ADMINISTRATION

EXPLOITATION

OPTIMISER SQL SERVER

Dimensionnement, supervision, performances du moteur et du code SQL

Rudi Bruchez

SUPPLÉMENTS en ligne

DUNOD

OPTIMISER SQL SERVER

Dimensionnement, supervision, performances du moteur et du code SQL

OPTIMISER SQL SERVER

Dimensionnement, supervision, performances du moteur et du code SQL

Rudy Bruchez
Consultant et formateur spécialisé sur SQL Server
MVP sur SQL Server
Certifié Microsoft MCDBA, MCT et MCITP

DUNOD

Illustration de couverture : Parc du Torres del paine © piccaya-Fotolia.com

ISBN 978-2-10-053750-1

Table des matières

Avant-propos

Avez-vous bonne mémoire ? Lorsque, dans une conversation, vous devez citer des chiffres, une anecdote lue dans le journal, prononcer l'adjectif qui décrit précisément votre pensée, y parvenez-vous sans hésiter ? Si vous y arrivez la plupart du temps de manière satisfaisante, n'y a-t-il pas, parfois, des mots qui vous échappent, des informations qui vous restent sur le bout de la langue, jusqu'à ce que vous vous en souveniez subitement, lorsqu'il est bien trop tard ?

Parfois, votre mémoire est plus qu'un outil de conversation : lorsque vous devez retrouver votre chemin dans les rues d'une métropole, ou vous souvenir s'il faut couper le fil bleu ou le fil rouge pour désamorcer une bombe prête à exploser… Même si vous n'êtes pas souvent confronté – espérons-le – à ce dernier cas, vous comprenez que parfois, accéder à vos souvenirs rapidement, de façon fluide, sans à-coups, est une nécessité.

Un système de gestion de bases de données relationnelles (SGBDR) est comme une mémoire : il contient des données importantes, sinon vitales, pour l'entreprise, et la capacité donnée aux acteurs de cette entreprise de pouvoir y accéder efficacement, rapidement, avec des temps de réponse consistants, est essentielle.

Le problème des performances des bases de données s'est posé dès leur apparition. L'abstraction bâtie entre le stockage physique des données et leur représentation logique à travers différents modèles – le modèle relationnel étant aujourd'hui prédominant – nécessite de la part des éditeurs de ces logiciels un très important travail de développement. Il faut d'abord traduire les requêtes exprimées dans des langages déclaratifs comme le SQL (*Structured Query Language*) en stratégies de recherche de données optimales, puis parvenir à stocker les informations de la façon la plus favorable à leur parcours et à leur extraction. *A priori*, rien dans la norme SQL ni dans la

théorie relationnelle ne concerne les performances. C'est une préoccupation qui ne peut venir qu'ensuite, comme on ne peint les façades que lorsque la maçonnerie est terminée.

Toutefois, les performances ne font pas partie de la façade, mais plutôt du moteur de l'ascenseur, et sont donc notre préoccupation quotidienne. Le modèle relationnel et le langage SQL sont des existants à notre disposition, notre SGBDR est installé et fonctionnel, notre responsabilité, en tant qu'administrateur de bases de données, développeur ou responsable IT, est d'en assurer le fonctionnement optimal. Pour cette tâche, il n'y a pas de norme, de commandements gravés dans la pierre, d'outils ou de modèles préexistants qui vont effectuer le travail pour nous. L'optimisation est une préoccupation constante, une tâche toujours à continuer, à modifier, à compléter. C'est aussi une forme d'artisanat, qui nécessite de bonnes connaissances du système, de la curiosité et de la patience.

Concrètement

Optimiser, cela veut dire concrétiser notre connaissance de la théorie, dans la pratique, c'est-à-dire prendre en compte la façon dont cette théorie a été mise en œuvre dans une implémentation particulière. Cet ouvrage est dédié à l'optimisation pour SQL Server 2005 et 2008, le SGBDR de Microsoft. Les deux versions ne présentant pas d'énormes différences de ce point de vue, la plupart des informations contenues dans ce livre peuvent s'appliquer indifféremment aux deux versions. Nous indiquerons dans le corps de l'ouvrage lorsque telle ou telle fonctionnalité est propre à une version.

Nous y aborderons tous les points de l'optimisation (matériel, configuration, requêtes SQL), d'un point de vue essentiellement pratique, tout en vous présentant ce qui se passe « sous le capot », connaissance indispensable pour vous permettre de mettre en œuvre efficacement les méthodes proposées. Notre objectif est de vous guider à travers les outils, concepts et pratiques propres à l'optimisation des performances, pour que vous puissiez, concrètement et de façon autonome, non seulement bâtir un système performant, mais aussi résoudre au quotidien les problèmes et lenteurs qui pourraient survenir. L'optimisation, le *tuning* sont des pratiques quotidiennes : ce livre cherche donc à vous donner tous les bons outils et les bons réflexes pour vous permettre d'être efficace jour après jour.

À qui s'adresse ce livre ?

Il s'agit donc d'un livre essentiellement pratique. Il s'adresse à toute personne – développeur, DBA (administrateur de bases de données), consultant, professionnel IT – qui doit s'assurer du fonctionnement optimal de SQL Server. Nous ne présenterons pas les bases théoriques des SGBDR, et ce livre ne contient pas d'initiation à SQL Server. L'optimisation étant un sujet plus avancé, nous partons donc du principe que vous êtes déjà familier avec les bases de données, et spécifiquement SQL Server. Pour profiter de toute la partie dédiée à l'optimisation du code SQL, vous devez connaître au moins les bases du langage SQL et de ses extensions Transact-SQL (utilisation de variables, structures de contrôle…).

L'optimisation recouvre plusieurs domaines, qui peuvent être clairement délimités dans votre entreprise, ou non. Les parties de ce livre développent les différents éléments d'optimisation à prendre en compte. Si vous vous occupez exclusivement d'administration et d'installation matérielles, la première partie, « Optimisation du système », traite du matériel et de la configuration. Elle présente l'architecture de SQL Server et les principes de l'optimisation, et jette donc des bases théoriques pour aborder les parties pratiques. Si vous êtes principalement développeur et que votre souhait est d'écrire du code SQL plus performant, la seconde partie, « Optimisation des requêtes », vous est dédiée. Elle traite non seulement du code SQL lui-même, mais des mécanismes transactionnels pouvant provoquer des attentes et des blocages, et de l'utilisation d'index pour offrir les meilleures performances possibles.

L'auteur

Rudi Bruchez travaille avec SQL Server depuis 1998. Il est aujourd'hui expert indépendant spécialisé en SQL Server, basé à Paris. Il est certifié MCDBA SQL Server 2000 et MCITP SQL Server 2005. Il est également MVP (*Most Valuable Professional*) SQL Server depuis 2008. Dans son activité de consultant et formateur, il répond quotidiennement à toutes les problématiques touchant aux parties relationnelles et décisionnelles de SQL Server, notamment en modélisation, administration et optimisation.

Vous trouverez une page dédiée aux éventuels errata de ce livre sur www.babaluga.com le site de l'auteur. Vous pouvez également consulter des articles concernant SQL Server sur rudi.developpez.com. L'auteur répond régulièrement sur le forum SQL Server de *developpez.net* et sur le *newsgroup microsoft.public.fr.sqlserver*.

Termes utilisés

Pour éviter les répétitions excessives, nous utiliserons les abréviations courantes dans le monde SQL Server, et le monde des systèmes de gestion de bases de données relationnelles (SGBDR). En voici la liste :

- **BOL** (*Books Online*) : l'aide en ligne de SQL Server.
- **SSMS** (*SQL Server Management Studio*) : l'outil client d'administration et de développement de SQL Server.
- **RTM** (*Release to manufacturing*) : version stable, en première livraison, avant la sortie de *service packs*.
- **SP** (*Service Pack*) : mise à jour importante d'une version, comportant des correctifs et des améliorations.
- **VLDB** (*Very Large DataBase*) : base de données de très grande volumétrie.
- **CTP** (*Common Technology Preview*) : versions de pré-sortie de SQL Server, livrées bien avant la sortie officielle du produit, et contenant uniquement les fonctionnalités implémentées complètement à ce moment.

- **OLTP** (*Online Transactional Processing*) : une utilisation particulière des données, principalement transactionnelle : écritures constantes, lectures de petits volumes. Ce sont les bases de données opérationnelles traditionnelles.
- **OLAP** (*Online Analytical Processing*) : une utilisation d'un entrepôt de données dans un but d'analyse. Ce qu'on nomme parfois l'utilisation décisionnelle, ou la *Business Intelligence*. Les bases de données OLAP sont utilisées principalement en lecture, pour des extractions massives.
- **MSDN** (*Microsoft Developer Network*) : documentation et articles en ligne sur Internet ou sur CD-ROM/DVD, représentant une base de connaissance très fournie sur les produits Microsoft, à destination principalement des développeurs. Accès libre sur http://msdn.microsoft.com/
- **Drapeau de trace** (*Trace flags*) : *switches* numériques, qui permettent de modifier le comportement de SQL Server. On les utilise comme paramètres au lancement du service, ou on les active/désactive à chaud, à l'aide des commandes `DBCC TRACEON` et `DBCC TRACEOFF`.

Exemples de code

Le code contenu dans cet ouvrage est disponible sur la page dédiée à l'ouvrage sur le site des éditions Dunod, *www.dunod.com*, et sur le site de l'auteur, *www.babaluga.com*

Il est fondé sur la base de données d'exemple de Microsoft, nommée AdventureWorks. Elle représente une entreprise fictive du même nom qui vend des bicyclettes. Vous pouvez télécharger AdventureWorks pour SQL Server 2005 ou SQL Server 2008 à l'adresse :
http://www.codeplex.com/MSFTDBProdSamples.

Nous utiliserons souvent en exemple la table `Person.Contact`, elle contient 19 972 lignes de contacts, avec des colonnes simples comme `FirstName`, `LastName` et `EmailAddress`.

Utilisation de l'anglais

Nous avons fait le choix d'utiliser une version anglaise de SQL Server, principalement parce que, à l'heure où nous rédigeons ce livre, SQL Server 2008 n'est pas encore sorti en version finale, et n'existe donc qu'en anglais. De plus, certaines traductions de l'interface française sont plus troublantes qu'utiles. Nous indiquerons parfois en regard le terme français. La correspondance n'est pas difficile à faire avec une version française : les entrées de menus et la disposition dans les boîtes de dialogue sont les mêmes.

Remerciements

Merci à Frédéric BROUARD, MVP SQL Server, fondateur de la société SQL Spot, pour ses conseils et suggestions et sa relecture. Merci également à Pascale DOZ, experte indépendante, pour sa relecture et son aide. Merci à Christian ROBERT, MVP SQL Server, pour ses astuces. Et enfin, merci également à Edgar Frank CODD pour avoir tout inventé.

PREMIÈRE PARTIE

Optimisation du système

Un SGBDR comme SQL Server, ça se soigne. Vous l'installez confortablement sur une machine adaptée, vous le laissez s'épanouir, s'étendre à son aise dans un mobilier choisi, vous le nourrissez de modèles de données bien structurés, vitaminés, surtout pas trop gras, vous l'ajustez pour qu'il donne le meilleur de lui-même, enfin, vous surveillez ses signes vitaux de façon proactive, pour lui garantir une vie longue et efficace. Ce sont tous ces éléments que nous allons détailler ici.

1

Règles de base de l'optimisation

Objectif

Que veut dire optimiser ? Ce court chapitre présente les bonnes pratiques et la démarche logique qui encadrent l'optimisation d'un système informatique, il vous invite à inclure la démarche d'optimisation dans l'approche globale de votre projet de base de données, d'en faire une composante justement dosée et toujours en évolution.

1.1 ÉTAPES DE L'OPTIMISATION ?

Que pouvons-nous, et que faut-il optimiser ? Ce qui vient le plus souvent à l'esprit, est ce qui est appelé en termes génériques la configuration. Configuration matérielle, c'est-à-dire les éléments de *hardware* constituant un serveur, comme configuration logicielle, c'est-à-dire les paramètres du système d'exploitation et du SGBDR lui-même. Bien qu'ils soient importants, ces éléments ne constituent que des étapes pour assurer un fonctionnement optimal. D'autres aspects, souvent négligés, sont cruciaux, et nous nous efforcerons de vous les présenter en détail pour vous permettre de tirer le meilleur parti de vos bases de données. Il ne s'agit pas seulement de penser performance lors de l'installation du serveur, mais dès la phase de conception de la structure des données, jusqu'à l'écriture du code.

La réflexion sur les performances doit donc prendre place dès les premiers moments du projet, et durant toute son évolution : modélisation, choix du matériel, installation, organisation du stockage physique et logique (une bonne indexation des tables est naturellement déterminante), codage SQL, supervision du serveur…

Dans les faits, malheureusement, cette réflexion est trop souvent négligée, et le besoin d'optimisation émerge en bout de chaîne, lorsque le système est en place et que les temps de réponse, soudainement ou progressivement, ne donnent plus satisfaction. Parfois, l'augmentation du volume de données ou du nombre d'utilisateurs simultanés a ralenti le temps d'exécution des requêtes de façon inacceptable. Vous vous retrouvez alors en face d'un constat, et dans l'obligation urgente de remédier au problème.

Cette situation n'est pas favorable pour effectuer un travail en profondeur, et dans le bon sens, mais c'est malheureusement la situation la plus fréquente, lorsqu'une planification prenant en compte les performances n'a pas été menée depuis le début du projet. Nous présenterons dans cet ouvrage tous les outils qui permettent d'identifier la source des ralentissements et d'y remédier avec précision. Pour autant, l'urgence ne doit pas vous pousser à répondre trop rapidement à la pression, et à choisir les solutions trop évidentes, celles qui sont maintenant presque des lieux communs de l'informatique, avec en tête, la mise à jour du matériel.

Augmenter la puissance du matériel est trop souvent une première réponse facile, un pis-aller basé sur un bon sens erroné. Combien de programmeurs en C faut-il pour changer une ampoule ? Un seul suffit, bien entendu. Il ne sert à rien de changer une ampoule à plusieurs, même si la blague en rajoute cinq de plus, qui seront nécessaires six mois plus tard pour comprendre l'algorithme. De même pour SQL Server. Les gains de performance ne seront pas automatiquement obtenus par un matériel plus puissant, car bien souvent, le problème provient d'une mauvaise architecture de la base de données, d'un manque d'index, de requêtes peu optimales… toutes sources de problèmes que l'augmentation de la puissance du matériel ne pourra pallier que très faiblement.

On ne peut donc optimiser au hasard. Pour améliorer les performances, nous ne pouvons faire l'économie de la recherche des causes. Cette recherche demande des outils appropriés – qui sont fort heureusement livrés avec SQL Server – et un peu d'obstination.

Souvent, donc, l'optimisation est urgente. Une base de données servant par exemple un site web marchand, ne peut se permettre un ralentissement prolongé. Lorsque les performances chutent, les sirènes sont déclenchées et les équipes entrent en mode d'urgence. Souvent, la première solution qui vient à l'esprit est de relancer la machine, ce qui, empiriquement, améliore souvent la situation, notamment lorsque celui-ci provient d'un problème de blocage de verrous. Si cela peut, ou pas, régler rapidement le problème, il est certain que cela ne donne aucune information concrète sur la source du ralentissement. Vouloir trop rapidement résoudre le problème nuit aux performances sur le long terme, car cela empêche de mener l'enquête nécessaire à l'identification des causes, donc à la mise en place d'une solution adap-

tée, pérenne. Souvent, les problèmes non résolus à la racine s'accumulent pour provoquer une sorte de situation d'urgence permanente où toute l'attention des techniciens est portée à la résolution en temps réel des problèmes, comme des médecins urgentistes. En optimisation, il faut savoir différer l'administration du médicament, pour s'assurer de bien comprendre la cause. Il est indispensable de prendre du recul, et ne pas travailler dans la précipitation. Cette recherche de la cause se fait nécessairement à chaud, lorsque le système souffre de lenteur, tout comme un médecin ne peut identifier une maladie qu'en auscultant le patient lorsqu'il est souffrant, et pas après sa guérison. Elle peut prendre un peu de temps, mais cette période douloureuse permettra d'en éviter bien d'autres plus tard.

1.2 FAUT-IL TOUT OPTIMISER ?

Faut-il privilégier exclusivement les performances par rapport à tout autre critère ? Évidemment non. Il est important de maintenir un équilibre entre simplicité, lisibilité et performance. L'optimiseur de requête de SQL Server est un des meilleurs du marché, et accomplit en général un excellent travail. Il est souvent inutile de suroptimiser vos requêtes, spécialement quand, d'une syntaxe à l'autre, le plan de requête généré par l'optimiseur est identique. De même, l'optimisation ne doit pas devenir une obsession, car dans ce cas, le temps qui lui est dédié peut devenir exagéré. Ici comme ailleurs, la loi de Pareto s'applique : 20 % des efforts d'optimisation bien choisis permettent 80 % des gains de performance. Les 80 % restants peuvent coûter beaucoup d'effort sans générer de résultats proportionnels.

Prenons un exemple. La collation détermine pour un serveur ou une base de donnée, l'ordre de classement des colonnes de type chaîne de caractères. Le choix d'une collation a un impact sur les performances de requêtes lorsqu'elles effectuent des comparaisons ou des recherches de chaînes. Une collation sensible à la casse, ou plus forte encore – de type binaire par exemple – améliore la rapidité de ces requêtes, car la comparaison de chaîne peut s'effectuer directement sur les octets sans avoir besoin d'utiliser une table d'équivalences. C'est donc un chemin d'optimisation possible, mais le jeu en vaut-il la chandelle, en sachant les contraintes fonctionnelles que cela implique ? Dans une base de données en collation binaire, non seulement toutes les requêtes devront mentionner le nom de tous les objets (tables, colonnes…) avec la casse correcte, mais toutes les recherches de chaînes de caractères sur les colonnes dans cette collation seront sensibles à la casse. Êtes-vous prêt à supporter ce surplus de contrainte pour une optimisation de performance certes intéressante mais pas toujours vitale ? C'est un équilibre qu'il vous appartient de juger.

1.3 MAINTENANCE D'UNE BASELINE

Afin de juger correctement des performances de votre système, vous devez bien le connaître. Lorsqu'un utilisateur final de votre base de données vous signale des problèmes de performance, comment pouvez-vous savoir si les lenteurs reportées sont exceptionnelles, ou « normales » ? Vous n'avez qu'un moyen : bâtir, à l'avance une situation de référence ou **baseline**, et la maintenir au fil du temps.

Une *baseline* est simplement la collecte sur une période caractéristique des compteurs importants qui permettent de connaître le comportement d'un système en temps normal. En journalisant régulièrement ces indicateurs, vous aurez une idée claire de la constitution physique de vos serveurs, comme vous pouvez dire pour vous-même si vous êtes fatigué ou de bonne humeur, parce que vous avez appris à vous connaître au fil du temps. Il s'agit aussi bien de recueillir les données de la charge matérielle (CPU, mémoire, réseau, disques), de l'utilisation du système d'exploitation (processus consommateurs de ressources, temps privilégié et temps utilisateur…) que de SQL Server lui-même (nombre de transactions et de requêtes par seconde, comportement des caches…). Nous reviendrons sur les aspects pratiques de cette *baseline* lorsque nous aurons abordé les outils permettant de collecter ces informations.

2

Architecture de SQL Server

Objectif

Afin d'obtenir de bonnes performances, il est naturellement important de connaître le logiciel utilisé. SQL Server est un système de gestion de bases de données relationnelles (SGBDR) qui à la fois respecte les standards de ce type d'outils, et offre des implémentations particulières. Nous allons voir sur quelles briques logicielles SQL Server est bâti, ce qui permettra ensuite de mettre en place des solutions d'optimisation en toute connaissance de cause.

2.1 ARCHITECTURE GÉNÉRALE

SQL Server est un système de gestion de base de données relationnelles (SGBDR). Il partage avec ses concurrents une architecture standardisée, fruit de décennies de recherches en ingénierie logicielle, menées par des chercheurs comme Edgar F. Codd, bien sûr, Chris Date et Peter Chen pour le modèle relationnel, mais aussi Michael Stonebreaker et d'autres pour différentes méthodologies et algorithmes.

Un **SGBDR** est en général une application client-serveur : un serveur centralisé abrite le SGBDR lui-même, dont la responsabilité est d'assurer le stockage des données, et de répondre aux demandes des clients.

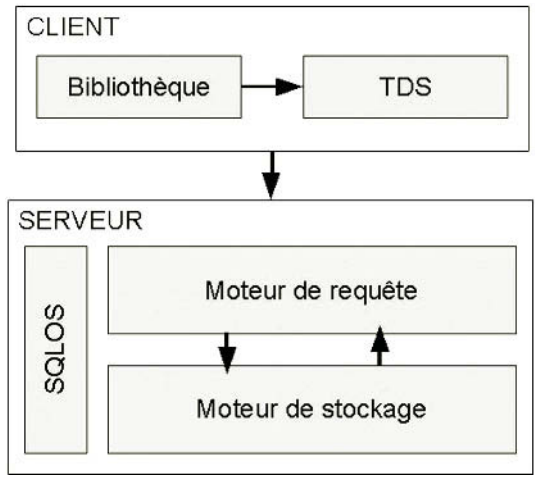

Figure 2.1 — Architecture client-serveur

Dans le schéma de la figure 2.1 extrêmement simplifié, nous avons un résumé de l'architecture client-serveur de SQL Server. Une application cliente (comme *SQL Server Management Studio* – SSMS) envoie des requêtes en SQL. L'unique manière d'obtenir des données, est de passer par le langage SQL. SQL Server n'est pas un serveur de fichiers : il stocke et protège ses données à l'aide de ses mécanismes internes, et n'offre aucun moyen d'accéder directement aux données des bases. La seule façon d'obtenir des lignes de table, est d'en faire la demande au serveur, à l'aide d'ordres SQL. Ces ordres sont transmis par une bibliothèque cliente, comme ODBC, OLEDB ou ADO.NET, dont la tâche est de faire la transition entre les langages procéduraux clients et les bibliothèques de bas niveau en offrant une couche d'abstraction. Cette bibliothèque adresse la requête à une bibliothèque réseau (*net-library*) adaptée aux protocoles réseaux utilisés (de nos jours, principalement TCP-IP), qui elle-même la transmet par paquets à travers la couche physique codée dans le protocole réseau de SQL Server, nommé **TDS** (*Tabular Data Stream*), un protocole initialement développé par Sybase[1] et que SQL Server partage avec ce SGBDR (bien que l'implémentation particulière de Microsoft soit nommée MS-TDS[2]).

Du côté serveur, SQL Server est composé de deux principaux moteurs, respectivement le **moteur relationnel**, *relational engine* (ou moteur de requête, *query engine*), et le **moteur de stockage**, *storage engine*, qui contiennent chacun différents modules. La requête provenant du client est prise en charge par le moteur relationnel, qui évalue le code SQL, vérifie les métadonnées et les privilèges, passe par une phase d'optimisation pour produire un **plan d'exécution** (*query plan*) optimisé, et gère

1. Jusqu'à la version 6.5 comprise, SQL Server était un développement partagé entre Sybase et Microsoft. La version 7 fut totalement réécrite par Microsoft.
2. Le protocole MS-TDS est maintenant décrit dans le MSDN : http://msdn.microsoft.com/en-us/library/cc448436.aspx

l'exécution de la requête. Les ordres d'extraction ou d'écriture proprement dits sont envoyés au moteur de stockage, dont la responsabilité est, outre de lire et écrire les pages de données, de gérer aussi la cohérence transactionnelle, notamment à travers le verrouillage. Toute la communication entre SQL Server et le serveur physique sur lequel il est installé, est prise en charge par une couche d'abstraction nommée **SQLOS** (OS pour *Operating System*), une sorte de système d'exploitation intégré au moteur, dont nous reparlerons plus loin dans ce chapitre.

La séparation des rôles entre moteur relationnel et moteur de stockage est une constante dans les architectures des SGBDR, certains permettent même le choix entre différents moteurs de stockage, comme MySQL.

Nomenclature SQL Server

Il est utile de spécifier ici les termes utilisés dans le cadre d'un serveur SQL Server. Il est par exemple à noter que les différents termes peuvent sembler troublants pour un développeur habitué à Oracle, car la nomenclature d'Oracle est sensiblement différente de celle de SQL Server.

Un serveur SQL est un service Windows, qui est appelé une instance. Ainsi, chaque installation de SQL Server sur une même machine crée une nouvelle instance. Une seule instance par défaut est possible par serveur : les autres instances sont appelées « instances nommées ». Il est ainsi possible d'installer des versions et des éditions différentes sur la même machine. Ensuite, une même instance peut comporter un grand nombre de bases de données, qui sont fortement isolées les unes des autres, notamment du point de vue de la sécurité. Ces bases de données contiennent des schémas, qui sont des conteneurs logiques conformes à la norme SQL, et ne sont en rien liés à des structures physiques de stockage ou à des utilisateurs de la base. Un objet tel qu'une table, une vue, une procédure stockée, etc. appartient à un et à un seul schéma.

2.2 STRUCTURES DE STOCKAGE

Le défi des SGBDR comme SQL Server est d'assurer les meilleures performances possibles lors de l'écriture et de la lecture de larges volumes de données. Pour ce faire, chaque éditeur soigne son moteur de stockage avec d'autant plus de sérieux que l'accès aux disques durs, quelles que soient leurs spécifications, est l'élément le plus lent d'un système informatique.

Pour assurer ce **stockage** de façon optimale, SQL Server utilise deux types de fichiers différents : fichiers de données et fichiers de journal de transaction.

2.2.1 Fichiers de données

Les **fichiers de données** offrent un stockage structuré organisé selon un modèle schématisé dans la figure 2.2. Un groupe de fichiers peut contenir un ou plusieurs fichiers, qui eux-mêmes contiennent des pages de données ou d'index, regroupés dans des extensions (*extents*).

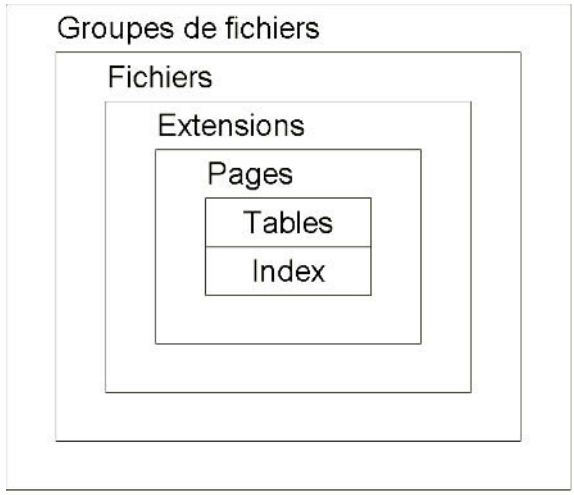

Figure 2.2 — Organisation du stockage dans un fichier de données.

Le groupe de fichiers est un contenu logique, qui fonctionne comme destination implicite ou explicite d'objets (partitions, tables, index). Lorsque vous créez une table ou un index, vous pouvez choisir le groupe de fichiers dans lequel il sera stocké. Une base de données comporte au minimum un groupe de fichiers, créé automatiquement au `CREATE DATABASE`. Ce groupe s'appelle PRIMARY, et il recueille les tables de catalogue (tables système de métadonnées). Il ne peut donc pas être supprimé. Chaque base possède un groupe de fichiers par défaut dans lequel sera créé tout objet si aucun autre groupe de fichiers n'est explicitement spécifié dans l'ordre de création. Ce groupe de fichiers par défaut peut-être changé, avec la commande :

```
ALTER DATABASE nom_de_la_base
MODIFY FILEGROUP [nom_du_groupe_de_fichiers] DEFAULT;
```

Les commandes pour connaître le groupe de fichiers par défaut sont :

```
SELECT *
FROM sys.filegroups
WHERE is_default = 1;
-- ou
SELECT FILEGROUPPROPERTY('PRIMARY', 'IsDefault')
```

Un groupe de fichiers peut comporter un ou plusieurs fichiers. Il s'agit simplement de fichiers disque, qui indiquent l'emplacement physique du stockage. Si le groupe de fichiers est multifichiers, SQL Server y gère les écritures à la manière d'un RAID 1 logiciel, en simple miroir. Le contenu est donc réparti plus ou moins également entre les différents fichiers.

Les objets (tables et index) sont donc attribués à des groupes de fichiers. La répartition des données s'effectue dans des pages de 8 Ko, composées d'un en-tête de

96 octets et d'un corps contenant des lignes de table ou des structures d'index. Une page peut stocker 8 060 octets de données (8192 – 96 – un peu de place laissée par sécurité).

La **page** est l'unité minimale de lecture/écriture de SQL Server. En d'autres termes, lorsque le moteur de stockage écrit ou lit sur le disque, il ne peut écrire ou lire moins qu'une page. L'impact sur les performances est simple : plus vos lignes sont compactes (peu de colonnes, avec un type de données bien choisi), plus elles pourront résider dans la même page, et moins le nombre nécessaire de pages pour conserver la table sera élevé.

Toutes les informations d'entrées-sorties retournées par le moteur de stockage (notamment dans le cas de récupérations de statistiques IO *via* la commande `SET STATISTICS IO ON` de la session[1]) sont exprimées en pages. Cette commande de session permet l'affichage de statistiques d'entrées/sorties après chaque instruction SQL. Elle est très utile pour juger du coût d'une requête. Nous en détaillons l'utilisation à la section 5.1.

Donc, SQL Server lit une page, pas moins, et une **page** contient une ou plusieurs **lignes**, mais une ligne doit résider tout entière dans la page. Cela signifie que la somme de la taille nécessaire au stockage de chaque colonne ne peut dépasser 8 060 octets, sauf exceptions qui suivent. Que se passe-t-il si la taille d'une ligne dépasse légèrement la moitié de la taille de la page ? Tout à fait comme lorsque vous essayez de placer dans une boîte deux objets qui n'y tiennent pas ensemble : vous devez acheter une seconde boîte, et y placer un objet dans chacune, laissant une partie de l'espace de la boîte inutilisée. C'est ce qu'on appelle la fragmentation interne, et cela peut donc constituer une augmentation de taille de stockage non négligeable sur les tables volumineuses, ainsi que diminuer les performances de lectures de données.

Les **extensions** permettent de gérer l'allocation de l'espace du fichier. Elles contiennent 8 pages contiguës, et sont donc d'une taille de 64 Ko. Pour ne pas laisser trop d'espace inutilisé, SQL Server partage les petites tables dans une même extension, appelée « extension mixte ». Lorsque la taille de la table dépasse les 8 pages, de nouvelles extensions (extensions uniformes) sont allouées uniquement à cette table. Un désordre des pages dans les extensions entraîne une fragmentation externe. Comme un ordre particulier des lignes dans les pages, et des pages dans les extensions n'ont de sens que pour une table ordonnée par un index *clustered*, nous traiterons des problématiques de fragmentation dans le chapitre consacré aux index.

Un fichier de données étant composé de pages, et seulement de pages, elles ne stockent pas uniquement le contenu de tables et d'index, bien que ce soit de très loin leur utilisation la plus fréquente. Quelques autres types existent, que vous pourrez

1. Session est le terme officiel pour une connexion utilisateur.

voir apparaître de temps en temps, principalement dans les résultats de requêtes sur les vues de gestion dynamique. Vous trouvez leur description dans les BOL, sous l'entrée *Pages and Extents*. Ce sont principalement des pages de pointage et d'information sur les structures : SPF, IAM, GAM, SGAM. Elles couvrent un certain nombre de pages ou d'étendues (cette quantité étant nommée un intervalle). Les pages SPF couvrent 8 088 pages (environ 64 Mo), les pages GAM, SGAM et IAM couvrent 64 000 extensions (environ 4 Go). Sans entrer dans les détails[1], voici leur signification :

- **SPF** (*Page Free Space*) : contient un octet d'information par page référencée, indiquant notamment si la page est libre ou allouée, et quel est son pourcentage d'espace libre.
- **GAM** (*Global Allocation Map*) : contient un bit par extension, qui indique si elle est libre (1) ou déjà allouée (0).
- **SGAM** (*Shared Global Allocation Map*) : contient un bit par extension, qui indique si elle est une extension mixte qui a au moins une page libre (1), ou si elle est une extension uniforme, ou qui n'a pas de page libre (0).
- **IAM** (*Index Allocation Map*) : indique quelles pages appartiennent à un objet dans un intervalle GAM.

La combinaison des informations contenues dans les pages de GAM, SGAM, IAM et SPF permet à SQL Server de gérer au mieux l'allocation des pages et extensions dans le fichier de données.

Affichage du contenu des pages

La structure détaillée des pages et du stockage des index et des tables à l'intérieur n'est en général pas déterminant pour les performances. Nous reviendrons sur certains aspects importants au fil de l'ouvrage. Sachez toutefois que vous pouvez inspecter le **contenu des pages** à l'aide d'une commande non documentée : `DBCC PAGE`. Elle est utile pour se rendre compte de la façon dont travaille le moteur de stockage, et pour expérimenter certains aspects du fonctionnement de SQL Server. Nous utiliserons cette commande dans le livre, notamment au chapitre 6. La syntaxe de DBCC PAGE est la suivante :

```
DBCC PAGE ( {'dbname' | dbid}, filenum, pagenum [, printopt={0|1|2|3} ])
```

où `filenum` est le numéro de fichier et `pagenum` le numéro de page, que vous trouvez dans d'autres commandes en général non documentées, qui vous retourne un pointeur vers une page dans le format #:#. Par exemple 1:543 veut dire la page 543 dans le fichier 1. `Printopt` permet d'indiquer le détail d'informations à retourner :

0 – en-tête de la page.

1 – contenu détaillé incluant la table d'offset (*row offset array*, ou *offset table*, l'adresse des lignes dans la page).

1. Disponibles par exemple sur le blog de Paul Randal : http://blogs.msdn.com/sqlserverstorageengine/

2 – comme un, à la différence que le contenu est retourné en un dump hexadécimal.

3 – retourne l'en-tête, plus le contenu de la page en format tabulaire (jeu de résultat) pour les index, ou détaillé pour le contenu de tables.

Le résultat de `DBCC PAGE` est envoyé par défaut dans le journal d'erreur de SQL Server (le fichier `errorlog`, qu'on trouve dans le répertoire de données de SQL Server, dans le sous-répertoire `LOG`). Pour retourner le résultat dans la session, vous devez activer le drapeau de trace[1] 3604 :

```
DBCC TRACEON (3604)
```

`DBCC PAGE` supporte aussi l'option `WITH TABLERESULTS`, qui vous permet de retourner le résultat en format table.

Le résultat d'une commande `DBCC` n'est pas aisé à manipuler par code, une syntaxe telle que `INSERT INTO … DBCC …` n'est pas prise en charge. Pour insérer le résultat d'une commande `DBCC` dans une table (temporaire par exemple), la seule solution est de passer par du SQL dynamique (c'est-à-dire de passer la commande `DBCC` dans une chaîne de caractères et de l'exécuter par un `EXECUTE`), et de profiter de la syntaxe `INSERT INTO … EXECUTE ()`. Par exemple :

```
CREATE TABLE #dbccpage (ParentObject sysname, Object sysname,
                        Field sysname, Value sysname)
GO
INSERT INTO #dbccpage
EXEC ('DBCC PAGE (AdventureWorks, 1, 0, 3) WITH TABLERESULTS');
GO
SELECT * FROM #dbccpage
```

Vous pouvez, grâce à `DBCC PAGE`, observer la structure d'allocation dans les fichiers de données. Dans l'en-tête renvoyé par `DBCC PAGE`, vous trouvez l'indication des pages de PFS, GAM et SGAM qui référencent la page. Au début de chaque intervalle GAM, vous trouvez une extension GAM qui ne contient que des pages de structure. La première extension d'un fichier est une extension GAM, qui contient en page 0 l'en-tête de fichier, en page 1 la première page PFS, en page 2 la première page GAM, etc. Mais, tout cela nous pouvons le découvrir en utilisant `DBCC PAGE`. Observons la première page du fichier :

```
DBCC PAGE (AdventureWorks, 1, 0, 3)
```

Dans l'en-tête de la page, nous récupérons les informations d'allocation :

```
Allocation Status

GAM (1:2) = ALLOCATED                SGAM (1:3) = NOT ALLOCATED
PFS (1:1) = 0x44 ALLOCATED 100_PCT_FULL
DIFF (1:6) = CHANGED                 ML (1:7) = NOT MIN_LOGGED
```

1. Voir nomenclature en début d'ouvrage. Le drapeau de trace permet de modifier des options internes de SQL Server.

Plus loin, nous pouvons observer les informations contenues dans l'en-tête. Par exemple :

```
FileGroupId = 1
FileIdProp = 1       Size = 30816       MaxSize = 65535
Growth = 2048
….
MinSize = 15360       Status = 0       UserShrinkSize = 65535
```

Nous savons donc quelle est la première page de PFS : 1:1. Un `DBCC PAGE` nous liste ensuite les pages qu'elle référence, avec pour chacune l'indication de sa nature : IAM ou non. Voici un extrait du résultat :

```
(1:651)    -            =  ALLOCATED   0_PCT_FULL  Mixed Ext
(1:652)    -            =  ALLOCATED   0_PCT_FULL  IAM Page  Mixed Ext
(1:653)    -            =  ALLOCATED   0_PCT_FULL  Mixed Ext
(1:654)    - (1:655)    =  ALLOCATED   0_PCT_FULL  IAM Page  Mixed Ext
```

La page 652 est donc une page IAM. Allons voir :

```
[…]
Metadata: AllocUnitId = 72057594052083712
Metadata: PartitionId = 72057594045726720       Metadata: IndexId = 1
Metadata: ObjectId = 402100473  m_prevPage = (0:0)  m_nextPage = (0:0)
pminlen = 90          m_slotCnt = 2          m_freeCnt = 6
m_freeData = 8182     m_reservedCnt = 0      m_lsn = (37:4181:117)

[…]

IAM: Single Page Allocations @0x6105C08E

Slot 0 = (1:653)              Slot 1 = (0:0)              Slot 2 = (0:0)
Slot 3 = (0:0)                Slot 4 = (0:0)              Slot 5 = (0:0)
Slot 6 = (0:0)                Slot 7 = (0:0)


IAM: Extent Alloc Status Slot 1 @0x6105C0C2

(1:0)        - (1:8456)     = NOT ALLOCATED
(1:8464)     - (1:8496)     =     ALLOCATED
(1:8504)     - (1:8672)     = NOT ALLOCATED
(1:8680)     -              =     ALLOCATED
(1:8688)     - (1:30808)    = NOT ALLOCATED
```

La **page IAM** contient un tableau d'allocation de l'extension, et pour chaque page de l'extension, le détail des pages allouées. Dans notre exemple, une seule page de l'extension est allouée à l'objet. On comprend donc qu'il s'agit d'une extension mixte. Plus loin, nous avons le détail des pages et extensions allouées à la table. Nous prendrons un exemple plus détaillé sur le sujet dans le chapitre traitant des index, afin de comprendre leur structure.

Sparse columns

SQL Server 2008 introduit une méthode de stockage optimisée pour les tables qui comportent beaucoup de colonnes pouvant contenir des valeurs NULL, permettant de limiter l'espace occupé par les marqueurs `NULL`. Pour cela, vous disposez du nouveau mot-clé `SPARSE` à la création de vos colonnes. Vous pouvez aussi créer une colonne virtuelle appelée `COLUMN_SET`, qui permet de retourner rapidement les colonnes sparse non null. Cette voie ouverte à la dénormalisation est à manier avec prudence, et seulement si nécessaire.

2.2.2 Journal de transactions

Le **journal de transactions** (*transaction log*) est un mécanisme courant dans le monde des SGBD, qui permet de garantir la cohérence transactionnelle des opérations de modification de données, et la reprise après incident. Vous pouvez vous reporter à la section 7.5 pour une description détaillée des transactions. En un mot, toute écriture de données est enrôlée dans une transaction, soit de la durée de l'instruction, soit englobant plusieurs instructions dans le cas d'une transaction utilisateur déclarée à l'aide de la commande `BEGIN TRANSACTION`. Chaque transaction est préalablement enregistrée dans le journal de transactions, afin de pouvoir être annulée ou validée en un seul bloc (propriété d'atomicité de la transaction) à son terme, soit au moment où l'instruction se termine, soit dans le cas d'une transaction utilisateur, au moment d'un `ROLLBACK TRANSACTION` ou d'un `COMMIT TRANSACTION`. Toute transaction est ainsi complètement inscrite dans le journal avant même d'être répercutée dans les fichiers de données. Le journal contient toutes les informations nécessaires pour annuler la transaction en cas de *rollback* (ce qui équivaut à défaire pas à pas toutes les modifications), ou pour la rejouer en cas de relance du serveur (ce qui permet de respecter l'exigence de durabilité de la transaction).

Le mécanisme permettant de respecter toutes les contraintes transactionnelles est le suivant : lors d'une modification de données (`INSERT` ou `UPDATE`), les nouvelles valeurs sont inscrites dans les pages du cache de données, c'est-à-dire dans la RAM. Seule la page en cache est modifiée, la page stockée sur le disque restant intacte. La page de cache est donc marquée « sale » (*dirty*), pour indiquer qu'elle ne correspond plus à la page originelle. Cette information est consultable à l'aide de la colonne `is_modified` de la vue de gestion dynamique (DMV) `sys.dm_os_buffer_descriptors` :

```
SELECT * FROM sys.dm_os_buffer_descriptors WHERE is_modified = 1
```

La transaction validée est également maintenue dans le journal. À intervalle régulier, un **point de contrôle** (*checkpoint*) aussi appelé point de reprise, est inscrit dans le journal, et les pages modifiées par des transactions validées avant ce checkpoint sont écrites sur le disque.

Les vues de gestion dynamique

À partir de la version 2005 de SQL Server, vous avez à votre disposition, dans le schéma `sys`, un certain nombre de vues systèmes qui portent non pas sur les métadonnées de la base en cours, mais sur l'état actuel du SGBD. Ces vues sont nommées « vues de gestion dynamique » (*dynamic management views*, DMV). Elles sont toutes préfixées par `dm_`. Vous

les trouvez dans l'explorateur d'objets, sous vues/vues système, et vous pouvez en obtenir la liste à l'aide des vues de métadonnées, par exemple ainsi :

```
SELECT *
FROM  sys.system_objects
WHERE name LIKE 'dm_%' ORDER BY name;
```

Vous y trouverez la plupart des vues disponibles. Certaines n'y figurent pas car elles sont non documentées, ou sont mises en œuvre sous forme de fonctions.

Les données renvoyées par ces vues dynamiques sont maintenues dans un cache spécial de la mémoire adressée par SQL Server. Elles ne sont pas stockées de façon persistante sur le disque : les compteurs sont remis à zéro lors du redémarrage du service SQL. Par conséquent, pour que ces vues offrent des informations utiles, vous ne devez les consulter qu'après un temps significatif d'utilisation du serveur. Nous utiliserons beaucoup les vues de gestion dynamiques tout au long de cet ouvrage.

Point de contrôle et récupération

Comment garantir la cohérence d'une base de données en cas de panne ou de redémarrage intempestif ? Les techniques appliquées dans SQL Server sont également standards aux SGBDR. Premièrement, le maintien de toutes les informations de transaction dans le journal, avec les marqueurs temporels de leur début et fin, assure qu'elles pourront être reproduites. Lorsque l'instance SQL redémarre, chaque base de données entre en étape de récupération (*recovery*) : les transactions du journal non encore répercutées dans les pages de données sont reproduites. Ce sont soit des transactions qui n'étaient pas terminées lors du dernier point de contrôle (*checkpoint*), soit qui ont commencé après celui-ci. Si la transaction n'est pas validée dans le journal (parce qu'elle a été annulée explicitement, ou parce qu'elle n'a pas pu se terminer au moment de l'arrêt de l'instance), elle est annulée. Selon le volume des transactions journalisées et la date du dernier point de contrôle, cette récupération peut prendre du temps. SQL Server s'assure donc d'émettre un point de contrôle suffisamment souvent pour maintenir une durée de récupération raisonnable.

Une option de l'instance permet de régler la durée maximum, en minutes, de la récupération : le *recovery interval*. Par défaut la valeur est 0, ce qui correspond à une auto-configuration de SQL Server pour assurer dans la plupart des cas une récupération de moins d'une minute. Changer cette valeur diminue les fréquences de point de contrôle et augmente le temps de récupération. Vous pouvez le faire dans les propriétés de l'instance dans SSMS, ou *via* `sp_configure` :

```
EXEC sp_configure 'show advanced option', '1';
RECONFIGURE;
EXEC sp_configure 'recovery interval', '3';
RECONFIGURE WITH OVERRIDE;
EXEC sp_configure 'show advanced option', '1';
```

La procédure stockée système sp_configure est utilisée pour modifier des options de configuration de l'instance SQL Server. Certaines options sont aussi visibles dans les boîtes de dialogues de configuration dans SSMS. Par défaut, sp_configure ne touche que les options courantes. Pour afficher et changer les options avan-

cées, vous devez utiliser sp_configure lui-même pour les activer avec sp_configure 'show advanced option', '1'.

La fréquence d'exécution des points de contrôle est donc calculée par rapport à cet intervalle, et selon la quantité de transactions dans le journal. Elle sera basse pour une base principalement utilisée pour la lecture, et de plus en plus haute selon le volume de modifications. Vous n'avez généralement pas besoin de modifier cette option. Si vous faites l'expérience de pointes d'écriture sur le disque de données, avec une file d'attente qui bloque d'autres lectures/écritures, et seulement dans ce cas, vous pouvez songer à l'augmenter, par exemple à cinq minutes, en continuant à surveiller les compteurs de disque, pour voir si la situation s'améliore. Nous reviendrons sur ces compteurs dans la section 5.3 concernant le moniteur système.

La conservation des transactions dans le journal après point de contrôle est contrôlée par l'option de base de données appelée **mode de récupération** (*recovery model*). Elle est modifiable par la fenêtre de propriétés d'une base de données dans SSMS, ou par la commande :

```
ALTER DATABASE [base] SET RECOVERY { FULL | BULK_LOGGED | SIMPLE }
```

Les trois options modifient le comportement du journal de la façon suivante :

- **FULL** – Toutes les transactions sont conservées dans le journal, en vue de leur sauvegarde. Cette option n'est donc à activer que sur les bases de données pour lesquelles vous voulez pratiquer une stratégie de sauvegarde impliquant des sauvegardes de journaux. Seule cette option, alliée aux sauvegardes, permet de réaliser des restaurations à un point dans le temps. Si le journal n'est pas sauvegardé, le fichier grandira indéfiniment pour conserver toutes les transactions enregistrées.
- **BULK_LOGGED** – Afin de ne pas augmenter exagérément la taille du journal, les opérations en lot (`BULK INSERT`, `SELECT INTO`, `CREATE INDEX`...) sont journalisées de façon minimale, c'est-à-dire que leurs transactions ne sont pas enregistrées en détail. Seules les adresses de pages modifiées sont inscrites dans le journal. Lors de la sauvegarde de journal, les extensions modifiées sont ajoutées à la sauvegarde. Si une sauvegarde de journal contient des modifications en lot, la sauvegarde ne pourra pas être utilisée pour une restauration à un point dans le temps.
- **SIMPLE** – Dans le mode simple, le journal de transactions est vidé de son contenu inutile (ce qu'on appelle la portion inactive du journal) après chaque point de contrôle. Aucune sauvegarde de journal n'est donc possible. Par contre, la taille du journal restera probablement raisonnable.

En SQL Server, le journal de transactions consiste en un ou plusieurs fichiers, au remplissage séquentiel, dans un format propriétaire complètement différent des fichiers de données. Ce format n'est pas publié par Microsoft, ce qui fait qu'il n'y a pas réellement de méthode officielle pour en lire le contenu et en retirer des infor-

mations exploitables. Il est également impossible de récupérer ou annuler une transaction validée à partir du journal. Il est par contre possible de faire des sauvegardes de journal et de les restaurer à un point dans le temps, annulant ainsi toutes les transactions validées à cet instant.

Le journal de transactions est donc une chaîne séquentielle d'enregistrements de transactions, comportant toutes les informations nécessaires à une réexécution de la transaction par le système lors d'une phase de restauration ou de récupération, ainsi qu'un numéro de séquence d'enregistrement, le *log sequence number*, ou LSN. Une restauration de journal peut également être effectuée jusqu'à un LSN. Encore faut-il savoir quel est le LSN d'une opération recherchée. Nous l'avons dit, le format du journal n'est pas connu. Il n'y a que deux manières de lire ce journal de transactions : acheter un logiciel tiers, ou utiliser des instructions SQL Server non documentées.

Affichage du contenu du journal de transactions

Quelques éditeurs, dont le plus connu est Lumigent avec son outil nommé « Log Explorer », ont déchiffré, avec des méthodes d'ingénierie inverse (*reverse engineering*), le format du journal, et proposent des outils plus ou moins puissants pour voir et utiliser le contenu du journal et le détail des opérations effectuées par chaque transaction.

SQL Server comporte également quelques commandes non documentées – c'est-à-dire cachées, et susceptibles d'être modifiées ou retirées sans préavis dans des versions suivantes du produit – qui affichent des informations sur le journal. La plus complète est une fonction système, `sys.fn_dblog` (ou `::fn_dblog`), que vous pouvez appeler ainsi dans le contexte de la base de données désirée :

```
SELECT * FROM sys.fn_dblog(DB_ID(),NULL)
```

Cette vue vous retourne les informations de LSN, le type d'opération effectuée, l'identifiant de transaction, et plusieurs éléments intéressants comme le nom de l'unité d'allocation (la table ou l'index affecté), le nombre de verrous posés, etc. Il ne manque que le détail des modifications, qui pourrait servir à savoir précisément ce qui s'est produit. Chaque ligne comporte une colonne nommée « Previous LSN », qui permet de suivre l'ordre des instructions d'une même transaction. Le type d'instruction est assez clair, comme par exemple `LOP_BEGIN_XACT`, `LOP_MODIFY_ROW` et `LOP_COMMIT_XACT`.

Cette fonction peut être utile pour obtenir le détail transactionnel d'une opération. Prenons un exemple. Nous chercherons d'abord à obtenir un journal aussi vide que possible. Nous mettons donc la base de données en mode simple, et nous provoquons manuellement un `CHECKPOINT`.

```
ALTER DATABASE AdventureWorks SET RECOVERY SIMPLE;
GO
CHECKPOINT;
```

SQL Server 2008 – En SQL Server 2005, nous pouvons aussi lancer la commande `BACKUP LOG AdventureWorks WITH TRUNCATE_ONLY;` pour vider manuellement le journal. Cette commande n'est plus prise en charge dans SQL Server 2008.

En exécutant la fonction `fn_dblog()`, nous constatons que le journal est pratiquement vide. Effectuons une modification simple dans la table `Sales.Currency`, qui contient les références de devises :

```
UPDATE Sales.Currency
SET Name = Name
WHERE CurrencyCode = 'ALL'
```

La modification demandée ne réalise aucun changement. SQL Server va-t-il tout de même faire un `UPDATE` ? Grâce à `fn_dblog()` nous constatons que non. Une seule opération est journalisée, de type `LOP_SET_BITS`, une mise à jour des informations d'une page, pas de son contenu.

Essayons maintenant de réellement changer quelque chose :

```
UPDATE Sales.Currency
SET Name = 'Lek2'
WHERE CurrencyCode = 'ALL'
```

Les opérations du tableau 2.1 sont inscrites dans le journal.

Tableau 2.1 — Opérations inscrites dans le journal

Operation	Context	AllocUnitName
LOP_BEGIN_XACT	LCX_NULL	NULL
LOP_MODIFY_ROW	LCX_CLUSTERED	Sales.Currency.PK_ Currency_CurrencyCode
LOP_SET_BITS	LCX_DIFF_MAP	Unknown Alloc Unit
LOP_DELETE_ROWS	LCX_MARK_AS_GHOST	Sales.Currency.AK_ Currency_Name
LOP_SET_BITS	LCX_PFS	Sales.Currency.AK_ Currency_Name
LOP_INSERT_ROWS	LCX_INDEX_LEAF	Sales.Currency.AK_ Currency_Name
LOP_COMMIT_XACT	LCX_NULL	NULL

Ici les opérations sont claires : modification de la ligne (`LOP_MODIFY_ROW` sur l'index *clustered*, donc sur la table), et ensuite opération de suppression puis d'insertion dans l'index `AK_Currency_Name`, qui contient la colonne modifiée dans sa clé.

Vous verrez parfois l'`UPDATE` journalisé non comme un `LOP_MODIFY_ROW`, mais comme un couple de `LOP_DELETE_ROWS` et `LOP_INSERT_ROWS`, car SQL Server gère deux

types de mises à jour : l'*update* direct et l'*update* différé (*deferred*), selon les cas. L'*update* différé correspond donc à une suppression de ligne suivie d'un ajout de ligne. C'est par exemple le cas lorsque la mise à jour augmente la taille de la ligne et la force à être déplacée dans une autre page de données.

2.2.3 Taille des fichiers

Physiquement, le journal est composé de **fichiers virtuels de journal** (*virtual log files*, ou VLF), qui sont des blocs logiques dans le même fichier de journal, de taille et de nombre variables. SQL Server essaie de conserver un nombre limité de VLF, car ceux-ci pèsent sur les performances. Lorsque le journal augmente de taille automatiquement, le nombre de VLF augmente et est plus important que si la taille avait été prévue correctement dès le départ. Ne soyez donc pas trop avare d'espace à la création de vos bases de données.

Vous pouvez maintenir plusieurs fichiers de journal sur la même base de données. Cela ne constituera pas une optimisation de performances par parallélisation des écritures, car, nous l'avons vu, le journal de transactions est par nature séquentiel. Chaque fichier sera utilisé à tour de rôle, SQL Server passant au fichier suivant lorsque le précédent sera plein. Cela ne sera donc utile que pour pallier un manque d'espace sur des disques durs de petite taille.

2.3 SQLOS

Si vous voulez obtenir les meilleures performances de votre serveur SQL, la chose la plus importante en ce qui concerne le système lui-même, est de le dédier à SQL Server. La couche représentée par le système d'exploitation lui-même, et les autres applications s'exécutant en concurrence, doit être réduite au maximum, en désactivant par exemple les services inutiles. SQL Server est livré avec d'autres services, notamment toute la partie *Business Intelligence*, qui fait appel à des moteurs différents : Analysis Service est un produit conceptuellement et physiquement tout à fait séparé du moteur relationnel, de même que Reporting Services utilise des technologies qui entrent en concurrence avec SQL Server (dans la version 2005, Reporting Services est fondé sur IIS, le serveur web de Microsoft, qui ne fait notoirement pas bon ménage avec SQL Server, en terme de performances s'entend. Dans SQL Server 2008, il est plus intégré et utilise notamment SQLOS). Il est essentiel de déplacer ces services sur des serveurs différents de ceux que vous voulez dédier aux performances de SQL Server, c'est-à-dire de la partie relationnelle, dont le travail de base est de stocker et de service de l'information avec la vélocité et la cohérence maximales. Un serveur est une ville trop petite pour deux personnalités comme SQL Server.

Bien que SQL Server soit étroitement lié au système d'exploitation Windows, il ne se repose pas, comme la quasi-totalité des applications utilisateurs, sur toutes ses

fonctionnalités d'accès et de partage des ressources système. Les recherches en bases de données, un domaine actif depuis longtemps, montrent que les meilleures performances sont atteintes en laissant le soin au SGBDR de gérer certaines parties de ce qui est traditionnellement de la responsabilité du système d'exploitation. Oracle, par exemple, cherche depuis longtemps à minimiser le rôle de l'OS, en contournant les systèmes de fichiers journalisés à l'aide de *raw devices* (*Oracle directIO*), en proposant un outil comme ASM (*Automatic Storage Manager*) pour gérer directement le système de fichiers, ou en intégrant depuis peu sa propre couche de virtualisation, basée sur Xen.

SQL Server n'échappe pas à cette règle. Il intègre sa couche de gestion de fonctionnalités bas niveau, nommée **SQLOS** (pour *SQL Server Operating System*). Elle prend en charge des fonctionnalités d'ordonnancement des *threads* de travail, la gestion de la mémoire, la surveillance des ressources, les entrées/sorties, la synchronisation des *threads*, la détection des verrous mortels (*deadlocks*), etc. qui permettent de gérer à l'intérieur même de SQL Server, proche des besoins du moteur, les éléments essentiels à un travail efficace avec les ressources de la machine. Vous trouvez un schéma des différents modules présents dans SQLOS en figure 2.3.

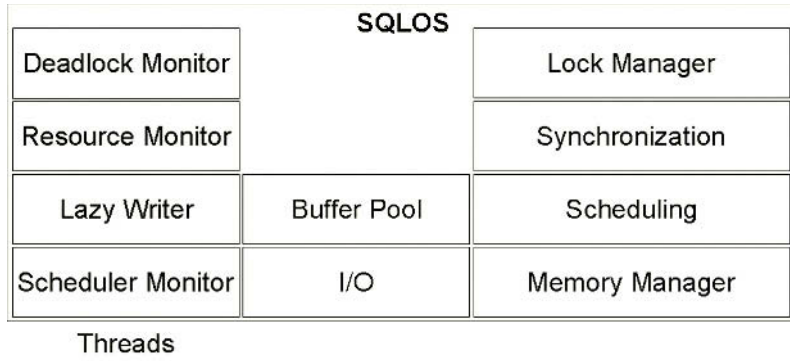

Figure 2.3 — Modules de SQLOS

Vous pouvez observer l'activité de SQLOS à l'aide de vues de gestion dynamique préfixées par `sys.dm_os_`. Les vues de gestion dynamique (*dynamic management views*, DMV) sont des vues qui retournent des informations sur l'état du système, et non sur des données ou des métadonnées stockées dans une base.

SQLOS permet de gérer efficacement les systèmes à multiprocesseurs. Un ordonnanceur (*scheduler*) est lié à chaque processeur physique, qui gère à l'intérieur de SQL Server, les *threads*, en mode utilisateur (*user mode*) (le système d'exploitation sous-jacent gère les *threads* en mode noyau (*kernel mode*)). La raison en est que SQL Server connaît mieux ses besoins de multitâche que le système d'exploitation. Windows gère les *threads* de façon dite préemptive, c'est-à-dire qu'il force les processus à s'arrêter pour laisser du temps aux processus concurrents. Les ordonnanceurs de SQL Server fonctionnent en mode non préemptif : à l'intérieur de SQLOS, les processus

laissent eux-mêmes la place aux autres, dans un mode collaboratif, ce qui permet d'obtenir de meilleures performances. Un ordonnanceur est lié à un processeur physique. Il gère plusieurs *threads*, qui sont eux-mêmes liés à des *workers*. Le *worker* est le processus de travail. Il exécute un travail complètement, ce qui évite les changements de contextes qui peuvent se produire en cas de multitâche préemptif. Un changement de contexte (*context switch*) représente la nécessité, pour un processus, de sauver son état lorsqu'il donne la main, puis de le recharger lorsqu'il reprend le travail, comme un travailleur doit déposer ses affaires dans son placard en partant, et les reprendre en revenant. Certains *threads* à l'intérieur de l'espace de SQL Server ne sont pas lancés par SQL Server, c'est le cas lors de l'exécution de procédures stockées étendues. Nous voyons tout cela schématiquement sur la figure 2.4, avec les vues de gestions dynamique qui correspondent.

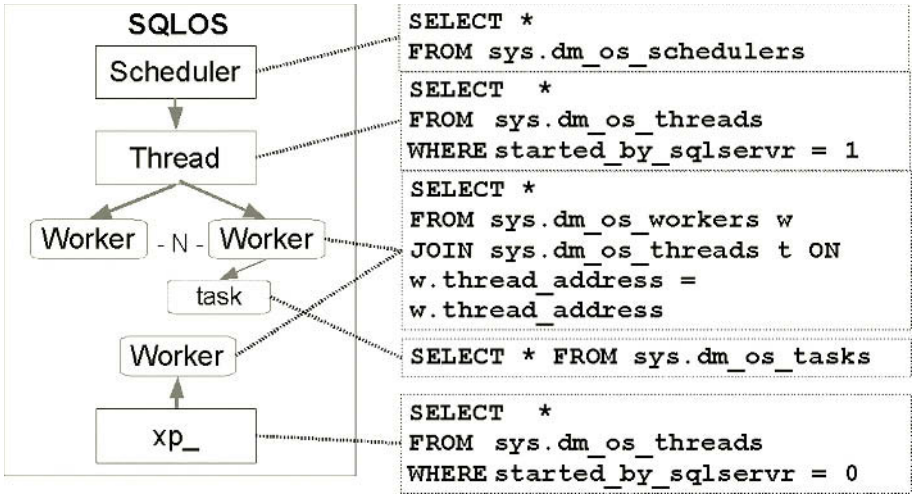

Figure 2.4 — SQLOS, l'ordonnanceur

Ces vues vous permettent de voir les processus en activité, la mémoire de travail utilisée par chacun, s'ils sont en travail ou en attente, etc.

Outre la gestion locale des processus, SQLOS assure aussi la gestion locale de l'allocation mémoire. Un *Memory Broker* calcule dynamiquement les quantités optimales de RAM pour les différents objets en mémoires. Ces objets sont des caches (*buffers*), des pools, qui gèrent des objets plus simples que les caches, par exemple la mémoire pour les verrous, et des *clerks*, qui sont des gestionnaires d'espace mémoire fournissant des services d'allocation et de notification d'état de la mémoire. Ces différents éléments sont reproduits sur la figure 2.5.

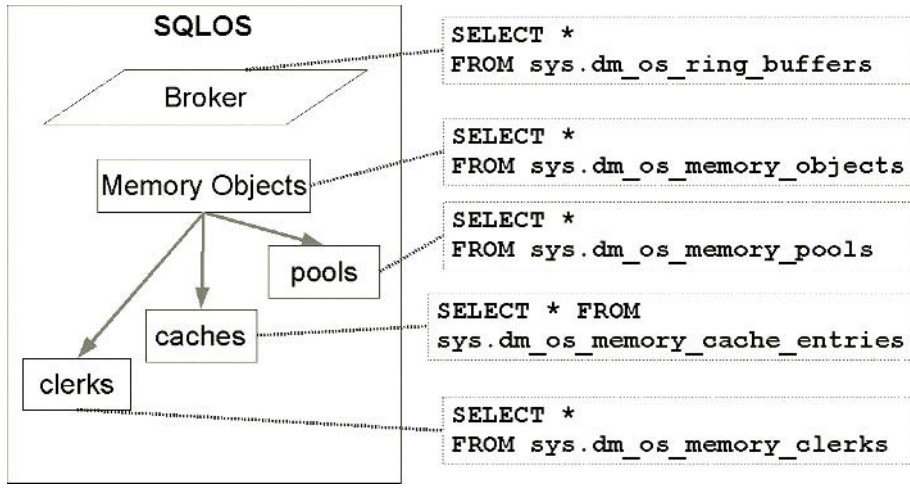

Figure 2.5 — SQLOS, la mémoire

Des vues de gestion dynamique donnent des informations sur les caches :

- `sys.dm_exec_cached_plans` ;
- `sys.dm_os_buffer_descriptors` ;
- `sys.dm_os_memory_cache_clock_hands` ;
- `sys.dm_os_memory_cache_counters` ;
- `sys.dm_os_memory_cache_entries` ;
- `sys.dm_os_memory_cache_hash_tables`.

Nous reverrons certaines de ces vues dans notre chapitre sur le choix du matériel. Regardez ces vues, lorsque vous avec une compréhension générale de SQL Server : elles vous sont utiles pour détecter des problèmes éventuels, comme de la pression mémoire, ou des contentions de CPU.

SQL Server 2008 ajoute quelques vues de gestions dynamiques sur SQLOS :
– `sys.dm_os_nodes` – donne des informations sur les nœuds SQLOS, c'est-à-dire les liens fondamentaux entre SQLOS et Windows.
– `sys.dm_os_process_memory` – donne le résumé de l'utilisation mémoire de l'instance SQL. Très utile pour obtenir les valeurs globales, en Ko.
–`sys.dm_os_sys_memory` – donne le résumé de l'utilisation mémoire de Windows sur le système.

3

Optimisation du matériel

Objectif

Tout comme une plante ne peut atteindre sa taille normale et porter ses fruits à pleine maturité que lorsque la terre et les conditions météorologiques sont adaptées, SQL Server est totalement dépendant de la qualité du matériel sur lequel il est installé. Ce chapitre présente les éléments importants de ce matériel.

3.1 CHOIX DE L'ARCHITECTURE MATÉRIELLE

SQL Server est par nature grand consommateur de ressources physiques. Un système de gestion de bases de données a besoin de bonnes performances des sous-systèmes de stockage, avec lesquels il travaille beaucoup, non seulement pour écrire et lire des données parfois en gros volumes, mais aussi pour maintenir les journaux de transaction, et `tempdb`, la base de données qui recueille les tables temporaires, les tables de travail internes ainsi que les versions de lignes dans les fonctionnalités de *row versioning*. La mémoire vive est également sollicitée. SQL Server doit conserver un certain nombre de choses dans l'espace d'adressage virtuel (le *virtual address space*, ou VAS) du système d'exploitation. Dans le VAS, SQL Server pose la mémoire des sessions utilisateurs, des verrous, du cache de plans d'exécution, la mémoire d'exécution des requêtes et le cache de données. L'exécution des requêtes peut être très consommatrice de RAM : le moteur d'exécution doit passer d'un opérateur de plan d'exécution à l'autre les lignes récupérées, et parfois tout un ensemble de lignes pour les traiter en une seule fois (pour réaliser un tri ou une table de hachage, par exemple). Ces opérations sont également consommatrices de processeur. Tous les éléments sont donc à prendre en considération. Dans un système traitant un volume raisonnable de

données et suffisamment pourvu en RAM, la vitesse du sous-système disque n'est pas le critère le plus décisif, car la plupart des données lues le seront depuis le cache de données, en RAM. Dans tous les cas, plus la quantité de RAM est importante, meilleures seront les performances. En ce qui concerne les CPU, un système multi-processeurs est aujourd'hui incontournable. SQL Server utilise intensivement le multi-processing, en répartissant ses requêtes sur des *threads* de travail, ou en parallélisant la même requête si l'activité de la machine est suffisamment légère pour le permettre au moment de l'exécution.

3.1.1 Processeur(s)

C'est une évidence, SQL Server utilise activement le multi-processing. N'hésitez pas à bâtir une machine de huit processeurs ou plus. L'édition standard ne supporte que quatre CPU. Entendez quatre processeurs physiques (quatre *sockets*). Si vous utilisez des processeurs *multi-core* (*dual* ou *quad*), cela augmente le nombre de processeurs supportés par cette édition. De même, le mode de licence par processeur de SQL Server est à entendre par *socket*. Pour quatre *quad-core*, donc seize processeurs logiques, vous vous acquittez d'une licence de quatre processeurs. Le *multi-core* est bien pris en charge, mais, l'*hyper-threading* peut poser des problèmes : nous le détaillerons dans la section suivante.

Parallélisme

Les opérations de bases de données, notamment la génération du plan d'exécution et la restitution de données, sont habituellement gourmandes en temps processeur. Les serveurs sont aujourd'hui de plus en plus multiprocesseurs, et les processeurs (notamment Intel) modernes intègrent des architectures de *multi-threading* ou de *multi-core*, qui leur permettent de travailler en parallèle. Deux cas de figure sont possibles : l'exécution en parallèle de requêtes différentes ou l'exécution en parallèle de différentes parties de la même requête. C'est ce deuxième cas de figure qu'on appelle la **parallélisation d'une requête**. La décision de paralléliser une requête est prise en temps réel par le moteur relationnel, selon l'état actuel du système : si l'activité est faible, et la requête est estimée coûteuse (longue), SQL Server sera plus enclin à paralléliser que si la requête est simple et que le système est chargé de demandes de petites requêtes. En d'autres termes, la parallélisation est plus intéressante sur un système analytique de type OLAP, que sur une base de données transactionnelle très active, de type site web. Le degré de parallélisme, ainsi que le seuil à partir duquel la durée estimée d'une requête la classe comme candidate à la parallélisation, sont des paramètres du système que vous pouvez modifier :

```
EXEC sp_configure 'show advanced options', 1
RECONFIGURE
GO

EXEC sp_configure 'affinity mask'
EXEC sp_configure 'max degree of parallelism'
EXEC sp_configure 'cost threshold for parallelism'
```

L'option « *affinity mask* » vous permet de n'attribuer à SQL Server que certains processeurs. La valeur par défaut, 0, indique que tous les CPU sont utilisés. Pour spécifier seulement certains processeurs, attribuez un entier correspondant au masque de bits représentant, du bit de poids faible au bit de poids fort, le ou les processeurs à utiliser. Par exemple, pour n'utiliser que les processeurs 0, 2 et 4, attribuez la valeur binaire 00010101, donc 21 en décimal.

Ce masque d'affinité ne peut donc gérer que 32 processeurs, ce qui est de toute façon la limite de Windows Server 2003 32 bits. Sur les architectures 64 bits, une option supplémentaire est disponible : `affinity64 mask` pour l'affinité sur 32 processeurs supplémentaires.

À moins que vous n'ayez à libérer des processeurs pour une autre application (par exemple une autre instance de SQL Server), il vaut mieux laisser SQL Server utiliser tous les processeurs à disposition. Dans le cas d'un système virtualisé, que nous verrons plus loin, ce sera de toute manière à la machine virtuelle de gérer les processeurs physiques qu'elle utilise, donc ceci ne concerne pas cette section.

L'option « *max degree of parallelism* » indique, sur le nombre de processeurs défini dans l'affinité, combien pourront être enrôlés dans la parallélisation d'une même requête. La valeur par défaut, 0, indique que tous les CPU peuvent être utilisés. Cette option modifie le comportement de SQL Server pour l'exécution de toutes les requêtes. Exemple :

```
EXEC sp_configure 'max degree of parallelism', 2
```

Pour permettre une parallélisation sur deux processeurs au maximum.

Lorsqu'une requête est parallélisée, elle doit utiliser le nombre de processeurs indiqué dans « *max degree of parallelism* », pas moins. En d'autres termes, si vous laissez SQL Server paralléliser sans limite sur un système comportant beaucoup de processeurs, tous les processeurs devront prendre en charge la requête, et le coût afférant à la synchronisation entre les processeurs sera sûrement plus important que le gain de performance de l'exécution en parallèle, faisant donc plus de mal que de bien.

De plus, le parallélisme peut poser problème sur des processeurs utilisant la technologie d'*hyper-threading*. Cette technologie propriétaire d'Intel, intégrée aux processeurs Pentium 4, est moins utilisée aujourd'hui mais pourrait faire son retour dans la future architecture « Nehalem ». Elle permet sous certaines conditions d'améliorer les performances d'applications *multi-threadées*. Un processeur *hyper-threadé* est vu par Windows comme deux processeurs[1], et SQL Server peut tenter de paralléliser une requête sur ce qui représente en réalité le même processeur physique. On obtiendra ainsi un ralentissement au lieu d'une amélioration, en provoquant des attentes *inter-threads*. Ces ralentissements pourront se constater grâce à un type particulier d'attente (*wait*). Nous reviendrons en détail sur la détection des attentes dans le chapitre 7. Sachez pour l'instant que vous pouvez obtenir des statistiques d'attentes

1. SQL Server utilise le membre de structure SYSTEM_INFO.dwNumberofProcessors de l'API Windows pour connaître le nombre de processeurs. Cette propriété retourne le nombre total de processeurs physiques et logiques

de processus SQL Server grâce à la vue de gestion dynamique `sys.dm_os_wait_stats`. La colonne `wait_type` indique pour quelle raison l'attente a eu lieu. Lors d'une exécution en parallèle, les threads ont besoin de s'échanger des données. Le type `CXPACKET` indique une attente de synchronisation d'échange de paquets. Ainsi, si vous constatez des ralentissements liés à des attentes de type `CXPACKET`, il ne vous reste plus qu'à jouer avec « *max degree of parallelism* ».

Si vous constatez, sur des processeurs *hyper-threadés*, une augmentation de l'utilisation du CPU mais une baisse de performances, liées à des messages visibles dans le journal d'erreur de SQL Server (*errorlog*) indiquant qu'un *thread* n'a pas réussi à acquérir un *spinlock* (une structure légère de synchronisation de *threads*), vous vous retrouvez certainement dans la situation décrite par Slava Oks ici : http://blogs.msdn.com/slavao/archive/2005/11/12/492119.aspx. La solution est de désactiver purement et simplement l'*hyper-threading* dans le BIOS de votre machine.

Comme remarque générale, considérant les problèmes potentiels que génère l'*hyper-threading*, faites des tests de charge *hyper-threading* désactivé puis activé, et si vous ne voyez pas de gain de performance notable avec l'*hyper-threading*, désactivez-le pour de bon.

NUMA

Comme nous allons le voir, sur une architecture NUMA, des bus séparés sont alloués aux processeurs pour accéder à la mémoire. La mémoire accédée sur le bus du processeur est nommée mémoire locale, et la mémoire accessible sur les autres bus est logiquement nommée mémoire distante. Vous vous doutez que l'accès à la mémoire distante est plus long et coûteux que l'accès à la mémoire locale. Il faut donc éviter de paralléliser une requête sur des processeurs qui accèdent à des bus différents. Vous le faites en limitant le « *max degree of parallelism* » au nombre de processeurs sur un nœud.

Les **bonnes pratiques** sont donc les suivantes :

- sur des systèmes très sollicités en requêtes courtes, désactivez totalement la parallélisation (`EXEC sp_configure 'max degree of parallelism', 1`) : cela évite que quelques requêtes de *reporting* mobilisent plusieurs processeurs qui auraient été utiles pour d'autres lots en attente ;
- sur des systèmes comportant plus de huit processeurs, limitez « *max degree of parallelism* » à 8. Une parallélisation sur plus de 8 processeurs devient excessivement coûteuse en synchronisation ;
- sur une architecture NUMA, « *max degree of parallelism* » ne doit pas être supérieur au nombre de processeurs sur un nœud NUMA.

Il existe une option de requête, `MAXDOP`, qui produit le même effet de façon localisée. Par exemple :

```
SELECT c.LastName, c.FirstName, e.Title
FROM HumanResources.Employee AS e
JOIN Person.Contact AS c ON e.ContactID = c.ContactID
OPTION (MAXDOP 2);
```

Comme elle l'emporte sur l'option de l'instance, cette option de requête peut être utile pour augmenter le nombre de processeurs défini par « *max degree of parallelism* », pour une requête spécifique.

Pression sur les processeurs

Évidemment, la pression sur les processeurs est montrée par leur niveau d'utilisation, dans le moniteur de performances ou dans le gestionnaire de tâches de Windows (*task manager*).

Process Explorer

Nous vous conseillons cet utilitaire développé par Mark Russinovich fait partie de la panoplie des outils « winternals » disponibles maintenant sur le site de Microsoft. Il vous offre une vision beaucoup plus complète que le gestionnaire de tâches.
http://technet.microsoft.com/en-us/sysinternals/.

Mais une forte utilisation des processeurs ne veut pas dire pression (car ils peuvent travailler beaucoup sans être stressés), et encore moins que le problème vient des processeurs eux-mêmes. La **pression** est montrée par le compteur de performances `System:Processor queue length`, qui indique la file d'attente sur les processeurs. Les compteurs de performances sont visibles dans le moniteur système, détaillé dans la section 5.3

Les vues de gestion dynamique sont aussi de bons indicateurs :

```
SELECT
    SUM(signal_wait_time_ms) as signal_wait_time_ms,
    CAST(100.0 * SUM(signal_wait_time_ms) / SUM (wait_time_ms)
        AS NUMERIC(20,2)) as [%signal (cpu) waits],
    SUM(wait_time_ms - signal_wait_time_ms) as resource_wait_time_ms,
    CAST(100.0 * SUM(wait_time_ms - signal_wait_time_ms) /
        SUM (wait_time_ms) AS NUMERIC(20,2)) as [%resource waits]
FROM sys.dm_os_wait_stats
```

Cela vous donne la raison des attentes dans SQL Server, soit sur le processeur (`%signal (cpu) waits`), soit sur les ressources (`%resource waits`), comme le disque ou les verrous. Un pourcentage important d'attente processeur (en dessus de 20 %) indique une pression sur les CPU. Pour connaître les détails par type d'attente :

```
SELECT wait_type, (wait_time_ms * .001) as wait_time_seconds
FROM sys.dm_os_wait_stats
GROUP BY wait_type, wait_time_ms
ORDER BY wait_time_ms DESC;
```

Vous pouvez aussi voir le nombre de tâches gérées simultanément par un ordonnanceur :

```
SELECT scheduler_id, current_tasks_count, runnable_tasks_count
FROM sys.dm_os_schedulers
WHERE scheduler_id < 255;
```

Ce qui vous donne une idée précise de la charge de travail que doit supporter le système. Un nombre élevé de `runnable_tasks_count`, par exemple 10 sur plusieurs processeurs, peut vous indiquer qu'une mise à jour matérielle (ou une simplification de votre code) est nécessaire. Voici quelques ressources utiles :

- Troubleshooting Performance Problems in SQL Server 2005
  http://www.microsoft.com/technet/prodtechnol/sql/2005/tsprfprb.mspx
- measure CPU pressure
  http://blogs.msdn.com/sqlcat/archive/2005/09/05/461199.aspx

3.1.2 Mémoire vive

Utilisation de toute la mémoire en 32 bits

Dans un environnement 32 bits, l'espace d'adressage est limité à 4 Go. Un processeur 32 bits utilise 32 bits pour positionner chaque octet de mémoire. Chaque bit représentant deux valeurs possibles (O ou 1), une adresse mémoire longue de 32 bits peut donc représenter 2 puissance 32 positions. 2^32 / 1 024 / 1 024 = 4 096 Mo ≈ 4 Go.

Cet espace d'adressage est divisé en deux parties égales. Les deux premiers gigaoctets sont disponibles aux applications, ce qu'on appelle la mémoire utilisateur (*user memory*), les deux suivants sont réservés au système d'exploitation, et constituent la mémoire du noyau (*kernel memory*). Même si votre serveur n'a pas 4 Go de RAM physique, Windows adresse toute la plage en effectuant un mapping d'adresses virtuelles vers les adresses physiques, à l'aide d'un processus appelé VMM, *Virtual Memory Manager*.[1] Le VMM est comme un escroc immobilier qui vend à des clients différents la même parcelle de terre. Grâce à lui, chaque processus en activité voit un espace de 2 Go de RAM.

