50), @table varchar(50), @taille_table int, @creatnd
varchar(50), @query1 varchar(5000)
--------------sélectionner la NDB courante-------------------------------------------------------
select @local=@@servername, @db=db_name()
-----------extraire le nom de la table de la requete-----------------------------------------------
select @taille_table=len(SUBSTRING ( @query , CHARINDEX(' ',@query) , 1000 ))
select @table=substring(@query, 1, len(@query)-@taille_table)
select @query=substring(@query, (len(@query)-@taille_table)+1, 5000)
select @primnd= (select primnd from sd.image where name=@table)
create table #creatnd (creatnd varchar(50))
set @query1='insert into #creatnd select creatnd from '+@primnd+'.'+@db+'.sd.prim where tab=  '"+@table+ '""
exec (@query1)
set @creatnd=(select creatnd from #creatnd)
set @new_table='_'+@creatnd+'_'+@table
set @query='alter table sd.'+@new_table+ @query
create table #db_tab (sgmnd varchar(50))
set @db_rp='insert into #db_tab select sgmnd from ' +@primnd+ '.' +@db+ '.sd.rp where tab= '"+@table+ '" '
exec( @db_rp)
if ( exists (select * from #db_tab))  While ( exists (select * from #db_tab))
 BEGIN
  set @sgmnd=(select top 1 sgmnd from #db_tab)
  set @q = ' Execute ' +@sgmnd+'.'+@db+'.dbo.sp_executesql N'"+@query+'"
  update '   +@sgmnd+'.'+@db+ '.sd.size set size= '+cast(@size as varchar(10))+' where tab= '"+@table+ '" '
  exec (@q)
 delete from #db_tab where sgmnd=@sgmnd
END
Else
 begin
     set @alter= ' Execute ' +@sgmnd+'.'+@db+'.dbo.sp_executesql N'"+@query+'"
     update '+@sgmnd+'.'+@db+ '.sd.size set size= '+cast(@size as varchar(10))+' where tab= '"+@table+ '"
     exec (@alter)
   end
```

Figure A-2 : Structure interne du code de la commande sd_alter_table

```
CREATE PROCEDURE sd_create_index @query varchar(5000)AS
declare  @db varchar(50), @db_rp varchar(5000), @table varchar(50),@orginalsite varchar(50),@q varchar(5000),@primnd
varchar(50),@new_table varchar(50),@local varchar(50), @taille_table int,@part1 varchar(5000),@part2 varchar(5000)
--------------sélectionner la NDB courante-------------------------------------------------------
select @local=@@servername, @db=db_name()
------------extraire le nom de la table de la requete-----------------------------------------------
select @taille_table=len(SUBSTRING ( @query , CHARINDEX(' ',@query) , 1000 ))
select @part1=substring(@query, 1, len(@query)-@taille_table)
select @table=substring(SUBSTRING (@query, CHARINDEX('ON',@query)+3, 1000 ), 1, CHARINDEX('(',SUBSTRING
(@query, CHARINDEX('ON',@query)+3, 1000 ))-1)
select @part2=substring(SUBSTRING (@query, CHARINDEX('ON',@query)+3, 1000 ), CHARINDEX('(',SUBSTRING (@query,
CHARINDEX('ON',@query)+3, 1000 )),1000 )
select @primnd= (select primnd from sd.image where name=@table)
set @new_table='_'+@primnd+'_'+@table
set @query='create index '+@part1+' on sd.'+ @new_table+@part2
create table #db_tab (sgmnd varchar(50))
set @db_rp='insert into #db_tab select sgmnd from ' +@primnd+ '.' +@db+ '.sd.rp where tab= '"+@table+ '" '
exec( @db_rp)
while ( exists (select * from #db_tab))
 begin
  set @primnd=(select top 1 sgmnd from #db_tab)
  set @q = ' Execute ' +@primnd+'.'+@db+'.dbo.sp_executesql N'"+@query+'" '
  exec (@q)
  delete from #db_tab where sgmnd=@primnd
 end
```

Figure A-3 : Structure interne du code de la commande *sd_create_index*

```
CREATE PROCEDURE sd_drop_index @index varchar(50)  AS
declare @table varchar(50), @primnd varchar(50), @db varchar(50), @creatnd varchar(50), @segment varchar(50),@query_index
varchar(5000),@sgmnd varchar(50),@q varchar(5000), @db_rp varchar(5000), @query varchar(5000)
set @db=db_name()
select @table=substring(@index, 1, len(@index)-len(SUBSTRING ( @index , CHARINDEX('.', @index) , 1000 )))
select @index= substring(@index, CHARINDEX('.', @index)+1 ,1000)
sselect @primnd =(select primnd from sd.image where name=@table)
create table #creatnd (creatnd varchar(50))
set @query='insert into #creatnd select creatnd from '+@primnd+'.'+@db+'.sd.prim where tab=  "'+@table+ ""
exec (@query)
set @creatnd=(select creatnd from #creatnd)
set @segment='_'+@creatnd+'_'+@table
set @query_index='drop index sd.'+@segment+'.' +@index
create table #db_tab (sgmnd varchar(50))
set @db_rp='insert into #db_tab select sgmnd from ' +@primnd+ '.' +@db+ '.sd.rp where tab= "'+@table+ '" '
exec( @db_rp)
if ( exists (select * from #db_tab))
While ( exists (select * from #db_tab))
 BEGIN
  set @sgmnd=(select top 1 sgmnd from #db_tab)
  set @q = '   Execute ' +@sgmnd+'.'+@db+'.dbo.sp_executesql N'"+@query_index+'"  '
  exec (@q)
   delete from #db_tab where sgmnd=@sgmnd
end
```

Figure A-4 : Structure interne du code de la commande *sd_drop_index*

```
CREATE procedure sd_drop_table @table varchar(50) as
declare @site varchar(50), @db varchar(50), @query varchar(5000), @q varchar(5000),@view varchar (50), @trigge
varchar(5000), @insert varchar(5000), @db_t varchar(50), @server varchar(50), @trigger_tab varchar(5000), @dbo
varchar(50),@primnd varchar(50), @sgmnd varchar(50),@db_rp varchar(5000),@drop_view varchar(50),@drop_table varchar(50)
@new_table varchar(50), @drop varchar (5000), @local varchar(50)
--------------sélectionner la NDB courante-----------------------------------------------------------
select @local=@@servername, @db=db_name()
set @db_t=@local+'.'+@db
set @server=@local+'.'+@db
set @dbo=@local+'.'+@db+'.dbo'
select @primnd= (select primnd from sd.image where name=@table)
set @new_table='_'+@primnd+'_'+@table
set @drop_view='drop view '+@table+''
set @drop_table='drop table sd.'+@new_table+' '
create table #db_tab (sgmnd varchar(50))
set @db_rp='insert into #db_tab select sgmnd from ' +@primnd+ '.' +@db+ '.sd.rp where tab= '''+@table+ ''' '
exec( @db_rp)
if ( exists (select * from #db_tab))
While ( exists (select * from #db_tab))
 BEGIN
  set @sgmnd=(select top 1 sgmnd from #db_tab)
  set @query =
            'delete from '+@sgmnd+ '.' +@db+'.sd.rp where tab= '''+@table+'''
             delete from '+@local+ '.' +@db+'.sd.image where name= '''+@table+'''
             delete from '+@sgmnd+'.' +@db+'.sd.prim where tab='''+@table+'''
             delete from '+@sgmnd+'.' +@db+'.sd.size where tab='''+@table+'''
             Execute ' +@local+ '.' +@db+'.dbo.sp_executesql N'''+@drop_view+'''
             Execute ' +@sgmnd+ '.' +@db+'.dbo.sp_executesql N'''+@drop_table+'''
   exec (@query)
   delete from #db_tab where sgmnd=@sgmnd
 END
else    begin
      set @drop=' delete from '+@local+ '.' +@db+'.sd.image where name= '''+@table+'''
              delete from '+@sgmnd+'.' +@db+'.sd.prim where tab='''+@table+'''
              delete from '+@sgmnd+'.' +@db+'.sd.size where tab='''+@table+'''
              Execute ' +@local+ '.' +@db+'.dbo.sp_executesql N'''+@drop_view+'''
              Execute ' +@sgmnd+ '.' +@db+'.dbo.sp_executesql N'''+@drop_table+'''
       exec (@drop)
end
```

Figure A-5 : Structure interne du code de la commande *sd_drop_table*

```
CREATE PROCEDURE sd_create_image @primary_node varchar(50), @table varchar(50)  AS
declare @db varchar(50), @secondary varchar(50), @creatnd varchar(50), @query varchar(5000), @q varchar(5000),@view
varchar(5000),@txt varchar(5000),@primary varchar(5000),@exec varchar(5000),
@type varchar(50),@size int, @query_size varchar(5000)

set @db=db_name()
create table #creatnd (creatnd varchar(50))
create table #size (size int)
set @query='insert into #creatnd select creatnd from '+@primary_node+'.'+@db+'.sd.prim where tab= '"+@table+ '" '
exec (@query)
set @creatnd=(select creatnd from #creatnd)
set @secondary='sd.'+@creatnd+'_'+@table
set @query_size ='insert into #size select size from '+@creatnd+'.'+@db+'.sd.Image where tab= '"+@table+ '" '
exec (@query_size)
set @size=(select size from #size)
create table #t (txt varchar(5000))
set @q =  'Execute ' +@primary_node+'.'+@db+'.dbo.help_text_image '"+@table+'" '
insert into #t exec (@q)
set @txt=(select top 1 txt from #t)
select @primary=substring(@txt, CHARINDEX('as',@txt)+3 , 5000)
set @view='create view ' +@secondary+' as '+@primary
set @type='Secondary'
set @exec=
'Execute dbo.sp_executesql N'" +@view+  '"
insert into SD.Image values ('"+@table+ '","'+@type+ '", '"+@primary_node+ '",1)'
exec (@exec)
```

Figure A-6 : Structure interne du code de la commande *sd_create_image*