Sur une machine dédiée à SQL Server, l'activité du système d'exploitation lui-même sera peu importante, et elle ne nécessitera pas les 2 Go de *kernel memory*. Vous avez la possibilité d'ajuster la distribution entre la mémoire utilisateur et la mémoire du système d'exploitation, à l'aide d'une option de chargement du noyau. Dans le fichier `boot.ini`, situé à la racine de votre partition de boot, sur la ligne correspondant à votre système d'exploitation dans la section `[operating systems]`, ajoutez l'option **/3GB**. Exemple :

```
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Windows Server 2003, Enterprise" /
fastdetect /3GB
```

Ce faisant, vous indiquez à Windows de libérer 1 Go supplémentaire pour les applications qui le prennent en charge (elles doivent avoir activé le bit

1. William BOSWELL, *Inside Windows Server 2003*, Addison Wesley, 2003.

IMAGE_FILE_LARGE_ADDRESS_AWARE dans leur en-tête, ce qui est bien sûr le cas de SQL Server), et de dédier 1 Go à la mémoire du noyau. C'est l'option que nous vous recommandons sur une machine dédiée, si vous avez jusqu'à 16 Go de mémoire physique. Au-delà, cette option est à éviter. En effet, dans ce cas Windows a besoin de mémoire supplémentaire pour des tables de page (*page table entries*, ou PTE) destinées à adresser cette mémoire. Si vous laissez l'option */3GB* sur une machine qui possède plus de 16 Go de RAM installés, la mémoire supplémentaire sera simplement ignorée, ce qui n'est sans doute pas le but recherché.

Vous pouvez savoir si */3GB* est activé depuis SQL Server, en interrogeant la vue sys.dm_os_sys_info, colonne virtual_memory_in_bytes :

```
SELECT
    CASE
        WHEN virtual_memory_in_bytes / 1024 / (2048*1024) < 1
        THEN 'Pas activé'
        ELSE '/3GB'
    END
FROM sys.dm_os_sys_info;
```

Bien, mais que faire de la mémoire au-delà de l'espace d'adressage virtuel des 4 Go ? Depuis le Pentium Pro, tous les processeurs Intel 32 bits (famille x86) incluent un mode de *mapping* de mémoire appelé **PAE** (*Physical Address Extension*), qui permet d'accéder à 64 Go. En mode PAE, l'unité de gestion mémoire (MMU, *Memory Management Unit*, un circuit intégré responsable de l'accès à la mémoire et des fonctionnalités de pagination) se voit ajouter des lignes d'adressage supplémentaires qui lui permettent de gérer des adresses sur 36 bits. 2^36 / 1 024 / 1 024 = 65 536 Mo = 64 Go.

Pour activer ce mode PAE sur Windows (à partir de Windows 2000), vous devez ajouter l'option /PAE dans le même fichier boot.ini :

```
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Windows Server 2003, Enterprise" /
fastdetect /PAE
```

Cette option active le chargement d'une image particulière du noyau de Windows (Ntkrnlpa.exe, ou Ntkrpamp.exe sur une machine multiprocesseurs, au lieu de Ntoskrnl.exe).

Ceci permet donc au système d'exploitation d'augmenter sa limite adressable à 64 Go, mais les applications restent limitées par leur propre capacité d'adressage 32 bits. Windows leur montre des espaces de mémoire virtuelle sur 4 Go, différents les uns des autres, qu'elles ne peuvent dépasser. L'utilisation de PAE seule est donc intéressante lorsque plusieurs processus demandent de la mémoire, mais dans notre cas, nous voulons faire profiter le seul processus sqlservr.exe de toute notre mémoire. Nous devons alors utiliser AWE.

AWE (*Address Windowing Extensions*) est un API Windows qui permet aux applications qui l'implémentent d'accéder à plus de mémoire physique que leur espace d'adressage virtuel 32 bits. Comme son nom l'indique, AWE effectue un fenêtrage en régions d'espaces d'adressage, et réserve un plus grand nombre de régions de

mémoire physique de même taille. Le processus de fenêtrage permet ensuite d'attribuer un espace à une région de mémoire, en balayage : toute la mémoire est accessible, mais pas en même temps. AWE est intéressant pour les applications qui manipulent de larges volumes de données… comme un SGBDR.

S'agissant d'une API, il faut donc que le code de l'applicatif l'utilise. Vous devez activer son utilisation dans SQL Server, à l'aide d'une option de configuration du système.

```
sp_configure 'show advanced options', 1
RECONFIGURE
sp_configure 'awe enabled', 1
RECONFIGURE
```

La mémoire allouée par AWE est verrouillée en RAM et ne sera jamais échangée (*swappée*) sur le disque. Elle ne pourra pas non plus être réclamée par le système d'exploitation et est donc idéale pour gérer du cache. SQL Server l'utilise uniquement pour le *buffer*. En d'autres termes, en 32 bits, la mémoire au-delà de la limite des 4 Go ne servira qu'au cache de pages de données. Tous les autres modules (la mémoire de travail, les sessions, les verrous, le cache de plans d'exécution) devront se placer dans l'espace d'adressage virtuel. C'est pour cette raison qu'il est important de choisir l'option `/3GB` quand les conditions le permettent. Notez que AWE est indépendant de PAE : il peut donc placer une partie du *buffer* aussi dans l'espace d'adressage virtuel.

Prise en charge du 64 bits

Depuis la version 2000, SQL Server est livré en deux compilations pour les architectures de processeurs 32 bits (x86) et 64 bits (ia64 maintenant appelé Itanium, x64, et amd64). Le choix d'une architecture 64 bits est de plus en plus intéressant, notamment en rapport coût / performances, et vous devriez aller dans ce sens si vous avez des besoins même moyennement importants. L'avantage, outre une capacité de calcul plus importante des processeurs eux-mêmes, est la quantité de mémoire adressable par un pointeur 64 bits (donc 8 octets). Par défaut, l'espace d'adressage virtuel pour les applications (*user memory*) est de 8 To (tera-octets), ou 7 To sur les processeurs Itanium, ce qui semble suffisant pour les quelques prochaines années. L'avantage ici est net pour SQL Server, et notamment pour la mémoire d'exécution et le cache de plans, qui ne sont plus limités aux 2 ou 3 premiers Go comme en 32 bits.

> **Décisionnel** – Le 64 bits est spécialement intéressant pour les services qui tournent autour de SQL Server, comme Analysis Services ou Reporting Services. Ceux-ci n'utilisent pas AWE et ne peuvent profiter de la mémoire supplémentaire en 32 bits. Ils restent tous en dessous des 4 Go.

Bien entendu, l'option `/3GB` n'a pas de sens en 64 bits. En revanche, et cela peut paraître étonnant, AWE peut se révéler intéressant, pour la raison que nous avons déjà évoquée : comme il verrouille les pages en mémoire vive, elles ne peuvent être

échangées sur le disque. AWE est donc pris en charge en 64 bits, et vous pouvez l'activer pour éviter le *swap*.

> **Clients** – Évitez d'installer des versions 32 bits sur une architecture 64 bits, et surtout les outils clients comme SSMS. Les applications 32 bits tournent dans une couche d'abstraction nommée WOW (*Windows On Windows*), ce qui peut diminuer leurs performances. Si vous voulez installer les outils clients sur votre serveur, installez les versions 64 bits. N'y installez pas BIDS (*Business Intelligence Development Studio*), qui ne possède pas de version 64 bits.

AWE permet aussi parfois un accès plus rapide à la mémoire. Lorsque le système d'exploitation manipule la mémoire, il accède aux entrées de la table de pages (PTE) qui pointent sur les portions de mémoire physique à l'aide d'un numéro appelé le PFN (*page frame number*) si cette portion est en mémoire. Lorsqu'elle est *swappée* sur le disque, la PTE ne contient plus de PFN. Mais le PFN contient certaines données utiles au système, et celui-ci en extrait régulièrement des informations. Chaque accès verrouille la PTE afin d'éviter qu'un autre processus ne la modifie. Windows essaie de rendre ce verrouillage aussi léger que possible (le service pack 1 de Windows Server 2003 a notamment apporté des améliorations de ce point de vue), mais il continue à exister.

La gestion des pages utilisant AWE est simplifiée. Les PTE sont verrouillées à leur acquisition, et le verrou est maintenu pour les protéger contre toute modification d'affectation, ou *swap* sur le disque. Cela améliore donc les performances des opérations d'affectation de mémoire, et accélère la phase d'acquisition de mémoire (*ramp-up*), particulièrement sur une architecture NUMA : nous allons voir pourquoi.

NUMA

NUMA est l'acronyme de *Non-uniform Memory Access*, une architecture matérielle spécifique destinée à optimiser l'accès à la mémoire des systèmes comportant un nombre important de processeurs.

Aujourd'hui, un processeur fonctionne beaucoup plus rapidement que la mémoire vive, et passe donc une partie de son temps à attendre la fin des opérations d'accès à celle-ci. Pour éviter ces attentes, les processeurs intègrent de la mémoire cache. Mais il ne s'agit que d'un pis-aller. L'attente est potentiellement plus importante sur un système comportant beaucoup de processeurs en architecture SMP (*Symmetric Multi-Processing*), car tous les processeurs accèdent à la mémoire à travers le même bus partagé. NUMA offre une architecture où la mémoire est séparée par processeurs, placée sur des bus de données différents, et est donc accessible en parallèle. Pour répondre aux cas où plusieurs processeurs demandent les mêmes données, NUMA intègre des solutions matérielles ou logicielles pour échanger les données entre les différentes mémoires, ce qui peut bien sûr ralentir les processeurs sollicités. Les gains de performance d'un système NUMA sont donc variables, dépendant en général du nombre de processeurs : NUMA devient intéressant à partir de 8 à 12 processeurs, où l'architecture SMP commence à devenir problématique. Pour obtenir un gain de cette architecture, les applications doivent reconnaître NUMA

(être *NUMA-aware*) pour configurer l'exécution de leur *threads* par nœuds NUMA. Les applications qui ne reconnaissent pas NUMA peuvent être ralenties, à cause des différences de temps pour accéder à des parties de mémoires situées sur des bus différents.

Pourquoi AWE est-il intéressant avec NUMA ? Comme le PFN (stocké dans la PTE, comme nous l'avons vu) montre à quel nœud NUMA appartient la page de mémoire, la lecture de cette information (à l'aide de l'API `QueryWorkingSetEx` depuis le Service Pack 1 de Windows Server 2003) nécessite une attribution et une libération de verrou. De plus, cette lecture doit être réalisée périodiquement, afin de prendre en compte les changements d'attribution physique de la page, qui peuvent résulter d'une pagination sur le disque. AWE verrouille les pages en mémoire, ce qui garantit donc qu'elles ne seront jamais paginées. Ainsi, une application *NUMA-aware* n'a besoin de récupérer l'attribution physique d'une page qu'une seule fois, sur de la mémoire AWE.

Soft-NUMA

Pour tester l'implémentation de leur moteur sous NUMA, les équipes de développement de SQL Server 2005 ont bâti une version logicielle de l'architecture NUMA, qui en simule l'architecture matérielle. Ils se sont aperçus que cette implémentation logicielle améliorait les performances sur les systèmes multiprocesseurs SMP, et même NUMA. Cet outil fait à la base pour le test, est donc passé en production sous le nom de soft-NUMA. Si vous avez au moins 8 processeurs, essayez de configurer soft-NUMA pour SQL Server, et comparez les performances. Pour ce faire, vous créez des affinités de processeurs par nœuds soft-NUMA dans la base de registre, dans la clé `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Microsoft SQL Server\90\NodeConfiguration\`. Pour les informations pratiques, reportez-vous aux BOL, section « *How to : Configure SQL Server to Use Soft-NUMA* »

Pression sur la mémoire

SQL Server réagit à la pression sur la mémoire, c'est-à-dire au manque de mémoire lorsque celle-ci est réclamée par des processus. Elle peut libérer des espaces mémoire qui étaient utilisés, par exemple par le cache de données ou le cache de procédures. En 32 bits, le cache de données peut résider dans la mémoire gérée par AWE, mais tout le reste doit être dans l'espace d'adressage virtuel, en dessous de la limite des 4 Go. Vous pouvez déterminer s'il manque de la mémoire pour le cache de données en surveillant les compteurs de performances de l'objet **SQL:Buffer Manager** :

- **Buffer cache hit ratio** – Pourcentage de pages qui ont pu être servies du cache, sans être cherchées sur le disque.
- **Page life expectancy** – Nombre de secondes durant lesquelles une page non utilisée reste dans le cache.
- **Stolen pages** – Nombre de pages « volées » du cache pour être utilisées ailleurs.

Nous détaillons ces compteurs dans la section 5.3.1. `Buffer cache hit ratio` doit être élevé (plus de 97 %). Plus `Page life expectancy` est élevé, mieux c'est :

SQL Server a assez de mémoire pour conserver ses pages dans le cache. Plus le nombre de pages volées est faible et mieux c'est.

Pour observer les caches, vous disposez des objets de compteur `SQL:Cache`, `SQL:Memory`, `SQL:Memory Manager` et `SQL:Plan Cache`. Si vous voyez des fluctuations (à la baisse) du nombre d'objets dans le cache, vous avez affaire à une pression mémoire (sauf si vous avez exécuté des commandes qui vident le cache de plans, comme un attachement de base de données, ou une commande `DBCC`).

Vous pouvez aussi utiliser les vues de gestion dynamique pour surveiller la mémoire. Cette requête vous donne la taille des différents caches :

```
SELECT
    Name,
    Type,
    single_pages_kb,
    single_pages_kb / 1024 AS Single_Pages_MB,
    entries_count
FROM sys.dm_os_memory_cache_counters
ORDER BY single_pages_kb DESC
```

Nous vous renvoyons aux ressources suivantes pour entrer dans les détails :
– Troubleshooting Performance Problems in SQL Server 2005 : http://www.microsoft.com/technet/prodtechnol/sql/2005/tsprfprb.mspx
– Vous trouverez des informations utiles dans cette entrée de blog : http://blogs.msdn.com/slavao/archive/2005/02/19/376714.aspx

3.1.3 Utilisation de la mémoire vive par SQL Server

Les administrateurs s'étonnent souvent de la manière dont SQL Server occupe la **mémoire vive**. Certains semblent s'inquiéter de l'augmentation progressive et sans retour de l'occupation en RAM du processus `sqlservr.exe`, et craignent une fuite de mémoire et une consommation exagérée de ressources. Il s'agit au contraire d'un comportement souhaitable. Il faut comprendre que SQL Server est conçu pour être seul, sans concurrence d'autres applicatifs, sur un serveur. Ainsi, par défaut (vous pouvez limiter la mémoire disponible, nous le verrons plus loin), il s'approprie toutes les ressources raisonnablement disponibles. Si vous lisez ce livre, vous cherchez sans doute à assurer un fonctionnement optimal de SQL Server. Vous ne pouvez le faire réellement qu'en lui dédiant une machine, et en lui permettant de s'attribuer la RAM dont il a besoin. Dans une configuration par défaut, une instance SQL Server ne se limite pas en RAM. Au démarrage, elle calcule la quantité de mémoire réservée pour ses différents éléments, et s'attribue ensuite physiquement (elle « valide ») la mémoire réservée selon ses besoins. Comme le comportement diffère selon les utilisations de la mémoire, nous allons détailler le comportement des éléments importants pour les performances.

SQL Server utilise la RAM pour différents caches, dont les plus importants concernent les pages de données, et le plan d'exécution des requêtes.

Buffer

Le cache de données (appelé le *buffer*, ou *buffer pool*) maintient en mémoire vive autant qu'il est possible de pages de données et d'index. La lecture à partir de la RAM étant environ mille fois plus rapide que sur un disque dur, on en comprend aisément l'intérêt. Ce cache est géré par un composant de SQLOS nommé le **buffer manager**, dont le travail est la lecture des pages du disque vers le *buffer*, et l'écriture des pages modifiées du *buffer* vers le disque.

Lors du démarrage de l'instance, SQL Server calcule et réserve l'espace d'adressage virtuel du *buffer* (la cible de mémoire, ou *memory target*). Il s'agit d'espace réservé, mais la mémoire physique ne sera prise que selon les besoins. La mémoire réellement utilisée est donc inférieure à l'espace réservé tant que les données demandées par les requêtes ne l'auront pas rempli (cette période avant remplissage est appelée le *ramp-up*). Vous pouvez voir la mémoire cible et la mémoire réellement utilisée (en pages de 8 Ko) dans les colonnes `bpool_commit_target` et `bpool_committed` de la vue de gestion dynamique `sys.dm_os_sys_info` :

```
SELECT bpool_commit_target, bpool_committed FROM sys.dm_os_sys_info;
```

Une commande DBCC permet de vider les pages du *buffer* : `DBCC DROPCLEANBUFFERS`. Comme son nom l'indique, seules les pages propres (non modifiées) sont supprimées, les autres attendent un point de contrôle pour être écrites sur le disque. Vous pouvez donc vous assurer de vider le plus de pages possibles en lançant d'abord la commande `CHECKPOINT` :

```
CHECKPOINT;
DBCC DROPCLEANBUFFERS;
```

Lorsque SQL Server a « validé » de la mémoire physique pour le *buffer*, il ne la rend plus au système. Depuis le moniteur système ou le gestionnaire de tâche (colonnes *Memory Usage* et *Virtual Memory Size*), vous verrez le processus `sqlserver.exe` prendre de plus en plus de mémoire virtuelle au fur et à mesure de l'exécution des requêtes, et ne jamais la rendre. Faisons-en la démonstration de l'intérieur de SQL Server.

```
/* nous créons premièrement des instructions de
   SELECT pour chaque table d'AdventureWorks */
SELECT 'SELECT * FROM [' + TABLE_CATALOG + '].[' +
       TABLE_SCHEMA + '].[' + TABLE_NAME + '];'
FROM AdventureWorks.INFORMATION_SCHEMA.TABLES;

/* pages validées du buffer */
SELECT bpool_committed, bpool_commit_target FROM sys.dm_os_sys_info;

-- nous avons copié ici le résultat de la génération de code
SELECT * FROM [AdventureWorks].[Production].[ProductProductPhoto];
/* […] */
SELECT * FROM [AdventureWorks].[Production].[ProductPhoto];
GO

-- pages validées du buffer
```

```
SELECT bpool_committed, bpool_commit_target FROM sys.dm_os_sys_info;
GO

-- vidons le buffer
CHECKPOINT;
DBCC DROPCLEANBUFFERS;
GO

-- pages validées du buffer
SELECT bpool_committed, bpool_commit_target FROM sys.dm_os_sys_info;
GO
```

Les résultats (sur notre portable) sont présentés sur le tableau 3.1.

Tableau 3.1 — Résultats des pages validées du *buffer*

	bpool_committed	bpool_commit_target
avant toute requête :	3776	25600
après les SELECT :	0	0
après avoir vidé le *buffer* :	0	0

SQL Server conserve donc la mémoire validée. Mais les pages sont-elles bien effacées du *buffer* ? Nous pouvons le vérifier à l'aide de la DMV `sys.dm_os_buffer_descriptors` qui détaille chaque page du *buffer* :

```
SELECT page_type, count(*) as page_count, SUM(row_count) as row_count
FROM sys.dm_os_buffer_descriptors
WHERE database_id = DB_ID('AdventureWorks')
GROUP BY page_type
ORDER BY page_type;
```

Après l'exécution de la commande `DBCC DROPCLEANBUFFERS`, cette requête donne bien un résultat vide.

La DMV que nous venons d'utiliser a un intérêt purement informatif. Il n'est pas crucial pour l'amélioration des performances de détailler le contenu du *buffer*. SQL Server effectue un très bon travail par lui-même et n'a pas besoin d'inspection. Mais la curiosité peut parfois vous amener à consulter le *buffer*. Voici une requête qui vous permet de détailler son contenu par tables :

```
USE AdventureWorks;
GO
SELECT object_name(p.object_id) AS ObjectName,
       bd.page_type,
       count(*) as page_count,
       SUM(row_count) as row_count,
       SUM(CAST(bd.is_modified as int)) as modified_pages_count
FROM   sys.dm_os_buffer_descriptors bd
JOIN   sys.Allocation_units a
      ON bd.allocation_unit_id = a.allocation_unit_id
```

```
JOIN   sys.partitions p
      ON p.partition_id = a.container_id
WHERE  bd.database_id = DB_ID('AdventureWorks') AND
       object_name(p.object_id) NOT LIKE 'sys%'
GROUP BY object_name(p.object_id), bd.page_type
ORDER BY ObjectName, page_type;
```

max server memory

Vous pourriez vous interroger : SQL Server ne va-t-il pas s'accorder trop de mémoire, au détriment du bon fonctionnement du système d'exploitation ? Évidemment non, la prise de mémoire est pensée au niveau de l'OS. Par défaut, SQL Server acquiert autant de mémoire possible sans mettre en danger le système. Il utilise pour ce faire l'API de notification mémoire (*Memory Notification API*) de Windows.

En réalité, SQL Server peut rendre au système la mémoire physique utilisé par le *buffer* dans un cas : lorsque, fonctionnant sur une plate-forme 32 bits de version au moins équivalente à Windows XP ou Windows 2003, et lorsque AWE est activé, SQL Server permet une allocation dynamique (*dynamic memory allocation*) du *buffer*.

L'instance continue à acquérir de la mémoire physique jusqu'à ce que la limite indiquée dans l'option de serveur *max server memory*[1] soit atteinte, ou que Windows signale qu'il n'a plus de mémoire disponible pour les applications. Lorsque Windows signale un manque de mémoire vive, SQL Server restitue de la mémoire (si les conditions vues précédemment sont remplies) jusqu'à atteindre la limite inférieure indiquée dans l'option *min server memory*. Cette dernière option ne signifie pas que, au démarrage, SQL Server va immédiatement occuper cette quantité de mémoire. Dans tous les cas, SQL Server acquiert sa mémoire graduellement. Elle donne simplement une limite en dessous de laquelle il n'y aura pas de restitution de mémoire à l'OS.

Faut-il fixer la mémoire ?

Un conseil de performance remontant aux précédentes versions de SQL Server était de fixer la mémoire, c'est-à-dire de donner une valeur identique à *min server memory* et *max server memory*. SQL Server 2005 et 2008 gèrent dynamiquement la mémoire dans SQLOS de façon optimale : il peut être contre performant de fixer la mémoire, et c'est inutile.

Dans le cas d'une allocation dynamique de mémoire, SQL Server ajuste sa consommation selon les disponibilités de la RAM. Autrement dit, si d'autres processus acquièrent et libèrent de la mémoire, SQL Server en rend et en reprend en miroir. SQL Server peut ainsi s'ajuster au rythme de plusieurs Mo par seconde.

1. Ces options sont activables avec `sp_configure` ou graphiquement dans SSMS.

SQL Server 2005, édition Entreprise – Le *ramp-up* est optimisé pour être plus rapide sur les machines ayant beaucoup de mémoire, parce que les requêtes de pages uniques sont converties en requêtes de 8 pages contiguës. En 2008 cette fonctionnalité n'est plus limitée à l'édition entreprise.

3.1.4 Disques

Le choix des **sous-systèmes disque** est bien sûr important. Il est probable que vous soyez de plus en plus dotés de SAN, où les performances sont gérées de façon moins directe que pour des disques locaux, chaque vendeur fournit son outil de gestion.

> Vous trouvez des conseils pour les SAN dans cette entrée de blog en trois parties : http://blogs.msdn.com/sqlcat/archive/2005/10/11/479887.aspx.

Pour le reste, les règles sont simples : les disques doivent être rapides et profiter d'un taux de transfert élevé. C'est moins important pour les disques qui hébergeront les fichiers de données, si la taille des bases est inférieure à la mémoire vive. À ce moment, la plupart des requêtes seront satisfaites par le cache de données (le *buffer*). Autant que possible, choisissez du RAID 10 (ou 01) plutôt que du RAID 5, car les performances en écriture du RAID 5 sont nettement plus faibles. Déplacez autant que possible vos journaux de transactions sur des disques dédiés, ce qui permettra une écriture séquentielle optimale. Dédiez le disque le plus rapide, de préférence un disque local, à `tempdb`. Tout ce qui est dans `tempdb` est jetable : vous n'avez donc pas besoin d'assurer une sécurité importante de ce disque. Un simple miroir pour éviter les interruptions de service suffit.

Vous avez quelques outils à disposition pour mesure la performance de vos disques :

- **IOMeter** et **SQLIO** sont des outils de pur contrôle des performances (*benchmarking*) de disque en ligne de commande. Ils sont libres et téléchargeables sur Sourceforge pour le premier, et sur le site de Microsoft pour le second. Ils vous donnent les performances de vos disques. Vous trouvez des conseils d'utilisation en consultant des entrées de blog et références :
  – http://blogs.msdn.com/sqlcat/archive/2005/11/17/493944.aspx
  – http://sqlblog.com/blogs/linchi_shea/archive/2007/02/21/parse-the-sqlio-exe-output.aspx
  – http://www.microsoft.com/whdc/device/storage/subsys_perf.mspx
- **SQLIOSim** est un outil de pur stress sur le disque, qui se fonde sur le même code d'accès au disque que SQL Server (qui n'a pas besoin d'être installé), pour mesurer la robustesse de votre stockage de masse. Plus d'information dans l'entrée de base de connaissance Microsoft 231619 :
  http://support.microsoft.com/kb/231619

Pression sur le disque

Le compteur de performances (voir section 5.3) à surveiller pour identifier une contention sur les disques est `PhysicalDisk: Avg. Disk Queue Length`. Il indique la taille de la file d'attente, c'est-à-dire les processus qui attendent la libération du

disque déjà utilisé par un autre processus, pour pouvoir lire ou écrire. Si cette valeur est régulièrement en dessus de 2 pour un certain temps, vous avez un problème.

PhysicalDisk: Avg. Disk Sec/Read et **PhysicalDisk:Avg. Disk Sec/Write** sont les temps moyens en secondes nécessaires à une lecture ou une écriture. On considère qu'une moyenne de moins de 10 millisecondes est très bonne, et qu'à partir de 20 millisecondes cela indique une lenteur. Dès 50 millisecondes, il y a une sérieuse contention.

Physical Disk: %Disk Time est le pourcentage de temps où le disque est en activité. Ce nombre devrait rester en dessous de 50.

Ces nombres sont à ajuster sur des disques RAID.

> Le calcul d'ajustement est disponible dans le document *Troubleshooting Performance Problems in SQL Server 2005* : http://www.microsoft.com/technet/prodtechnol/sql/2005/tsprfprb.mspx.

La requête suivante procure les demandes d'entrées/sorties en attente, par fichier de bases de données :

```
SELECT
    vf.database_id,
    vf.file_id,
    vf.io_stall,
    pio.io_pending_ms_ticks
FROM sys.dm_io_virtual_file_stats(NULL, NULL) vf
JOIN sys.dm_io_pending_io_requests pio
ON vf.file_handle = pio.io_handle
```

`io_pending_ms_ticks` est le temps d'attente en millisecondes.

Vous pouvez aussi obtenir des statistiques d'attente totales par fichier :

```
SELECT
    UPPER(LEFT(mf.physical_name, 2)) as disk,
    DB_NAME(vf.database_id) as db,
    mf.name as file_name,
    vf.io_stall,
    vf.io_stall_read_ms,
    vf.io_stall_write_ms
FROM sys.dm_io_virtual_file_stats(NULL, NULL) vf
JOIN sys.master_files mf
    ON vf.database_id = mf.database_id AND vf.file_id = mf.file_id
ORDER BY disk, db, file_name
```

3.1.5 Virtualisation

La **virtualisation** est une technique en plein essor, dont l'objectif est de séparer la couche logicielle de la couche matérielle : un environnement complet, c'est-à-dire le système d'exploitation et les applications qu'il héberge, s'exécute dans une machine virtuelle. Les ressources physiques que cette machine virtuelle utilise sont contrôlées par le logiciel de virtualisation.

Cette architecture permet de déplacer ou dupliquer un environnement, ou de monter en charge très rapidement, sans réinstallation et sans aucune modification de la configuration logicielle. Elle permet également de rationaliser le nombre de machines physiques, en multipliant leur capacité d'accueil. C'est un paradigme qui regroupe les notions de montée en charge verticale et horizontale (*scale up* et *scale out*) en une seule.

La virtualisation permet de mettre en place très rapidement des environnements de test ou de développement, mais elle est de plus en plus utilisée pour héberger de vrais serveurs de production, lorsqu'une bonne performance doit être assurée. Pour ce type de besoin, le logiciel de virtualisation doit s'appuyer sur une architecture matérielle adaptée.

Parmi les quelques outils logiciels de virtualisation sur le marché, les deux plus répandus sont Microsoft Virtual Server et VMware Server (puis XenEnterprise et VirtualIron). Ils sont utiles pour les environnements de test et de développement. Pour obtenir les performances nécessaires à une machine de production, une solution plus puissante comme VMware ESX est requise. Son nom complet est VMware ESX Server, car il s'agit en réalité d'un système d'exploitation en tant que tel. Alors que les logiciels traditionnels de virtualisation s'installent sur un système d'exploitation tel que Windows ou un UNIX, ESX regroupe les deux couches : OS et virtualisation, ce qui permet un niveau de performance largement supérieur. Il comporte également des outils d'administration qui facilitent la gestion centralisée des machines virtuelles, leur clonage et leur migration, notamment.

L'allocation des ressources peut être réalisée dynamiquement. Un module comme VMware DRS (*Distributed Resource Scheduler*) permet d'attribuer les ressources entre les machines virtuelles selon leur besoin, par rapport à des règles prédéfinies.

L'usage de machines virtuelles facilite également la sauvegarde des données : une machine virtuelle peut être sauvegardée directement au niveau du système hôte (ESX lui-même dans ce cas), et un système de basculement automatique peut être configuré à l'aide de l'outil VMware HA (*High Availability*). Une machine virtuelle peut être automatiquement relancée sur un autre serveur faisant partie de la « ferme ».

Nous n'allons pas détailler l'optimisation de serveurs virtualisés, c'est un travail en soi[1]. Nous énumérons simplement quelques règles de base à suivre si vous voulez mettre en place ce type d'infrastructure.

- Vous trouvez sur le site de VMware un outil de *benchmarking* nommé Vmmark.
- Choisissez des processeurs Dual-Core (ou mieux, Quad-Core). Des tests effectués par Dell montrent un gain de performance d'environ 50 % sur des Intel Xeon Dual-Core 32 bits par rapport à des processeurs single-core, et d'autres

1. Si vous vous y intéressez, nous vous conseillons l'ouvrage de Edward Haletky, *VMware ESX Server in the Enterprise*, que vous trouverez dans la bibliographie.

tests sur des Xeon Dual-Core 64 bits vont jusqu'à un gain de performance de 81 % par rapport à des Xeon single-core.

- Une machine virtuelle peut être supervisée exactement comme un serveur physique, les outils et méthodes que nous présentons dans le chapitre 5 sont donc tout à fait valables pour les machines virtuelles, à la différence près qu'une machine virtuelle peut montrer des signes vitaux tout à fait normaux, mais des performances diminuées, parce que d'autres machines virtuelles s'exécutant sur le même matériel consomment trop de ressources.
- Processeurs, mémoire, réseau et disques peuvent être « partitionnés » pour permettre leur utilisation par plusieurs machines virtuelles.

Il est à noter qu'à l'heure actuelle Microsoft ne supporte officiellement qu'un seul produit de virtualisation pour un déploiement de SQL Server en *cluster* : le sien, Virtual Server.

> Vous pouvez consulter l'article de base de connaissance 897615 : *Support policy for Microsoft software running in non-Microsoft hardware virtualization software* (http://support.microsoft.com/kb/897615) pour vous tenir informé de l'évolution de cette politique.

3.2 CONFIGURATION DU SERVEUR

Microsoft essaie de faire en sorte que SQL Server s'autoconfigure autant que possible. Un certain nombre de règles et d'algorithmes sont implémentés pour lui permettre de s'ajuster aux conditions du système hôte. La **configuration** interviendra donc souvent lorsque c'est nécessaire, après un test de charge, plutôt que de façon systématique à l'installation.

Dans des cas où le besoin se fait sentir d'optimiser au maximum, dans les moindres détails, les performances du serveur, quelques possibilités de configuration fine existent :

- En utilisant l'option -x au lancement de SQL Server, vous pouvez désactiver le maintien par l'instance des statistiques d'utilisation des CPU et de taux d'utilisation du cache. Ces statistiques ne sont pas particulièrement coûteuses, mais c'est une option dans les cas où toute optimisation est bonne à prendre. Vous pouvez ajouter cette option à l'aide du *Configuration Manager*, dans la configuration du service SQL, onglet *Advanced*, option *Startup Parameters*.
- Lors de chaque opération de lecture de données, SQL Server vérifie l'intégrité des pages lues. Par défaut, cette vérification se fait à l'aide d'un calcul de somme de contrôle (*checksum*) sur le contenu de la page, qui est ensuite comparé à la somme de contrôle stockée dans l'en-tête de la page à l'écriture des données. Cette opération est également légère et utile pour d'éventuelles (et fort rares) corruptions de données. Vous pouvez néanmoins la désactiver,

ou la remplacer par la méthode « historique » de vérification d'intégrité physique, moins complète mais plus légère, nommée *torn page detection* (un bit est posé à l'écriture de chaque bloc de 512 octets, pour indiquer l'écriture correctement effectuée de ce bloc). Pour une base de données existante, utilisez l'instruction `ALTER DATABASE` :

```
-- pour vérifier le type de vérification de page :
SELECT name, page_verify_option_desc FROM sys.databases;
-- pour désactiver la vérification par CHECKSUM :
ALTER DATABASE [AdventureWorks] SET PAGE_VERIFY NONE;
-- pour remplacer par une vérification par TORN PAGE DETECTION :
ALTER DATABASE [AdventureWorks] SET PAGE_VERIFY TORN_PAGE_DETECTION;
```

4

Optimisation des objets et de la structure de la base de données

Objectif

La toute première étape de l'optimisation est la conception ou *design*, la définition de l'architecture logique de vos bases de données. La qualité du modèle de données influencera tout le reste, et sera un atout ou un handicap pour toute la vie de la base. La modélisation est d'abord logique, faite d'entités et de relations, puis elle est physique, avec l'importance du choix des types de données pour chaque colonne, puisque SQL est un langage fortement typé. Enfin, une étape plus « physique » encore est de préparer soigneusement les structures du serveur : bases de données et tables.

4.1 MODÉLISATION DE LA BASE DE DONNÉES

La plupart des progiciels maintiennent ou *sérialisent*, des informations. Les outils de bureautique traitent évidemment des documents divers contenus dans des fichiers, les applications serveurs doivent souvent communiquer avec des systèmes hétérogènes et s'échangent des informations sous des formes standardisées : protocoles SMTP, HTTP, SOAP… La façon dont chaque éditeur implémente le stockage local de ces informations lui reste toutefois propre. Dans ce monde de concurrence logi-

cielle, les SGBDR occupent une place particulière. Leur charpente, ainsi que bon nombre de détails de leur implémentation font l'objet d'un consensus, bâti soit sur le résultat de recherches (modèle et algèbre relationnels, indexation), soit sur des normes établies (comme la norme SQL). Bien entendu, la manière dont les octets qui composent les données sont organisés sur le disque dépend de chaque éditeur, mais leur structure logique, leur organisation en tables et relations – ce qu'on appelle **le modèle relationnel** – repose sur une base commune. Ce modèle s'est imposé durant ces vingt dernières années à la suite des recherches d'Edgar F. Codd, qui cherchait à rendre les bases de données indépendantes d'une implémentation ou d'un langage particulier.

C'est donc une spécificité des systèmes de gestion de bases de données de vous laisser organiser la structure des informations. La plupart des autres types de progiciels stockent leurs données de manière figée, parce que leurs besoins sont simplement différents. Cette liberté qui vous est laissée est un des éléments qui fait toute la puissance d'un SGBDR, mais non sans conséquences : d'une bonne structuration des données dépendent les performances de tout le système. Nous pouvons même dire que l'optimisation d'une base de données repose en premier lieu sur la qualité de son modèle de données. Il est illusoire d'attendre des miracles de requêtes SQL exécutées sur des données mal organisées. Pour cette raison, nous allons passer du temps sur ce sujet.

4.1.1 Terminologie du modèle relationnel

La modélisation se base sur la création préalable d'un **modèle conceptuel de données** (MCD), une représentation de la structure d'entités et de relations indépendante de toute implémentation physique. Le résultat pourra ensuite être appliqué à n'importe quel SGBDR, moyennant une adaptation à ses particularités d'implémentation, à travers un **modèle physique de données** (MPD). La terminologie utilisée dans le MCD diffère légèrement de celle utilisée dans les implémentations physiques. Pour mémoire, en voici un résumé dans le tableau 4.1.

Tableau 4.1 — Terminologie du modèle relationnel

MCD	MPD	Définition
Type	Type	Le type de données d'un attribut.
Attribut ou propriété	Colonne	Donnée élémentaire à l'intérieur d'une entité.
Entité ou relation	Table	Concept, objet, concert ou abstrait, qui représente un élément identifiable du système à modéliser.
Association	Relation	Un lien entre deux entités.
Tuple	Ligne	Une instance d'entité = une ligne dans une table.

MCD	MPD	Définition
Identifiant	Clé	Attribut ou ensemble d'attributs qui permet d'identifier uniquement chaque *tuple* d'une entité.
Cardinalité	Cardinalité	Décrit la nature d'une association en indiquant, à chaque extrémité, le nombre d'occurrences de *tuples* possible.

La première étape de modélisation, l'étape conceptuelle, gagne à être simple et concrète. Votre modèle doit épouser la réalité. Une bonne façon de s'y prendre est de reproduire dans un schéma les phrases simples décrivant les états et les activités à modéliser. Les noms (sujet et complément) seront repris en entités, et le verbe sera l'association.

Cardinalité

La **cardinalité** représente le nombre d'occurrence, de *tuples*, de lignes. Dans un modèle, elle décrit le nombre d'occurrences possible de chaque côté d'une association. Par exemple, une association entre un contact et ses numéros de téléphone est en général une cardinalité de un-à-plusieurs, car on peut attribuer plusieurs numéros au même contact. Mais cela pourrait être :

- **Un-à-un** – Un seul numéro de téléphone est permis par contact.
- **Un-à-plusieurs** – Nous l'avons dit, plusieurs numéros par contact.
- **Plusieurs-à-plusieurs** – Pour des numéros professionnels communs, au bureau, qui pourraient servir à joindre plusieurs personnes.

Les entités qui possèdent des cardinalités « plusieurs » se voient souvent dotées de discriminants, c'est-à-dire des attributs qui identifient le *tuple* parmi les autres. Par exemple, ce serait ici un type de numéro de téléphone : fixe, mobile, ou privé, professionnel. Ces discriminants pourront ensuite, dans l'implémentation physique, faire partie d'une clé unique si nécessaire.

Choix des clés

Une **clé d'entité** est un attribut, ou un ensemble d'attribut, qui permet d'identifier un *tuple* unique. C'est un concept essentiel de la modélisation, comme de l'implémentation physique relationnelle. Une association un-à-… ne peut se concevoir que si la partie un de l'association est garantie unique. Si cette règle est violée, c'est toute la cohérence du modèle qui est remise en question. De même, de nombreuses règles métiers impliquent l'unicité. Il ne peut y avoir qu'un seul conducteur de camion au même moment, un chèque doit porter un numéro unique et ne peut être encaissé que sur un seul compte bancaire. Un numéro ISBN ne peut être attribué qu'à un seul livre encore disponible, etc.

Dans un modèle, plusieurs types de clés sont possibles :

- **Primaire** – La clé primaire est, parmi les clés candidates, celle qui est choisie pour représenter la clé qui garantit aux yeux du modèle, l'unicité dans l'entité.

Dans sa représentation physique dans SQL Server, chaque ligne doit contenir une valeur, elle ne pourra donc pas accepter de marqueur NULL. Elle sera le plus souvent celle qui sera choisie comme clé étrangère, dans les associations.

- **Candidate** – La clé candidate pourrait avoir été la clé primaire. Elle constitue une autre façon de garantir un *tuple* unique. Elle n'a pas été choisie comme clé primaire soit par malchance pour elle, soit pour des raisons bassement physiques (sa taille par exemple, qui influe sur la taille des clés étrangères et potentiellement de l'index *clustered* – nous verrons ce concept dans le chapitre 6).
- **Naturelle** – Par opposition à une clé technique, une clé naturelle est créée sur un attribut signifiant, qui existe dans le monde réel modélisé par le schéma. Par exemple, un numéro ISBN, un EAN13, un numéro national d'identité, un numéro de téléphone, etc. Le danger de la clé naturelle est son manque potentiel de stabilité, le fait que son format peut changer à la suite d'une décision que vous ne maîtrisez pas, et la possibilité qu'elle soit réattribuée à travers le temps. L'avantage est que la clé elle-même est déjà signifiante.
- **Technique** (*surrogate*) – Une clé technique est une clé en général numérique, créée par défaut d'existence d'une clé naturelle, pour unifier les *tuples*. Elle est souvent implémentée physiquement avec un procédé de calcul d'auto-incrément, comme un séquenceur ou la propriété `IDENTITY` en SQL Server. Elle a souvent tendance à devenir « naturelle » pour les utilisateurs des données, qui parlent alors du client 4567 ou de la facture 3453. Si vous souhaitez vous prévenir de cette « dérive », il est bon de la cacher autant que possible dans les interfaces utilisateur.
- **Intelligente** – Une clé intelligente est une clé qui contient plusieurs informations. Elle permet donc d'extraire des contenus utiles. Par exemple, un numéro ISBN contient le numéro de zone linguistique, et le numéro d'éditeur. Le Numéro national d'identité (numéro de sécurité sociale) donne le sexe ainsi que l'année, le mois, le département et la commune de naissance. Elle peut permettre de gagner du temps dans les recherches, cette logique pouvant être extraite de la clé par programmation, sans recherche dans des tables de référence (qu'il faut créer pour contrôle, mais qu'il ne sera pas toujours utile de requêter dans les extractions). Une clé intelligente peut être générée par le système avec des informations utiles. Par exemple, une clé destinée à unifier un système réparti peut contenir un identifiant de base, de serveur ou de *data center*.
- **Étrangère** – Une clé étrangère est une clé déplacée dans une table fille, correspondant à une clé de la table mère.

Modélisation des relations

Les relations sont exprimées par des **cardinalités**, c'est-à-dire le nombre possible de *tuples* de chaque côté de la relation. On en distingue plusieurs types :

- **Cardinalité identifiante** – La clé primaire d'une table est migrée dans la clé primaire d'une autre table : relation de un-à-un.

- **Cardinalité non identifiante** – La clé primaire est migrée dans un attribut non-clé de la table fille : relation de un-à-plusieurs.
- **Cardinalité optionnelle** – Relation non identifiante avec valeur non nécessaire : relation de zéro-à-plusieurs.
- **Cardinalité récursive** – L'attribut clé est lié à un autre attribut de la même table.
- **Cardinalité plusieurs-à-plusieurs** – Non-identifiant dans les deux sens. En implémentation physique, cette cardinalité nécessite une table intermédiaire.

Lorsqu'une clé est migrée plusieurs fois dans la même table fille, on parle de **rôles** joués par la clé. Il est de bon aloi de préfixer le nom de l'attribut déplacé par son rôle. Par exemple, pour un lien avec un identifiant de personne : `VendeurPersonneID`, et `ClientPersonneID`.

4.1.2 Normalisation

Un bon modèle de données n'exprime une information qu'une seule fois, au bon endroit. Dès que la même information est placée à deux endroits du modèle, sa valeur est compromise… parce qu'elle peut être différente. Si nous pouvons trouver le numéro de téléphone d'un contact dans la table `Contact` et dans la table `Adresse` qui lui est liée, que faire lorsque les deux numéros ne sont pas les mêmes (et croyez-nous, tôt ou tard il y aura des différences) ? Il y a peut-être un numéro erroné, mais lequel ? Ou il s'agit simplement de son numéro privé et de son numéro professionnel… mais lequel ? Toute information doit être précisée, délimitée et unifiée : tel est le but de la normalisation.

Ce qui sous-tend le concept de **normalisation**, est en fin de compte, le monde réel. Il n'y a rien de particulièrement technique, il n'y a pas d'ingénierie particulièrement obscure et sophistiquées dans la modélisation de données, fort heureusement. La plupart des tentatives de sophistication, surtout si elles entraînent le modèle à s'éloigner de la réalité, sont des errements malheureux. Que cela signifie-t-il ? Qu'un modèle de données relationnel est formé d'entités et de relations : des tables, et des relations entre ces tables. Chaque entité décrit une réalité délimitée, observable dans le champ modélisé par la base de données. Il suffit d'observer autour de soi, de comprendre la nature élémentaire des processus à l'œuvre, pour dessiner un modèle de données. Avons-nous des clients, des magasins, des employés, des produits ? Nous aurons donc les tables `Client`, `Magasin`, `Employe`, `Produit`. Un employé vend-il à ses clients des produits dans un magasin ? Nous aurons donc une table de vente, qui référence un vendeur, un magasin, un client, un produit, à une date donnée, avec un identifiant de vente, peut-être un numéro de ticket de caisse. Mais, vend-il plusieurs produits lors de la même vente ? Nous aurons donc à supprimer l'identifiant de produit dans la table de ventes, et à créer une table de produits vendus, liée à la vente. L'employé est-il toujours dans le même magasin ? Sans doute, alors retirons l'identifiant de magasin de la vente, et ajoutons l'identifiant du magasin dans la table des employés. Et si l'employé peut changer de magasin ? Créons alors une table intermé-

diaire qui lie l'employé au magasin, avec – c'est une bonne idée – des dates de validité. Vous trouverez sur la figure 4.1 un modèle très simple de ces entités.

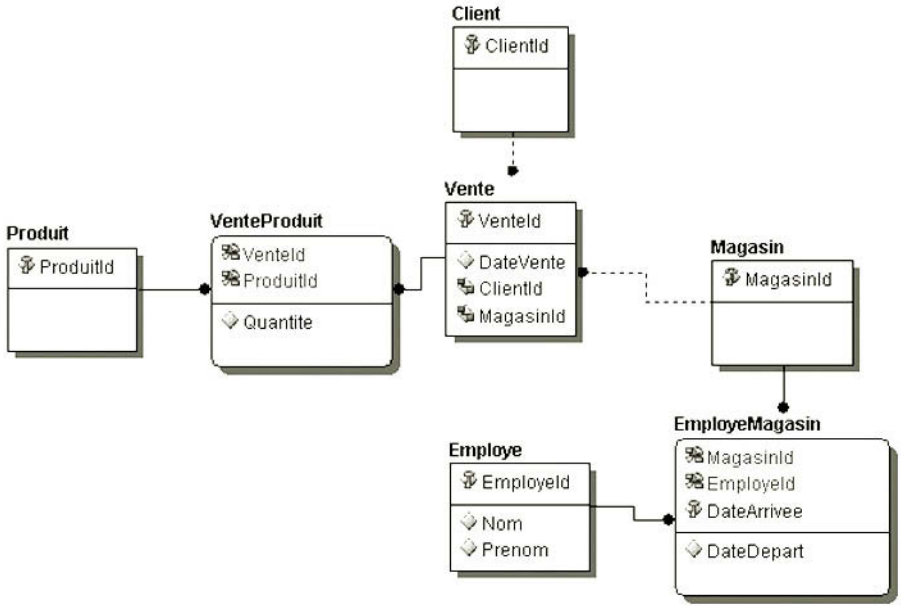

Figure 4.1 — Premier modèle Commerce

Règles métier

Puisque nous abordons l'idée de dates de validité, vous devez être extrêmement attentif lors de la phase de modélisation aux règles métier (*business rules*). Elles doivent être placées autant que possible dans la structure. Les règles complexes pourront être implémentées en déclencheurs (*triggers*).

En résumé, il faut stocker autant d'informations utiles possibles, pour permettre de retrouver des données qui ont du sens et de la valeur, et, par conséquent, placer l'information au bon endroit, c'est-à-dire dans la bonne entité, celle à laquelle elle appartient naturellement, en respectant un principe important : l'économie. Cela veut dire, éviter la duplication de données, qui alourdit le modèle, conceptuellement et physiquement, et diminue les performances.

La granularité des données est aussi à prendre en compte : désirez-vous des données détaillées (tout un ticket de caisse, produit par produit), ou seulement des totaux ? En général, en application OLTP, les détails sont conservés : vous en aurez forcément besoin un jour. Dans une application OLAP, on ne garde parfois que des agrégats, par exemple des totaux par jour.

Formes normales

Normaliser un modèle de données veut dire le rendre conforme à des règles nommées « formes normales ». Ce qu'on appelle les **formes normales** sont des lois importantes de structuration du modèle, qui permettent de s'assurer que celui-ci évite les pièges les plus élémentaires, en assurant une base de conception favorable à un stockage utile et optimal des données. Par utile nous entendons favorable à l'extraction des données désirées, dans la plupart des cas de figures. Pour être complet, les formes normales principales sont au nombre de six. Toutefois, les trois premières sont les plus importantes et les plus utiles dans la « vie réelle » de la modélisation de données. Chaque forme normale complète les formes précédentes. Il est donc nécessaire de les appliquer dans l'ordre.

Première forme normale (1FN)

Tout attribut doit contenir une valeur atomique, c'est-à-dire que toutes ses propriétés doivent être élémentaires. En clair, cela signifie que vous utiliserez une colonne par pièce d'information à stocker, et qu'en aucun cas vous ne vous permettrez de colonnes concaténant des éléments différents d'information. Cette règle est importante pour deux raisons : elle va vous permettre toute la souplesse nécessaire à la restitution et à la manipulation des données, et elle va vous permettre, bien plus en aval, de vous assurer de bonnes performances. Créer le modèle d'une base de données est aussi un exercice de prudence et d'anticipation. Ne prenez pas de décision à la légère. Si vous estimez par exemple qu'une ligne d'adresse peut être contenue dans une seule colonne (`'12, boulevard de Codd'`, par exemple), êtes-vous sûr que vous n'aurez jamais besoin – donc que personne ne vous demandera jamais – d'un tri des adresses par rue et numéro de rue ? Si vous modélisez la base de données d'un service de gestion des eaux, ne voudrez-vous pas fournir aux employés chargés de relever les compteurs une liste des habitants, dans l'ordre du numéro d'immeuble de la rue ? Comment ferez-vous sans avoir isolé ce numéro dans un attribut (une colonne) dédié ? Comme beaucoup, vous allez vous échiner à récupérer ce numéro avec des fonctions telles que `SUBSTRING()`, en prenant en compte toutes les particularités de saisie possibles. Vous n'avez alors plus de garantie de résultats corrects, seulement celle d'une baisse de performances. En résumé, toute valeur dont vous savez, ou soupçonnez, que vous allez devoir la manipuler, l'afficher, la chercher individuellement devra faire l'objet d'un attribut, donc être stockée dans sa propre colonne. Ne créez des colonnes contenant des types complexes que lorsque leur relation est indissociable.

Attributs de réserve

Une autre réflexion : ne créez pas d'attributs inutiles. Créer des colonnes réservées à un usage ultérieur, par exemple, n'a aucun sens. Vous alourdissez le modèle et la taille de vos tables pour rien. Les versions modernes des SGBDR comme SQL Server permettent très facilement d'ajouter ou de supprimer des colonnes après création de la table. Même lorsque les objets font parties d'une réplication, il est devenu aisé, à partir de SQL Server 2005, de répliquer aussi les changements de structure. Il n'y a donc aucune justification rationnelle à créer des colonnes réservées.

Deuxième forme normale (2FN)

L'entité doit respecter la 1FN. Tous les attributs doivent être un fait concernant la clé tout entière, et non pas un sous-ensemble de la clé. Cette forme normale concerne les entités dont la clé est composite. Dans ce cas, chaque attribut doit dépendre de tous les attributs formant la clé. Par exemple, dans notre exemple de magasin, l'entité `VenteProduit` représente un produit vendu lors d'un acte de vente. Sa clé est composite : une clé de produit, une clé de vente. Nous voulons connaître le prix de vente unitaire du produit, ainsi que le prix d'achat au fournisseur, pour connaître notre marge. Où devons-nous mettre ces attributs ? Sur la figure 4.2, ils ont été placés dans l'entité `VenteProduit`.

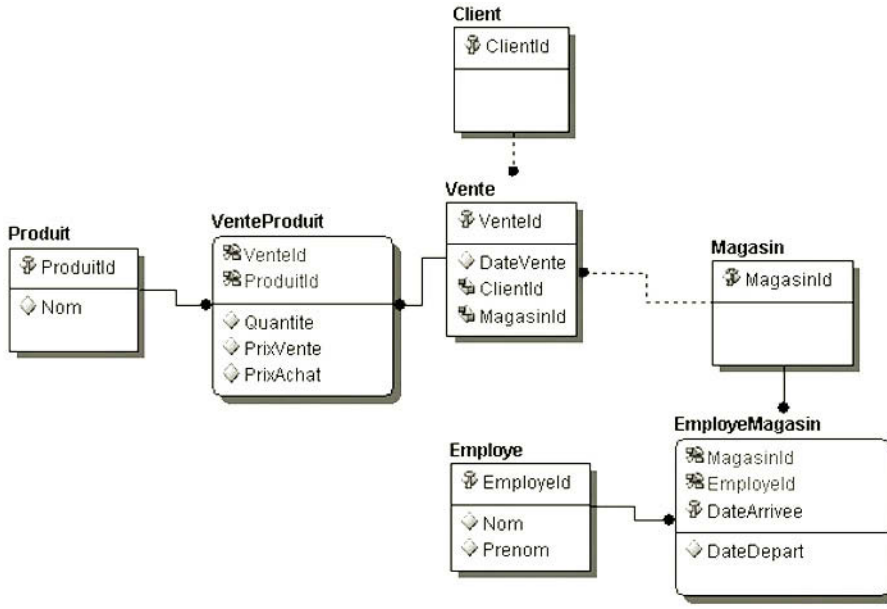

Figure 4.2 — Deuxième modèle Commerce

Est-ce logique ? Le prix d'achat au fournisseur dépend-il de toute la clé, c'est-à-dire du produit et de la vente ? Certainement pas : le prix d'achat ne dépend pas de la vente effectuée, à moins que nous gérions notre stock en flux tendu, et que nous commandions le produit chez notre fournisseur à chaque commande du client. À ce moment, peut-être négocions-nous un prix particulier selon la quantité commandée ? Alors, cela aurait peut-être du sens de placer le prix d'achat dans cette entité. Mais dépendrait-il réellement de la vente ? Si nous réalisons plusieurs ventes le même jour, et que nous effectuons une commande groupée auprès du fournisseur, il s'agit alors plus d'une notion de commande fournisseur, qui peut être liée à la vente, comme par exemple sur la figure 4.3.

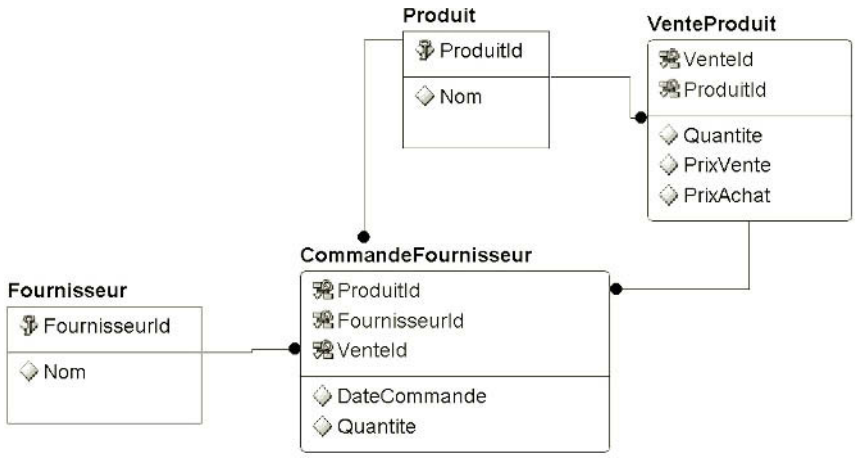

Figure 4.3 — Troisième modèle Commerce

Mais en faisant cela, nous rendons la clé de la vente, partie de la clé de la commande, ce qui signifie que nous ne pouvons créer de commande que pour une seule vente. Nous devons donc créer une entité intermédiaire, comme à la figure 4.4.

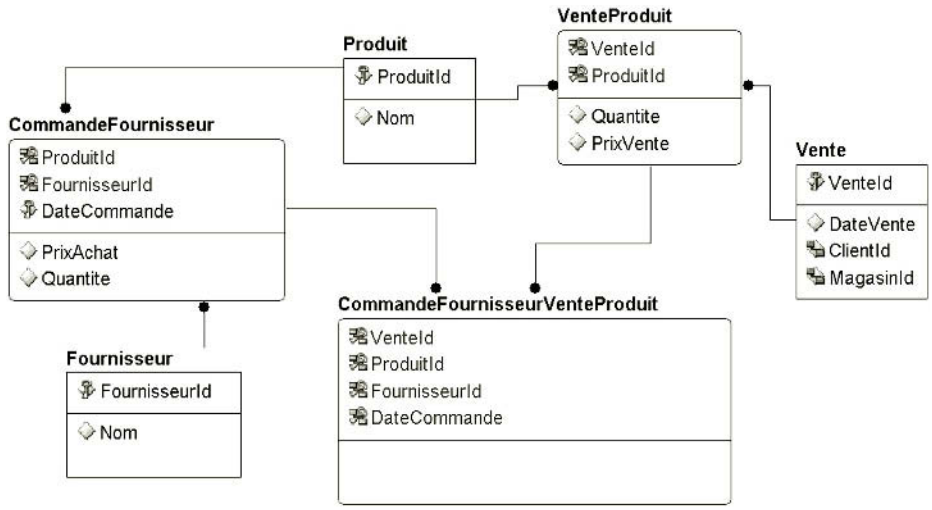

Figure 4.4 — Quatrième modèle Commerce

La clé qui compose l'entité `CommandeFournisseurVenteProduit` devient longue. C'est un problème si elle doit être répercutée dans des tables filles en tant que clé étrangère, et c'est un atout si nous devons identifier rapidement à partir de cette entité, dans le code, la référence de nos entités liées.

Prévision

Dans la phase de modélisation conceptuelle, on ne réfléchit pas à l'utilité des clés par rapport aux requêtes d'extraction. Dans la modélisation physique, il devient utile, voir nécessaire de le faire, pour intégrer les performances des requêtes déjà au niveau de la modélisation – c'est l'objet de cet ouvrage. Nous pensons qu'il est très utile d'avoir en tête les facilités apportées à nos requêtes, et à notre recherche d'information, pour intégrer ces besoins et ces contraintes dans notre modélisation physique.

Malheureusement, ce modèle est loin d'être parfait. Nous avons, notamment, deux quantités : une quantité de produits commandés, et une quantité de produits vendus. Ne peut-on pas déduire le nombre de produits commandés en additionnant le nombre de produits vendus ? Si oui, il est important de supprimer l'attribut `Quantite` de l'entité `CommandeFournisseur`. Pourquoi ? Car si nous la conservons, la question sera : où aller chercher l'information ? Si plus tard, à l'aide d'une requête, nous trouvons une différence entre la quantité de produits commandés et la somme des produits vendus, quelle est la quantité qui nous donne ce qui s'est vraiment passé dans la réalité ?

Par contre, si nous n'avons que l'attribut `Quantite` de l'entité `VenteProduit`, que se passe-t-il si une vente est annulée par le client après sa passation de commande, ou si le client modifie sa quantité ? Comment savoir quelle est la quantité de produits réellement commandée ? Ici, le modèle doit être complété : si le client modifie la quantité de produits voulus, il nous faut un moyen d'en conserver l'historique, peut-être en amendant la vente, ou en créant une deuxième vente liée à la première, par une clé récursive. Et si la vente est annulée, il nous faut certainement également un moyen d'exprimer dans notre schéma ce qu'il advient des produits commandés, afin de ne pas en perdre la trace, et de pouvoir les attribuer à des ventes ultérieures, pour gérer efficacement notre stock.

Troisième forme normale (3FN)

L'entité doit respecter la 2FN. Tous les attributs doivent être des faits concernant directement la clé, rien que la clé. Ils ne doivent en aucun cas dépendre d'un attribut non-clé. Dans notre exemple précédent, si l'entité `Vente` contient un attribut `magasin`, peut-on dire qu'il viole la 3FN ? Douteux parce que la vente a bien lieu dans un magasin ? Certes, les deux informations sont liées, mais le magasin dépend-il de la vente (la clé), et seulement de la vente ? Non, parce que dans le monde que nous modélisons, le magasin dépend aussi d'un attribut non-clé : l'employé, car nous savons que cet employé est attribué à un magasin. Il viole donc la 3FN. En ajoutant des attributs qui dépendent d'un attribut non-clé, nous sommes certains d'introduire de la redondance dans le modèle. Ici, l'identifiant de magasin n'a besoin d'être écrit qu'une seule fois, dans l'entité `Employé`, et non pas à chaque vente.

Un signe courant et évident de violation de la 3NF, est la présence dans une entité d'attributs numérotés. Des attributs tels que `Adresse1`, `Adresse2`, `Adresse3` peuvent à la rigueur être douloureusement acceptés, mais `AdressePrivee1`, `AdressePrivee2`, `AdressePrivee3`, `AdresseProfessionnelle1`, `AdresseProfessionnelle2`, `AdresseProfessionnelle3`, est clairement un problème. Il

est temps de créer une entité `Adresse`, avec un attribut discriminant, spécifiant le type d'adresse. La création de « séquences » d'attributs est un cercle vicieux : vous avez l'impression d'y mettre un doigt, et bientôt vous y avez enfoncé le bras tout entier. À chaque besoin nouveau, vous vous retrouvez à ajouter une nouvelle colonne, avec un incrément de séquence. Que ferez-vous lorsque vous en serez arrivé à la colonne `Propriete20`, ou `Telephone12` ? Modélisez votre base correctement, ou si vous devez maintenir un modèle existant, « refactorez ».

Refactoring

Le concept de *refactoring* indique les techniques de modification du code tout au long de son existence. Il faut, en matière de modèle de données, songer à modifier la structure lorsque des nouveaux besoins se font sentir. Il faut résister à l'envie de coder en dur des modifications de logique, pour éviter de récrire ses requêtes après modification de schéma. Adapter son schéma est plus long, mais bien plus solide et évolutif.

Lorsque la base est normalisée, il est parfois utile de **dénormaliser**, c'est-à-dire de dupliquer au besoin – et uniquement lorsque c'est absolument nécessaire – le contenu d'une colonne. La plupart du temps, dénormaliser n'est pas nécessaire. Souvent les dénormalisations sont faites avant même de tester les performances des requêtes, ou pour résoudre des problèmes de performances qui proviennent d'une mauvaise écriture de code SQL. Même si la dénormalisation se révèle utile, des mécanismes de SQL Server comme les colonnes calculées ou les vues indexées permettent de dupliquer logiquement les données sans risque d'incohérences. L'autre solution pour dénormaliser consiste à créer des colonnes physiques qui vont contenir des données provenant d'autres tables, et qui sont maintenues par code, en général par déclencheur ou procédure stockée. Cette approche comporte quelques inconvénients : ajouter du code augmente les risques d'erreur, diminue les performances générales et complique la gestion de la base. Nous vous conseillons en tout cas vivement, dans ce cas, d'utiliser une méthodologie simple, claire et consistante. Créez par exemple des déclencheurs, avec une convention de dénomination qui permet de les identifier rapidement, et optimisez leur code.

Une chose en tout cas est importante : la dénormalisation n'est pas la règle, mais tout à fait l'exception. Elle ne doit être menée qu'en dernier ressort. En tout cas, comme son nom l'indique, elle consiste à modifier un schéma déjà normalisé. En aucun cas, elle ne peut être une façon préliminaire de bâtir son modèle.

Logiquement, plus votre modèle est normalisé, plus le nombre de tables augmente, et moins vous avez de colonnes par table. Cela a de nombreux avantages de performance : vous diminuez le nombre de données dupliquées, donc le volume de votre base, vous réduisez le nombre de lectures nécessaires pour extraire un sous-ensemble de données (on cherche en général à récupérer une partie seulement des lignes et colonnes, pas une table tout entière), vous augmentez la possibilité de créer des index *clustered* (plus sur ce concept dans le chapitre 6), et donc la capacité à obtenir des résultats triés sans des opérations de tri coûteuses et à générer des plans

d'exécution utilisant l'opérateur de jointure de fusion (*merge join*), un algorithme de jointure très rapide, etc.

Citoyens, n'ayez pas peur de la normalisation ! Ne tombez pas dans le piège consistant à croire que l'augmentation du nombre de tables va rendre votre code SQL compliqué, et va faire baisser les performances. Créer des tables qui ne respectent pas au moins les trois premières formes normales vous entraînera dans une multitude de problèmes avec lesquels vous serez obligé de vivre quotidiennement. Prenez dès le début de bonnes habitudes. En ce qui concerne la complexité, vous pouvez la dissimuler par la création de vues, qui sont, du point de vue du modèle relationnel, conceptuellement identiques aux tables.

> Pour aller plus loin, nous vous conseillons de vous procurer un bon ouvrage sur la modélisation relationnelle, comme par exemple « UML 2 pour les bases de données », de Christian Soutou.

Modèle OLAP

En 1993, Edward F. Codd publia un livre blanc nommé « *Providing OLAP (On-line Analytical Processing) to User-Analysts: An IT Mandate* », dans le but de préciser les exigences d'applications destinées à l'analyse de vastes volumes de données, ce qu'on appelle aujourd'hui de plusieurs termes : systèmes d'aide à la décision (*decision support system*, DSS), *business intelligence*, décisionnel, etc. Malgré la légère polémique autour de cet article, commandité par Hyperion (l'entreprise s'appelait alors Arbor Software) donc peut-être partiel, sans doute plus écrit par ses collaborateurs que par Codd lui-même, et en tout cas non supporté par Chris Date, l'associé de Edward Codd, l'écrit popularisa le terme OLAP, par opposition à OLTP. Un système **OLTP** (*OnLine Transactional Processing*) est en perpétuelle modification, les opérations qui s'y exécutent sont encapsulées dans des transactions, ce qui permet un accès multi-utilisateurs en lecture comme en écriture. C'est pour ce genre de système que le modèle relationnel, les douze lois de Codd (qui sont en réalité au nombre de treize) et les formes normales furent conçus.

Un système **OLAP** (*OnLine Analytical Processing*) est utilisé principalement en lecture, pour des besoins d'analyse et d'agrégation de larges volumes de données. Alors que les requêtes OLTP se concentrent en général sur un nombre réduit de lignes, les requêtes analytiques doivent parcourir un grand ensemble de données pour en extraire les synthèses et les tendances. Les systèmes OLAP sont en général implémentés en relation avec des entrepôts de données (*data warehouses*) parfois constitués en « succursales » nommées *data marts*. Ces entrepôts centralisent toutes les données (opérationnelles, financières, etc.) de l'entreprise (ou du groupe) – souvent avec une perspective historique – provenant de tous les services, pour offrir une vision globale de l'activité et des performances (figure 4.5).

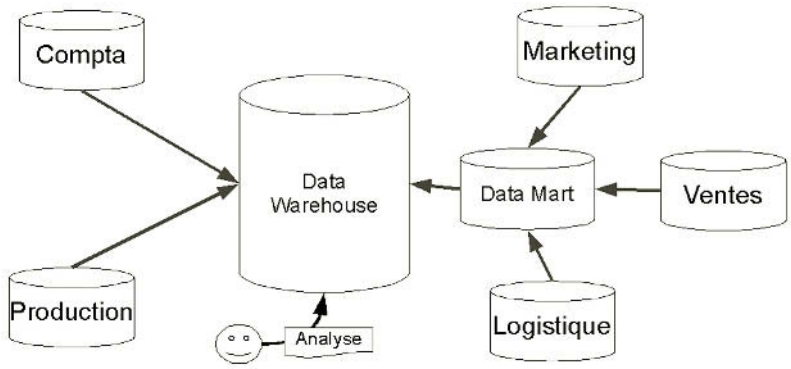

Figure 4.5 — Les entrepôts de données

Pour faciliter l'analyse des données, une représentation dite multi-dimensionnelle, sous forme de « cubes », s'est développée. Il s'agit de présenter les indicateurs chiffrés (données de ventes, quantités, prix, ratios, etc.), qu'on appelle aussi **mesures**, en les regroupant sur des axes d'analyse multiples, nommés **dimensions**. Les agrégations (regroupements) sont effectuées à travers ces dimensions. On peut par exemple vouloir afficher les données de ventes par mois, par types de produits, sur les magasins d'un département donné, pour un fournisseur. Le modèle relationnel normalisé n'est pas optimal pour cette forme de requêtes sur des volumes très importants de données. De cette constatation, deux modèles principaux propres à l'OLAP se sont développés : le modèle en étoile et le modèle en flocon.

Le **modèle en étoile** (*star schema*, figure 4.6), fortement dénormalisé, présente une table centrale des indicateurs (une table de faits, ou *fact table*), qui contient outre les mesures les identifiants de liaison à chaque dimension. Les tables de dimensions « entourent » la table de faits. Elles peuvent contenir des colonnes fortement redondantes. On peut par exemple, dans une dimension client, trouver des colonnes `Ville` et `Pays` qui contiennent les noms de lieu en toutes lettres, dupliqués des dizaines ou des centaines de milliers de fois. Les erreurs de saisie ne peuvent pas se produire, puisque ces tables sont alimentées exclusivement par des processus automatiques, et les lectures en masse sont facilitées par l'élimination de la nécessité de suivre les jointures. La volumétrie n'est pas non plus un problème, ces systèmes étant bâtis autour de solutions de stockage adaptées.

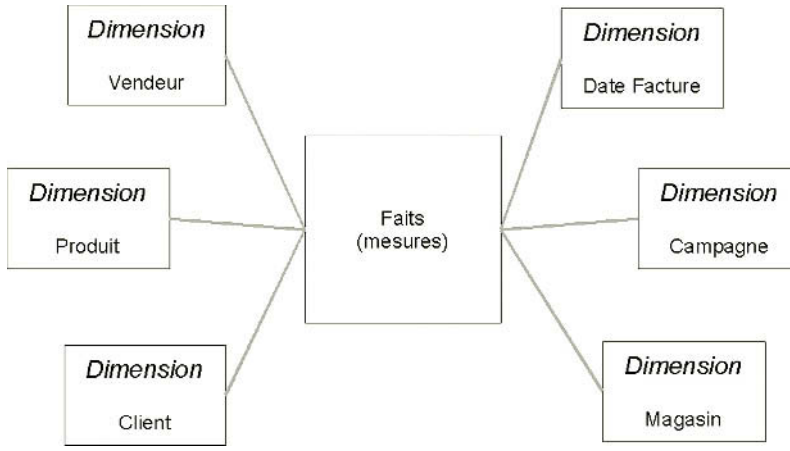

Figure 4.6 — Le modèle en étoile

Le **modèle en flocon** (*snowflake schema*, figure 4.7) est une version du modèle en étoile qui conserve, sur certaines dimensions, des relations dans le but de limiter le volume de stockage nécessaire. Il peut être utilisé si la quantité de données dupliquées est vraiment trop importante. En général, un modèle en étoile est préférable.

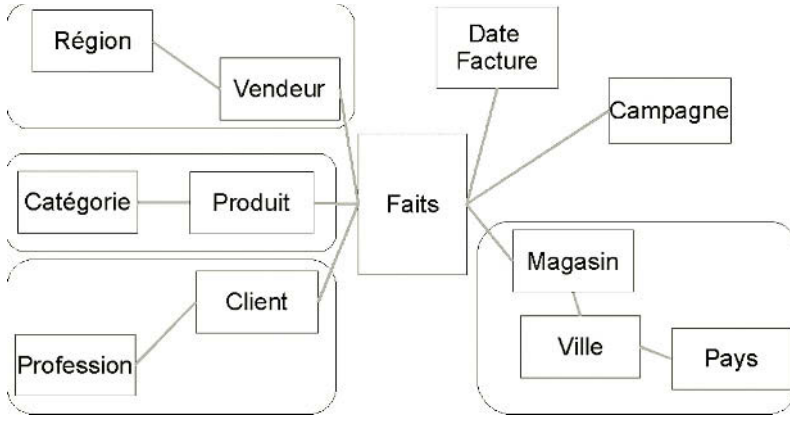

Figure 4.7 — Le modèle en flocon

Cet ouvrage se concentre sur l'optimisation d'un système OLTP. Nous tenions toutefois à vous présenter les fondamentaux de l'OLAP afin de vous donner l'orientation nécessaire au cas où vous auriez à modéliser un entrepôt de données.

> Pour aller plus loin sur le sujet, un livre fait référence : « *Entrepôts de données. Guide pratique de modélisation dimensionnelle* », 2e édition, de Ralph Kimball et Margy Ross, aux éditions Vuibert informatique (édition anglaise : *The Data Warehouse*

Toolkit: The Complete Guide to Dimensional Modeling, chez Wiley).
Ralph Kimball est un des deux théoriciens importants du décisionnel moderne, et les principes de ce livre ont été appliqués dans les outils BI de SQL Server.

4.1.3 Bien choisir ses types de données

Lors du passage au modèle physique de données (MPD), c'est-à-dire lors de l'adaptation du MCD à une implémentation de SGBDR particulière – SQL Server en ce qui nous concerne –, plusieurs considérations sont importantes pour les performances :

Choisissez les bons types de données. Nous avons vu que la 1FN prescrit de dédier chaque attribut à une valeur atomique. Physiquement, il est également important de dédier le type de données adapté à cette valeur. Par exemple, si vous devez exprimer un mois de l'année, faut-il utiliser un type `DATE` (ou `DATETIME` en SQL Server 2005), ou un `CHAR(6)`, dans lequel vous inscrirez par exemple `'200801'` pour le mois de janvier 2008 ? Si l'idée vous vient d'utiliser un `CHAR(6)`, que ferez-vous lorsqu'il vous sera demandé de retourner par requête la différence en mois entre deux dates ? Si vous avez opté dès le départ pour le type qui convient, la fonction `DATEDIFF()` est tout ce dont vous avez besoin. De plus, le type `DATE` est codé sur trois octets, alors qu'un `CHAR(6)`, par définition, occupe le double. Sur une table de dix millions de lignes, cela représente donc un surpoids d'environ trente mégaoctets, totalement inutile et contre performant.

Prévoir la volumétrie

À la mise en place d'une nouvelle base de données, estimez sa volumétrie future, pour dimensionner correctement votre machine, vos disques et la taille de vos fichiers de données et de journal. Vous pouvez utiliser les fonctionnalités intégrées de dimensionnement des outils de modélisation comme Embarcadero ER/Studio ou Sybase PowerDesigner/PowerAMC. HP publie aussi des outils de *sizing* : http://activeanswers.compaq.com/ActiveAnswers/Render/1,1027,4795-6-100-225-1,00.htm. Vous pouvez aussi créer votre propre script, par exemple fondé sur la colonne `max_length` de la vue `sys.columns`, en prenant en compte que la valeur de `max_length` pour un objet large est -1.

Une colonne comportant un type de taille trop importante augmente la taille de la ligne. Il s'ensuit que moins de lignes peuvent résider dans la même page, et plus de pages doivent être lues, ou parcourues, pour obtenir le même jeu de données. Il est important, dans ce cadre, de faire la distinction entre types précis et types approchés, types de taille fixe ou de taille variable, ainsi qu'entre les types qui restent dans la page et ceux qui peuvent s'en échapper. Passons en revue quelques types de données système de SQL Server.

Numériques

Les **types numériques** sont soit entiers, soit décimaux, à virgule fixe ou flottante. Les types entiers sont codés en un bit (`BIT`), un octet (`TINYINT`), deux octets (`SMALLINT`), quatre octets (`INT`) ou huit octets (`BIGINT`). Le **`BIT`** est utilisé en général pour exprimer des valeurs booléennes (vrai ou faux). Il accepte le marqueur `NULL`. Le stoc-

kage des colonnes de type BIT est optimisé par le regroupement de plusieurs de ses colonnes dans un seul octet de stockage. Ainsi, si vous avez de une à huit colonnes de type BIT, SQL Server utilisera un octet dans sa page de données, de neuf à seize, deux octets, etc. Vous pouvez donc créer plusieurs de ces colonnes sans inquiétude. Notez que les chaînes de caractères TRUE et FALSE sont convertibles en BIT, explicitement ou implicitement, ce qui est utile pour reproduire dans son code un comportement proche d'un type booléen. Si vous pouvez travailler directement en 0 ou 1, faites-le, cela évitera la phase de conversion :

```
-- ok
SELECT CAST('TRUE' as bit), CAST('FALSE' as bit)

SET LANGUAGE 'french'
-- erreur
SELECT CAST('VRAI' as bit), CAST('FALSE' as bit)

-- ok
DECLARE @bool bit
SET @bool = 'TRUE'
SELECT @bool
```

Une autre option est de créer une colonne de type entier ou **BINARY**, et y stocker des positions 0 ou 1, bit par bit (en créant une colonne utilisée comme bitmap, champ de bits). Les opérateurs T-SQL de gestion des bits (*bitwise operators*) permettent de manipuler chaque position. C'est un moyen efficace de gérer un état complexe, car il permet d'exprimer à la fois des positions particulières, et une vue générale, indexable, de la situation de la ligne. Une recherche sur un état complexe pourra profiter de cette approche. En revanche, une recherche impliquant seulement une position de bits sera moins efficace. Prenons un exemple :

```
USE tempdb
GO

CREATE TABLE dbo.PartiPolitique (
       code char(5) NOT NULL PRIMARY KEY CLUSTERED,
       nom varchar(100) NOT NULL UNIQUE,
       type int NOT NULL DEFAULT(0)
)
GO

INSERT INTO dbo.PartiPolitique (code, nom)
SELECT 'PCF', 'Parti Communiste Français' UNION ALL
SELECT 'MRC', 'Mouvement Républicain et Citoyen' UNION ALL
SELECT 'PS', 'Parti Socialiste' UNION ALL
SELECT 'PRG', 'Parti Radical de Gauche' UNION ALL
SELECT 'VERTS', 'Les Verts' UNION ALL
SELECT 'MODEM', 'Mouvement Démocrate' UNION ALL
SELECT 'UDF', 'Union pour la Démocratie Française' UNION ALL
SELECT 'PSLE', 'Nouveau Centre' UNION ALL
SELECT 'UMP', 'Union pour un Mouvement Populaire' UNION ALL
SELECT 'RPF', 'Rassemblement pour la France' UNION ALL
SELECT 'DLR', 'Debout la République' UNION ALL
```

```
SELECT 'FN', 'Front National' UNION ALL
SELECT 'LO', 'Lutte Ouvrière' UNION ALL
SELECT 'LCR', 'Ligue Communiste Révolutionnaire'
GO

CREATE INDEX nix$dbo_PartiPolitique$type
ON dbo.PartiPolitique ([type])
GO

-- à droite
UPDATE dbo.PartiPolitique
SET type = type | POWER(2, 0)
WHERE code in ('MODEM', 'UDF', 'PSLE', 'UMP', 'RPF',
               'DLR', 'FN', 'CPNT')

-- à gauche
UPDATE dbo.PartiPolitique
SET type = type | POWER(2, 1)
WHERE code in ('MODEM', 'UDF', 'PS', 'PCF', 'PRG', 'VERTS', 'LO', 'LCR')

-- antilibéral
UPDATE dbo.PartiPolitique
SET type = type | POWER(2, 2)
WHERE code in ('PCF', 'LO', 'LCR')

-- au gouvernement
UPDATE dbo.PartiPolitique
SET type = type | POWER(2, 3)
WHERE code in ('UMP', 'PSLE')

-- qui est au centre ?

-- scan
SELECT code
FROM dbo.PartiPolitique
WHERE type & (1 | 2) = (1 | 2)

-- seek... mais incorrect
SELECT code
FROM dbo.PartiPolitique
WHERE type = 3
```

Les deux dernières requêtes utilisent deux stratégies de recherche différentes. Elles produisent un plan d'exécution très différent également, car la première requête ne peut pas utiliser d'index pour l'aider. Reportez-vous à la section 6.2 traitant du choix des index, pour plus de détails.

Vous pouvez aussi créer une table de référence indiquant quelle est la valeur de chaque bit, et gérer dans votre code des variables de type `INT` pour exprimer lisiblement quelles sont les valeurs de chaque bit. Mais dans cet ouvrage dédié aux performances, nous devons préciser que ces méthodes sont souvent une facilité de code qui diminue la performance des requêtes.

En règle générale, une clé technique, auto-incrémentale (IDENTITY) ou non, sera codée en INT. L'entier 32 bits permet d'exprimer des valeurs jusqu'à 2^31-1 (2 147 483 647), ce qui est souvent bien suffisant. Si vous avez besoin d'un peu plus, comme les types numériques sont signés, vous pouvez disposer de 4 294 967 295 valeurs en incluant les nombres négatifs dans votre identifiant. Vous pouvez aussi indiquer le plus grand nombre négatif possible comme semence (*seed*) de votre propriété IDENTITY, pour permettre à la colonne auto-incrémentale de profiter de cette plage de valeurs élargie :

```
CREATE TABLE dbo.autoincrement (id int IDENTITY(-2147483648, 1)).
```

Pour les identifiants de tables de référence, un SMALLINT ou un TINYINT peuvent suffire. Il est tout à fait possible de créer une colonne de ces types avec la propriété IDENTITY.

N'oubliez pas que les identifiants sont en général les clés primaires des tables. Par défaut, la clé primaire est *clustered* (voir *les index clustered* en section 6.1). L'identifiant se retrouve donc très souvent clé de l'index *clustered* de la table. La clé de l'index *clustered* est incluse dans tous les index *nonclustered* de la table, et a donc une influence sur la taille de tous les index. Plus un index est grand, plus il est coûteux à parcourir, et moins SQL Server sera susceptible de l'utiliser dans un plan d'exécution.

Chaînes de caractères

Les **chaînes de caractères** peuvent être exprimées en **CHAR** ou en **VARCHAR** pour les chaînes ANSI, et en NCHAR ou NVARCHAR pour les chaînes UNICODE. Un caractère **UNICODE** est codé sur deux octets. Un type UNICODE demandera donc deux fois plus d'espace qu'un type ANSI. La taille maximale d'un (VAR)CHAR est 8000. La taille maximale d'un N(VAR)CHAR, en toute logique, 4000. La taille du CHAR est fixe, celle du VARCHAR variable. La différence réside dans le stockage. Un CHAR sera toujours stocké avec la taille déclarée à la création de la variable ou de la colonne, et un VARCHAR ne stockera que la taille réelle de la chaîne. Si une valeur inférieure à la taille maximale est insérée dans un CHAR, elle sera automatiquement complétée par des espaces (caractère ASCII 32). Ceci concerne uniquement le stockage, les fonctions de chaîne de T-SQL et les comparaisons appliquant automatiquement un RTRIM au contenu de variables et de colonnes. Démonstration :

```
DECLARE @ch char(20)
DECLARE @vch varchar(10)
SET @ch = '1234'
SET @vch = '1234  ' -- on ajoute deux espaces
SELECT ASCII(SUBSTRING(@ch, 5, 1)) - 32 = espace

SELECT LEN (@ch), LEN(@vch) -- les deux donnent 4
IF @ch = @vch
    PRINT 'c''est la même chose' -- eh oui, c'est la même chose

SELECT DATALENGTH (@ch), DATALENGTH(@vch)
    -- @vch = 6, les espaces sont présents.
```

```
SET @vch = @ch
SELECT DATALENGTH(@vch) -- 20 !
```

Ce comportement est toujours valable dans le cas de chaînes UNICODE, ou lors du mélange de chaînes ANSI avec des chaînes UNICODE, sauf dans le cas du `LIKE` ! Dans ce cas, les espaces finaux sont importants. Il s'agit d'une conformité avec la norme SQL-92 (implémentée seulement pour UNICODE…) :

```
DECLARE @vch varchar(50)
DECLARE @nvch nvarchar(100)

SET @vch = 'salut!'
SET @nvch = 'salut!  '

IF @vch = @nvch PRINT 'ok'; -- ça marche...

IF @vch LIKE @nvch PRINT 'ok' - ça ne marche pas...
```

Donc, utilisez les `'%'` pour obtenir des résultats conformes à vos attentes dans ces cas.

Qu'en est-il donc de l'utilité du `RTRIM` dans la récupération ou l'insertion de données ? Le `RTRIM` est inutile dans un traitement de `CHAR`, puisque le moteur de stockage ajoute de toute manière des espaces pour compléter la donnée. Par contre, pour l'insertion ou l'affichage de `VARCHAR`, un `RTRIM` peut être utile si vous soupçonnez que les chaînes insérées peuvent contenir des espaces. Comme nous le voyons dans l'exemple, les espaces explicites à la fin d'une chaîne sont conservés dans le `VARCHAR`. Cela inclut le cas où vous copiez une valeur d'un type `CHAR` vers un type `VARCHAR` : les espaces sont conservés. Méfiez-vous de ne pas alourdir inutilement vos tables en oubliant ce détail lors de l'importation ou de l'échange de données.

Note – Ce comportement correspond au cas où l'option `SET ANSI_PADDING` était activée lors de la création de la colonne ou de la variable. Cette option est par défaut à `ON`, et il est recommandé de la laisser ainsi, pour garantir un comportement cohérent des colonnes, en accord avec la norme SQL.

Vous pouvez vérifier vos colonnes de type `VARCHAR` avec une requête comme celle-ci :

```
SELECT
    AVG(DATALENGTH(EmailAddress)) as AvgDatalength,
    AVG(LEN(EmailAddress)) as AvgLen,
    SUM(DATALENGTH(EmailAddress)) - SUM(LEN(EmailAddress)) as Spaces,
    COLUMNPROPERTY(OBJECT_ID('Person.Contact'),
        'EmailAddress', 'precision') as ColumnLength
FROM Person.Contact;
```

qui vous donne le remplissage moyen de la colonne, contre la taille moyenne réelle, et la différence en octets totaux.

Attention, cette requête n'est valable que pour les colonnes ANSI. Divisez `DATALENGTH()` par deux, dans le cas de colonnes UNICODE.

Comment connaître les colonnes de taille variable dans votre base de données ? Une requête de ce type va vous donner la réponse :

```
SELECT
    QUOTENAME(TABLE_SCHEMA) + '.' + QUOTENAME(TABLE_NAME) as tbl,
    QUOTENAME(COLUMN_NAME) as col,
    DATA_TYPE + ' (' +
        CASE CHARACTER_MAXIMUM_LENGTH
            WHEN -1 THEN 'MAX'
            ELSE CAST(CHARACTER_MAXIMUM_LENGTH as VARCHAR(20))
        END + ')' as type
FROM INFORMATION_SCHEMA.COLUMNS
WHERE DATA_TYPE LIKE 'var%' OR DATA_TYPE LIKE 'nvar%'
ORDER BY tbl, ORDINAL_POSITION;
```

Vous pouvez aussi utiliser la fonction `COLUMNPROPERTY()`.

En conclusion, la règle de base est assez simple : utilisez le plus petit type de données possible. Si votre colonne n'a pas besoin de stocker des chaînes dans des langues nécessitant l'UNICODE (chinois, japonais, etc.) créez des colonnes ANSI. Elles seront raisonnablement valables pour toutes les langues utilisant un alphabet latin. Elles vous feront économiser 50 % d'espace. Certains outils Microsoft ont tendance à créer automatiquement des colonnes UNICODE. N'hésitez pas à changer cela à l'aide d'un `ALTER TABLE` par exemple. Il n'y a pratiquement aucun risque d'effet de bord dans votre code SQL, à part l'utilisation de `DATALENGTH` ou de `LIKE`, comme nous l'avons vu précédemment. La différence de temps d'accès par le moteur de stockage entre une colonne fixe ou variable, en termes de recherche de position de la chaîne, est négligeable. En revanche, la différence de volume de données à transmettre lors d'un `SELECT`, peut peser sur les performances. Comme règle de base, n'utilisez des `CHAR` que pour des longueurs inférieures à 10 octets, et de préférence sur des colonnes dans lesquelles vous êtes sûrs de stocker toujours la même taille de chaîne (par exemple, pour une colonne de codes ISO de pays, ne créez pas un `VARCHAR(2)`, mais un `CHAR(2)`, car le `VARCHAR` crée deux octets supplémentaires pour le pointeur qui permet d'indiquer sa position dans la ligne). Évitez les `CHAR` de grande taille : vous risquez de remplir inutilement votre fichier de données. Imaginons que vous créiez une table ayant un `CHAR(5000)` ou un `NCHAR(2500)`, vous auriez ainsi la garantie que chaque page de données ne peut contenir qu'une ligne. Si vous ne stockez que dix caractères dans cette colonne, vous avec besoin, pour dix octets, d'occuper en réalité 8 060 octets…

Inversement, des colonnes variables peuvent provoquer de la fragmentation dans les pages ou dans les extensions, selon que les mises à jour modifient la taille de la ligne. Donc, songez à tester régulièrement la fragmentation, et à défragmenter vos tables lorsque nécessaire. Nous aborderons ce point dans le chapitre 6.

Valeurs temporelles

La plupart des bases de données doivent stocker des données temporelles. En SQL Server 2005, vous disposez de deux types de données : `DATETIME` et `SMALLDATETIME`. Ne confondez pas avec `TIMESTAMP`, qui contrairement à d'autres SGBDR et à la norme SQL, ne correspond pas dans SQL Server à un type de données de date, mais à un compteur unique par base de données, incémenté automatiquement à la modification de toute colonne portant ce type de données. `DATETIME` est codé sur huit octets, quatre octets stockant la date, et quatre l'heure. `SMALLDATETIME` est codé sur quatre octets, deux octets stockant la date, et deux l'heure. Cela permet au type `DATETIME` de couvrir une période allant du 1er janvier 1753 au 31 décembre 9999, avec une précision de la partie heure de 3,33 millisecondes. Le type `SMALLDATETIME` va du 1 janvier 1900 au 6 juin 2079, avec une précision à la minute. Il n'y a pas, en SQL Server 2005, de possibilité de stocker une date séparément d'une heure, sauf à l'adapter dans un autre type de données (en `CHAR` par exemple). Cela reste très peu utilisé en pratique, car posant le problème de la manipulation des dates : comment trouver le nombre de jours séparant deux `CHAR` ? L'habitude prise est donc d'exprimer des dates seulement en spécifiant une heure de 00:00:00, et les heures seules en passant une date au 1er janvier 1900.

Le type **`SMALLDATIME`** est deux fois plus court que le type `DATETIME` et est donc à préférer tant que possible. Il est idéal pour exprimer toute date/heure dont la granularité est au-dessus de la minute. Si vous devez stocker des temps plus précis, comme par exemple des lignes de journalisation, n'utilisez surtout pas `SMALLDATETIME` : il vous serait impossible de trier correctement les lignes à l'intérieur de la même minute. Il fut un temps où certains langages client, comme Visual Basic, ne manipulaient pas correctement le type `SMALLDATETIME`. Ce temps est révolu depuis longtemps.

Le type **`DATETIME`** est plus précis, mais pas exactement précis. Méfiez-vous de son arrondi : vous pouvez passer, en chaîne de caractères, une valeur de temps en millisecondes, qui sera arrondie automatiquement par SQL Server, ce qui peut vous réserver des surprises. Imaginez que votre application cliente soit précise à la milliseconde, et insère deux lignes, à deux millisecondes près. Pour SQL Server, la valeur `DATETIME` sera la même, comme le prouve l'exemple suivant :

```
DECLARE @date1 datetime
DECLARE @date2 datetime

SET @date1 = '20071231 23:59:59:999'
SET @date2 = '20080101 00:00:00:001'

IF @date1 = @date2
    PRINT 'pas de différence'
```

Pour SQL Server, ces deux dates correspondent au 1er janvier 2008, minuit exactement (`2008-01-01 00:00:00.000`). Attention donc également aux comparaisons. Puisque les types `DATETIME` et `SMALLDATETIME` contiennent toujours la partie heure, une comparaison doit se faire aux 3 millisecondes, ou à la minute près. C'est un problème récurrent de performances, car il faut souvent convertir la valeur de la colonne pour lui donner l'heure `00:00:00.000` dans la clause `WHERE`, ce qui empêche toute utilisation d'index. Si vous savez que vous avez ce type de recherche à faire, stockez systématiquement vos `DATETIME` avec la partie horaire à 0, ou créez une colonne calculée indexée qui reprend la date seulement, et recherchez sur cette colonne. Avec SQL Server 2008, le choix est plus étendu.

SQL Server 2008 ajoute plusieurs nouveaux types de données temporels :

- **`DATE`** – stocke seulement la date, sur trois octets ;
- **`TIME`** – stocke seulement la partie horaire. `TIME` est beaucoup plus précis que la partie heure d'un `DATETIME`. Sa plage de valeurs va de `00:00:00.0000000` à `23:59:59.9999999`. Sa précision est donc de 100 nanosecondes, sur trois à cinq octets ;
- **`DATETIME2`** – stocké sur huit octets, `DATETIME2` permet une plus grande plage de valeurs que `DATETIME`. Sa plage de dates va de `0001-01-01` à `9999-12-31`, avec une précision de temps (paramétrable) à 100 nanosecondes. Si vous diminuez la précision, vous pouvez stocker un `DATETIME2` sur six octets. Ce nouveau type a été introduit principalement par compatibilité avec les dates .NET, et avec la norme SQL ;
- **`DATETIMEOFFSET`** – basé sur un `DATETIME2`, `DATETIMEOFFSET` ajoute une zone horaire. Codé sur huit à dix octets.

> Pour éviter les problèmes en rapport avec les formats de dates liés aux langues, passer les dates en ISO 8601 non séparé : `'AAAAMMJJ'`.

Type XML

SQL Server 2005 a introduit le **type de données XML**, qui fait partie de la norme SQL:2003. Une colonne ou variable `XML` peut contenir jusqu'à 2 Go d'instance XML bien formée ou valide, codé en UTF-16 uniquement. Une colonne `XML` peut être validée par une collection de schéma XSD (un schéma, dans le monde XML, est la description de la structure d'un document XML. Le sujet en soi mériterait un ouvrage). Voici un exemple de création de collection de schémas :

```
CREATE XML SCHEMA COLLECTION [Person].[AdditionalContactInfoSchemaCollection]
AS N'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
       ...
</xsd:schema>'
```

Le schéma se voit dans la vue système `sys.xml_schema_collections`, ou dans l'explorateur d'objets de SSMS, dans une base de données, dans le dossier Programmability/Types/XML Schema Collections. On l'appelle une collection de schémas, parce que plusieurs schémas peuvent être utilisés, de façon complémentaire, pour valider le même document ou fragment. Vous pouvez ensuite créer une table qui contient une colonne de type `XML`, validée par cette collection de schémas :

```
CREATE TABLE [Person].[Contact](
       [ContactID] [int] IDENTITY(1,1) NOT NULL,
       -- ...
       [AdditionalContactInfo]
[xml]([Person].[AdditionalContactInfoSchemaCollection]) NULL
);
```

Nous obtenons alors une colonne `XML` typée, par opposition à une colonne non validée par un schéma, dite non typée. La collection de schémas est une contrainte de colonne, au même titre qu'une contrainte `CHECK`, par exemple. On pourrait se dire que la validation par le schéma est une étape supplémentaire effectuée par le moteur, et qu'elle est donc nuisible aux performances. En réalité, c'est le contraire. Comme pour la contrainte `CHECK`, la vérification est effectuée à l'écriture dans la colonne (et selon la complexité de la collection de schémas, cette étape peut se révéler coûteuse), mais on lit en général plus souvent que l'on écrit. Par ailleurs, à l'extraction, SQL Server pourra s'appuyer sur les informations fournies par la contrainte pour faciliter son travail. Le schéma fournit des informations précises sur le type de données des items dans le XML, ce qui permet à SQL Server de stocker l'instance XML sous une forme binaire (un *blob* XML) au lieu d'une représentation textuelle, ce qui rend la colonne `XML` beaucoup plus compacte. Les valeurs atomiques sont aussi stockées avec leur type de données (l'information de type est donnée dans la collection de schéma), ce qui permet un *parsing* beaucoup plus efficace. Enfin, la vérification de la validité d'une requête XQuery par rapport aux types de valeurs peut être vérifiée à sa compilation, et des optimisations peuvent être effectuées. Le typage améliore aussi l'efficacité des index XML[1]. Bref, cherchez autant que possible à créer des colonnes `XML` typées.

Vous pouvez indexer vos colonnes `XML`, ce qui crée une représentation interne en table relationnelle, et crée un index *B-Tree* sur cette structure. Pour plus d'information sur la structure des index, reportez-vous à la section 6.1.

Types divers

Il existe un type de données dont nous ne devrions même pas parler ici, tant il est à éviter : **`sql_variant`**. Comme son nom l'indique, il correspond à un variant dans les langages comme Visual Basic, c'est-à-dire qu'il adapte automatiquement son type

1. Plus d'infos dans l'article *Technet* « Performance Optimizations for the XML Data Type in SQL Server 2005 » sur http://msdn.microsoft.com/en-us/library/ms345118.aspx.

par rapport à la valeur stockée. C'est évidemment un type à éviter absolument, tant par son impact sur les performances que par son laxisme en terme de contraintes de données. Le type `sql_variant` peut contenir 8 016 octets, il est donc indexable à vos risques et périls (une clé d'index supérieure à 900 octets générera une erreur). On ne peut le concaténer, ni l'utiliser dans une colonne calculée, ou un `LIKE`... Bref, oubliez-le.

Le type de données **`UNIQUEIDENTIFIER`** correspond à un GUID (*Globally Unique Identifier*), une valeur codée sur 16 octets générés à partir d'un algorithme qui la rend unique à travers le monde et au delà (le GUID n'est pas garanti unique, mais les probabilités que deux GUID identiques puissent être générés sont statistiquement infinitésimales). Vous voyez des GUID un peu partout dans le monde Microsoft, notamment dans la base de registre. On entend en général par le terme GUID, l'implémentation par Microsoft de la norme UUID (*Universally Unique Identifier*), qui calcule notamment une partie de la chaîne avec l'adresse matérielle (MAC) de l'ordinateur. Un exemple de GUID serait :

CC61D24F-125B-4347-BB37-9E7908613279.

Certaines personnes l'utilisent pour générer des clés primaires de tables, profitant de l'unicité globale du GUID pour obtenir une clé unique à travers différentes tables ou bases de données, ou pour générer des identifiants non séquentiels, par exemple pour un identifiant d'utilisateur sur une application Web, et ainsi éviter qu'un pirate ne devine l'ID d'un autre utilisateur. En termes de performances, utiliser un `UNIQUEIDENTIFIER` en clé primaire est une très mauvaise idée, à plus forte raison si l'index sur la clé est *clustered*. Cela tient bien sûr à la taille de la donnée (vous verrez dans le chapitre 6 que tous les index incorporent la clé de l'index *clustered*). C'est aussi un excellent moyen de fragmenter la table : si vous créez un index *clustered* sur une colonne où les données s'insèrent à des positions aléatoires, vous provoquez une intense activité de séparation de pages (les pages doivent se séparer et déplacer une partie de leurs lignes ailleurs, pour accommoder les nouvelles lignes insérées, plus de détails dans le chapitre sur les index), donc une baisse de performances, ainsi qu'une forte fragmentation, d'où une nouvelle baisse de performances. Enfin, cela a de fortes chances d'augmenter significativement la taille de votre base, puisqu'une clé primaire est souvent déplacée dans plusieurs tables filles en clés étrangères. Tout cela faisant 16 octets par clé, le stockage est plus lourd, les index plus gros, et les jointures plus coûteuses pour les requêtes.

Si vous songez à utiliser des GUID dans une problématique de bases de données dispersées, optez plutôt pour une clé primaire composite, à deux colonnes : une colonne donnant un identifiant de serveur ou de base de données, et l'autre identifiant un compteur `IDENTITY` propre à la table. Vous aurez non seulement l'information de la base source de la ligne dans la clé, mais de plus celle-ci pèsera 5 ou 6 octets (un `TINYINT` ou `SMALLINT` pour l'ID de base, un `INT` pour l'ID de ligne) au lieu de 16.

Vous générez les valeurs de GUID dans vos colonnes `UNIQUEIDENTIFIER` à l'aide de la fonction T-SQL `NEWID()`, que vous pouvez placer dans la contrainte `DEFAULT` de la colonne. Si vous devez utiliser un `UNIQUEIDENTIFIER` et que vous voulez l'indexer, vous pouvez utiliser (seulement en contrainte `DEFAULT`) la fonction `NEWSEQUENTIALID()`, qui génère un GUID plus grand que tous ceux présents dans la colonne. Cette fonction a été créée pour éviter la fragmentation. Toutefois, si vous l'utilisez, vous perdez l'avantage de la génération aléatoire, et un pirate intelligent pourra facilement deviner des GUID précédents. Dans les problématiques d'identifiant Web, il vaut mieux programmer dans l'application cliente un algorithme de génération de hachage à partir des identifiants de la base de données, pour les protéger des curieux. Cela permet de conserver de bonnes performances du côté SQL Server.

Objets larges

SQL Server proposait les types `TEXT`, `NTEXT` et `IMAGE`, pour stocker respectivement des objets larges de type chaînes ANSI et UNICODE, et des objets binaires, comme des documents en format propriétaire, des fichiers image ou son. Ces types sont toujours à disposition, mais ils sont obsolètes et remplacés par **`(N)VARCHAR(MAX)`** et **`VARBINARY(MAX)`**. Comme leurs anciens équivalents, ils permettent de stocker jusqu'à 2 Go de données (2^31 – 1 octets, précisément), mais leur comportement est sensiblement différent. Ces types sont ce qu'on appelle des LOB (*Large Objects*, objets larges), bien que dans les BOL, vous verrez utiliser le nom « valeurs larges » (*large values*) pour les nouveaux types (le type XML en fait aussi partie). Ils sont stockés à même le fichier de données, dans des pages de 8 Ko organisées d'une façon particulière.

Les types `TEXT`, `NTEXT` et `IMAGE`, conformément à leur comportement dans SQL Server 2000, placent dans la ligne de la table un pointeur de 16 octets vers une structure externe à la page. Cette structure est organisée en arbre équilibré (*B-Tree*), comme un index, pour permettre un accès rapide aux différentes positions à l'intérieur du LOB. Lors de toute restitution de la ligne, le moteur de stockage doit donc suivre ce pointeur et récupérer ces pages, même si la colonne ne contient que quelques octets. En SQL Server 2000, nous avions un moyen de forcer SQL Server à stocker dans la page elle-même le contenu du LOB, s'il ne dépassait pas une certaine taille, à l'aide de la procédure stockée `sp_tableoption`. Cette option existe toujours sur SQL Server 2005 et peut être utile pour optimiser des colonnes `(N)TEXT`. Par exemple :

```
EXEC sp_tableoption nomtable 'text in row', taille;
```

où *taille* est la taille en octets des données à inclure dans la page, dans une plage allant de 24 à 7 000 (la chaîne 'ON' est aussi acceptée, elle correspond à une taille de 256 octets). Dès que cette option est activée, si le contenu de la colonne est inférieur à la taille indiquée, il est inséré dans la page, comme une valeur traditionnelle. Pourquoi la limite inférieure est-elle de 24 octets ? Simplement parce que, quand `'text in row'` est activé, le moteur de stockage ne place plus de pointeur vers la structure LOB. Si la donnée insérée est plus grande que la taille maximum spécifiée, il inscrira dans la page la racine de la structure *B-Tree* du LOB, qui pèse… 24 bits.

Pour désactiver cette option, il suffit de fixer la taille à 0, ou 'OFF' (attention, cela force la conversion de tous les BLOB existants, ce qui peut prendre un certain temps selon la cardinalité de la table).

Cette opération n'étant utile que pour les colonnes de type (N)TEXT et IMAGE, nous ne nous y attarderons pas. Utilisez les nouveaux types. Ceux-ci gèrent automatiquement l'inclusion dans la ligne lorsque le contenu peut tenir dans la page (à concurrence de 8 000 octets), en agissant alors comme un VARCHAR. S'il ne peut tenir dans la page, il est déplacé dans une structure LOB, et agit donc comme un objet large. Le terme « MAX » permet d'assurer l'évolution vers des tailles de LOB supérieures dans les versions futures de SQL Server sans modifier la définition des tables. Ce fonctionnement représente le meilleur des deux mondes.

Une autre option de table gère le comportement des valeurs larges ((N)VARCHAR(MAX), VARBINARY(MAX) et XML) :

```
EXEC sp_tableoption nomtable 'large value types out of row',
     ['ON'|'OFF'];
```

- ON (1) – Le contenu est toujours stocké dans une structure LOB, la page ne contient que le pointeur à 16 octets.
- OFF (0) – Le contenu est stocké dans la page s'il peut y tenir.

Il s'agit donc du raisonnement inverse de 'text in row'. ON veut dire ici que le LOB est toujours extérieur à la page. La valeur par défaut est OFF. Vous pouvez inspecter les valeurs des deux options pour vos tables, avec la vue système sys.tables :

```
SELECT SCHEMA_NAME(schema_id) + '.' + Name as tbl,
    text_in_row_limit,
    large_value_types_out_of_row
FROM sys.tables
ORDER BY tbl;
```

Il peut être intéressant de fixer 'large value types out of row' à ON, au cas par cas. Si une table comporte une colonne de valeur large (un XML par exemple) qui stocke des structures qui ne sont que très peu retournées dans les requêtes d'extraction, activer cette option permet de conserver des pages plus compactes, et donc d'optimiser la plupart des lectures qui ne demandent pas le LOB.

Lorsque l'option 'large value types out of row' est changée, dans un sens ou dans l'autre, les lignes existantes ne sont pas modifiées. Elles le seront au fur et à mesure des UPDATE.

Dépassement de ligne

Dans le chapitre concernant la structure du fichier de données, nous avons dit qu'une ligne ne pouvait dépasser 8 060 octets, soit l'espace disponible dans une page de données de 8 Ko. Nous venons de voir que le stockage des LOB permet de s'affranchir de cette limite. Ce stockage comporte toutefois des inconvénients : la structure en *B-Tree* de gestion des LOB est un peu lourde lorsqu'il s'agit de gérer des valeurs qui vont jusqu'à 8 Ko, mais qui ne peuvent tenir dans la page à cause de la

taille des autres colonnes. Une solution à ce problème pourrait être d'effectuer un partitionnement verticale, en créant une autre table contenant une colonne `VARCHAR(8000)` et l'identifiant, pour une relation un-à-un sur la première table. À partir de SQL Server 2005, ce n'est plus nécessaire, un mécanisme automatique permet aux colonnes de longueur variable de dépasser la limite de la page. Cette fonctionnalité est appelée **dépassement de ligne** (*row overflow*), elle ne s'applique qu'aux types variables `(N)VARCHAR` et `VARBINARY`, ainsi qu'aux types de données .NET. Prenons un exemple simple :

```
USE tempdb
GO

CREATE TABLE dbo.longcontact (
    longcontactId int NOT NULL PRIMARY KEY,
    nom varchar(5000),
    prenom varchar(5000)
)
GO

INSERT INTO dbo.longcontact (longcontactId, nom, prenom)
SELECT 1, REPLICATE('N', 5000), REPLICATE('P', 5000)
GO
```

Avant SQL Server 2005, la création de cette table aurait été acceptée, mais avec l'affichage d'un avertissement disant que l'insertion d'une ligne dépassant 8 060 octets ne pourrait aboutir. Et dans les faits, l'`INSERT` aurait déclenché une erreur. En SQL Server 2005, tout fonctionne comme un charme. Observons ce qui a été stocké :

```
SELECT p.index_id, au.type_desc, au.used_pages
FROM sys.partitions p
JOIN sys.allocation_units au ON p.partition_id = au.container_id
WHERE p.object_id = OBJECT_ID('dbo.longcontact');
```

L'insertion s'est donc déroulée sans problème pour une ligne d'une taille totale de 10 004 octets (2 × 5 000, plus 4 octets d'`INTEGER`). À l'insertion, SQL Server a automatiquement déplacé une partie de la ligne dans une autre page, créée à la volée, qui sert uniquement à continuer la ligne. Cette page est d'un type spécial, comme nous l'avons vu dans la requête sur `sys.allocation_units`. Alors que la ligne est normalement dans une page de type `In-row data`, le reste doit être placé dans une page de type `Row-overflow data`.

Nous voyons qu'il y a quatre pages en tout, dont deux pages de *row overflow*. Pourquoi deux pages, alors que nous n'avons ajouté que de quoi remplir une seule page de dépassement ? Vérifions à l'aide de `DBCC IND` :

```
DBCC IND ('tempdb', 'longcontact', 1);
```

Nous voyons le résultat sur la figure 4.8. Pour chaque type de page, nous avons une page de type 10, c'est-à-dire une page IAM (*Index Allocation Map*), en quelque sorte la table des matières du stockage. Tout s'explique.

PageFID	PagePID	IAMFID	IAMPID	ObjectID	IndexID	PartitionNumber	PartitionID	iam_chain_type	PageType
1	94	NULL	NULL	1589580701	1	1	72057594039107584	In-row data	10
1	78	1	94	1589580701	1	1	72057594039107584	In-row data	1
1	41	NULL	NULL	1589580701	1	1	72057594039107584	Row-overflow data	10
1	126	1	41	1589580701	1	1	72057594039107584	Row-overflow data	3

Figure 4.8 — Résultat de DBCC IND

Essayons maintenant ceci :

```
UPDATE dbo.longcontact
SET prenom = 'Albert'
WHERE longcontactId = 1;
```

Et relançons la requête sur `sys.allocation_units`. Nous n'avons plus que trois pages : l'IAM et la page de données pour l'allocation `In-row data`, et seulement l'IAM pour l'allocation `Row-overflow data`. La page de *row overflow* a été supprimée, toute la ligne est revenue dans la page. La page d'IAM n'a pas été supprimée, elle pourra être réutilisée au prochain dépassement.

SQL Server ne remet pas toujours la colonne dans la page In row lorsque la valeur diminue. Il ne vérifie que la nouvelle valeur peut tenir dans la page que si elle a diminué d'au moins 1 000 octets, ce qui lui évite de trop fréquentes vérifications.

Essayons autre chose :

```
TRUNCATE TABLE dbo.longcontact
GO
INSERT INTO dbo.longcontact (longcontactId, nom, prenom)
SELECT 2, REPLICATE('N', 5000), REPLICATE('P', 3100)
INSERT INTO dbo.longcontact (longcontactId, nom, prenom)
SELECT 3, REPLICATE('N', 5000), REPLICATE('P', 3100)
GO
```

Ici, nous forçons l'insertion de deux nouvelles lignes qui vont générer du dépassement, mais avec un remplissage de colonnes qui peut faire tenir deux lignes dans la même page In-row… ce que va faire SQL Server. Il va créer en effet une seule page In-row, et déplacer les deux colonnes `nom` dans deux pages de *row overflow*. Il faut noter que les pages de *row overflow* contiennent des valeurs entières. Une colonne ne peut pas être répartie sur plusieurs pages. C'est pour cette raison que le *row overflow* s'applique à des types variables limités à une taille de 8 000 octets (et à une taille minimum de 24 octets, qui est l'espace nécessaire pour stocker le pointeur vers la page de *row overflow*).

Cela nous amène à une dernière question : une page de *row overflow* peut-elle partager des données venant de plusieurs lignes, ou faut-il une nouvelle page pour chaque ligne en dépassement ? La seconde solution augmenterait le volume de stoc-

kage de façon potentiellement importante. Vérifions ce qui se passe, à l'aide d'une table légèrement modifiée :

```
CREATE TABLE dbo.troplongcontact (
    troplongcontactId int NOT NULL PRIMARY KEY,
    nom char(8000),
    prenom varchar(5000)
)
GO

INSERT INTO dbo.troplongcontact (troplongcontactId, nom, prenom)
SELECT 2, REPLICATE('N', 8000), REPLICATE('P', 1000)
INSERT INTO dbo.troplongcontact (troplongcontactId, nom, prenom)
SELECT 3, REPLICATE('N', 8000), REPLICATE('P', 1000)
GO
```

Ici, nous ne conservons qu'une seule colonne de taille variable. Nous sommes donc certain de ne générer du *row overflow* que sur cette colonne.

Que nous dit un `DBCC IND` ? Nous voyons le résultat sur la figure 4.9.

PageFID	PagePID	IAMFID	IAMPID	ObjectID	IndexID	PartitionNumber	PartitionID	iam_chain_type	PageType	IndexLevel
1	194	NULL	NULL	1621580815	1	1	72057594039173120	In-row data	10	NULL
1	193	1	194	1621580815	1	1	72057594039173120	In-row data	1	0
1	195	1	194	1621580815	1	1	72057594039173120	In-row data	2	1
1	196	1	194	1621580815	1	1	72057594039173120	In-row data	1	0
1	192	NULL	NULL	1621580815	1	1	72057594039173120	Row-overflow data	10	NULL
1	127	1	192	1621580815	1	1	72057594039173120	Row-overflow data	3	0

Figure 4.9 — Résultat de DBCC IND

Nous voyons quatre pages `In row`, et deux pages `Data overflow`, de différents `DataTypes`. Ceux qu'on voit ici sont :

- 1 – page de données.
- 2 – page d'index.
- 3 – page de LOB.
- 10 – IAM.

Il y a donc deux pages de données In Row, et une page LOB. Observons la page LOB à l'aide de `DBCC PAGE` :

```
DBCC TRACEON (3604)
DBCC PAGE (tempdb, 1, 127, 3)
```

Dans le résultat, nous trouvons ceci :

```
Blob row at: Page (1:127) Slot 0 Length: 1014 Type: 3 (DATA)
Blob Id:9306112
…
Blob row at: Page (1:127) Slot 1 Length: 1014 Type: 3 (DATA)
Blob Id:1128660992
```

La longueur correspond à nos 1 000 octets, plus quelques octets supplémentaires. CQFD.

La fonctionnalité de *row overflow* est bien pratique, mais qu'en est-il des performances ? Elles sont nécessairement moins bonnes. À chaque lecture de page, le moteur de stockage doit aller chercher ailleurs d'autres pages pour lire les données si des colonnes en *row overflow* sont demandées, ce qui provoque des lectures aléatoires (*random IO*) supplémentaires. De plus, comme les pages de *row overflow* contiennent la totalité de la valeur d'une colonne, cela génère souvent de la fragmentation interne : les pages ne sont pas remplies de façon optimale. En un mot, réservez le *row overflow* à des cas limités, pour gérer un éventuel dépassement de page, pour des colonnes qui sont en général compactes mais peuvent de temps en temps contenir une chaîne plus longue.

Les statistiques de pages lues, retournées par `SET STATISTICS IO ON`, et la trace SQL, affichent différemment les pages lues. La trace SQL montre, dans la colonne `reads`, le nombre total de pages lues (In row, Row overflow et LOB Data), y compris les pages d'IAM. Les statistiques IO renvoyées dans la session séparent les pages In row des pages Row overflow et LOB. Voici un exemple de statistiques retournées pour une table contenant deux pages In row, deux pages de *row overflow*, et trois pages de LOB (`VARCHAR(MAX)`), pages IAM comprises :

```
Table 'bigcontact'. Scan count 1, logical reads 2, physical reads 0, read-
ahead reads 0, lob logical reads 5, lob physical reads 0, lob read-ahead
reads 0.
```

L'utilisation de `SET STATISTICS IO ON` permet donc d'obtenir des résultats plus précis.

FILESTREAM

Le type **`FILESTREAM`** est nouveau dans SQL Server 2008. C'est une implémentation du type de données `DATALINK` de la norme SQL. SQL Server n'est pas prévu pour gérer efficacement le stockage et la manipulation d'objets larges, et maintenir des LOB en grande quantité pose des problèmes de performances. Souvent, pour éviter ces problèmes, nous stockons les fichiers sur le système de fichiers, et référençons le chemin dans une colonne `VARCHAR`. Le problème avec cette approche est qu'il faut en général ouvrir un partage sur le réseau, maintenir ces fichiers d'une façon ou d'une autre, et gérer les sauvegardes de ces fichiers en plus des sauvegardes SQL.

En gérant des types `FILESTREAM`, vous créez un groupe de fichiers particulier pour les stocker, et ce groupe de fichiers utilise un stockage NTFS, plus adapté, plutôt que le stockage traditionnel des fichiers de données. La cohérence transactionnelle sur ces fichiers est aussi assurée.

Pour utiliser `FILESTREAM`, vous devez d'abord activer la fonctionnalité, à l'aide de `sp_configure` :

```
EXEC sp_configure 'filestream_access_level', '[niveau d'activation]'
RECONFIGURE
```

et de SQL Server Configuration Manager (figure 4.10).

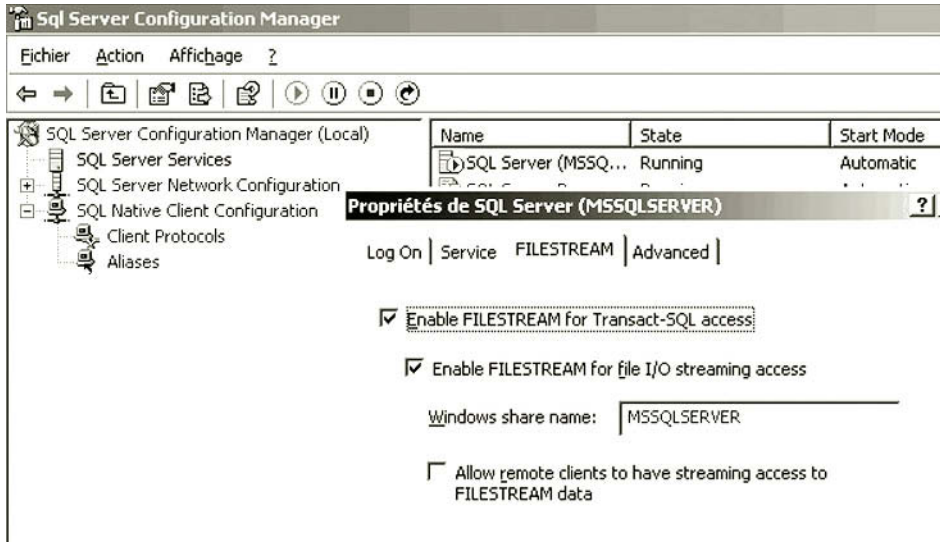

Figure 4.10 — Configurer FILESTREAM

Vous pouvez également employer la procédure stockée `sp_filestream_configure`, qui prend deux paramètres : `@enable_level` et `@share_name`. `@enable_level` indique le niveau d'activation :

0 – désactivé.

1 – activé pour T-SQL uniquement : pas d'accès des fichiers directs hors de SQL Server.

2 – activé aussi pour l'accès direct aux fichiers, en accès local seulement.

3 – activé aussi pour l'accès direct aux fichiers à travers un partage réseau.

`@share_name` permet de spécifier un nom de partage réseau, qui est celui de l'instance SQL Server par défaut. Ensuite, lancez la commande `RECONFIGURE` pour que les changements soient pris en compte.

Cette procédure va peut-être disparaître à la sortie définitive de SQL Server 2008, et être remplacée par une option d'installation de l'instance. Voir http://www.sqlskills.com/blogs/bobb/2008/02/28/ConfiguringFilestreamInCTP6ItsDifferent.aspx et les BOL si vous avez des problèmes à sa mise en œuvre. De plus, dans la version bêta, un redémarrage de l'ins-

tance est souvent nécessaire pour activer cette fonction. Cela pourrait être amélioré à la sortie finale.

Vous créez ensuite un groupe de fichiers destiné au FILESTREAM :

```
ALTER DATABASE AdventureWorks
ADD FILEGROUP FileStreamGroup1 CONTAINS FILESTREAM

ALTER DATABASE AdventureWorks
ADD FILE
  ( NAME = N'AdventureWorks_media',
    FILENAME = N'E:\sqldata\AdventureWorks_media')
TO FILEGROUP [FileStreamGroup1]
GO
```

Ce fichier crée en réalité un répertoire à son emplacement, comme nous le voyons sur la figure 4.11.

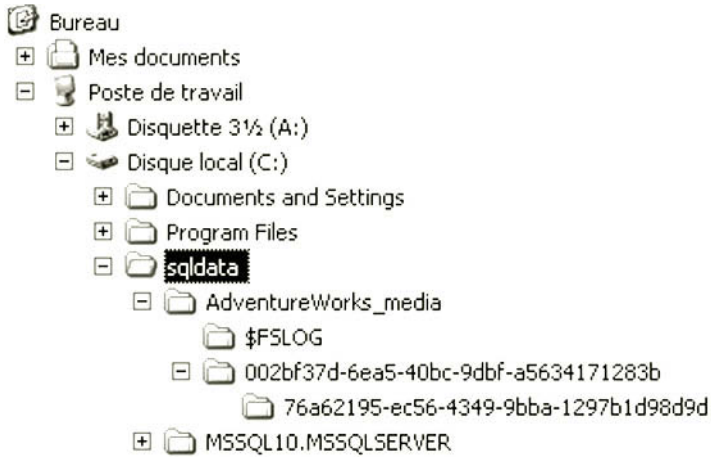

Figure 4.11 — Répertoire des fichiers FILESTREAM

Puis, vous créez une table contenant une colonne de type FILESTREAM :

```
CREATE TABLE dbo.document (
    documentId uniqueidentifier NOT NULL ROWGUIDCOL
        DEFAULT (NEWID()) PRIMARY KEY NONCLUSTERED ,
    nom varchar(1000) NOT NULL,
    document varbinary(max) FILESTREAM);
```

Le GUID (uniqueidentifier ROWGUIDCOL) est nécessaire pour la gestion du FILESTREAM par le moteur de stockage. Terminons par un exemple de code, qui vous montre l'insertion, et l'extraction à partir du FILESTREAM. Remarquez la méthode PathName() :

```
INSERT INTO dbo.document (nom, document)
```

```
SELECT 'babaluga', CAST('plein de choses à dire' as varbinary(max))

SELECT
    nom,
    CAST(document as varchar(max)) as document,
    document.PathName()
FROM dbo.document
```

Volume de données

Attention au nombre de fichiers stockés dans les répertoires `FILESTREAM`. À partir de quelques centaines de milliers de fichiers, des lenteurs sérieuses peuvent se faire sentir. La source du problème peut être le *scan* du répertoire à chaque ajout de fichier, pour générer un nom en format 8.3 lié. Vous pouvez désactiver ce comportement dans NTFS à l'aide de l'exécutable `fsutil` :

```
fsutil behavior set disable8dot3 1
```

Vous devez redémarrer la machine pour que ceci soit pris en compte.

4.2 PARTITIONNEMENT

Mathématiquement, l'augmentation de la taille d'une base de données entraîne une diminution de ses performances en lecture et en écriture. Grâce à la grande qualité des techniques d'optimisation mises en œuvre dans les SGBDR modernes, et à condition que votre modèle de données soit intelligemment construit, cette augmentation de temps de réponse n'est en rien linéaire. SQL Server peut retrouver avec une grande vélocité quelques lignes dans une table qui en compte des millions. Toutefois, il n'y a pas de miracle : augmentation de taille signifie multiplication des pages de données et d'index, donc plus de lectures et plus de temps pour parcourir les objets.

Pour pallier cette inflation de taille, vous pouvez partitionner vos tables. Le **partitionnement** consiste à séparer physiquement des structures, soit pour en paralléliser le traitement, soit pour retirer des lectures une partie des données moins souvent sollicitée. Le partitionnement peut soit porter sur des lignes (partitionnement horizontal), soit sur des colonnes (partitionnement vertical).

Le partitionnement horizontal est utile dans le cas de tables comportant un attribut temporel, par exemple une table de journalisation, de mouvements comptables, de factures, d'actes ou d'opérations diverses. Les écritures portent très souvent sur des dates récentes, et la grande majorité des lectures concerne une période limitée, par exemple les quelques dernières années. En SQL Server 2000, nous traitions ce problème en créant des tables supplémentaires qui dupliquaient la structure de la table originale, par exemple à l'aide d'un ordre `SELECT INTO`, qui permet de créer une table et d'y insérer le résultat d'un `SELECT` à la volée. Il est tout à fait possible de continuer à utiliser cette technique dans SQL Server 2005/2008. Voici un

exemple qui utilise la structure de contrôle TRY CATCH qui n'existe que depuis la version 2005[1] :

```
BEGIN TRANSACTION
BEGIN TRY
    SELECT *
    INTO Sales.SalesOrderHeader_Archive2002
    FROM Sales.SalesOrderHeader
    WHERE YEAR(OrderDate) = 2002

    DELETE
    FROM Sales.SalesOrderHeader
    WHERE YEAR(OrderDate) = 2002

    COMMIT TRANSACTION
END TRY
BEGIN CATCH
    ROLLBACK TRANSACTION
END CATCH
```

Vous pouvez simplifier cet exemple à partir de 2005 à l'aide de l'instruction DELETE … OUTPUT. La lecture des lignes n'est ainsi effectuée qu'une seule fois :

```
BEGIN TRANSACTION
BEGIN TRY
    SELECT
        0 as SalesOrderID, RevisionNumber, OrderDate, DueDate, ShipDate
    INTO Sales.SalesOrderHeader_Archive2002
    FROM Sales.SalesOrderHeader
    WHERE 1 = 2[2]

    DELETE
    FROM Sales.SalesOrderHeader
    OUTPUT DELETED.SalesOrderID, DELETED.RevisionNumber,
        DELETED.OrderDate, DELETED.DueDate, DELETED.ShipDate
        INTO Sales.SalesOrderHeader_Archive2002
    WHERE YEAR(OrderDate) = 2002

    COMMIT TRANSACTION
END TRY
BEGIN CATCH
    ROLLBACK TRANSACTION
END CATCH
```

Que faire de cette nouvelle table d'archive ? Vous pouvez modifier vos applications clientes pour ajouter par exemple un bouton de recherche sur les archives, qui

1. L'utilisation d'une fonction dans le critère de recherche empêche l'utilisation éventuelle d'un index. Nous le verrons dans la partie traitant de l'optimisation des requêtes. Dans ce cas, c'est peu problématique, dans la mesure où un scan de la table sera de toute manière effectué, le nombre de ligne à retourner étant important.
2. Nous avons volontairement raccourci dans notre exemple le nombre de colonnes, car dans ce cas nous devions éviter de copier la colonne SalesOrderID comme colonne de type IDENTITY (auto-incrémentale). L'OUTPUT refusant d'insérer une valeur explicite dans un IDENTITY.

lance une requête SELECT sur cette table au lieu de la table originelle. Mauvaise solution. La bonne réponse est : créez une vue, qui agrège les différentes tables par des UNION ALL, et effectuez la requête sur cette vue :

```
CREATE VIEW Sales.vSalesOrderHeaderWithArchives
AS
    SELECT
        SalesOrderID, RevisionNumber, OrderDate, DueDate,
        ShipDate, 0 as Source
    FROM Sales.SalesOrderHeader
    UNION ALL
    SELECT
        SalesOrderID, RevisionNumber, OrderDate, DueDate,
        ShipDate, 2002 as Source
    FROM Sales.SalesOrderHeader_Archive2002
```

Ici, pour déterminer dans nos requêtes de quelle table source provient chaque ligne, nous ajoutons une colonne générée dans la vue.

Vous pouvez ensuite modifier votre requête pour effectuer la recherche sur cette vue, ou prévoir deux types de recherche : une recherche sur la table originelle, et une recherche sur toute l'étendue des données, en prévenant vos utilisateurs que la recherche simple est plus rapide, mais ne comporte que des valeurs récentes.

La recherche simple sera-t-elle plus rapide ? Pas forcément. Si nous lui en donnons les moyens, l'optimiseur de SQL Server est capable de savoir si la requête que nous lançons sur cette vue concerne bien toutes les tables, ou si l'accès à une des tables suffit. Comment cela ? En plaçant une contrainte CHECK sur la structure des tables sous-jacentes.

Pour illustrer la différence que va amener cet ajout, commençons par analyser les tables affectées par cet ordre SQL :

```
SET STATISTICS IO ON
GO
SELECT *
FROM Sales.vSalesOrderHeaderWithArchives
WHERE OrderDate BETWEEN '20020301' AND '20020401';
```

Nous voulons les commandes passées en mars 2002...

Résultat des statistiques IO :

```
(305 row(s) affected)
Table 'SalesOrderHeader_Archive2002'. Scan count 1, logical reads 18...
Table 'SalesOrderHeader'. Scan count 1, logical reads 700...
```

Donc, 18 pages lues pour SalesOrderHeader_Archive2002, 700 pages lues pour SalesOrderHeader.

Nous avons créé la table Sales.SalesOrderHeader_Archive2002 pour qu'elle ne contienne que les lignes dont l'OrderDate est compris dans l'année 2002. La table Sales.SalesOrderHeader ne contient donc plus que des lignes dont l'OrderDate est différent de 2002. Exprimons cela en contraintes CHECK sur la colonne OrderDate :

```
ALTER TABLE Sales.SalesOrderHeader
```

```
WITH CHECK ADD CONSTRAINT chk$SalesOrderHeader$OrderDate
  CHECK (OrderDate NOT BETWEEN '20020101' AND '20021231 23:59:59.997');
GO
ALTER TABLE Sales.SalesOrderHeader_Archive2002
WITH CHECK ADD CONSTRAINT chk$SalesOrderHeader_Archive2002$OrderDate
  CHECK (OrderDate BETWEEN '20020101' AND '20021231 23:59:59.997');
```

Le `WITH CHECK` n'est pas nécessaire, car la vérification de la contrainte sur les lignes existantes est effectuée par défaut. Nous désirons simplement mettre l'accent sur le fait que la contrainte doit être reconnue comme étant appliquée aux lignes existantes, pour que l'optimiseur puisse s'en servir.

Relançons le même ordre `SELECT`, et observons les statistiques de lectures :

```
Table 'SalesOrderHeader_Archive2002'. Scan count 1, logical reads 18...
```

Seule la table `SalesOrderHeader_Archive2002` a été affectée. Grâce à la contrainte spécifiée, l'optimiseur sait avec certitude qu'il ne trouvera aucune ligne dans `Sales.SalesOrderHeader` qui corresponde à la fourchette de dates demandée dans la clause `WHERE`. Il peut donc éliminer cette table de son plan d'exécution.

En utilisant les contraintes **`CHECK`**, vous pouvez manuellement partitionner horizontalement vos tables, les rejoindre dans une vue, et ainsi augmenter vos performances de lecture sans sacrifier les fonctionnalités.

Cette technique conserve un désavantage : l'administration (création de tables d'archive) reste manuelle. À partir de SQL Server 2005, des fonctions intégrées de partitionnement horizontal de table et d'index simplifient l'approche.

Partitionnement intégré

Les fonctionnalités de partitionnement sont disponibles sur SQL Server 2005 dans l'édition Entreprise seulement.

Tout ce que nous venons de faire manuellement, et plus encore, est pris en charge par le moteur de stockage. Depuis SQL Server 2005, une couche a été ajoutée dans la structure physique des données, entre l'objet et ses unités d'allocation : la **partition**. L'objectif premier du partitionnement est de placer différentes parties de la même table sur différentes partitions de disque, afin de paralléliser les lectures physiques et de diminuer le nombre de pages affectées par un *scan* ou par une recherche d'index.

La lecture en parallèle sur plusieurs unités de disques n'est vraiment intéressante, bien entendu, que lorsque les partitions sont situées sur des disques physiques différents, et de préférence sur des bus de données différents, ou sur une baie de disques ou un SAN, sur lequel on a soigneusement organisé les partitions. Chaque disque physique n'ayant qu'une seule tête de lecture, il ne sert à rien de compter sur un simple partitionnement logique sur le même disque : les accès ne pourront se faire qu'en

série. De plus, l'intérêt ne sera réel que pour des VLDB (bases de données de grande taille). En effet, dans les bases de données de taille moyenne, les lectures sur le disque devraient être autant que possible évitées par une quantité suffisante de RAM. La plupart des pages sollicitées sont alors déjà contenues dans le *buffer* et tout accès aux disques est évité.

Mais même dans la RAM, il peut être intéressant de n'attaquer qu'une partie de la table. Un autre intérêt du partitionnement est la simplicité et la rapidité avec laquelle on peut ajouter ou retirer des partitions de la table. Nous allons le voir en pratique.

Le partitionnement se prépare avant la création de la table : il ne peut se faire sur une table existante. Pour appliquer un partitionnement à des données déjà présentes, il faut les faire migrer vers une nouvelle table, partitionnée. De plus, l'opération ne peut se faire en SQL Server 2005 que par code T-SQL. SQL Server 2008 propose un assistant (clic droit sur la table dans l'explorateur d'objet, menu *Storage* à partir de la CTP 6 de février 2008), mais pour rester compatible avec les deux versions, nous vous présentons ici les commandes T-SQL.

Pour partitionner, vous devez suivre les étapes suivantes dans l'ordre :

- Créer autant de groupes de fichiers que vous désirer créer de partitions.
- Créer une fonction de partitionnement (*partition function*). Elle indique sur quel type de données et pour quelles plages de valeurs, les partitions vont être créées.
- Créer un plan de partitionnement (*partition scheme*) (traduit dans l'aide en français par « schéma » de partitionnement). Il attribue chaque partition déclarée dans la fonction, à un groupe de fichiers.
- Créer une table qui se repose sur le plan de partitionnement.

Le partitionnement consiste à placer chaque partition sur son propre groupe de fichiers. Vous devez donc préalablement créer tous vos groupes de fichiers. Si vous cherchez à partitionner une très grande table pour optimiser les lectures physiques, faites au moins en sorte que les fichiers dans lesquels vous allez placer les lignes le plus souvent requêtées se trouvent sur des disques physiques différents.

```
USE Master;

ALTER DATABASE AdventureWorks ADD FILEGROUP fg1;
ALTER DATABASE AdventureWorks ADD FILEGROUP fg2;
ALTER DATABASE AdventureWorks ADD FILEGROUP fg3;
GO
ALTER DATABASE AdventureWorks
ADD FILE
( NAME = data1,
  FILENAME = 'c:\temp\AdventureWorksd1.ndf',
  SIZE = 1MB, maxsize = 100MB, FILEGROWTH = 1MB)
TO FILEGROUP fg1;

ALTER DATABASE AdventureWorks
```

```
ADD FILE
( NAME = data2,
  FILENAME = 'c:\temp\AdventureWorksd2.ndf',
  SIZE = 1MB, maxsize = 100MB, FILEGROWTH = 1MB)
TO FILEGROUP fg2;

ALTER DATABASE AdventureWorks
ADD FILE
( NAME = data3,
  FILENAME = 'c:\temp\AdventureWorksd3.ndf',
  SIZE = 1MB, maxsize = 100MB, FILEGROWTH = 1MB)
TO FILEGROUP fg3;
```

Ceci étant fait, créons une fonction de partitionnement pour notre table `SalesOrderHeader` :

```
USE AdventureWorks;
CREATE PARTITION FUNCTION pfOrderDate (datetime)
AS RANGE RIGHT
FOR VALUES ('20020101', '20030101');
```

Nous passons en paramètre le type de données auquel s'applique la fonction de partitionnement, nous indiquons aussi si les plages vont être créées pour des valeurs situées à gauche ou à droite des frontières exprimées. Ici, par exemple, la dernière partition va contenir tous les `OrderDate` plus grands que le 01/01/2003, c'est-à-dire à droite de cette valeur.

Cette fonction déclare donc trois partitions : tout ce qui vient avant la première valeur, tout ce qui est entre `'20020101'` et `'20030101'`, puis tout ce qui est à droite de `'20030101'`.

Ce que nous pouvons vérifier grâce aux vues de métadonnées :

```
SELECT pf.name, pf.type_desc, pf.boundary_value_on_right, rv.boundary_id,
rv.value, pf.create_date
FROM sys.partition_functions pf
JOIN sys.partition_range_values rv ON pf.function_id = rv.function_id
ORDER BY rv.value;
```

Pour attribuer les partitions aux groupes de fichiers, nous créons un plan de partitionnement :

```
CREATE PARTITION SCHEME psOrderDate
AS PARTITION pfOrderDate
TO (fg1, fg2, fg3);
```

Il nous reste ensuite à créer la table. Il suffit de récupérer le script de création de la table `Sales.SalesOrderHeader` et de le modifier légèrement :

```
CREATE TABLE [Sales].[SalesOrderHeader_Partitionne](
    [SalesOrderID] [int] IDENTITY(1,1) NOT NULL,
    --...
    [ModifiedDate] [datetime] NOT NULL DEFAULT (getdate())
) ON psOrderDate(OrderDate);
```

Comme vous le voyez, au lieu de créer la table sur un groupe de fichiers, nous la créons sur le plan de partitionnement. Nous donnons en paramètre la colonne sur laquelle le partitionnement aura lieu, ce qui porte le nom de clé de partitionnement. Cette clé ne peut être composite : elle doit être contenue dans une seule colonne. Pour que cet exemple fonctionne, nous devons « tricher » et supprimer la clé primaire, car nous ne pouvons pas aligner notre index unique sur le partitionnement de la table. Nous reviendrons sur cette problématique quand nous aborderons le sujet des index au chapitre 6. Sachez simplement que les index peuvent être aussi partitionnés. Par défaut, les index suivent le même plan de partitionnement que la table, ils sont dits « alignés ». Mais ils peuvent aussi être partitionnés différemment, ou n'être pas partitionnés et résider intégralement sur un seul groupe de fichiers. Ce choix explicite doit être indiqué dans l'instruction de création de l'index, comme nous le voyons dans le code précédent.

Alimentons ensuite la table :

```
INSERT INTO Sales.SalesOrderHeader_Partitionne
(    RevisionNumber, OrderDate, DueDate, ShipDate, Status,
     OnlineOrderFlag, PurchaseOrderNumber,
     AccountNumber, CustomerID, ContactID, SalesPersonID,
     TerritoryID, BillToAddressID,
     ShipToAddressID, ShipMethodID, CreditCardID,
     CreditCardApprovalCode, CurrencyRateID,
     SubTotal, TaxAmt, Freight, Comment, rowguid, ModifiedDate
)
SELECT RevisionNumber, OrderDate, DueDate, ShipDate,
    Status, OnlineOrderFlag, PurchaseOrderNumber,
    AccountNumber, CustomerID, ContactID, SalesPersonID,
    TerritoryID, BillToAddressID,
    ShipToAddressID, ShipMethodID, CreditCardID,
    CreditCardApprovalCode, CurrencyRateID,
    SubTotal, TaxAmt, Freight, Comment, rowguid, ModifiedDate
FROM Sales.SalesOrderHeader;
```

Nous pouvons voir ensuite comment les lignes ont été attribuées aux partitions par les vues système d'allocation :

```
SELECT object_name(object_id) AS Name,
       partition_id,
       partition_number,
       rows,
       allocation_unit_id,
       type_desc,
       total_pages
FROM sys.partitions p JOIN sys.allocation_units a
   ON p.partition_id = a.container_id
WHERE object_id=object_id('Sales.SalesOrderHeader_Partitionne')
ORDER BY partition_number;
```

Cette requête est d'utilité publique, et peut être utilisée pour toute table, même non partitionnée. Elle permet de connaître les attributions de pages selon les dif-

férents types (type_desc) : `IN ROW DATA` – Ligne à l'intérieur de la page ; `LOB DATA` – objet large ; `ROW OVERFLOW DATA` – Ligne partagée sur plusieurs pages (voir section 4.1.2).

Vous pouvez également employer directement dans la table une fonction qui renvoie, dans quelle partition doit se trouver chaque ligne (selon la fonction de partitionnement. Elle ne va pas vérifier physiquement si la ligne se trouve dans le bon groupe de fichier). Cela permet de tester le respect des plages :

```
SELECT
       COUNT(*) as cnt,
       MAX(OrderDate) as MaxDate,
       MIN(OrderDate) as MinDate,
       $PARTITION.pfOrderDate(OrderDate) AS Partition
FROM Sales.SalesOrderHeader_Partitionne
GROUP BY $PARTITION.pfOrderDate(OrderDate)
ORDER BY Partition;
```

Le partitionnement permet des facilités d'administration non négligeables. Ajouter, supprimer, déplacer, séparer ou fusionner des partitions peut être pris en charge très rapidement à l'aide des instructions dédiées `SWITCH`, `MERGE` et `SPLIT`. Il n'entre pas dans le cadre de cet ouvrage de vous présenter en détail leur utilisation. Voici pour exemple comment déplacer une partition sur une autre table (existante) :

```
ALTER TABLE Sales.SalesOrderHeader_Partitionne
SWITCH PARTITION 1
TO Sales.SalesOrderHeader_Old;
```

Ces opérations sont pratiquement instantanées, car elles agissent directement au niveau du stockage de la partition, et non en déplaçant les lignes d'une table à une autre. Elles permettent ainsi une administration extrêmement efficace, par exemple pour un archivage régulier.

Attention – La création d'index non alignés empêche l'utilisation de ces instructions. On ne peut pas déplacer une partition si un index non aligné existe sur la table. Dans notre exemple de partitionnement, si nous avions créé une clé primaire sur la colonne `SalesOrderID`, nous aurions dû la faire *nonclustered* et non-alignée, comme ceci : `CONSTRAINT [PK_SalesOrderHeader_Partitionne_SalesOrderID] PRIMARY KEY NONCLUSTERED (SalesOrderID) ON [PRIMARY]`. Cela nous aurait empêchés de réaliser des `SWITCH` après coup.

Enfin, grâce à la possibilité de réaliser des sauvegardes de fichiers ou de groupes de fichiers, la partitionnement de table permet d'optimiser vos stratégies de *backup*.

4.3 TEMPDB

tempdb est la base de données système qui recueille tous les objets temporaires créés dans SQL Server. Elle est détruite et reconstruite à chaque démarrage de l'instance SQL, à partir de la base de données model, tout comme les bases utilisateurs. Si vous voulez changer une option de tempdb, ou sa taille initiale, vous pouvez modifier ces options dans model. tempdb est en mode de récupération simple. Cela ne peut être changé, et il n'y a aucune raison de vouloir mettre tempdb dans un mode complet.

tempdb contient non seulement les objets temporaires (tables, procédures, clés de chiffrement, etc.) créés avec les préfixes # ou ##, et les variables de type table, mais vous pouvez également y créer des objets « persistants », comme dans n'importe quelle base de données. Ces objets y seront conservés jusqu'à leur suppression manuelle par un DROP, ou jusqu'au prochain redémarrage de l'instance.

Outre les objets temporaires, tempdb recueille aussi deux autres types de structures : des **dépôts de version** (*version stores*) pour des lignes impliquées dans du suivi de version (*row versioning*, voir section 4.2.1), et des **objets internes** (*internal objects*). Le *row versioning* est utilisé pour quelques fonctionnalités comme le niveau d'isolation SNAPSHOT, les pseudo-tables de déclencheurs, MARS (*Multiple Active Result Sets*), la génération d'index ONLINE et certaines commandes DBCC comme CHECKDB.

Les objets internes permettent de stocker des données intermédiaires pour tous types d'opérations SQL Server (plans de requête, variables LOB ou XML, résultats de curseurs, messages Service Broker en transit, etc.).

Les objets internes et les dépôts de versions ne sont pas inscrits dans le journal de transaction de tempdb. Ils sont optimisés autant que faire se peut.

> Vous trouverez plus de détails sur leur stockage interne dans le livre blanc « *Working with tempdb in SQL Server 2005* » (http://www.microsoft.com/technet/prodtechnol/sql/2005/workingwithtempdb.mspx).

Les objets utilisateurs (tables temporaires, fonctions table et variables de type table) génèrent, eux, des écritures dans le journal de transactions.

Au fil des versions de SQL Server, tempdb apporte de plus en plus d'optimisations, dans la façon de gérer la journalisation des transactions, l'allocation et la libération des objets. C'est un problème critique, car tempdb subit une grande quantité de créations et destructions d'objets, contrairement à une base de données utilisateur, qui ne subit en utilisation normale que très peu de modifications d'objets. Ainsi, lorsque tempdb est très sollicité, à cause par exemple d'une forte utilisation de tables temporaires dans le code SQL, il peut se produire une contention sur les structures du fichier de données de tempdb (due aux allocations et libérations de pages et d'extensions pour gérer le contenu), ou sur les tables de métadonnées (due à un grand nombre de créations ou de suppressions d'objets). Ce risque de contention est diminué par différentes méthodes (diminution du verrouillage des structures de données, ges-

tion en cache des tables). À partir de SQL Server 2005, les variables de type table et les tables temporaires locales de petite taille sont gardées en cache mémoire, et pour les tables de moins de 8 Mo, la page d'allocation (IAM) et une page de données sont conservées en mémoire au cas où la table serait recréée.

Attention – Certaines pratiques empêchent SQL Server de cacher les tables temporaires. La modification de sa structure après création par exemple, ou si elles sont utilisées dans du code SQL dynamique, ou un batch de requêtes plutôt que dans une procédure stockée.

Si malgré cela, une contention se produit dans `tempdb`, les performances peuvent s'en retrouver sérieusement affectées. C'est une situation rare, rencontrée sur des serveurs très sollicités. Pour détecter une contention, vous pouvez surveiller les attentes à l'aide de la vue de gestion dynamique `sys.dm os waiting tasks`, qui liste les tâches en attentes. La contention des structures de données se traduit sur des attentes de libération de *latches* sur des pages de PFS ou de SGAM. Les *latches* sont des verrous légers de protection d'accès physique (voir section 7.2.1).

```
SELECT session_id, wait_duration_ms, resource_description
FROM sys.dm_os_waiting_tasks
WHERE wait_type like 'PAGE%LATCH_%' AND resource_description like '2:%'
```

Des compteurs du moniteur de performances donnent aussi des indications sur les *latches* en général. Ils se trouvent dans l'objet de compteurs `SQL:Latches` (reportez-vous à la section 5.3).

Vous pouvez lire ce document pour obtenir plus de détails sur l'optimisation de `tempdb` sur http://www.microsoft.com/technet/prodtechnol/sql/2005/workingwithtempdb.mspx

Drapeau de trace 1118

Le document mentionné précédemment évoque le drapeau de trace 1118, qui placé globalement, change la méthode d'allocation de nouveaux objets dans `tempdb` pour éviter la contention sur les SGAM. Ce drapeau de trace est d'usage délicat, et peut être contre performant. Référez-vous à cet article de blog pour une analyse raisonnée : http://sqlblog.com/blogs/linchi_shea/archive/2007/08/10/reduce-the-contention-on-tempdb-with-trace-flag-1118-take-2.aspx

Taille et utilisation de tempdb

Les objets qui peuvent prendre de la place dans `tempdb` sont des objets utilisateur volumineux, comme des tables temporaires. Les tables temporaires sont supprimées à la fin de la session qui les a créées. Si une session reste ouverte, et qu'une table temporaire n'a pas été explicitement supprimée à l'aide d'un `DROP`, elle peut occuper de l'espace un certain temps. C'est donc une habitude importante à prendre, de supprimer une table temporaire lorsqu'on n'en a plus besoin. De même, si la table

temporaire est enrôlée dans une transaction, le journal de transactions de `tempdb` ne pourra se tronquer que lorsque la transaction sera terminée.

Les dépôts de versions peuvent aussi occuper une place importante dans `tempdb`. Ils contiennent chaque ligne impliquée dans un *row versioning*. Il existe uniquement deux dépôts de versions dans `tempdb` : l'un est dédié aux opérations sur index ONLINE, et l'autre à toutes les utilisations de *row versioning*. Nous verrons dans le chapitre sur les index ce qu'impliquent les opérations d'index ONLINE, sachez simplement que plus la création de l'index en mode ONLINE prendra de temps, plus le dépôt de versions est susceptible de grandir. L'autre dépôt de versions est susceptible de grandir avec le nombre d'opérations impliquant du *row versioning*, et dans le cas où une transaction est maintenue, qui utilise une fonctionnalité de *row versioning*.

La vue de gestion dynamique `sys.dm tran active snapshot database transactions` permet d'obtenir la liste toutes les transactions qui génèrent, ou peuvent potentiellement accéder à des versions de lignes. Elle peut vous servir à détecter des transactions longues qui maintiennent des versions :

```
SELECT *
FROM sys.dm_tran_active_snapshot_database_transactions
ORDER BY elapsed_time_seconds DESC;
```

La vue de gestion dynamique `sys.dm_db_file_space_usage`, affiche l'utilisation des fichiers de données. Comme son nom ne l'indique pas, elle s'applique uniquement à `tempdb`. Donc, quel que soit le contexte de base dans lequel vous l'exécutez, elle affichera les informations de `tempdb`. Elle donne le nombre de pages dédiées aux dépôts de versions (`version store reserved page count`), aux objets internes (`internal object reserved page count`) et aux objets utilisateur (`user object reserved page count`), par fichier de données. Pour obtenir le volume en octets, il vous suffit de multiplier ces nombres par 8192. Exemple :

```
SELECT
  SUM(user_object_reserved_page_count)*8 as user_object_kb,
  SUM(internal_object_reserved_page_count)*8 as internal_object_kb,
  SUM(version_store_reserved_page_count)*8 as version_store_kb
FROM sys.dm_db_file_space_usage;
-- ou en Mo
SELECT
SUM(user_object_reserved_page_count)*1.0/128 as user_object_mb,
SUM(internal_object_reserved_page_count)*1.0/128 as internal_object_mb,
SUM(version_store_reserved_page_count)*1.0/128 as version_store_mb
FROM sys.dm_db_file_space_usage;
```

La vue `sys.dm db session space usage` est intéressante car elle indique le nombre d'allocations et de déallocations d'objets internes et d'objets utilisateur par session. Si vous souhaitez connaître l'impact d'une procédure ou d'un batch de requêtes sur `tempdb`, exécutez le code voulu, puis, dans la même session, lancez une requête comme celle-ci :

```
SELECT *
FROM sys.dm_db_session_space_usage
WHERE session_id = @@SPID;
```

La vue `sys.dm db task space usage` donne la même information par tâche en activité. Elle peut être jointe à `sys.dm exec requests` pour trouver le plan d'exécution :

```
SELECT
    tsu.*,
    DB_NAME(er.database_id) as db,
    er.cpu_time, er.reads, er.writes, er.row_count,
    eqp.query_plan
FROM sys.dm_db_task_space_usage tsu
JOIN sys.dm_exec_requests er
    ON tsu.session_id = er.session_id AND
       tsu.request_id = er.request_id
CROSS APPLY sys.dm_exec_query_plan(er.plan_handle) eqp;
```

Plus d'informations avec des exemples de supervision vous sont données dans les BOL, sous l'entrée « Résolution des problèmes d'espace disque insuffisant dans `tempdb` » (*Troubleshooting Insufficient Disk Space in tempdb*).

Modification de l'emplacement de tempdb

Vous pouvez modifier l'emplacement des fichiers de données et de journal de `tempdb`. Comme la base est recréée à chaque démarrage, vous devez appliquer les commandes suivantes, puis redémarrer votre instance pour que le changement soit effectif.

```
ALTER DATABASE tempdb
MODIFY FILE (name = tempdev, filename = 'E:\Sqldata\tempdb.mdf')
GO
ALTER DATABASE tempdb
MODIFY FILE (name = templog, filename = 'E:\Sqldata\templog.ldf')
GO
```

Après redémarrage du service, vous pouvez exécuter `sp_helpfile tempdb` ou `SELECT * FROM tempdb.sys.database_files` pour vérifier que le changement est effectif.

4.4 CONTRÔLE DE L'ATTRIBUTION DES RESSOURCES

Gouverneur de requêtes

SQL Server intègre un contrôleur de coût de requête, qui permet d'éviter l'exécution d'instructions qui sont – avant leur exécution – estimées lourdes. Il est nommé **gouverneur de requêtes** (*query governor*). Vous le configurez pour toute l'instance :

```
exec sp_configure 'show advanced options', 1
reconfigure
exec sp_configure 'query governor cost limit', 10
reconfigure
exec sp_configure 'show advanced options', 0
reconfigure
```

ou pour une session :

```
SET QUERY_GOVERNOR_COST_LIMIT 10;
```

Le nombre passé en paramètre correspond au nombre de secondes estimées par le moteur SQL, par rapport à une configuration de machine de test de Microsoft : il ne s'agit pas de vraies secondes de votre serveur. L'estimation elle-même ne compte bien entendu pas les attentes diverses auxquelles l'exécution peut être soumise. Il s'agit donc d'une estimation très… estimée, mais cela peut offrir un premier niveau de contrôle de charge de votre serveur.

Si une instruction est demandée, dont l'estimation dépasse ce nombre, la session recevra en retour le message d'erreur suivant, et l'exécution sera annulée avant même de commencer.

```
Msg 8649, Level 17, State 1...
The query has been canceled because the estimated cost of this query
(1090057) exceeds the configured threshold of 10.
Contact the system administrator.
```

Au programme client de récupérer l'erreur...

SQL Server 2008 : le gouverneur de ressources

SQL Server 2008 offre un outil d'attribution de ressources système (CPU et mémoire), par session, nommé le **gouverneur de ressources** (*resource governor*). Il permet de configurer ces ressources en *pool* (*resource pools*), qui indiquent quelles sont les limites de ressources attribuables à un groupe de charges de travail. Deux *pools* système existent, le *pool* par défaut (*default*), et le *pool* interne (*internal*). Vous pouvez créer vos propres *pools* afin de limiter des charges de travail. La figure 4.12 présente l'interface de création de *pools*.

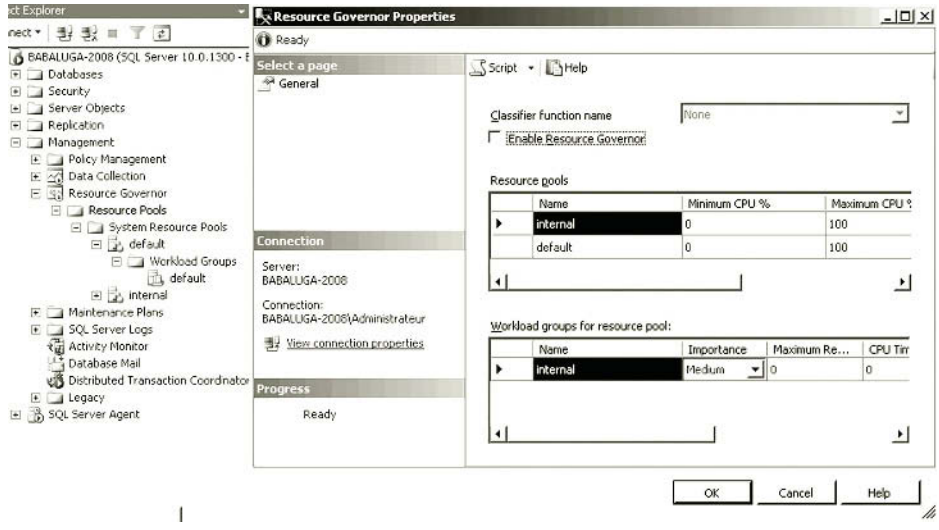

Figure 4.12 — Configurer le gouverneur de ressources

Nous pouvons aussi créer un *pool* par code :

```
CREATE RESOURCE POOL [intrus]
WITH( min_cpu_percent=0,
      max_cpu_percent=20,
      min_memory_percent=0,
      max_memory_percent=10)
```

Lors de chaque modification du gouverneur de ressources, vous devez activer la modification :

```
ALTER RESOURCE GOVERNOR RECONFIGURE;
```

Ainsi, le gouverneur de ressources est activé. Si vous souhaitez le désactiver (les sessions ne seront ainsi plus contrôlées) :

```
ALTER RESOURCE GOVERNOR DISABLE;
```

Un `ALTER RESOURCE GOVERNOR RECONFIGURE` le réactive.

Les ressources sont partagées entre les pools au prorata des valeurs indiquées, selon un algorithme décrit dans l'entrée des BOL 2008 « Concepts du gouverneur de ressources » (*Resource Governor Concepts*).

Dans les *pools* sont placés des groupes de charges de travail. Ces groupes rassemblent simplement des instructions SQL lancées sur le serveur, qui sont attribuées à ce groupe automatiquement selon des règles de classification. Voici un exemple de création de groupe :

```
CREATE WORKLOAD GROUP [méchantes requêtes]
WITH( group_max_requests=10,
      importance=Low,
      request_max_cpu_time_sec=0,
      request_max_memory_grant_percent=25,
      request_memory_grant_timeout_sec=0,
      max_dop=1)
USING [intrus]
GO
ALTER RESOURCE GOVERNOR RECONFIGURE;
```

Comme vous le voyez, vous pouvez définir l'importance du groupe et le degré de parallélisme (`max_dop`).

Vous créez ensuite une fonction de classification, que le gouverneur de ressources va appeler pour attribuer chaque nouvelle ouverture de session à un groupe. Vous ne pouvez créer qu'une seule fonction, qui effectuera le routage global. En l'absence de fonction de classification, toutes les sessions utilisateurs seront attribuées au groupe `default`. Lorsque la fonction est écrite, vous la déclarez au gouverneur de ressources, comme dans l'exemple suivant :

```
CREATE FUNCTION dbo.rgclassifier_v01() RETURNS SYSNAME
WITH SCHEMABINDING
AS
BEGIN
    DECLARE @grp_name AS SYSNAME
```

```
      IF (SUSER_NAME() = 'Nicolas') or (APP_NAME() LIKE '%Elysee%')
          SET @grp_name = 'méchantes requêtes'
      ELSE
          SET @grp_name = NULL
    RETURN @grp_name
END
GO
-- Register the classifier function with Resource Governor
ALTER RESOURCE GOVERNOR WITH (CLASSIFIER_FUNCTION= dbo.rgclassifier_v01)
GO
ALTER RESOURCE GOVERNOR RECONFIGURE
GO
```

Lorsque la fonction retourne `NULL`, la session est attribuée au groupe par défaut.

L'appartenance des sessions au groupe peut s'obtenir ainsi :

```
SELECT *
FROM sys.dm_exec_sessions es
JOIN sys.resource_governor_workload_groups rswg
    ON es.group_id = rswg.group_id;
```

Vous pouvez déplacer dynamiquement un groupe d'un *pool* à un autre sans perturber les sessions actives :

```
ALTER WORKLOAD GROUP [méchantes requêtes]
USING [default];
GO
ALTER RESOURCE GOVERNOR RECONFIGURE
GO
```

Vous disposez de compteurs de performances indiquant l'état du gouverneur de ressources (entrée BOL 2008 « *Resource Governor Monitoring* »), ainsi que des vues de métadonnées et de gestion dynamique (entrée BOL 2008 « *Resource Governor DDL and System Views* »).