```
CREATE PROCEDURE sd_drop_image @image varchar(50) AS
declare @drop varchar(5000), @secondary varchar(50)
set @secondary=substring(@image,  CHARINDEX('_', @image)+1, 100)
set @drop=' drop view sd.'+@image+ '
            delete from from SD.Image where tab="'+@secondary+'"  '
exec (@drop)
```

Figure A-7 : Structure interne du code de la commande *sd_drop_image*

```
CREATE PROCEDURE sd_select @query varchar(5000) AS
DECLARE @view varchar (20), @t numeric (10,5), @table varchar (50),@q varchar(5000),
@sd_db varchar(50), @db varchar(50), @local_serv varchar(50), @nb_image int,
@patindex int, @miquery varchar(5000), @int1 int, @int2 int, @view_query varchar(1000),
@query_no_where varchar(5000)
----------
SELECT @query_no_where=substring(@query, 1,len(@query)-len(substring( @query, patindex('%where%',@query),100)))

SELECT @patindex=CHARINDEX('from', @query)
select @int1=len(@query)
select @int2=len(SUBSTRING ( @query , @patindex , 100 ))-4
select "clause from" =SUBSTRING ( @query , @patindex , 100 )
select @miquery=SUBSTRING(@query , 1 , @int1-@int2 )+'%s%s'
EXEC master.dbo.xp_sscanf @query, @miquery,@view_query OUTPUT
-------------------------
while len(@view_query)>0
BEGIN
   SELECT @patindex=CHARINDEX(',', @view_query)
  if @patindex=0
  begin
    exec analyse_obj_from @view_query
    break
  end
  ELSE
  BEGIN
   select @view=SUBSTRING( @view_query , 1 , @patindex-1 )
   select @view_query=substring(@view_query, @patindex+1,len(@view_query)-len(@view))
exec analyse_obj_from @view
 END
-----------------------------**************fin traitement*************----------------------
END
set @query='select '+@query
exec (@query)
```

Figure A-8 : Structure interne du code de la commande *sd_select*

```
CREATE PROCEDURE sd_update @query varchar(5000) AS
DECLARE @view varchar (20), @t numeric (10,5), @table varchar (50),@q varchar(5000),@local varchar(50),@sd_db
varchar(50), @db varchar(50), @local_serv varchar(50), @nb_image int,@patindex int, @miquery varchar(5000), @int1 int,
@int2 int,
@view_query varchar(1000),@query_no_update varchar(5000)
SELECT @query_no_update=substring(@query, patindex('%select%',@query),len(@query)-len(substring( @query,
patindex('%select%',@query),100)))
------------sélectionner la NDB courante----------------------------------------------------------
select @local=@@servername, @db=db_name()
------------extraire le nom de la table de la requete-------------------------------------------
select @table=substring(@query, 1, len(@query)-len(SUBSTRING ( @query , CHARINDEX(' ',@query) , 1000 )))
exec analyse_obj_from @table
-------------------------------------------------
if @query_no_update is not null
begin
SELECT @patindex=CHARINDEX('from', @query_no_update)
select @int1=len(@query_no_update)
select @int2=len(SUBSTRING ( @query_no_update , @patindex , 100 ))-4
select "clause from" =SUBSTRING ( @query_no_update , @patindex , 100 )
select @miquery=SUBSTRING(@query_no_update , 1 , @int1-@int2 )+'%s%s'
EXEC master.dbo.xp_sscanf @query_no_update, @miquery,@view_query OUTPUT
--------------------------
while len(@view_query)>0
BEGIN
  SELECT @patindex=CHARINDEX(',', @view_query)
  if @patindex=0
  begin
    exec analyse_obj_from @view_query
    break
  end
  ELSE
  BEGIN
   select @view=SUBSTRING( @view_query , 1 , @patindex-1 )
     select @view_query=substring(@view_query, @patindex+1,len(@view_query)-len(@view))
   exec analyse_obj_from @view
  END
END
end
exec (@query)
```

Figure A-9 : Structure interne du code de la commande *sd_update*

```
CREATE PROCEDURE sd_insert @query varchar(5000) AS

DECLARE @view varchar (20), @t numeric (10,5), @table varchar (50),@q varchar(5000),@sd_db varchar(50), @db varchar(50),
@local_serv varchar(50), @nb_image int,@patindex int, @miquery varchar(5000), @int1 int, @int2 int, @view_query
varchar(1000),@query_no_insert varchar(5000)

SELECT @query_no_insert=substring(@query, patindex('%select%',@query),len(@query)-len(substring( @query,
patindex('%select%',@query),100)))
if @query_no_insert is not null
begin
SELECT @patindex=CHARINDEX('from', @query_no_insert)
select @int1=len(@query_no_insert)
select @int2=len(SUBSTRING ( @query_no_insert , @patindex , 100 ))-4
select @miquery=SUBSTRING(@query_no_insert , 1 , @int1-@int2 )+'%s%s'
EXEC master.dbo.xp_sscanf @query_no_insert, @miquery,@view_query OUTPUT
---------------------------
while len(@view_query)>0
BEGIN
  SELECT @patindex=CHARINDEX(',', @view_query)
  if @patindex=0
  begin
    exec analyse_obj_from @view_query
    break
  end
  ELSE
  BEGIN
   select @view=SUBSTRING( @view_query , 1 , @patindex-1 )
   select @view_query=substring(@view_query, @patindex+1,len(@view_query)-len(@view))
   exec analyse_obj_from @view
  END
-----------------------------**************fin traitement*************----------------------
END
end
set @query='insert '+@query
exec (@query)
```

Figure A-10 : Structure interne du code de la commande *sd_insert*

```
CREATE PROCEDURE sd_create_node   @node varchar(50), @type varchar(50)   AS

insert into dell1.metabase.sd.nodes (node,type) values (@node,@type)
```

Figure A-11 : Structure interne du code de la commande *sd_create_node*

```
CREATE PROCEDURE sd_create_scalable_database @db varchar(50),@node varchar(50), @type varchar(50), @extent int
AS
declare @query varchar(5000),@data varchar(5000), @node_extent varchar(50)
set @query='exec '+@node+'.db_1.dbo.create_sdb  '''+@db+''', '''+@type+''' '
set @data='insert into dell1.metabase.sd.sdb values ('''+@db+ ''','''+@node+ ''','''+@type+''') '
exec (@data)
while (@extent >1)
begin
set @extent=@extent-1
select @node_extent=node from sd.nodes where type=@type and node!=@node
exec sd_create_node_database @db,@node_extent, @type
end
```

Figure A-12 : Structure interne du code de la commande *sd_create_scalable_database*

```
CREATE PROCEDURE sd_drop_node_database @sdb varchar(50), @node varchar(50)  AS
declare @query varchar(5000),@data varchar(50),@sdb_node varchar(50), @type varchar(50), @query_type varchar(5000),
@query_image varchar(5000), @image varchar(50),@q varchar(5000), @query_node varchar(5000), @new_node varchar(50),
@insert varchar(5000)
select @sdb_node = (select node from sd.sdb where sdb_name=@sdb)
create table #type (NDBtype varchar(50))
set @query_type =' insert into #type select NDBtype from '+@sdb_node+'.'+@sdb+'.sd.ndb where node= '''+@node+ ''' '
exec (@query_type)
set @type=(select NDBtype from #type)
if (@type='client')
begin
create table #image (name varchar(50), type varchar(50), primnd varchar(50),size int)
set @query_image =' insert into #image select *  from '+@node+'.'+@sdb+'.sd.image '
exec (@query_image)
if ( exists (select * from #image))
While ( exists (select * from #image))
 BEGIN
  set @image=(select top 1 name from #image)
  set @q = ' Execute '+@node+'.'+@sdb+ '.dbo.sd_drop_table '''+@image+'''        '
  exec (@q)
  delete from #image where name=@image
  end
end
else if (@type='server')
begin
create table #node (node varchar(50))
set @query_node =' insert into #node select node from dell1.metabase.sd.nodes  where node not in (select node from '+@sdb_node+
'.' +@sdb+ '.sd.ndb) '
exec (@query_node)
set @new_node=(select node from #node)
if (@new_node is not null)
begin
exec sd_create_node_database @sdb, @new_node, 'server'
set @insert ='insert into '+@new_node+'.' +@sdb+'.sd.prim select * from ' +@node+'.' +@sdb+'.sd.prim
insert into '+@new_node+'.' +@sdb+'.sd.size select * from ' +@node+'.' +@sdb+'.sd.size'
end end
set @query=' exec '+@node+'.db_1.dbo.drop_ndb '''+@sdb+'''  '
exec (@query)
```

Figure A-13 : Structure interne du code de la commande *sd_drop_node_database*

```
CREATE PROCEDURE sd_drop_scalable_database @sdb varchar(50) AS
declare @sdb_node varchar(50),@query varchar(5000),@ndb varchar(50),@q varchar(5000)
select @sdb_node=(select node from dell1.metabase.sd.sdb where sdb_name=@sdb)

create table #ndb (node varchar(50))
set @query='insert into #ndb select node from '+@sdb_node+'.' +@sdb+'.sd.ndb'
exec (@query)
if ( exists (select * from #ndb))
While ( exists (select * from #ndb))
 BEGIN
  set @ndb=(select top 1 node from #ndb)
  set @q = '
        exec '+@ndb+'.db_1.dbo.drop_ndb '''+@sdb+'''
        delete from  '+@sdb_node+'.'+@sdb+'.sd.ndb where node=  '''+@ndb+ '''   '
  exec (@q)
  delete from #ndb where node=@ndb
END
delete from dell1.metabase.sd.sdb where sdb  name=@sdb
```

Figure A-14 : Structure interne du code de la commande *sd_drop_scalable_database*

GLOSSAIRE

B

BD Client – Une NDB client gère seulement l'interface application/utilisateur.

BD Pair – Une NDB de type pair unifie les capacités d'une NDB de type serveur et une NDB de type client.

BD Serveur – Une BD(NDB) de type serveur sauvegarde les segments des tables scalables sans avoir leur interface application/utilisateur.

Base de Données Scalable – Une base de données scalable (SDB) est une collection dynamique de bases de nœuds SD-SQL Server distribuées. SD-SQL Server peut ajouter ou supprimer des bases de nœuds d'une SDB d'une façon dynamique.

Base de Nœud – Une base de nœud (NDB) est une base de données enregistrée comme une base de données d'une SDB. Les NDBs partagent le même nom que leur SDB. Une NDB peut être une base de données de type client, serveur ou pair. Chaque NDB du système SD-SQL Server est présentée par une base de données du SGBD SQL Server. Chaque base de données détient un gestionnaire spécifique à SD-SQL Server.

C

Capacité d'un Segment – La capacité d'un segment est le nombre maximum de ses tuples. Si une insertion fait dépasser à un segment sa capacité, le segment deviendra surchargé et éclatera. Tous les segments d'une table scalable ont la même capacité.

Commande SD-SQL Server – Une commande SD-SQL Server est une commande de l'interface application/utilisateur SD-SQL Server. Les commandes SD-SQL Server permettent de gérer les tables scalables, les NDBs, les SDBs et les nœuds. Les commandes sur les tables scalables généralisent les commandes SQL usuelles. Elles son typiquement nommées sur leur initiales (exp. SD_SELECT au lieu de SELECT).

E

Eclatement – Un éclatement est déclenché lorsqu'un segment excède sa capacité. Il crée un ou plusieurs nouveaux segments pour y transférer les tuples, excédant le segment qui éclate, tout en laissant tous les segments à moitié pleins. L'éclatement est effectué selon un partitionnement par intervalle. Il est lancé par un déclencheur AFTER disponible au niveau de chaque segment. Ce déclencheur vérifie la surcharge d'un segment lors de toute insertion dans celui-là. Chaque éclatement fait appel à l'éclateur (splitter).

G

Gestionnaire SD-SQL Server – Un gestionnaire SD-SQL Server est une implémentation prototype du concept d'un gestionnaire SD-DBS. Un gestionnaire SD-SQL Server est opérationnellement une collection de procédures stockées sur chaque NDB.

I

Image Binding – L'image binding détermine dans une requête SQL chaque image sur laquelle dépend le nom d'une table ou d'une vue. Elle concerne la clause FROM des expressions SELECT.

Image Primaire – Une image primaire est la définition du partitionnement d'une table scalable. Elle est crée sur la NDB client qui crée la table scalable. Elle est présentée comme un vue partitionnée distribuée. Elle définit l'union de tous les segments scalables qui composent une table scalable. Si un segment d'une table scalable n'est pas défini dans l'image primaire, l'image sera alors ajustée.

Image Secondaire – Une image secondaire est une vue partitionnée distribuée qui définit le partitionnement actuel d'une table scalable. Elle est crée sur un NDB client différent de celui qui a crée la table scalable.

Interface d'Application – Une interface d'application manipule les tables scalables et leurs vues par l'intermédiaire des commandes SD-SQL Server essentiellement.

M

Métabase – La méta-base (MDB) est une base de données spécifique à SD-SQL Server. Elle se trouve sur le nœud primaire du SD-SQL Server. Elle enregistre les métadonnées sur tous les nœuds et toutes les SDBs qui constituent la configuration courante de SD-SQL Server. Elle contient aussi le code SD-SQL Server sous forme de procédures stockées.

N

Noeud Client – Un nœud client est un nœud qui détient des bases de données de type client.

Noeud libre – Pour une SDB, un nœud libre est un nœud qui ne détient aucune NDB de la SDB. Un nœud libre peut utiliser une NDB d'une autre SDB.

Noeud Pair – Un nœud pair est un nœud qui joue le rôle d'un nœud serveur et d'un nœud client en même temps. Ainsi, le nœud pair peut contenir des bases de données de type client et de type serveur.

Noeud Primaire – Un noeud primaire est le premier noeud crée dans la configuration d'un SD-DBS. Il est crée sur SD-SQL Server en exécutant le script *install*. Il détient la métabase (MDB) pour enregistrer les méta-données sur les autres nœuds (secondaire) du SD-DBS.

Nœud SD-SQL Server – Un nœud SD-SQL Server est un nœud SQL Server lié (SQL linked server). Il représente l'implémentation d'un nœud SD-DBS spécifique au SGBD SQL Server.

Nœud Serveur – Un nœud serveur est un nœud qui détient des bases de données de type serveur.

R

Requête Scalable – Une requête scalable est une requête qui peut invoquer une table scalable à travers son image ou à travers une vue scalable de son image (indirectement). Elle s'exécute

correctement pour n'importe quelle modification dynamique du nombre de segments de la table scalable invoquée. Une requête scalable est formulée sur l'interface usager/application SD-SQL Server. Elle utilise des commandes SQL dédiées au SGBD SQL Server. Une requête scalable peut adresser aussi des tables statiques (crée sur SQL Server).

S

Script Install – Un script d'installation est un script SQL Server qui permet, lors de son exécution, l'installation du nœud primaire de SD-SQL Server.

SD-SQL Server – SD-SQL Server est un SD-DBS désignant les données dans des bases de données SQL Server. Il gère les bases de données scalables et les tables scalables sur une collection de nœuds SD-SQL Server.

Segment – Un segment est une table qui représente un fragment d'une table scalable. Les segments partitionnent leur table scalable selon un partitionnement par intervalle. Un segment peut être primaire ou secondaire. Un segment primaire est crée sur une NDB client au moment de la création de sa table scalable. Un segment secondaire résulte de l'éclatement d'un autre segment d'une table scalable.

Splitter – Le splitter est un processus asynchrone qui est lancé par le déclencheur d'un éclatement. Il réalise l'éclatement d'une façon asynchrone avec l'insertion qui cause la surcharge.

Système de Gestion de Bases de Données Distribuées et Scalable – Un Système de gestion de bases de données distribuées et scalables (SD-DBS) applique les principes des structures de données distribuées et scalables (SDDSs) sur les systèmes de bases de données. Il gère les bases de données scalables avec des tables scalables qui montent en échelle à travers les nœuds SD-DBS.

T

Table Scalable – Une table scalable T est formellement un tuple (T, S) où T est l'image primaire de la table scalable T et S est l'ensemble de ses segments distribués sur des NDBs. Une table scalable monte en échelle suite à l'éclatement de ses segments surchargés.

Table Statique – Une table statique est une table qui est inconnue par SD-SQL Server. Toute table relationnelle, dans les SGBDs actuels, est une table statique pour SD-SQL Server, et donc ce n'est pas une table scalable.

V

Vue Scalable – Une vue scalable est une vue qui peut dépendre d'une table scalable directement ou à travers une vue. Une vue scalable peut adresser une table scalable à travers le nom de son image primaire ou à travers une de ses images secondaires.

Vue Statique – Une vue statique est une vue qui n'est pas scalable, donc c'est une vue inconnue pour SD-SQL Server.

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

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

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

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

1. Introduction

Databases are now often huge and growing at a high rate. Large tables are then typically hash or range partitioned into segments stored at different storage sites. Current Data Base Management Systems (DBSs), e.g., SQL Server, Oracle or DB2, provide static partitioning only [1],[5],[11]. The database administrator (DBA) in need to spread these tables over new nodes has to manually redistribute the database (DB). A better solution has become urgent, [1].

This situation is similar to that of file users forty years ago in the centralized environment. Efficient management of distributed data present specific needs. The Scalable Distributed Data Structures (SDDSs) addressed these needs for files, [6][7]. An SDDS scales transparently for an application through distributed splits of its buckets, hash, range or k-d based. In [8], the concept of a Scalable Distributed DBS (SD-DBS) was derived for databases. The SD-DBS architecture supports the *scalable (distributed relational) tables*. As an SDDS, a scalable table accommodates its growth through the splits of its overflowing segments, located at SD-DBS *storage* nodes. Also like in an SDDS, the splits can be in principle hash, range or k-d based with respect to the partitioning key(s). The storage nodes can be P2P or grid DBMS nodes. The users or the application, manipulate the scalable tables from a *client* node that is not a storage node, or from a *peer* node that is both, again as in an SDDS. The client accesses a scalable table only through its specific view, termed (*client*) *image*. It is a particular updateable distributed partitioned union view stored at a client. The application manipulates scalable tables using *scalable* (application) views. These views involve scalable tables through the references to the images.

Every image, one per client, hides the scalable table partitioning and dynamically adjusts to its evolution. The images of the same scalable table may differ among the clients and from the actual partitioning. The image adjustment is lazy. It occurs only when a query to the scalable table finds an outdated image. To prove the feasibility of an SD-DBS, we have built the prototype termed SD-SQL Server. The system generalizes the basic SQL Server capabilities to the scalable tables. It runs on a collection of SQL Server linked nodes. For every standard SQL command under SQL Server, there is an SD-SQL Server command for a similar action on scalable tables or views. There are also commands specific to SD-SQL Server client image or node management.

Below we present the architecture of our prototype and its application command interface as it stands in its 2005 version, [14]. The current architecture addresses more features than [8]. With respect to the interface, we discuss the syntax and semantics of each command. Numerous examples illustrate the actual use of SD-SQL Server. We hope to convince that the use of the scalable tables should be about as simple as of the static ones in practice.