5

Analyse des performances

Objectif

Vous ne pouvez agir sans savoir. L'amélioration de performance doit se fonder sur les faits, et les améliorations doivent être vérifiables ensuite. Les outils de surveillance et de suivi vous donnent les éléments indispensables à la compréhension du problème, donc à sa résolution. Ce chapitre présente les outils d'analyse à votre disposition.

5.1 SQL SERVER MANAGEMENT STUDIO

SSMS (*SQL Server Management Studio*) propose plusieurs outils pour analyser la performance de vos requêtes SQL. Un plan d'exécution graphique est affichable, nous en parlerons en détail section 8.1. Nous présentons ici les autres options.

Les **statistiques du client** sont des chiffres collectés par la bibliothèque d'accès aux données, et qui sont visibles en tableau dans SSMS. Elles vous permettent de vous faire une idée des instructions exécutées, du nombre de transactions, et surtout du volume de données qui a transité jusqu'à vous pour un batch de requêtes. Lorsque vous activez cette fonctionnalité, à l'aide du bouton de la barre d'outils illustré sur la figure 5.1, vous obtenez un nouvel onglet de résultat.

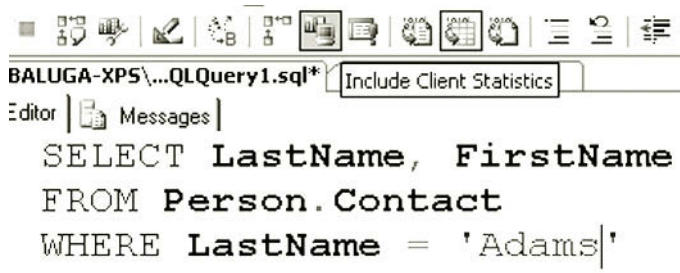

Figure 5.1 — Afficher les statistiques clients dans SSMS

Si vous exécutez le même batch plusieurs fois, vous obtenez un tableau comparatif des statistiques d'exécution, avec des indicateurs fléchés vous indiquant la tendance par rapport à l'exécution précédente (figure 5.2).

Editor | Results | Messages | Client Statistics

	Trial 2	Trial 1	Average
Client Execution Time	13:14:46	13:14:40	
Query Profile Statistics			
Number of INSERT, DELETE and UPDATE statements	0	→ 0	→ 0.0000
Rows affected by INSERT, DELETE, or UPDATE statem...	0	→ 0	→ 0.0000
Number of SELECT statements	1	→ 1	→ 1.0000
Rows returned by SELECT statements	0	→ 0	→ 0.0000
Number of transactions	0	→ 0	→ 0.0000
Network Statistics			
Number of server roundtrips	1	→ 1	→ 1.0000
TDS packets sent from client	1	→ 1	→ 1.0000
TDS packets received from server	1	→ 1	→ 1.0000
Bytes sent from client	214	↑ 180	→ 197.0000
Bytes received from server	2368	→ 2368	→ 2368.0000
Time Statistics			
Client processing time	0	→ 0	→ 0.0000
Total execution time	15	↑ 0	→ 7.5000
Wait time on server replies	15	↑ 0	→ 7.5000

Figure 5.2 — Résultat des statistiques clients dans SSMS

Vous pouvez réinitialiser ces statistiques à l'aide de la commande « Reset Client Statistics » du menu Query.

L'option de session `SET STATISTICS TIME ON` retourne les statistiques de temps d'exécution, visibles dans l'onglet « Messages ». Voici un exemple de retour :

```
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 1 ms.
```

```
SQL Server Execution Times:
   CPU time = 16 ms,  elapsed time = 22 ms.
```

Les statistiques de temps ne sont pas des mesures précises : elles changent à tout moment, selon la charge du serveur, du réseau et les verrous. Néanmoins cela vous donne une idée du temps de compilation, et la différence entre le temps CPU et le temps total (*elapsed*) vous indique des attentes sur les verrous, les entrées/sorties et le réseau.

L'option de session `SET STATISTICS IO ON` retourne les statistiques d'entrées/sorties fournies par le moteur SQL Server, en unité de pages. Elles sont une mesure plus précise, car elles indiquent de façon constante le vrai coût d'une requête pour le moteur de stockage. Exemple de retour :

```
Table 'Contact'. Scan count 1, logical reads 1276, physical reads 0,
read-ahead reads 0, lob logical reads 0, lob physical reads 0,
lob read-ahead reads 0.
```

Vous voyez le nombre de *scan* de table ou d'index (mal traduit dans la version française par « analyse »), le nombre de reads logiques, de reads physiques (les appels aux pages non trouvées dans le *buffer*), les lectures anticipées (SQL Server peut appeler à l'avance des pages qu'il prévoit d'utiliser plus tard dans l'exécution de la requête), et les lectures de pages LOB (objets larges et *row overflow*).

Les résultats de `SET STATISTICS IO ON`, comme le texte des messages d'erreur, sont récupérables par trace, dans l'événement `Errors and Warnings : User Error Messages`.

Vous pouvez activer ces options pour toutes les sessions en les activant automatiquement dans les options SSMS (figure 5.3).

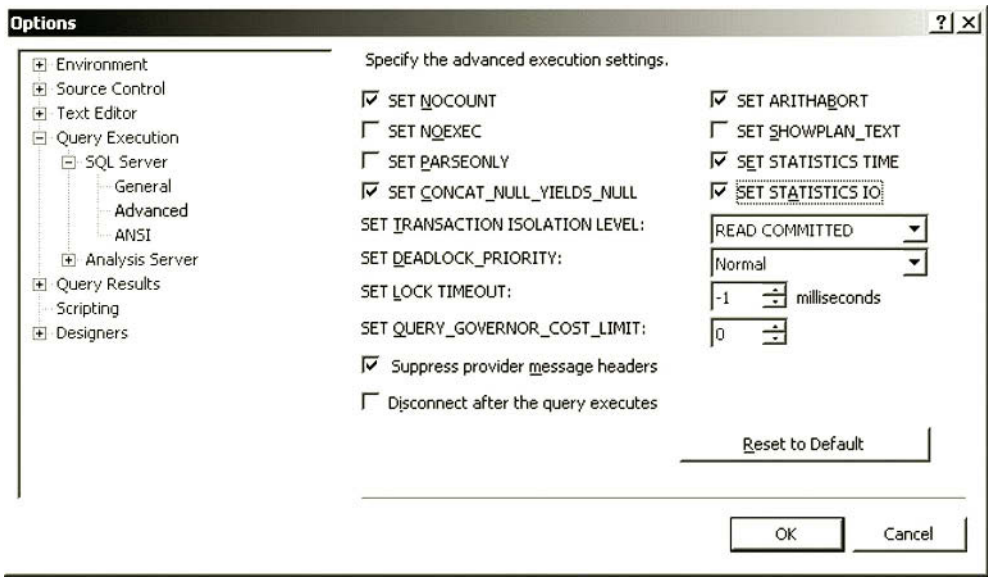

Figure 5.3 — Options de la session SSMS

5.2 SQL TRACE ET LE PROFILER

SQL Trace est une technologie intégrée au moteur SQL Server, qui permet de fournir à un client le détail de quantité d'événements se produisant dans les différentes parties de SQL Server.

SQL Trace peut être comparé à un débogueur. Devant un programme complexe, organisé en multiples classes et modules, un développeur qui doit identifier la cause d'un problème à partir d'un rapport de *bug* n'a qu'une méthode efficace à disposition : reproduire le problème en suivant pas à pas le comportement du code à l'aide de son outil de débogage, observant les valeurs attribuées aux variables, les changements d'état des classes, l'itération des boucles. Sans cette capacité à entrer dans les opérations du code, à observer finement ce qui se passe, le travail de débogage serait réduit à une recherche fastidieuse faite d'essais, d'instructions `PRINT`, de lecture attentive de code. Un serveur SQL ajoute des niveaux de complexité supplémentaires, qui rendent la méthode expérimentale cauchemardesque : on peut rarement deviner à la simple lecture d'une requête quel en est le plan d'exécution, c'est-à-dire la stratégie que va appliquer SQL Server pour en assurer l'exécution optimale. Pour le savoir, il faudrait faire soi-même tout le travail de l'optimiseur : inspecter les tables, lister les index et leurs statistiques, estimer le nombre de lignes impactées, essayer plusieurs stratégies, estimer leur coût… Il est donc pratiquement impossible de deviner ce qui va précisément se passer. De plus, l'exécution concurrentielle des requêtes rend les performances dépendantes du contexte : charge de travail des processeurs, attente sur des verrous, performance des tables temporaires…

Sans SQL Trace, SQL Server serait une boîte noire, nous n'aurions aucun moyen d'identifier précisément les raisons d'un ralentissement soudain, ou d'une charge excessive du système.

Dans chaque instance de SQL Server, un sous-système nommé le **contrôleur de trace** (*trace controller*) est responsable de la collecte d'événements produits par des fournisseurs propres à chaque partie du moteur, et à l'envoi de ces événements aux clients inscrits. Ces fournisseurs ne sont activés que si au moins un client écoute activement un des événements qu'ils produisent. Le contrôleur de trace distribue ensuite les résultats à des fournisseurs d'entrées/sorties (*trace I/O providers*), ce qui permet à une trace d'être directement sauvée dans un fichier binaire par le serveur, ou envoyée à un client, comme le profiler.

Le **profiler**, incorrectement traduit dans certaines versions de SQL Server par « gestionnaire de profils » est donc en réalité un programme client qui affiche les informations d'une trace SQL. Il est l'outil incontournable de l'optimisation. Il vous permet de suivre le comportement de votre serveur, en interceptant tous les événements envoyés par SQL Trace, dont bien entendu les requêtes SQL, accompagnés de précieuses informations sur leur impact et leur exécution. Il est important de bien le connaître et nous allons donc en détailler les fonctionnalités.

Le profiler peut afficher en temps réels un flux de trace SQL, mais aussi charger une trace enregistrée préalablement dans un fichier de trace ou dans une table SQL. Il peut sauvegarder une trace dans ces deux destinations, et également rejouer une trace sur un serveur SQL, c'est-à-dire exécuter à nouveau toutes les requêtes SQL contenues dans la trace, ce qui est fort utile pour faire un test de charge, par exemple.

Vous en trouvez l'icône dans le menu de SQL Server, dossier *Performances tools* (outils de performance). Lorsque vous créez une nouvelle session, une fenêtre à deux onglets permet de sélectionner un certain nombre d'options.

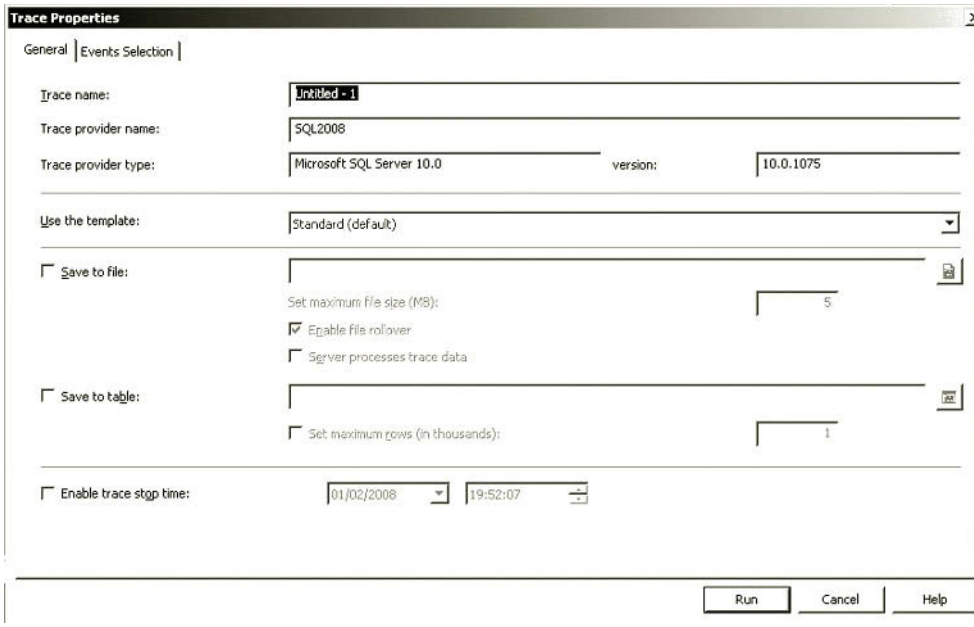

Figure 5.4 — Premier onglet du profiler

Dans l'onglet *General*, vous pouvez nommer votre trace, ce qui est utile lorsque vous en créez plusieurs (le profiler est une application MDI (*Multiple Documents Interface*) qui peut exécuter plusieurs traces en même temps), et pour prédéfinir le nom du fichier lorsque vous sauvegarderez la trace. En grisé, vous pouvez voir le nom et la version du fournisseur SQL Trace. Les profilers 2005 et 2008 peuvent lancer des traces sur des versions antérieures de SQL Server. Dans ce cas, les événements et colonnes à disposition reflètent les possibilités de la version du serveur.

Vous pouvez fonder votre nouvelle trace sur un modèle (*template*), c'est-à-dire un squelette de trace comprenant toutes les propriétés choisies dans le profiler, y compris les choix d'événements, les colonnes et les filtres. Vous pouvez utiliser le choix de modèles prédéfini, mais aussi créer vos modèles personnalisés en sauvant votre trace comme nouveau modèle.

Vous pouvez choisir de visualiser votre trace en temps réel, en laissant les événements se dérouler devant vos yeux, et vous pouvez aussi sauvegarder cette trace dans un fichier binaire ou XML, ou dans une table SQL Server, au fur et à mesure de son exécution en indiquant la destination dans cet onglet, ou ensuite, à l'aide de la commande *Save As…* du menu *File*.

L'écriture à la volée d'une trace volumineuse dans une table peut être très pénalisante pour le serveur SQL. Préférez l'enregistrement en fichier binaire. Vous aurez ensuite la possibilité de la recharger dans le profiler et de le sauvegarder dans

une table SQL au besoin. Si vous choisissez néanmoins d'écrire directement dans une table, faites-le de préférence sur un serveur différent de celui tracé.

Vous pouvez indiquer une taille maximale du fichier de trace et segmenter vos fichiers de trace avec l'option *enable file rollover*. Lorsque la taille maximale indiquée est atteinte, un nouveau fichier numéroté est créé. Si votre fichier est nommé `matrace.trc`, les suivants seront `matrace_1.trc`, `matrace_2.trc`, et ainsi de suite. Si vous décochez l'option *enable file rollover*, l'enregistrement s'arrêtera lorsque la taille sera atteinte. La visualisation en temps réel, elle, se poursuivra. Pensez à l'affichage dans le profiler et à l'enregistrement de la trace comme des redirections séparées d'une même trace. Si vous choisissez un enregistrement dans une table, vous pouvez limiter le nombre de milliers de lignes à y écrire, avec l'option *Set maximum rows*.

Si votre objectif n'est pas d'analyser en temps réel l'activité de votre serveur, mais de laisser tourner votre trace pour, par exemple, établir une *baseline*, vous pouvez indiquer un arrêt automatique de la trace avec l'option *Enable trace stop time*. Nous verrons qu'il est préférable dans ce cas de lancer une trace directement sur le serveur.

L'onglet « Events Selection » est le cœur du profiler : il permet de choisir les informations retournées : événements, colonnes, et filtres.

Les classes d'événements récupérables sont groupées en catégories. Elles publient un certain nombre d'informations présentées dans des colonnes. Toutes les colonnes ne sont pas alimentées pour chaque classe. Vous trouverez les colonnes utiles dans les BOL, sous l'entrée correspondant à l'événement.

Par défaut, et pour simplifier l'affichage, cet onglet ne vous montre que les événements sélectionnés dans le modèle choisi. Pour faire votre sélection parmi tous les événements disponibles, cochez *Show all events*. Vous obtenez ainsi une liste des catégories et des classes d'événements associés. Les catégories sont reproduites dans le tableau 5.1.

Tableau 5.1 — Catégories d'événements de trace
* Génère une activité importante

Catégorie	Description
Broker	Événements Service Broker.
CLR	Exécution des objets .NET intégrés à SQL Server.
Cursors	Opérations de curseurs T-SQL.
Database	Changement de taille des fichiers de base de données, et événements de mirroring.
Depreciation	Alertes d'utilisation de fonctionnalités obsolètes.
Errors and Warnings	Avertissements et exceptions diverses.

Catégorie	Description
Full Text	Alimentation des index de texte intégral (FTS).
Locks	Acquisition, escalade, libération et *timeout* de verrous*, et alertes de verrous mortels (*deadlocks*).
OLEDB	Appels OLEDB effectués par SQL Server en tant que client : par exemple lors de requêtes distribuées sur des serveurs liés.
Objects	Création, modification et suppression d'objets de base (Ordres DDL `CREATE`, `ALTER`, `DROP`)
Performance	Informations diverses de performance, notamment les plans d'exécution d'ordres SQL.
Progress Report	Information d'avancement d'une opération d'indexation en ligne (*online indexing*). Voir chapitre 6.
Query Notifications	Événements générés par la fonctionnalité *Query Notifications*. Voir encadré en fin de chapitre 8.
Scans	*Scans* de tables et d'index.
Security Audit	Ouvertures et fermeture de sessions, échec de connexion et divers audit de privilèges et d'exécution de commandes (`BACKUP`, `DBCC...`).
Server	Principalement, changement de mémoire de l'instance.
Sessions	Sessions ouvertes au moment du démarrage de la trace.
Stored Procedures	Exécution de procédures stockées (T-SQL seulement), et détail des ordres exécutés dans la procédure.
TSQL	Ordres SQL envoyés au serveur en batch. Aussi instructions XQuery.
Transactions	Détail de la gestion des transactions.
User Configurable	Événements utilisateur.

Certaines classes d'événements sont très spécifiques et ne vous seront que d'une utilité occasionnelle. Elles sont toutes décrites en détail dans les BOL, sous l'entrée « *SQL Server Event Class Reference* », avec, pour chaque événement, les colonnes retournées. Nous présentons dans cet ouvrage les événements importants pour l'analyse de performance, ceux que vous serez amenés à utiliser souvent.

De même, si nous nous concentrons sur les performances, quelques colonnes seulement sont utiles. Vous trouverez la liste des colonnes disponible dans les BOL, sous l'entrée « *Describing Events by Using Data Columns* ». Nous distinguerons les colonnes contenant des valeurs indiquant les performances, de celles affichant des informations sur l'événement, et qui vous permettront par exemple de grouper ou de filtrer les résultats.

Vous pouvez générer vos propres événements dans votre code SQL, à l'aide de la procédure stockée `sp_trace_generateevent`. Vous pouvez passer dix événements différents, numérotés en interne de 82 à 91 (ce sont donc ces ID que vous passez en paramètre à la procédure `sp_trace_generateevent`), et que vous retrouvez dans le profiler sous `User Configurable : UserConfigurable:0` à `UserConfigurable:9`. Voici un exemple de déclenchement d'événement :

```
DECLARE @userdata varbinary(8000)
SET @userdata = CAST('c''est moi' as varbinary(8000))

EXEC sys.sp_trace_generateevent
      @event_class = 82,
      @userinfo = N'J''existe !!',
      @userdata = @userdata
```

Cet événement sera visible comme `UserConfigurable:0`. Le contenu de `@userinfo` sera affiché dans la colonne `TextData` et le contenu de `@userdata` dans la colonne `BinaryData`.

Événements

Les événements les plus utiles sont :

- **Database\Data File Auto Grow** : une augmentation automatique de la taille d'un fichier de données s'est produite. La base de données affectée est visible dans la colonne `DatabaseID`.
- **Database\Log File Auto Grow** : une augmentation automatique de la taille d'un fichier de journal s'est produite. La base de données affectée est visible dans la colonne `DatabaseID`.
- **Errors And Warnings\Attention** : se déclenche lorsqu'un client est déconnecté brusquement.
- **Errors And Warnings\Exception** : erreurs retournées par SQL Server au client, plus de détails plus loin.
- **Errors And Warnings\Execution Warnings** : avertissements déclenchés par SQL Server à l'exécution de la requête.
- **Errors And Warnings\Missing Column Statistics** : type spécial d'avertissement, il manque des statistiques sur une colonne (voir la section 6.4).
- **Errors And Warnings\Sort Warnings** : avertissement sur une opération de tri qui n'a pu être complètement exécutée en mémoire, et a donc « bavé » dans `tempdb` (voir la section 8.1.2).
- **Locks** : événements sur les verrous. Les événements de pose de verrous sont trop fréquents pour être utiles. Les événements de timeout vous permettent de tracer des attentes trop longues annulées par une valeur de LOCK_TIMEOUT de la session (voir la section 7.2). Les événements de verrou mortel (*deadlock*) seront détaillés dans la section 7.5.2).
- **Performance\Auto Stats** : création ou mise à jour automatique de statistiques (voir section 6.4).

- **Performance\ShowPlan…** : affichage du plan d'exécution (voir section 8.1).
- **Server\Server Memory Change** : modification de la mémoire vive utilisée par SQL Server, peut être utile pour tracer une pression mémoire, en alerte de l'agent SQL, par exemple ;
- **Stored Procedures\RPC:Completed** : un appel RPC (*Remote Procedure Call* – appel de procédure depuis un client, avec la syntaxe `CALL`, ou un objet `StoredProcedure` de la bibliothèque cliente). Contient les statistiques d'exécution (durée, lectures…).
- **Stored Procedures\SP:Completed** : fin de l'exécution d'une procédure stockée. Contient les statistiques d'exécution (durée, lectures…).
- **Stored Procedures\SP:Recompile** : recompilation d'une procédure ou d'une partie de celle-ci (voir la section 9.1).
- **Stored Procedures\SP:StmtCompleted** : analyse de l'exécution de chaque instruction d'une procédure stockée. Contient les statistiques d'exécution (durée, lectures…).
- **TSQL\SQL:BatchCompleted** : fin de l'exécution d'un batch de requêtes. Contient les statistiques d'exécution (durée, lectures…).
- **TSQL\SQL:StmtCompleted** : fin de l'exécution d'une instruction dans un batch. Contient les statistiques d'exécution (durée, lectures…).

Colonnes

Les **colonnes** peuvent servir à filtrer ou regrouper vos événements. Vous pouvez créer de multiples filtres. Sans ces filtres, une trace en production est pratiquement illisible : à peine l'avez-vous démarrée, que des milliers, voir des dizaines de milliers d'événements sont déjà tracés. **Filtrer** est non seulement indispensable pour la visibilité des informations, mais pour diminuer l'impact de la trace. En effet, le filtre sera appliqué du côté du serveur, et allégera donc la trace.

Pour filtrer vous pouvez soit cliquer sur le bouton « Column filters… » de l'onglet « Events Selection… » de la fenêtre de propriétés de la trace, soit en cliquant directement sur l'en-tête d'une colonne dans cette même fenêtre. Une boîte de dialogue de filtre (figure 5.5) s'ouvre.

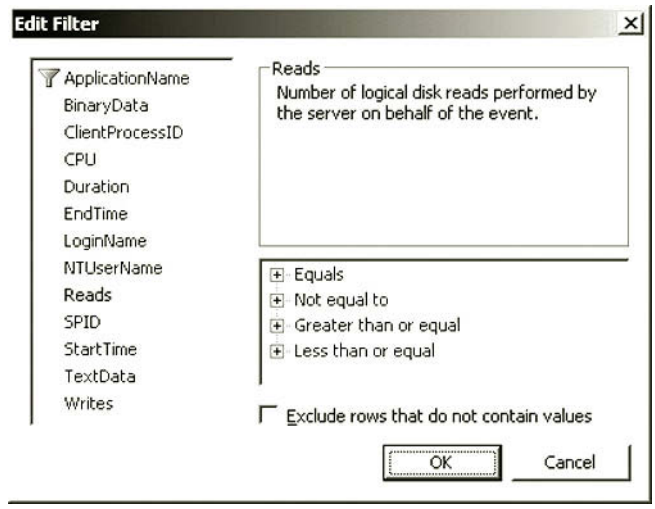

Figure 5.5 — Les filtres de colonnes

Vous pouvez accumuler les filtres sur une même colonne, avec différents opérateurs. Les colonnes filtrées montrent une icône d'entonnoir. Avant SQL Server 2005, les événements qui ne retournaient rien dans la colonne filtrée étaient affichés quand même, ce qui était assez peu pratique. Depuis SQL Server 2005, l'option « Exclude rows that do no contain values » permet de filtrer ces événements. Pensez-y, elle n'est pas sélectionnée par défaut.

Quelques idées de filtre : filtre sur un `SPID` particulier pour tracer une seule session, filtre sur une `ApplicationName` pour analyser ce que fait une application cliente, filtre sur `TextData` pour trouver les instructions SQL qui s'appliquent à un objet. Dans ce cas, la syntaxe du `LIKE` est la même qu'en SQL : le caractère `%` (pourcent) remplace une chaîne de caractères. Pour une trace destinée à détecter les requêtes les plus coûteuses, un filtre sur le nombre de reads (opérateur égal ou plus grand que) est la meilleure méthode : les reads ne varient pas d'un appel à un autre à données égales, et plus le nombre de reads est important, plus l'instruction est lourde, non seulement pour sa propre exécution, mais aussi, à travers les ressources qu'elle consomme et les verrous qu'elle pose peut-être, pour toutes les autres s'exécutant simultanément.

Vous pouvez aussi grouper à l'affichage, par colonne(s). Cela vous permet d'obtenir une vue non plus chronologique, mais dans un ordre qui vous siet mieux. Dans ce cas, la lecture peut être moins intuitive. Cliquez sur le bouton « Organize columns… » (ou clic droit sur les en-têtes de colonne, et commande « organize columns… »). Dans la fenêtre qui s'ouvre, les boutons Up et Down permettre d'organiser l'ordre des colonnes. Un Up jusqu'au nœud « Groups » crée un regroupement. Sur la figure 5.6, vous voyez un regroupement par `ApplicationName`, puis `SPID`.

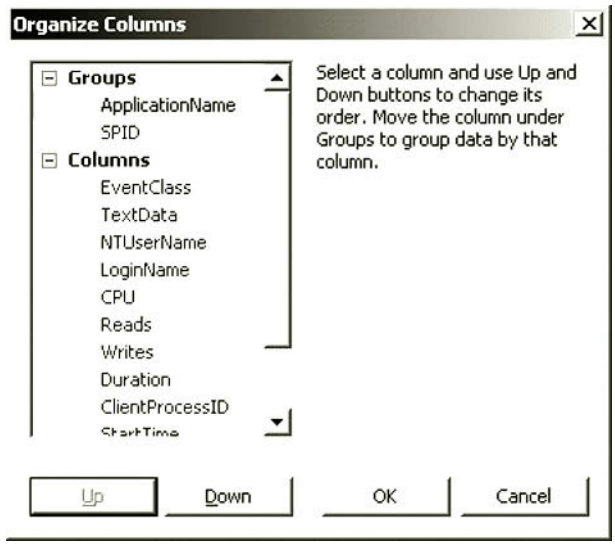

Figure 5.6 — Regroupement par colonnes

L'affichage en temps réel triera les événements par ces colonnes de regroupement, vous n'aurez donc plus une vision strictement chronologique. De même, ces colonnes seront toujours visibles à gauche, même si vous défilez sur la droite pour afficher plus de colonnes. Si vous ne sélectionnez qu'une seule colonne de groupement, vous obtiendrez une vision arborescente, réduite par défaut.

Colonnes utiles aux performances

- **CPU** : indique le temps processeur consommé par l'événement, en millisecondes. Si ce temps processeur est supérieur à la valeur de `Duration`, votre requête a été parallélisée sur plusieurs CPU.
- **Duration** : indique la durée totale d'exécution de l'événement, y compris le temps d'envoi du résultat au client.
- **Reads** : indique le nombre de pages lues par la requête. Il s'agit bien d'unités de pages de base de données, de 8 Ko chacune. Pour obtenir le nombre d'octets lus, vous pouvez multiplier cette valeur par 8192. Notez que vous ne pouvez pas obtenir le détail séparé des pages lues du *buffer* et récupérées du disque qui forment ce nombre total de *reads*. Cette information n'est disponible que dans les statistiques récupérées en exécutant une requête avec l'option de session `SET STATISTICS IO ON` dans un client comme SSMS, ou en corrélant vos requêtes avec des compteurs du moniteur de performances.
- **RowCounts** : nombre total de lignes affectées (lues, insérées, modifiées, supprimées) par l'ordre, le batch, ou la procédure.
- **TextData** : texte de l'ordre SQL exécuté.
- **Writes** : nombre de pages de données écrites.

Ces colonnes indiquant des mesures chiffrées de l'exécution d'ordres SQL, elles ne sont bien entendu retournées que par les événements déclenchés à la fin d'un ordre. Dans les couples d'événements tels que `SQL:BatchStarting` et `SQL:BatchCompleted` ou `SP:Starting` et `SP:Completed`, c'est l'événement *Completed* qui vous intéressera. L'événement *Starting* ne sera utile que si vous désirez observer ce qui se passe entre le début et la fin de l'ordre (par exemple à l'aide de l'événement `SP:StmtCompleted` qui détaille l'exécution des parties d'une procédure stockée), ou si, à cause d'une exception, votre ordre ne se termine pas correctement.

Attention – Les colonnes `CPU` et `Duration` enregistrent en interne des durées en microsecondes (un millionième de seconde, 10 puissance -6). Elles sont affichées dans le profiler en millisecondes, comme pour les versions précédentes de SQL Server, mais lorsque vous travaillez hors du profiler, par exemple dans une trace sauvegardée dans une table, ces valeurs seront en microsecondes. Vous devez donc diviser par 1000 pour obtenir des millisecondes.

D'autres colonnes sont indicatives, et utiles également pour filtrer vos événements :

- **ApplicationName** : nom de l'application cliente, passée dans la chaîne de connexion. Par exemple SSMS se fait connaître sous le nom *Microsoft SQL Server Management Studio*. Si vous développez des applications clients « maison », il est utile de le spécifier, soit dans la chaîne de connexion (en y ajoutant « `Application Name=` »), soit à l'aide de la propriété idoine de votre bibliothèque client (par exemple la propriété `ApplicationName` de l'objet `SqlConnectionStringBuilder` en ADO.NET), cela vous permettra de savoir d'où viennent vos requêtes, et d'afficher les événements provenant d'une application seulement. Les développeurs peuvent aussi générer cet `ApplicationName` en y incluant le numéro de version de leur produit. Cela rend aisé la vérification des mises à jour sur tous les clients, et cela permet, par exemple, de créer un déclencheur DDL qui vérifie cette valeur à l'aide de la fonction système `APP_NAME()` et interdit l'ouverture de session si la version du client n'est pas suffisante.
- **DatabaseID** : l'identifiant de base de données courante, utile pour n'afficher que les requêtes effectuées dans une base de données. Vous pouvez trouver cet ID dans la colonne `database_id` de la vue système `sys.databases`, ou à l'aide de la fonction `DB_ID()`. Notez qu'il s'agit du contexte de base dans lequel l'ordre est exécuté, et non la localisation des objets touchés. Si vous requêtez la table `AdventureWorks.Person.Contact` depuis la base `master`, en utilisant comme ici le nom complet base.schéma.objet, votre `DatabaseID` sera 1 (valeur toujours attribuée à la base master).
- **DatabaseName** : nom de la base de données en toutes lettres. Plus intuitif que `DatabaseID`.

- **GroupID** : en SQL Server 2008, identifiant du groupe de ressources du gouverneur de ressources. Voir la section 4.5.
- **HostName** : nom de la machine cliente qui a ouvert la session. Correspondant au résultat de la fonction niladique `HOST_NAME()`.
- **IndexId** : identifiant de l'index. Correspond à la colonne `index_id` de la vue système `sys.indexes`. Utile pour tracer les *scans*.
- **IsSystem** : indique si l'événement est déclenché par une session système. Valeur 0 s'il s'agit d'une session utilisateur. Très utile pour filtrer les événements.
- **LineNumber** : contient le numéro de ligne de l'ordre, dans le batch ou dans la procédure stockée. Sur les erreurs, indique la ligne sur laquelle elle s'est produite.
- **LoginName** : contient le nom de la connexion (*login*) qui a ouvert la session. Nom complet avec domaine en cas de connexion Windows intégrée (le nom simple sera alors récupérable dans la colonne `NTUserName`).
- **ObjectID** : identifiant de l'objet. Correspond à la colonne `object_id` des vues systèmes `sys.objects` ou `sys.tables`. Utile pour tracer les *scans*.
- **SPID** : identifiant de session attribué par SQL Server. Correspond à la valeur récupérée par la variable système `@@SPID`. Utile pour filtrer une trace sur une seule session. Attention cependant : dès la session fermée, ce même numéro peut être réattribué par SQL Server à une nouvelle session.

Détection d'exceptions

Les événements de la catégorie « Errors and Warnings » tracent les erreurs et exceptions générées par SQL Server. C'est un bon moyen de surveiller votre serveur. Parfois, certaines applications clientes provoquent des erreurs qui ne sont pas récupérées dans leur code, ce qui fait que ces erreurs restent invisibles. Parfois également, des erreurs se produisent, et les utilisateurs pressent la touche ENTRÉE sur des messages difficiles à comprendre, sans appeler le support informatique. Ou, lorsque vous définissez vos propres erreurs, vous aimeriez savoir quand elles se déclenchent. La meilleure manière de récupérer une erreur occasionnelle lorsque vous vous y attendez, est de définir une alerte dans l'agent SQL. Vous pouvez vous faire notifier par l'agent du déclenchement de toute erreur, par sa gravité, son numéro ou un texte qu'elle contient. La programmation d'une alerte est hors du sujet de ce livre, nous vous montrons simplement, sur la figure 5.7, un exemple d'alerte déclenchée sur une erreur de gravité 23, déclenchée dans le contexte de la base `AdventureWorks`. La réponse à cette alerte peut être l'envoi d'une notification par e-mail ou NET SEND, ou l'activation d'un travail de l'agent.

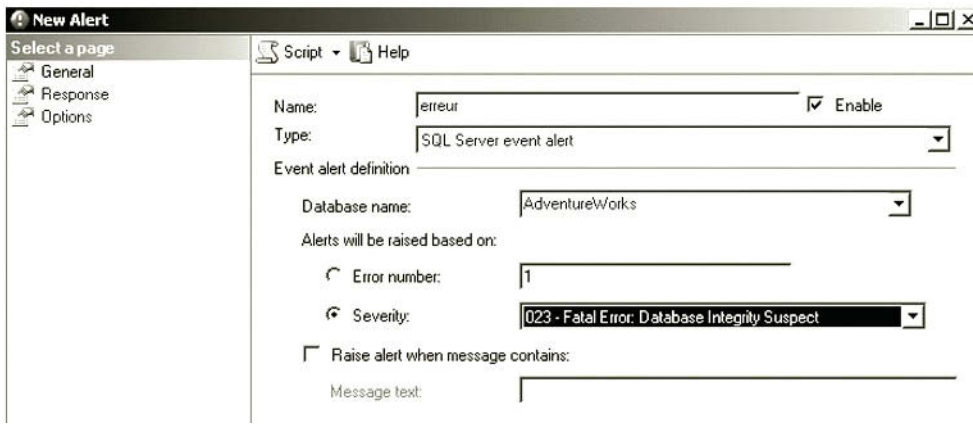

Figure 5.7 — Alerte sur événement de trace

Vous pouvez aussi, de temps en temps, tracer votre serveur à l'aide du profiler, pour vous assurer qu'aucune erreur ne passe indétectée. Les erreurs générées par SQL Server et renvoyées à la session cliente, sont appelées exceptions. Cela peut être l'information qu'un objet déclaré dans le code n'existe pas, la violation d'une contrainte, etc. Vous récupérez le numéro d'erreur dans l'événement `Exceptions`, et le message d'erreur tel qu'il est envoyé au client, dans l'événement `User Error Message`. Vous voyez sur la figure 5.8 un exemple de trace d'erreur. La signification des numéros d'erreur est indiquée dans les BOL.

```
Exception               Error: 2601, Severity: 14, State: 1
Exception               Error: 2714, Severity: 16, State: 3
User Error Message      There is already an object named 'GetContactsParameter' in the database.
SQL:BatchCompleted        CREATE PROCEDURE dbo.GetContactsParameter   @LastName nvarchar(50) = NULL  ...
SP:CacheMiss              CREATE PROCEDURE dbo.GetContactsLocalVariable   @LastName nvarchar(50) = NU...
Exception               Error: 2601, Severity: 14, State: 1
Exception               Error: 2714, Severity: 16, State: 3
User Error Message      There is already an object named 'GetContactsLocalVariable' in the database.
SQL:BatchCompleted        CREATE PROCEDURE dbo.GetContactsLocalVariable   @LastName nvarchar(50) = NU...
```

Figure 5.8 — Récupération des erreurs par le profiler

5.2.1 Utiliser le résultat de la trace

Une **trace** sauvegardée dans un fichier ou dans une table peut être ouverte à l'aide du profiler (menu File / Open / Trace File… ou Trace Table…). Les lignes peuvent être filtrées et groupées comme sur une trace en temps réel. Pour réaliser une analyse approfondie des résultats de la trace, la meilleure solution est d'enregistrer ce résultat dans une table SQL. Pour ce faire, rechargez le fichier de trace dans le profiler, puis sauvez le résultat dans une table à l'aide de la commande File / Save As / Trace Table… La trace peut aussi être insérée dans une table directement depuis SQL

Server, à l'aide de la fonction `fn_trace_gettable()`. Cette fonction accepte deux paramètres, le nom du fichier de trace, et le nombre de fichiers à importer en cas de présence de fichiers de *rollover* :

```
SELECT * INTO dbo.matrace
FROM sys.fn_trace_gettable('c:\temp\matrace.trc', default);
```

Lit les fichiers de trace dont le nom de base est `matrace.trc` et les insère dans la table `dbo.matrace`. La valeur `'default'` indique que tous les fichiers de *rollover* présents doivent être insérés.

Ensuite, rien n'est plus facile que de lancer des requêtes sur la table pour obtenir les informations désirées. Voici par exemple une requête listant par ordre décroissant les requêtes les plus coûteuses :

```
SELECT
    CAST(TextData as varchar(8000)) as TextData,
    COUNT(*) as Executions,
    AVG(reads) as MoyenneReads,
    AVG(CPU) as MoyenneCPU,
    AVG(Duration) / 1000 as MoyenneDurationMillisecondes
FROM dbo.matrace
WHERE EventClass IN (
    SELECT trace_event_id
    FROM sys.trace_events
    WHERE name LIKE 'S%Completed')
GROUP BY CAST(TextData as varchar(8000))
ORDER BY MoyenneReads DESC, MoyenneCPU DESC;
```

Pour ne prendre que les requêtes, nous cherchons dans la vue `sys.trace_events` les événements qui commencent par `'S'` (SQL ou SP) et qui se terminent par `'Completed'`.

Rejouer une trace

Une trace peut être **rejouée**, c'est-à-dire que les événements qu'elle contient peuvent être réexécutés sur un serveur SQL, si toutefois les événements contenus dans la trace sont rejouables. Pour créer une trace rejouable, le plus simple est d'utiliser le modèle `TSQL_Replay`, qui ajoute tous les événements nécessaires à une trace entièrement rejouable. Lorsque vous ouvrez une trace enregistrée, un menu « Replay » est ajouté, ainsi qu'une barre d'outils (voir figure 5.9).

Figure 5.9 — Boutons de *replay*

Vous pouvez poser un point d'arrêt sur une ligne d'événement (touche F9), faire du pas à pas (F10), ou exécuter la trace jusqu'à la ligne sélectionnée (CTRL+F10). Lorsque vous lancez l'exécution, une fenêtre permet de choisir les options, disposées dans deux onglets.

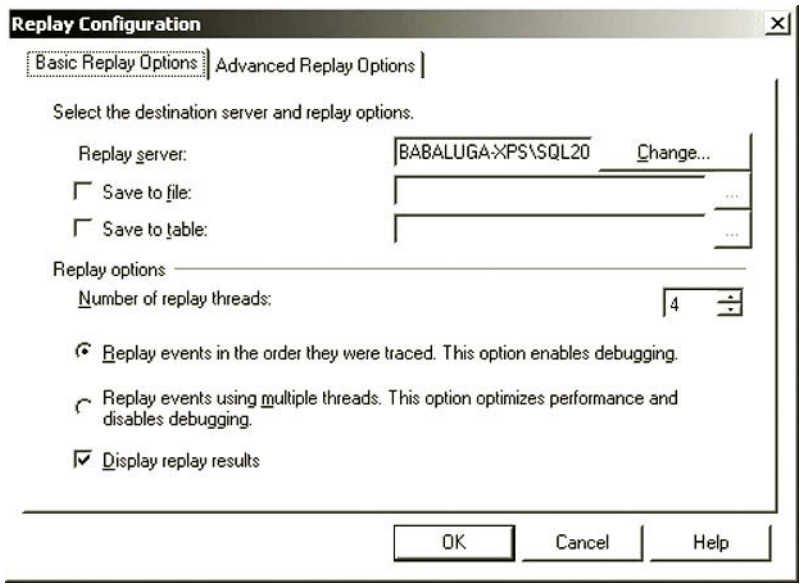

Figure 5.10 — Premier onglet des options de replay

Dans le premier, reproduit sur la figure 5.10, vous indiquez si le résultat de l'exécution doit être enregistré dans un fichier ou une table. Si ces deux options sont décochées, il sera simplement affiché dans le profiler. En utilisant les fonctionnalités de *multi-threading*, vous pouvez faire exécuter des instructions en parallèle. Le profiler rejoue les événements sur plusieurs *threads*, groupés par SPID, dans l'ordre de la colonne `EventSequence`. Vous reproduisez donc un environnement multi-utilisateurs. La trace est jouée plus vite, mais vous perdez les possibilités de débogage (point d'arrêt, pas à pas…).

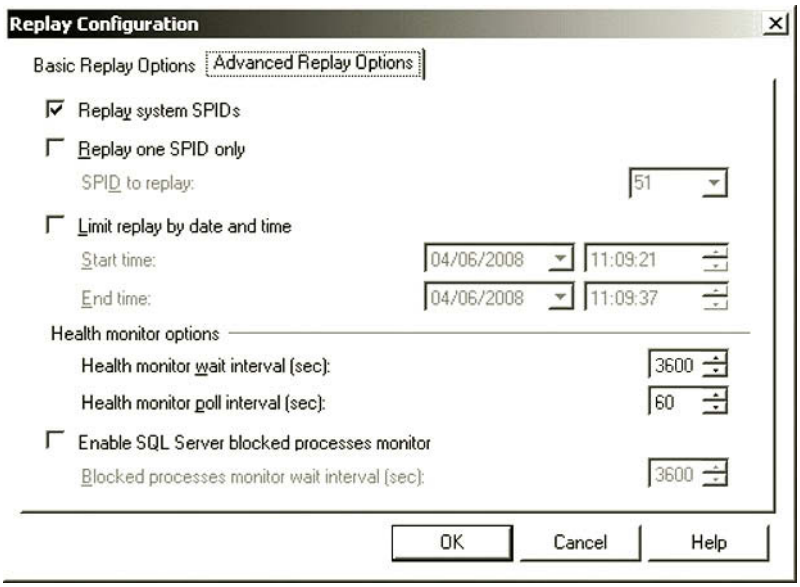

Figure 5.11 — Deuxième onglet des options de replay

Dans le premier, reproduit sur la figure 5.11, vous désactivez l'exécution d'instructions lancée par des sessions système pour ne garder que le comportement des sessions utilisateur. Vous pouvez n'exécuter que les instructions d'un SPID, en sachant que le même SPID peut être réutilisé par des sessions différentes, l'une après l'autre. Vous pouvez indiquer les intervalles de temps des événements de la trace qui doivent être rejoués. La partie Health Monitor permet de configurer un *thread* du Profiler pour tuer automatiquement les processus qui sont bloqués depuis un nombre défini de secondes. La valeur d'attente 0 signifie que le profiler ne tuera aucun processus bloqué. Nous traitons de la surveillance de processus bloqués dans la section 7.5.

Test de charge

La capacité de rejouer des traces est intéressante pour faire un **test de charge**, sur un serveur de test ou sur un serveur final avant une mise en production, à l'aide d'une trace représentative de l'activité réelle ou prévue du serveur. Vous pouvez ouvrir plusieurs fois la même trace, et la rejouer à partir d'instances différentes du profiler, ou même de machines clientes différentes. Cette approche pose un problème : il faut s'assurer que la trace ne comporte pas d'instructions qui ne peuvent être rejouées plus d'une fois, notamment des insertions explicites de valeurs vérifiées par des contraintes d'unicité. Si c'est le cas, vous pouvez soit les retirer de la trace, soit les encapsuler dans des transactions que vous terminez en `ROLLBACK`. Dans les deux cas, la gestion du *replay* devient plus, voire trop, contraignante.

Deuxième point : comment automatiser les *replay* ? Le profiler (`profiler90.exe` en SQL Server 2005) comporte quelques paramètres d'appel en ligne de commande (faire `profiler90.exe /?` pour les voir), mais qui ne permettent pas de lancer une trace directement en *replay*. Une solution peut être d'exporter le contenu des ordres SQL, par la commande File / Export / Extract SQL Server Events, qui génère un fichier de batches avec extension `.sql`. Vous pouvez ensuite exécuter l'outil en ligne de commande `sqlcmd.exe` avec le paramètre `-i` :

```
sqlcmd.exe -E -S monserveur -i c:\temp\monbatch.sql
```

que vous pouvez lancer de multiples fois dans un script, dans une boucle par exemple.

Une autre solution consiste à utiliser les outils que l'équipe de support utilisateur de Microsoft a développés pour faciliter l'utilisation de traces pour les tests de charge. Ils sont packagés sous le nom de **RML Utilities for SQL Server**, et peuvent être téléchargés sur le site de Microsoft (cherchez avec le nom du package). Ils nécessitent le Framework .NET 3.5, qu'ils installent automatiquement si nécessaire. Ces outils ne sont pas compatibles SQL Server 2008, notamment à cause des nouveaux types de données. Au moment de la rédaction de ce livre, la version 2008 est en développement, elle sera donc disponible tôt ou tard.

5.2.2 Diminution de l'impact de la trace

Comme nous l'avons dit, le profiler n'est qu'un client de la fonctionnalité serveur nommée SQL Trace. Le profiler est très utile en tant qu'outil d'optimisation, de débogage, et pour essayer, en temps réel, de découvrir la source d'un problème de performances. Son défaut est qu'il utilise des ressources supplémentaires, soit du temps processeurs lorsqu'il est lancé sur le serveur, soit de la bande passante réseau s'il tourne sur un poste client (ce qui est meilleur). Lorsque vous souhaitez planifier des traces, ou exécuter des traces de longue haleine, par exemple pour construire une *baseline*, ou pour identifier à travers une période représentative les requêtes les plus coûteuses, la solution la plus souple et la moins consommatrice de ressources, est de maintenir la trace à même SQL Server. Pour cela, vous disposez de quelques procédures stockées système. Vous pouvez exporter la définition de la trace définie dans le profiler en ordres SQL, ce qui rend très facile la création de scripts de trace. Dans le profiler, choisissez dans le menu « File » la commande Export / Script Trace Definition, puis votre version de SQL Server. Voici un exemple simplifié de script généré :

```
declare @rc int
declare @TraceID int
declare @maxfilesize bigint
set @maxfilesize = 5

exec @rc = sp_trace_create @TraceID output, 0,
                           N'InsertFileNameHere', @maxfilesize, NULL
if (@rc != 0) goto error

declare @on bit
```

```
set @on = 1
exec sp_trace_setevent @TraceID, 14, 1, @on
-- ...

-- Set the Filters
declare @intfilter int
declare @bigintfilter bigint

exec sp_trace_setfilter @TraceID, 10, 0, 7,
    N'SQL Server Profiler - 05b0b8f9-5048-4129-a0c1-9b7782ba8e6c'
-- Set the trace status to start
exec sp_trace_setstatus @TraceID, 1

-- display trace id for future references
select TraceID=@TraceID
```

- `sp_trace_create` : crée une trace en retournant en paramètre OUTPUT son identifiant numérique. Vous indiquez en paramètre le chemin du fichier dans lequel la trace sera enregistrée. Vous pouvez également indiquer la taille du fichier et le nombre maximum de fichiers de *rollover*, ainsi que la date et heure d'arrêt de la trace.
- `sp_trace_setevent` : ajoute un événement à partir de son identifiant. Celui-ci peut être retrouvé dans la vue `sys.trace_events`. Exemple de requête pour lister les événements :

```
SELECT tc.name as categorie, te.name as evenement, trace_event_id
FROM sys.trace_categories tc
JOIN sys.trace_events te ON tc.category_id = te.category_id
ORDER BY categorie, evenement;
```

- `sp_trace_setfilter` : ajoute un filtre. Les paramètres indiquent l'ID d'événement, un numérique indiquant un booléen pour organiser les filtres sur un même événement (`AND` ou `OR` entre les filtres), un numérique indiquant l'opérateur de comparaison, et la chaîne UNICODE à comparer.
- `sp_trace_setstatus` : démarre (0), arrête (1) ou supprime (2) la trace. C'est ce qui vous donne le contrôle sur l'exécution de votre trace.

La vue système `sys.traces` vous permet de retrouver la liste des traces définies. Lorsque vous avez une trace définie, vous pouvez planifier son exécution à l'aide de `sp_trace_setstatus`, par exemple dans des travaux de l'agent SQL.

Traces système

SQL Server possède deux traces spéciales, qu'il gère lui-même. La première est appelée la **trace par défaut** (*default trace*). Elle porte l'identifiant 1 et est activée par défaut. Elle n'est pas listée par `sys.traces`, mais vous pouvez utiliser la fonction `fn_trace_getinfo` pour la voir :

```
SELECT * FROM fn_trace_getinfo(1);
```

Cette trace n'a pas de nom, elle est utilisée en interne pour alimenter des rapports, et peut être utilisée en cas de *crash*, pour aider à en identifier la cause. Elle est écrite par défaut dans le répertoire de données, sous `\LOG\log.trc` (avec un nommage en *rollover*). Elle est peu coûteuse et peut être conservée. Si vous voulez la désactiver pour profiter des moindres moyens d'optimiser un serveur très chargé, vous pouvez changer l'option de serveur `'default trace enabled'` à 0 :

```
EXEC sp_configure 'show advanced options' , 1
RECONFIGURE
EXEC sp_configure 'default trace enabled', 0
EXEC sp_configure 'show advanced options' , 0
RECONFIGURE
```

La seconde est une trace que vous pouvez activer, appelée la boîte noire (*blackbox trace*). Comme son nom l'indique, elle vous permet de conserver un historique récent des opérations réalisées sur le serveur, afin d'offrir des informations en cas de défaillance. Elle est un peu plus lourde que la trace par défaut, car elle recueille plus d'événements. Si vous ne voulez pas avoir une boîte noire permanente, vous pouvez l'activer en cas de suspicion de problème, ou pour déboguer des crashs réguliers. Pour cela, créez une trace avec `sp_trace_create`, et le paramètre `@Options = 8` :

```
DECLARE @traceId int
EXEC sys.sp_trace_create @traceId OUTPUT, @Options = 8;
EXEC sys.sp_trace_setstatus @traceId, 1;
SELECT @traceId;
```

Vous pouvez modifier le chemin d'écriture du fichier, et la taille de celui-ci (5 Mo n'est pas toujours suffisant), avec les paramètres de `sp_trace_create`. Pour vous assurer que la boîte noire tourne toujours, vous pouvez automatiser son démarrage en encapsulant son activation par `sp_trace_setstatus` dans une procédure stockée, et en faisant en sorte que cette procédure se lance au démarrage du service, à l'aide de la procédure `sp_procoption` :

```
EXEC sys.sp_procoption 'ma_procedure', 'STARTUP', 'ON';
```

5.3 MONITEUR SYSTÈME

Le **moniteur système**, ou moniteur de performance, n'est pas propre à SQL Server. C'est un utilitaire de supervision intégré à Windows, qui est comme le moniteur cardiaque du système. Il vous permet de suivre en temps réel tous les compteurs – des données chiffrées – indiquant le comportement de votre serveur. Un certain nombre d'applications, dont SQL Server, ajoutent leur propre jeu de compteurs à ceux proposés par Windows. Vous pouvez ainsi, dans la même interface, suivre les signes vitaux de Windows et de SQL Server.

Vous trouvez le moniteur système soit dans le menu Windows, sous Outils d'administration/performances, soit en lançant l'exécutable `perfmon.exe`. À l'ouverture, quelques compteurs sont déjà présélectionnés, il vous appartient de les retirer si

vous n'en avez pas besoin, et de choisir les vôtres. Vous pouvez soit afficher l'évolution des compteurs en temps réel, soit enregistrer les mesures dans un fichier journal. Cette dernière solution est idéale pour construire une *baseline*. L'affichage peut être graphique (courbes d'évolution), ce qui est utile pour les compteurs dont il est intéressant de suivre la progression dans le temps, ou sous forme de tableau, pour obtenir la valeur actuelle précise. Vous pouvez également, pour chaque compteur, modifier l'échelle de sa ligne dans le graphe, afin de pouvoir mélanger des unités de mesures diverses sur le même affichage, ainsi que la couleur et l'épaisseur de son trait.

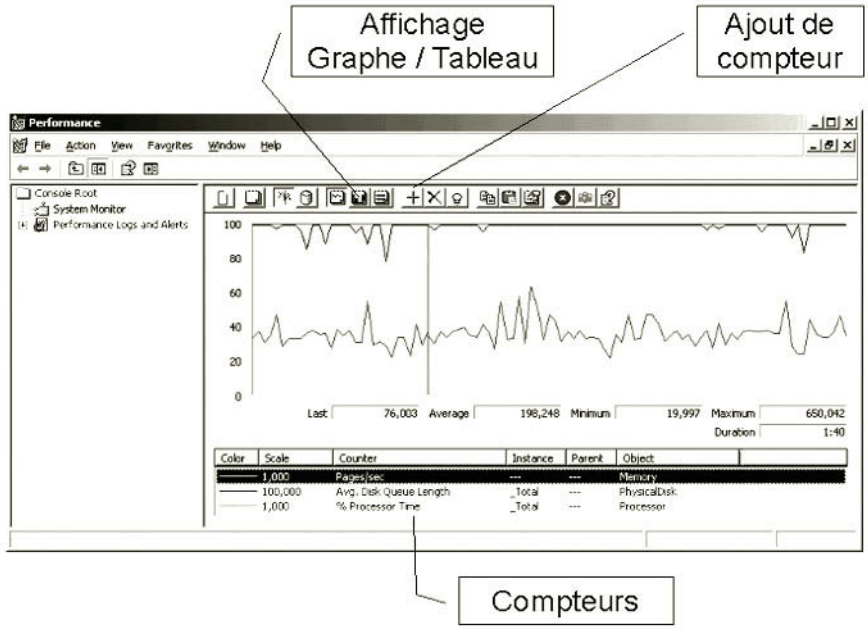

Figure 5.12 — Écran par défaut du moniteur de performances.

Attention aux différences d'échelle d'affichage dans le graphe. Il est facile de se méprendre en comparant deux mesures à des échelles différentes. Vous pouvez modifier la valeur d'échelle dans les propriétés du compteur, onglet Data, liste de choix « scale ».

Pour ajouter des compteurs, cliquez sur le bouton affichant le signe + dans la barre d'outils, ou utilisez la combinaison de touches CTRL+I. Vous obtenez ainsi la fenêtre de sélection de compteurs. Vous pouvez sélectionner de quelle machine vous voulez obtenir les compteurs, avec la syntaxe `\\NOM_MACHINE`. Lorsque vous sélectionnez un compteur, le bouton « explain... » permet d'obtenir une petite description.

Les compteurs sont regroupés par objets, et certains s'appliquent à des instances. L'objet est une collection de compteurs qui s'appliquent au même sous-système, par exemple les processeurs (CPU), les disques physiques, la mémoire vive, des modules de SQL Server… Les instances représentent les unités de ces sous-systèmes que le compteur mesure. Par exemple, l'objet processeur a autant d'instances que le système compte de processeurs logiques (deux instances pour un *Dual Core* ou un processeur *hyper-threadé*), ou l'objet disque logique a autant d'instances qu'il existe de partitions. Si vous voulez recueillir un compteur pour toutes les instances d'un objet, vous pouvez soit sélectionner le bouton radio « all instances… », soit choisir une instance qui regroupe les autres, la plupart du temps présente, et qui porte le nom `_Total`. Pour certains éléments, il peut être intéressant d'afficher à la fois un compteur par instance, et un compteur total. Vous obtenez une bonne perception du comportement du multitâche, par exemple, en visualisant une ligne par CPU individuel et une moyenne sur une ligne dont vous augmentez l'épaisseur. Cela vous permet en un coup d'œil de juger de l'équilibre de la charge de travail accordé par le système à chaque CPU.

Chaque objet dispose de compteurs spécifiques, qui sont les pourvoyeurs d'informations chiffrées. Ces informations sont exprimées soit en valeur courante (par exemple : `LogicalDisk/Current Disk Queue Length`), en moyenne (`LogicalDisk/Avg Disk Queue Length`), en pourcentage (`LogicalDisk/% Disk Read Time`) ou en totaux cumulés (`Memory/Committed Bytes`), soit en quantité par secondes (`LogicalDisk/Disk Reads/sec`).

Comme vous pouvez afficher dans la même fenêtre des compteurs provenant de machines différentes, le contexte complet du compteur est constitué de quatre parties : `\\Ordinateur\Objet(Instance)\Compteur`.

Il nous est arrivé, lorsque nous avions deux écrans à disposition, de conserver une fenêtre du moniteur système ouverte en permanence sur notre second écran, avec quelques compteurs essentiels montrant l'état d'un serveur SQL. Dans ce cas, nous choisissions un délai de rafraîchissement de 5 ou 10 secondes, afin de ne pas trop impacter le serveur. Il est à noter qu'une utilisation du moniteur sur une machine cliente est moins pénalisante qu'une utilisation locale au serveur. Dans ce cas, nous cachions tous les éléments d'affichage inutiles, pour ne conserver qu'une vue de rapport, comme reproduit sur la figure 5.13.

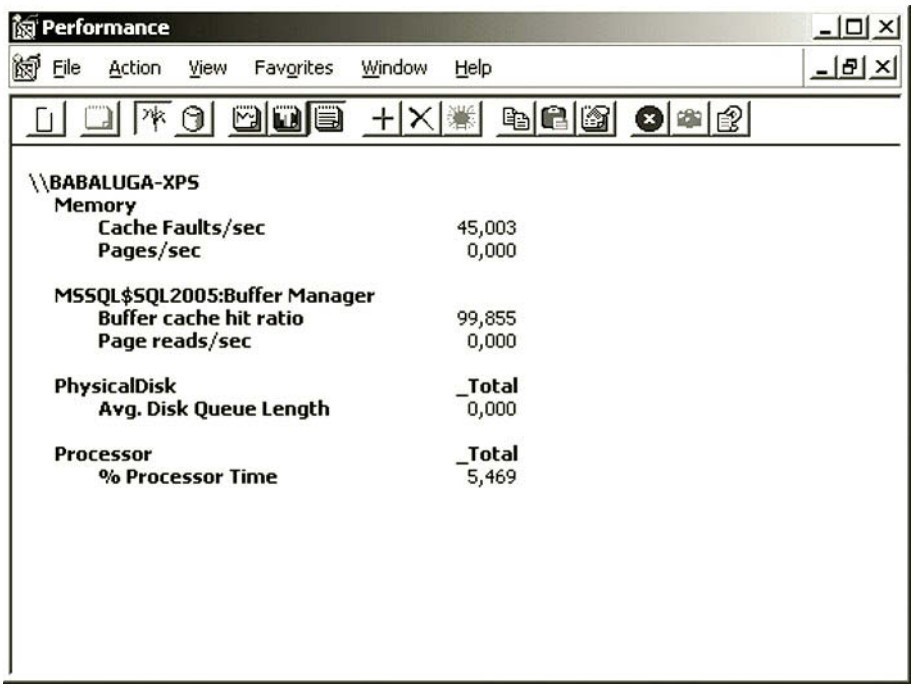

Figure 5.13 — Affichage minimal

Lorsque vous avez obtenu une vision satisfaisante, et que vous avez choisi les bons compteurs, vous pouvez sauver cette configuration dans un fichier de configuration de console (Le moniteur de performance est un composant logiciel enfichable (*snap-in*) de l'outil Windows MMC, *Microsoft Management Console*), portant l'extension `.msc`. Cela vous permet de rouvrir très rapidement des jeux de compteur. Vous pouvez vous faire une deuxième fenêtre montrant l'activité des processeurs en affichage graphe. Voici un script WSH (*Windows Script Host*) que vous pouvez utiliser pour lancer en une seule commande plusieurs fenêtres de moniteur à partir de fichiers `.msc` (deux dans l'exemple) :

```
Option Explicit
Dim oWsh : Set oWsh = CreateObject("WScript.Shell")
oWsh.Run "mmc ""C:\mypath\CPU.msc"""
oWsh.Run "mmc ""C:\mypath\SQL MEMORY AND DISK.msc"""
Set oWsh = Nothing
```

Nous allons passer en revue les indicateurs utiles pour suivre l'activité de SQL Server.

5.4 CHOIX DES COMPTEURS

Pour obtenir des informations sur les performances de SQL Server, vous devez panacher des compteurs fournis par le système d'exploitation, et des compteurs propres à SQL Server. Certains **compteurs du système d'exploitation** ne peuvent donner des informations précises, par exemple sur l'utilisation du cache de mémoire. Nous avons parlé de la couche SQLOS. SQL Server vous fournit donc des compteurs spécifiques qui permettent d'explorer plus en détail ce que fait SQLOS. Nous allons séparer les compteurs essentiels, qui donnent des informations vitales sur la santé du serveur, des compteurs simplement utiles.

5.4.1 Compteurs essentiels

MSSQL:Buffer Manager\Buffer cache hit ratio

Ce compteur indique le pourcentage de pages lues par le moteur de stockage, qui ont pu être servies depuis le cache de données (le *buffer*), sans accéder au disque, calculé sur les quelques derniers milliers de pages demandées. Ce ratio doit être élevé : moins SQL Server doit accéder au disque, plus il sera rapide. On considère dans une application OLTP à volumétrie moyenne, que cette valeur doit être au-dessus de 97 ou 98 %, ce qui est déjà peu. Un bon chiffre est au-dessus de 99,7 %. Si le ratio est bas, et que vous observez également une activité et une file d'attente importantes sur le disque, augmentez la mémoire vive de votre système (ou cherchez des *scans* de table qui peuvent être éliminés par la création d'index).

MSSQL:Databases\Transactions/sec

Nombre de transactions par seconde. Le choix de l'instance de compteur permet d'afficher toutes les transactions, ou les transactions dans un contexte de base particulier, ce qui est utile pour tracer l'activité comparative de vos bases de données, et l'activité dans `tempdb`. Une utilisation régulière de ce compteur vous permet de connaître l'activité moyenne de votre serveur, à travers le temps et durant les périodes de la journée, ainsi que de détecter les augmentations de charge. Une valeur élevée de transactions par seconde, avec des résultats raisonnables d'autres compteurs (utilisation du disque, pourcentage d'activité des processeurs) indique une machine qui travaille bien. Un bon signe de l'effectivité de vos efforts d'optimisation d'un serveur est l'augmentation du nombre de transactions par seconde, liée à la diminution de la charge des processeurs.

Une quantité importante de transactions dans `tempdb` indique généralement soit un recours fréquent aux tables temporaires dans le code, soit un fonctionnement en niveau d'isolation snapshot. Vous pouvez vérifier que cette activité n'est pas due à une création de tables de travail internes, destinées à résoudre des plans de requête, en observant le compteur `MSSQL:Acces Methods\Worktables Created/sec`, qui indique le nombre de tables de travail créées par seconde. Les tables de travail stockent des résultats intermédiaires lors de l'exécution de code SQL. Vous pouvez aussi vous

baser sur le compteur `MSSQL:General Statistics\Active Temp Tables` pour connaître le nombre de tables temporaires dans `tempdb`, ou `MSSQL:General Statistics\Temp Tables Creation Rate` pour une quantité de tables temporaires utilisateur créées par seconde. Référez-vous à la section dédiée à `tempdb` pour plus de détails.

MSSQL:General Statistics\Temp Tables Creation Rate

Indique le nombre de tables temporaires utilisateurs créées par seconde. Le compteur `MSSQL:General Statistics\Temp Tables For Destruction` retourne le nombre de tables temporaires marquées pour suppression pour le thread de nettoyage. Ces compteurs vous donnent une bonne idée du poids des tables temporaires dans votre code SQL, et des problèmes potentiels qu'elles peuvent produire (contention des tables de métadonnées dans `tempdb`, par exemple, voir section 8.3.1).

PhysicalDisk\Avg. Disk Queue Length

Longueur moyenne de la file d'attente sur un disque. Le disque dur étant à accès séquentiel (une seule tête de lecture, donc une seule lecture/écriture à la fois), les processus qui veulent accéder au disque lorsque celui-ci est déjà en utilisation, sont placés dans une file d'attente. Il s'agit donc de temps perdu à faire la queue, comme aux caisses d'un supermarché. Dans l'idéal, ce compteur devrait toujours être à 0, voire 1, avec des possibilités de pointes. Si la valeur est de façon consistante à 2 ou plus, vous avez un goulot d'étranglement sur le disque. Que faire ? Les possibilités sont, dans l'ordre logique d'action :

- diminuer les besoins de lecture sur le disque, en optimisant les requêtes, ou la structure physique de vos données, notamment en créant des index pour éviter les *scans* ;
- ajouter de la RAM, pour augmenter la taille du *buffer*, et diminuer les besoins de lecture sur le disque ;
- déplacer les fichiers de base de données ou de journal de transaction les plus utilisés, sur d'autres disques, pour partager la charge : faire du *scale-out* (répartition de charge) ;
- utiliser un disque plus rapide, avec un meilleur temps d'accès, un contrôleur plus rapide, ou utiliser un sous-système disque plus performant (SAN, *stripe*…). En d'autres termes, faire du *scale-up*.

Processor\% Processor Time

Pourcentage d'utilisation des processeurs logiques (deux instances sur un Dual Core). La valeur retournée est soit la moyenne de tous les processeurs si vous affichez toutes les instances, soit par instance. C'est le compteur de base pour avoir une idée de départ de la charge de votre serveur. Une machine dont les processeurs sont constamment proches du 100 % a un problème. Soit elle a atteint les limites de ses capacités, et il faut donc songer à acquérir du nouveau matériel, que ce soit en *scale-up* ou en *scale-out* (répartition de charge), soit un élément du système entraîne une surcharge. Vous amener à identifier cet élément est un des objectifs de ce livre. Il est

important de comprendre qu'une forte activité des processeurs ne signifie pas que la puissance ou la quantité des CPU doit être augmentée. Le travail du processeur est influencé par tous les autres sous-systèmes. Si le disque est inutilement sollicité, ou trop lent, si la file d'attente sur le disque implique de nombreuses attentes de threads, si la mémoire vive fait défaut et entraîne donc une plus forte utilisation du disque, etc. les CPU devront travailler plus. Donc, le temps processeur n'est que la partie émergée de l'iceberg, vous devez inspecter toutes les raisons possibles de cette surcharge de travail, avant de tirer vos conclusions.

5.4.2 Compteurs utiles

Memory\Pages/sec

Indique le nombre de pages de mémoire (dans le sens Windows du terme) qui sont écrites sur le disque, ou lues depuis le disque. Un nombre élevé (par exemple plus de 2 500 pages par secondes, de façon consistante sur un serveur de puissance moyenne, peut indiquer un manque de mémoire. À corréler avec les compteurs de cache de SQL Server, et l'activité disque.

MSSQL:Access Methods\Forwarded Records/sec

Indique un nombre le lignes lues *via* un renvoi d'enregistrement par secondes. Cela peut permettre de détecter un trop grand nombre de renvois d'enregistrements[1]. Sur une table sans index *clustered*, le moteur de stockage crée des pointeurs de renvoi lorsque la taille d'une ligne modifiée augmente (parce qu'elle contient des colonnes de taille variable), et qu'elle ne peut plus tenir dans sa page originelle. Ce pointeur reste en place jusqu'à un compactage (*shrink*) de la base de données ou jusqu'à ce que la ligne diminue suffisamment de taille pour réintégrer sa page d'origine. Un trop grand nombre de renvois diminue les performances d'IO, à cause de l'étape supplémentaire de lecture du pointeur, et de déplacement sur son adresse. Cela ne devient problématique sur une table que lorsqu'un bon pourcentage de la table contient des renvois d'enregistrements. Vous pouvez détecter ces tables à l'aide de la fonction de gestion dynamique `sys.dm_db_index_physical_stats` :

```
SELECT
    OBJECT_NAME(object_id) as ObjectName, index_type_desc,
    record_count, forwarded_record_count,
    (forwarded_record_count / record_count)*100 as
        forwarded_record_ratio
FROM sys.dm_db_index_physical_stats(
    DB_ID('AdventureWorks'),
    NULL, NULL, NULL, DEFAULT)
WHERE forwarded_record_count IS NOT NULL;
```

1. Bien que nous parlions de ligne lorsque nous évoquons la couche logique des bases de données, le terme enregistrement est utilisé lorsque nous parlons du moteur de stockage.

Attention – Le dernier paramètre (mode) doit être indiqué à une valeur différente de `LIMITED` ou `NULL` (qui équivaut à `LIMITED`), sinon toutes les valeurs de la colonne forwarded_record_count seront renvoyées à `NULL`.

Ne vous inquiétez pas d'un nombre raisonnable de Forwarded Records/sec, cela peut arriver dans des tables temporaires sur `tempdb`. Vérifiez alors le nombre de tables temporaires, et le nombre de transactions dans `tempdb`. Si la valeur ne correspond pas à une activité de `tempdb`, cherchez dans votre base utilisateurs quelles sont les tables *heap* qui contiennent des renvois d'enregistrements, à l'aide de la fonction précédente. Pour diminuer le nombre de renvois, vous pouvez soit faire un shrink (`DBCC SHRINKDATABASE` ou `DBCC SHRINKFILE`) de votre base de données, ce qui est une mauvaise solution en soi, sauf si vous faites un shrink sans diminuer la taille physique du fichier, à l'aide de l'option `NOTRUNCATE` :

```
DBCC SHRINKDATABASE (AdventureWorks, NOTRUNCATE);
```

Cela réorganise les pages sans diminuer la taille des fichiers. La vraie bonne solution reste toutefois de réorganiser la table en créant un index *clustered*, que vous pouvez supprimer ensuite si vous tenez absolument à conserver une table *heap* (il n'y aurait pas de raison, la table *clustered* est à recommander)[1].

Attention – Les augmentations et diminutions automatiques des fichiers de données et de journal sont très pénalisantes pour les performances, et génèrent une fragmentation importante. Elles sont à éviter.

MSSQL:Access Methods\Full Scans/sec

Indique un nombre de *scans* de table (table *heap* ou index *clustered*) par seconde. Le *scan* de table n'est pas un problème en soi sur les petites tables, il est par contre très pénalisant sur les tables à partir d'une taille raisonnable (s'entend en nombre de pages. Une table contenant beaucoup de lignes contenant des colonnes de petite taille peut provoquer moins de lectures qu'une table ayant moins de lignes, mais contenant des colonnes plus grandes). Ce compteur est utile pour détecter un manque d'index ou des requêtes mal écrites. Un *scan* provient soit d'un manque index utile pour résoudre la clause de recherche (qu'il soit absent ou trop peu sélectif), soit de la présence de requêtes qui ne filtrent pas (ou mal) les données, par exemple de requêtes sans clause `WHERE`. Pour plus de détails, reportez-vous au chapitre 6.

1. Plus d'informations sur les problèmes de performances possibles sur des renvois d'enregistrements dans cette entrée de blog : http://blogs.msdn.com/mssqlisv/archive/2006/12/01/knowing-about-forwarded-records-can-help-diagnose-hard-to-find-performance-issues.aspx

MSSQL:Access Methods\Page Splits/sec

Indique un nombre de séparations (*split*) de page par seconde. Un nombre élevé de *page splits* indique que des index font face à d'intensives réorganisations dues à des modifications de données sur leurs clés. Cela pourra vous pousser à utiliser un `FILLFACTOR` (voir section 6.1) plus élevé sur ces index, ou à modifier votre stratégie d'indexation. Pour chercher sur quels index les *splits* se sont produits, la fonction de gestion dynamique `sys.dm_db_index_operational_stats` peut vous donner des pistes. Pour plus de détails, reportez-vous au chapitre 6.

MSSQL:Access Methods\Table Lock Escalations/sec

Retourne un nombre d'escalade de verrous par seconde. Pour plus d'information sur les escalades de verrous, reportez-vous au chapitre 7.

Pour identifier quelles requêtes peuvent provoquer une escalade de verrous, vous pouvez tenter de corréler les pics de ce compteur avec une trace SQL (voir la corrélation de journal de compteur et de trace en section 5.3.4). La colonne `index_lock_promotion_count` de la fonction `sys.dm_db_index_operational_stats` est également utile.

MSSQL:Buffer Manager\Database pages

Indique le nombre de pages de données cachées dans le *buffer*. Sur un système qui dispose de suffisamment de RAM, cette valeur devrait peu évoluer. Si vous observez des changements importants de ce compteur, cela veut dire que SQL Server doit libérer des pages du *buffer*. Il manque certainement de la mémoire vive (il en manque physiquement, ou l'option « `max server memory` » limite son utilisation par SQL Server).

MSSQL:Buffer Manager\Free list stalls/sec

Indique la fréquence à laquelle des demandes d'octroi de pages de *buffer* sont suspendues parce qu'il n'y a plus de pages libres dans le cache. Cette valeur doit être la plus petite possible. Une valeur dépassant 2 est un indicateur de manque de mémoire.

MSSQL:Buffer Manager\Page life expectancy

Indique le nombre de secondes pendant lesquelles une page de données va rester dans le *buffer* sans références, c'est-à-dire sans qu'un processus n'accède à cette page. Selon Microsoft, 300 secondes est la valeur minimum à obtenir, et l'idéal est la valeur la plus élevée possible. Si ce compteur indique 300, cela signifie qu'une page va être vidée du cache après 5 minutes à moins qu'elle soit utilisée dans ce laps de temps. SQL Server ajuste cette valeur selon la quantité de mémoire disponible pour le *buffer*. Moins il y a de mémoire, plus la durée de vie est courte, pour permettre à SQL Server de libérer de la place dans le *buffer* pour des données plus utiles. C'est donc un bon indicateur : si ce compteur tombe en dessous de 300, vous manquez manifestement de mémoire pour le *buffer* ou vous avez malencontreusement limité l'option « max server memory ».

MSSQL:Buffer Manager\Page reads/sec et MSSQL:Buffer Manager\Page writes/sec

Indiquent le nombre de pages lues depuis le disque et écrites sur le disque par le gestionnaire de cache de données. En corrélant ces valeurs avec vos informations provenant du disque, vous pouvez déterminer si l'activité disque est principalement générée par SQL Server. De même, plus ces valeurs sont élevées, moins votre cache travaille efficacement. En d'autres termes, plus vous manquez de mémoire vive.

MSSQL:Buffer Manager\Total pages

Nombre total de pages de cache de données. Vous trouvez l'utilisation des pages dans d'autres compteurs du même groupe : dans **database pages**, les pages utilisées pour les données, dans **free pages**, les pages disponibles, dans **stolen pages** les pages libérées sous pression mémoire, pour d'autres utilisations, en général le cache de procédures. **Target pages** indique le nombre idéal de pages de *buffer* que SQL Server estime pour votre système. La différence entre Target pages et Total pages devrait évidemment être aussi mince que possible.

MSSQL:Databases\Log growths

Nombre total d'augmentations de la taille physique des fichiers de journal de transaction pour une base de données. Cela vous indique si la taille du journal choisie à la base était bonne. L'augmentation automatique du journal diminue les performances d'écriture en fragmentant et en multipliant le nombre de VLF (*Virtual Log Files*).

MSSQL:Databases\Percent Log Used

Pourcentage du journal en utilisation. Nous vous conseillons de créer une alerte de l'agent SQL testant cette valeur, pour vous notifier par exemple lorsqu'elle dépasse les 80 %, ou plus, selon votre système.

MSSQL:General Statistics\Active Temp Tables

Nombre de tables temporaires utilisateur dans `tempdb`, à un moment donné.

MSSQL:General Statistics\Processes Blocked

Ce compteur, nouveauté de SQL Server 2005, affiche le nombre de processus bloqués, c'est-à-dire de processus qui attendent, depuis un certain temps, la libération de verrous tenus par un autre processus, pour continuer leur travail. Lorsqu'un verrou est maintenu trop longtemps, il peut provoquer un « embouteillage » de verrous, les processus s'attendant les uns sur les autres. Vous pouvez utiliser ce compteur dans une alerte du moniteur système ou de l'agent SQL pour vous avertir de cette situation (une augmentation rapide et continue de ce compteur en étant le signe). Vous avez d'autres moyens de détection de cette situation, comme nous le verrons dans la section 7.5.1.

MSSQL:General Statistics\User Connections

Nombre de sessions ouvertes. Une augmentation et une stabilisation de ce nombre peuvent indiquer des problèmes liés à une application qui ne gère pas bien les déconnexions.

MSSQL:Locks\Average Wait Time (ms)

Nombre moyen d'attentes pour la libération de verrous, en millisecondes. Une valeur élevée indique une contention due à des transactions ou à des requêtes trop longues, qui verrouillent les données trop longtemps. Le meilleur remède est de modifier le code, pour diminuer l'étendue des transactions et la durée des requêtes, ou leur niveau d'isolation. Si cela n'est pas possible, il peut être utile de passer la base de données dans laquelle le verrouillage est important, en mode `READ COMMITTED SNAPSHOT` (voir section 7.3).

MSSQL:Locks\Number of Deadlocks/sec

Nombre de verrous mortels par secondes. Si cette valeur est régulièrement au-dessus de… 0, vous avez un réel problème. Reportez-vous à la section 7.5.2.

MSSQL:Plan cache\Cache Objects Count

Nombre d'objets de cache par type de cache. L'instance « SQL Plans » indique le cache de plans d'exécution. Attention à la diminution violente de cette valeur : signe de pression sur la mémoire qui risque d'impacter fortement les performances.

MSSQL:SQL Errors\Erros/sec

Signe d'erreurs, par exemple après modification de structure. Les erreurs sont à examiner à l'aide d'une trace SQL.

MSSQL:SQL Statistics\Batch requests/sec

Nombre de requêtes SQL envoyées au serveur. Indique la demande de travail à laquelle votre serveur est soumis.

MSSQL:SQL Statistics\SQL Compilations/sec

Nombre de compilations de plans par seconde. Une valeur élevée peut indiquer un manque de mémoire pour le cache de plans, ou un besoin de configurer l'auto-paramétrage plus agressivement (voir section 9.2).