The related papers discussed the internal design and the processing performance of SD-SQL Server, [9], [14]. The scalable table processing creates an overhead and our design challenge was to minimize it [3]. The performance analysis proved this overhead negligible for practical purpose. The present capabilities of SQL Server let a scalable table to reach 250 segments at least. This should suffice for scalable tables reaching very many terabytes. SD-SQL Server is the first system with the discussed capabilities, to the best of our knowledge. Our results pave the way towards the use of the scalable tables as the basic DBMS technology.

Below, Section 2 presents the SD-SQL Server architecture. Section 3 discusses the user interface. Section 4 discusses the related work. Section 5 concludes the presentation.

2. SD-SQL Server Architecture

Fig 1 shows the current SD-SQL Server architecture, adapted from the reference architecture for an SD-DBS in [8]. The system is a collection of SD-SQL Server nodes. An *SD-SQL Server node* is a linked SQL Server node that in addition is declared as an SD-SQL Server node. This declaration is made as an SD-SQL Server command or is part of a dedicated SQL Server script run on the first node

of the collection. We call the first node the *primary node*. The primary node registers all other current SD-SQL nodes. We can add or remove these dynamically, using specific SD-SQL Server commands. The primary node registers the nodes on itself, in a specific SD-SQL Server database called the *meta-database* (MDB). An *SD-SQL Server database* is an SQL Server database that contains an instance of SD-SQL Server specific *manager* component. A node may carry several SD-SQL Server databases.

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

Each manager takes care of the SD-SQL Server specific operations, the user/application command interface especially. The procedures constituting the manager of an NDB are themselves kept in the NDB. They apply internally various SQL Server commands. The SQL Servers at each node entirely handle the inter-node communication and the distributed execution of SQL queries. In this sense, each SD-SQL Server runs at the top of its linked SQL Server, without any specific internal changes of the latter.

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

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

When a scalable query comes in, the client checks whether it actually involves a scalable table. If so, it must address its image, directly or through a scalable view. The client searches therefore for the images that the query invokes. For every image, it checks whether it conforms to the actual partitioning of its table, i.e., unions all the existing segments. We recall that a client view may be outdated. The client uses **C**, as well as some server meta-tables pointed to by **C**, defining the actual partitioning. The manager dynamically adjusts any outdated image. In particular, it changes internally the scheme of the underlying SQL Server partitioned and distributed view, representing the image to the SQL Server. The manager executes the query, when all the images it uses prove up to date.

A *server* NDB stores the segments of scalable tables. Every segment at a server belongs to a different table. At each server, a segment is internally an SQL Server table with specific properties. First, SD-SQL Server refers to in the specific catalog in each server NDB, named **S** in the figure. The meta-data in **S** identify the scalable table each segment belongs to. They indicate also the segment size. Next, they indicate the servers in the SDB that remain available for the segments created by the splits at the server NDB. Finally, for a *primary* segment that is the 1st one created for a scalable table, the meta-data at its server provide the actual partitioning of the table.

Next, each segment has an *AFTER* trigger attached, not shown at the figure. It verifies after each insert whether the segment overflows. If so, the server splits the segment, by range partitioning it with respect to the table (partition) key. It moves out enough upper tuples so make the remaining (lower) tuples fitting the splitting segment size. For the migrating tuples, the server creates remotely one or more new segments that are each half-full (notice the difference to a B-tree split creating a single new segment). Furthermore, every segment in a multi-segment scalable table carries an SQL Server *check constraint*. Each constraint defines the partition (primary) key range of the segment. The ranges partition the key space of the table. These conditions let the SQL Server distributed partitioned view to be updateable, by the inserts and deletions in particular. This is a necessary and sufficient condition for a scalable table under SD-SQL Server to be updateable as well.

Finally a *peer* NDB is both a client and a server NDB. Its node DB carries all the SD-SQL Server meta-tables. It may carry both the client images and the segments. The meta-tables at a peer node form logically the catalog termed **P** at the figure. This one is operationally, the union of **C** and **S** catalogs.

To illustrate the architecture, Fig 1 shows the NDBs of some SDB, on nodes *D1*…*Di+1*. The NDB at *D*1 is a client NDB that thus carries only the images and views, especially the scalable ones. This node could be the primary one, being only of type peer or client. It interfaces the applications. The NDBs on all the other nodes till *Di* are servers. They carry only the segments and do not interface any applications. The nodes could be peer or server, only. Finally, the NDB at *Di*+1 is a peer, providing all the capabilities. Its node has to be a peer node. The NDBs carry a scalable table termed *T*. The table has a scalable index *I*. We suppose that *D1* carries the primary image of *T*, locally named *T*. The image unions the segments of *T*, at servers *D*2…*Di*, with the primary segment at *D*2. Peer *Di+1* carry a secondary image of *T*. That one is supposed different, including the primary segment only. Both images are outdated. Server *Di* just split indeed its segment and created a new segment of *T* on *Di*+1. It updated the meta-data on the actual

partitioning of *T* at *D*2. None of the two images refers to this segment as yet. Each will be actualized only once it gets a scalable query to *T*. The split has also created the new segment of *I*.

Notice finally in the figure that segments of *T* are all named *_D1_T*. This represents the couple (creator node, table name). Notice here only that the name provides the uniqueness with respect to different client (peer) NDBs in an SDB.

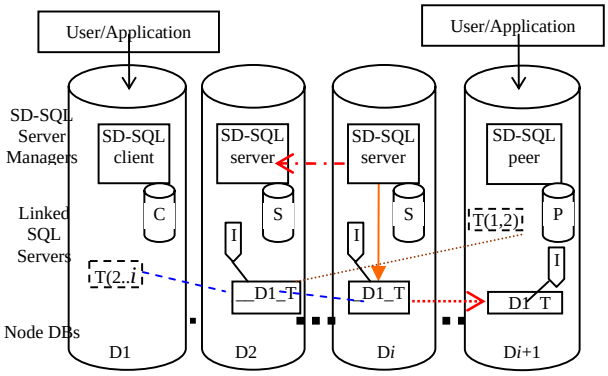

Fig 1 SD-SQL Server Architecture

3. Application Interface

3.1 Overview

The application manipulates SD-SQL Server objects essentially through new SD-SQL Server dedicated commands. The commands for the tables and views perform the usual SQL schema manipulations and queries implying however now the scalable tables (through the images) or the (scalable) views of the scalable tables. We qualify these commands of *scalable*. They address all the existing segments, regardless of their actual number, and their effects may propagate to the future ones. A scalable command may include additional parameters specific to the scalable environment, with respect to its original static counterpart.

Most SD-SQL Server commands apply also to the static tables and views. The application using SD-SQL Server may also directly invoke the (static) SQL Server commands. These calls are transparent to SD-SQL Server managers. Their use should remain limited to the static tables.

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

The operational capabilities of SD-SQL Server scalable commands are sufficient for most applications. The *SELECT* statement in a scalable query supports the SQL Server allowed selections, restrictions, joins, sub-queries, aggregations, aliases…etc. However, the queries to the scalable multi-database views are not possible at present. The reasons are the limitation of the SQL Server meta-tables that SD-SQL Server uses for the parsing, [14], [15]. Moreover, the scalable *INSERT* command over a scalable table lets for any insert accepted by SQL Server for a distributed partitioned view. This can be a new tuple insert, as well as a multi-tuple insert through a *SELECT* expression. The *UPDATE* and *DELETE* statement offer similar capabilities. In contrast, some of SQL Server specific SQL clauses are not supported at present by the scalable commands; for instance, the *CASE OF* clause.

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

3.2 Node Management

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

Node creation. One expands the existing SD-SQL Server configuration with new nodes through the following command:

sd_create_node 'new_node[, node_type]

The *'new_node'* parameter is the name of the spare for the new node. The node executing the command initiates in particular the meta-data of the new one, according to its *'node_type'* parameter. It can be *server*, *client* or *peer*. The default is *server*.

Example 1. The script has created the primary SD-SQL Server node at SQL Server linked node at our *Dell1* machine. We could set up this node as server or peer, consider that we made the latter choice. The following commands issued at Dell1 create further nodes:

sd_create_node 'Dell2' /* Server by default */

sd_create_node 'Dell3, 'client'

sd_create_node 'Ceria1','peer'

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

sd_drop_ node 'node_name'

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

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

sd_drop_ node 'Ceria1'

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

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

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

sd_alter_ node 'Dell3', 'ADD server'

3.3 Scalable Database Management

Creation. We create an SDB using the command:

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

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

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

sd_create_scalable_database 'SkyServer, 'Dell1'

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

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

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

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

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

We drop an NDB by the command:

sd_drop_node_database 'sdb_name', 'node_name'

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

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

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

Removal. We drop an SDB using the command:

sd_drop_scalable_database 'db_name'

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

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

sd_drop_scalable_database 'SkyServer'

3.4 Scalable table management

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

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

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

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

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

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

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

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

The user creates furthermore the scalable table *Neighbors*, modelled upon the static table with the same name in *Skyserver*, [2]. That table has three key attributes. The

objid is one of them and is the foreign key of *PhotoObj*. For this reason, the user wishes it to be the partition key. Finally, s/he chooses the segment capacity to be 500 tuples. Accordingly, the user issues the command:

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

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

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

The command carries at least one of its clauses. The '*SQL: ALTER TABLE* clauses' parameter contains the SQL Server *ALTER TABLE* clauses with their usual syntax. Accordingly, the SD-SQL Server command provides the same capabilities for a scalable table. SD-SQL Server propagates the alteration to every segment. However, the effect of the decrease to the segment size is lazy. No segment splits before next insert into it.

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

sd_alter_table *'PhotoObj ADD t INT, 10000*

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

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

sd_drop_index 'SQL: Drop Index T.I clauses'

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

sd_create_index 'run_index ON Photoobj (run)'

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

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

sd_drop_table ['SQL: DROP TABLE T clauses']

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

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

sd_drop_table 'Dell3. SkyServer.PhotoObj'

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

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

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

sd_drop_image '[image_name]'

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

sd_create_image 'Ceria1', 'PhotoObj'

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

sd_drop_image 'SD.Dell3_Photoobj'

3.5 Scalable Queries

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

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

The *'SQL: Select clauses'* parameter is the SQL *SELECT* command clauses with their usual syntax. The application may invoke in the scalable query the aggregations, joins, aliases, sub-queries…etc. The *'Segment Size'* etc parameters are optional. They serve *SELECT INTO* clause creating a scalable table. The *'Primary Key'* and *'Partiton Key'* address the new table, whenever needed. The columns can be inherited from the source tables, or created by the query e.g., by the aggregate functions. The application may use in the query any SQL Server table or view name, i.e., local or prefixed. Only the local names may however designate a scalable table or view at present.

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

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

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

sd_select '* INTO PhotoObj1 FROM PhotoObj', 500

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

sd_ insert 'SQL Insert clauses'

sd_update 'SQL: Update clauses'

sd_delete 'SQL: Delete clauses'

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

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

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

4. Related Works

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

The editors of [12] contributed themselves with two on-line reorganization methods, termed respective *new-space* and *in-place* reorganization. The former method created a new disk structure, and switches the processing to it. The latter approach balanced the data among existing disk pages as long as there was room for the data. Among the other contributors to [12], concerned a command named '*Move Partition Boundary*' for Tandem Non Stop SQL/MP. The command aimed on on-line changes to the adjacent database partitions. The new boundary should decrease the load of any nearly full partition, by assigning some tuples into a less loaded one. The command was intended as a manual operation. We could not find whether it was ever realized.

A more recent proposal of efficient global reorganizing strategy is in [11]. One proposes there an automatic advisor, balancing the overall database load through the periodic reorganizing. The advisor is intended as DB2 offline utility. Another attempt, in [4], the most recent one to our knowledge, describes yet another sophisticated reorganizing technique, based on the database clustering. Termed AutoClust, the technique mines for the closed sets, then groups the records according to the resulting attribute clusters. The AutoClust processing should to start when the average query response time drops below a user defined threshold. It is unknown whether AutoClust was put into practice.

With respect to the partitioning algorithms used in other major DBMSs, the parallel DB2 uses the (static) hash partitioning. Oracle offers both, hash and range partitioning, but over the shared disk multiprocessor architecture only. All well-known DBSs support the distributed partitioned union-all views. Only SQL Server let such views be updatable. This is the rationale for our choice. How the scalable tables may be created at other main DBSs remains an open problem.

5. Conclusion

The syntax and semantics of SD-SQL Server commands make the use of scalable tables about as simple as that of the static ones. It lets the user/application to easily take advantage of the new capabilities of our system. Through the scalable distributed partitioning, they should allow for much larger tables or for a faster response time of complex queries, or for both.

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

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

References

1. Ben-Gan, I., and Moreau, T. Advanced Transact SQL for SQL Server 2000. Apress Editors, 2000
2. Gray, J. & al. Data Mining of SDDS SkyServer Database. WDAS 2002, Paris, Carleton Scientific (publ.)
3. Gray, J. The Cost of Messages. Proceeding of Principles Of Distributed Systems, Toronto, Canada, 1989
4. Guinepain, S & Gruenwald, L. Research Issues in Automatic Database Clustering. ACM-SIGMOD, March 2005
5. Lejeune, H. Technical Comparison of Oracle vs. SQL Server 2000: Focus on Performance, December 2003
6. Litwin, W., Neimat, M.-A., Schneider, D. LH*: A Scalable Distributed Data Structure. ACM-TODS, Dec. 1996
7. Litwin, W., Neimat, M.-A., Schneider, D. Linear Hashing for Distributed Files. ACM-SIGMOD International Conference on Management of Data, 1993
8. Litwin, W., Rich, T. and Schwarz, Th. Architecture for a scalable Distributed DBSs application to SQL Server 2000. 2nd Intl. Workshop on Cooperative Internet Computing (CIC 2002), August 2002, Hong Kong
9. Litwin, W & Sahri, S. Implementing SD-SQL Server: a Scalable Distributed Database System. Intl. Workshop on Distributed Data and Structures, WDAS 2004, Lausanne, Carleton Scientific (publ.), to app
10. Microsoft SQL Server 2000: SQL Server Books Online
11. Rao, J., Zhang, C., Lohman, G. and Megiddo, N. Automating Physical Database Design in a Parallel Database, ACM SIGMOD '2002 June 4-6, USA
12. Salzberg, B & Lomet, D. Special Issue on Online Reorganization, Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, 1996
13. Özsu, T & Valduriez, P. Principles of Distributed Database Systems, 2nd edition, Prentice Hall, 1999.
14. Litwin, W., Sahri, S., Schwarz, Th. SD-SQL Server: a Scalable Distributed Database System. CERIA Research Report 2005-12-13, December 2005.
15. Litwin, W., Sahri, S., Schwarz, Th. Scalable Command Processing in SD-SQL Server: a Scalable Distributed Database System. 7th Intl. Workshop on Distributed Data and Structures (WDAS-7) Santa Clara, CA, 2006.

An Overview of a Scalable Distributed Database System SD-SQL Server

Witold Litwin[1], Soror Sahri[1], and Thomas Schwarz[2]

[1] CERIA, Paris-Dauphine University
75016 Paris, France
witold.litwin@dauphine.fr, Soror.Sahri@Dauphine.fr
[2] Santa Clara University,
California, USA
tjschwarz@scu.edu

Abstract. We present a scalable distributed database system called SD-SQL Server. Its original feature is dynamic and transparent repartitioning of growing tables, avoiding the cumbersome manual repartitioning that characterize current technology. SD-SQL Server re-partitions a table when an insert overflows existing segments. With the comfort of a single node SQL Server user, the SD-SQL Server user has larger tables or gets a faster response time through the dynamic parallelism. We present the architecture of our system, its implementation and the performance analysis. We show that the overhead of our scalable table management should be typically negligible.

1 Introduction

Databases (DBs) are now often huge and grow fast. Large tables are typically hash or range partitioned into segments stored at different storage sites. Current Data Base Management Systems (DBSs) such as SQL Server, Oracle or DB2, provide only static partitioning [1,5,11]. Growing tables have to overflow their storage space after some time. The database administrator (DBA) has then to manually redistribute the database. This operation is cumbersome, users need a more automatic solution, [1].

This situation is similar to that for file users forty years ago in the centralized environment. Efficient management of distributed data presents specific needs. Scalable Distributed Data Structures (SDDSs) address these needs for files, [6,7]. An SDDS scales transparently for an application through distributed splits of its buckets, whether hash, range or k-d based. In [7], we derived the concept of a Scalable Distributed DBS (SD-DBS) for databases. The SD-DBS architecture supports *scalable (distributed relational) tables*. As an SDDS, a scalable table accommodates its growth through the splits of its overflowing segments, located at SD-DBS *storage* nodes. Also like in an SDDS, we can use hashing, range partitioning, or k-d-trees. The storage nodes can be P2P or grid DBMS nodes. The users or the application, manipulate the scalable tables from a *client* node that is not a storage node, or from a *peer* node that is both, again as in an SDDS. The client accesses a scalable table only through its specific view, called the (*client*) *image*. It is a particular updateable distributed partitioned union view stored at a client. The application manipulates

D. Bell and J. Hong (Eds.): BNCOD 2006, LNCS 4042, pp. 16 – 35, 2006.

scalable tables using images directly, or their *scalable* views. These views involve scalable tables through the references to the images.

Every image, one per client, hides the partitioning of the scalable table and dynamically adjusts to its evolution. The images of the same scalable table may differ among the clients and from the actual partitioning. The image adjustment is lazy. It occurs only when a query to the scalable table finds an outdated image. To prove the feasibility of an SD-DBS, we have built a prototype called SD-SQL Server. The system generalizes the basic SQL Server capabilities to the scalable tables. It runs on a collection of SQL Server linked nodes. For every standard SQL command under SQL Server, there is an SD-SQL Server command for a similar action on scalable tables or views. There are also commands specific to SD-SQL Server client image or node management.

Below we present the architecture and the implementation of our prototype as it stands in its 2005 version. Related papers [14, 15] discuss the user interface. Scalable table processing creates an overhead and our design challenge was to minimize it. The performance analysis proved this overhead negligible for practical purpose. The present capabilities of SQL Server allow a scalable table to reach 250 segments at least. This should suffice for scalable tables reaching very many terabytes. SD-SQL Server is the first system with the discussed capabilities, to the best of our knowledge. Our results pave the way towards the use of the scalable tables as the basic DBMS technology.

Below, Section 2 presents the SD-SQL Server architecture. Section 3 recalls the basics of the user interface. Section 4 discusses our implementation. Section 5 shows experimental performance analysis. Section 6 discusses the related work. Section 7 concludes the presentation.

2 SD-SQL Server Architecture

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

We call an SD-SQL Server database in short a *node database* (NDB). NDBs at different nodes may share a (proper) database name. Such nodes form an SD-SQL Server *scalable (distributed) database* (SDB). The common name is the *SDB name.* One of NDBs in an SDB is *primary*. It carries the meta-data registering the current NDBs, their nodes at least. SD-SQL Server provides the commands for scaling up or down an SDB, by adding or dropping NDBs. For an SDB, a node without its NDB is

(an SD-SQL Server) *spare* (node). A spare for an SDB may already carry an NDB of another SDB. Fig 1 shows an SDB, but does not show spares.

Each manager takes care of the SD-SQL Server specific operations, the user/application command interface especially. The procedures constituting the manager of an NDB are themselves kept in the NDB. They apply internally various SQL Server commands. The SQL Servers at each node entirely handle the inter-node communication and the distributed execution of SQL queries. In this sense, each SD-SQL Server runs at the top of its linked SQL Server, without any specific internal changes of the latter.

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

Internally, each client stores the images, the local views and perhaps *static* tables. These are tables created using the SQL Server *CREATE TABLE* command (only). It also contains some SD-SQL Server meta-tables constituting the catalog **C** at Fig 1. The catalog registers the client images, i.e., the images created at the client.

When a scalable query comes in, the client checks whether it actually involves a scalable table. If so, the query must address the local image of the table. It can do it directly through the image name, or through a scalable view. The client searches therefore for the images that the query invokes. For every image, it checks whether it conforms to the actual partitioning of its table, i.e., unions all the existing segments. We recall that a client view may be outdated. The client uses **C**, as well as some server meta-tables pointed to by **C** that define the actual partitioning. The manager dynamically adjusts any outdated image. In particular, it changes internally the scheme of the underlying SQL Server partitioned and distributed view, representing the image to the SQL Server. The manager executes the query, when all the images it uses prove up to date.

A *server* NDB stores the segments of scalable tables. Every segment at a server belongs to a different table. At each server, a segment is internally an SQL Server table with specific properties. First, SD-SQL Server refers to in the specific catalogue in each server NDB, called **S** in the figure. The meta-data in **S** identify the scalable table each segment belongs to. They indicate also the segment size. 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, i.e., the first segment 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 in 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 that the remaining (lower) tuples fit the size of the splitting segment. For the migrating tuples, the server creates remotely one or more new segments that are each half-full (notice the difference to a B-tree split creating a single new segment). Furthermore, every segment in a multi-segment scalable table carries an SQL Server *check constraint*. Each constraint defines the partition

(primary) key range of the segment. The ranges partition the key space of the table. These conditions let the SQL Server distributed partitioned view to be updateable, by the inserts and deletions in particular. This is a necessary and sufficient condition for a scalable table under SD-SQL Server to be updateable as well.

Finally a *peer* NDB is both a client and a server NDB. Its node DB carries all the SD-SQL Server meta-tables. It may carry both the client images and the segments. The meta-tables at a peer node form logically the catalog termed **P** at the figure. This one is operationally, the union of **C** and **S** catalogs.

Every SD-SQL Server node is *client*, *server* or *peer* node. The peer accepts every type of NDB. The client nodes only carry client NDBs & server nodes accept server NDBs only. Only a server or peer node can be the primary one or may carry a primary NDB. 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, provided it is a peer. It interfaces the applications. The NDBs on all the other nodes until *Di* are server NDBs. They carry only the segments and do not interface (directly) any applications. The NDB at *D2* could be here the primary NDB. Nodes *D2*…*Di* could be peers or (only) servers. Finally, the NDB at *Di*+1 is a peer, providing all the capabilities. Its node has to be a peer node.

The NDBs carry a scalable table termed *T*. The table has a scalable index *I*. We suppose that *D*1 carries the *primary* image of *T*, named *T* at the figure. This image name is also as the SQL Server view name implementing the image in the NDB. SD-SQL Server creates the primary image at the node requesting the creation of a scalable table, while creating the table. Here, the primary segment of table *T* is supposed at *D*2. Initially, the primary image included only this segment. It has evolved since, following the expansion of the table at new nodes, and now is the distributed partitioned union-all view of *T* segments at servers *D*2…*Di*. We symbolize this image with the dotted line running from image *T* till the segment at *Di*. Peer *Di*+1 carries a *secondary* image of table *T*. Such an image interfaces the application using *T* on a node other than the table creation one. This image, named *D1_T*, for reasons we discuss below, differs from the primary image. It only includes the primary segment. We symbolize it with the dotted line towards *D*2 only. Both images are outdated. Indeed, server *Di* just split its segment and created a new segment of *T* on *Di*+1. The arrow at *Di* and that towards *Di*+1 represent this split. As the result of the split, server *Di* updated the meta-data on the actual partitioning of *T* at server *D*2 (the dotted arrow from *Di* to *D*2). The split has also created the new segment of the scalable index *I*. None of the two images refers as yet to the new segment. Each will be actualized only once it gets a scalable query to *T*. At the figure, they are getting such queries, issued using respectively the SD-SQL Server *sd_select* and *sd_insert* commands. We discuss the SD-SQL Server command interface in the next sections.

Notice finally that in the figure that the segments of *T* are all named *_D1_T*. This represents the couple (creator node, table name). It is the proper name of the segment as an SQL Server table in its NDB. Similarly for the secondary image name, except for the initial '_'. The image name is the local SQL Server view name. We explain these naming rules in Section 4.1.

3 Command Interface

3.1 Overview

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

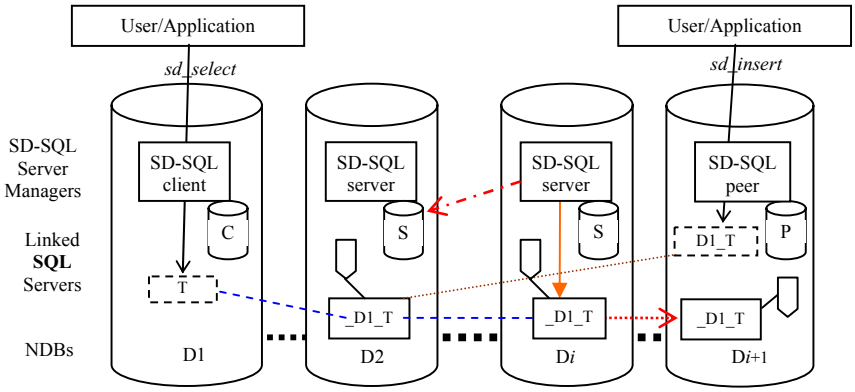

Fig. 1. SD-SQL Server Architecture

Details of all the SD-SQL Server commands are in [14, 15]. The rule for an SD-SQL Server command performing an SQL operation is to use the SQL command name (verb) prefixed with *'sd_'* and with all the blanks replaced with '_'. Thus, e.g., SQL *SELECT* became SD-SQL *sd_select*, while SQL *CREATE TABLE* became *sd_create_table*. The standard SQL clauses, with perhaps additional parameters follow the verb, specified as usual for SQL. The whole specification is however within additional quotes ' '. The rationale is that SD-SQL Server commands are implemented as SQL Server stored procedures. The clauses pass to SQL Server as the parameters of a stored procedure and the quotes around the parameter list are mandatory.

The operational capabilities of SD-SQL Server should suffice for many applications. The *SELECT* statement in a scalable query supports the SQL Server allowed selections, restrictions, joins, sub-queries, aggregations, aliases…etc. It also allows for the INTO clause that can create a scalable table. However, 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 *sd_insert* command over a scalable table lets for any insert accepted by SQL Server for a distributed partitioned view. This can be a new tuple insert, as well as a multi-tuple insert through a *SELECT* expression, including the *INTO* clause. The *sd_update* and *sd_delete* commands offer similar capabilities. In contrast, some of SQL Server specific SQL clauses are not supported at present by the scalable commands; for instance, the *CASE OF* clause.

We recall the SD-SQL Server command interface by the motivating example. modelled upon our benchmark application that is SkyServer DB, [2].

3.2 Motivating Example

A script file creates the first ever (primary) SD-SQL Server (scalable) node at a collection of linked SQL Server nodes. We can create the primary node as a peer or a server, but not as a client. After that, we can create additional nodes using the *sd_create_node* command. Here, the script has created the primary SD-SQL Server node at SQL Server linked node at our *Dell1* machine. We could set up this node as server or peer, we made the latter choice. The following commands issued at *Dell1* create then further nodes, alter *Dell3* type to peer, and finally create our *SkyServer* SDB at *Dell1*:

sd_create_node 'Dell2' / Server by default */;*

sd_create_node 'Dell3, 'client';

sd_create_node 'Ceria1','peer'

sd_alter_ node 'Dell3', 'ADD server';

sd_create_scalable_database 'SkyServer, 'Dell1'

Our SDB has now one NDB termed *SkyServer*. This NDB is the primary one of the SDB and is a server NDB. To query , *SkyServer* one needs at least one client or peer NDB. We therefore append a client NDB at *Dell3* client node:

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

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

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

We did not provide the complete syntax, using '…' to denote the rest of the scheme beyond the key attribute. The *objid* attribute is the partition key implicitly, since it is here the only key attribute. The user creates furthermore a scalable table *Neighbors*, modelled upon the similar one in the static *Skyserver*. That table has three key

attributes. The *objid* is one of them and is the foreign key of *PhotoObj*. For this reason, the user wishes it to be the partition key. The segment capacity should now be 500 tuples. Accordingly, the user issues the command:

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

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

sd_alter_table 'PhotoObj ADD t INT, 1000

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

sd_create_index 'run_index ON Photoobj (run)'

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

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

sd_create_image 'Ceria1', 'PhotoObj'

The image internal name is *SD.Dell3_Photoobj*, as we discuss in Section 0 and [15]. The *Ceria1* user who wishes a different local name such as *PhotoObj* uses the SQL Server *CREATE VIEW* command. Once the *Ceria1* user does not need its image anymore, s/he may remove it through the command:

sd_drop_image 'SD.Dell3_Photoobj'