MSSQL:SQL Statistics\SQL Re-Compilations/sec

Nombre de recompilations par seconde. Une valeur élevée indique un problème de performance des requêtes. Identifiez les procédures incriminées à l'aide du profiler, et appliquez les solutions proposées dans la section 9.1.2.

Network Interface\Current Bandwidth et Network Interface\Bytes Total/sec

Indique la capacité de votre connexion réseau, et son débit actuel, respectivement. La comparaison des deux donne l'utilisation actuelle (attention aux différences d'échelle dans l'affichage en graphe du moniteur de performance).

PhysicalDisk\Split IO/sec

Requêtes d'entrées/sorties que Windows doit diviser pour honorer. Sur un disque simple, un nombre élevé est un indicateur de fragmentation. Attention, n'en tenez pas compte sur un système RAID, il est normal d'y trouver des *splits*.

Au sujet des compteurs de disque

L'objet `PhysicalDisk` fournit des compteurs par disque physique, et l'objet `LogicalDisk` par partition (lettre de lecteur).

Les objets de compteurs de disque ont un effet sur les performances du serveur, même lorsqu'on ne les affiche pas avec le Moniteur Système. Pour cette raison, sur Windows NT ces compteurs sont totalement désactivés, et sur Windows 2000, seuls les compteurs physiques (`PhysicalDisk`) sont activés (donc l'objet `LogicalDisk` était désactivé). Un outil en ligne de commande, nommé `diskperf`, est à disposition pour activer ou désactiver ces compteurs. À partir de Windows Server 2003, ces compteurs sont activés par défaut, leur impact n'étant pas réellement significatif sur les machines modernes. `diskperf` est toujours présent si vous voulez désactiver une partie des compteurs. Sachez toutefois qu'une activation ou désactivation nécessite un redémarrage de la machine, ce qui rend son utilisation peu intéressante…

Process\% Processor Time

Ce compteur donne le temps CPU total utilisé par un processus. C'est un cumul des compteurs `% Privileged Time` et `% User Time` du même objet.

% Privileged Time

C'est le pourcentage de temps CPU utilisé par le processus en mode privilégié. Le temps privilégié (*privileged time*, ou *kernel time*), est le temps d'activité des CPU dédiés aux appels système (gestion de la mémoire, entrées/sorties, etc.). C'est souvent le cas d'un service Windows qui a besoin d'accéder aux données privées du système, qui sont protégées contre des accès de processus s'exécutant en mode utilisateur. Les appels implicites au noyau, comme les fautes de page, sont aussi comptés comme du temps privilégié. Sur une machine dédiée à SQL Server, le pourcentage de temps privilégié devrait être faible, SQL Server exécutant la majeure partie de son travail en temps utilisateur. Vous pouvez obtenir une vision rapide du ratio temps privilégié / temps utilisateur à l'aide du gestionnaire de tâches (*Windows task manager*). L'onglet « performance » montre notamment des graphes de CPU. Vous pouvez afficher la partie de temps privilégié (appelée ici *kernel time*), en activant dans le menu View l'option `show kernel times` (figure 5.14).

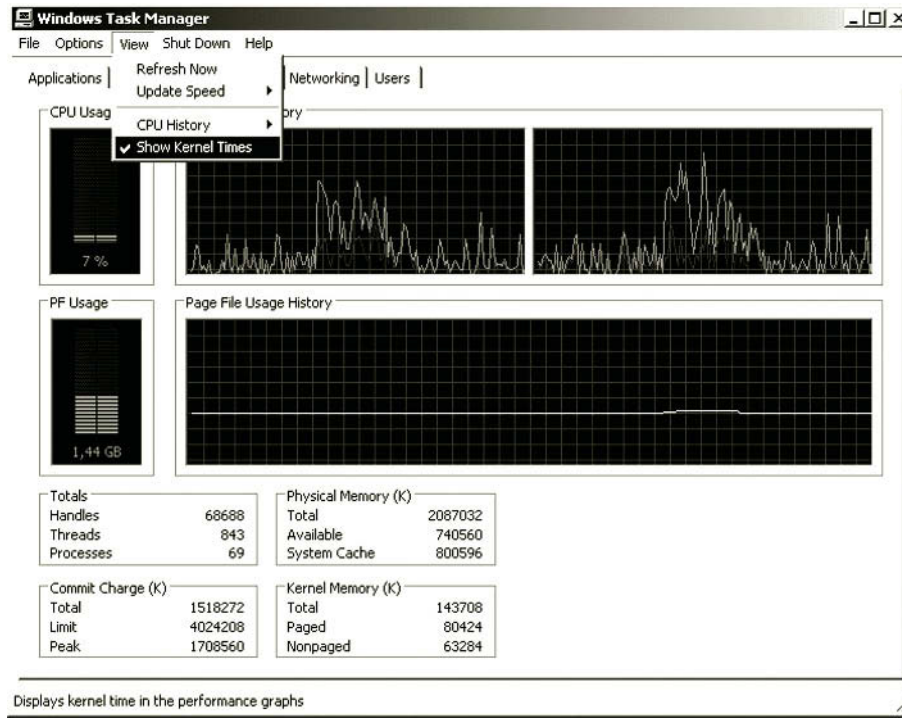

Figure 5.14 — Affichage du temps privilégié

System\Context Switches/sec

Indique le nombre de changements de contexte à la seconde. Dans un environnement multiprocesseurs, un nombre de *threads* plus important que le nombre de processeurs s'exécute en multitâche préemptif[1]. Le système d'exploitation est responsable de l'attribution des temps d'utilisation processeur à chaque *thread*. Il organise donc une répartition d'exécution, comme un animateur de débat télévisé qui distribue les temps de parole tout au long de l'émission. Chaque *thread* s'exécute dans un contexte, son état est maintenu, notamment dans les registres du processeur. Passer d'un *thread* à l'autre implique donc de remplacer le contexte du *thread* remis dans la pile, par le contexte du *thread* à qui l'OS redonne le processeur.

Nous avons vu que SQL Server implémente une couche appelée SQLOS, qui est un environnement semblable à un système d'exploitation en miniature. Cet environnement gère ses propres *threads*, appelés « *worker threads* », *mappés* par défaut à des *threads* Windows. Un réservoir (*pool*) de *worker threads* est maintenu par SQLOS pour exécuter les tâches du serveur SQL. Le nombre de *threads* dans ce *pool* est déter-

1. En multitâche préemptif, un temps défini est attribué par le système d'exploitation à chaque processus pour exécuter sa tâche. Si le travail n'est pas fini à l'expiration de ce délai, le processus est renvoyé dans la pile et laisse sa place au processus suivant dans la file d'attente, et ainsi de suite.

miné par l'option de configuration du serveur « max worker threads ». La valeur par défaut de cette option est 0, qui correspond à une auto-configuration de SQL Server. Cette valeur est optimale dans la grande majorité des cas. Si vous avez effectué une mise à jour directe d'une instance SQL Server 2000 vers 2005, vérifiez que cette valeur est bien à 0. En effet, en SQL Server 2000, la valeur par défaut était 255, et elle demeure lors de la mise à jour. L'auto-configuration en SQL Server 2005 suit une logique relative au nombre de processeurs, expliquée dans les BOL sous l'entrée « *max worker threads Option* ». Pour connaître la valeur actuelle sur votre système :

```
SELECT max_workers_count
FROM sys.dm_os_sys_info
```

Il est hasardeux d'augmenter le nombre de *worker threads*, et en général inutile, voire contre performant[1]. Cette option n'est pas une option intéressante pour augmenter les performances.

Donc, les changements de contexte sont coûteux, mais nécessaires. Des valeurs comme 10 000 ou 15 000 sont normales. Une augmentation exagérée de *context switches* devient un problème, car les processeurs passent trop de leur temps à changer de contexte. Dans ce cas, une augmentation du nombre de processeurs est à envisager. Dans un système en architecture SMP qui comporte beaucoup de processeurs, où le nombre de changements de contexte est élevé de façon consistante, et où les processeurs sont en utilisation presque maximum en permanence, vous pouvez essayer de configurer SQL Server pour l'utilisation de fibres. Les fibres sont des *threads* plus légers, qui peuvent changer de contexte sans quitter le mode utilisateur. Après activation de ce mode, SQLOS mappe ses *worker threads* à des fibres et non plus à des *threads* Windows. Passer du mode *thread* au mode fibre se fait en changeant l'option de serveur « `lightweight pooling` », comme ceci :

```
EXEC sp_configure 'show advanced options', 1
RECONFIGURE
EXEC sp_configure 'lightweight pooling', 1
RECONFIGURE
EXEC sp_configure 'show advanced options', 0
RECONFIGURE
```

Ce changement n'est pas à faire à la légère, car il peut aussi avoir un effet négatif. En tout cas, le gain a peu de chances d'être spectaculaire. Notez également que SQL Mail n'est pas disponible en mode fibre (ce qui a peu d'impact, car vous l'avez certainement remplacé par l'outil Database Mail), ni le support des objets de code .NET.

Un compteur **Thread\Context Switches/sec** vous permet aussi d'entrer dans les détails : quels *threads* sont beaucoup déplacés.

1. Comme nous l'explique Ken Henderson dans cet article de blog : http://blogs.msdn.com/khen1234/archive/2005/11/07/489778.aspx

System\Processor Queue Length

Indique le nombre de *threads* qui sont dans la file d'attente du processeur. Ce sont donc des *threads* qui attendent que le système d'exploitation leur donne accès à un processeur. Il y a une seule file d'attente, quel que soit le nombre de processeurs. En divisant ce nombre par la quantité de processeurs, vous avez une idée du nombre de *threads* en attente sur chacun. Au-dessus de 10 *threads* en attente par processeur, commencez à vous inquiéter.

5.4.3 Compteurs pour tempdb

MSSQL:Access Methods\Worktables Created/sec

nombre de tables de travail créées par seconde. Les tables de travail sont utilisées par le moteur d'exécution de SQL Server pour résoudre des plans de requête contenant du *spooling*. Ce nombre devrait rester inférieur à 200. Les moyens d'éviter les tables de travail sont d'améliorer la syntaxe des requêtes (et d'éviter les requêtes sans clause `WHERE`, qui doivent traiter trop de lignes), de diminuer l'utilisation des curseurs ou des objets larges (LOB), et de créer les index nécessaires aux tris préliminaires dans les requêtes.

MSSQL:Access Methods\Workfiles Created/sec

Nombre de fichiers de travail créés par seconde. Les fichiers de travail sont utilisés par le moteur d'exécution dans des plans de requête utilisant le hachage. Pour diminuer les fichiers de travail, évitez le traitement de jeux de résultats trop importants : le hachage est utilisé dans les jointures et les regroupements qui doivent traiter un nombre important de tuples.

MSSQL:General Statistics\Temp Tables Creation Rate

Le nombre de tables temporaires et variables de type tables créées par seconde. Utile pour différencier, dans une utilisation importante de `tempdb`, ce qui revient à la création de tables temporaires utilisateurs par rapport aux dépôts de versions et aux fichiers et tables de travail.

MSSQL:Transactions\Free Space in tempdb (KB)

Utile pour détecter un manque d'espace libre dans `tempdb`. Une bonne idée est de créer une alerte de l'agent SQL sur un dépassement de cette valeur.

MSSQL:Transactions\Version Store Size (KB)

Indique la taille des dépôts de versions pour les versions de ligne (*row versioning*). Cette valeur devrait évoluer dynamiquement au fur et à mesure de l'utilisation du row versioning, et donc diminuer quand les besoins de versions diminuent. Une taille de dépôt de versions qui continue d'augmenter indique un problème, probablement une transaction ouverte trop longtemps, empêchant la libération des versions de ligne.

MSSQL:Transactions\Version Generation Rate (KB/s) et MSSQL:Transactions\Version Cleanup Rate (KB/s)

Indiquent le nombre de Ko de dépôts de versions créés et supprimés par seconde. Des valeurs importantes indiquent une utilisation importante du *row versioning*, mais si les compteurs généraux du système (disque, mémoire, CPU) ne montrent pas de pression excessive, c'est le signe d'un fonctionnement de `tempdb` non bloquant, ce dont on peut se féliciter. Par contre, la corrélation avec une augmentation de la pression sur le système, peut nous donner des indications de sa cause.

5.4.4 Compteurs utilisateur

L'objet de compteurs **MSSQL:User Settable** permet de récupérer des compteurs générés explicitement dans votre code. Utilisez-le pour afficher des statistiques sur des opérations dont vous voulez connaître la fréquence d'utilisation, ou le volume. Vous pouvez par exemple renvoyer un nombre de lignes affectées par un traitement, ou le nombre de lignes dans une table.

Vous disposez de dix compteurs, numérotés de 1 à 10. Vous devez mettre à jour la valeur du compteur, en exécutant la procédure stockée `sp_user_counterX`, où X est le numéro du compteur. Par exemple, pour passer le nombre de lignes affectées par la dernière instruction de votre procédure au compteur 1, vous écririez :

```
EXEC sp_user_counter1 @@ROWCOUNT
```

Cette valeur sera attribuée au compteur, qui la restituera tant que vous n'exécuterez pas `sp_user_counter1` à nouveau. Ainsi, pour afficher le nombre de lignes d'une table (sa cardinalité), vous pouvez créer un déclencheur sur `INSERT` et `DELETE`, qui exécute cette procédure.

5.4.5 Identification de la session coupable

Lorsque vous constatez une augmentation anormale de l'activité CPU, vous souhaitez sans doute entrer dans les détails et identifier la session SQL Server responsable. C'est possible en corrélant l'identifiant de *thread* Windows et le `kpid` des informations de session SQL Server. Le désavantage de cette opération est son côté fastidieux, et surtout sa durée : il se peut qu'une fois la première partie de l'opération effectuée, la session ait déjà terminé son travail, et donc qu'on arrive trop tard pour recueillir les informations nécessaires.

Vous devez premièrement trouver l'identifiant du *thread* qui consomme un niveau important de CPU. Vous disposez pour cela des compteurs de performances `Thread : % Processor Time` et `Thread : ID Thread`. Sélectionnez toutes les instances Sqlservr. Affichez le résultat en mode rapport et cherchez le *thread* consommateur, comme indiqué en figure 5.15.

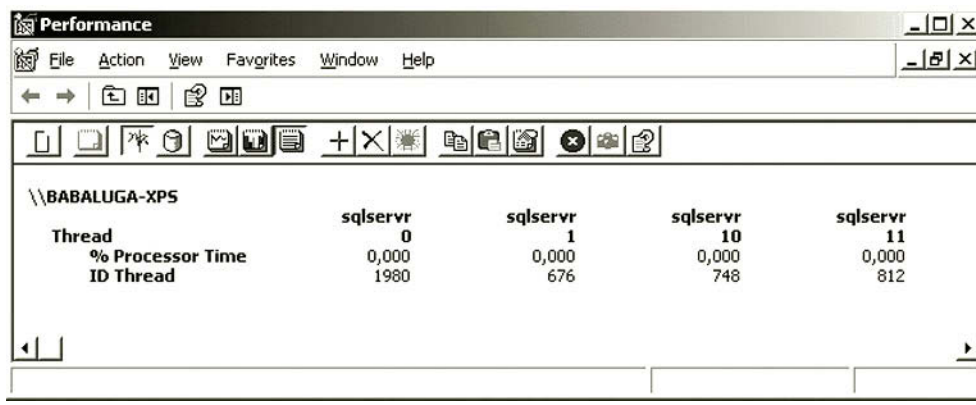

Figure 5.15 — Recherche des threads

Lorsque le *thread* est identifié, vous trouvez la correspondance dans SQL Server à partir de la colonne `kpid` de la vue `sys.sysprocesses` (une vue qui correspond à l'ancienne table système du même nom), qui représente le même identifiant :

```
SELECT *
FROM sys.sysprocesses
WHERE kpid = ID_Thread;
```

5.4.6 Utiliser les compteurs dans le code SQL

La vue système `sys.dm_os_performance_counters` permet de retrouver dans du code SQL des valeurs de compteurs de performance. Elle remplace la table `master.dbo.sysperfinfo` de SQL Server 2000. Cette table est toujours disponible, mais elle est dépréciée. Il est donc recommandé d'utiliser la vue de gestion dynamique pour la remplacer. Cette vue ne permet de retrouver que les compteurs propres aux objets SQL Server, et pas les compteurs des objets système.

Vous pouvez ainsi, dans votre code, requêter ou réagir aux valeurs de compteur. Voici par exemple une requête qui vous indique quel est le pourcentage du journal de transaction rempli, pour chaque base de données :

```
SELECT instance_name,
       cntr_value as LogFileUsedSize
FROM sys.dm_os_performance_counters
WHERE counter_name = 'Log File(s) Used Size (KB)';
```

5.5 PROGRAMMATION DE JOURNAUX DE COMPTEURS

Le moniteur de performances ne permet pas seulement de suivre en temps réel l'activité du serveur, il est aussi capable d'enregistrer cette activité dans des fichiers, afin de conserver une trace rechargeable ensuite dans le même outil pour obtenir une vue sur une longue période. C'est bien sûr l'outil de base pour établir la situation de référence ou *baseline* dont nous avons déjà parlé. Pour ouvrir un fichier de journal de compteur, lancez le moniteur système et, au lieu de suivre les compteurs en temps réel, choisissez de lire les données d'un fichier. Vous avez un bouton sur la barre d'outil, en forme de cylindre, qui vous permet de charger le fichier. Combinaisons de touches : CTRL+T pour voir l'activité du serveur en temps réel, CTRL+L pour ouvrir un fichier de journal.

Pour créer un journal de compteurs, un dossier nommé « performance logs and alerts », comportant une entrée nommée « counter logs » est visible dans l'arborescence de la console (figure 5.16).

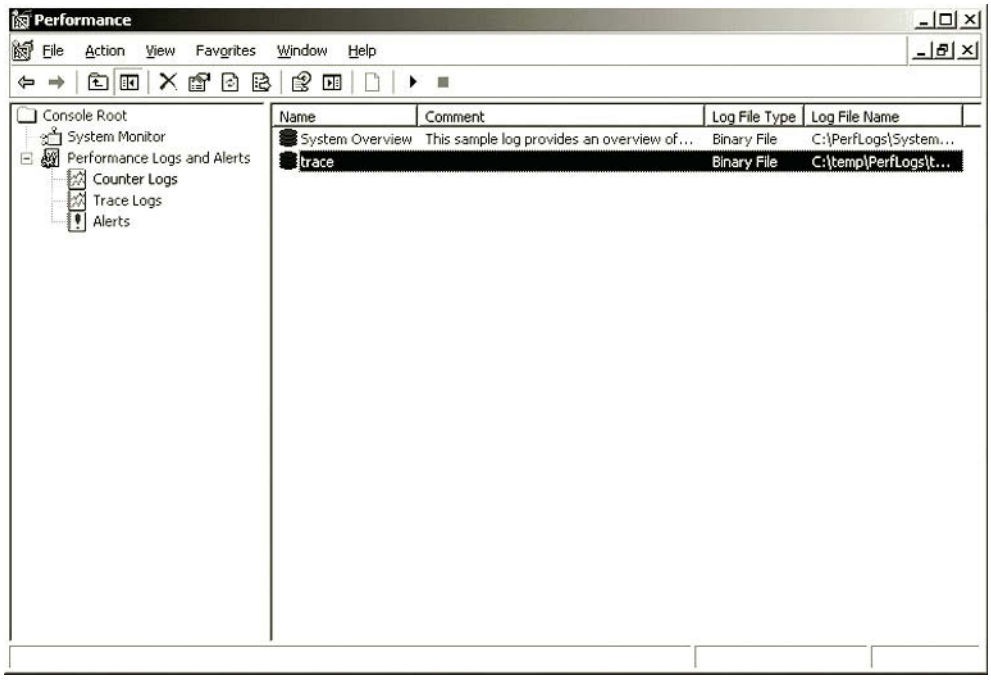

Figure 5.16 — Traces de performance

Vous voyez sur la figure 5.16 une entrée « Trace logs », qui correspond aux traces générées par l'architecture ETW (*Event Tracing for Windows*). Cette partie est utile en SQL Server 2008 pour gérer des sessions d'événements étendus (voir plus loin section 5.3.5).

Par un clic droit sur l'entrée « counter logs », ou dans la fenêtre de droite, vous pouvez créer un nouveau journal en sélectionnant « new log settings… » dans le menu. La première étape consiste à donner un nom à votre définition de journal, puis à sélectionner les compteurs désirés, en cliquant sur le bouton « Add counters… », dans la fenêtre reproduite en figure 5.17.

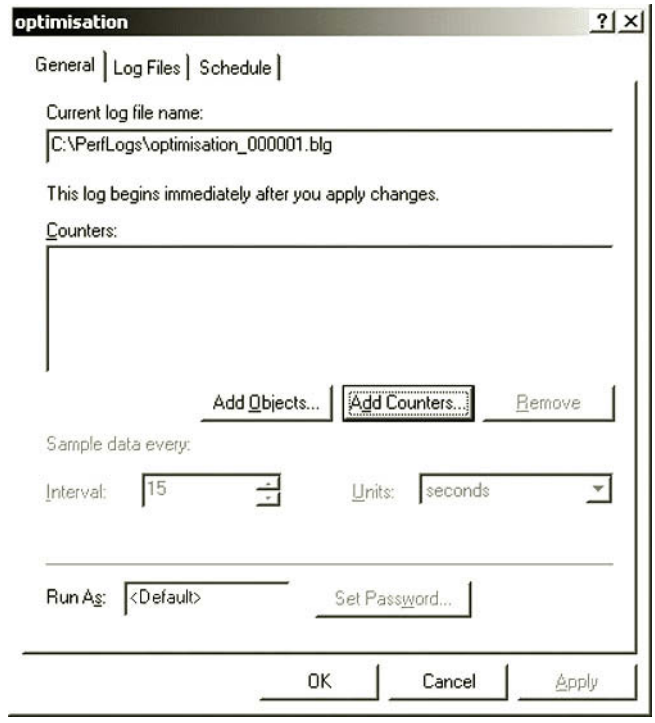

Figure 5.17 — Premier onglet de la configuration de trace de performances

Vous ajoutez ensuite vos compteurs comme dans le moniteur système. Choisissez dans cette même fenêtre l'intervalle d'échantillonnage (*sample data every…*). Un échantillonnage trop fin augmente exagérément la taille des fichiers de journal, et impacte les performances du serveur. Inversement, un intervalle trop important risque de ne pas montrer assez précisément les variations de compteurs et les pics brusques d'activité. Essayez avec l'intervalle qui vous convient. De dix à quinze secondes peut être une bonne valeur de départ.

La fenêtre de configuration de la définition du journal a trois onglets. Le deuxième permet de configurer la destination (figure 5.18).

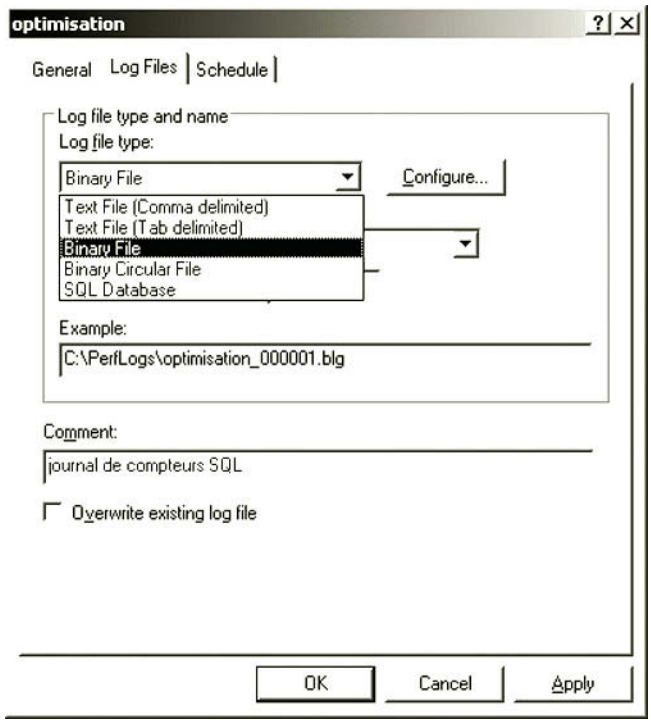

Figure 5.18 — Deuxième onglet de la configuration de trace de performances

Vous pouvez enregistrer les informations de compteurs dans un ou plusieurs fichiers texte ou binaire, ou dans une table SQL. Comme pour le profiler, évitez d'enregistrer directement dans une base de données, pour d'évidentes raisons de performances. Le fichier binaire sera le plus rapide et le plus compact. Choisissez un fichier texte si vous souhaitez importer ensuite votre journal dans une table SQL pour l'inspecter par requêtes, ou si vous importez les résultats dans un outil de supervision ou un générateur de graphes. Vous pouvez limiter la taille du fichier, ou programmer une rotation.

Outils en ligne de commande

Les journaux de performance et les sessions d'événements ETW peuvent être gérés en ligne de commande, grâce à l'exécutable `logman.exe`. Vous pouvez créer, lancer et arrêter des traces. Par exemple, pour démarrer et arrêter un journal de compteur nommé « sqlcounters » créé dans l'interface :

```
logman start sqlcounters
logman stop sqlcounters
```

Les fichiers générés par le journal peuvent être facilement convertis entre des formats texte, binaire, et une table SQL, à l'aide de l'exécutable `relog.exe`, livré avec Windows.

5.6 PROGRAMMATION D'ALERTES

Le moniteur système permet également de **programmer des alertes**, un peu dans l'esprit de l'agent SQL. Vous pouvez utiliser indistinctement l'un ou l'autre outil pour programmer un envoi de notification lorsqu'un compteur atteint un certain seuil, mais aussi – ce qui peut se révéler très pratique – pour déclencher une collecte d'information spécifique.

Imaginez que vous vouliez démarrer une trace SQL et un journal de compteurs dès que l'activité des processeurs atteint 90 %, afin d'obtenir les données au bon moment, c'est-à-dire quand la montée en charge se produit. Vous pourriez programmer une alerte dans l'agent SQL, mais vous pouvez aussi le faire dans le moniteur système. Les actions déclenchées seront les mêmes, voyons-le avec le moniteur système.

Dans le dossier « Performance logs and alerts » du moniteur système, faites un clic droit sur le sous-dossier « Alerts », et choisissez « New Alerts Settings… ». Après avoir choisi un nom parlant, sélectionnez un compteur de performance avec le bouton « Add ». Pour cet exemple, nous prendrons `Processor(_Total)\% Processor Time`.

Le deuxième onglet, « Action », vous permet de réagir à l'alerte. Nous allons utiliser la possibilité de démarrer un journal de compteurs, et de lancer une commande du système. Ainsi, nous pouvons lancer, dès que le besoin s'en fait sentir, un journal de compteurs. Vous pouvez aussi, comme nous l'avons vu, planifier le journal de compteurs pour s'arrêter après une certaine durée, ce qui vous permettrait par exemple de conserver dans des fichiers numérotés séquentiellement, des traces de trente minutes, dès que la charge des processeurs atteint 90 %. En utilisant l'exécution d'une commande, vous pouvez en même temps démarrer une trace. Le plus simple serait, à l'aide d'une commande SQLCMD, de lancer une trace stockée sur le serveur. Nous avons vu la procédure `sp_trace_setstatus` dans la section 5.2.2.

Dans le dernier onglet, montré en figure 5.19, vous indiquez les plages de temps pendant lesquels l'alerte sera active, c'est-à-dire durant quelle période elle peut se déclencher.

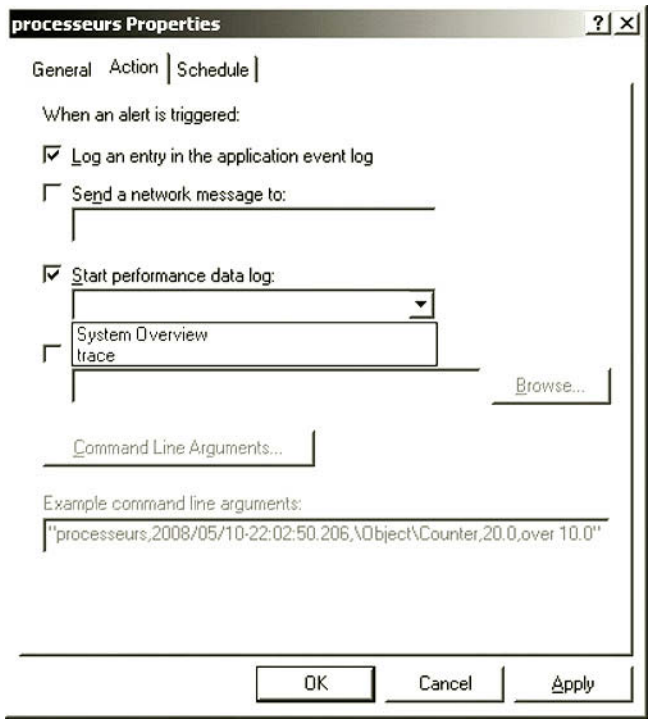

Figure 5.19 — Réaction à l'alerte de performances

5.7 CORRÉLATION DES COMPTEURS DE PERFORMANCE ET DES TRACES

Nous avons vu que vous pouvez enregistrer aussi bien les traces SQL que les journaux de compteur. Si vous collectez les deux jeux d'informations, vous pouvez utiliser l'interface du profiler pour corréler les informations dans un affichage regroupé, très utile pour identifier les causes de surcharge.

Lorsque vous disposez de vos fichiers enregistrés, le journal de compteurs et le fichier de trace, ouvrez d'abord ce dernier dans le profiler. Vous verrez, ensuite, dans le menu Fichier, la commande « Importez les données de performance ». Sélectionnez le fichier de journal de compteur, choisissez les compteurs enregistrés que vous voulez voir apparaître (figure 5.20).

Vous verrez ensuite dans le profiler les deux fenêtres l'une sous l'autre : les événements de trace, puis les graphes de performance (figure 5.21).

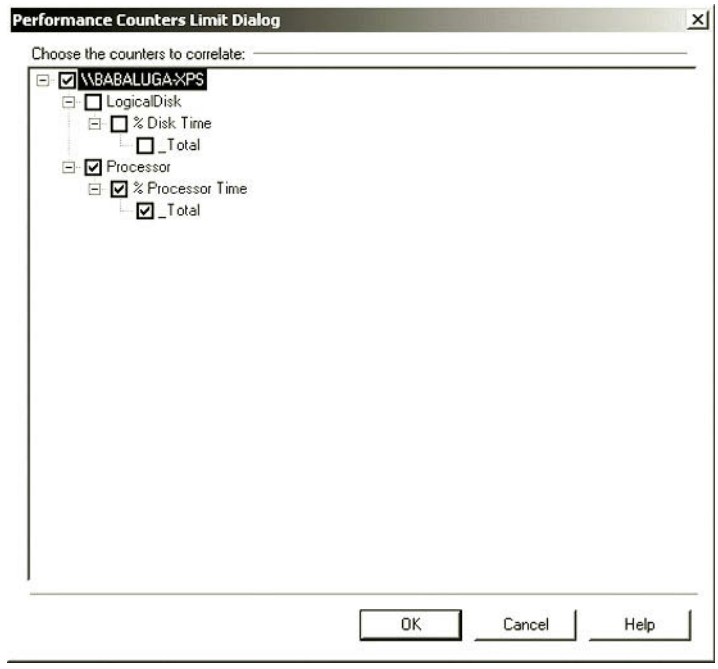

Figure 5.20 — Choix des compteurs pour la corrélation

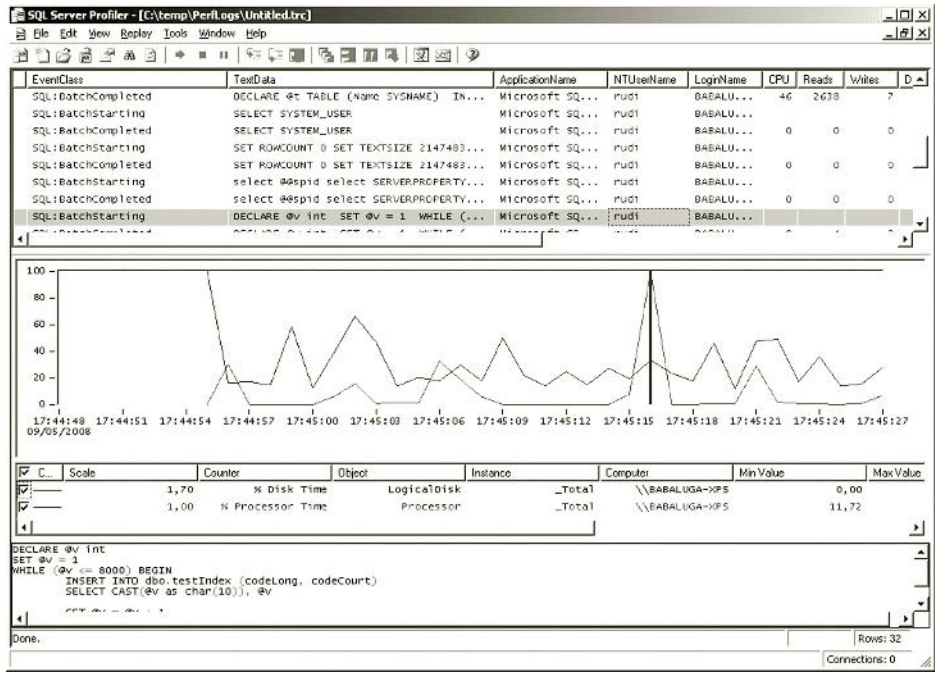

Figure 5.21 — Résultat de la corrélation

Vous pouvez ensuite vous situer sur une ligne d'événement, ou à un moment du graphe, et l'affichage de l'autre fenêtre sera automatiquement positionné sur la période correspondante. Cela permettra une analyse détaillée de l'impact de vos requêtes sur le serveur. Cette fonctionnalité n'est possible qu'avec des traces et des journaux de compteurs sauvegardés préalablement, la corrélation en temps réel n'est pas possible.

5.8 ÉVÉNEMENTS ÉTENDUS (SQL SERVER 2008)

SQL Server 2008 inclut une nouvelle architecture de gestion d'événements, appelée événements étendus (*extended events*, ou *XEvents*), fondés sur l'architecture de *Event Tracing for Windows* (ETW). Elle permet de définir des événements récupérables par plusieurs clients, de façon synchrone ou asynchrone[1]. Au moment où nous rédigeons ce livre, SQL Server 2008 est toujours en version bêta et tous les événements ne sont pas implémentés. La puissance de XEvents est bien supérieure aux méthodes de trace actuelle. Nous allons en parcourir les fonctionnalités.

5.8.1 Architecture

Les fournisseurs de ces événements sont organisés en packages, de simples containers, comme des espaces de noms, identifiés par un nom et un GUID, qui contiennent divers objets :

- **events** : des événements déclenchés, ils correspondent principalement aux compteurs de performance SQL Server et aux événements de trace SQL.
- **actions** : des actions attribuées dynamiquement au déclenchement d'un événement, elles peuvent être utiles pour ajouter des informations supplémentaires aux événements(par exemple le SPID, le code SQL de l'instruction, etc.).
- **targets** : des « consommateurs » d'événements. SQL Server en implémentent plusieurs, soit pour écrire les événements dans un fichier, soit pour les traiter en mémoire (accessible par des vues de gestion dynamique). Par exemple, la destination « Event pairing » matche les couples d'événements, et permet d'identifier des événements non terminés (transaction non validée, verrou non libéré, etc.).
- **types** : indiquent les formats d'événement pour assurer leur prise en main dans le processus de déclenchement (notamment leur mise en *buffer*).

1. Un fournisseur ETW est aussi présent en SQL Server 2005. Vous pouvez trouver des informations sur son fonctionnement dans cette entrée de blog : http://blogs.msdn.com/sqlqueryprocessing/archive/2006/11/12/using-etw-for-sql-server-2005.aspx

- **predicates** : des clauses de filtre. Le filtrage se fait au plus près de l'événement, ce qui évite de déclencher l'événement s'il ne correspond pas au filtre. Les prédicats peuvent conserver leur état, on peut donc par exemple compter les déclenchements d'un événement dans un prédicat, et utiliser ce compteur comme filtre.
- **maps** : des tableaux de référence qui permettent de lier des identifiants courts dans l'événement, à des correspondances claires dans la map. Cela permet, comme dans un modèle relationnel, de diminuer la taille des données d'événement.

Les événements sont classés en « channels » (sur le modèle de ETW). Les événements faisant partie du channel « Analytic » sont ceux qui seront principalement intéressants pour l'analyse de performances. Ils contiennent des événements correspondant aux traces SQL et aux compteurs de performances fournis par SQL Server (avec beaucoup plus d'informations qu'une simple valeur de compteur). Les colonnes `keyword` et `channel` de chaque événement permettent d'en retrouver le *package* et le *channel*, et de les classer. La colonne `keyword` correspond à une information claire tirée d'un lien sur une map.

```
SELECT map_value as Keyword
FROM sys.dm_xe_map_values
WHERE name = 'keyword_map';
```

Vous trouvez une requête permettant de lister les événements disponibles dans l'entrée des BOL SQL Server 2008 nommée « *How to: View the Events for Registered Packages* ».

Le coût de déclenchement des événements étendus a été optimisé autant que possible, et est léger (Microsoft déclare 2 microsecondes de processeur cadencé à 2 GHz). Les événements asynchrones sont à préférer, car ils ont un impact plus faible sur les performances du serveur. Les événements asynchrones sont gérés par des *buffers*, des parties de mémoire qui stockent les événements avant de les envoyer aux clients qui en font la demande ou de les écrire sur le disque. Cette deuxième partie sera effectuée par un *thread* en *background*, ce qui est moins coûteux.

5.8.2 Création d'une session

Pour récupérer des événements étendus, vous devez d'abord créer une session, qui est la déclaration nommée d'un groupe d'événements tracés. Ces sessions étant enregistrées dans des tables systèmes, elles existent donc encore après un redémarrage de l'instance, et elles peuvent être configurées pour redémarrer automatiquement avec elle.

Une session se crée avec la commande `CREATE EVENT SESSION`, dans laquelle nous indiquons les événements à récupérer (`ADD EVENT nom_du_package.evenement`), les destinataires (`ADD TARGET`), etc. :

```
CREATE EVENT SESSION masession
```

```
ON SERVER
ADD EVENT sqlos.async_io_requested
ADD EVENT sqlos.async_io_completed
ADD EVENT sqlserver.database_transaction_begin
ADD EVENT sqlserver.database_transaction_end
ADD TARGET package0.etw_classic_sync_target
    (SET default_etw_session_logfile_path = N'C:\temp\sqletw.etl');
```

Pour démarrer la session :

```
CREATE EVENT SESSION masession
ON SERVER
STATE = start;
```

Vous ne pouvez créer la session et la démarrer en même temps, un peu comme avec les procédures stockées de définition de SQL Trace. Les deux opérations ne peuvent en effet être simultanément transactionnelles : on ne peut pas annuler la création des métadonnées et l'envoi des événements en même temps, puisqu'il est impossible de « désenvoyer » les événements.

Les sessions peuvent être requêtées dans les vues systèmes suivantes :

- `sys.server_event_sessions`
- `sys.server_event_session_actions`
- `sys.server_event_session_events`
- `sys.server_event_session_fields`
- `sys.server_event_session_targets`

Pour améliorer les performances de la session, permettez le vidage de *buffers* en cas de manque, ceci en activant l'option `ALLOW SINGLE EVENT LOSS`, ou `ALLOW MULTIPLE EVENT LOSS`. Le nombre d'événements perdus est indiqué avec l'option `ALLOW SINGLE EVENT LOSS`. L'option `ALLOW MULTIPLE EVENT LOSS` libère en revanche la totalité d'un *buffer*, qui peut contenir un grand nombre d'événements, et rend impossible le retour du nombre d'événements perdus. Dans la plupart des cas, l'option `ALLOW SINGLE EVENT LOSS` est recommandée, car elle offre de bonnes conditions de performances.

NO_EVENT_LOSS

Certains événements bas niveau ne peuvent être paramétrés avec `NO_EVENT_LOSS`, car l'attente sur la libération de *buffer* serait trop handicapante pour le système. C'est le cas d'événements comme `FILE_IO` ou `CONTEXT_SWITCH`. Dans la grande majorité des cas, `ALLOW_SINGLE_EVENT_LOSS` est parfait pour récupérer des informations utiles.

Toute l'architecture des événements étendus est visible à travers des vues de gestion dynamique préfixées par `sys.dm_xe_`.

La vraie question, au moment où nous écrivons ce livre, est : que faire des événements déclenchés ? Deux solutions :

- Enregistrer le résultat de la session dans un fichier .etl, à l'aide du destinataire `etw_classic_sync_target` comme nous l'avons vu dans notre exemple de code. Ce fichier .etl en format binaire est importable directement dans les outils xperf (voir section 5.4) sur Vista ou Windows Server 2008. Avec Windows Server 2003, l'exécutable `tracerpt.exe` (installé avec Windows, dans `system32`), permet de transformer le contenu du fichier au format CSV, qui peut être ensuite importé dans votre outil de supervision.
- Récupérer les événements directement dans la vue de gestion dynamique `sys.dm_xe_session_targets`, à l'aide des destinataires `asynchronous_bucketizer` (écriture asynchrone dans un cache mémoire) et `pair_matching`. Vous trouverez un exemple de destinataire `pair_matching`, qui permet de tracer des événements liés, comme les blocages de processus, dans les BOL, sous l'entrée « *How to: Determine Which Queries Are Holding Locks* ».

5.9 OUTILS DE SUPERVISION

Un certain nombre d'outils, qu'ils soient livrés par Microsoft ou non, sont utiles pour tracer les performances ou des informations précises du système. Dans cette section, nous allons rapidement en lister quelques-uns, afin de vous encourager à les découvrir et à les utiliser.

Windows Performance Toolkit

Le *Windows Performance Toolkit*, aussi appelé **Xperf**, est un outil disponible pour Vista et Windows Server 2008, composé principalement de deux exécutables : `xperf.exe`, qui permet des captures de trace, et `xperfview.exe`, qui les affiche. `xperf.exe` est fondé sur ETW (*Event Tracing for Windows*), que vous pouvez activer dans le traditionnel moniteur de performances (*perfmon*) *via* le nœud « Performance Log and Events / Trace Logs ». Son avantage est de pouvoir corréler des traces de format ETW, par exemple des traces du système, et des traces des événements étendus de SQL Server 2008.

Il est téléchargeable sur le site de Microsoft à l'adresse http://www.microsoft.com/whdc/system/sysperf/perftools.mspx. L'enregistrement de trace est possible sur Windows Server 2003 en copiant à la main l'exécutable `xperf.exe`, mais le processing du résultat de la trace n'est possible que sur Vista ou Windows Server 2008.

SQLDIAG

SQLDIAG est un exécutable installé avec SQL Server, dont le but est de collecter le plus possible d'informations du système, normalement pour les envoyer aux équipes support de Microsoft. Il récupère des données du système et de SQL Server, à la demande (prenant un *snapshot*) ou de façon continue. Il peut être invoqué en ligne de commande, ou installé comme un service. Il est capable de récupérer ses informations d'instances SQL Server en *cluster*. Il enregistre son résultat dans des

fichiers texte dans un répertoire spécifique. Il peut être configuré en ligne de commande, ou par un fichier de configuration. Il peut récupérer des journaux de performance Windows, des traces SQL, les journaux d'événement Windows, les informations de configuration du système et de SQL Server, les informations de blocage de sessions, etc. C'est donc un outil extrêmement intéressant pour le diagnostic de problèmes. Vous trouverez plus d'informations dans les BOL, sous l'entrée « *SQLdiag Utility* ».

SQLNexus

SQLNexus est un outil développé par Bart Duncan et Ken Henderson, qui simplifie l'utilisation de SQLDIAG, tout d'abord en fournissant un fichier de configuration pour SQLDIAG et des scripts pour obtenir des informations supplémentaires des vues de gestion dynamique, ensuite en stockant les *output* dans une base de données, et en offrant des rapports Reporting Services très complets pour analyser les résultats. Vous pouvez le télécharger sur CodePlex : http://www.codeplex.com/sqlnexus

Log Parser

Log Parser n'est pas spécifiquement destiné à SQL Server. C'est un outil très puissant de lecture, de recherche et de reporting qui travaille sur tous types de fichiers journaux (fichiers log, XML, CSV) et dans les journaux d'événement de Windows, la base de registre, etc. Vous pouvez lire, chercher avec un langage de requête de type SQL, formater le résultat en texte ou en HTML, et même générer des graphes. Vous pouvez télécharger Log Parser ici : http://www.microsoft.com/technet/scriptcenter/tools/logparser/. Une interface graphique libre de droits a été créée par un programmeur, vous la trouvez ici : http://www.lizardl.com/.[1]

Windows Sysinternals

Les outils **Sysinternals**, développés par Mark Russinovich (un expert reconnu des méandres de Windows) et Bryce Cogswell, sont maintenant disponibles sur le site de Microsoft, depuis que Winternals, la société de Russinovich, a été rachetée en 2006. Vous trouverez à l'adresse http://technet.microsoft.com/en-us/sysinternals/ des outils de diagnostic système indispensables et simples d'utilisation. Parmi eux :

- **DiskMon** : trace l'activité disque.
- **Process Monitor** : affiche en temps réel l'activité des processus et *threads*, de la base de registre et des activités disque.
- **Process Explorer** : affiche en détail l'activité des processus : *handles*, DLL utilisées, fichiers ouverts, etc. Très utile pour entrer dans les détails.
- **TCPView** : affiche en temps réel les ports TCP et UDP ouverts. Un équivalent graphique de netstat.

1. Sachez également qu'un livre est publié sur cet outil : *Microsoft Log Parser Toolkit*, chez Syngress.

SQL Server 2005 Performance Dashboard Reports

Le « SQL Server 2005 Performance Dashboard » est une collection de rapports Reporting Services requêtant les vues de gestion dynamique. Elle est téléchargeable sur le site de Microsoft, sous forme de fichier d'installation `.msi`. Après installation, vous disposez des fichiers de définition de rapports `.rdl`, dans le répertoire choisi à l'installation, ainsi qu'un fichier de script `setup.sql`, à exécuter avant toute chose. Ce script crée quelques objets dans `msdb`, sans plus. Tous les rapports sont basés sur des DMV. Ensuite, pour afficher les rapports dans SSMS, vous pouvez utiliser la fonctionnalité de rendu de rapports locaux (Reporting Services n'est donc pas requis).

Pour ajouter un rapport local, choisissez « Custom Reports » dans le menu affiché en figure 5.22.

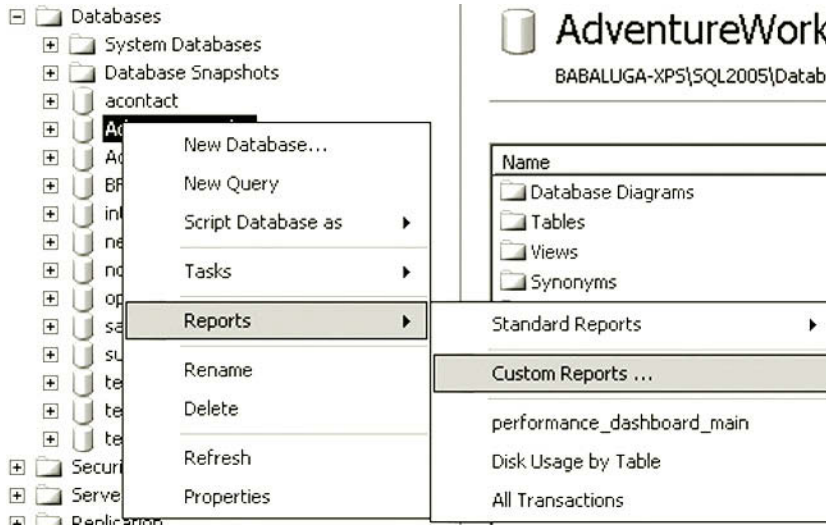

Figure 5.22 — Choix des rapports

Allez chercher ensuite le rapport `performance_dashboard_main.rdl` dans le répertoire d'installation (figure 5.23). Le rapport s'affiche dans une nouvelle fenêtre (figure 5.24).

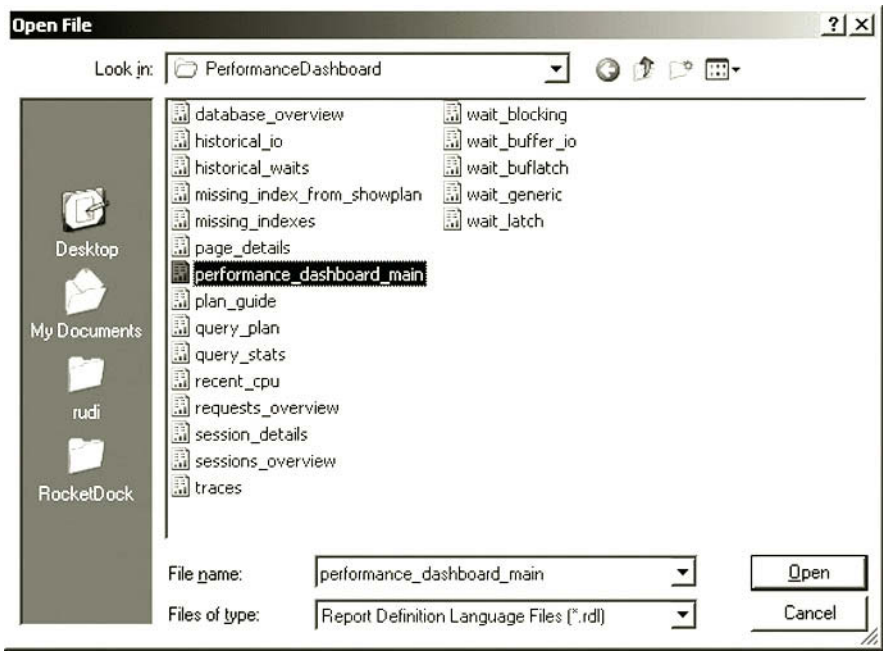

Figure 5.23 — Rapport performance_dashboard_main

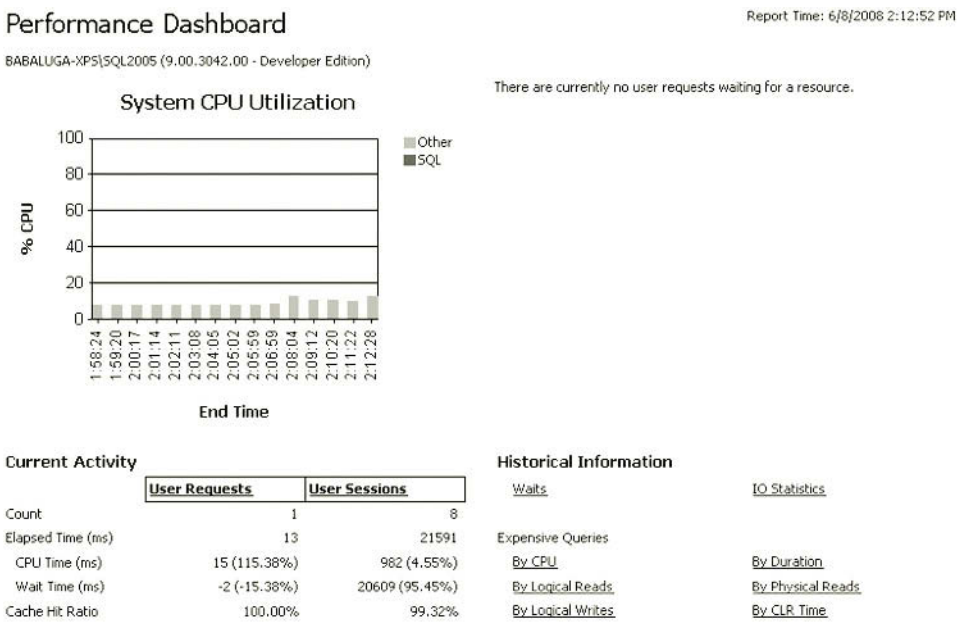

Figure 5.24 — Rapport principal

Ce rapport donne des liens sur tous les autres, c'est donc votre point d'entrée sur les informations de charge, les requêtes les plus coûteuses, etc. Une fois choisi, il continuera à apparaître dans le menu « Reports ».

Server Performance Advisor (SPA)

De son nom complet, « Microsoft Windows Server 2003 Performance Advisor », **SPA** est un outil graphique qui simplifie la collecte de données de performances et offre une vue commentée des performances générales du système, avec des conseils. Vous pouvez le télécharger sur le site de Microsoft, avec une recherche sur son nom complet.

Un autre outil de ce type existe, nommé PAL (*Performance Analysis of Logs*). Il s'agit d'un script VBScript hébergé sur CodePlex (le serveur de développements communautaires tournant autour des produits Microsoft), qui avale un journal de compteurs (voir section 5.3.2) et produit un résultat en tableau HTML indiquant si les valeurs sont en deçà ou au-delà des limites conseillées par Microsoft. Site de l'outil : http://www.codeplex.com/PAL/.

Management Data Warehouse (SQL Server 2008)

SQL Server 2008 intègre un outil de centralisation des données de performances, appelé « *Management Data Warehouse* », qui permet, à travers des lots Integration Services créés par assistant, de centraliser des collections données (*data collections*) de gestion et de performance basées sur les traces, des compteurs de performances, les vues de gestion dynamique, etc., dans un entrepôt centralisé (une base de données) qui peut collecter les données de plusieurs instances SQL. Des rapports Reporting Services locaux sont ensuite disponibles pour consulter ces statistiques. C'est un outil intéressant de surveillance centralisée de vos instances SQL.

DEUXIÈME PARTIE

Optimisation des requêtes

Le travail d'optimisation sur un serveur de production est avant tout un travail de modélisation logique et physique des données et d'amélioration de la qualité du code SQL. Même les machines les plus puissantes peinent à exécuter des requêtes mal écrites, et doivent attendre lorsque des verrous sont posés sur les données désirées. C'est au niveau du code que les problèmes se posent en général, et c'est là où l'effet de levier est le plus important. Ce n'est que lorsqu'on a la garantie que le code et le schéma sont optimisés que la mise à jour matérielle doit être envisagée. Nous allons donc passer du temps sur ce sujet essentiel.

6

Utilisation des index

Objectif

Vous n'ignorez pas que l'indexation est un sujet important. Sans index, les SGBDR n'auraient aucune utilité, car ils ne pourraient permettre aucune recherche rapide dans les données. Toute manipulation dans les tables obligerait le moteur à les parcourir de bout en bout. Même une simple contrainte d'unicité serait irréalisable. Comprendre et utiliser les index au maximum de leurs possibilités est ce qui va vous donner le levier le plus important pour augmenter les performances de vos bases de données. Nous allons donc les voir en détail.

6.1 PRINCIPES DE L'INDEXATION

Vous êtes administrateur de bases de données, votre vie professionnelle est donc bien remplie. Mais cela ne soulage pas des grandes questions métaphysiques : quel est le sens de la vie ? Pour y répondre, vous décidez de rencontrer un philosophe. Qu'allez-vous faire pour en trouver un dans votre ville ? Bien entendu, vous allez consulter l'annuaire téléphonique, et plus précisément les pages jaunes. Si vous ne disposez pas d'annuaire, vous n'avez d'autre solution que d'errer dans les rues, pour y chercher au hasard, sur une plaque ou une boîte aux lettres, la mention du métier de philosophe. SQL Server n'est pas différent. Lorsque vous filtrez le contenu d'une table dans une requête, à l'aide en général d'un prédicat dans une clause `WHERE`, SQL Server doit parcourir la table pour évaluer ce prédicat sur chaque ligne. Cette opération est évidemment très coûteuse, proportionnellement à la taille de la table. Toute la table doit être mise en mémoire et parcourue, c'est-à-dire que toutes les pages de

données qui contiennent la table sont chargées, ce qui peut signifier des gigaoctets pour un simple `SELECT`… Ce parcours est appelé un **scan**.

Le seul moyen d'éviter cela, est de créer un index. Un index est proprement une sorte de table des matières, contenant dans une structure arborescente toutes les valeurs présentes dans une ou plusieurs colonnes, qui permet une recherche optimisée sur les valeurs qu'elle contient. Cet arbre est appelé **arbre équilibré** (*balanced tree*, ou *B-Tree*).

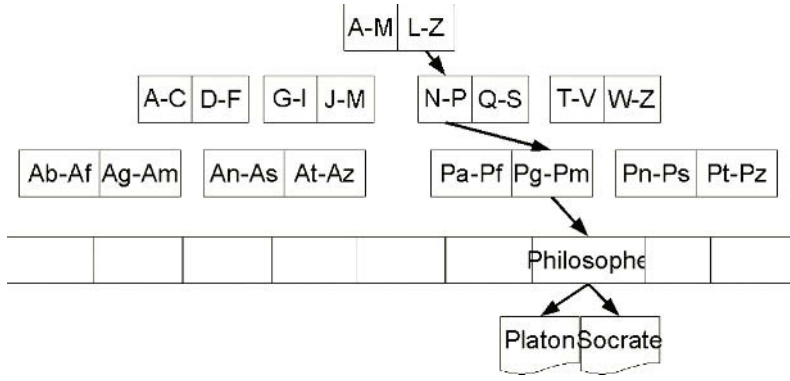

Figure 6.1 — Arbre équilibré

La figure 6.1 montre une recherche à travers l'arbre de l'index. Cette représentation en arbre est schématique : l'index est physiquement stocké dans des pages de données identiques à celles qui abritent le contenu des tables. Nous verrons ce stockage un peu plus loin. L'arbre *B-Tree* est constitué de nœuds, ou niveaux, qui sont parcourus pour trouver l'information. La figure 6.2 montre un index à quatre niveaux. Le premier nœud, appelé nœud racine (*root node*), est le point d'entrée de la recherche. Dans notre cas, nous cherchons le mot « philosophe ». Nous sommes donc dirigés d'abord vers la plage alphabétique comportant la lettre P. Au fur et à mesure que nous descendons les nœuds intermédiaires (*intermediate node*), le choix s'affine, jusqu'au dernier niveau, le nœud feuille (*leaf node*), où réside chaque occurrence de la clé, avec la référence des lignes de la table qui la contiennent. Ces références se nomment des RID (*Row ID*), ce sont des combinaisons de numéros de fichier, page et ligne, qui permettent au moteur de stockage de pointer sur la ligne.

Cette recherche à travers l'arborescence de l'index est représentée dans le plan d'exécution comme un *seek*. La récupération du RID est appelée un **bookmark lookup**, ou **RID lookup**. Vous en trouvez un exemple dans le plan d'exécution graphique reproduit en figure 6.3.

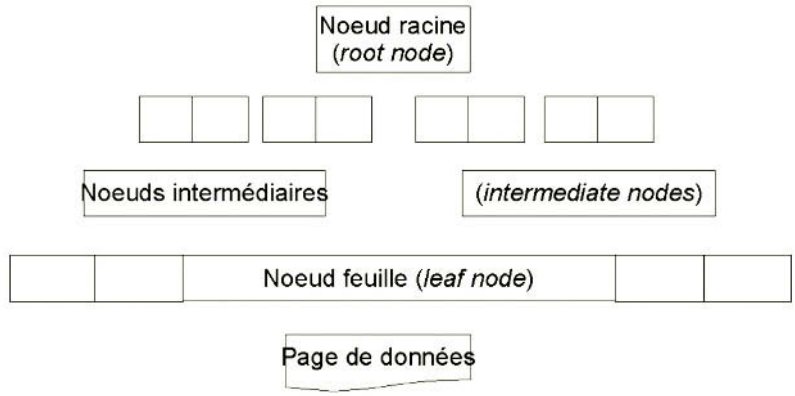

Figure 6.2 — Nœuds de l'index

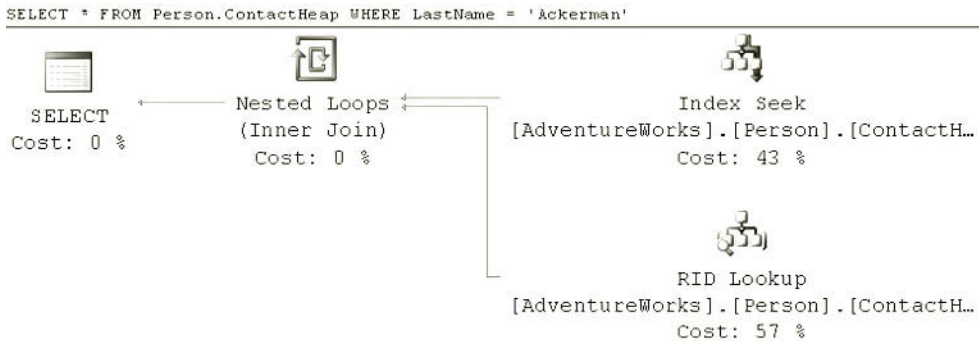

Figure 6.3 — Plan d'exécution, *seek* et *RID lookup*

Vous voyez dans ce plan d'exécution que RID lookup coûte 57 % relativement au coût total de la requête. Il s'agit donc d'une opération lourde, nous reviendrons sur ce point. La recherche de chaque RID est exprimée par un opérateur de boucle imbriquée (*nested loop*). Cela veut simplement dire que pour RID trouvé par le *seek* de l'index, il devra se produire une recherche de RID dans la table *heap*. Un peu comme les épreuves de l'émission télévisée « Intervilles ». Une équipe doit transporter des seaux d'eau à travers un parcours difficile, mais chaque participant ne peut bien sûr transporter que deux seaux à la fois. Ils doivent donc constamment revenir au point de départ pour prendre les seaux restants. C'est le *nested loop* : à chaque apparition de nouveau *match* dans l'index, le moteur de requête « déréférence » le pointeur RID et va chercher physiquement la ou les lignes dans la page de données.

Chaque niveau de l'index est une liste doublement liée entre les pages de ce niveau. En d'autres termes, chaque page d'index contient les identifiants de la page

précédente, et de la page suivante, du niveau. Cela permet à SQL Server de parcourir (*scan*) le niveau dans l'ordre de la clé.

> ***Scan* avancé – édition Entreprise**
>
> L'édition Entreprise offre une optimisation avancée des *scans* (*Advanced Scanning*) : lorsque plusieurs requêtes doivent parcourir simultanément la même table, l'édition Entreprise peut partager ce parcours entre les processus. La fonctionnalité est appelée « parcours en manège » (*Merry-go-round scanning*). Un processus se joint au parcours déjà initié, et lorsque celui-ci est terminé, il le poursuit sur les pages qui n'ont pas été parcourues depuis son entrée dans le circuit. Cela peut fortement diminuer le temps de réponse de requêtes qui doivent effectuer des *scan* importants, en partageant le travail et en diminuant l'attente sur les verrous.

Contrairement à d'autres SGBD, qui implémentent des index de type bitmap, ou en hachage, il n'y a pour l'instant qu'une seule structure physique d'index en SQL Server : le *B-Tree* (à part l'index de texte intégral, un peu à part). Même les index XML, et les index spatiaux sur les données géographiques sont en interne des index de structure *B-Tree*, sur une table interne pour le XML, et sur une décomposition hiérarchique de l'espace pour les index spatiaux. L'index *B-Tree* peut être de deux types : *nonclustered* et *clustered* (parfois appelés non ordonné, et ordonné).

6.1.1 Index *clustered*

Nous avons vu que le niveau feuille de l'index pointe sur un RID, et que l'opération de RID lookup permet de retrouver les lignes de la table même. Cette structure, où l'index est physiquement séparé de la table et où les clés sont liées aux lignes par un pointeur, est un index *nonclustered*. Un autre type d'index *B-Tree*, nommé *clustered*, réorganise physiquement les pages de données selon la clé de l'index. Commençons par une démonstration.

```
USE tempdb
GO

CREATE TABLE dbo.indexdemo (
       id int NOT NULL,
       texte char(100) NOT NULL
              DEFAULT (REPLICATE(CHAR(CEILING(RAND()*100)), 100))
);
GO

INSERT INTO dbo.indexdemo (id) VALUES (1)
INSERT INTO dbo.indexdemo (id) VALUES (2)
INSERT INTO dbo.indexdemo (id) VALUES (3)
INSERT INTO dbo.indexdemo (id) VALUES (4)
INSERT INTO dbo.indexdemo (id) VALUES (5)
INSERT INTO dbo.indexdemo (id) VALUES (6)
INSERT INTO dbo.indexdemo (id) VALUES (7)
INSERT INTO dbo.indexdemo (id) VALUES (8)
GO
```

```
SELECT * FROM dbo.indexdemo
GO
```

À l'issue de l'exécution de ce code, nous obtenons un résultat semblable à ce qui est présenté figure 6.4.

	id	texte
1	1	NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN...
2	2	RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR...
3	3	$$$...
4	4	[illegible]
5	5	%%...
6	6	44444444444444444444444444444444444444444444444444444444444444444...
7	7	!!
8	8	MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM...

Figure 6.4 — Résultat du SELECT

Les lignes apparaissent « naturellement » dans l'ordre d'insertion, qui correspond ici à un ordre de nombres dans la colonne ID, simplement parce que nous avons inséré les lignes avec un incrément de la valeur de ID. Si nous avions inséré des ID dans le désordre, ils seraient apparus dans le désordre. Voyons maintenant ce qui se passe si nous supprimons, puis ajoutons une ligne :

```
DELETE FROM dbo.indexdemo WHERE id = 4
INSERT INTO dbo.indexdemo (id) VALUES (9)
GO
SELECT * FROM dbo.indexdemo
GO
```

Résultat sur la figure 6.5.

	id	texte
1	1	NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN...
2	2	RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR...
3	3	$$$...
4	9	UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU...
5	5	%%...
6	6	44444444444444444444444444444444444444444444444444444444444444444...
7	7	!!
8	8	MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM...

Figure 6.5 — Résultat du SELECT

Vous voyez que la ligne portant l'ID 9 est insérée à la place de la ligne supprimée. La suppression a libéré 104 octets dans la page, une insertion de la même taille s'est faite dans l'espace laissé libre. Cela permet de remplir la page. Une table *heap* n'est vraiment organisée selon aucun ordre logique ou physique. Si nous voulons obtenir un résultat ordonné, nous devons le spécifier dans la requête :

```
SELECT * FROM dbo.indexdemo ORDER BY id;
```

Ce qui nécessitera un tri dans le plan de la requête, comme nous le voyons dans le plan graphique, sur la figure 6.6.

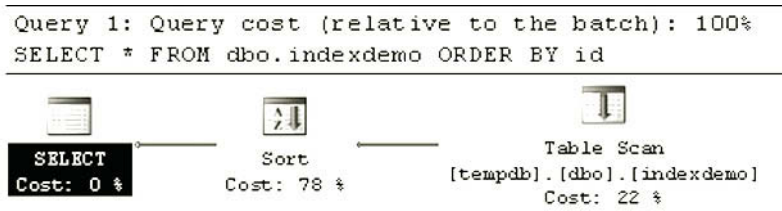

Figure 6.6 — Plan d'exécution

Nous pouvons créer un index sur la colonne `ID`, pour permettre ce tri. Essayons :

```
CREATE NONCLUSTERED INDEX nix$dbo_indexdemo$id
ON dbo.indexdemo (id ASC);
GO

SELECT * FROM dbo.indexdemo ORDER BY id;
```

Nous voyons la différence de plan d'exécution sur la figure 6.7.

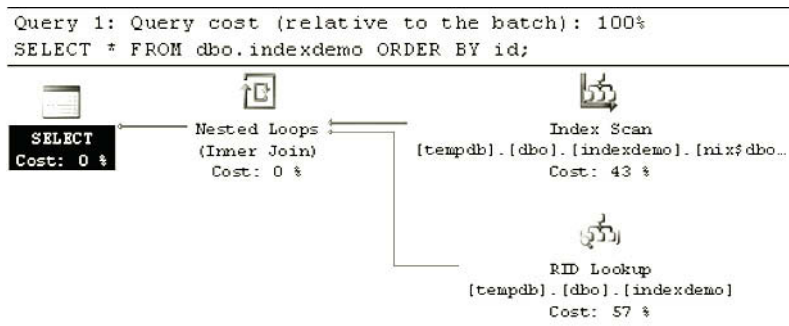

Figure 6.7 — Plan d'exécution

SQL Server parcourt le nœud feuille de l'index, qui lui donne rapidement l'ordre par ID. Ensuite, il doit faire un RID lookup sur chaque ligne.

Ici, l'exemple donne un plan d'exécution peu optimal. En effet, la boucle imbriquée implique de faire une lecture par occurrence dans le parcours de l'index, donc de lire chaque fois une page. Mais, notre petite table tient tout entière dans une page de données. Donc, au lieu de lire une page, puis de trier, SQL Server fait dix lectures. À ce volume de données, ce n'est pas important, et SQL Server choisit un plan d'exécution « acceptable ». Si la cardinalité de la table est plus importante, l'optimiseur fera le choix d'un *scan*, même en présence de l'index.

Supprimons cet index, et créons à la place, un index ***clustered*** :

```
DROP INDEX nix$dbo_indexdemo$id
ON dbo.indexdemo;
GO

CREATE CLUSTERED INDEX cix$dbo_indexdemo$id
ON dbo.indexdemo (id ASC);
GO
```

que donne un simple

```
SELECT * FROM dbo.indexdemo;
```

Nous voyons le résultat sur la figure 6.8.

	id	texte
1	1	NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN...
2	2	RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR...
3	3	$$$...
4	5	%%...
5	6	44444444444444444444444444444444444444444444444444444444444444444444...
6	7	!!
7	8	MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM...
8	9	UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU...

Figure 6.8 — Résultat du SELECT

L'ordre des lignes a changé, sans même que nous n'ayons spécifié une clause `ORDER BY`. Que s'est-il passé ? Simplement, la création de l'index *clustered* a réorganisé les lignes de la table dans la page de données, selon l'ordre de la clé de l'index *clustered*. En d'autres termes, la table est maintenant physiquement ordonnée selon l'index *clustered*. Désormais, toute ligne insérée ou supprimée sera placée à un endroit précis dans l'ordre de la table. Si nous supprimons une ligne, en ajoutons une autre, nous verrons que, contrairement à l'exemple précédent sur notre table *heap*, les lignes retournées par ce simple `SELECT` seront toujours dans l'ordre le l'ID, car c'est l'ordre physique des lignes dans la table.

Attention – Ce comportement ne vous dédouane pas de spécifier la clause `ORDER BY` si vous souhaitez récupérer un jeu de résultat dans un tri précis. Vous n'avez aucune garantie que l'ordre des lignes sera consistant, même s'il existe un index *clustered*. Dans certains opérateurs de requête (les `UNION`, les jointures par exemple), SQL Server peut choisir de pré-trier un résultat intermédiaire pour rendre les opérations suivantes plus rapides. De même, le moteur de stockage peut retourner les pages dans un ordre favorable aux performances plutôt que dans leur ordre « logique ».

Qu'en est-il de la requête avec `ORDER BY id`, et de son plan d'exécution ? Réponse figure 6.9.

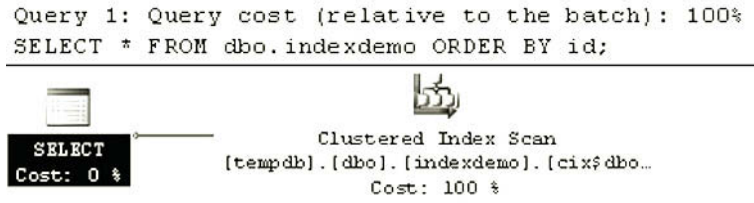

Figure 6.9 — Plan d'exécution

Plus besoin d'opérateur de tri : la table est déjà dans l'ordre demandé. Il suffit de faire un *scan* de l'index *clustered*. Mais, pourquoi n'avons-nous plus de RID lookup ? Comment SQL Server fait-il pour aller chercher la colonne `[texte]` que nous voulons afficher ? Utilisons `DBCC IND` pour observer la table, et l'index. Le dernier paramètre de `DBCC IND` indique l'ID de l'index. Un index *clustered* a toujours l'ID 1. Regardons donc ce que donne `DBCC IND` sur l'index 0 (la table elle-même), et l'index 1 (l'index *clustered*) :

```
DBCC IND ('tempdb', 'dbo.indexdemo', 0);
DBCC IND ('tempdb', 'dbo.indexdemo', 1);
```

Résultat sur la figure 6.10.

PageFID	PagePID	IAMFID	IAMPID	ObjectID	IndexID	PartitionNumber	PartitionID	iam_chain_type	PageType	IndexLevel
1	210	NULL	NULL	2053582354	1	1	72057594039959552	In-row data	10	NULL
1	206	1	210	2053582354	1	1	72057594039959552	In-row data	1	0

PageFID	PagePID	IAMFID	IAMPID	ObjectID	IndexID	PartitionNumber	PartitionID	iam_chain_type	PageType	IndexLevel
1	210	NULL	NULL	2053582354	1	1	72057594039959552	In-row data	10	NULL
1	206	1	210	2053582354	1	1	72057594039959552	In-row data	1	0

Figure 6.10 — Résultat de DBCC IND

Ici nous constatons plusieurs choses : le résultat pour la table et pour l'index *clustered* est le même. La page d'index et la page de table ont le même ID, c'est donc la même. Nous voyons aussi que la page de données se trouve au niveau 0 (le nœud feuille) de l'index *clustered* (1). Cela signifie donc que la page de données, et la page du niveau feuille de l'index *clustered*, est la même.

En effet, lorsque vous créez un index *clustered*, la table devient partie de l'index : le nœud feuille de l'index *clustered* est… la table elle-même ! Il est important de comprendre ce fait. Une table de type *heap*, et une table *clustered*, sont deux objets bien différents. Lorsque SQL Server effectue un *seek* dans un index *clustered*, il ne quitte jamais l'index, et la notion de RID n'existe plus. Quand la recherche arrive au dernier niveau de l'index, elle est terminée, car on se trouve déjà au bon endroit : dans la page de données. Ainsi, contrairement à un index *nonclustered*, ce qui se trouve dans le nœud feuille n'est pas uniquement la clé de l'index, mais aussi toutes les autres colonnes de la table.

Cela implique également que la table *clustered* comporte un ordre de pages. Comme la table est le dernier niveau de l'index, elle maintient comme les index une liste doublement liée de ses pages. Cela permet un *scan* de la table, pour retrouver des lignes dans l'ordre de la clé de l'index *clustered*. Une table *heap*, par contre, n'a aucune notion de page précédente ou page suivante, puisque ses lignes ne sont organisées dans aucun ordre explicite. Pour un *heap*, SQL Server sait qu'une page fait partie de la table, en inspectant les pages IAM de la table.

Bien qu'elle n'ait aucune particularité d'un index, la table *heap* est elle aussi, en interne, considérée comme un index : elle apparaît dans la vue `sys.indexes`, comme un index de type *heap*, et ses pages sont allouées également par des pages d'IAM (*Index Allocation Map*).

Qu'en est-il des index *nonclustered* sur une table *clustered* ? Ils sont aussi d'une autre espèce. Observons leur différence fondamentale. Pour cela, nous allons ajouter un certain nombre de lignes dans notre table de démo, et une troisième colonne que nous allons indexer :

```
DECLARE @i int
SELECT @i = MAX(id)+1 FROM dbo.indexdemo;

WHILE @i <= 2000 BEGIN
    INSERT INTO dbo.indexdemo (id) SELECT @i;
    SET @i = @i + 1
END

ALTER TABLE dbo.indexdemo
ADD petittexte CHAR(1) NULL;
GO
UPDATE dbo.indexdemo
SET petittexte =  CHAR(ASCII('a')+(id%26))
```

```
GO

CREATE NONCLUSTERED INDEX nix$dbo_indexdemo$petittexte[1]
ON dbo.indexdemo (petittexte ASC);
GO

SELECT * FROM dbo.indexdemo WHERE petittexte = 'c';
```

Nous pouvons voir le plan d'exécution de la dernière requête `SELECT` dans la figure 6.11.

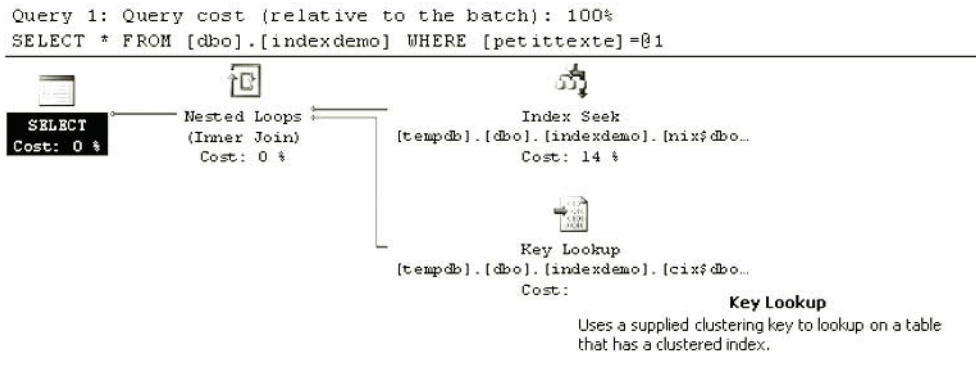

Figure 6.11 — Plan d'exécution

Pourquoi avons-nous augmenté le nombre de ligne à 2000 ?

Sans cela, le plan d'exécution choisi aurait été un *scan* de table. Le coût du lookup étant important, l'optimiseur choisit plus volontiers un *scan* de la table si le nombre de pages est petit.

Ici, plus de RID lookup, mais un « *Key Lookup* ». La description de cet opérateur nous le dit clairement : c'est l'utilisation de la clé de l'index *clustered* pour effectuer la recherche dans la table. La conclusion est simple : l'index *nonclustered*, dans son nœud feuille, ne fournit plus de RID pour la recherche, mais la clé de l'index *clustered*, de sorte que, dans ce cas, il y a en réalité un double *seek* d'index. Lorsqu'un *seek* est effectué dans un index *nonclustered* créé sur une table *clustered*, arrivé au nœud feuille de l'index *nonclustered*, la clé de l'index *clustered* est trouvée, et ce dernier est traversé à son tour. Ceci est schématisé par la figure 6.12.

1. L'utilisation des $ dans le nommage des index et contraintes est une pratique personnelle, pas forcément conseillée. SQL Server supporte le $ dans un identifiant d'objet, la norme SQL, non.