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

USE Skyserver /* SQL Server command */

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

sd_select ' FROM PhotoObj' ;*

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

The first query loads into our *PhotoObj* scalable table tuples from some other *PhotoObj* table or view created in some *Skyserver* DB at node *Ceria5*. This DB could be a static, i.e., SQL Server only, DB. It could alternatively be an NDB of "our" *Skyserver* DB. The second query creates scalable table *PhotoObj1* with segment size of 500 and copies there 5000 tuples from *PhotoObj*, having the smallest values of *objid*. See [15] for examples of other scalable commands.

4 Command Processing

We now present the basics of SD_SQL Server command processing. We start with the naming rules and the meta-tables. Next, we discuss the scalable table evolution.

We follow up with the image processing. For more on the command processing see [14].

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. Each NDB has a dedicated user account 'SD' for SD-SQL Server itself. 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 public stored procedures under the same names. An SQL Server may call them from any user account.

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 it[1]. 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.

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.

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.

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*. In Fig 1, e.g., the proper name of secondary image denoted *T* (2) would actually be *D1_T*.

We recall that since SD-SQL Server commands are public stored procedures, SQL Server automatically prefixes all the proper names of SD-SQL public objects with *dbo.* in every NDB, to prevent a name conflict with any other owner within NDB. The rules avoid name conflicts between the SD-SQL Server private application objects and SD-SQL system objects, as well as between SD-SQL Server system objects themselves. See [14] for more examples of the naming rules.

[1] In the current version of the prototype.

4.2 Meta-tables

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.

- *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.
- *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 sizes.
- *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:

- 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.
- 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*-catalogue, at a peer, is simply the union of *C*-catalog and *S*-catalog. In addition, each NDB has two tables:

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

- *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*'.
- *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'*.
- *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 Evolution

A scalable table *T* grows by getting new segments, and shrinks by dropping some. The dynamic *splitting* of overflowing segments performs the former. The *merge* of under-loaded segments may perform the latter. There seems to be little practical interest for merges, just as implementations of B-tree merges are rare. We did not consider them for the current prototype. We only present the splitting now. The operation aims at several goals. We first enumerate them, then we discuss the processing:

1. The split aims at removing the overflow from the splitting segment by migrating some of its tuples into one or several new segment(s). The segment should possibly stay at least half full. The new segments should end up half full. The overall result is then at least the typical "good" load factor of 69 %.
2. Splitting should not delay the commit of the insert triggering it. The insert could timeout otherwise. Through the performance measures in Section 0, we expect the split to be often much longer than a insert.
3. The allocation of nodes to the new segments aims at the random node load balancing among the clients and /or peers. However, the splitting algorithm also allocates the same nodes to the successive segments of different scalable tables of the same client. The policy aims at faster query execution, as the queries tend to address the tables of the same client.
4. The concurrent execution of the split and of the scalable queries should be serializable. A concurrent scalable query to the tuples in an overflowing segment, should either access them before any migrate, or only when the split is over.

We now show how SD-SQL Server achieves these 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, denoted *b* below, and defined in the *sd_create_table* command. The overflow may consist of arbitrarily many tuples, brought by a single *sd_insert* command with the *SELECT* expression (unlike in a record-at-the-time operations, e.g., as in a B-tree). A single *sd_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 of a *multi-segment* (*insert*) split. The bulk inserts correspond to the *sd_insert* with the *SELECT* expression.

In every *sd_insert* case, the *AFTER* trigger at every segment getting tuples tests for overflow, [8]. The positive result leads to the split of the segment, according to the following *segment partitioning scheme*. We first discuss the basic case of the partition key attribute being the (single-attribute) primary key. We show next the case of the multi-attribute key, where the partition key may thus present duplicates. We recall that the partition key under SQL Server must be a (single) key attribute. In every case, the scheme adds $N \geq 1$ segments to the table, with N as follows.

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 partition key. Each server aims at cutting its P, starting from the high-end, into successive portions $P_1 \ldots P_N$ consisting each of *INT (b/2)* tuples. Each portion goes to a different server to become a possibly half-full new segment of T. The number N is the minimal one leaving at most b tuples in the splitting segment. To fulfil goal (1) above, we thus always have $N = 1$ for a tuple insert, and the usual even partitioning (for the partition key without duplicates). A single segment bulk insert basically leads to $N \geq 1$ half-full new segments. The splitting one ends up between half-full and full. We have in both cases:

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

The same scheme applies to every splitting segment for a multiple bucket insert. If the partition key presents the duplicates, the result of the calculus may differ a little in practice, but arbitrarily in theory. The calculus of each P_i incorporates into it all the duplicates of the lowest key value, if there is any. The new segment may start more than half-full accordingly, even overflowing in an unlikely bad case. The presence of duplicates may in this way decrease N. It may theoretically even happen that all the partition key values amount to a single duplicate. We test this situation in which case no migration occurs. The split waits till different key values come in. The whole duplicate management is potentially subject of future work optimizing the P_i calculus.

The *AFTER* trigger only tests for overflow, to respond to goal (2). If necessary, it launches the actual splitting process as an asynchronous job called *splitter* (for performance reasons, see above). 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 repeatable read isolation level. SQL Server uses then the shared and exclusive tuple locks according to the basic 2PL protocol. 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 are no (more) scalable queries or other commands in progress involving the segment. Otherwise it would encounter at least a shared lock at the tuple. SQL Server would then block the split until the end of the concurrent operation. In unlikely cases, a deadlock may result. The overall interaction suffices to provide the serializability of every command and of a split, [14]. If the splitter does not find the tuple in *Primary*, it terminates. 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 can another splitter lock it. It should first lock 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* NDBs without any segment of *T* as yet. It searches for such nodes through the query to NDB meta-table, requesting every NDB in the SDB which is a server or peer and not yet in *RP* for *T*. Let $M \geq 0$ be the number of NDBs 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. To satisfy goal (3) above, the selection is nevertheless driven, at the base of the randomness generation, by *T* 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 that the SDB has not enough of NDBs 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. It uses the meta-tables *Nodes* and *SDB* in the MDB and *NDB* at the primary SDB node. If it does not succeeds the splitting halts with a message to the administrator. This one may choose to add nodes using *sd_create_node* command. Otherwise, the splitter updates the *NDB* meta-table, asks SQL Server to create the new NDBs (by issuing the *sd_create_node_database command*) 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 $p = \text{INT}(b/2)$, 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 $i = 1 \ldots N$ as follows.

It starts with the SQL Server query in the form of :

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

The option "with ties" takes care of the duplicates[2]. Next, the splitter finds the partition key *c* of *S* using the SQL Server system tables and alters S_i scheme accordingly. To find *c*, it joins SQL Server system tables *information_schema.Tables* and *information_schema.TABLE_CONSTRAINTS* on the *TABLE_SCHEMA*, *CONSTRAINT_SCHEMA* and *CONSTRAINT_NAME* columns. It also determines the indexes on *S* using the SQL Server stored procedure *sp_helpindex*. It creates then the same indexes on S_i using the SQL Server *create index* statements. Finally, it creates the check constraint on S_i as we describe soon. Once all this done, it registers the new

[2] Actually, it first performs the test whether the split can occur at all, as we discussed, using the similar query with count(*).

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. As the last step, it deletes from *S* the copied tuples. It then moves to the processing of next P_i if any remains. Once the loop finishes, the splitter commits which makes SQL Server to release all the locks.

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. SQL Server needs for updates through a distributed partitioned view, a necessity for SD-SQL Server. Because of our duplicates management, we have: $C(S) = \{ c : l \leq c < h \}$. 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.

4.4 Image Processing

4.4.1 Checking and Adjustment

A scalable query 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 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 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 newly found segments.

The view *dbo.T* should remain the correct image until the scalable query finishes exploring *T*. This implies that no split modifies partitioning of *T* since the client requested the segment node names in *PrimNd.D.SD.RP*, until *Q* finishes mani-

pulating *T*. Giving our splitting scheme, this means in practice that no split starts in the meantime the deletion phase on any *T* segment. To ensure this, the manager requests from SQL Server to process every *Q* as a distributed transaction at the repeatable read isolation level. We recall that the splitter uses the same level. 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 splitting segment. According to its 2PL protocol, SQL Server would then block any *T* split until *Q* terminates. Vice versa, *Q* in progress would not finish, or perhaps even start the S_A count and the *T* segment names retrieval until any split in progress ends. *Q* will take then the newly added *T* segments into account as well. In both cases, the query and split executions remain serializable.

Finally, we use a *lazy schema validation* option for our linked SQL Servers, [1, 10]. 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 a discrepancy between the compiled query based on the view, *dbo.T* in our case, and some alterations of schema *T* by SD-SQL Server user since, requiring *Q* recompilation on the fly.

Example. Consider query *Q* to *SkyServer* peer NDB at the *Ceria1*:

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 *Ceria1* manager processing *Q* finds *Size* = 1 in the tuple with of *Name* = *'PhotoObj'* retrieved from its *Image* table. The client finds also *Ceria1* in the *PrimeNd* of the tuple. Suppose further that *PhotoObj* has in fact also two secondary segments at *Dell1 and Dell2*. The counting of the tuples with *Table* = '*PhotoObj'* and *CreatNd* = *'Ceria1' in Ceria1.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 *Ceria1.SkyServer.SD.RP* the *SgmNd* values in the previously counted tuples. It gets {*Dell1*, *Dell2, Ceria1*}. It generates the actual segment names as '*_Dell1_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.

4.4.2 Binding

A *scalable* query consists of a query command to an SD-SQL Server client (peer) followed by a (scalable) *query expression.* We recall that these commands are *sd_select*, *sd_insert*, *sd_update* and *sd_delete*. Every scalable query, unlike a static one, starts with an *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 the

processing of scalable queries under SD-SQL Server. We only discuss image binding and refer to more documentation for each command to [14].

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, but then is 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:

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.

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 themselves. A multi-database base view does not have direct remote dependants in *sysobjects*. That is why we at present do not allow scalable multi-database views. The client searches, recursively if necessary, 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*.

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 repeatable read level 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.

5 Performance Analysis

To validate the SD-SQL Server architecture, we evaluated its scalability and efficiency over some Skyserver DB data [2]. Our hardware consisted of 1.8 GHz P4 PCs with either 785 MB or 1 GB of RAM, linked by a 1 Gbs Ethernet. We used the SQL Profiler to take measurements. We measured split times and query response times under various conditions. The split time was from about 2.5 s for a 2-split of 1000-tuple *PhotoObj* segment, up to 150 seconds for a 5-split of a 160 000 tuple segment. Presence of indexes naturally increased this time, up to almost 200 s for the 5-split of a 160 000 tuple segment with three indexes.

The query measures included the overhead of the image checking alone, of image adjustment and of image binding for various queries, [14]. Here, we discuss two queries:

(Q1) *sd_select 'COUNT (*) FROM PhotoObj'*

(Q2) *sd_select 'top 10000 x.objid from photoobj x, photoobj y where x.obj=y.obj and x.objid>y.objid*

Query (Q1) represents the rather fast queries, query (Q2) the complex ones, because of its double join and larger result set. The measures of (Q1) concern the execution time under various conditions, Fig. 2. The table had two segments of various sizes. We measured (Q1) at SD-SQL Server peer, where the *Image* and *RP* tables are at the same node, and at a client where they are at different nodes. We executed (Q1) with image checking (IC) only, and with the image adjustment (IA). We finally compared these times to those of executing (Q1) as an SQL Server, i.e., without even IC. As the curves show, IC overhead appeared always negligible. (Q1) always executed then under 300 msec. In contrast the IA overhead is relatively costly. On a peer node it is about 0.5s, leading to the response time of 0.8sec. On a client node, it increases to about 1s, leading to (Q1) response time of 1.5s. The difference is obviously due to the remote accesses to *RP* table. Notice that IA time is constant, as one could expect. We used the LSV option, but the results were about the same without it.

The measures of (Q2) representing basically longer to process typical queries than (Q1), involved *PhotoObj* with almost 160 K tuples spreading over two, three or four segments. The query execution time with IC only (or even without, directly by SQL Server) is now between 10 – 12 s. The IA overhead is about or little over 1 s. It grows a little since SQL Server ALTER VIEW operation has more remote segments to access for the check constraint update (even if it remains in fact the same, which indicates a clear path for some optimizing of this operation under SQL Server). IA overhead becomes relatively negligible, about 10 % of the query cost. We recall that IA should be in any case a rare operation.

Finally, we have measured again (Q1) on our *PhotoObj* scalable table as it grows under inserts. It had successively 2, 3, 4 and 5 segments, generated each by a 2-split. The query counted at every segment. The segment capacity was 30K tuples. We aimed at the comparison of the response time for an SD-SQL Server user and for the one of SQL Server. We supposed that the latter (i) does not 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 to the same comfort as that of an SD-SQL Server

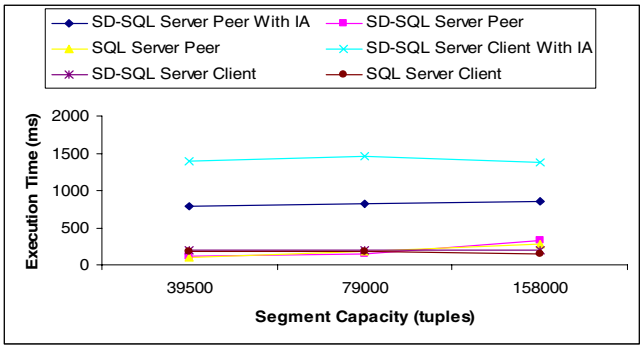

Fig. 2. Query (Q1) execution performance

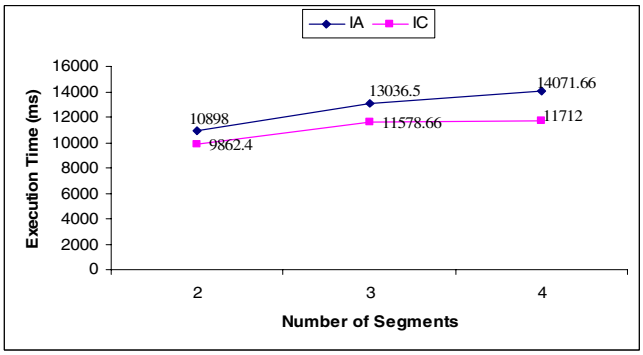

Fig. 3. Query (Q2) with image checking only (IC) and with image adjustment (IA)

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 (Q1) for a static table. For SD-SQL Server, we measured (Q1) 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.

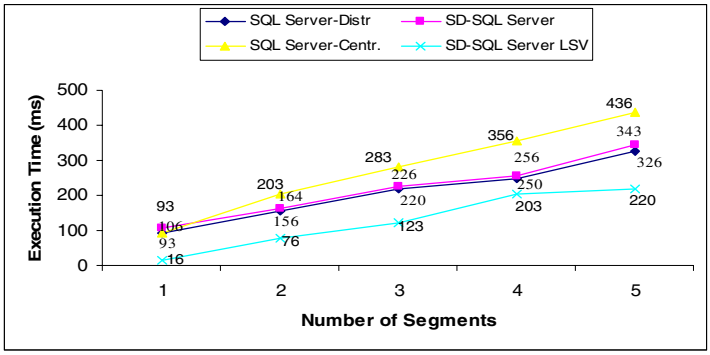

Fig. 4. Query (Q1) execution on SQL Server and SD-SQL Server (Client/Peer)

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.

6 Related Works

Parallel and distributed database partitioning has been studied for many years, [13]. It naturally triggered the work on the reorganizing of the partitioning, with notable results as early as in 1996, [12]. The common goal was global reorganization, unlike for our system.

The editors of [12] contributed themselves with two on-line reorganization methods, called respectively *new-space* and *in-place* reorganization. The former method created a new disk structure, and switches the processing to it. The latter approach balanced the data among existing disk pages as long as there was room for the data. Among the other contributors to [12], one 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 ascertain whether it was ever implemented.

A more recent proposal of efficient global reorganizing strategy is in [11]. One proposes there an automatic advisor, balancing the overall database load through the periodic reorganizing. The advisor is intended as a DB2 offline utility. Another attempt, in [4], the most recent one to our knowledge, describes yet another sophisticated reorganizing technique, based on database clustering. Called AutoClust, the technique mines for closed sets, then groups the records according to the resulting attribute clusters. AutoClust processing should start when the average query response time drops below a user defined threshold. We do not know whether AutoClust was put into practice.

With respect to the partitioning algorithms used in other major DBMSs, parallel DB2 uses (static) hash partitioning. Oracle offers both hash and range partitioning, but over the shared disk multiprocessor architecture only. Only SQL Server offers the updatable distributed partitioned views. This was the major rationale for our choice, since scalable tables have to be updatable. How the scalable tables may be created at other systems remains thus an open research problem.

7 Conclusion

The proposed syntax and semantics of SD-SQL Server commands make the use of scalable tables about as simple as that of the static ones. It lets the user/application to easily take advantage of the new capabilities of our system. Through the scalable distributed partitioning, they should allow for much larger tables or for a faster response time of complex queries, or for both.

The current design of our interface is geared towards a "proof of concept" prototype. It is naturally simpler than a full-scale system. Further work should expand it. Among the challenges at the processing level, notice that there is no user account management for the scalable tables at present. Concurrent query processing could be perhaps made faster during splitting. We tried to limit the use of exclusive locks to as little as necessary for 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. Our design could apply almost as is to other DBSs, once they offer the updatable distributed partitioned (union-all) views. Next, 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 the original SkyServer database and for advising this work from its start. G. Graefe (Microsoft) provided the helpful information on SQL Server linked servers' capabilities. MS Research partly funded this work, relaying the support of CEE project ICONS. Current support comes from CEE Project EGov.

References

1. Ben-Gan, I., and Moreau, T. Advanced Transact SQL for SQL Server 2000. Apress Editors, 2000
2. Gray, J. & al. Data Mining of SDDS SkyServer Database. WDAS 2002, Paris, Carleton Scientific (publ.)
3. Gray, J. The Cost of Messages. Proceeding of Principles Of Distributed Systems, Toronto, Canada, 1989
4. Guinepain, S & Gruenwald, L. Research Issues in Automatic Database Clustering. ACM-SIGMOD, March 2005
5. Lejeune, H. Technical Comparison of Oracle vs. SQL Server 2000: Focus on Performance, December 2003

6. Litwin, W., Neimat, M.-A., Schneider, D. LH*: A Scalable Distributed Data Structure. ACM-TODS, Dec. 1996
7. Litwin, W., Neimat, M.-A., Schneider, D. Linear Hashing for Distributed Files. ACM-SIGMOD International Conference on Management of Data, 1993
8. Litwin, W., Rich, T. and Schwarz, Th. Architecture for a scalable Distributed DBSs application to SQL Server 2000. 2nd Intl. Workshop on Cooperative Internet Computing (CIC 2002), August 2002, Hong Kong
9. Litwin, W & Sahri, S. Implementing SD-SQL Server: a Scalable Distributed Database System. Intl. Workshop on Distributed Data and Structures, WDAS 2004, Lausanne, Carleton Scientific (publ.), to app
10. Microsoft SQL Server 2000: SQL Server Books Online
11. Rao, J., Zhang, C., Lohman, G. and Megiddo, N. Automating Physical Database Design inParallel Database, ACM SIGMOD '2002 June 4-6, USA
12. Salzberg, B & Lomet, D. Special Issue on Online Reorganization, Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, 1996
13. Özsu, T & Valduriez, P. Principles of Distributed Database Systems, 2nd edition, Prentice Hall, 1999.
14. Litwin, W., Sahri, S., Schwarz, Th. SD-SQL Server: a Scalable Distributed Database System. CERIA Research Report 2005-12-13, December 2005.
15. Litwin, W., Sahri, S., Schwarz, Th. Architecture and Interface of Scalable Distributed Database System SD-SQL Server. The Intl. Ass. of Science and Technology for Development Conf. on Databases and Applications, IASTED-DBA 2006, to appear.