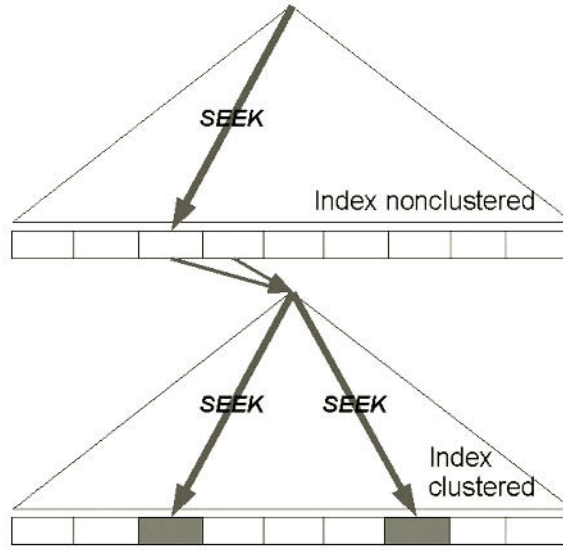

Figure 6.12 — Parcours du *Key Lookup*

Mais c'est encore un peu plus que cela. La clé de l'index *clustered* est ajoutée à la fin de la clé de l'index *nonclustered*. Ce qui fait que, implicitement, l'index *nonclustered* est un index composite, contenant en plus la ou les clés de l'index *clustered*. Ce n'est pas montré dans la définition de l'index, ni dans la vue système `sys.index_columns`, mais si vous observez l'index, soit avec `DBCC SHOW_STATISTICS` (que nous verrons plus loin), soit avec `DBCC PAGE`, vous pouvez voir cela. Par exemple :

```
DBCC IND ('tempdb', 'dbo.indexdemo', 4)
GO

DBCC TRACEON (3604);
DBCC PAGE (tempdb, 1, 218, 3);
GO
```

Chez nous, l'index *nonclustered* `nix$dbo_indexdemo$petittexte` a l'index_id 4. En exécutant `DBCC IND`, nous voyons que la page contenant le nœud racine porte le numéro 218. Nous utilisons `DBCC PAGE` pour voir le contenu de ce nœud (plus de détail sur l'utilisation de ces commandes DBCC dans la section « Structure interne de l'index »). Le résultat est présenté figure 6.13.

	PagePID	IAMFID	IAMPID	ObjectID	IndexID	PartitionNumber	PartitionID	iam_chain_type	PageType	IndexLevel
1	211	NULL	NULL	2053582354	4	1	72057594040090624	In-row data	10	NULL
2	207	1	211	2053582354	4	1	72057594040090624	In-row data	2	0
3	218	1	211	2053582354	4	1	72057594040090624	In-row data	2	1
4	219	1	211	2053582354	4	1	72057594040090624	In-row data	2	0
5	220	1	211	2053582354	4	1	72057594040090624	In-row data	2	0
6	221	1	211	2053582354	4	1	72057594040090624	In-row data	2	0

	FileId	PageId	R...	Le...	ChildFileId	ChildPageId	petittexte (key)	id (key)	UNIQUIFIER (key)	KeyHashValue
1	1	218	0	1	1	207	NULL	NULL	NULL	(7b01df369dd9)
2	1	218	1	1	1	220	NULL	937	NULL	(ad004a8eea73)
3	1	218	2	1	1	221	NULL	1673	NULL	(8d007c51f585)
4	1	218	3	1	1	219	a	26	NULL	(7e003a5ccc3e)

Figure 6.13 — Résultat de DBCC PAGE

Nous y voyons trois colonnes de clé : `petittexte`, `ID`, et un unifiant (UNIQUIFIER), ici toujours `NULL`. La colonne `ID` fait donc bien partie de la clé de l'index, déjà au nœud racine.

Uniquifier

Il est utilisé dans le cas d'un index *clustered* non unique. Un index *clustered* comporte, en interne, nécessairement des clés uniques (sinon, comment un Key Lookup pourrait trouver une seule ligne ?). Si un tel index est créé sans contrainte d'unicité (index unique, ou sur une clé primaire ou unique), SQL Server ajoute une valeur unifiante à la fin de la clé lorsque des valeurs doublonnées sont présentes dans la clé.

Cela veut-il dire que SQL Server va beaucoup déplacer les lignes de pages ?

```
SELECT
    i.index_id, i.name, i.type_desc,
    i.is_primary_key, fill_factor,
    INDEX_COL(OBJECT_NAME(i.object_id), i.index_id, ic.index_column_id)
        as column_name
FROM sys.indexes i
JOIN sys.index_columns ic
    ON  i.object_id = ic.object_id
    AND i.index_id = ic.index_id
WHERE i.object_id = OBJECT_ID('dbo.indexdemo')
ORDER BY i.index_id, ic.key_ordinal;
```

Donc, un *seek* est un parcours de l'arbre équilibré de l'index, pour obtenir avec la ligne correspondante avec un maximum d'efficacité. Plus le nombre de *seeks* est élevé, plus l'index est réellement utilisé à sa pleine capacité. Alors qu'un *scan* (« analyse » dans la traduction française de SQL Server) est le parcours complet du nœud feuille (*leaf node*) de l'index. Le nœud feuille est le dernier niveau, autrement dit la base de l'index, où chaque recherche se termine. Un *scan* sur un index *clustered* correspond à un *scan* sur la table, puisque le nœud feuille d'un index *clustered* est la table elle-même.

Le **lookup** (*bookmark lookup*), ou « recherche de clés » dans la traduction française, correspond à la nécessité, lorsqu'une recherche sur l'index *nonclustered* a atteint le nœud feuille, de retrouver les lignes correspondantes dans la table. Les colonnes lookup de la vue `sys.dm_db_index_usage_stats` indiquent que l'index a participé à une opération de recherche de clés. Elle n'a de sens que sur un index *clustered*. En effet, la recherche de clé ne se produit qu'à partir d'un index *nonclustered*. Cette recherche peut se faire soit sur un RID (*Row ID*, ou identifiant de ligne) dans le cas d'une table « *heap* » (sans index *clustered*), soir sur la clé de l'index *clustered* si la table en comporte un. Dans ce dernier cas, la recherche de clé se fait donc par un parcours de l'index *clustered*. C'est ce parcours qui est indiqué dans les colonnes `lookup` de la vue dynamique.

Le *bookmark look*up est une opération lourde, parce qu'elle provoque des lectures aléatoires (*random IO*), c'est-à-dire des lectures non pas en séquence de lignes dans les pages, mais de lignes qui peuvent se situer n'importe où. Comme lorsque vous cherchez dans un dictionnaire : il est beaucoup plus coûteux de prendre chaque nom dans l'index de fin d'ouvrage, et de vous rendre à la bonne page, que d'avoir un dictionnaire dont les entrées sont déjà triées, et de feuilleter les pages sur la lettre qui vous intéresse. L'optimiseur essaie donc d'éviter autant que possible le *bookmark lookup*, en choisissant un *scan* dès que le nombre de lignes à retourner atteint un certain seuil.

Un index a une profondeur (le nombre de niveaux de l'arbre) et une étendue (le nombre de clés, et physiquement de pages à chaque niveau). Bien entendu, l'étendue augmente au fur et à mesure que l'on descend dans les niveaux.

6.1.2 Choix de l'index

Dans quel cas choisir un index *clustered*, et dans quel cas choisir un index *nonclustered* ?

Comme le dernier niveau de l'index est la ligne elle-même, toute recherche à travers **l'index *clustered*** sera très rapide, puisque l'étape de lookup sera inutile. L'index *clustered* est à choisir soigneusement, car, bien entendu, on ne peut créer qu'un seul index *clustered* par table (comme il n'existe qu'un seul ordre alphabétique dans un dictionnaire… ce n'est que dans le monde de la physique quantique qu'un objet peut se dupliquer et prendre deux natures différentes. Une table ne peut, elle, être physiquement triée par deux clés différentes à la fois).

Comme l'index *clustered* ordonne physiquement les lignes, il est aussi idéal sur des recherches de plages de valeurs. Les clauses de recherche utilisant un `BETWEEN`, ou une syntaxe comme :

```
SELECT *
FROM dbo.indexdemo
WHERE id > 10 AND id < 50;
```

profitera aussi grandement d'un index *clustered*.

D'un autre côté, tout ajout d'une ligne entraîne inévitablement une réorganisation physique de la table. L'ordre des lignes doit se maintenir, et cela a un coût. L'ajout d'une ligne dont la clé *clustered* doit s'insérer à l'intérieur de l'index, peut provoquer une séparation (*split*) de page, et donc entraîner des écritures coûteuses du moteur de stockage. Pour éviter ce coût de maintenance, il est fortement recommandé de créer un index *clustered* sur une colonne qui augmente de façon séquentielle (on dit aussi monotone), comme un auto-incrémental (`IDENTITY`) ou une colonne d'horodatage. Pour la même raison, les mises à jour de la valeur de la clé sont déconseillées. Pour ces raisons, on crée souvent l'index *clustered* sur la clé primaire de la table – d'ailleurs la clé primaire est par défaut créée *clustered*. Dans un modèle de données bien conçu, à quelques rares exceptions près, la clé primaire est immuable (on ne change pas un identifiant référencé dans des tables filles…). L'index *clustered* est aussi préférable sur une colonne, ou un ensemble de colonnes, unique. Nous avons vu qu'en interne, SQL Server doit unifier les clés de l'index *clustered* si elles ne le sont pas déjà, simplement parce que pour ordonner les lignes, il ne peut tolérer de doublon. Créer un index *clustered* sur une clé non unique augmente la taille de l'index. C'est un défaut avec lequel on va parfois composer, car les avantages de l'index *clustered* sont réels. Ceci dit, il ne faut pas sous-estimer l'importance de conserver un index *clustered* aussi petit que possible. Comme sa clé est copiée dans la clé de tous les autres index de la table, il influence la taille de tous les index, et donc leur efficacité et leur utilité. Prenons un peu d'avance sur la section traitant des statistiques, et faisons une expérience :

```
DROP INDEX cix$dbo_indexdemo$id
ON dbo.indexdemo;

CREATE CLUSTERED INDEX cix$dbo_indexdemo$texte
ON dbo.indexdemo (texte ASC);
GO
DBCC SHOW_STATISTICS ('dbo.indexdemo', 'nix$dbo_indexdemo$petittexte');
```

Nous supprimons l'index *clustered* sur notre table de test, et nous le recréons, mais cette fois-ci sur la colonne texte, qui est de type `CHAR(100)`. Voyons comment l'index `nix$dbo_indexdemo$petittexte`, sur la colonne `petittexte (CHAR(1))`, a réagi. La commande `DBCC SHOW_STATISTICS` nous permet d'obtenir des détails sur la structure de l'index, qui sont ceux qu'utilise l'optimiseur SQL pour juger de sa pertinence pour répondre à une requête. Voyons un extrait du résultat (nous n'avons gardé que les lignes et colonnes utiles pour notre propos) :

```
Name                           Rows    Density      Average key length
-------------------------------------------------------------------------
nix$dbo_indexdemo$petittexte   2007    0,1304348    100,0992

All density   Average Length Columns
---------------------------------------------
0,03703704    0,09915297     petittexte
0,003496503   100,0992       petittexte, texte
```

Notons au passage ce qui s'est produit : `nix$dbo indexdemo$petittexte` a été recréé, puisqu'il doit contenir la clé de l'index *clustered*, et que celui-ci a changé. Méfiez-vous de cet effet de bord : la modification d'un index *clustered* entraîne la recréation de tous les autres index, et donc un travail important, augmenté d'un blocage (verrouillage) de la table durant une période souvent longue (voyez plus loin l'option `ONLINE` pour limiter ce temps de blocage).

L'index `nix$dbo_indexdemo$petittexte`, dont la clé était auparavant de 5 octets (1 octet pour la colonne `petittexte`, et 4 octets pour la colonne `ID`), en fait maintenant (environ) 101. L'optimiseur va orienter son choix en regard de cette information. Plus la clé est longue, plus l'index sera lourd à parcourir.

La taille maximale d'une clé d'index est de 900 octets. Nous en sommes loin ici, mais déjà, l'utilité de l'index est compromise. Un index dont la clé fait 900 octets, est très peu utile, et même dangereux si l'index est *clustered*.

Couverture de requête

L'index comporte une ou plusieurs colonnes de la table dans sa clé. Cela veut donc dire que la valeur de ces colonnes est connue de l'index, et que toutes ces valeurs sont présentes dans son nœud feuille. Par exemple, un index sur la colonne `LastName` de la table `Person.Contact` contient tous les `LastName`. Dans ce cas, nous souhaitons dans notre requête ne récupérer que les noms, l'index tout seul suffit à y répondre :

```
SELECT DISTINCT LastName
FROM Person.Contact
WHERE LastName LIKE 'A%'
ORDER BY LastName;
```

Donnera ce plan d'exécution[1] :

```
|--Stream Aggregate(GROUP BY:([LastName]))
    |--Index Seek(OBJECT:([nix$Person_Contact$LastName]),
       SEEK:([LastName] >= N'A' AND [LastName] < N'B'),
       WHERE:([LastName] like N'A%') ORDERED FORWARD)
```

Vous voyez que, non seulement l'index a été utilisé, mais en plus en *seek* (avec une réécriture du `LIKE` en clause acceptable pour un *seek*), et non pas en *scan*, même si l'estimation de la cardinalité retournée est de 911 lignes. Ceci parce qu'il n'y a pas de *bookmark lookup*, donc une stratégie de *seek* est beaucoup plus intéressante.

Cette utilisation de l'index pour retourner les colonnes demandées, est appelée **couverture**. Ici, l'index `nix$Person_Contact$LastName` *couvre* notre requête. Un index couvrant est extrêmement intéressant pour les performances, car l'index est une structure non seulement optimisée pour la recherche, mais plus compacte que la table, puisqu'il contient moins de colonnes : *scanner* le nœud feuille d'un index *non-*

1. Sur les plans affichés en texte, nous avons simplifié le résultat pour améliorer la lisibilité. Ces plans seront plus détaillés chez vous.

clustered est beaucoup plus rapide que de *scanner* la table. De plus, seules les pages d'index seront verrouillées, permettant une plus grande concurrence d'accès à la table.

Que se passe-t-il dès que nous rajoutons une colonne supplémentaire dans le SELECT ?

```
SELECT LastName, FirstName
FROM Person.Contact
WHERE LastName LIKE 'A%'
ORDER BY LastName, FirstName;
```

Voici le plan d'exécution :

```
|--Sort(ORDER BY:([LastName] ASC, [FirstName] ASC))
    |--Clustered Index Scan(OBJECT:([PK_Contact_ContactID]),
       WHERE:([LastName] like N'A%'))
```

Raté ! Il manque à l'index nix$Person_Contact$LastName l'information de FirstName. Pour éviter le *bookmark lookup*, l'optimiseur préfère parcourir la table. Différence de performances ? Un SET STATISTICS IO ON nous donne en lectures, 5 pages pour la première requête, 1135 pour la seconde… Voilà qui est dommage, simplement pour une colonne de plus à retourner. Pour éviter cela, nous allons rendre l'index couvrant. En SQL Server 2000, il n'y avait d'un moyen de le faire : ajouter les colonnes à couvrir en fin de clé de l'index. C'était efficace, mais avec un léger désavantage : les colonnes ajoutées alourdissaient toute la structure de l'index, puisqu'elles faisaient partie de la clé. Depuis SQL Server 2005, nous pouvons utiliser une commande d'**inclusion**, qui ajoute nos colonnes dans le nœud feuille de l'index seulement. Ainsi, tous les nœuds intermédiaires conservent leur compacité. Les colonnes incluses n'ont pas besoin d'y être, puisqu'elles ne servent pas à résoudre la clause de filtre, mais l'affichage dans la clause SELECT. Bien entendu, si vous vouliez permettre la recherche par LastName et FirstName, vous ajouteriez la colonne FirstName dans la clé de l'index, au lieu de la mettre en inclusion. La syntaxe de l'inclusion est la suivante :

```
CREATE INDEX nix$Person_Contact$LastName
ON Person.Contact (LastName) INCLUDE (FirstName)
WITH DROP_EXISTING;
```

Nous avons ajouté l'option DROP_EXISTING pour recréer l'index sans avoir à le supprimer préalablement. Qu'en est-il de notre requête ? Voici son nouveau plan d'exécution :

```
|--Sort(ORDER BY:([LastName] ASC, [FirstName] ASC))
    |--Index Seek(OBJECT:([nix$Person_Contact$LastName]),
       SEEK:([LastName] >= N'A' AND [LastName] < N'B'),
       WHERE:([LastName] like N'A%') ORDERED FORWARD)
```

Le *seek* est revenu. Compte tenu des gains de performance importants de la couverture de requête, n'hésitez pas à pratiquer l'inclusion, même si vous avez plusieurs colonnes. N'exagérez pas non plus, n'oubliez pas que l'index à un coût de mainte-

nance. La mise à jour de colonnes incluses provoque des écritures dans l'index, et potentiellement des *splits* de pages d'index lorsque la colonne est de taille variable.

Index filtré et contrainte d'unicité

Nous le savons, `NULL` n'est pas une valeur, c'est même le contraire d'une valeur : c'est l'absence de valeur, l'inconnu. Comment gérer cet inconnu dans une contrainte d'unicité ? La norme SQL indique en toute logique qu'une contrainte d'unicité doit accepter de multiples marqueurs `NULL`. Dans la prise en charge par SQL de la logique à trois états, le `NULL` est discriminant. Une requête comme :

```
SELECT *
FROM Person.Contact
WHERE Title = 'Mr.'
```

Les lignes dont le `Title` est `NULL` ne sont pas retournées. Ces lignes pourraient correspondre à un `Title = 'Mr.'`, mais nous n'en savons rien, nous ne pouvons donc les considérer. Le même phénomène s'applique à l'unicité. Une colonne contenant le marqueur `NULL` pourrait correspondre à une valeur déjà entrée, mais nous n'en savons rien. Par conséquent, une contrainte d'unicité devrait accepter les `NULL`, et remettre sa vérification au moment où la colonne est réellement alimentée d'une valeur.

SQL Server ne réagit pas ainsi. Un seul marqueur `NULL` est accepté dans une contrainte d'unicité ou dans un index unique, sans doute en partie parce que, pour SQL Server, `NULL` est une clé d'index comme une autre (une recherche `WHERE … IS NULL` utilise l'index comme toute autre comparaison d'égalité). Pour gérer ce problème nous avons, en SQL Server 2005, deux solutions : implémenter la contrainte à l'aide d'un déclencheur, et donc nous passer de l'index unique, ou créer une colonne calculée persistée et indexée, basée sur la colonne à unifier, dans laquelle nous remplaçons les marqueurs `NULL` par une valeur unique, par exemple calculée sur la clé primaire, que nous garantissons différente de toute valeur possible de la colonne source. Cette deuxième solution peut être intéressante dans certains cas, mais elle est un peu difficile à gérer, et elle augmente la taille des données.

> **En SQL Server 2008**, une clause de filtre est introduite à la création d'index. Elle est intéressante pour notre cas, mais aussi pour optimiser des index sur des colonnes dont seules quelques valeurs sont sélectives. Vous pouvez simplement ajouter une clause `WHERE` à la commande de création d'index *nonclustered* :
>
> ```
> CREATE UNIQUE INDEX uqf$Person_Contact$EmailAddress
> ON Person.Contact (EmailAddress)
> WHERE EmailAddress IS NOT NULL;
> ```
>
> Les lignes qui ne correspondent pas à la clause `WHERE` ne sont simplement pas prises en compte dans l'index. De fait, ce filtre nous permet d'implémenter un index unique répondant à la norme SQL. La clause `WHERE` accepte les comparaisons simples, et l'utilisation du `IN`.

Mais la création d'index filtrés n'est pas limitée aux index uniques, vous pouvez l'utiliser sur des colonnes très peu sélectives pour certaines valeurs et très sélectives

pour d'autres (en incluant dans le filtre seulement les valeurs très sélectives), pour diminuer la taille de l'index et le rendre plus efficace. En voici un exemple, qui crée un index filtré sur une colonne de type `bit`, censée représenter un flag actif/inactif :

```
CREATE TABLE dbo.testfiltered (
    id int NOT NULL PRIMARY KEY,
    Active bit NOT NULL,
    fluff char(896) NOT NULL DEFAULT ('b'));

INSERT INTO dbo.testfiltered (id, Active)
SELECT ContactId, 0 FROM AdventureWorks.Person.Contact

UPDATE t1 SET Active = 1
FROM dbo.testfiltered t1
JOIN (SELECT TOP (10) id FROM dbo.testfiltered ORDER BY NEWID()) t2
    ON t1.id = t2.id

CREATE INDEX nix$dbo_testfiltered$active
ON dbo.testfiltered (Active)
WHERE Active = 1;

CREATE INDEX nix$dbo_testfiltered$activeNotFiltered
ON dbo.testfiltered (Active);

DBCC SHOW_STATISTICS ('dbo.testfiltered', 'nix$dbo_testfiltered$active')
DBCC SHOW_STATISTICS ('dbo.testfiltered',
    'nix$dbo_testfiltered$activeNotFiltered')

SELECT *
FROM dbo.testfiltered
WHERE Active = 1;
```

Dans cet exemple, l'index `nix$dbo_testfiltered$active` est choisi par l'optimiseur : il est plus compact que l'index non filtré, et donc plus rapide à parcourir. Nous avons délibérément ajouté une colonne nommée `fluff` pour alourdir la table, donc le nombre de pages qui la contiennent. Nous avons aussi utilisé une astuce pour mettre à jour dix lignes aléatoires, à l'aide d'un `ORDER BY NEWID()` : `NEWID()` retournant un GUID différent à chaque ligne. Cette astuce est coûteuse en performances, elle n'est à utiliser que dans ce type de tests.

6.1.3 Création d'index

Après avoir présenté les différents types d'index, il nous reste à parler simplement de la création de ceux-ci. Nous avons vu dans nos codes d'exemple, la syntaxe de création. Nous pouvons la résumer ainsi (les valeurs soulignées représentent les valeurs par défaut et les instructions en italiques ne sont valables que pour SQL Server 2008) :

```
CREATE [ UNIQUE ] [ CLUSTERED | NONCLUSTERED ] INDEX index_name
    ON <objet> ( colonne [ ASC | DESC ] [ ,...n ] )
    [ INCLUDE ( colonne [ ,...n ] ) ]
```

```
    [ WHERE <filtre> ]
    [ WITH ( <option> [ ,...n ] ) ]
    [ ON { partition_schema ( colonne )
         | filegroup
         | default
         }
    ]
    [ FILESTREAM_ON { filestream_filegroup | partition_schema | "NULL" } ]

<option> ::=
{
    PAD_INDEX = { ON | OFF }
  | FILLFACTOR = fillfactor
  | SORT_IN_TEMPDB = { ON | OFF }
  | DROP_EXISTING = { ON | OFF }
  | ONLINE = { ON | OFF }
  | ALLOW_ROW_LOCKS = { ON | OFF }
  | ALLOW_PAGE_LOCKS = { ON | OFF }
  | MAXDOP = max_degree_of_parallelism
  | DATA_COMPRESSION = { NONE | ROW | PAGE}
     [ ON PARTITIONS ( { <partition_number_expression> | <range> }
     [ , ...n ] ) ]
}
```

Voyons rapidement quelques-unes de ces options.

UNIQUE permet de créer une contrainte sur la ou les colonnes qui composent la clé. Au niveau du moteur, elle est strictement équivalente à la contrainte d'unicité, qui crée elle-même un index unique de ce type.

Pour chaque colonne de la clé, vous pouvez indiquer l'ordre de tri : **ASC** ou **DESC**. Si vous triez principalement en ordre descendant (par exemple les lignes les plus récentes d'abord, à l'aide d'un index sur une date), créez l'index directement en `DESC`, cela rendra le *scan* plus rapide.

Les options **FILLFACTOR** et **PAD_INDEX** déterminent l'espace laissé libre dans les pages d'index. Lorsque vous créez un index sur une table très souvent modifiée, chaque modification entraîne une opération de maintenance de l'index. Nous nous souvenons que l'index maintient chacun de ses niveaux dans l'ordre de la clé, à l'intérieur des pages d'index comme de page en page, à l'aide d'une liste doublement liée. Cette maintenance entraîne des réorganisations dans la page, et entre les pages, lorsque des valeurs de clé sont insérées. Imaginons une armoire de boîtes à fiches, contenant par exemple des dossiers individuels. Lorsque de nouveaux dossiers sont ajoutés, ils peuvent remplir complètement une boîte à fiche. Il faut alors prendre une boîte vide, l'insérer au bon endroit dans l'armoire, et y placer les nouveaux dossiers (ce qu'on appelle un *split* de page). Lorsque des dossiers sont retirés pour suppression, ils libèrent de la place dans une boîte. Ceci rend l'ensemble moins compact : un certain nombre de boîtes ne sont qu'en partie pleines, voire presque vides et le tout prend plus de place dans l'armoire. Chercher un dossier parmi l'ensemble demande plus de travail, car il faudrait ouvrir plus de boîtes. C'est le problème que rencontre SQL Server lors d'un *scan*. On appelle ce phénomène la **fragmentation interne**

(dans les pages). Pour combattre cette situation, il n'y a qu'un moyen, c'est la réorganisation complète, de temps en temps. Réorganiser à chaque petit changement est impraticable : les employés du service passeraient leur temps à tout changer.

Un autre problème apparaît quand l'armoire elle-même n'a pas assez de place pour contenir de nouvelles boîtes sur toutes les étagères. À ce moment, l'ajout d'une nouvelle boîte ne peut se faire qu'au fond de l'armoire. Il faut alors inscrire sur un *post-it* fixé sur la boîte à fiche pleine le numéro de la boîte à fiche qui contient les dossiers suivants, pour qu'on puisse aller la chercher au fond de l'armoire. Dès lors, parcourir les dossiers veut dire passer de boîte en boîte non plus dans l'ordre des étagères, mais avec de nombreux renvois à l'emplacement du fond, et avec à chaque fois un retour à l'endroit quitté sur l'étagère. C'est ce qu'on appelle de la **fragmentation externe** : la liste doublement liée d'un niveau d'index pointe sur des pages qui ne sont pas contiguës, un *scan* doit donc faire de nombreux allers et retours entre les extensions. Un *split* de page pleine entraîne en général à la fois de la fragmentation interne, puisque de nouvelles pages sont créées qui contiennent de l'espace vide, et de la fragmentation externe, puisque ces nouvelles pages sont souvent créées dans d'autres extensions. Une suppression de ligne ne crée que de la fragmentation interne. Notons également que les *splits* ne se produisent pas qu'aux insertions : une mise à jour de ligne, qui augmente la taille d'une colonne de type variable, peut provoquer aussi le déplacement des lignes, soit dans un index *nonclustered* dont il est une partie de la clé, soit dans un index *clustered*, c'est-à-dire la table elle-même[1]. La fragmentation interne est ennuyeuse parce qu'elle fait grossir la table inutilement. La fragmentation externe l'est encore plus, parce qu'elle diminue les performances en lecture physique en provoquant des lectures aléatoires, et qu'elle diminue aussi les performances d'écriture. Les clauses `FILLFACTOR` et `PAD_INDEX` servent à réduire la fragmentation externe.

`FILLFACTOR` permet d'indiquer un pourcentage d'espace vide dans les pages du nœud feuille de l'index, à sa création. Cela permet de conserver suffisamment d'espace libre pour accommoder de nouvelles lignes sans provoquer de *splits*, bien entendu au détriment de la compacité des pages. Sa valeur est 0, qui correspond à 100 %. Pour vos tables très fortement modifiées, vous pouvez attribuer une valeur manuelle. Par exemple, un `FILLFACTOR` de 80 laissera 20 % d'espace libre dans la page. Pour des tables utilisées plutôt en lecture (comme dans des applications OLAP), laissez-le en valeur par défaut, pour obtenir les meilleures performances de vos index en les conservant le plus compacts possible.

La colonne `fill_factor` de la vue système `sys.indexes` vous donne le `FILLFACTOR` défini de vos index :

```
SELECT SCHEMA_NAME(o.schema_id) + '.' + o.name as table_name,
    i.name as index_name,
    i.fill_factor
FROM sys.indexes i
```

1. Les tables *heap* se réorganisent différemment : des renvois d'enregistrements (*forwareded records*) sont créés dans ces cas).

```
JOIN sys.objects o ON i.object_id = o.object_id
WHERE fill_factor <> 0
ORDER BY table_name;
```

Il est évident que le FILLFACTOR n'est appliqué qu'à la création et reconstruction de l'index. Il n'est pas automatiquement maintenu. Si vous réorganisez votre armoire en décidant de ne remplir vos boîtes à fiches qu'à moitié pour laisser de la place pour les futurs dossiers, vous n'allez pas toujours conserver ce remplissage à 50 % à l'ajout de chaque dossier : cela équivaudrait à ne rien changer au problème, en prenant plus de place pour le tout. Si vous voulez maintenir une valeur de FILLFACTOR, il faut donc appliquer un plan de maintenance de vos index, pour les réorganiser ou les reconstruire régulièrement.

Pour cela, vous disposez de deux commandes :

ALTER INDEX... REBUILD reconstruit totalement l'index. Vous pouvez, en édition Entreprise, indiquer une reconstruction ONLINE. Les options de l'index spécifiées à la création (dont le FILLFACTOR) sont conservées. En cas de reconstruction d'un index *clustered*, les index *nonclustered* ne sont pas reconstruits. Pour tout reconstruire en une fois, utilisez cette syntaxe :

```
ALTER INDEX ALL ON Person.Contact REBUILD;
```

Vous pouvez changer au passage les options, comme le FILLFACTOR :

```
ALTER INDEX nix$Person_Contact$LastName
ON Person.Contact
REBUILD WITH (FILLFACTOR = 50);
```

En mode de récupération simple (ou en mode journalisé en bloc), cette opération est loguée de façon minimale, et est donc plus légère.

ALTER INDEX... REORGANIZE réorganise seulement le nœud feuille de l'index. C'est une commande plus rapide, qui est toujours exécutée ONLINE, même en édition Standard. Les pages sont compactées selon la valeur du FILLFACTOR indiquée à la création de l'index. C'est une commande idéale pour traiter rapidement un index peu fragmenté.

Mais, bien sûr, la question est de savoir si un index est fragmenté… La fonction de gestion dynamique sys.dm_db_index_physical_stats est là pour ça. Démonstration :

```
CREATE INDEX nix$Person_Contact$FirstName
ON Person.Contact (FirstName)

SELECT index_id, *
FROM sys.indexes
WHERE name = 'nix$Person_Contact$FirstName'
AND object_id = OBJECT_ID('Person.Contact')

SELECT * FROM sys.dm_db_index_physical_stats
    (DB_ID(N'AdventureWorks'),
     OBJECT_ID(N'Person.Contact'), 22, NULL , 'DETAILED')
```

```
ORDER BY index_level DESC;

ALTER INDEX nix$Person_Contact$FirstName
ON Person.Contact
REBUILD WITH (FILLFACTOR = 20)
-- ou :
CREATE INDEX nix$Person_Contact$FirstName
ON Person.Contact (FirstName)
WITH FILLFACTOR = 20,
     DROP_EXISTING;

SELECT *
FROM sys.dm_db_index_physical_stats
    (DB_ID(N'AdventureWorks'),
    OBJECT_ID(N'Person.Contact'), 22, NULL , 'DETAILED')
ORDER BY index_level DESC;
```

Les colonnes intéressantes retournées par cette fonction sont présentées dans le tableau 6.1.

Tableau 6.1 — Résultat de sys.dm_db_index_physical_stats

Colonne	Signification
index_depth	Profondeur de l'index (nombre de niveaux). 1 au minimum
index_level	Niveau de l'index. En mode `'DETAILED'`, la fonction retourne une ligne par niveau.
avg_fragmentation_in_percent	Fragmentation externe moyenne
fragment_count	Nombre de groupes de pages consécutives au nœud feuille (ou à chaque niveau en vision détaillée). Un index totalement défragmenté devrait avoir seulement un fragment, puisque toutes les pages se suivent. Plus ce nombre est important, plus il y a de fragmentation externe.
avg_fragment_size_in_pages	Indique combien de pages en moyenne comportent les fragments. Plus ce nombre est important, moins il y a de fragmentation externe, car plus il y a de pages consécutives.
page_count	Nombre total de pages sur le niveau
avg_page_space_used_in_percent	Pourcentage moyen de remplissage de la page. Donne la fragmentation interne.
record_count	Nombre d'enregistrements sur le niveau, donc de clés présentes
min_record_size_in_bytes	Taille du plus petit enregistrement

Colonne	Signification
max_record_size_in_bytes	Taille du plus grand enregistrement
avg_record_size_in_bytes	Taille moyenne des enregistrements

Attention aux performances – L'appel de `sys.dm_db_index_physical_stats` en mode `'DETAILED'` oblige SQL Server à parcourir extensivement l'index. Éviter d'appeler la fonction avec ce mode sur tous les index d'une base (paramètres à `NULL`).

Au plus simple, vous pouvez vous baser sur les colonnes `avg_fragmentation_in_percent` et `fragment_count`. La recommandation de Microsoft est de réorganiser l'index lorsque l'`avg_fragmentation_in_percent` est en dessous de 30 %, et de reconstruire en dessus.

L'option `SORT_IN_TEMPDB` force SQL Server à stocker temporairement les résultats intermédiaires de tris opérés pour créer l'index. Vous trouvez plus d'informations sur ces tris dans l'entrée de BOL « *tempdb and Index Creation* ». Ces tris peuvent générer des larges volumes sur des tables importantes, des index *clustered*, des index *nonclustered* composites ou des index *nonclustered* comportant beaucoup de colonnes incluses. Quand l'option `SORT_IN_TEMPDB` est à `OFF` (par défaut), toute l'opération de création se fait dans les fichiers de la base de données qui contient l'index, ce qui provoque beaucoup de lectures et d'écritures sur le même disque, et ralentit les opérations d'entrées/sorties normales d'utilisation de la base. Avec `SORT_IN_TEMPDB` à `ON`, les lectures et écritures sont séparées : lectures sur la base, écritures sur `tempdb`, et ensuite l'inverse. Si `tempdb` est placé sur un disque dédié, cela peut nettement améliorer les performances. De plus, cela offre plus de chances d'une contiguïté des extensions contenant l'index, car l'écriture de la structure de l'index dans le fichier de bases de données se fait en une fois. Ne négligez pas cette option sur des tables à grande volumétrie, elle offre souvent de bons gains de performances à la création des index, quand `tempdb` a son disque dédié.

`DROP_EXISTING` permet de supprimer l'index et de le recréer en une seule commande. Les options de l'index doivent être spécifiées à nouveau – notamment le `FILLFACTOR` –, ils ne sont pas conservés de l'index existant.

La création d'un index pose un verrou sur la table, qui empêche la lecture et l'écriture en cas d'index *clustered* (verrou de modification de schéma), et l'écriture en cas d'index *nonclustered* (verrou partagé). L'option `ONLINE` permet de créer ou recréer un index sans verrouiller la table. SQL Server utilise pour ce faire la fonctionnalité de row versioning. Pendant toute la phase de création d'index, les modifications de données sont possibles, elles seront ensuite automatiquement synchronisées sur le nouvel index. Cette option est valide pour un index *nonclustered* aussi bien que *clustered*. L'opération `ONLINE` est plus lourde, spécialement lorsque les données de la table

sont modifiées, car SQL Server doit maintenir deux structures d'index, et elle utilise activement `tempdb`, mais elle offre l'avantage d'un meilleur accès concurrentiel aux tables. Cette option n'est disponible qu'en édition Entreprise. Pour plus de détail sur son fonctionnement interne, reportez-vous aux BOL, entrée « *How Online Index Operations Work* ».

`ALLOW_ROW_LOCKS` et **`ALLOW_PAGE_LOCKS`** contrôlent le verrouillage sur l'index (donc sur la table dans le cas d'un index *clustered*). Par défaut, les verrous sont posés sur des clés d'index, et escaladés au besoin. Les verrous sur les clés permettent une meilleure concurrence d'accès, mais peuvent être plus coûteux à gérer si beaucoup de lignes sont appelées. `ALLOW_ROW_LOCKS` à `OFF` désactive la pose de verrous au niveau clé (ligne), et donc force les verrous sur les pages. Couplé avec `ALLOW_PAGE_LOCKS` à `OFF`, cela force les verrous sur la table. En général il n'y a rien à changer dans ces options. Elles peuvent éventuellement faire gagner du temps sur des requêtes larges de type OLAP, mais dans ces requêtes, le passage en niveau d'isolation `READ UNCOMMITTED` est plus efficace. Pour plus de précisions sur le contrôle de la granularité de verrouillage, reportez-vous à la section 7.2.2.

L'option **`MAXDOP`** spécifie le degré de parallélisme à appliquer à l'opération de création de l'index. Le parallélisme pour un `CREATE INDEX` n'est pris en charge que par l'édition Entreprise.

Vous pouvez créer des index composites, c'est-à-dire des index multicolonnes. Ils sont utiles pour résoudre des clauses de filtres qui utilisent toujours les mêmes colonnes (par exemple une recherche systématique avec date de début et date de fin). Dans ce cas, l'ordre de déclaration des colonnes dans la clé est important. Placez la colonne sur laquelle vous cherchez le plus souvent, en premier. Ainsi l'index peut être utile également pour les recherches sur cette colonne uniquement. Vous pouvez vous représenter la clé d'un index comme la concaténation de toutes les colonnes qui la composent (plus la clé de l'index *clustered* à la fin). SQL Server pourra utiliser cet index pour une recherche sur la première colonne, sur la première et la deuxième, etc. Une recherche exclusivement sur la deuxième colonne ne pourra profiter de l'index, car on ne peut le parcourir qu'en le prenant au début de la clé. C'est tout à fait comme un annuaire : vous pouvez chercher par nom, ou par nom et prénom, puisqu'il est organisé par ordre alphabétique de noms de famille. Chercher les personnes ayant un même prénom dans un annuaire, est impossible. Vous ne pouvez que parcourir toutes les pages.

La clé d'un index est limitée à 16 colonnes, et 900 octets. N'en arrivez pas jusque-là… Cette limite n'est pas applicable aux colonnes incluses.

Si les colonnes de l'index composite sont cherchées toujours ensembles, placez les colonnes les plus sélectives (contenant le plus de valeurs uniques) en premier, cela augmentera la sélectivité générale de l'index, et le rendra plus utile, et plus intéressant pour l'optimiseur. Faites ici encore la comparaison avec un annuaire : il est organisé par nom et prénom, non par prénom et nom, aussi parce qu'il est plus rapide, pour trouver Pierre Bourdieu, de chercher Bourdieu, et dans ceux-ci, de trouver Pierre, que l'inverse.

6.1.4 Optimisation de la taille de l'index

Les index sont stockés dans des pages, tout comme les données. L'arbre *B-Tree* possède une profondeur et une étendue. Le principe de l'index est de limiter l'étendue par la profondeur : un niveau d'index trop large devient inefficace, il faut alors placer un niveau supplémentaire pour en diminuer la surface. Corollairement, plus la clé de l'index est petite, plus le nombre de clé par page est important, moins l'index doit être profond, et plus son parcours est optimisé.

Prenons un exemple pratique. Voici le code, nous le détaillerons ensuite :

```
CREATE DATABASE testdb
GO
ALTER DATABASE testdb SET RECOVERY SIMPLE
GO

USE testdb

CREATE TABLE dbo.testIndex (
    codeLong char(900) NOT NULL PRIMARY KEY NONCLUSTERED,
    codeCourt smallint NOT NULL,
    texte char(7150) NOT NULL DEFAULT ('O')
)
GO

DECLARE @v int
SET @v = 1
WHILE (@v <= 8000) BEGIN
    INSERT INTO dbo.testIndex (codeLong, codeCourt)
    SELECT CAST(@v as char(10)), @v

    SET @v = @v + 1
END
GO

CREATE UNIQUE INDEX uq$testIndex$codeCourt ON dbo.testIndex (codeCourt)

SELECT name, index_id, type_desc
FROM sys.indexes
WHERE object_id = OBJECT_ID('dbo.testIndex')

/* -- résultat
name                      index_id    type_desc
------------------------- ----------- ------------
NULL                      0           HEAP
PK__testIndex__0425A276   2           NONCLUSTERED
uq$testIndex$codeCourt    3           NONCLUSTERED

*/

SELECT index_depth, index_level, page_count, record_count
FROM sys.dm_db_index_physical_stats(
     DB_ID(), OBJECT_ID('dbo.testIndex'), 2, NULL, 'DETAILED')
```

```
/* -- résultat
index_depth index_level page_count           record_count
----------- ----------- -------------------- --------------------
6           0           1441                 8000
6           1           321                  1441
6           2           72                   321
6           3           16                   72
6           4           3                    16
6           5           1                    3
*/

SELECT index_depth, index_level, page_count, record_count
FROM sys.dm_db_index_physical_stats(
     DB_ID(), OBJECT_ID('dbo.testIndex'), 3, NULL, 'DETAILED')

/* -- résultat
index_depth index_level page_count           record_count
----------- ----------- -------------------- --------------------
2           0           14                   8000
2           1           1                    14
*/
```

Nous créons d'abord une base de données pour y jouer, que nous mettons en mode de récupération simple afin d'éviter une augmentation inutile de la taille du journal. Nous créons ensuite une table de test, en faisant en sorte que chaque ligne remplisse une page. Nous y ajoutons deux colonnes pour y appliquer des index : l'une fait 900 octets (la taille maximale d'une clé d'index), l'autre un simple `smallint` (2 octets). Nous indexons ces deux colonnes. L'une est clé primaire, l'autre clé candidate. Le fait que ces index soient uniques n'a pas d'importance dans cet exemple, la structure de l'index n'est en rien différente.

Nous créons une insertion en boucle, de façon à y insérer 8000 lignes. Nous utilisons `sys.indexes` pour retrouver l'identifiant de chaque index. L'index lourd porte l'`ID` 2, le léger, l'`ID` 3. Nous en observons ensuite la taille à l'aide de la fonction système `sys.dm_db_index_physical_stats`. Ce que nous constatons, c'est que l'index lourd est profond, et nécessairement étendu (peu de clés peuvent résider dans la même page) : six niveaux pour un total de 1 854 pages. L'index léger, lui, comporte deux niveaux pour un total de 15 pages. Observons la différence sur les requêtes :

```
SET STATISTICS IO ON

SELECT * FROM dbo.testIndex WHERE CodeLong = '49'
-- Table 'testIndex'. Scan count 0, logical reads 7
SELECT * FROM dbo.testIndex WHERE CodeCourt = 49
-- Table 'testIndex'. Scan count 0, logical reads 3
```

La recherche avec l'index léger doit lire trois pages, contre sept pages pour celle qui s'applique à l'index lourd.

Structure interne de l'index

Nous avons vu dans le chapitre traitant des structures de stockage comment SQL Server répartit les données et les index dans des extensions et des pages. Profitons de cette connaissance, et de l'existence de l'instruction non documentée `DBCC PAGE`, pour suivre à la trace l'utilisation d'un index. Reprenons pour cela la table exemple précédente, et voyons comment le moteur de stockage cherche à l'intérieur de l'index.

Pour cela, nous devons utiliser une autre commande DBCC non documentée : `DBCC IND`, qui donne la structure interne d'un index : chaque page de l'index est retourné, avec son ID, son niveau, et la référence de la liste doublement liée : page précédente et page suivante du même niveau. Sa syntaxe est :

```
DBCC IND ('base de données', 'table', PartitionId);
```

Donc, imaginons que nous soyons ce moteur de stockage. L'optimiseur SQL a décidé d'une stratégie pour répondre à cette requête :

```
SELECT * FROM dbo.testIndex WHERE CodeLong = '49'
```

L'index dont la clé est la colonne `CodeLong`, et dont l'`ID` est 2, va naturellement être utilisé. Nous pouvons le vérifier avec l'option de session `SET SHOWPLAN_TEXT ON`, qui affiche le plan d'exécution en texte, au lieu d'exécuter l'instruction. Le résultat appliqué à notre requête donne ceci :

```
  |--Nested Loops(Inner Join, OUTER REFERENCES:([Bmk1000]))
       |--Index Seek(
         OBJECT:([testdb].[dbo].[testIndex].[PK__testIndex__0425A276]),
         SEEK:([testdb].[dbo].[testIndex].[codeLong]=[@1])
         ORDERED FORWARD)
       |--RID Lookup(OBJECT:([testdb].[dbo].[testIndex]),
         SEEK:([Bmk1000]=[Bmk1000]) LOOKUP ORDERED FORWARD)
```

L'index `PK__testIndex...` est utilisé pour la recherche, et ensuite un RID lookup est effectué, pour rechercher la ligne référencée dans l'index *nonclustered* par un RID (*Row ID*). Nous avons vu dans les statistiques IO, que cela nécessitait la lecture de sept pages. On se souvient que l'index sur `CodeLong` a six niveaux. Tout ici est donc logique : descente sur six niveaux d'index, plus lecture de la page de données référencée par le RID. Essayons de faire ce parcours nous-mêmes. Le code suivant alterne les requêtes de `DBCC PAGE` et leur résultat :

```
CREATE TABLE #ind (
    PageFID bigint,
    PagePID bigint,
    IAMFID bigint,
    IAMPID bigint,
    ObjectID bigint,
    IndexID bigint,
```

```
    PartitionNumber bigint,
    PartitionID bigint,
    iam_chain_type varchar(20),
    PageType int,
    IndexLevel int,
    NextPageFID bigint,
    NextPagePID bigint,
    PrevPageFID bigint,
    PrevPagePID bigint
)
GO

INSERT INTO #ind
EXEC ('DBCC IND(''testdb'', ''dbo.testIndex'', 2)')

SELECT * FROM #ind WHERE IndexLevel = 5

-- comment résoudre cette requête avec un index
SET STATISTICS IO ON
SELECT * FROM dbo.testIndex WHERE code = '49'
/*
Table 'testIndex'. Scan count 0, logical reads 7
*/

DBCC TRACEON (3604);
GO

DBCC PAGE (testdb, 1, 8238, 3)
/*
FileId PageId      Row    Level  ChildFileId ChildPageId codeLong (key)
------ ----------- ------ ------ ----------- ----------- ----------
1      8238        0      5      1           4179        NULL
1      8238        1      5      1           8239        253
1      8238        2      5      1           1122        45
*/
-- row, key 45, ChildPageId : 1122
DBCC PAGE (testdb, 1, 1122, 3)
/*
FileId PageId      Row    Level  ChildFileId ChildPageId codeLong (key)
------ ----------- ------ ------ ----------- ----------- ----------
1      1122        0      4      1           4180        45
1      1122        1      4      1           9582        487
1      1122        2      4      1           9121        55
[...]
*/
-- row, key 487, ChildPageId : 9582
DBCC PAGE (testdb, 1, 9582, 3)
/*
FileId PageId      Row    Level  ChildFileId ChildPageId codeLong (key)
------ ----------- ------ ------ ----------- ----------- ----------
1      9582        0      3      1           9122        487
1      9582        1      3      1           9207        495
1      9582        2      3      1           9368        504
[...]
```

```
*/
-- row, key 487, ChildPageId : 9122
DBCC PAGE (testdb, 1, 9122, 3)
/*
FileId PageId      Row    Level  ChildFileId ChildPageId codeLong (key)
------ ----------- ------ ------ ----------- ----------- ----------
1      9122        0      2      1           9039        487
1      9122        1      2      1           9076        489
1      9122        2      2      1           9123        491
1      9122        3      2      1           9160        493
*/
-- row, key 489, ChildPageId : 9076
DBCC PAGE (testdb, 1, 9076, 3)
/*
FileId PageId      Row    Level  ChildFileId ChildPageId codeLong (key)
------ ----------- ------ ------ ----------- ----------- ----------
1      9076        0      1      1           9073        489
1      9076        1      1      1           9075        4897
1      9076        2      1      1           3623        49
1      9076        3      1      1           9079        4906
*/
-- row, key 49, ChildPageId : 3623
DBCC PAGE (testdb, 1, 3623, 3)
/*
FileId PageId      Row    Level  codeLong (key) HEAP RID
------ ----------- ------ ------ ----------------------------------
1      3623        0      0      49             0x380C000001000000
1      3623        1      0      490            0x590E000001000000
1      3623        2      0      4900           0x8C23000001000000
[...]
*/
```

Nous créons d'abord une table temporaire pour stocker le résultat de `DBCC IND`. Nous cherchons ensuite dans cette table l'ID de la page d'entrée de l'index : son nœud racine. Il s'agit de la page qui est au niveau le plus élevé. Nous regardons ensuite son contenu avec `DBCC PAGE`. Cela nous retourne une structure qui comporte notamment les colonnes `ChildPageId` et `codeLong (Key)`, indiquant respectivement l'ID de la page fille (celle qui se trouve au nœud intermédiaire suivant, c'est-à-dire au niveau inférieur de l'index), et la clé qui représente la ligne. Chaque entrée d'index représente une plage de valeurs de clés située entre la clé indiquée dans la ligne et la clé de la ligne suivante de ce nœud. Par exemple ici, la ligne 2 du nœud 5 de l'index, pointe sur des valeurs de clé comprises entre `'253'` incluse et `'45'` exclue (n'oublions pas que cette colonne est un `char`, pas un numérique, et que la clé est donc triée alphabétiquement). La ligne 3 pointe sur toutes les clés égales ou supérieures à `'45'`. C'est là que nous devons descendre. Nous suivons donc cette voie sur six niveaux, et nous voyons que chaque page référencée dans `ChildPageId` est une page située au niveau inférieur. Lorsque nous arrivons au nœud feuille, le niveau 0, le `ChildPageId` est remplacé par un `HEAP RID`, qui est un stockage binaire du numéro de fichier, de page, et de ligne dans la page, de la ligne qui correspond à cette valeur. Dans le cas d'un index sur une table *clustered*, nous aurions trouvé la clé de l'index *clustered*.

6.2 VUES DE GESTION DYNAMIQUE POUR LE MAINTIEN DES INDEX

Nous l'avons vu, un index créé n'est pas forcément utilisé. Notamment, l'optimiseur peut choisir de parcourir la table au lieu d'utiliser l'index, si celui-ci est peu sélectif. Tout index possède un coût de maintenance, c'est-à-dire qu'il doit répercuter en temps réel toutes les modifications apportées à la table sur les colonnes qui composent sa clé. Un index inutilisé est donc pénalisant pour les performances, ainsi bien sûr que pour l'espace de stockage : il consomme de l'espace disque sans aucune utilité. Il faut donc le supprimer. Mais comment savoir si un index est utilisé ou non ? Vous avez à disposition une vue dynamique qui vous offre toutes les informations nécessaires. Cette vue s'appelle `sys.dm db index usage_stats`. Elle affiche les informations suivantes du tableau 6.1.

Tableau 6.2 — Résultat de sys.dm_db_index_usage_stats

Colonne	Signification
user_seeks	nombre de recherches à travers l'index dues à une requête utilisateur
user_scans	nombre de parcours du nœud feuille de l'index dus à une requête utilisateur
user_lookups	nombre de recherches de clés (*bookmark lookups*) dues à une requête utilisateur
user_updates	nombre de mises à jour de l'index dues à une requête DML (INSERT, UPDATE, DELETE) utilisateur
last_user_seek	dernière recherche à travers l'index due à une requête utilisateur
last_user_scan	dernier parcours du nœud feuille de l'index dû à une requête utilisateur
last_user_lookup	dernière recherche de clés (*bookmark lookups*) due à une requête utilisateur
last_user_update	dernière mise à jour de l'index due à une requête DML (INSERT, UPDATE, DELETE) utilisateur
system_seeks	nombre de recherches à travers l'index dues à une requête système
system_scans	nombre de parcours du nœud feuille de l'index dus à une requête interne
system_lookups	nombre de recherches de clés (*bookmark lookups*) dues à une requête interne
system_updates	nombre de mises à jour de l'index dues à une requête DML (INSERT, UPDATE, DELETE) interne
last_system_seek	dernière recherche à travers l'index due à une requête système

Colonne	Signification
last_system_scan	dernier parcours du nœud feuille de l'index dû à une requête système
last_system_lookup	dernière recherche de clés (*bookmark lookups*) due à une requête système
last_system_update	dernière mise à jour de l'index due à une requête DML (INSERT, UPDATE, DELETE) système

Si un *scan* s'exécute sur une table, c'est soit qu'une recherche dans une clause `WHERE` ne peut être effectuée à travers un index (que l'index soit trop peu sélectif, soit qu'il n'y a pas de bon index pour la clause de filtre). On peut donc utiliser le nombre de *scan* pour détecter un manque d'index ou un index trop peu sélectif.

6.2.1 Obtention des informations opérationnelles de l'index

La fonction dynamique `sys.dm_db_index_operational_stats` permet d'obtenir des **informations** sur la vie et les opérations de maintenance d'un index. Il s'agit d'une fonction, qu'il faut donc appeler en passant des paramètres : nom de la base, nom de la table, nom de l'index, nom de la partition. Des paramètres passés à `NULL` permettent de retourner des lignes pour chaque index de la base/table. Vous pouvez ainsi mesurer si l'index est plus ou moins coûteux à maintenir, s'il doit opérer de nombreux *splits* de pages pour accommoder de nouvelles lignes et quels sont les temps d'attentes sur l'index.

Tableau 6.3 — Résultat de sys.dm_db_index_operational_stats

Colonne	Signification
partition_number	numéro de partition si la table est partitionnée. Base 1
leaf_insert_count	Nombre total d'insertions au niveau feuille de l'index
leaf_delete_count	Nombre total de suppressions au niveau feuille de l'index
leaf_update_count	Nombre total de mises à jour au niveau feuille de l'index
leaf_ghost_count	Nombre total de lignes au niveau feuille marquées pour suppression, mais pas encore supprimées. Elles seront supprimées par un thread de nettoyage qui s'exécute à intervalles réguliers.
nonleaf_insert_count	Nombre total d'insertions aux niveaux intermédiaires de l'index. Valeur 0 sur une ligne correspondant à une table *heap*

Colonne	Signification
nonleaf_delete_count	Nombre total de suppressions aux niveaux intermédiaires de l'index. Valeur 0 sur une ligne correspondant à une table *heap*
nonleaf_update_count	Nombre total de mises à jour aux niveaux intermédiaires de l'index. Valeur 0 sur une ligne correspondant à une table *heap*
leaf_allocation_count	Nombre total d'allocations de page sur le niveau feuille d'un index ou un *heap*. Sur un index, une allocation de page correspond à un *page split*
nonleaf_allocation_count	Nombre total d'allocations de page causées par un *split*, sur les niveaux intermédiaires de l'index. Valeur 0 sur une ligne correspondant à une table *heap*
leaf_page_merge_count	Nombre total de fusions de page sur le niveau feuille
nonleaf_page_merge_count	Nombre total de fusions de page sur les niveaux intermédiaires. Valeur 0 sur une ligne correspondant à une table *heap*
range_scan_count	Nombre total de *scans* (de table ou d'une plage plus petite)
singleton_lookup_count	Nombre total de récupérations de lignes distinctes depuis l'index ou la table *heap*
forwarded_fetch_count	Nombre de lignes récupérées à travers un enregistrement de renvoi (*forwarding record*). Valeurs existant seulement sur une table *heap*. Donc toujours 0 sur des index
lob_fetch_in_pages	Nombre total de pages d'objets larges (*large object* (LOB)) récupérées, depuis des unités d'allocation LOB_DATA. Signifie la présence de colonnes TEXT, IMAGE, ou VARCHAR(MAX) ou VARBINARY(MAX), ou XML
lob_fetch_in_bytes	Nombre total d'octets d'objets larges récupérés
lob_orphan_create_count	Nombre total de valeurs d'objets larges orphelines crées pour des opérations en lot. Valeur toujours à 0 sur les index *nonclustered*
lob_orphan_insert_count	Nombre total de valeurs d'objets larges orphelines insérées durant les opérations en lot. Valeur toujours à 0 sur les index *nonclustered*
row_overflow_fetch_in_pages	Nombre total de pages de dépassement de page (*row-overflow*) retournées depuis une unité d'allocation ROW_OVERFLOW_DATA

Colonne	Signification
row_overflow_fetch_in_bytes	Nombre total d'octets de dépassement de page (*row-overflow*) retournés depuis une unité d'allocation ROW_OVERFLOW_DATA
column_value_push_off_row_count	Nombre total de valeurs de colonne poussées dans une autre page, pour accommoder soit des valeurs LOB, soit des valeurs en dépassement de page (*row-overflow*), durant une insertion ou une mise à jour
column_value_pull_in_row_count	Nombre total de lignes réintégrées dans une page de données (unité d'allocation IN_ROW_DATA), depuis des pages de LOB et de dépassement (unité d'allocation LOB_DATA ou ROW_OVERFLOW_DATA), lorsqu'une mise à jour de données a diminué la taille de la ligne
row_lock_count	Nombre total de verrouillages de granularité ligne demandés
row_lock_wait_count	Nombre total d'attentes de libération d'un verrou de ligne
row_lock_wait_in_ms	Temps total d'attente de libération de verrous de ligne, en millisecondes
page_lock_count	Nombre total de verrouillages de granularité page demandés
page_lock_wait_count	Nombre total d'attentes de libération d'un verrou de page
page_lock_wait_in_ms	Temps total d'attente de libération de verrous de page, en millisecondes
index_lock_promotion_attempt_count	Nombre total de tentative d'escalades de verrous
index_lock_promotion_count	Nombre total d'escalades de verrous réellement effectués
page_latch_wait_count	Nombre total d'attentes de libération d'un *latch* (*latch contention*)
page_latch_wait_in_ms	Temps total d'attente de libération de *latch*, en millisecondes
page_io_latch_wait_count	Nombre total d'attentes de libération d'un *latch* d'entrée/sortie de page
page_io_latch_wait_in_ms	Temps total d'attente de libération de *latch* d'entrée/sortie de page, en millisecondes

Pour comprendre ce tableau, nous devons expliquer quelques principes supplémentaires :

Lorsqu'une ligne dans une table *heap* est mise à jour, que la taille de la ligne grandit et qu'il ne reste plus de place dans la page, la ligne est déplacée dans une autre page. Pour éviter une mise à jour de tous les index *nonclustered* pour accommoder le nouveau RID de la ligne, SQL Server pose, à la place de l'ancienne ligne, une référence vers la nouvelle position de ligne, ce qu'on appelle un enregistrement de renvoi (*forwarding record*).

Lors d'une opération de traitement par lot (*bulk operation*), SQL Server crée des LOB pour stocker temporairement les données. Ces LOB sont dits orphelins (*orphan LOB*).

Vous pouvez donc utiliser cette fonction système pour collecter des statistiques physiques, d'utilisation, de charge ou de contention sur vos tables et vos index. Exemple :

```
SELECT object_name(s.object_id) as tbl,
       i.name as idx,
       range_scan_count + singleton_lookup_count as [pages lues],
       leaf_insert_count+leaf_update_count+ leaf_delete_count
         as [écritures sur nœud feuille],
       leaf_allocation_count as [page splits sur nœud feuille],
       nonleaf_insert_count + nonleaf_update_count +
         nonleaf_delete_count as [écritures sur nœuds intermédiaires],
       nonleaf_allocation_count
         as [page splits sur nœuds intermédiaires]
FROM sys.dm_db_index_operational_stats (DB_ID(),NULL,NULL,NULL) s
JOIN sys.indexes i
       ON i.object_id = s.object_id and i.index_id = s.index_id
WHERE objectproperty(s.object_id,'IsUserTable') = 1
ORDER BY [pages lues] DESC
```

6.2.2 Index manquants

Lors de la génération du plan d'exécution de chaque requête (phase d'optimisation), le moteur relationnel teste différents plans d'exécution et sélectionne le moins coûteux. Dans certains cas, ce moteur d'optimisation a la capacité de constater que la requête aurait été bien mieux servie si un index avait été présent. Cette information est retournée lorsqu'on affiche un plan d'exécution détaillé en XML.

Les informations d'**index manquants** sont également stockées dans un espace de cache, et peuvent être requêtées à travers trois vues système, qui offrent toutes les informations nécessaires pour la création de l'index : colonnes qui devraient composer la clé, colonnes à inclure dans le nœud feuille, avec un compteur qui indique le nombre de fois où l'index aurait été utile.

Cette information n'est pas exhaustive, dans le sens ou l'optimiseur ne va pas détecter tous les index manquants. Il ne le fait que sur certaines requêtes, lorsqu'il a la capacité de comprendre qu'un index aurait été utile. Cela ne vous décharge pas du travail d'optimiser les autres requêtes par la création d'index.

Mais, concernant les vues dynamiques d'index manquants, voici une requête qui vous donne toutes les informations :

```
SELECT object_name(object_id) as objet, d.*, s.*
FROM sys.dm_db_missing_index_details d
INNER JOIN sys.dm_db_missing_index_groups g
    ON d.index_handle = g.index_handle
INNER JOIN sys.dm_db_missing_index_group_stats s
    ON g.index_group_handle = s.group_handle
WHERE database_id = db_id()
ORDER BY s.user_seeks DESC, object_id
```

Comme elle vous dit tout, vous pouvez même faire générer par une requête le code DDL nécessaire à la création des index, ainsi, vous n'avez plus qu'à copier ce code et à l'exécuter.

Toutefois, ne créez pas forcément tous les index conseillés, mais concentrez-vous sur ceux dont le nombre de « *user seeks* » est le plus importants : cette mesure vous indique combien de fois l'index aurait pu être utile.

Voici un exemple de code pour générer vos index :

```
SELECT
    'CREATE INDEX nix$' + lower(object_name(object_id)) + '$'
    + REPLACE(REPLACE(REPLACE(COALESCE(equality_columns,
      inequality_columns), ']', ''), '[', ''), ', ', '_')
    + ' ON ' + statement + ' (' + COALESCE(equality_columns,
      inequality_columns) + ') INCLUDE (' + included_columns + ')',
    object_name(object_id) as objet, d.*, s.*
FROM sys.dm_db_missing_index_details d
JOIN sys.dm_db_missing_index_groups g
    ON d.index_handle = g.index_handle
JOIN sys.dm_db_missing_index_group_stats s
    ON g.index_group_handle = s.group_handle
WHERE database_id = db_id()
ORDER BY s.user_seeks DESC, objet
```

Nous verrons dans le chapitre 8 comment lire un plan d'exécution, ce qui vous aidera à déterminer quels index construire pour vos requêtes.

6.3 VUES INDEXÉES

Une **vue** est une requête `SELECT` stockée sur le serveur. On la référence comme une table dans une requête. Elle ne matérialise pas le résultat du `SELECT`. Donc, chaque appel à la vue génère des appels aux tables sous-jacentes. Vous pouvez indexer vos vues pour matérialiser les données. Lorsqu'une vue est indexée, elle maintient le résultat de son `SELECT` dans un index *clustered*, qui doit lui aussi être maintenu à chaque modification les lignes dans les tables sous-jacentes. Cette option est très intéressante pour les larges volumes, sur des requêtes complexes qui mettent en jeu de nombreuses jointures de tables. Elles permettent une sorte de dénormalisation

contrôlé par le système. Dans les cas de volumes plus petits, en général des index bien placés sur les tables sous-jacentes suffisent.

À partir de SQL Server 2005, les vues indexées sont prises en charge par l'édition Standard. Une fonctionnalité supplémentaire est présente dans l'édition Entreprise : l'optimiseur de requête peut choisir d'utiliser un index sur une vue, même si la requête ne mentionne pas explicitement la vue, mais simplement une table sous-jacente et que l'optimiseur trouve un intérêt supérieur à utiliser cet index.

Votre index doit être créé avec certaines options de session qui garantissent la consistance des données de la vue :

- ANSI_NULLS ON (aussi à la création de la table) ;
- QUOTED IDENTIFIER ON, et quelques autres options (voir les BOL) ;
- La vue doit avoir été créée avec l'option `WITH SCHEMABINDING` (ce qui la lie à la structure des tables sous-jacentes, de sorte que les modifications de structure de celles-ci sont protégées) ;
- Les indicateurs de table sont interdits, on ne peut donc forcer un index, ou un niveau d'isolation, dans la requête ;
- Les fonctions référencées dans la vue doivent être déterministes (elles doivent toujours retourner la même valeur, à données égales).

Ces options forcent simplement le retour de la vue à être consistant, donc indexable. Les options telles que `SET ANSI_NULLS` doivent être placées pour toutes les sessions qui requêtent la vue, ou modifient les données sous-jacentes, afin que SQL Server puisse garantir que le résultat du `SELECT` de la vue est toujours le même. Cette condition peut être contraignante dans un environnement où vous ne maîtrisez pas tous les points d'entrées, et notamment avec des applications qui utilisent des bibliothèques d'accès aux données anciennes, comme ODBC.

> Si vous utilisez PHP sous Windows, Microsoft a développé un pilote spécifique, téléchargeable à partir de ce lien : http://www.microsoft.com/sql/technologies/php/.

La première étape est de créer un index *clustered* unique sur la vue, qui va la matérialiser. L'unicité est nécessaire, vous devez donc trouver un moyen de retourner une colonne ou un jeu de colonnes unique dans votre vue.

Voici un exemple de création de vue, et d'utilisation par l'optimiseur, même en mentionnant seulement la table sous-jacente (nous sommes en édition Developer, qui correspond aux fonctionnalités de l'édition Entreprise) :

```
CREATE VIEW Person.SomeContacts
WITH SCHEMABINDING
AS
    SELECT ContactID, FirstName, LastName
    FROM Person.Contact
    WHERE LastName LIKE '%Ad%'
```

```
GO

CREATE UNIQUE CLUSTERED INDEX cix$Person_SomeContacts
ON Person.SomeContacts (ContactID)

CREATE NONCLUSTERED INDEX nix$Person_SomeContacts$LastName_FirstName
ON Person.SomeContacts (LastName, FirstName)
GO

SELECT FirstName, LastName
FROM Person.Contact
WHERE LastName LIKE '%Ad%'
```

Le plan d'exécution généré est simple, un *scan*... mais de quel index ? Voici le plan :

```
  |--Index Scan(OBJECT:([AdventureWorks].[Person].[SomeContacts].
     [nix$Person_SomeContacts$LastName_FirstName]))
```

6.4 STATISTIQUES

Vous avez fait la connaissance sur un *chat* Internet d'une jeune fille qui vous semble charmante et qui vit à Courtrai, en Belgique flamande. Elle semble apprécier vos grandes qualités de cœur et vous convenez d'un premier rendez-vous, afin de vous connaître en vrai. Comme vous êtes galant, vous proposez de faire le voyage à Courtrai. Elle vous donne rendez-vous au café De Brouwzaele.

Si vous n'avez jamais entendu parler de Courtrai, vous ne savez probablement rien de la taille de la ville, de la complexité de ses rues et donc vous ne pouvez pas juger, avant de vous y rendre, si vous trouverez le café De Brouwzaele facilement.

Logiquement, votre premier réflexe sera de vous renseigner sur le nombre d'habitants. Si Courtrai est une ville de quelques milliers d'âmes, vous pouvez vous dire qu'elle est peu étendue et que le café De Brouwzaele sera facile à trouver. Par contre, en cherchant sur Wikipédia, vous vous apercevez que Courtrai compte quelque 73 000 habitants. Sachant cela, vous vous attendez à plus de difficultés. Vous allez donc demander l'adresse du café De Brouwzaele. Kapucijnestraat 19… bien, mais, combien y a-t-il de rues à Courtrai, et où se trouve Kapucijnestraat par rapport à la gare ? La connaissance de ces détails risque d'influencer fortement votre méthode de déplacement.

Pour SQL Server, la problématique est exactement la même. Imaginons que nous envoyons cette requête :

```
SELECT *
FROM Production.TransactionHistory
WHERE ProductId = 800
```

La première chose que sait SQL Server, c'est qu'il y a 113 443 lignes dans la table `Production.TransactionHistory` (dans ce cas, il maintient un compte de lignes dans

la définition des index). Ensuite, il sait qu'il dispose d'un index sur la colonne `ProductId`, qui va lui permettre de filtrer rapidement les enregistrements où `ProductId = 800` :

```
SELECT *
FROM sys.indexes i
WHERE OBJECT_NAME(i.object_id)
```

L'index s'appelle `IX_TransactionHistory_ProductId`.

L'optimiseur de SQL Server se base sur l'évaluation du coût d'un plan d'exécution afin de déterminer la meilleure stratégie. C'est ce qu'on appelle une **optimisation basée sur le coût** (*cost-based optimization*).

Mais pour évaluer le coût d'un plan d'exécution, SQL Server ne peut se contenter des seules informations que nous venons de voir. Il doit disposer d'une estimation du nombre de lignes qui va être retourné, pour juger au plus juste du coût de celle-ci. C'est là où interviennent les statistiques.

Si SQL Server dispose de statistiques de distribution des valeurs contenues dans une colonne, il peut évaluer avec une bonne précision le nombre de lignes concernées lorsqu'une requête filtre sur ces valeurs, car il sait à l'avance combien de lignes environ correspondent au critère.

Les statistiques sont donc un échantillonnage des données. Elles sont créées et maintenues automatiquement par SQL Server sur la clé de tout index créé. Elles sont visibles dans les vues `sys.stats` et `sys.stats_columns`.

Voyons les statistiques de l'index sur `ProductId` :

```
SELECT c.name as ColumnName, s.name as IndexName
FROM sys.stats s
JOIN sys.stats_columns sc ON s.object_id = sc.object_id
                          AND s.stats_id  = sc.stats_id
JOIN sys.columns c ON s.object_id = c.object_id
                   AND sc.column_id = c.column_id
WHERE s.name = 'IX_TransactionHistory_ProductId'
```

Tableau 6.4 — Nom de l'index

ColumnName	IndexName
ProductID	IX_TransactionHistory_ProductID

Cela montre qu'il existe des statistiques dans cet index pour la colonne `ProductId`.

Elles sont en quelque sorte les informations vitales de l'index, ou son diplôme : elles permettent à SQL Server, dans sa stratégie, d'estimer le coût d'utilisation de l'index. En d'autres termes, grâce aux statistiques, SQL Server va pouvoir connaître à l'avance, approximativement, le résultat d'un :

```
SELECT COUNT(*)
FROM Production.TransactionHistory
WHERE ProductId = 800
```

Si le ratio entre le nombre de lignes qui répond au critère recherché et le nombre total de ligne est faible, SQL Server va choisir d'utiliser l'index. Si par contre il est important, et qu'une bonne partie des lignes de la table doit être retournée par la requête, SQL Server va certainement faire le choix de parcourir la table plutôt que d'utiliser l'index, parce qu'il sera moins coûteux de procéder de cette façon que de résoudre ligne par ligne les adresses de pointeurs contenues dans le dernier niveau de l'index.

6.4.1 Statistiques sur les index

Dans le cas qui nous occupe, `ProductId = 800` correspond à 416 lignes / 133 443. Voyons le plan d'exécution estimé en XML :

```
DBCC FREEPROCCACHE
GO
SET SHOWPLAN_XML ON
GO
SELECT *
FROM Production.TransactionHistory
WHERE ProductId = 800
GO
SET SHOWPLAN_XML OFF
GO
```

Un extrait de ce plan :

```
<RelOp NodeId="0" PhysicalOp="Clustered Index Scan"
    LogicalOp="Clustered Index Scan" EstimateRows="418"
    EstimateIO="0.586088" EstimateCPU="0.124944" AvgRowSize="54"
    EstimatedTotalSubtreeCost="0.711032"
    Parallel="0" EstimateRebinds="0"
    EstimateRewinds="0">
```

Nous voyons que l'optimiseur choisit de parcourir la table (donc ici un *scan* de l'index *clustered*, puisque le dernier niveau de l'index *clustered* correspond aux données de la table) au lieu d'utiliser l'index. Nous voyons aussi dans l'attribut `EstimateRows` que les statistiques permettent à l'optimiseur d'avoir une idée assez précise du nombre de lignes correspondant à la clause `WHERE`.

Essayons maintenant d'indiquer un nombre plus petit de lignes à retourner :

```
SELECT ProductId, COUNT(*)
FROM Production.TransactionHistory
GROUP BY ProductId;
```

Nous voyons par exemple que le `ProductId` 760 ne se retrouve que six fois dans la table. Essayons avec cela :

```
DBCC FREEPROCCACHE
GO
```

```
SET SHOWPLAN_XML ON
GO
SELECT *
FROM Production.TransactionHistory
WHERE ProductId = 760
GO
SET SHOWPLAN_XML OFF
GO
```

Un extrait du plan d'exécution estimé :

```
<RelOp NodeId="2" PhysicalOp="Index Seek" LogicalOp="Index Seek"
    EstimateRows="5.28571" EstimateIO="0.003125"
    EstimateCPU="0.000162814"
    AvgRowSize="15" EstimatedTotalSubtreeCost="0.00328781" Parallel="0"
    EstimateRebinds="0" EstimateRewinds="0">
...
<Object Database="[AdventureWorks]" Schema="[Production]"
    Table="[TransactionHistory]"
    Index="[IX_TransactionHistory_ProductID]" />
```

Cette fois-ci nous voyons que l'index est utilisé : c'est la solution la moins coûteuse. L'estimation des lignes à retourner, basée sur les statistiques de distribution, donne 5,28571 lignes. Nous verrons plus loin d'où SQL Server tire cette approximation.

6.4.2 Colonnes non indexées

Les statistiques ne sont pas uniquement utiles pour les index. L'optimiseur peut profiter de la présence de statistiques même sur des colonnes qui ne font pas partie de la clé d'un index. Cela lui permettra d'évaluer le nombre de lignes retournées dans une requête qui filtre sur les valeurs de cette colonne, et donc de choisir un bon plan d'exécution. Par exemple, si la colonne participe à un `JOIN`, cela permettra à l'optimiseur de choisir le type de jointure le plus adapté.

En SQL Server 2000, comme les statistiques de colonne sont stockées dans la table système `sysindexes` avec la définition des index, la création de statistiques avait comme conséquence de diminuer le nombre possible d'index pouvant être créés sur la table.

Vous étiez limités à 249 index *nonclustered* par table, donc à 249 index plus statistiques.

Sur des tables contenant beaucoup de colonnes, la création automatique des statistiques pouvait vous faire atteindre cette limite. Il vous fallait alors supprimer des statistiques (avec la commande `DROP STATISTICS`) pour permettre la création de nouveaux index.

Dans SQL Server 2005, la limite du nombre de statistiques de colonne sur une table a été augmentée à 2000, plus 249 statistiques d'index, poussant le nombre de statistiques possibles sur une table à 2249.

Pour voir les statistiques créées automatiquement :

```
SELECT * FROM sys.stats WHERE auto_created = 1
```

leur nom commence par `_WA_Sys_`.

6.4.3 Sélectivité et densité

Ces statistiques de distribution permettent de déterminer la **sélectivité** d'un index. Plus le nombre de données uniques présentes dans la colonne est élevé, plus l'index est dit sélectif. La plus haute sélectivité est donnée par un index unique, où toutes les valeurs sont distinctes, et les plus basses sélectivités sont amenées par des colonnes qui contiennent beaucoup de doublons, comme les colonnes de type `BIT`, ou `CHAR(1)` avec seulement quelques valeurs (par exemple H et F pour une colonne de type sexe).

À l'inverse, la **densité** représente le nombre de valeurs dupliquées présentes dans la colonne. Plus il y a de valeurs dupliquées, plus la densité est élevée.

Sur l'index dans son entier, le pourcentage moyen de lignes dupliquées donne la sélectivité et la densité : plus ce pourcentage est faible, plus l'index est sélectif, et plus il est élevé, plus l'index est dense. La densité s'oppose donc à la sélectivité.

6.4.4 Consultation des statistiques

Nous avons quelquefois entendu des imprécisions au sujet des statistiques. Les statistiques ne sont pas des index, et ne sont pas stockées comme les index ou dans les pages d'index. Notamment les informations de distribution ne se situent pas au niveau des nœuds de l'arborescence de l'index. Si c'était le cas, le plan d'exécution ne pourrait pas faire le choix d'utiliser l'index ou non... avant de l'utiliser, puisqu'il devrait entrer dans l'arborescence pour retrouver l'estimation de lignes à retourner, puis ensuite quitter l'index pour faire un *scan* si ce choix lui paraît meilleur.

Ce genre de décision doit se prendre avant de commencer le processus de recherche des données.

Ainsi, les données de statistiques sont disponibles en tant que métadonnées de l'index, et non pas dans sa structure propre. Elles sont stockées dans une colonne de type LOB, dans une table système visible à travers la vue `sys.sysindexes` (du nom d'une ancienne table système de SQL Server 2000).

La requête

```
SELECT statblob
FROM sys.sysindexes
```

permet de voir que la colonne `statblob`, qui contient ces informations de statistiques, retourne `NULL`. La colonne contient bien des données, mais elles ne sont pas

visibles par ce moyen, et la valeur retournée par requête sera toujours `NULL`. D'ailleurs,

```
SELECT * FROM sys.indexes;
```

qui est la vue de catalogue officielle affichant les données d'index, ne retourne pas cette colonne. La seule façon de retourner les données de statistiques est d'utiliser une commande DBCC

```
DBCC SHOW_STATISTICS
```

ou d'afficher les propriétés des statistiques dans l'explorateur d'objets de SQL Server Management Studio (nœud Statistiques), page Détails, qui affiche les résultats du `DBCC SHOW_STATISTICS` dans une zone de texte.

Exemple avec notre index :

```
DBCC SHOW_STATISTICS ( 'Production.TralinsactionHistory',
    X_TransactionHistory_ProductID)
```

Cette commande retourne un en-tête, la densité par colonne de l'index, et un histogramme de distribution des données, selon l'échantillonnage opéré. Observons d'abord le résultat de l'en-tête (tableau 6.5).

Tableau 6.5 — Résultat de l'en-tête

Name	Updated	Rows	Rows Sampled
IX_TransactionHistory_ProductID	Apr 26 2006 11:45AM	113443	113443
Steps	**Density**	**Average key length**	**String Index**
200	0,01581469	8	NO

Vous trouvez dans l'en-tête les informations générales des statistiques : nom, date de dernière mise à jour, nombre de lignes dans l'index et nombre de lignes échantillonnées, nombre d'étapes (*steps*) effectuées pour effectuer cet échantillonnage, densité de l'index, longueur moyenne de la clé de l'index. Le booléen `String Index` indique si la colonne contient des résumés de chaîne, qui est une fonctionnalité de SQL Server 2005 que nous détaillerons plus loin.

Tableau 6.6 — Densité

All density	Average Length	Columns
0,0022675736961	4	ProductID
8,815E-06	8	ProductID, TransactionID

Le deuxième résultat indique la densité de chaque colonne de l'index. La clé de l'index dépendant toujours de la première colonne, la densité individuelle est calcu-

lée seulement pour celle-ci. Les densités calculées sont ensuite celles des colonnes agrégées dans l'ordre de leur présence dans l'index (le préfixe de la clé), une colonne après l'autre, si l'index est composite, ou si l'index est *nonclustered* et présent sur une table *clustered*. Dans ce cas, comme nous le voyons ici, chaque index *nonclustered* contiendra la ou les colonnes de l'index *clustered* en dernière position.

La densité est donnée par le nombre moyen de lignes pour une valeur de la colonne divisé par le nombre total de lignes. Le nombre moyen de lignes est calculé en prenant le nombre total de ligne (T) divisé par le nombre de valeurs distinctes (VD) dans la colonne. Ce qui donne :

(T / VD) / T qui équivaut à 1 / VD

Et en effet :

```
SELECT 1.00 / COUNT(DISTINCT ProductId))
FROM Production.TransactionHistory
```

retourne bien 0,0022675736961

Plus la densité est faible, plus la sélectivité est élevée, et plus l'index est utile. La sélectivité maximale est offerte par un index unique, comme ici l'index *clustered* unique sur la clé primaire de la table.

La densité de la colonne unique est donc 1 / VD, qui correspond logiquement à 1 / T, puisque le nombre de valeurs distinctes équivaut au nombre de lignes de la table.

```
SELECT 1.00 / COUNT(*)
FROM Production.TransactionHistory
```

retourne 0,0000088149996, ce qui représente une très basse densité, donc une sélectivité très élevée. Pour vérifier que nous obtenons bien la même valeur que la deuxième densité (`ProductID`, `TransactionID`), forçons la notation scientifique en retournant une donnée `FLOAT` :

```
SELECT CAST(1.00 as float) / COUNT(*)
FROM Production.TransactionHistory
```

qui donne une valeur approximative de 8,815E-06, CQFD.

Le couple `ProductID + TransactionID` ne peut être que d'une sélectivité maximale, puisqu'incluant une colonne unique, chaque ligne est donc unique, et sa densité est 1 / T.

Tableau 6.7 — Échantillonnage

RANGE _HI_KEY	RANGE _ROWS	EQ_ROWS	DISTINCT _RANGE_ROWS	AVG _RANGE_ROWS
1	45	45	0	1
3	45	307	1	45
316	144	307	1	45

RANGE _HI_KEY	RANGE _ROWS	EQ_ROWS	DISTINCT _RANGE_ROWS	AVG _RANGE_ROWS
319	218	118	2	72
...				
779	37	771	7	5,285714

Avec les deux affichages précédents, l'optimiseur SQL sait quel est le nombre de lignes dans la table, et le nombre de valeurs distinctes dans la colonne indexée. Il lui faut maintenant pouvoir évaluer, pour une certaine valeur de colonne, quel sera le nombre approximatif de lignes concernées (combien `ProductId = 760` concerne-t-il de lignes ?). Pour cela, il maintient un échantillonnage des valeurs établi selon un certain nombre de sauts (les *steps* que nous avons vus dans les informations de *header*), sous forme d'un tableau. Dans notre exemple, un tableau de 200 *steps*. SQL Server crée un maximum de 200 *steps* par statistiques. La signification des colonnes est détaillée dans le tableau 6.8.

Tableau 6.8 — Signification des colonnes

Colonne	Contenu
RANGE_HI_KEY	La valeur de colonne de l'échantillon, donc de la dernière ligne de l'ensemble échantillonné (le *bucket*)
RANGE_ROWS	Le nombre de lignes entre l'échantillon et l'échantillon précédent, ces deux échantillons non compris
EQ_ROWS	Nombre de lignes dans l'ensemble dont la valeur est égale à celle de l'échantillon
DISTINCT_RANGE_ROWS	Nombre de valeurs distinctes dans l'ensemble
AVG_RANGE_ROWS	Nombre moyen de lignes de l'ensemble ayant la même valeur, donc RANGE_ROWS / DISTINCT_RANGE_ROWS

Nous voyons donc ici comment l'optimiseur a estimé le nombre de lignes retournées par la clause `WHERE ProductId = 760` : il s'est positionné sur l'échantillon 779, puisque la valeur 760 est contenue dans l'ensemble échantillonné de cette ligne, et a retrouvé la valeur de `AVG_RANGE_ROWS`. Dans la case de `ProductId = 800`, l'optimiseur a trouvé la ligne où le `RANGE_HI_KEY = 800`. Le nombre d'`EQ_ROWS` est 418. Pourquoi l'optimiseur fait-il le choix de parcourir la table au lieu d'utiliser l'index ? Sachant qu'il aura à retourner 418 lignes, cela fait donc potentiellement un maximum de 418 pages de 8 Ko en plus des pages de l'index à parcourir. Nous savons, grâce à la requête suivante, combien de pages sont utilisées par chaque index :

```
SELECT o.name AS table_name,
       p.index_id,
       i.name AS index_name,
```

```
       au.type_desc AS allocation_type,
       au.data_pages, partition_number
FROM   sys.allocation_units AS au
JOIN   sys.partitions AS p ON au.container_id = p.partition_id
JOIN   sys.objects AS o ON p.object_id = o.object_id
LEFT JOIN sys.indexes AS i  ON p.index_id = i.index_id
                           AND i.object_id = p.object_id
WHERE  o.name = N'TransactionHistory'
ORDER BY o.name, p.index_id
```

Tableau 6.9 — Allocation des index

index_name	allocation_type	data_pages
PK_TransactionHistory_TransactionID	IN_ROW_DATA	788
IX_TransactionHistory_ProductID	IN_ROW_DATA	155
IX_TransactionHistory_ReferenceOrderID _ReferenceOrderLineID	IN_ROW_DATA	211

Nous pouvons donc assumer qu'un *scan* de la table (donc de l'index *clustered*) coûtera la lecture de 788 pages, donc 788 *reads*. Cela représente bien plus que 418 pages. Pourquoi choisir un *scan* ?

Analysons ce qui se passe réellement lorsque nous utilisons un plan ou un autre. Nous pouvons le découvrir en forçant l'optimiseur à utiliser l'index en ajoutant un indicateur de table dans la requête (nous verrons les indicateurs de table dans la section 8.2.1) :

```
SET STATISTICS IO ON
GO
SELECT *
FROM Production.TransactionHistory
       WITH (INDEX = IX_TransactionHistory_ProductID)
WHERE ProductId = 800
```

Voici les résultats de pages lues (reçues grâce à `SET STATISTICS IO ON`), et le plan d'exécution utilisé :

```
Table 'TransactionHistory'.
Scan count 1, logical reads 1979, physical reads 3,
    read-ahead reads 744,
lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.
```

Relançons ensuite la requête en laissant SQL Server choisir son plan :

```
SELECT *
FROM Production.TransactionHistory
WHERE ProductId = 800
```

Résultat :

```
Table 'TransactionHistory'.
```

```
Scan count 1, logical reads 792, physical reads 0, read-ahead reads 44,
lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.
```

SQL Server essaie autant que possible d'éviter le *bookmark lookup*. Le plan généré utilise donc une jointure entre les deux index : l'index *nonclustered* `IX_TransactionHistory_ProductID` est utilisé pour chercher les lignes correspondant au critère, et une jointure de type *nested loop* est utilisée pour parcourir l'index *clustered* à chaque occurrence trouvée, pour extraire toutes les données de la ligne (`SELECT *`).

Cette jointure doit donc lire bien plus de pages que celles de l'index, puisqu'il faut aller chercher toutes les colonnes de la ligne. À chaque ligne trouvée, il faut parcourir l'index *clustered* pour retrouver la page de données et en extraire la ligne. En l'occurrence cela entraîne la lecture de 1 979 pages.

Nous avons vu que l'index *clustered* utilise 788 pages. Le *scan* de cet index, selon les informations d'entrées/sorties, a coûté 792 lectures de page. D'où sortent les quatre pages supplémentaires ?

Selon la documentation de la vue `sys.allocation_units` dont la valeur 788 est tirée, la colonne data_pages exclut les pages internes de l'index et les pages de gestion d'allocation (IAM). Les quatre pages supplémentaires sont donc probablement ces pages internes. Nous pouvons le vérifier rapidement grâce à `DBCC IND` :

```
DBCC IND (AdventureWorks, 'Production.TransactionHistory', 1)
```

où le troisième paramètre est l'ID de l'index. L'index *clustered* prend toujours l'ID 1.

Le résultat de cette commande nous donne 792 pages, donc 788 sont des pages de données, plus une page d'IAM (`PageType = 10`) et trois pages internes (`PageType = 2`). Voilà pour le mystère.

Résumés de chaîne

Les **résumés de chaîne** (*string summary*) sont une addition aux statistiques dans SQL Server 2005. Comme vous l'avez vu dans le *header* retourné par `DBCC SHOW_STATISTICS`, la valeur de `String Index` indique si les statistiques pour cette clé de l'index contiennent des résumés de chaîne. Il s'agit d'un échantillonnage à l'intérieur d'une colonne de type chaîne de caractères (un `VARCHAR` par exemple), qui permet à l'optimiseur d'avoir une meilleure estimation du nombre de lignes que la requête va retourner. Ces statistiques ne sont pas utiles pour un *seek* d'index, bien sûr, puisque la recherche ne pourra se faire sur une clé d'index, mais elles permettent simplement d'améliorer l'estimation de cardinalité. Les résumés de chaîne ne couvrent que 80 caractères de la donnée (les 40 premiers et 40 derniers si la chaîne est plus longue).

6.4.5 Maintien des statistiques

Consultation

Nous pouvons vérifier la présence de statistique de plusieurs manières :

- `sp_helpstats` : est l'ancienne méthode d'obtention des statistiques. Ne l'utilisez plus et préférez les vues de catalogues suivantes.
- `sys.stats` : liste des statistiques présentes dans la base de données.
- `sys.stats_columns` : colonnes présentes pour chaque statistique.

```
SELECT *
FROM sys.stats s
JOIN sys.stats_columns sc ON s.object_id = sc.object_id
                         AND  s.stats_id  = sc.stats_id
JOIN sys.columns c ON s.object_id = c.object_id
                         AND  sc.column_id = c.column_id
```

- **`STATS_DATE()`** : vous pouvez récupérer la date de dernière mise à jour des statistiques en utilisant la nouvelle fonction `STATS_DATE()` de SQL Server 2005, à laquelle vous passez un ID de table et un ID d'index.

```
SELECT  OBJECT_NAME(i.object_id) AS table_name,
        i.name AS index_name,
        STATS_DATE(i.object_id, i.index_id)
FROM sys.indexes AS i
WHERE OBJECT_NAME(i.object_id) = N'TransactionHistory'
```

- **`DBCC SHOW_STATISTICS`** : comme nous l'avons déjà vu, `DBCC SHOW_STATISTICS` affiche les détails d'un jeu de statistiques.

Création

Les statistiques sur les index sont créées automatiquement, sans possibilité de désactivation : un index sans statistiques serait inutilisable.

Les statistiques sur les colonnes non indexées peuvent être créées manuellement ou automatiquement.

Par défaut, les bases de données sont créées avec l'option `AUTO_CREATE_STATISTICS` à `ON`, c'est-à-dire que les statistiques dont l'optimiseur de requêtes a besoin pour générer son plan d'exécution seront générées à la volée. Certes, il y aura un ralentissement du service de la requête dû à la création des statistiques, mais cela ne se produira que la première fois : potentiellement ce ralentissement sera moindre que celui induit par un plan d'exécution mal évalué. Cette fonctionnalité, de création et mise à jour automatique des statistiques par SQL Server, est appelée *Auto Stats*. Pour créer manuellement des statistiques sur une ou plusieurs colonnes, vous pouvez utiliser la commande **`CREATE STATISTICS`**:

```
CREATE STATISTICS statistics_name
ON { table | view } ( column [ ,...n ] )
    [ WITH
```

```
        [ [ FULLSCAN
          | SAMPLE number { PERCENT | ROWS }
        [ NORECOMPUTE ]
    ] ;
```

Il peut être intéressant par exemple de créer manuellement des statistiques sur une combinaison de colonnes pour améliorer l'estimation du plan d'une requête dont la clause `WHERE` filtre sur ces colonnes.

`WITH FULLSCAN` vous permet d'indiquer que toutes les valeurs doivent être parcourues. Sur les tables volumineuses, l'échantillonnage ne se fait pas sur toutes les valeurs, les statistiques seront donc moins précises. En indiquant `WITH FULLSCAN` (ou `WITH SAMPLE 100 PERCENT`, qui est équivalent), vous forcez SQL Server à prendre en compte toutes les lignes. Pour les tables de taille moyenne, l'échantillonnage se fait de toute manière sur toutes les valeurs. SQL Server s'assure un minimum de 8 Mo de données échantillonnées (1024 pages ou plus), ou la taille de la table si elle pèse moins.

À l'inverse, vous pouvez demander un échantillonnage moins précis avec l'option **`WITH SAMPLE`**. Notez que si SQL Server considère que votre sampling n'est pas suffisamment élevé pour générer des statistiques utiles, il corrige lui-même la valeur à la hausse.

Avec **`NORECOMPUTE`**, vous forcez SQL Server à ne jamais mettre à jour automatiquement ces statistiques. Ne l'utilisez que si vous savez ce que vous faites.

Vous pouvez supprimer des statistiques avec l'instruction **`DROP STATISTICS`**. Vous ne devriez normalement jamais avoir à faire cela.

Vous pouvez, grâce à la procédure **`sp_createstats`** créer en une seule fois des statistiques pour toutes les colonnes de toutes les tables de votre base. Cela est superflu dans la plupart des cas : pourquoi maintenir des statistiques sur des colonnes jamais filtrées ?

Mise à jour

La **mise à jour des statistiques** est importante. Pour reprendre notre exemple, supposons qu'avant de partir pour Courtrai, vous en parliez avec votre grand-père qui vous annonce qu'il connaît bien Courtrai pour y avoir passé ses vacances d'été pendant plusieurs années avec sa famille lorsqu'il était enfant. Par politesse, vous évitez de commenter les choix de destination de vacances de vos arrières grands-parents, d'autant plus qu'il se souvient d'avoir eu en son temps un plan détaillé de la ville. Intéressant, cela vous permettrait de vous faire une idée plus précise de vos déplacements. Après quelques recherches au grenier, il vous ressort un vieux plan très jauni, qui date de 1933. Pensez-vous réellement qu'il va vous être très utile, sachant que la ville a été très endommagée par les bombardements de 1944, et qu'une grande partie de celle-ci a été reconstruite après la guerre ?

Le contenu de votre table évolue, et vous ne pouvez baser *ad vitam aeternam* l'évaluation de la sélectivité d'une colonne sur les statistiques telles qu'elles ont été créées. Cette mise à jour doit se faire régulièrement et prendre en compte les larges

modifications de données. Comme la création, elle peut se déclencher automatiquement ou manuellement.

Automatiquement avec l'option de base de données **AUTO UPDATE STATISTICS**, qu'il est vivement recommandé de laisser à sa valeur par défaut ON.

```
SELECT DATABASEPROPERTYEX('IsAutoUpdateStatistics')
    -- pour consulter la valeur actuelle
ALTER DATABASE AdventureWorks SET AUTO_UPDATE_STATISTICS [ON|OFF]
    -- pour modifier la valeur
```

En SQL Server 2000, la mise à jour automatique des statistiques était déclenchée par un nombre de modifications de lignes de la table. Chaque fois qu'un INSERT, DELETE ou une modification de colonne indexée était réalisé, la valeur de la colonne rowmodctr de la table sysindexes était incrémentée. Pour que la mise à jour des statistiques soit décidée, il fallait que la valeur de rowmodctr soit au moins 500.

> Vous trouverez une explication plus détaillée de ce mécanisme dans le document de la base de connaissances Microsoft 195565 (http://support.microsoft.com/kb/195565).

Avec SQL Server 2005, ce seuil (*threshold*) est géré plus finement. La colonne rowmodctr est toujours maintenue mais on ne peut plus considérer sa valeur comme exacte. En effet, SQL Server maintient des compteurs de modifications pour chaque colonne, dans un enregistrement nommé colmodctr présent pour chaque colonne de statistique. Cependant cette colonne n'est pas visible. Vous pouvez toujours vous baser sur la valeur de rowmodctr pour déterminer si vous pouvez mettre à jour les statistiques sur une colonne, ou si la mise à jour automatique va se déclencher bientôt, car les développeurs ont fait en sorte que sa valeur soit « raisonnablement » proche de ce qu'elle donnait dans les versions suivantes.

Vous pouvez activer ou désactiver la mise à jour automatique des statistiques sur un index, une table ou des statistiques, en utilisant la procédure stockée sp_autostats :

```
sp_autostats [@tblname =] 'table_name'
             [, [@flagc =] 'stats_flag']
             [, [@indname =] 'index_name']
```

Le paramètre @flagc peut être indiqué à ON pour activer, OFF pour désactiver, NULL pour afficher l'état actuel.

sp_updatestats met à jour les statistiques de toutes les tables de la base courante, mais seulement celles qui ont dépassé le seuil d'obsolescence déterminé par rowmodctr (contrairement à SQL Server 2000 qui mettait à jour toutes les statistiques). Cela vous permet de lancer une mise à jour des statistiques de façon contrôlée, durant des périodes creuses, afin d'éviter d'éventuels problèmes de performances dus à la mise à jour automatique, dont nous allons parler dans la section suivante.

Nous avons vu que les mises à jour de statistiques sont indispensables à la bonne performance des requêtes, et que leur maintenance est nécessaire pour assurer une bonne qualité à travers le temps. Mais, quand cette opération se déclenche-t-elle ?

Tout comme la création automatique, elle se produit au besoin, c'est-à-dire au moment où une requête va s'exécuter et pour laquelle ces statistiques sont utiles, aussi bien dans le cas d'une requête ad hoc (c'est-à-dire d'une requête écrite une fois pour un besoin particulier), que d'une procédure stockée. Si le seuil de modification de la table est atteint, il va d'abord lancer une mise à jour des statistiques nécessaires, puis ensuite générer son plan d'exécution. Cela peut évidemment provoquer une latence à l'exécution de la requête. Pour les plans d'exécution déjà dans le cache, la phase d'exécution inclut une vérification des statistiques, un `UPDATE` éventuel de celles-ci, puis une recompilation, ce qui dans certains cas peut se révéler encore plus lourd.

Cela pose rarement un problème, mais si vous êtes dans le cas où cela affecte vos performances, vous avez quelques solutions à votre disposition :

- `AUTO_UPDATE_STATISTICS_ASYNC` (option de base de donnéees) vous permet de réaliser une mise à jour des statistiques de façon asynchrone. Cela veut dire que lorsque le moteur SQL décide de mettre à jour ses statistiques il n'attend pas la fin de cette opération pour exécuter la requête déclenchante. Elle démarre donc plus vite, mais ne s'exécute pas forcément plus rapidement, car avec un plan d'exécution potentiellement obsolète. Les exécutions suivantes profiteront des statistiques rafraîchies. En temps normal, cette option n'est pas nécessaire.
- Vous pouvez mettre à jour manuellement vos statistiques durant vos fenêtres d'administration, en utilisant par exemple `sp_updatestats`, ou une tâche d'un plan de maintenance.

6.5 DATABASE ENGINE TUNING ADVISOR

Le *Database Engine Tuning Advisor* (DTA), traduit maladroitement en SQL Server 2005 par « assistant de paramétrage de base de données », est un outil graphique prenant en charge l'analyse d'une charge de travail (*workload*) pour fournir automatiquement des recommandations de création de structures physiques de données (*physical design structures*, PDS) destinées à améliorer les performances : index, vues indexées et partitions. La charge de travail est soumise par le **DTA** au moteur d'optimisation de SQL Server, avec différentes options, pour juger des PDS les plus aptes à améliorer les performances du *workload*.

DTA remplace l'outil disponible en SQL Server 2000, nommé *Index Tuning Wizard* (ITW). Vous apprécierez le glissement sémantique : du magicien (*wizard*), nous passons au conseiller (*advisor*), un sursaut d'humilité bienvenu (même si le DTA est plus puissant que l'ITW) : bien que très utile et intéressant pour faire un

premier travail d'optimisation, par exemple à la mise en place d'une nouvelle base en production, l'outil n'est pas miraculeux, et ne peut remplacer le travail d'optimisation manuel, qui, par l'alliance des outils de tuning de SQL Server, de vos connaissances et de votre réflexion, reste un travail continu et essentiellement artisanal, ce qui en fait l'intérêt.

Pour s'assurer d'obtenir des conseils de bonne qualité de la part du DTA, l'élément essentiel est de disposer d'un bon *workload*. Le DTA analyse un jeu de requêtes SQL s'appliquant à une ou plusieurs bases de données, dans lesquelles vous pouvez éventuellement choisir des tables spécifiques. Ce *workload* doit être aussi représentatif que possible : toutes les requêtes importantes doivent y figurer. Si ce n'est pas le cas, le DTA risque de fournir des recommandations incomplètes, voire nuisibles aux performances des requêtes absentes. Il peut être de différents formats. DTA se nourrit soit de simples fichiers de *batches* de requêtes T-SQL, où vous pouvez séparer les *batches* par l'instruction `GO`, soit de traces SQL sauvegardées dans un fichier ou dans une table. Vous pouvez également insérer les commandes SQL à analyser dans un fichier XML au format spécifique au DTA[1], ce qui vous permet d'ajouter un poids relatif à chaque requête pour favoriser les recommandations sur des requêtes plus importantes que d'autres.

La trace SQL doit contenir les événements suivants :

- RPC:Completed
- SQL:BatchCompleted
- SP:StmtCompleted

et au moins les colonnes `EventClass` et `TextData`. La colonne `Duration` est aussi utilisée par le DTA, qui se concentre lorsqu'elle est présente sur les instructions les plus longues à exécuter. Le modèle (*template*) du profiler nommé « tuning » est déjà prêt pour vos sessions de trace adaptées. Évidemment, il est important de tracer l'activité de la base à un moment de forte utilisation, lorsqu'elle travaille vraiment, sur un serveur de production, afin d'obtenir le *workload* le plus proche possible du réel. Vous pouvez configurer votre trace pour la rotation de fichiers (*rollover*), le DTA reconnaît les fichiers numérotés. Si vous sauvegardez votre trace dans une table, celle-ci doit être locale au serveur sur lequel vous allez lancer le DTA, et la trace doit être arrêtée.

Si vous utilisez un fichier de requêtes, inscrivez-y simplement une seule fois chaque requête, en essayant de donner aux critères de filtre dans la clause `WHERE` de valeurs moyennes et représentatives : nous avons vu que, grâce aux statistiques, l'optimiseur est sensible à la sélectivité de ces valeurs. Les requêtes d'insertion, de mise à jour et de suppression sont aussi importantes. Non content d'être elles aussi optimisables, elles donnent l'occasion au DTA de tester le coût de maintenance des index qu'il va suggérer.

1. Ce document XML est décrit par le schéma DTASchema.xsd disponible à l'adresse http://schemas.microsoft.com/sqlserver/. Ce schéma décrit toutes les parties que nous aborderons dans ce chapitre, notamment la définition de la session.

Vous pouvez lancer le DTA depuis SSMS (menu Outils/Assistant paramétrage de base de données ou Tools/Database engine tuning advisor), ou depuis le menu Microsoft SQL Server 200*X* / Performance Tools / Database Engine Tuning Advisor. L'exécutable s'appelle `dtashell.exe`. Le DTA peut être aussi piloté en ligne de commande, *via* `dta.exe`, nous l'aborderons plus loin. Dans SSMS, vous pouvez également sélectionner un *batch* de requêtes, et lancer le DTA avec un clic droit, et la commande « Analyze Query in Database Engine Tuning Advisor ». Pour utiliser le DTA, vous devez faire partie du rôle serveur `sysadmin`, ou du rôle de base de données `db_owner`, des bases à analyser.

Après connexion au serveur, le DTA s'ouvre sur la fenêtre reproduite en figure 6.14.

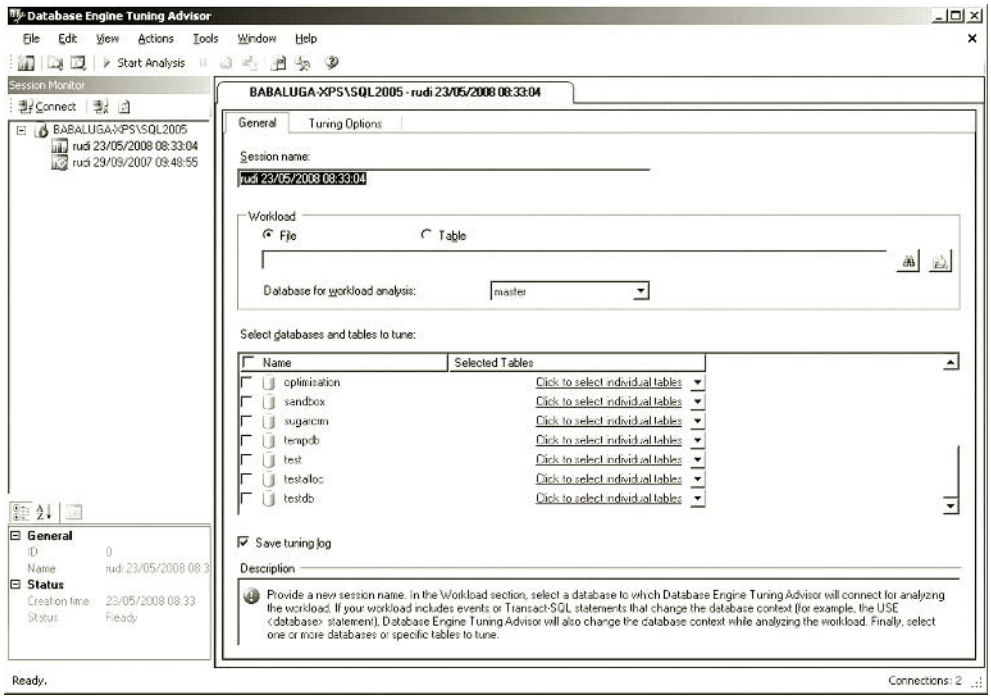

Figure 6.14 — Premier écran du DTA

Vous voyez sur la gauche un moniteur de sessions : vous pouvez conserver vos sessions, avec les paramètres et recommandations produites, et les rouvrir ou les relancer au besoin. Un clic droit sur une session vous permet, entre autres, de la « cloner » pour baser une nouvelle analyse sur son jeu de paramètres. Vous pouvez la supprimer de la même manière.

Les sessions (paramètres, recommandations, rapports, etc.) sont enregistrées sur le serveur SQL, dans la base de données système `msdb`. Un jeu de 17 tables est créé dans cette base au premier enregistrement d'une session. Elles portent toutes le pré-

fixe DTA_. Une procédure stockée, `msdb.dbo.sp_DTA_help_session`, vous permet de voir rapidement les sessions enregistrées. C'est cette procédure qui est appelée par le DTA pour remplir les informations de la fenêtre du moniteur.

La partie Workload vous permet de choisir un fichier ou une table. La table doit avoir été générée par le profiler. Vous choisissez ensuite la base de données dans laquelle les requêtes du *workload* s'exécutent, il s'agit du contexte de départ, comme lorsque vous ouvrez une fenêtre dans SSMS. Les éventuelles instructions USE dans le *workload* seront honorées et modifieront le contexte, ce qui permet, dans les faits, de travailler sur plusieurs bases de données dans la même session. Le DTA est encore plus subtil lorsqu'il analyse une trace : si la colonne `LoginName` est présente, il évaluera l'instruction avec les privilèges et dans le contexte de la connexion (*login*) originale (par exemple pour résoudre le nom des objets non préfixés par un schéma, selon le schéma par défaut de l'utilisateur) ; de plus, il se sert de la colonne d'événement `DatabaseName` pour savoir dans quel contexte évaluer chaque instruction. Si cette colonne est absente, il utilise `DatabaseID`, sinon la base de données indiquée dans la fenêtre du DTA est utilisée. En bref, le DTA essaie autant que possible de reproduire la session utilisateur dans laquelle s'est exécutée l'instruction.

Attention – Si vous voulez que le DTA se base sur la colonne `DatabaseID`, assurez-vous que la session s'exécute sur le même serveur que celui sur lequel la trace a été enregistrée, et que les bases n'ont pas été détachées/attachées, sinon, le `DatabaseID` peut ne plus correspondre aux bases originales. Il est plus sûr de se baser sur la colonne `DatabaseName`.

Ensuite, vous pouvez choisir quelles sont les bases de données, et éventuellement les tables, qui seront concernées par l'analyse. Si vous avez dans votre trace des requêtes sur des objets dont vous ne voulez pas modifier la structure physique, et sur lesquelles vous ne souhaitez aucun conseil, ne les sélectionnez pas à ce niveau. Pour sélectionner toutes les bases et toutes leurs tables, cliquez dans la case située sur l'entête de la grille.

La coche « Save tuning log » permet de conserver le journal, qui enregistre des messages importants sur l'analyse, notamment quand une table ne peut être analysée, soit parce qu'elle n'existe pas, soit parce que sa cardinalité est faible (moins de dix lignes), et qu'il est inutile d'ajouter quelque index que ce soit.

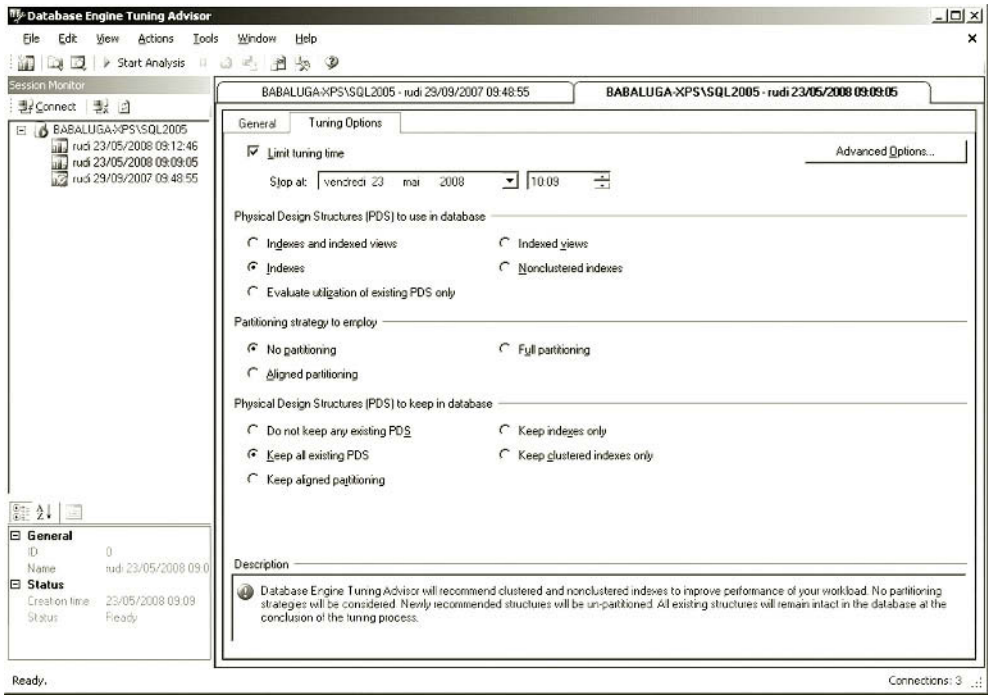

Figure 6.15 — Deuxième écran du DTA

Le deuxième onglet, reproduit dans la figure 6.15, permet d'indiquer les options de recommandations.

La première option permet de limiter la durée de l'analyse. Cela peut être utile sur un serveur de production pour lancer une analyse durant une fenêtre administrative, et éviter de déborder sur la reprise du travail. Le désavantage est que la limitation de la durée de l'analyse diminue la qualité des recommandations : le DTA travaille de façon incrémentale, en analysant une partie des instructions à la fois, en affinant au fur et à mesure ses recommandations. Un bon compromis pour réaliser une analyse correcte sans impact sur la production est de déplacer l'analyse sur un serveur de test, nous le verrons un peu plus loin. Si à la fin du temps indiqué, le DTA n'a pas pu terminer son travail, vous en trouverez la mention dans le journal, avec un message du genre « *Database Engine Tuning Advisor was unable to tune the entire workload…* ».

La partie nommée « *Physical Design Structures (PDS) to use in database* » permet d'indiquer quelles structures seront suggérées par le DTA. Les options sont mutuellement exclusives (boutons radio). Vous pouvez demander des suggestions d'index et vues indexées, seulement vues indexées, seulement index, et seulement index *non-clustered*. L'option « *Evaluate utilization of existing PDS only* » est utile en conjonction avec l'option « *Do not keep any existing PDS* », pour tester l'utilité des index et vues

indexées actuelles, et supprimer éventuellement celles qui sont inutiles. Vous pouvez obtenir des informations similaires avec la vue de gestion dynamique `sys.dm_db_index_operational_stats`, présentée dans la section 6.2.1.

La partie « *partitioning strategy to employ* » permet, sur une édition Entreprise, de suggérer le partitionnement de tables. Avec l'option « *Full partitioning* », des suggestions de partitionnement de tables peuvent être faites. « *Aligned partitioning* » indique au DTA d'aligner les index suggérés sur le schéma de partitionnement de la table.

La partie « *Physical Design Structures to Keep in the Database* » permet d'indiquer ce que le DTA doit conserver, des PDS existants. « *Do not keep existing PDS* » est utile pour optimiser un *workload* complet, par exemple à la mise en production d'une nouvelle base. Par opposition, choisissez « *Keep all existing PDS* » lorsque votre *workload* n'est pas complet ou représentatif, où lorsque vous utilisez le DTA pour analyser une sélection dans SSMS. Cela vous évitera de supprimer des PDS utiles par ailleurs. Dans ce cas, il est aussi logique de ne spécifier que des recommandations d'index *nonclustered*, ou de vues indexées.

Attention – Si le DTA vous recommande de supprimer beaucoup de vos index existants, alors que vous savez qu'ils sont utiles, c'est certainement parce que votre *workload* n'est pas représentatif, et qu'un nombre de requêtes importantes n'y figure pas. Le DTA considère toujours le *workload* comme représentatif.

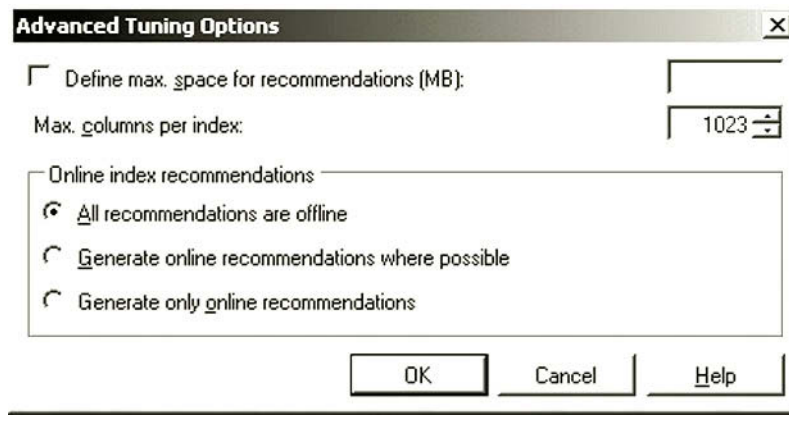

Figure 6.16 — Options avancées du DTA

Les options avancées, montrées dans la figure 6.16 permettent d'agir plus finement, en limitant l'augmentation de taille de la base de données provoquée par la totalité des recommandations, ou le nombre de colonnes proposées dans un index composite. Vous pouvez aussi, sur une édition Entreprise, spécifier que les sugges-

tions d'index doivent l'option `ONLINE`. Évidemment, comme le DTA ne crée aucun index en réalité, l'augmentation de taille provoquée par les index suggérés n'est qu'une estimation et peut se révéler différente lors de la création effective des PDS.

Lorsque vous avez préparé votre session, vous pouvez lancer le travail en cliquant sur le bouton « *Start Analysis* » de la barre de bouton. Une progression est montrée dans une nouvelle fenêtre, avec notamment l'affichage du journal (*tuning log*). Si vous constatez des erreurs ou des avertissements durant l'analyse, vous pouvez la stopper, avec ou sans génération des recommandations. Le DTA génère les « meilleures recommandations possibles » selon les instructions déjà analysées à la demande d'arrêt.

Lorsque l'analyse est terminée, deux onglets sont créés. Premièrement, « *recommendations* » liste les opérations recommandées (création ou suppression de PDS) et indique une estimation de pourcentage de gain de performances apporté par ces changements. La taille estimée des index est aussi affichée. Vous pouvez sélectionner les opérations choisies, voir le script de création et le copier dans le presse-papiers (figure 6.17).

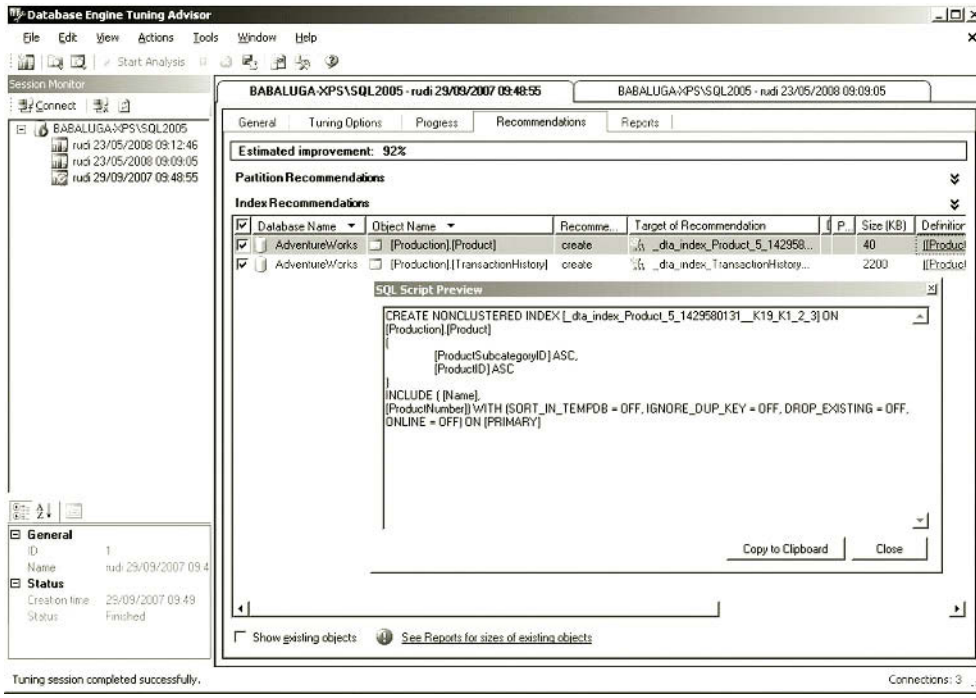

Figure 6.17 — Génération du code des recommandations.

Ensuite, l'onglet « *report* » (figure 6.18) offre une vue complète des résultats de l'analyse, vous permettant de voir en détail le travail effectué.

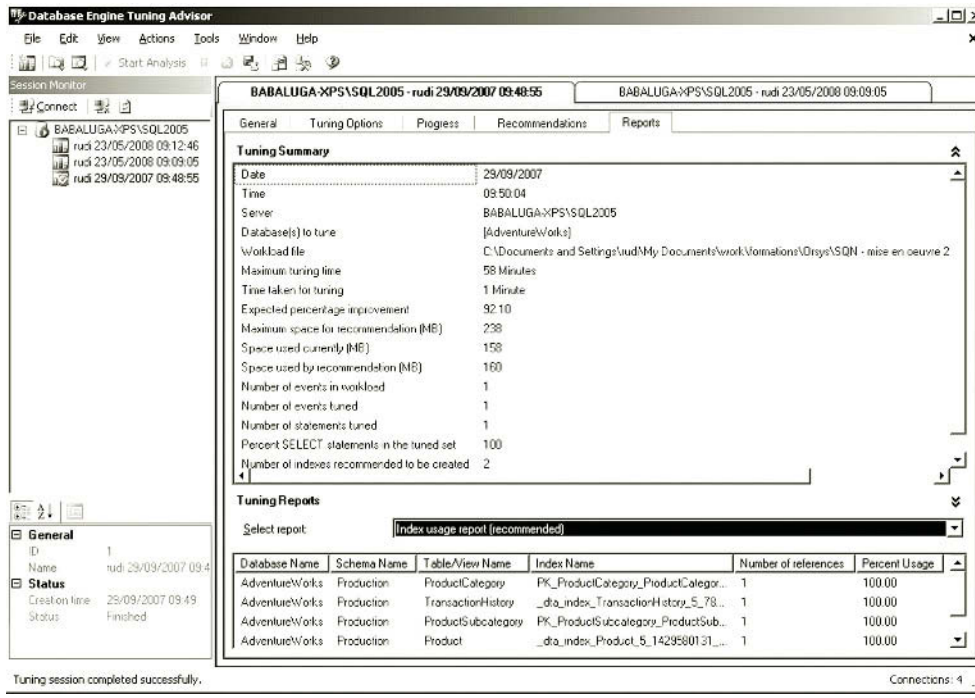

Figure 6.18 — Rapport des recommandations.

Analyse « comme si »

Le DTA permet une **analyse « prospective »** (*what-if analysis*). Cela veut dire que vous pouvez évaluer les différences de performances en rapport avec des modifications de PDS que vous prévoyez, par exemple analyser l'impact de la présence d'index, sans les créer réellement. Le DTA ne créera pas les structures physiques, mais il fera « comme si », et considérera les structures dans l'optimisation des requêtes. Vous pouvez ainsi indiquer des index (*clustered* et *nonclustered*), vues indexées et des partitionnements d'index. Cela vous permet effectivement de tester différentes solutions, comme différents partitionnements ou différents index *clustered*, de façon purement hypothétique, et de choisir la meilleure solution. Comme le DTA se base sur l'optimiseur SQL pour analyser ses requêtes, vous êtes assez proche de la vérité dans les résultats. C'est donc une alternative valable aux tests « réels ». Cela peut aussi vous permettre de tester les recommandations du DTA pour évaluer leur utilité.

Pour effectuer une analyse « comme si » dans une session graphique du DTA, utilisez la commande « Import Session Definition... » du menu Files, et insérez dans une définition de session XML. Voici un exemple de définition d'un index *nonclustered* fourni par une recommandation du DTA :

```
<?xml version="1.0" encoding="utf-16"?>
```

```
<Configuration SpecificationMode="Relative">
  <Server>
    <Name>BABALUGA-XPS\SQL2005</Name><Database>
  <Name>AdventureWorks</Name>
  <Schema>
    <Name>Production</Name>
    <Table>
      <Name>Product</Name>
      <Recommendation>
        <Create>
          <Index Clustered="false" Unique="false"
           Online="false" IndexSizeInMB="0.039063">
            <Name>_dta_index_Product_5_1429580131__K19_K1_2_3</Name>
            <Column SortOrder="Ascending" Type="KeyColumn">
              <Name>[ProductSubcategoryID]</Name>
            </Column>
            <Column SortOrder="Ascending" Type="KeyColumn">
              <Name>[ProductID]</Name>
            </Column>
            <Column SortOrder="Ascending" Type="IncludedColumn">
              <Name>[Name]</Name>
            </Column>
            <Column SortOrder="Ascending" Type="IncludedColumn">
              <Name>[ProductNumber]</Name>
            </Column>
            <FileGroup>[PRIMARY]</FileGroup>
          </Index>
        </Create>
      </Recommendation>
    </Table>
  </Schema>
</Database></Server>
</Configuration>
```

Vous pouvez faire générer cette définition automatiquement par le DTA d'après une session d'analyse, pour tester les recommandations du DTA sélectionnées en mode hypothétique, et donc essayer plusieurs parties des recommandations. Dans votre session, sélectionnez la commande « Evaluate Recommendations » du menu Actions. Une nouvelle session est alors créée, avec l'insertion automatique dans la définition de session, des PDS résultant de l'analyse.

Pour comparer les configurations, utilisez l'outil en ligne de commande `dta.exe` en spécifiant le nombre de lignes à traiter avec l'option `-n`, cela vous donne la garantie que les charges de travail seront identiques, et donc que vous comparerez bien les mêmes choses.

Utilisation en ligne de commande

L'exécutable `dta.exe` vous permet d'exécuter une session d'analyse en mode ligne de commande. Cela rend le DTA programmable et planifiable. Toutes les options sont exprimables par des paramètres passés à l'exécutable, ou dans un fichier XML d'entrée. Voici un exemple d'appel en ligne de commande :

```
dta.exe -S BABALUGA-XPS\SQL2005 -E -D AdventureWorks,AdventureWorksDW -d
AdventureWorks -s MaSession -if workload.sql -of rec.sql -fa IDX -A 0 -n 10000
```

qui signifie : connecte-toi au serveur BABALUGA-XPS\SQL2005 en sécurité intégrée, dans le contexte de `AdventureWorks` ; crée la session nommée `MaSession` pour analyser les bases `AdventureWorks` et `AdventureWorksDW` avec le fichier de requêtes `workload.sql`. Écris le script de génération des recommandations dans le fichier `rec.sql`. N'ajoute que des index, fais une analyse sans durée limite, mais en ne prenant que les 10 000 premières instructions du *workload* d'entrée. Pour interrompre `dta.exe` avant la fin du travail, utilisez la combinaison de touches CTRL+C.

Le DTA peut recevoir ses paramètres dans un fichier XML, mais aussi produire ses recommandations dans un output en format XML (avec le paramètre `-ox`), pour être par exemple utilisé comme outil d'optimisation intégré dans un outil tiers.

Vous trouverez la syntaxe d'appel complète dans les BOL, ou avec un

```
dta.exe -?
```

Analyse sur un serveur de test

Pour réaliser une bonne analyse, sans conséquence sur le serveur de production, DTA permet de déplacer la charge de l'analyse sur un **serveur de test**. Il utilise pour ce faire une méthodologie intelligente : une copie automatique de la base est effectuée sur le serveur de test, mais uniquement des métadonnées (structures d'objets) et des statistiques de distribution des colonnes. Les données elles-mêmes sont inutiles pour le processus d'optimisation, et il serait coûteux de les répliquer sur la machine de test. DTA s'en passe donc. De plus, pour coller au plus près de la configuration du serveur de production, il utilise les informations matérielles qu'il a glanées sur celui-ci (à l'aide de la procédure stockée étendue `xp_msver`) pour fonder son optimisation sur le serveur de test. On profite donc d'une analyse au plus près de la réalité, sans les inconvénients.

Pour ce faire, vous devez être sysadmin sur les deux serveurs, ou exécuter l'analyse sous un login présent sur les deux instances.

Vous devez ensuite utiliser un fichier XML, qui spécifie dans l'élément `<TuningOptions><TestServer></TestServer></TuningOptions>` le nom du serveur de test, et le passer en paramètre en ligne de commande, à `dta.exe`. L'outil graphique ne supporte pas le travail avec un serveur de test.

Ensuite, DTA importe dans une base de données de travail, sur le serveur de test, les métadonnées de la base de production : tables vides, index et objets de code, statistiques de distribution, informations du matériel de production (nombre de processeurs, quantité de mémoire).

7

Transactions et verrous

Objectif

La transaction est le mécanisme de base du maintien de la cohérence d'un SGBD client-serveur. Il est donc important de bien le connaître. L'outil mis en place par les transactions pour assurer cette cohérence est le verrouillage. Un verrouillage excessif pénalise les performances. Il est indispensable de maîtriser ce sujet et de limiter autant que possible l'étendue et la durée des transactions, donc du verrouillage, pour éviter les attentes trop longues, les blocages, et les étreintes fatales (*deadlocks*).

7.1 DÉFINITION D'UNE TRANSACTION

Un SGBDR est un système multi-utilisateurs. De ce point de vue, ses responsabilités sont doubles : il doit assurer un accès concurrent aux données, et assurer que celles-ci restent dans un état cohérent, par rapport aux contraintes définies par la structure de la base, aux modifications d'ensembles de données, et aux potentialités d'écritures simultanées. Cette protection est assurée par le mécanisme de la transaction. Les transactions permettent de contrôler la cohérence des données et de ne permettre une validation que lorsque celle-ci est respectée. Lors de toute modification, la base est dans un état incohérent durant un laps de temps plus ou moins long. Par exemple, si nous voulons mettre tous les noms de famille de la table `Person.Contact` en majuscules, nous exécutons le code suivant :

```
UPDATE Person.Contact
SET LastName = UPPER(LastName);
```

qui met à jour près de 20 000 lignes. Que se passe-t-il si une erreur se produit au milieu de cette modification, ou si, en même temps, une autre session change le nom d'un contact ? Du point de vue du SGBDR, ce sont des incohérences. Pour assurer que cela ne se produise pas, toute modification (insertion, suppression, mise à jour) de données est obligatoirement protégée par un mécanisme transactionnel qui garantit, à travers quatre critères, la consistance de cette modification. Ces critères sont nommés « l'*acid*ité » de la transaction, par leur acronyme (ACID). Une transaction doit être :

- **Atomique** – Une transaction est atomique, parce qu'elle représente une seule unité de modification, même si la mise à jour porte sur plusieurs milliers de lignes, ou plusieurs tables. Cela signifie qu'une telle modification sera totalement validée, ou totalement annulée. La base de données ne restera jamais dans un état intermédiaire, où une partie seulement des modifications est enregistrée.
- **Cohérente** – Elle est cohérente, parce qu'elle ne peut violer les contraintes de la base de données, ou laisser la base dans un état intermédiaire. Si dans une mise à jour de cent lignes, une ligne viole une contrainte d'intégrité référentielle (une clé étrangère), la transaction sera totalement annulée. La base passe d'un état cohérent avant la transaction, à un état cohérent après validation.
- **Isolée** – La transaction s'exécute dans un contexte d'isolation vis-à-vis des autres transactions. Les lignes touchées par une modification sont protégées contre un accès par une autre transaction, que ce soit en lecture (selon le niveau d'isolation, nous le verrons), ou surtout, en écriture.
- **Durable** – Lorsque la transaction est validée, elle est durable, c'est-à-dire que les modifications sont définitivement inscrites dans la base de données, sauf défaillance matérielle du disque, bien entendu.

Pour illustrer l'utilité de cette *acid*ité, imaginez-vous une partie de billard. Lorsque c'est votre tour de jouer, vous restez à la table tant que vous placez une bille dans une poche. Quand vous avez fini, vos points sont enregistrés, on ne peut revenir en arrière. Votre tour est donc atomique et durable. Vous ne pouvez violer les règles du jeu, sinon votre action est nulle, c'est donc consistant. Et, le plus important, personne ne peut jouer en même temps que vous : lorsque c'est votre tour, personne d'autre ne touche les billes sur la table, sinon, le jeu ne signifie plus rien.

La durabilité et l'atomicité sont possibles grâce au journal de transactions, qui permet de rejouer ou d'annuler une transaction, quelle que soit sa durée. L'isolation, qui est l'attribut qui garantit la cohérence des modifications tout en diminuant la concurrence, est gérée à travers le mécanisme de verrouillage.

Toute instruction DML de modification de données (`INSERT`, `UPDATE`, `DELETE`, `MERGE`) est encapsulée dans une transaction. Un seul `UPDATE` par exemple, quel que soit le nombre de lignes qu'il affecte, est donc ACID. Le `SELECT` ne génère aucune transaction. Pour enrôler plusieurs instructions dans une seule transaction, et donc

les rendre ACID, vous devez déclarer une transaction utilisateur explicite, à l'aide des instructions TCL (*Transaction Control Language*) du langage SQL :

- `BEGIN TRANSACTION` – débute une transaction explicite ;
- `SAVE TRANSACTION` – introduit un point de sauvegarde. On pourra ensuite valider ou annuler la partie de la transaction jusqu'à ce point ;
- `COMMIT TRANSACTION` – valide la transaction et la termine ;
- `ROLLBACK TRANSACTION` – annule la transaction et la termine.

Ces commandes permettent donc d'étendre les principes d'ACIDité de la transaction à plusieurs instructions DML ou DDL (en SQL Server, les instructions DDL sont transactionnelles). Dans les faits, cela transforme une suite d'ordres SQL en un seul ordre logique.

Validation automatique et annulation automatique

En SQL Server, le terme « transaction implicite » est utilisé pour décrire un fonctionnement connu sous le nom de « validation automatique » (*autocommit*). Par défaut, une session est en mode autocommit, c'est-à-dire qu'une instruction DML transactionnelle est automatiquement validée ou annulée (si elle provoque une exception) après exécution. Vous pouvez changer ce mode à l'aide de la commande `SET IMPLICIT_TRANSACTIONS { ON | OFF }`. En mode `IMPLICIT_TRANSACTIONS ON`, toute instruction doit valider ou annuler la transaction par un `COMMIT` ou un `ROLLBACK`. Ce mode est dangereux et en général inutile, ne l'utilisez donc pas.

De même, l'option `SET XACT_ABORT { ON | OFF }` permet d'annuler automatiquement une transaction explicite lorsqu'une instruction provoque une exception. Elle est activée par défaut. Elle influe sur la valeur de la fonction `XACT_STATE()` qui peut être testée, en général dans la partie `CATCH` d'un bloc `TRY CATCH`, pour connaître l'état de la transaction. Après une exception, si `XACT_ABORT` est à `OFF`, `XACT_STATE()` vaudra 1 (transaction active et validable. Si elle est à `ON`, `XACT_STATE()` vaudra -1 (transaction active et en erreur). Lorsque `XACT_STATE()` vaut -1, seul un `ROLLBACK` est possible.

Dans la pratique, il est rare de vouloir gérer une validation partielle de la transaction, cela peut provoquer plus facilement de la confusion que des fonctionnalités utiles.

7.2 VERROUILLAGE

Le principe de la transaction qui nous intéresse plus particulièrement en regard des performances, est l'**isolation**. Il consiste à assurer qu'une seule transaction à la fois puisse accéder aux mêmes données. Cette isolation est réalisée à travers le mécanisme de verrouillage. Dès qu'une transaction accède à une ressource, celle-ci est verrouillée (le moteur de stockage pose un verrou sur l'objet : la ligne, la page ou la table), afin de la protéger des accès concurrents. Il existe plusieurs types de verrous (ou modes de verrouillage), qui sont compatibles ou non entre eux, et plusieurs granularités. Nous allons les passer en revue.

7.2.1 Modes de verrouillage

Vous trouvez dans le tableau 7.1 une liste des types de verrous les plus fréquemment rencontrés dans SQL Server.

Tableau 7.1 — Les modes de verrouillage

Mode	Nom	Description
S	Shared	Verrou partagé
X	Exclusive	Verrou exclusif : une écriture est en cours sur la ressource verrouillée.
U	Update	Verrou de mise à jour
IS	Intent Shared	Verrou d'intention partagé. Un verrou partagé (S) est posé à une granularité plus fine.
IX	Intent Exclusive	Verrou d'intention exclusif. Un verrou exclusif (X) est posé à une granularité plus fine.
Sch-S	Schema Shared	Verrou partagé de schéma : lors de l'utilisation d'un objet, ce verrou empêche une modification de la structure de cet objet.

Certains verrous sont mutuellement compatibles : deux verrous compatibles peuvent être simultanément posés sur la même ressource. Les verrous incompatibles sont mutuellement exclusifs. Le tableau 7.2 présente la compatibilité des principaux modes de verrouillage.

Tableau 7.2 — Compatibilité des verrous

Mode	S	X	U	IS	IX	SIX
S	oui	non	oui	oui	non	non
X	non	non	non	non	non	non
U	oui	non	non	oui	non	non
IS	oui	non	oui	oui	oui	oui
IX	non	non	non	oui	oui	non
SIX	non	non	non	oui	non	non

Les verrous **partagés** (*shared*, S) sont des verrous de lecture. Ils sont posés lors de tout `SELECT`, dans les niveaux d'isolation à partir de `READ COMMITTED` (voir section 7.3). Comme leur nom l'indique, ils sont compatibles entre eux. Vous pouvez donc poser des verrous partagés sur des ressources qui en ont déjà, et ainsi lire en

même temps les mêmes lignes. Par contre, ils permettent de protéger les opérations de lecture contre l'écriture simultanée par une autre transaction, ce qui pourrait déboucher sur des lectures sales. Ils sont incompatibles avec les verrous d'écriture (X), ce qui permet, dans l'autre sens, de prévenir une lecture sur une ressource en cours de modification.

Prenons la requête suivante, qui met à jour la table d'en-têtes de commandes pour ajouter dix jours à la date d'envoi, puis ajoute dans la même transaction dix jours à la date limite de paiement dans la table de détail de commandes :

```
BEGIN TRAN
BEGIN TRY
       UPDATE Purchasing.PurchaseOrderHeader
       SET ShipDate = DATEADD(day, 10, ShipDate)

       UPDATE Purchasing.PurchaseOrderDetail
       SET DueDate = DATEADD(day, 10, DueDate)

       COMMIT TRAN
END TRY
BEGIN CATCH
       IF (XACT_STATE()) <> 0
              ROLLBACK TRAN
END CATCH
```

Si une lecture a lieu entre les deux instructions, elle peut trouver des dates de paiement antérieures aux dates d'envoi, ce qui est une incohérence fonctionnelle. Pour éviter cela, une lecture ne peut se faire que lorsqu'un verrou S est posé sur les ressources. Ce verrou étant incompatible avec le verrou X, il doit attendre la libération des verrous X des ressources souhaitées.

Les verrous de type **exclusifs** (*exclusive*, X) sont des verrous d'écriture, posés sur des lignes en cours de modification (`INSERT`, `DELETE`, `UPDATE`). Ils sont incompatibles avec tout type de verrou, y compris eux-mêmes : quelles que soient les options de la base ou de la session, il est interdit à deux transactions d'écrire simultanément les mêmes ressources. Si cela était possible, ce serait le chaos[1]. On ne peut poser ni verrou exclusif, ni verrou partagé, etc., sur une ressource qui possède déjà un verrou exclusif. On ne peut donc pas non plus lire une ligne en cours de modification, dans un niveau d'isolation qui pose un verrou partagé à la lecture.

Les verrous de **mise à jour** (*update*, U) sont posés dans les niveaux d'isolation qui maintiennent les verrous partagés toute la durée de la transaction (`REPEATABLE READ` et `SERIALIZABLE`), ils permettent d'éviter une forme courante de verrou mortel (*deadlock*), lorsque la lecture sert, dans une instruction suivante, aux mises à jour. Dans ce cas, le verrou partagé doit être converti en verrou exclusif. Mais, si une autre transaction maintient elle aussi des verrous partagés sur ces ressources, la première doit

1. Il existe sur certains SGBDR un niveau d'isolation de la transaction nommé CHAOS, qui permet les écritures simultanées. SQL Server ne le supporte pas. Le niveau d'isolation minimal est READ UNCOMMITTED, comme nous le verrons plus loin.

attendre leur libération. Lorsque la seconde transaction essaie à son tour de modifier les données, elle doit également attendre sur la première, d'où situation inextricable et *deadlock* (également nommé interblocage ou verrou mortel). Pour éviter cela, SQL Server pose également un verrou de mise à jour. Un seul verrou U peut-être posé dans le même temps, sur la même ressource. Cela permet à la transaction de convertir le verrou S en verrou X sans attendre, si elle possède le verrou U.

Les verrous **intentionnels** (*intent*, I) permettent de protéger les ressources aux granularités supérieures, contre un verrouillage incompatible. Si une transaction pose un verrou sur une ligne, il est nécessaire d'empêcher les autres transactions de poser des verrous sur la page qui contient cette ligne, ou sur la table, afin de ne pas se retrouver avec des verrous incompatibles par une voie détournée. Les verrous d'intention permettent d'optimiser les performances de verrouillage, en évitant la recherche des verrous de granularité plus fine. Si une transaction veut verrouiller la table, elle n'a pas besoin de chercher s'il existe des verrous de ligne ou de page, il lui suffit de vérifier la présence de verrous effectifs ou d'intention, sur la table.

Prenons un exemple, exécutons une requête de modification sur une table d'AdventureWorks :

```
USE AdventureWorks
GO

BEGIN TRAN

UPDATE TOP (1) Sales.Currency
SET ModifiedDate = current_timestamp
```

Examinons les verrous :

```
SELECT
   tl.resource_type,
   tl.resource_subtype,
   DB_NAME(tl.resource_database_id) as db,
   tl.resource_description,
   CASE tl.resource_type
      WHEN 'OBJECT' THEN OBJECT_NAME(tl.resource_associated_entity_id)
      ELSE COALESCE(OBJECT_NAME(p.object_id),
         CAST(tl.resource_associated_entity_id as sysname))
   END as obj,
   tl.request_mode,
   tl.request_status
FROM sys.dm_tran_locks tl
LEFT JOIN sys.partitions p
   ON tl.resource_associated_entity_id = p.hobt_id
WHERE
   tl.request_session_id = @@SPID AND
   tl.resource_database_id = DB_ID()
ORDER BY
   CASE resource_type
      WHEN 'KEY' THEN 1
      WHEN 'RID' THEN 1
      WHEN 'PAGE' THEN 2
```

```
        WHEN 'EXTENT' THEN 3
        WHEN 'HOBT' THEN 4
        WHEN 'ALLOCATION_UNIT' THEN 5
        WHEN 'OBJECT' THEN 6
        WHEN 'DATABASE' THEN 7
        WHEN 'FILE' THEN 8
        ELSE 9
    END
```

Le résultat est visible sur la figure 7.1. Nous voyons que des verrous d'intention exclusifs ont été placés sur la page, et sur la table.

resource_type	resource_subtype	db	resource_description	obj	request_mode	request_status
KEY		AdventureWorks	(820030fe24dd)	Currency	X	GRANT
PAGE		AdventureWorks	1:295	Currency	IX	GRANT
OBJECT		AdventureWorks		Currency	IX	GRANT
DATABASE		AdventureWorks		0	S	GRANT
METADATA	SCHEMA	AdventureWorks	schema_id = 9	0	Sch-S	GRANT

Figure 7.1 — Verrous

Les verrous intentionnels comportent le mode du verrou originel : IS pour *Intent Shared*, IX pour *Intent Exclusive*, etc.

Les verrous de **schéma** (*schema*, Sch-M) sont des verrous sur les structures. Il serait désagréable de voir la table dans laquelle on est en train de lire, supprimée par une autre transaction.

Les verrous d'**étendue** (*range*, RangeM-M) sont des verrous sur les ressources qui « entourent » la ressource verrouillée. Elles permettent de prévenir les lectures fantômes en niveau d'isolation `SERIALIZABLE`. Nous en parlerons quand nous présenterons les niveaux d'isolation.

Nous pouvons observer le verrouillage à travers plusieurs vues de gestion dynamiques que nous présenterons au long de ce chapitre. La vue principale affichant les verrous en cours est `sys.dm_tran_locks`.

À propos des latches

Vous verrez parfois, par exemple dans les sessions en attente (en exécutant la vue `sys.dm_os_waiting_tasks`, notamment), passer le mot *latch*, par exemple avec des attentes de type `PAGEIOLATCH`.

Les *latches*, traduits en français dans la documentation par verrous internes ou verrous de ressources internes, sont des verrous de synchronisation, légers, rapides et posés habituellement durant un temps très court, destinés à protéger les structures physiques lors de leur accès par le moteur de stockage et leur transmission au moteur

relationnel. Lorsqu'une ligne est lue, par exemple, SQL Server doit protéger non seulement la ligne, mais également la page (pour éviter qu'une insertion ne modifie l'en-tête qui indique *l'offset* de la ligne). Les *latches* ne sont maintenus que sur des pages de données présentes dans le *buffer* (alors que des verrous peuvent rester posés sur des pages sur le disque), car toute page en cours d'utilisation physique est placée dans le *buffer*. Cela aide ainsi à garantir une grande rapidité du *latch*.

> Il existe aussi d'autres types de *latches* non-*buffer*, sur des structures internes.

Vous pouvez obtenir des statistiques sur la gestion des *latches* avec l'objet de compteur de performances `SQL:Latches`.

Les *latches* d'entrées/sorties sont des types particuliers de *latches* de *buffer*, utilisés pour la synchronisation entre les pages du *buffer* et les pages du disque, on les voit dans les attentes de type `PAGEIOLATCH`. Ces attentes sont souvent typiques des latences du disque. Vous pouvez utiliser une requête comme celle-ci pour détecter ces attentes :

```
SELECT wait_type,
       waiting_tasks_count,
       wait_time_ms
FROM sys.dm_os_wait_stats
WHERE wait_type like 'PAGEIOLATCH%'
ORDER BY wait_type
```

7.2.2 Granularité de verrouillage

Les verrous sont posés sur des ressources. Les verrous de données (il existe des verrous posés sur des objets de code, pour éviter leur modification durant l'exécution) peuvent concerner les ressources suivantes : la ligne (ou la clé d'index), la page, la table, et la base de données. On parle de granularité pour indiquer à quel niveau de ressource s'effectue le verrouillage. En SQL Server, la granularité par défaut est la ligne, ou la clé d'index. Cela permet de verrouiller le moins de ressources possible et offre donc un excellent niveau de concurrence. Par exemple, lors d'insertions, il prévient la contention sur la page qui se produit sur les tables *clustered* dans les SGBDR dont la granularité minimale est la page.

Cette **granularité** comporte néanmoins un désavantage : si la commande affecte un grand nombre de lignes, cela implique de poser un grand nombre de verrous. Chaque verrou coûte du temps processeur et de la mémoire vive. Pour limiter les coûts, SQL Server évalue le nombre de lignes à traiter, et lors de l'exécution, peut convertir des verrous en granularité plus large. Pour saler le contenu de votre assiette, vous prenez une pincée. Si vous devez saler la marmite d'un régiment, vous versez à partir d'un grand sachet de sel, car le faire pincée par pincée vous prendrait des heures.

Par exemple, au lieu de poser 19 972 verrous sur la table `Person.Contact` pour modifier toutes les lignes, il ne va poser qu'un seul verrou, sur la table. Cette conversion s'appelle l'escalade. Elle fait l'économie d'un verrouillage intense, mais elle bloque aussi potentiellement plus de ressources que nécessaire, et donc diminue la concurrence d'accès.

Escalade de verrous

Vous pouvez obtenir des informations précises sur le verrouillage de vos index par la fonction `sys.dm_db_index_operational_stats()`. Notamment, la colonne `index_lock_promotion_count` indique le nombre d'escalades :

```
SELECT
    OBJECT_NAME(ios.object_id) as table_name,
    i.name as index_name,
    i.type_desc as index_type,
    row_lock_count,
    row_lock_wait_count,
    row_lock_wait_in_ms,
    page_lock_count,
    page_lock_wait_count,
    page_lock_wait_in_ms,
    index_lock_promotion_attempt_count,
    index_lock_promotion_count
FROM sys.dm_db_index_operational_stats(
    DB_ID('AdventureWorks'),
    OBJECT_ID('Person.Contact'), NULL, NULL) ios
JOIN sys.indexes i ON ios.object_id = i.object_id
                    AND ios.index_id = i.index_id
ORDER BY ios.index_id;
```

Parfois, vous voudrez contrôler le niveau de granularité, soit en l'augmentant pour augmenter les performances (c'est très rarement nécessaire), soit pour empêcher la pose de verrous aux granularités supérieures, qui bloquent d'autres transactions. La deuxième option est plus courante, dans des cas de mises à jour d'une grande partie ou de la totalité d'une table, par exemple. L'opération prend du temps, et si un verrou est posé sur la table, il bloque tout autre travail sur celle-ci.

Augmentation de la granularité

Nous verrons dans la section 8.2.1 que nous pouvons contrôler la granularité de verrouillage par table. Nous avons aussi la possibilité de contrôler le verrouillage de tous les accès aux tables *clustered* à travers les options de création d'index `ALLOW_ROW_LOCKS` et `ALLOW_PAGE_LOCKS`. Nous les avons survolés en section 6.1.3. `ALLOW_ROW_LOCKS = OFF` désactive le verrouillage de lignes, `ALLOW_PAGE_LOCKS = OFF` désactive le verrouillage de page. Appliqués sur un index *clustered*, ils gèrent effectivement la granularité de verrouillage de la table. Ils sont très rarement utiles. Vous pouvez les essayer avec l'exemple de code suivant, qui présente deux requêtes utiles : la première retourne les index dont ces options ont été modifiées, la seconde permet de tracer les verrous posés par un spid à l'aide des vues de gestion dynamique :

```
SELECT
    OBJECT_NAME(object_id) as table_name,
    name as index_name,
    allow_row_locks,
    allow_page_locks
FROM sys.indexes
WHERE allow_row_locks & allow_page_locks = 0
ORDER BY table_name, index_name;

ALTER INDEX PK_Contact_ContactID
  ON Person.Contact
  SET (ALLOW_ROW_LOCKS = OFF, ALLOW_PAGE_LOCKS = OFF);

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
BEGIN TRANSACTION
SELECT LastName, FirstName, EmailAddress
FROM Person.Contact
WHERE LastName = 'Adams';

SELECT
    CASE es.transaction_isolation_level
        WHEN 0 THEN 'non spécifié'
        WHEN 1 THEN 'READ UNCOMMITTED'
        WHEN 2 THEN 'READ COMMITTED'
        WHEN 3 THEN 'REPEATABLE'
        WHEN 4 THEN 'SERIALIZABLE'
        WHEN 5 THEN 'SNAPSHOT'
    END as transaction_isolation_level,
    tl.request_session_id as spid, tl.resource_type, tl.request_mode,
    tl.request_type,
    CASE
        WHEN tl.resource_type = 'object' THEN
            OBJECT_NAME(tl.resource_associated_entity_id)
        WHEN tl.resource_type = 'database' THEN
            DB_NAME(tl.resource_associated_entity_id)
        WHEN tl.resource_type IN ('key','page') THEN
            (SELECT object_name(i.object_id) + '.' + i.name
             FROM sys.partitions p
             JOIN sys.indexes i ON p.object_id = i.object_id
             AND p.index_id = i.index_id
             WHERE p.hobt_id = tl.resource_associated_entity_id)
        ELSE CAST(tl.resource_associated_entity_id as varchar(20))
END as objet
FROM sys.dm_tran_locks tl
LEFT JOIN sys.dm_exec_sessions es
        ON tl.request_session_id = es.session_id
WHERE request_session_id = @@spid
ORDER BY
    CASE resource_type
        WHEN 'DATABASE' THEN 10
        WHEN 'METADATA' THEN 20
        WHEN 'OBJECT' THEN 30
        WHEN 'PAGE' THEN 40
        WHEN 'KEY' THEN 50
```

```
        ELSE 100
    END

ROLLBACK

ALTER INDEX PK_Contact_ContactID
   ON Person.Contact
   SET (ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON);
```

En modifiant les options d'index, vous observerez les différences de verrouillage.

Diminution de la granularité

Vous pourriez parfois avoir besoin d'empêcher l'escalade de verrous, notamment sur les requêtes de modification qui touchent une grande partie de la table. Vous avez pour cela deux moyens. Le plus propre est de découper votre requête de modification en étapes, pour ne toucher qu'une partie de la table à chaque fois. À l'aide de la requête listée plus haut, vous pouvez voir la différence de verrouillage de ces deux instructions :

```
BEGIN TRAN
UPDATE Person.Contact
SET LastName = UPPER(LastName);

UPDATE TOP (1000) Person.Contact
SET LastName = UPPER(LastName);
ROLLBACK
```

La première pose un verrou exclusif sur la table, et un `UPDATE Person.Contact WITH (ROWLOCK)` n'y change rien, car il n'affecte que le plan d'exécution, pas le comportement d'escalade lors de l'exécution même. La seconde, par contre, pose des verrous sur les lignes (chaque clé de l'index *clustered*. Ainsi, vous pourriez créer une boucle `WHILE` de ce type :

```
DECLARE @incr int, @rowcnt int

SET @incr = 1
SET @rowcnt = 1

WHILE @rowcnt > 0 BEGIN
    UPDATE TOP (1000) Person.Contact
    SET LastName = UPPER(LastName)
    WHERE ContactID >= @incr

    SET @rowcnt = @@ROWCOUNT
    SET @incr = @incr + 1000
END
```

Une seconde solution, relevant plus du bricolage, existe. Elle se base sur le fait qu'un verrou d'intention posé à un niveau de granularité supérieur n'empêche pas les verrous de granularité inférieure, et prévient bien sûr la pose de verrous incompatibles. Vous pouvez poser un verrou d'intention exclusif sur la table, par exemple de cette manière :

```
BEGIN TRAN
SELECT TOP (0) * FROM Person.Contact WITH (UPDLOCK, HOLDLOCK)
```

Le verrou reste posé jusqu'au `ROLLBACK` ou `COMMIT` de la transaction. Vous pourriez créer un travail de l'agent qui lance la commande, avec un `WAITFOR` pour limiter la durée de la transaction (attention à la taille du journal de transactions !). Tant que cette transaction est ouverte, les autres transactions ne pourront pas escalader leurs verrous sur la table. Elles essaieront régulièrement durant l'exécution, et vous verrez leurs essais dans la colonne `index_lock_promotion_attempt_count` de la fonction `sys.dm_db_index_operational_stats()`.

Les lectures et écritures se feront sans problème.

Vous pouvez totalement désactiver l'escalade sur votre serveur à l'aide du drapeau de trace 1211 : `DBCC TRACEON (1211, -1)` ou `-T1211` au démarrage de l'instance. C'est une solution vraiment déconseillée, vous verrez sans aucun doute une baisse générale de performances due à un nombre de verrous excessif.

Vous retrouvez des conseils précis dans l'article de la base de connaissance 323630 : http://support.microsoft.com/kb/323630

Configurer l'escalade de verrous sur les tables partitionnées, en SQL Server 2008

En SQL Server 2005, il n'existe pas de granularité de verrou par partition. L'escalade ne peut se faire que de la page à la table, même si la requête n'a besoin d'accéder qu'à une seule partition. En SQL Server 2008, la partition devient une granularité de verrouillage possible. Vous pouvez gérer finement cette possibilité table par table si vous le souhaitez. La commande `ALTER TABLE` a été enrichie de l'option : `SET ( LOCK_ESCALATION = { AUTO | TABLE | DISABLE } )`. Voici un exemple de syntaxe :

```
ALTER TABLE dbo.matable SET (LOCK_ESCALATION = AUTO)
```

AUTO – permet la sélection automatique de la granularité des verrous. Sur une table partitionnée, l'escalade pourra se produire sur la partition. Sur une table non partitionnée, l'escalade se produira sur la table.

TABLE – l'escalade se fera sur la table, qu'elle soit partitionnée ou non. Il s'agit de la valeur par défaut, et correspondant au comportement de SQL Server 2005.

DISABLE – prévient l'escalade dans la plupart des cas. Le verrouillage de table n'est pas complètement désactivé.

7.3 NIVEAUX D'ISOLATION DE LA TRANSACTION

Nous l'avons vu, l'isolation de la transaction, à travers le mécanisme de verrouillage, a une forte influence sur la cohérence des données. Comme toute application multi-utilisateurs, SQL Server doit équilibrer son comportement entre le maintien de cette cohérence, et le besoin de permettre des accès concurrents. Ce sont deux exigences

opposées. Plus la cohérence est garantie, plus le nombre possible d'accès concurrents est faible, et vice-versa. Cela a donc fort un impact sur les performances. Afin de contrôler cet équilibre, nous pouvons agir sur l'isolation, en modifiant le **niveau d'isolation** de la transaction, dans le contexte de la session, ou par table requêtée.

La norme SQL définit quatre niveaux d'isolation, qui répondent à quatre problématiques connues de la cohérence transactionnelle : les mises à jour perdues, les lectures sales, les lectures non renouvelables, et les lectures fantômes.

La **mise à jour perdue** (*lost update*) se produit lorsque deux transactions lisent puis modifient simultanément les mêmes données (les mêmes lignes) à partir de cette lecture. La dernière mise à jour étant celle qui est prise en compte, des modifications appliquées par les transactions précédentes peuvent être écrasées.

La **lecture sale** (*dirty read*) décrit le cas où une ligne est lue alors qu'elle est en cours de modification par une autre transaction, soit parce que l'instruction est en cours d'exécution, soit parce que la transaction qui a modifié la ligne n'est pas encore validée (ce qui selon le principe de l'atomicité de la transaction est strictement équivalent). Le risque encouru est donc d'obtenir des informations incomplètes ou non valides.

La **lecture non renouvelable** (*non-repeatable read*) indique la possibilité, pour une transaction, de trouver des valeurs différentes dans les lignes lues à deux moments de la transaction, parce qu'une autre transaction aura modifié les données entre les deux lectures. C'est un danger pour les transactions qui comptent sur la cohérence de ces données dans la durée de vie de la transaction pour leur traitement.

La **lecture fantôme** (*phantom read*) est forme aggravée de la lecture non répétable. Elle décrit l'apparition de nouvelles lignes lors de nouvelles lectures, lorsque dans la même transaction, un deuxième `SELECT` portant sur les mêmes lignes (la même clause `WHERE`), voit apparaître de nouvelles lignes, parce qu'une autre transaction aura inséré des données répondant à la clause de filtre.

Répondre à ces différents problèmes implique, pour une transaction, de choisir différentes stratégies de verrouillage. Ces stratégies sont dictées par le niveau d'isolation de la transaction. La norme SQL décrit quatre niveaux d'isolation, qui répondent chacun à un problème transactionnel. Ce sont, dans l'ordre croissant de cohérence :

- READ UNCOMMITTED ;
- READ COMMITTED ;
- REPEATABLE READ ;
- SERIALIZABLE.

SQL Server, à partir de la version 2005, ajoute un niveau d'isolation propriétaire, basé sur le *row versioning*, appelé SNAPSHOT.

Ces niveaux d'isolation permettent de choisir un équilibre entre la cohérence des données et la concurrence d'accès : plus la cohérence transactionnelle est maintenue par un verrouillage fort, plus les données sont isolées des autres transactions, et donc moins la concurrence d'accès simultanée entre sessions est possible.

Le niveau d'isolation peut être changé pour la session à l'aide de l'instruction `SET TRANCATION ISOLATION LEVEL` comme nous le verrons ci-après. Il peut aussi être configuré dans une même requête, table par table, à l'aide d'un indicateur de table, tout comme on peut choisir une granularité de verrou :

```
SELECT * FROM Person.Contact WITH (READUNCOMMITTED);
-- ou
SELECT * FROM Person.Contact WITH (NOLOCK);
```

Les deux syntaxes précédentes sont strictement équivalentes. `NOLOCK` est simplement un alias propre à SQL Server pour indiquer un niveau `READ UNCOMMITTED`. Mais voyons ces niveaux.

Le niveau d'isolation **`READ UNCOMMITTED`** est le niveau le plus permissif, posant le moins de verrou, et permettant donc l'apparition du plus grand nombre d'erreurs transactionnelles. Dans ce niveau, les lectures sales (*dirty reads*) sont possibles : une transaction peut lire des lignes qui sont en train d'être modifiées par une autre transaction, ou modifier des lignes en cours de lecture. L'extraction de données non validées (« sales ») est donc possible. Dans ce niveau, SQL Server continue à poser des verrous exclusifs (X) lors de modifications : il est toujours impossible d'écrire en même temps les mêmes lignes. Par contre, en lecture, aucun verrou n'est posé. Seul un verrou de schéma (Sch-S) est maintenu, empêchant la modification de la structure des tables lues. De même, les verrous exclusifs ne sont pas honorés par la lecture. Ce niveau est donc très favorable aux performances, en diminuant les opérations de verrouillage, et les attentes de libération de verrous.

Vous entendrez certainement des mises en garde contre le niveau d'isolation `READ UNCOMMITTED`. Les lectures sales inquiètent certaines personnes. Dans la pratique, pour des générations de rapport, de l'extraction de données ou du simple affichage, il est rare de se préoccuper d'obtenir une ligne validée ou non. À un centième de secondes près, vous obtenez une valeur différente, quelle importance ? Compte tenu du gain de performances apporté par ce mode, nous ne pouvons que vous le conseiller pour des requêtes de lecture. La vraie contre-indication se présente dans un environnement où des transactions modifient plusieurs lignes en rapport les unes avec les autres, dans des tables différentes ou dans la même table. Exemple classique, si vous gérez une application bancaire et que vous effectuez par `SELECT` un total des avoirs d'un client, vous risquez, en niveau `READ UNCOMMITTED`, de lire des totaux erronés si en même temps une transaction retire un montant d'un compte courant pour le poser sur un compte d'épargne. Si le `SELECT` intervient lorsque la ligne est supprimée du compte courant, mais pas encore ajoutée au compte d'épargne, le total sera inférieur au crédit du client. Dans ce genre de cas, le niveau d'isolation `SNAPSHOT` peut permettre une plus grande concurrence, sans ralentir les requêtes par des attentes de verrous. Lorsque ce risque n'est pas présent, un `SET TRANSACTION ISOLATION`

`LEVEL READ UNCOMMITTED` en début de procédure stockée dédiée à la lecture ou au reporting, est une façon simple d'augmenter la concurrence et les performances. N'oubliez pas de laisser la session dans son niveau par défaut à la fin de la procédure, avec un `SET TRANSACTION ISOLATION LEVEL READ COMMITTED`.

Le niveau d'isolation **`READ COMMITTED`** est le niveau par défaut d'une session SQL Server. Il protège la transaction contre les lectures sales, en ne permettant des lectures que sur les lignes validées. Pour cela, SQL Server pose un verrou partagé sur les données en cours de lecture. Les verrous partagés sont compatibles entre eux (ce qui veut dire qu'on peut en placer plusieurs sur la même ressource), mais incompatibles avec les verrous exclusifs. Un verrou S ne pourra donc pas être placé sur une ligne déjà dotée d'un verrou X. La lecture d'une ligne en cours de modification devra donc attendre. Réciproquement, une tentative d'écriture devra attendre la libération de tous les verrous partagés sur la ressource. Ce niveau est un peu plus contraignant que le premier, et limite un peu plus les performances, en générant des verrous partagés, et des attentes en lecture. Par contre, il ne protège pas des lectures non renouvelables, car les verrous partagés sont libérés dès la fin de l'instruction de lecture, et ne persistent pas jusqu'à la fin de la transaction.

Le niveau d'isolation **`REPEATABLE READ`**, comme son nom l'indique, permet les lectures renouvelables, en maintenant les verrous partagés jusqu'à la fin de la transaction. Faisons le test :

```
USE AdventureWorks
GO

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ

BEGIN TRAN

SELECT TOP 10 * FROM Sales.Currency
```

En exécutant notre requête sur la vue `sys.dm_tran_locks`, nous trouvons le résultat de la figure 7.2.

resource_type	resource_subtype	db	resource_description	obj	request_mode	request_status
KEY		AdventureWorks	(850004f2ffe2)	Currency	S	GRANT
KEY		AdventureWorks	(8500748d5277)	Currency	S	GRANT
KEY		AdventureWorks	(88004c69db63)	Currency	S	GRANT
KEY		AdventureWorks	(8d0019603707)	Currency	S	GRANT
KEY		AdventureWorks	(820030fe24dd)	Currency	S	GRANT
KEY		AdventureWorks	(88005971ca4e)	Currency	S	GRANT
KEY		AdventureWorks	(9400f711c05f)	Currency	S	GRANT
KEY		AdventureWorks	(94002b4eab7a)	Currency	S	GRANT
KEY		AdventureWorks	(8200bab12ca0)	Currency	S	GRANT
KEY		AdventureWorks	(85009ba5e6b2)	Currency	S	GRANT
PAGE		AdventureWorks	1:295	Currency	IS	GRANT
OBJECT		AdventureWorks		Currency	IS	GRANT
DATABASE		AdventureWorks		0	S	GRANT
METADATA	SCHEMA	AdventureWorks	schema_id = 9	0	Sch-S	GRANT

Figure 7.2 — Verrous partagés en `REPEATABLE READ`

Des verrous partagés (S) sont maintenus sur chaque clé de l'index *clustered* (`KEY`), c'est-à-dire sur chaque ligne impactée dans la requête. Ces lignes ne pourront donc pas être modifiées tant que la transaction est encore active. Nous voyons également que des verrous d'intention partagés sont maintenus sur la page, et sur la table (`OBJECT`).

Ainsi, aucune autre session ne pourra modifier ces lignes jusqu'à la fin de la transaction. À cause du verrouillage plus important, n'utilisez ce mode que lorsque vous souhaitez, dans une transaction, utiliser le résultat de votre lecture pour des mises à jour dans les instructions suivantes. Évitez également d'écrire du code transactionnel qui s'exécute dans une logique inverse (par exemple `SELECT` puis `UPDATE` dans une transaction, `UPDATE` puis `SELECT` dans une autre), car vous vous exposeriez à un risque important de *deadlock*. Lorsque vous le faites dans le même ordre, SQL Server pose des verrous de mise à jour potentielle (update, U) pour éviter le problème.

Le niveau d'isolation **`SERIALIZABLE`** représente la protection maximale contre les erreurs transactionnelles, mais la concurrence d'accès minimale. Il protège la transaction contre les lectures fantômes, c'est-à-dire l'apparition de lignes supplémentaires lors de plusieurs lectures des mêmes données. Pour cela, SQL Server doit verrouiller non seulement les lignes lues, mais également les « alentours » de la ligne, pour empêcher l'insertion de données correspondant au filtre de la clause `WHERE`. La nature du verrouillage dépend de la structure de la table. Si la clause `WHERE` correspond à une recherche sur une colonne avec contrainte d'unicité (clé primaire, clé unique, index unique), aucun verrou supplémentaire ne sera posé, puisqu'aucune ligne ne pourra être ajoutée qui correspondra à la clause `WHERE`. Par contre, dans les autres cas, SQL Server posera soit un verrou d'étendue de clés (*key range*), soit, lorsque nécessaire, un verrou sur la table entière. Les verrous d'étendues de clés ne se rencontrent qu'en niveau `SERIALIZABLE`, et ne s'appliquent qu'aux index (index *clus-*

tered et *nonclustered*). Ils ont deux composants : une partie étendue, et une partie ligne. Leur mode, visible par exemple dans la colonne `request_mode` de la vue `sys.dm_tran_locks`, est écrit de cette façon : « étendue-ligne ». Par exemple, un verrou de ressource partagé et d'étendue partagée sera écrit « RangeS-S », un verrou de ressource exclusif avec verrou d'étendue exclusif, « RangeX-X ». La liste est disponible dans les BOL, sous l'entrée « Verrouillage d'étendues de clés » (*Key-Range Locking*).

Sur une table sans index *clustered*, et dont les colonnes de la clause `WHERE` ne sont pas indexées, SQL Server sera obligé de poser un verrou de table. Démonstration :

```
USE tempdb
GO

CREATE TABLE dbo.testSerializable
    (nombre int, texte varchar(50) COLLATE French_BIN)
GO

INSERT INTO dbo.testSerializable VALUES (1, 'un')
INSERT INTO dbo.testSerializable VALUES (2, 'deux')
INSERT INTO dbo.testSerializable VALUES (3, 'trois')
INSERT INTO dbo.testSerializable VALUES (4, 'quatre')
GO

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE

BEGIN TRAN

-- sans index
UPDATE dbo.testSerializable
SET texte = 'DEUX'
WHERE nombre = 2

-- requête sur sys.dm_tran_locks

ROLLBACK
GO

CREATE NONCLUSTERED INDEX nix$testSerializable$nombre
ON dbo.testSerializable (nombre)
GO

BEGIN TRAN

-- avec index
UPDATE dbo.testSerializable
SET texte = 'DEUX'
WHERE nombre = 2

-- requête sur sys.dm_tran_locks

ROLLBACK
GO
```

Les deux requêtes sur la vue `sys.dm_tran_locks` ne sont pas incluses dans le code pour des raisons de place, vous les connaissez. Leur résultat est visible dans la figure 7.3 pour le premier cas (*heap* sans index), et dans la figure 7.4 pour le second cas (*heap* avec index *nonclustered*).

request_session_id	resource_type	resource_subtype	db	resource_description	obj	request_mode	request_status
54	OBJECT		tempdb		testSerializable	X	GRANT

Figure 7.3 — Verrous sur un *heap*

request_session_id	resource_type	resource_subtype	db	resource_description	obj	request_mode	request_status
54	KEY		tempdb	(5200583e2945)	testSerializable	RangeS-U	GRANT
54	KEY		tempdb	(5100f421a1f5)	testSerializable	RangeS-U	GRANT
54	RID		tempdb	1:78:1	testSerializable	X	GRANT
54	PAGE		tempdb	1:78	testSerializable	IX	GRANT
54	PAGE		tempdb	1:45	testSerializable	IU	GRANT
54	OBJECT		tempdb		testSerializable	IX	GRANT

Figure 7.4 — Verrous sur une table *clustered*

Vous pouvez voir que dans le premier cas SQL Server n'a d'autre choix que de poser un verrou exclusif sur la table. Par contre, lorsqu'un index existe, il peut poser des verrous d'étendues sur les clés de l'index qui devraient être accédées si se produit un ajout de ligne répondant à la clause `nombre = 2`. Les verrous sont des RangeS-U : partagé sur l'étendue, mise à jour sur la ressource. Un *Intent Update* est posé sur la page de l'index.

N'utilisez le niveau `SERIALIZABLE` qu'en cas d'absolue nécessité (changement ou calcul de clé, par exemple), et n'y restez que le temps nécessaire. Le verrouillage important entraîne une réelle baisse de concurrence.

Le niveau d'isolation **`SNAPSHOT`** existe depuis SQL Server 2005. Il permet de diminuer le verrouillage, tout en offrant une cohérence de lecture dans la transaction. Alors que dans le niveau `READ UNCOMMITTED`, des lectures sales restent possibles, le niveau `SNAPSHOT` permet de lire les données en cours de modification, dans l'état cohérent dans lequel elles se trouvaient *avant* modification. Pour cela, SQL Server utilise une fonctionnalité interne appelée version de lignes (*row versioning*). Pour pouvoir l'utiliser, le support du niveau d'isolation `SNAPSHOT` doit avoir été activé dans la base de données. Exemple :

```
ALTER DATABASE sandbox SET ALLOW_SNAPSHOT_ISOLATION ON
```

Quand ceci est fait, toutes les transactions qui modifient des données dans cette base, maintiennent une copie (une version) de ces données avant modification, dans `tempdb`, quel que soit leur niveau d'isolation. Si une autre session en niveau d'isolation `SNAPSHOT` essaie de lire ces lignes, elle lira la copie des lignes, et non pas

les lignes en cours de modification, ce qui lui garantit une lecture cohérente de l'état de ces données, avant modification. Expérimentons-le :

```
USE sandbox;
GO

CREATE TABLE dbo.testSnapshot
    (nombre int, texte varchar(50) COLLATE French_BIN);
GO

INSERT INTO dbo.testSnapshot VALUES (1, 'un');
INSERT INTO dbo.testSnapshot VALUES (2, 'deux');
INSERT INTO dbo.testSnapshot VALUES (3, 'trois');
INSERT INTO dbo.testSnapshot VALUES (4, 'quatre');
GO

-- session 1
BEGIN TRAN;

UPDATE dbo.testSnapshot
SET texte = 'DEUX'
WHERE nombre = 2;

-- session 2
SET TRANSACTION ISOLATION LEVEL SNAPSHOT;

SELECT *
FROM dbo.testSnapshot
WHERE nombre = 2;
```

La deuxième session lit sans problème la ligne. La valeur retournée pour la colonne texte est `'deux'`, et non `'DEUX'`, comme ce serait le cas dans une lecture en niveau d'isolation `READ UNCOMMITTED` : la valeur lue est la version de la ligne précédant la modification, par la ligne en cours de modification.

Vous pouvez voir les versions de lignes en cours à l'aide des vues de gestion dynamique `sys.dm tran active snapshot database transactions` et `sys.dm_tran_version_store`. `sys.dm tran active snapshot database transactions` donne le spid (`session_id`) et la durée en secondes depuis la création de la version (`elapsed_time_seconds`), `sys.dm_tran_version_store` vous retourne une ligne par version de ligne, avec l'indication de la base de données (`database_id`) source. Attention aux requêtes sur `sys.dm_tran_version_store` : elles peuvent impacter les performances, compte tenu de l'importante quantité de résultats qu'elles peuvent retourner.

Une lecture dans ce niveau génère aussi une version :

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT;

BEGIN TRAN;

SELECT *
FROM dbo.testSnapshot
```

```
WHERE nombre = 2;

SELECT session_id, transaction_id, elapsed_time_seconds
FROM sys.dm_tran_active_snapshot_database_transactions;

ROLLBACK;
```

Pourquoi cela ? Afin de permettre, dans la même transaction, d'éviter les lectures non renouvelables et les lectures fantômes. De fait, ce niveau d'isolation correspond à un niveau SERIALIZABLE, sans le verrouillage. Une requête dans sys.dm_tran_locks ne montre aucun verrou maintenu. Les autres transactions pourront donc modifier la ligne WHERE nombre = 2 à loisir. Durant toute la durée de la transaction, notre session en isolation SNAPSHOT verra la version de ligne équivalant à la première lecture.

Mais que se passera-t-il si nous cherchons à modifier ensuite la ligne dans notre transaction ? Imaginons qu'après avoir exécuté le SELECT précédent, une autre session a mis à jour la ligne de dbo.testSnapshot, WHERE nombre = 2. Pourrons-nous à notre tour effectuer un UPDATE, basé sur une version de ligne précédant la mise à jour externe ? Non. Au moment de l'UPDATE dans notre transaction, un conflit sera détecté, et une erreur sera retournée : la cohérence des mises à jour est toujours garantie, pour nous protéger contre les *lost updates*. Le mode d'isolation SNAPSHOT est donc à proprement parler un mode de verrouillage optimiste : les données lues sont conservées sans verrouillage, avec la supposition qu'il y a peu de probabilité qu'un conflit apparaisse.

Compteurs de performances

Ces compteurs vous donnent une visibilité sur les statistiques du niveau d'isolation snapshot : MSSQL:Transactions\Longest Transaction Running Time, MSSQL:Transactions\Snapshot Transactions, MSSQL:Transactions\Update conflict ratio.

Identification du niveau d'isolation

Vous pourriez avoir envie de savoir dans quel niveau d'isolation se trouve votre session, ou une autre session en cours d'exécution. En SQL Server 2000, la commande DBCC USEROPTIONS était votre seul salut. Cette commande retourne un jeu de résultats comportant deux colonnes : [Set Option] et Value. Pour la manipuler en programmation, vous deviez insérer le résultat dans une table temporaire, en l'appelant dans du SQL dynamique. Par exemple :

```
CREATE TABLE #setoption (so varchar(64), val varchar(64))

INSERT INTO #setoption (so, val)
EXEC ('DBCC USEROPTIONS')

SELECT val
FROM #setoption
WHERE so = 'isolation level'
```

Pour connaître le niveau d'isolation d'une autre session que la vôtre, vous deviez utiliser une commande DBCC non documentée : `DBCC PSS`, dont la fonction est de produire des informations de bas niveau sur les utilisateurs connectés, et les sessions. Depuis 2005, vous avez à disposition la vue de gestion dynamique `sys.dm_exec_sessions`. Par exemple, pour connaître le niveau d'isolation de votre propre session :

```
SELECT transaction_isolation_level,
   CASE transaction_isolation_level
      WHEN 0 THEN 'Unspecified'
      WHEN 1 THEN 'ReadUncomitted'
      WHEN 2 THEN 'Readcomitted'
      WHEN 3 THEN 'Repeatable'
      WHEN 4 THEN 'Serializable'
      WHEN 5 THEN 'Snapshot'
   END AS isolation_level_description
FROM sys.dm_exec_sessions
WHERE session_id = @@SPID
```

Mais l'intérêt dans le code est limité. Si vous voulez forcer votre niveau d'isolation, exécutez simplement un `SET TRANSACTION ISOLATION LEVEL`… quel que soit votre niveau actuel. Ce sera plus rapide que de tester le niveau auparavant.

7.4 ATTENTES

Comme l'isolation de la transaction empêche les accès concurrents non compatibles sur les mêmes ressources de verrouillage, cela veut dire qu'une transaction qui veut accéder à une ressource utilisée par une autre transaction doit attendre la libération des verrous non compatibles.

Par défaut, une transaction attend indéfiniment la libération de verrous. Vous pouvez limiter cette attente par l'option de session `LOCK_TIMEOUT` :

```
SET LOCK_TIMEOUT 1000;
SELECT @@LOCK_TIMEOUT as lock_timeout;
```

La valeur est exprimée en millisecondes. Le défaut est -1, c'est-à-dire sans limite. Vous pouvez aussi utiliser l'indicateur de table `READPAST` (voir la section 8.2.1).

Les attentes ne sont pas seulement dues au verrouillage mais peuvent être provoquées par les entrées/sorties, la mise en attente du processus par l'ordonnanceur, etc. Nous verrons ensuite les principaux types d'attentes.

Une attente peut aussi se produire sur l'attribution de mémoire pour la requête. Dans des situations de pression sur la mémoire, cela peut même provoquer des dépassements de délai (*timeouts*).

Vous obtenez des informations sur les attentes à travers les vues de gestion dynamique `sys.dm_tran_locks`, `sys.dm_os_wait_stats`, `sys.dm_os_waiting_tasks` et `sys.dm_exec_requests`.

La DMV `sys.dm_os_wait_stats` accumule les statistiques sur les attentes (*waits*). Pour chaque type d'attente (l'indication de la raison de l'attente d'un processus SQL Server), cette vue indique principalement le nombre d'attentes et leur durée totale en millisecondes. Les valeurs s'incrémentent au fur et à mesure de l'activité de l'instance, et elles sont remises à zéro – comme pour toutes les DMV – lors d'un redémarrage de celle-ci. Dans ce cas, vous pouvez aussi remettre explicitement les compteurs à zéro, par exemple avant de lancer un lot de requêtes test, à l'aide de cette commande DBCC :

```
dbcc sqlperf('sys.dm_os_wait_stats', clear)
```

Voyons en pratique une attente. Exécutons ces deux requêtes dans deux sessions différentes :

```
-- dans la session 1
BEGIN TRAN

UPDATE Sales.Currency
SET Name = 'Franc Francais'
WHERE CurrencyCode = 'EUR'

-- dans la session 2
BEGIN TRAN

SELECT *
FROM Sales.Currency WITH (TABLOCK)
WHERE CurrencyCode = 'CHF'
```

Que voyons-nous avec une requête sur `sys.dm_tran_locks` (légèrement modifiée pour afficher les deux spid en présence, vous trouverez cette requête sur le site d'accompagnement du livre) ? Réponse dans la figure 7.5.

	request_session_id	resource_type	resource_subtype	db	resource_description	obj	request_mode	request_status
1	52	KEY		AdventureWorks	(9700c505f685)	Currency	X	GRANT
2	52	KEY		AdventureWorks	(7d025fba3b3b)	Currency	X	GRANT
3	52	KEY		AdventureWorks	(9700b8e106b1)	Currency	X	GRANT
4	52	PAGE		AdventureWorks	1:2101	Currency	IX	GRANT
5	52	PAGE		AdventureWorks	1:295	Currency	IX	GRANT
6	52	OBJECT		AdventureWorks		Currency	IX	GRANT
7	52	DATABASE		AdventureWorks		0	S	GRANT
8	53	OBJECT		AdventureWorks		Currency	S	WAIT
9	53	DATABASE		AdventureWorks		0	S	GRANT

Figure 7.5 — Attentes

Nous voyons que le spid 53 cherche à poser un verrou partagé sur la table `Currency`. Le `request_status` est `WAIT`, ce qui indique que le verrou est en attente. Nous

avons introduit ici, pour l'exemple, une petite perversion : la deuxième requête ne serait pas en attente si l'indicateur de table `WITH (TABLOCK)` ne forçait un verrou à la granularité de table. Comme la transaction du spid 52 a posé un verrou d'intention exclusif sur la table `Currency`, le spid 53 ne peut poser de verrou partagé sur cet objet.

Au passage, nous voyons aussi, sur la figure 7.5, que deux pages sont verrouillées par le spid 52. Pourquoi deux pages, alors que nous devons modifier une seule ligne ? Vérifions :

```
DBCC TRACEON (3604)
DBCC PAGE (AdventureWorks, 1, 295, 0)
DBCC PAGE (AdventureWorks, 1, 2101, 0)
```

L'en-tête de la page 295 nous montre ceci : `Metadata: ObjectId = 597577167 … Metadata: IndexId = 1`, et l'en-tête de la page 2101 : `Metadata: ObjectId = 597577167 … Metadata: IndexId = 2`. Il s'agit bien sûr du verrouillage d'une page par index : l'index *clustered* de la table (`IndexId = 1`), et de l'index *nonclustered* sur la colonne `Name` (`IndexId = 2`), que nous sommes en train de modifier.

Types d'attentes

Pour référence, nous listons ici les types d'attentes que vous rencontrerez le plus souvent, et qui peuvent vous aider à détecter des problèmes. Ces attentes sont visibles dans la colonne `wait_type` des vues de gestion dynamique `sys.dm_os_wait_stats` et `sys.dm_os_waiting_tasks`.

> Une liste exhaustive est donnée dans l'excellent document « *SQL Server 2005 Waits and Queues* », à l'adresse http://www.microsoft.com/technet/prodtechnol/sql/bestpractice/performance_tuning_waits_queues.mspx

- `ASYNC_NETWORK_IO` – « bufferisation » d'une session TCP, souvent à cause d'un client qui met du temps à traiter l'envoi de SQL Server, ou qui maintient ouvert un jeu de résultat. Le compteur de performance `MSSQL:Wait statistics\Network IO waits` est aussi utile pour ce type de diagnostic.
- `CXPACKET` – échange entre des *threads* parallélisés sur la même requête. Peut indiquer un problème lié au parallélisme. Si vous avez beaucoup d'attentes de ce type, désactivez le parallélisme.
- `LCK_...` - attentes sur des verrous, voir l'encadré pour une surveillance par compteurs de performances.
- `LOGBUFFER` et `WRITELOG` – écriture et lecture des journaux de transaction.
- `LOGMGR_RESERVE_APPEND` – attente de troncature pour vérifier si la libération d'espace dans le journal sera suffisante pour écrire l'enregistrement actuel. C'est un signe de lenteur de disque, sur des journaux en mode de récupération simple.
- `PAGEIOLATCH` – attentes sur des synchronisations entre les pages du *buffer* et les pages du disque, bon indicateur de contention disque.

- SOS_SCHEDULER_YIELD – se produit lorsqu'une tâche s'interrompt volontairement pour redonner sa place, dans le modèle non prémptif de l'ordonnanceur SQLOS. Voir cette entrée de blog pour plus de détails : http://blogs.msdn.com/sqlcat/archive/2005/09/05/461199.aspx

Surveiller les attentes de verrous

Le compteur **MSSQL:Wait statistics\Lock waits** donne des informations importantes sur les attentes en cours sur le système. sur les instances :
– **Average wait time (ms)** : Temps d'attente moyen par type.
– **Cumulative wait time (ms) per second** : temps total d'attente dans la seconde ;
– **Waits in progress** : nombre de processus en cours d'attente.
– **Waits started per second** : nombre d'attentes démarrées par seconde.

7.5 BLOCAGES ET *DEADLOCKS*

Nous avons présenté les attentes de façon générale, et nous avons vu qu'il y a plusieurs types d'attentes. Les attentes spécifiques sur la libération de verrous par une autre transaction, sont appelées des **blocages** : l'attente est involontaire, et dépend d'une autre transaction. La transaction en attente détecte bien entendu cette situation, et elle se considère comme bloquée. Cette information de blocage est visible dans les vues de gestion dynamique `sys.dm_os_waiting_tasks` et `sys.dm_exec_requests` (la colonne `blocking_session_id` y indique le sipd qui bloque la requête en cours). Voici deux exemples de requêtes pour détecter les blocages :

```
SELECT session_id, wait_duration_ms, wait_type,
    blocking_session_id, resource_description
FROM sys.dm_os_waiting_tasks
WHERE blocking_session_id IS NOT NULL;

SELECT blocked.session_id, st.text as blocked_sql,
    blocker.session_id as blocker_session_id, blocker.last_request_end_time
FROM sys.dm_exec_requests blocked
JOIN sys.dm_exec_sessions blocker
    ON blocked.blocking_session_id = blocker.session_id
CROSS APPLY sys.dm_exec_sql_text(blocked.sql_handle) st;
```

Bien sûr, une session qui est bloquée par une première, peut elle aussi en bloquer une suivante, et ainsi de suite. Lorsque les verrous sont maintenus trop longtemps sur des ressources, il peut se produire ainsi des chaînes de blocage qui finissent par paralyser le serveur. Il est donc important de surveiller ces situations.

7.5.1 Détection des blocages par notification d'événements

Avant SQL Server 2005, le seul moyen pour surveiller et être averti de blocages survenant sur le serveur, était de planifier une vérification régulière de la table système `sysprocesses`. Cette approche est toujours possible. Vous pouvez placer

dans un travail de l'agent SQL planifié pour s'exécuter toutes les quelques minutes une requête comme celle-ci (adaptée à SQL Server 2005-2008) :

```
SELECT
      es2.session_id as spid_blocking,
      es1.session_id as spid_blocked,
      er1.start_time as blocked_start,
      er1.row_count,
      CASE er1.transaction_isolation_level
            WHEN 0 THEN 'Unspecified'
            WHEN 1 THEN 'ReadUncommitted'
            WHEN 2 THEN 'Readcommitted'
            WHEN 3 THEN 'RepeatableRead'
            WHEN 4 THEN 'Serializable'
            WHEN 5 THEN 'Snapshot'
      END as transaction_isolation_level,
      DB_NAME(er1.database_id) as db,
      est1.text as sql_command_blocked,
      er1.wait_type,
      es1.host_name as host_name_blocked,
      es1.program_name as program_name_blocked,
      es1.login_name as login_name_blocked,
      es2.host_name as host_name_blocking,
      es2.program_name as program_name_blocking,
      es2.login_name as login_name_blocking
FROM sys.dm_exec_requests er1
JOIN sys.dm_exec_sessions es1
      ON er1.session_id = es1.session_id
CROSS APPLY sys.dm_exec_sql_text(er1.sql_handle) est1
JOIN sys.dm_exec_sessions es2
      ON er1.blocking_session_id = es2.session_id
ORDER BY spid_blocking
```

qui envoie son résultat, s'il y en a un, par e-mail, à l'aide de Database Mail.

Il existe maintenant une méthode intégrée qui utilise la **notification d'événements** (*event notification*). Cette technologie est basée sur les événements DDL (*Data Definition Language*) disponibles depuis SQL Server 2005 notamment pour écrire des déclencheurs DDL, ainsi que sur Service Broker (voir ci-dessous). Elle permet de répondre à des événements en mode asynchrone, et d'envoyer ces événements aussi bien sur le serveur SQL local que sur une machine distante. L'événement que nous allons utiliser s'appelle `BLOCKED_PROCESS_REPORT`.

Cet événement doit être activé dans les options du serveur pour être déclenché lorsqu'un processus est bloqué depuis un certain temps.

```
EXEC sp_configure 'show advanced options', 1
RECONFIGURE
GO
EXEC sp_configure 'blocked process threshold', 30
RECONFIGURE
GO
EXEC sp_configure 'show advanced options', 0
RECONFIGURE
GO
```

La valeur de l'option « *blocked process threshold* » est exprimée en secondes. Elle indique à partir de quelle durée de blocage un événement sera déclenché : dans l'exemple précédent, après 30 secondes, et chaque 30 secondes suivantes, jusqu'à ce que le blocage soit libéré.

En général, un blocage en provoque d'autres à son tour, produisant une chaîne de processus bloqués les uns par les autres. L'événement `BLOCKED_PROCESS_REPORT` ne montre pas cette chaîne, il se produit pour chaque processus bloqué, individuellement.

Vous devez mettre en place la notification. La fonctionnalité d'*event notification* est basée sur Service Broker, un système orienté service d'échanges de messages intégré à SQL Server (un concept parfois nommé SODA : *Service Oriented Database Architecture*). La description de Service Broker sortant du cadre de cet ouvrage, nous nous contenterons de vous montrer comment mettre pratiquement en œuvre une notification d'événement. Vous devez, dans l'ordre, mettre en place une file d'attente Service Broker, un service, éventuellement une route pour indiquer où les messages seront dirigés. Dans l'exemple suivant, les noms d'objet (queue, service…) peuvent êtres modifiés, seul le nom du contrat ([http://schemas.microsoft.com/SQL/Notifications/PostEventNotification]) est fixé.

```
CREATE QUEUE NotifyQueue ;
GO

CREATE SERVICE NotifyService
ON QUEUE NotifyQueue
([http://schemas.microsoft.com/SQL/Notifications/PostEventNotification]);
GO

CREATE ROUTE NotifyRoute
WITH SERVICE_NAME = 'NotifyService',
ADDRESS = 'LOCAL';
GO

CREATE EVENT NOTIFICATION BlockedProcessReport
    ON SERVER
    WITH fan_in
FOR  BLOCKED_PROCESS_REPORT
TO SERVICE 'NotifyService', 'current database';
GO
```

L'information du blocage sera transmise par Service Broker sous forme d'un document XML qui sera placé dans la file d'attente `NotifyQueue`. Testons-le en créant une table puis en insérant une ligne dans une session après avoir démarré une transaction explicite. Dans une autre session, essayons de lire la table (dans un niveau d'isolation au moins équivalant à `READ COMMITTED`) :

```
USE sandbox
GO
```

```
CREATE TABLE dbo.testblocage ( id int )

BEGIN TRAN
INSERT dbo.testblocage ( id ) VALUES( 1 )
GO

--  dans une autre session
SELECT * FROM sandbox.dbo.testblocage
```

Le SELECT est mis en attente de libération du verrou exclusif posé par la transaction d'insertion. Au bout du temps déterminé par l'option 'blocked process threshold', la notification d'événement posera une ligne dans la file d'attente NotifyQueue. Nous pouvons simplement faire un SELECT sur cette queue :

```
SELECT CAST(message_body as XML) as msg
FROM NotifyQueue;
```

Le résultat est le suivant :

```
<EVENT_INSTANCE>
  <EventType>BLOCKED_PROCESS_REPORT</EventType>
  <PostTime>2008-05-15T13:16:40.797</PostTime>
  <SPID>4</SPID>
  <TextData>
    <blocked-process-report monitorLoop="22162">
      <blocked-process>
        <process id="process9a8c58" taskpriority="0" logused="0"
 waitresource="RID: 7:1:204:0" waittime="13843" ownerId="289041"
 transactionname="SELECT" lasttranstarted="2008-05-15T13:16:26.950"
 XDES="0x102331d0" lockMode="S" schedulerid="2" kpid="4224"
 status="suspended" spid="54" sbid="0" ecid="0" priority="0"
 transcount="0" lastbatchstarted="2008-05-15T13:16:26.950"
 lastbatchcompleted="2008-05-15T13:11:12.077"
 lastattention="2008-05-15T13:11:12.077"
 clientapp="Microsoft SQL Server Management Studio - Query"
 hostname="BABALUGA-XPS" hostpid="1804" loginname="BABALUGA-XPS\rudi"
 isolationlevel="read committed (2)" xactid="289041" currentdb="1"
 lockTimeout="4294967295" clientoption1="671090784"
 clientoption2="390200">
          <executionStack>
            <frame line="1"
sqlhandle="0x0200000077178b24288485b9f690b6810f187eb9cd7ba294" />
          </executionStack>
          <inputbuf>
SELECT * FROM sandbox.dbo.testblocage   </inputbuf>
        </process>
      </blocked-process>
      <blocking-process>
        <process status="sleeping" spid="53" sbid="0" ecid="0"
priority="0" transcount="1" lastbatchstarted="2008-05-15T13:16:19.700"
lastbatchcompleted="2008-05-15T13:16:19.700"
clientapp="Microsoft SQL Server Management Studio - Query"
hostname="BABALUGA-XPS" hostpid="1804" loginname="BABALUGA-XPS\rudi"
isolationlevel="read committed (2)" xactid="289008" currentdb="7"
```

```
lockTimeout="4294967295" clientoption1="671090784"
clientoption2="390200">
          <executionStack />
          <inputbuf>

CREATE TABLE dbo.testblocage ( id int )

BEGIN TRAN
INSERT dbo.testblocage ( id ) VALUES( 1 )
   </inputbuf>
        </process>
      </blocking-process>
    </blocked-process-report>
  </TextData>
  <DatabaseID>7</DatabaseID>
  <TransactionID>289041</TransactionID>
  <Duration>13843000</Duration>
  <EndTime>2008-05-15T13:16:40.793</EndTime>
  <ObjectID>0</ObjectID>
  <IndexID>0</IndexID>
  <ServerName>BABALUGA-XPS\SQL2005</ServerName>
  <Mode>3</Mode>
  <LoginSid>AQ==</LoginSid>
  <EventSequence>1134</EventSequence>
  <IsSystem>1</IsSystem>
  <SessionLoginName />
</EVENT_INSTANCE>
```

Vous pouvez voir que le rapport inclut les informations des deux sessions, la bloquante (`<blocking-process>`) et la bloquée (`<blocked-process>`), avec les *input buffers* (les caches de la dernière instruction de la session) de chacune.

La façon correcte de recevoir l'événement est d'utiliser l'instruction `RECEIVE`, qui permet de lire une file d'attente Service Broker et de la vider des lignes récupérées. `RECEIVE` peut insérer à son tour le résultat dans… une variable de type table seulement. C'est pour cette raison que nous en utilisons une dans le code suivant. Nous plaçons l'extraction de la queue dans une transaction, afin d'annuler la suppression générée par `RECEIVE` en cas de problème. L'exemple démontre aussi l'utilisation de XQuery pour l'extraction des informations dans la structure XML, à l'aide des « méthodes » `value` et `query`.

```
CREATE TABLE dbo.BlockedProcesses (
   message_body xml NOT NULL,
   report_time datetime NOT NULL,
   database_id int NOT NULL,
   process xml NOT NULL
)
GO

BEGIN TRY
   BEGIN TRAN
   DECLARE @BlockedProcesses TABLE (
      message_body xml NOT NULL,
```

```
      report_time datetime NOT NULL,
      database_id int NOT NULL,
      process xml NOT NULL
   );
   DECLARE @rowcount int;

   RECEIVE cast( message_body as xml ) as message_body,
      cast( message_body as xml ).value( '(/EVENT_INSTANCE/PostTime)[1]',
'datetime' ) as report_time,
      cast( message_body as xml ).value( '(/EVENT_INSTANCE/DatabaseID)[1]',
'int' ) as database_id,
      cast( message_body as xml ).query( '/EVENT_INSTANCE/TextData/blocked-
process-report/blocked-process/process' ) as process
   FROM NotifyQueue
   INTO @BlockedProcesses;
   SET @rowcount = @@ROWCOUNT

   INSERT INTO dbo.BlockedProcesses
   SELECT * FROM @BlockedProcesses;
   IF (@rowcount <> @@ROWCOUNT)
      ROLlBACK
   ELSE
      COMMIT TRAN
END TRY
BEGIN CATCH
   ROLLBACK TRAN
END CATCH
```

Vous pouvez bien sûr utiliser Database Mail (`sp_send_dbmail`) pour vous notifier.

Comment éviter les blocages

Les blocages très courts sont inévitables en environnement concurrentiel, ils ne posent en général pas de problème sérieux. Par contre, nous l'avons vu, il faut autant que possible éviter les blocages longs. Pour cela, il faut optimiser le code SQL et créer de bons index pour diminuer la durée des transactions. Il faut aussi savoir jouer sur le niveau d'isolation des transactions. Les niveaux `READ UNCOMMITTED` et `SNAPSHOT` (ou `READ COMMITTED SNAPSHOT`) sont précieux.

7.5.2 Verrous mortels

Les verrous mortels, ou *deadlocks*, aussi poétiquement appelés étreintes fatales, ou moins poétiquement interblocages, sont des animaux particuliers qu'on rencontre occasionnellement dans les buissons des applications transactionnelles. Ils ressemblent un peu aux scénarios des westerns qui respectent l'unité de lieu, comme Rio Bravo. Dans la dernière partie de ce film d'Howard Hawks, les bons, menés par le shérif John T. Chance, détiennent en otage le frère du chef des méchants, Nathan Burdette. Et les méchants se sont emparés de l'adjoint du shérif, Dude. Burdette ne libérera Dude que si Chance lui rend son frère, et Chance ne libérera le frère que si Burdette rend à Dude sa liberté. Nous savons que dans ce genre de situation, il n'y a qu'une échappatoire : un des deux chefs doit mourir (mauvais exemple ici, car Burdette ne meurt pas).

SQL Server fonctionne de la même façon : un *deadlock* se produit lorsqu'une transaction attend la libération d'un verrou de la part d'une autre transaction, qui elle-même attend la libération d'un verrou de la première transaction. La situation est sans solution, et SQL Server le détecte à l'aide d'un thread de surveillance dédié, qui lance une recherche à travers toutes les sessions ouvertes[1]. Lorsqu'il rencontre une situation de *deadlock*, il choisit une victime (la transaction qui a effectué le moins de modification de données), et annule (`ROLLBACK`) cette transaction. Une erreur 1205 est envoyée à la session choisie comme victime, avec ce message :

```
Transaction (Process ID XX) was deadlocked on lock resources with
another process and has been chosen as the deadlock victim.
Rerun the transaction.
```

Dans des applications de bases de données correctement implémentées, un *deadlock* devrait être une situation rare. Elle ne peut se produire que dans le cas d'une transaction explicite, dans un contexte particulier. Illustrons-le par un exemple de code :

```
-- dans la session 1
BEGIN TRAN

UPDATE HumanResources.Employee SET MaritalStatus = 'S'
WHERE Gender = 'F'

UPDATE c SET Title = 'Miss'
FROM Person.Contact c
JOIN HumanResources.Employee e WITH (READUNCOMMITTED)
       ON c.ContactID = e.ContactID
WHERE e.Gender = 'F'

COMMIT TRAN

-- dans la session 2
BEGIN TRAN

UPDATE c SET Suffix = 'Mrs'
FROM Person.Contact c
JOIN HumanResources.Employee e WITH (READUNCOMMITTED)
       ON c.ContactID = e.ContactID
WHERE e.Gender = 'F'

UPDATE HumanResources.Employee SET MaritalStatus = 'M'
WHERE Gender = 'F'

COMMIT TRAN
```

Comme vous le voyez, les deux transactions effectuent des opérations similaires dans un ordre inversé. C'est la porte ouverte à un *deadlock*. Il suffit que les transac-

1. Pour ne pas consommer trop de ressources, le *thread* de surveillance se lance toutes les cinq secondes, et augmente automatiquement sa fréquence si des situations de *deadlock* sont détectées. Plus d'informations dans les BOL, entrée « *Detecting and Ending Deadlocks* ».

tions se retrouvent au même moment entre leurs deux instructions, pour que les instructions suivantes soient fatales. Il est aisé, à partir de cet exemple, de comprendre comment minimiser le risque de *deadlock* : réduire autant que possible la durée de ses transactions, et éviter d'effectuer les mêmes opérations dans des ordres différents (pour cela, il suffit de centraliser la logique des modifications dans une seule procédure stockée).

Malgré tout, une situation de *deadlock* est toujours possible, à plus forte raison si vous devez maintenir des bases de données dont vous n'avez pas écrit les procédures. Il est donc prudent de mettre en place une détection de *deadlocks*. C'est aisé en créant une alerte de l'agent SQL qui vous notifie au déclenchement de l'erreur 1205. Si vous commencez à rencontrer des *deadlocks*, vous avez quelques outils qui vous permettent de remonter à la source du problème.

Analyse des verrous mortels

Les drapeaux de trace 1204 et 1222 activent un mode de rapport détaillé de *deadlock*. Ils peuvent être utilisés séparément ou ensemble, pour afficher les informations de deux façons différentes. Il n'est possible de les activer que globalement[1], soit en les ajoutant en paramètres de lancement de SQL Server : `-T1204` ou `-T1222`, dans le gestionnaire de configuration (voir figure 7.6)

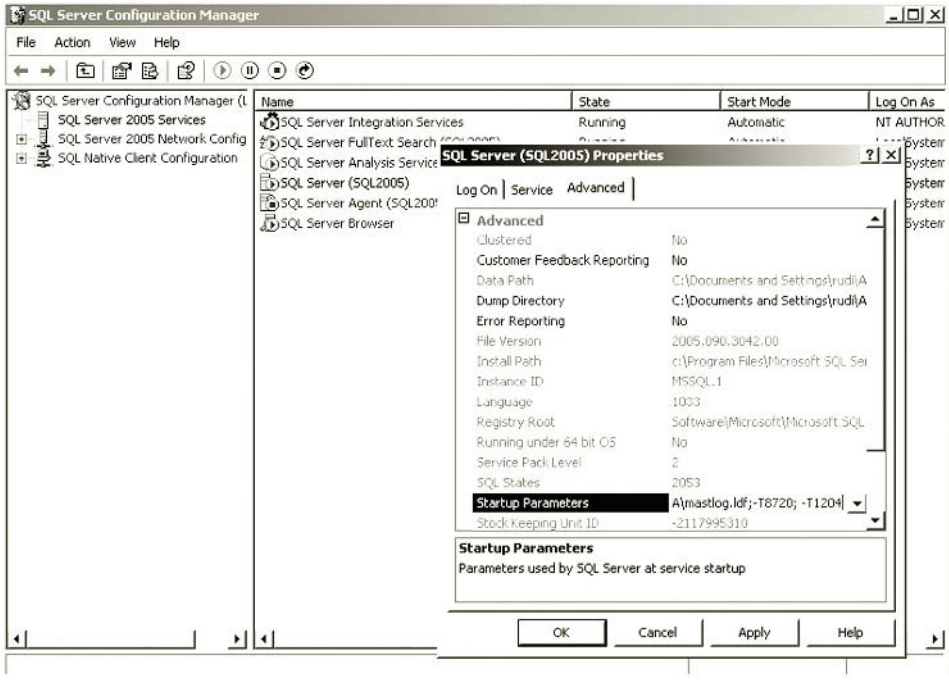

Figure 7.6 — Paramètres de démarrage

1. En SQL Server 2000 ces flags étaient activables par session, ce n'est plus le cas en SQL Server 2005.

soit à l'aide de DBCC TRACEON, en indiquant -1 en second paramètre, ce qui indique l'activation d'un drapeau global :

```
DBCC TRACEON(1204, -1)
```

Après activation, le moniteur de *deadlocks* inscrira des informations sur la chaîne de *deadlocks* dans le journal d'erreurs de SQL Server (le fichier ERRORLOG, situé dans le sous-répertoire LOG du répertoire où SQL Server stocke ses données). Voici le résultat d'une journalisation réalisée après activation du drapeau 1204 :

```
Deadlock encountered .... Printing deadlock information
Wait-for graph

Node:1

KEY: 5:72057594044153856 (07005a186c43) CleanCnt:3 Mode:X Flags: 0x0
 Grant List 0:
   Owner:0x1016D080 Mode: X        Flg:0x0 Ref:0 Life:02000000
   SPID:52 ECID:0 XactLockInfo: 0x11EBAF64
   SPID: 52 ECID: 0 Statement Type: UPDATE Line #: 1
   Input Buf: Language Event: UPDATE c
SET Title = 'Miss'
FROM Person.Contact c
JOIN HumanResources.Employee e WITH (READUNCOMMITTED)
       ON c.ContactID = e.ContactID
WHERE e.Gender = 'F'

 Requested By:
   ResType:LockOwner Stype:'OR'Xdes:0x10687A80 Mode: U SPID:51 BatchID:0
   ECID:0 TaskProxy:(0x1090A378) Value:0x1016b6c0 Cost:(0/19956)

Node:2

KEY: 5:72057594043236352 (2e0032505f99) CleanCnt:2 Mode:X Flags: 0x0
 Grant List 0:
   Owner:0x1016B420 Mode: X        Flg:0x0 Ref:0 Life:02000000
   SPID:51 ECID:0 XactLockInfo: 0x10687AA4
   SPID: 51 ECID: 0 Statement Type: UPDATE Line #: 1
   Input Buf: Language Event: UPDATE HumanResources.Employee
SET MaritalStatus = 'M'
WHERE Gender = 'F'

 Requested By:
   ResType:LockOwner Stype:'OR'Xdes:0x11EBAF40 Mode: U SPID:52 BatchID:0
   ECID:0 TaskProxy:(0x11BEA378) Value:0x1016b480 Cost:(0/13864)

Victim Resource Owner:
 ResType:LockOwner Stype:'OR'Xdes:0x11EBAF40 Mode: U SPID:52 BatchID:0
 ECID:0 TaskProxy:(0x11BEA378) Value:0x1016b480 Cost:(0/13864)
```

Comme vous le voyez, vous avez les informations de base nécessaire à comprendre la source du *deadlock* : les spids, et la dernière commande exécutée dans chaque spid (l'*inputbuffer*). La section « Victim Resource Owner » indique qui a été choisi

comme victime. Le drapeau 1222 affiche les informations dans un ordre différent, et donne plus d'informations sur les deux sessions concernées par le *deadlock*.

Vous pouvez également tracer les *deadlocks* avec le profiler, et obtenir une vue graphique de la chaîne. Les événements `Lock:Deadlock`, `Lock:Deadlock Chain` et `Lock:Deadlock Graph` vous indiquent l'apparition d'une *deadlock*, la chaîne de résolution, et un détail en format XML. Vous trouverez en figure 7.7 les événements lors d'un *deadlock*.

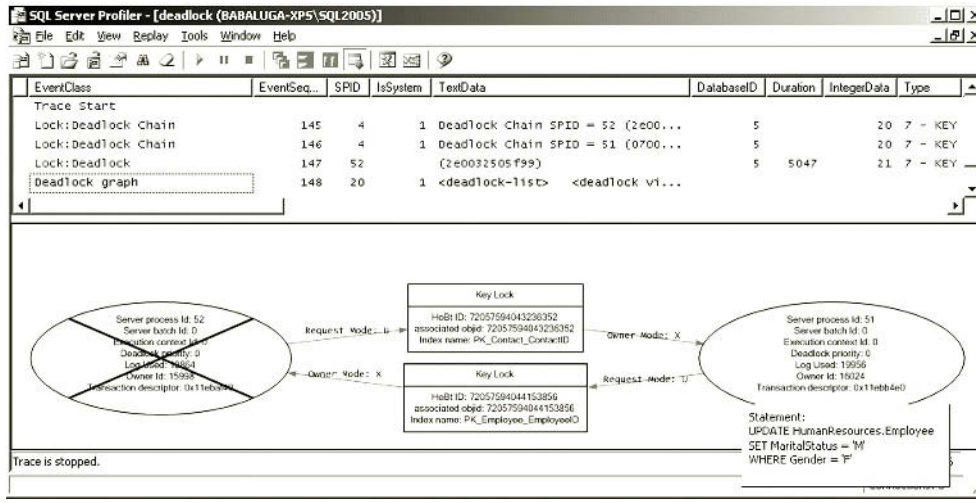

Figure 7.7 — *Deadlock graph*

Vous voyez que la chaîne de *deadlock* et le *graph*, sont des événements déclenchés par le système, sur des SPID inférieurs à 50 (donc des spids système). La colonne `EventSequence` permet de trier correctement la séquence d'événements. Un certain nombre de colonnes ajoutent des informations. Nous voyons ici `IntegerData`, qui retourne un numéro de *deadlock* – SQL Server leur attribue un numéro incrémentiel – et `Type` indique sur quelle ressource de verrouillage il s'est produit. L'événement *Deadlock graph* s'affiche graphiquement dans le profiler. Vous voyez les ressources en jeu, les types de verrous posés, sur quelques objets. La victime est marquée d'une croix. En laissant votre souris sur un des processus, vous obtenez *l'inputbuffer* en *tooltip*.

Maintenant que vous avez tous les éléments en main, vous pouvez dire à vos *deadlock*, comme John T. Chance : « *Sorry don't get it done, Dude. That's the second time you hit me. Don't ever do it again* ».

8

Optimisation du code SQL

Objectif

Les efforts d'optimisation doivent d'abord se porter sur le code SQL et la structure des données, avant de considérer des mises à jour matérielles. C'est là où les problèmes se posent en général, et c'est là où l'effet de levier est le plus important. Ce n'est que lorsqu'on a la garantie que le code et le schéma sont optimisés que la mise à jour matérielle doit être envisagée. Dans ce chapitre, nous allons vous présenter les plans d'exécutions générés par le moteur d'optimisation de SQL Server, qui vont vous permettre de juger de la qualité de votre code SQL. Nous parlerons ensuite de différentes façons d'écrire du code plus performant.

8.1 LECTURE D'UN PLAN D'EXÉCUTION

Lorsqu'une requête est reçue par le moteur relationnel, celui-ci vérifie d'abord si son **plan d'exécution** est présent dans le cache de plans. Si non, il doit créer ce plan. Cette phase est segmentée en quatre étapes. Nous avons schématisé l'exécution d'une requête dans la figure 8.1.

Le moteur de requête **parse** (évalue) d'abord la requête pour en vérifier la syntaxe, et la réécrit dans une structure qui sera utilisée pour l'optimisation.

La phase de **normalisation** consiste à vérifier l'existence des objets référencés, et la logique de la requête par rapport à ces objets (`SELECT * FROM ProcédureStockée`, par exemple, est illogique).

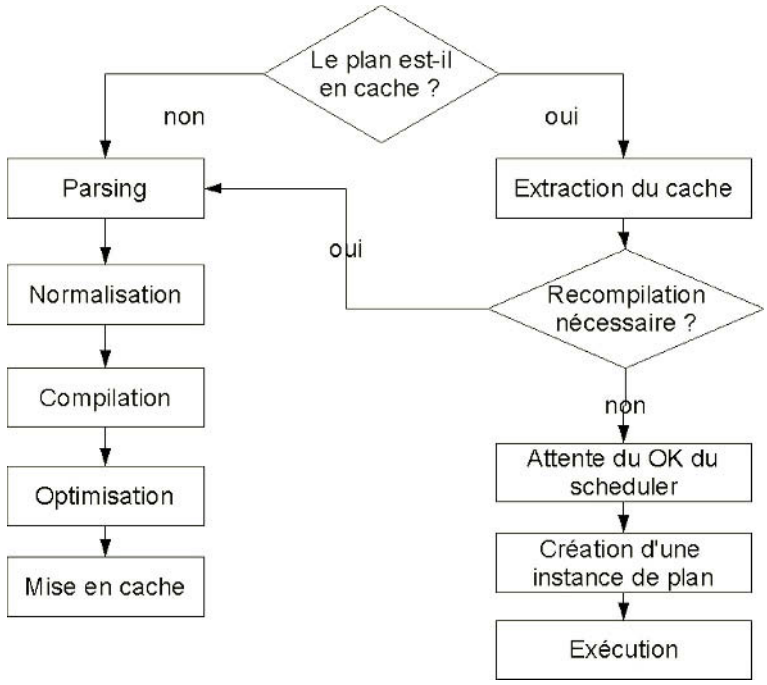

Figure 8.1 — Schéma de compilation

La **compilation** consiste à bâtir un arbre de séquence (*sequence tree*) pour décomposer la requête. À cette étape des conversions implicites de types de données sont appliquées si nécessaire, ainsi que le remplacement de l'appel aux vues, par l'appel aux tables sous-jacentes. Un graphe de requête (*query graph*) est créé, qui est ce qui est envoyé à l'optimiseur.

L'**optimisation** intervient pour les requêtes DML (`SELECT`, `INSERT`, `UPDATE` et `DELETE`). L'optimiseur reçoit un graphe de requête, et essaie de trouver la meilleure stratégie d'exécution. L'optimiseur SQL Server est fondé sur le coût (*cost-based optimizer*), c'est-à-dire qu'il va orienter son choix vers le plan le moins coûteux. Pour cela, il teste d'abord la possibilité d'utiliser un plan léger (*trivial plan*). Dans le cas par exemple d'un simple `INSERT … VALUES …`, inutile de chercher bien loin, la manière de réaliser cette commande est évidente. Si la requête ne peut être satisfaite par un plan léger le moteur d'optimisation va bâtir en parallèle des plans d'exécution, utilisant sa connaissance de la structure des objets (présence d'index, de contraintes `CHECK`, partitions, statistiques de distribution…). Il attribue un coût à chaque plan, un coût de temps de d'exécution et de consommation d'entrées/sorties évalué de façon heuristique. Pour ne pas prolonger cette phase elle-même coûteuse, il retient le plan ayant un coût raisonnable. Ainsi, le plan choisi n'est pas forcément le meilleur dans l'absolu, mais le premier trouvé suffisamment efficace pour résoudre la requête.

Le plan ainsi généré est mis en cache, après avoir été potentiellement paramétré en cas d'auto-paramétrage.

Ce plan d'exécution est l'information essentielle dont nous avons besoin pour évaluer la qualité du code de notre requête et ses performances estimées, et pour considérer la création d'index utiles. Nous avons plusieurs moyens de le visualiser. Le plus simple est d'afficher une représentation graphique de ce plan dans SSMS. Deux versions de ce plan y sont disponibles, le plan dit estimé, et le plan dit réel. Le plan estimé est celui qui est produit par l'optimiseur, et qui est affiché avant l'exécution de la requête. Le plan d'exécution réel est le même plan exactement que le plan estimé, puisque ce plan est utilisé sans aucune sorte de modification dynamique en cours d'exécution de la requête. Simplement, le plan réel inclut quelques statistiques d'exécutions effectives, qui peuvent différer des statistiques estimées, principalement sur le nombre de lignes affectées.

La combinaison de touches CTRL+L affiche le plan estimé des instructions de la fenêtre de requête active, ou des instructions sélectionnées. CTRL+M prépare l'affichage du plan réel, dans un onglet supplémentaire de la fenêtre de résultat. Vous pouvez aussi les activer par des boutons de la barre d'outils de l'éditeur SQL, indiqués sur la figure 8.2.

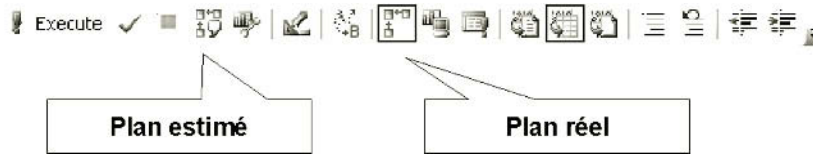

Figure 8.2 — Barre d'outils

Le plan d'exécution est ainsi affiché dans une représentation graphique, qui est l'adaptation synoptique d'un plan généré en format XML. Ce plan peut être sauvegardé dans un fichier, avec un clic droit sur le plan, et la commande « *Save execution plan as…* ». L'extension par défaut est `.sqlplan`. Lorsque vous ouvrez un fichier portant cette extension avec SSMS, la représentation graphique est générée, ce qui est pratique pour afficher des plans exportés d'une vision XML, ou à partir d'une trace.

Nous trouvons, en figure 8.3, un plan graphique simple.

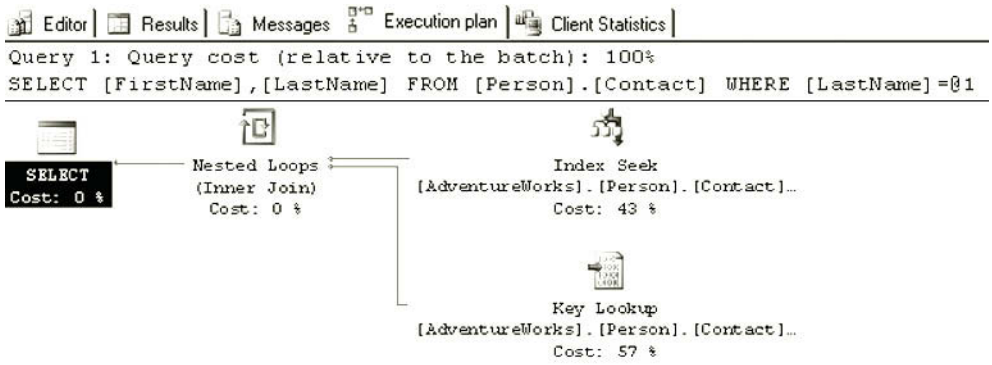

Figure 8.3 — Plan d'exécution graphique

correspondant à la requête :

```
SELECT FirstName, LastName
FROM Person.Contact
WHERE LastName = 'Abercrombie';
```

Vous voyez au passage que la requête a été auto-paramétrée par le moteur relationnel, puisque le texte (statement) visible dans le plan d'exécution est :

```
SELECT [FirstName], [LastName]  FROM [Person].[Contact]
WHERE [LastName] =@1;
```

Nous verrons ce qu'est l'auto-paramétrage dans la section 9.3.

Chaque icône représente un opérateur du plan, c'est-à-dire une opération effectuée sur les données. Nous les détaillerons dans la section suivante. Le plan se lit de droite à gauche. La première opération effectuée se trouve en haut et à droite. Chaque opérateur produit une table en mémoire dont le résultat qui est envoyé à l'opérateur suivant, ce qu'expriment les flèches. Le nombre de lignes transportées est représenté visuellement par l'épaisseur des flèches. Sur la figure 8.3, toutes les flèches sont fines, elles ne conduisent donc qu'un petit nombre de lignes. Chaque opérateur ainsi que l'instruction entière affichent un coût relatif en pourcentage. Lors de l'affichage du plan de plusieurs instructions, le coût par instruction montre une estimation du poids relatif de chacune, ce qui est pratique pour avoir un aperçu des performances comparées de plusieurs syntaxes de requête. La touche F4 permet d'afficher les propriétés du plan ou de chaque opérateur, selon la sélection effectuée à la souris. De même, en laissant le pointeur de la souris posé sur un opérateur ou une flèche, on peut en voir les détails dans une fenêtre de type info-bulle. Nous voyons en figure 8.4 l'affichage des propriétés et de l'info-bulle.

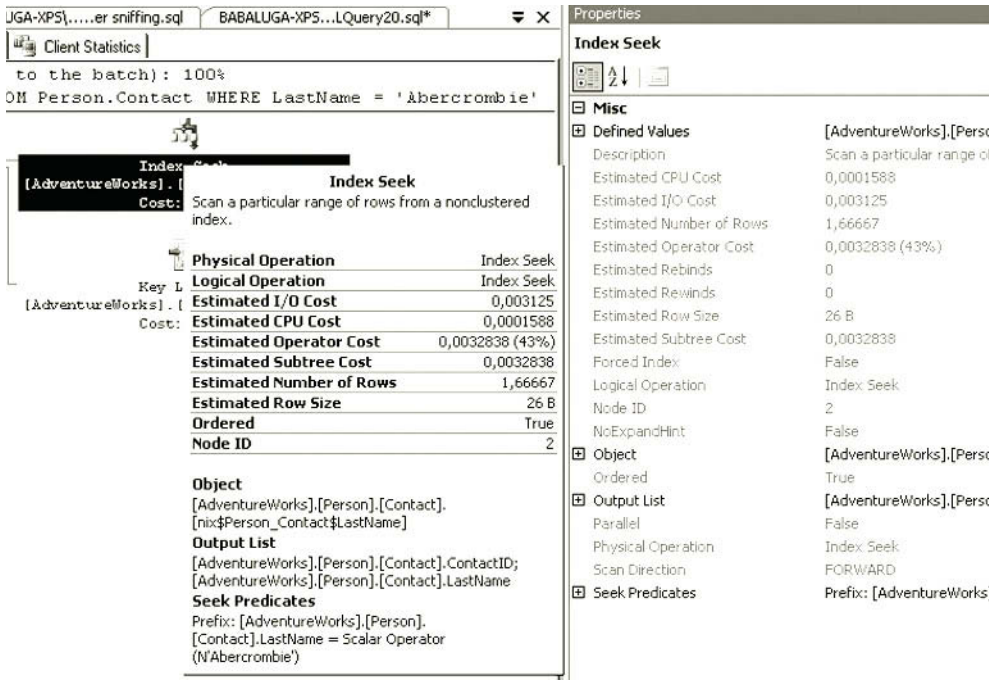

Figure 8.4 — Propriétés et info-bulle.

Sont notamment utiles l'estimation du nombre de lignes (produit par l'analyse des statistiques de distribution sur la colonne), et la taille moyenne estimée de la ligne en octets. La taille multipliée par le nombre nous donne une estimation du volume de données que le moteur de stockage va devoir transmettre d'opérateur à opérateur (nous pouvons voir directement ce calcul dans l'info-bulle des flèches). Le coût estimé de l'opérateur est une addition des coûts CPU et I/0 estimés. Vous voyez également sur quel objet est effectuée l'opération (sur la figure 8.4, c'est une recherche dans l'index `nix$Person_Contact$LastName`), quelles sont les colonnes résultantes qui sont envoyées à l'opérateur suivant (*output list*), et le prédicat de filtre (le scalaire `N'Abercrombie'`, qui est converti automatiquement en `nvarchar` pour correspondre au type de données de la colonne `LastName`). Le coût du sous-arbre (*subtree cost*) est simplement la somme du coût de tous les opérateurs précédents plus le courant, indiquant donc ce qu'a déjà coûté le trajet.

> Si vous avez désactivé la création ou la mise à jour automatique de statistiques sur votre base de données, vous verrez apparaître un signe d'alerte sur les opérateurs de votre plan qui n'auront pas les statistiques nécessaires à l'estimation d'une cardinalité. Dans le plan en XML, l'information sera contenue dans un élément `<Warnings><ColumnsWithNoStatistics>`.

Il peut exister au plus deux versions d'un même plan en cache : une version monoprocesseur et une version parallélisée. SQL Server choisit d'utiliser l'un ou l'autre selon la charge des CPU à l'exécution. Si le plan parallélisé est choisi, tous les opérateurs enrôlés dans l'exécution parallèle seront agrémentés d'une image l'indiquant, comme en figure 8.5.

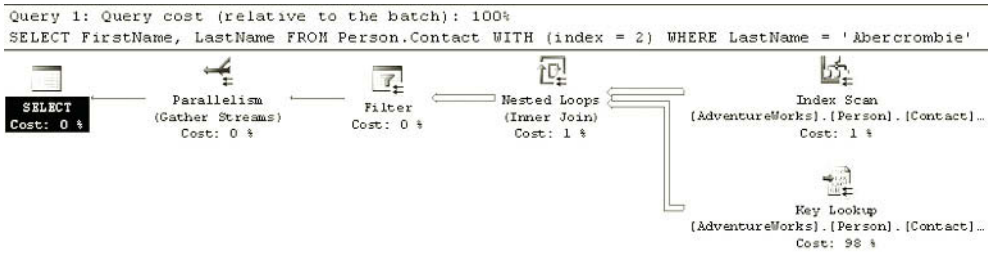

Figure 8.5 — Plan d'exécution parallélisé

Pour ce plan, nous avons forcé un mauvais plan en ajoutant un indicateur d'index mal à propos. L'exécution en parallèle doit joindre les résultats provenant des différents threads, ce qu'il fait ici dans l'opérateur « Parallelism (Gather Streams) ».

Grâce à cet affichage graphique, vous avez tous les éléments pour comprendre rapidement comment votre requête est exécutée. Afin de l'optimiser, concentrez-vous sur les opérateurs les plus coûteux, et les flèches les plus épaisses. Un des objectifs de l'optimisation est notamment de filtrer le nombre de lignes à traiter au plus tôt dans l'arbre du plan d'exécution, le pire étant représenté par un nombre très important de lignes traitées au début, pour aboutir à un petit nombre en fin de plan. Visuellement, les flèches sont très épaisses à droite, puis très fines à gauche.

Par exemple, que constatons-nous sur la figure 8.5 ? Comme nous avons forcé l'utilisation d'un index inutile pour résoudre le filtre de la clause `WHERE`, SQL Server est obligé de nous obéir : il parcourt le nœud feuille de l'index (*index scan*) pour en extraire les `ContactID` contenus dans la clé de l'index (tous les index contiennent en plus la clé de l'index *clustered*). Le résultat, 19 973 lignes sont retournées, pour un total de 125 Ko, comme nous pouvons le voir sur la figure 8.6.

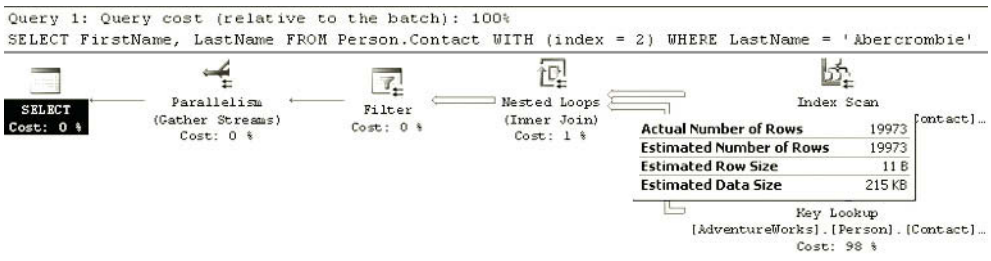

Figure 8.6 — Envoi d'un opérateur à l'autre

Ces lignes donnant le `ContactID`, il faut maintenant aller chercher les colonnes `FirstName` et `LastName` dans la table. Pour cela, une recherche à travers l'index *clustered* doit être faite pour chaque `ContactID`, ce qui est représenté ici par l'opérateur de boucle imbriquée (*nested loop*, dont nous parlerons dans la section 8.1.2 sur les algorithmes de jointure). Pour chaque ligne de la flèche du haut, une recherche est effectuée avec l'opérateur du bas, ici un *bookmark lookup* de type *Key lookup* : une recherche (*seek*) dans l'index *clustered*. C'est cette opération qui plombe toute la requête, représentant 98 % de son coût total. En effet, il faut parcourir 19 973 fois l'index *clustered*. La flèche qui la relie au *nested loop* le montre (dans le plan estimé, cette flèche est fine, parce que l'estimation qui est faite par SQL Server est de une seule ligne retournée). Dans les propriétés de l'opérateur *Nested loop* (ou de la flèche qui vient du *Key lookup*), nous avons l'information du nombre de lignes – donc de *seeks* de l'index *clustered* – qui ont été traitées par *thread*, comme nous le voyons sur la figure 8.7.

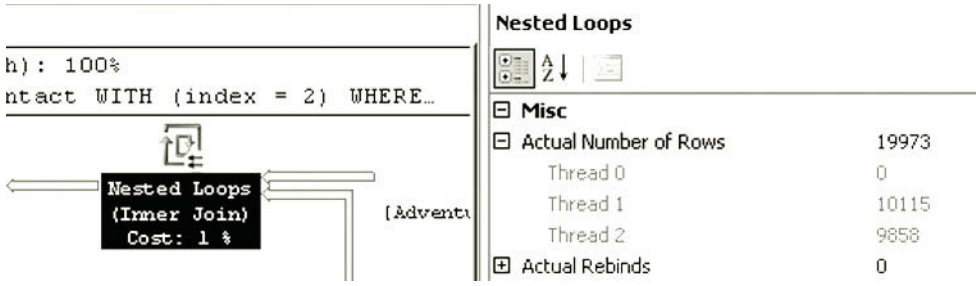

Figure 8.7 — *Nested loop* sur plusieurs threads

Lorsque toutes les lignes sont retournées avec la valeur de `FirstName` et `LastName`, elles sont envoyées (1,4 Mo passe du *nested loop* au *filter*) à un opérateur de filtre, qui, travaillant en mémoire, va éliminer les lignes qui ne correspondent pas au prédicat de recherche `LastName = N'Abercrombie'`. Il n'en restera que trois lignes, envoyées à la synchronisation des threads parallélisés, puis au client comme jeu de résultats. Nous verrons un peu plus loin les opérateurs plus en détail.

Récupération des plans d'exécution

Outre l'affichage graphique intégré à SSMS, vous avez plusieurs moyens de générer des plans d'exécution.

Tout d'abord, un certain nombre de commandes de session permettent l'affichage du plan :

- `SET SHOWPLAN_XML ON` : affiche le plan d'exécution estimé complet en format XML, sans exécuter l'instruction.
- `SET SHOWPLAN_TEXT ON` : affiche le plan d'exécution estimé simplifié en format texte (une ligne de jeu de résultat par opérateur du plan), sans exécuter l'instruction.

- `SET SHOWPLAN_ALL ON` : affiche le plan d'exécution estimé détaillé (avec estimations et alertes) en format texte (une ligne de jeu de résultat par opérateur du plan), sans exécuter l'instruction.
- `SET STATISTICS XML ON` : affiche le plan d'exécution complet réel en format XML, après exécution de l'instruction.
- `SET STATISTICS PROFILE ON` : affiche le plan d'exécution détaillé réel en format texte (une ligne de jeu de résultat par opérateur du plan), après exécution de l'instruction.

Les commandes `SET SHOWPLAN...` doivent être lancées dans leur propre batch, sans aucune autre instruction. Vous devez donc les séparer de vos requêtes par un `GO` dans SSMS.

Les commandes non XML sont considérées par Microsoft comme en voie d'obsolescence, elles seront peut-être supprimées dans une version ultérieure de SQL Server. SQL Server 2008 les supporte toujours.

Les commandes `SET SHOWPLAN...` affichent donc le plan estimé, et les commandes `SET STATISTICS` le plan réel. Les quelques informations supplémentaires du plan réel sont visibles dans le plan XML dans les éléments `<RunTimeInformation>` qui y sont ajoutés. Sur l'affichage graphique du plan, vous les trouvez dans les affichages détaillés lorsque vous glissez votre souris sur un opérateur, ils commencent par « *Actual...* ».

Vous pouvez aussi récupérer ces plans à partir d'une trace SQL. Les événements à sélectionner sont dans le groupe d'événements « Performance » :

L'événement **Performance statistics** se déclenche quand un plan d'exécution est inséré dans le cache de plans, quand il est recompilé, ou quand il est vidé du cache. La colonne `EventSubClass` contient l'identifiant du type d'événement, qui peut être un des suivants :

0 – un nouveau batch n'est pas encore dans le cache. La colonne TextData contient le code SQL de la requête.

1 – des instructions dans une procédure stockée ont été compilées.

2 – des instructions dans un batch de requêtes ad hoc ont été compilées.

3 – une requête a été supprimée du cache et les données historiques de performances vont être détruites.

4 – une procédure stockée a été supprimée du cache et les données historiques de performances vont être détruites.

5 – un déclencheur a été supprimé du cache et les données historiques de performances vont être détruites.

Les types 4 et 5 ne sont disponibles qu'en SQL Server 2008.

Les événements 1, 4 et 5 retournent les colonnes `DatabaseID` et `ObjectID`, le nom de l'objet (procédure ou déclencheur) peut être retrouvé grâce à la fonction `OBJECT_NAME()` qui accepte en deuxième paramètre un `DB_ID` depuis le service pack 2 de SQL Server 2005.

Dans les événements 1 et 2, la colonne `BinaryData` contient le plan d'exécution généré en format binaire, de même les colonnes `CPU` et `Duration` donnent les temps de compilation. La colonne `IntegerData` donne la taille en kilo-octets, du plan généré. Les colonnes `BigintData1` et `BigintData2` donnent respectivement le nombre de fois où ce plan a été recompilé, et la taille mémoire qui a été nécessaire à la compilation, en Ko.

Tous retournent les colonnes `SqlHandle` et `PlanHandle` (sauf l'événement 0 qui ne retourne que le `SqlHandle`) qui donnent des identifiants utiles pour obtenir plus d'informations à partir des vues de gestion dynamique que nous présenterons dans la section suivante.

Les événements **`Showplan XML`**, **`Showplan XML For Query Compile`** et **`Showplan XML Statistics Profile`** affichent des plans en XML dans la colonne `TextData` :

- **`Showplan XML`** : retourne le plan d'exécution qui est utilisé par la requête, sans les statistiques d'exécution. Cela correspond au plan estimé. L'événement se déclenche après l'événement `SQL` ou `SP Starting`, mais avant `Showplan XML Statistics Profile`.
- **`Showplan XML For Query Compile`** : retourne le plan d'exécution tel qu'il est compilé et mis en cache. Cet événement se déclenche durant la phase de compilation, avant exécution (donc avant un événement `Starting`), et avant la mise en cache (événement `SP:CacheInsert`).
- **`Showplan XML Statistics Profile`** : retourne le plan d'exécution utilisé par la requête avec les statistiques d'exécution. Cela correspond au plan réel. L'événement se déclenche juste avant l'événement `SQL` ou `SP Completed`.

Dans les trois cas, `BinaryData` contient le coût estimé de la requête, et `IntegerData` le nombre estimé de lignes retournées ; `ObjectID` et `ObjectName` contiennent l'ID et le nom de l'objet, `ObjectType` le type d'objet.

Autres événements

Les événements `Showplan Text`, `Showplan Text (Unencoded)`, `Showplan All`, `Showplan All For Query Compile` et `Showplan Statistics Profile` existent pour compatibilité. Vous pouvez les utiliser pour voir les plans en format texte. Les événements produisant des plans en XML sont plus complets et devraient être préférés.

Le profiler affiche le plan d'exécution graphique, comme SSMS, sur les événements qui retournent ce plan en XML, ou en format binaire, comme nous le voyons sur la figure 8.8.

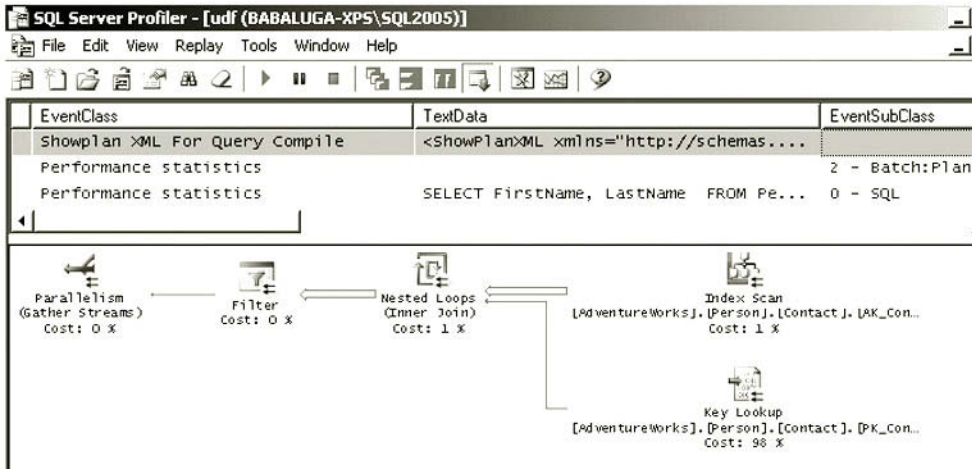

Figure 8.8 — Plan d'exécution dans le profiler

Un dernier mot : la génération en masse de plans d'exécution dans une trace est non seulement inutile et provoque un raz-de-marée d'informations, mais est aussi très pénalisante pour les performances. Tracez ces événements seulement au besoin, en filtrant votre trace.

Vues de gestion dynamique

Vous trouvez le code SQL de vos requêtes, ainsi que leur plan d'exécution, dans certaines vues de gestion dynamique, celles qui lisent le contenu du cache de plans, et celles qui affichent les requêtes exécutées dans le SQL Server, en temps réel. Deux fonctions permettent de retourner respectivement le code SQL de l'instruction (`sys.dm_exec_sql_text`) et le plan d'exécution XML (`dm_exec_query_plan`). Exemple d'extraction des textes et des plans des tâches en cours :

```
SELECT er.session_id, er.start_time, er.status, er.command,
       st.text, qp.query_plan
FROM sys.dm_exec_requests er
CROSS APPLY sys.dm_exec_sql_text(er.sql_handle) st
CROSS APPLY sys.dm_exec_query_plan(er.plan_handle) qp
```

La colonne `session_id` représente les SPID, vous pouvez donc éliminer votre SPID avec un `WHERE session_id <> @@SPID`, pour éviter de voir cette requête que vous venez de lancer, dans la liste.

Le plan d'exécution (colonne `query_plan`) étant de type XML, vous pouvez y appliquer du XQuery. Exemple d'extraction du code SQL de l'intérieur du plan :

```
WITH XMLNAMESPACES (DEFAULT
    'http://schemas.microsoft.com/sqlserver/2004/07/showplan')
SELECT er.session_id, er.start_time, er.status, er.command,
    qp.query_plan.value('(/ShowPlanXML/BatchSequence/Batch/Statements/
        StmtSimple/@StatementText)[1]', 'varchar(8000)') as sql_text
FROM sys.dm_exec_requests er
CROSS APPLY sys.dm_exec_query_plan(er.plan_handle) qp
```

(la ligne de requête XQuery est coupée dans l'exemple).

Ce qui a permis à Bob Beauchemin de créer une intéressante procédure de recherche d'opérateur dans un plan, que nous reproduisons ici. Vous pouvez la trouver dans cette entrée de blog : http://www.sqlskills.com/blogs/bobb/2006/03/03/MoveOverDevelopersSQLServerXQueryIsActuallyADBATool.aspx.

```
CREATE PROCEDURE LookForPhysicalOps (@op VARCHAR(30))
AS
SELECT sql.text, qs.EXECUTION_COUNT, qs.*, p.*
FROM sys.dm_exec_query_stats AS qs
CROSS APPLY sys.dm_exec_sql_text(sql_handle) sql
CROSS APPLY sys.dm_exec_query_plan(plan_handle) p
WHERE query_plan.exist('
declare default element namespace "http://schemas.microsoft.com/sqlserver/2004/
07/showplan";
/ShowPlanXML/BatchSequence/Batch/Statements//RelOp/
@PhysicalOp[. = sql:variable("@op")]
') = 1
GO

EXECUTE LookForPhysicalOps 'Clustered Index Scan'
EXECUTE LookForPhysicalOps 'Hash Match'
EXECUTE LookForPhysicalOps 'Table Scan'
```

8.1.1 Principaux opérateurs

Les opérateurs de plan sont séparés en deux niveaux. L'**opérateur logique** représente l'action logique entreprise, au niveau de l'algèbre relationnelle (par exemple, *Full Outer Join*), ou du concept de recherche, tandis que l'**opérateur physique** est la méthode algorithmique utilisée pour implémenter cette opération (par exemple, *nested loop*, ou boucle imbriquée).

Dans notre exemple, il s'agit d'une recherche d'index (*index seek*), un opérateur à la fois logique et physique. Les premiers opérateurs présents sont en général une extraction de données (ou un calcul sur un scalaire).

Vous trouvez la description des opérateurs et leurs icônes dans les BOL, sous « *Graphical Execution Plan Icons (SQL Server Management Studio)* », nous listons ici les plus courants et les plus significatifs.

Bookmark Lookup – recherche de lignes dans une table après parcours ou recherche dans un index *nonclustered*, à l'aide d'une clé d'index *clustered* ou d'un RowID. Le *bookmark lookup* est le plus souvent exprimé dans le plan soit par un opérateur *Key Lookup*, soit par un opérateur *RID Lookup*.

Clustered Index Scan – parcours du nœud feuille d'un index *clustered*, donc de la table elle-même.

Clustered Index Seek – recherche dans un index *clustered*. Cette recherche n'est jamais suivie d'un *bookmark lookup*, puisque la table fait partie de l'index *clustered*.

Split et **Collapse** – ces opérateurs se rencontrent dans certaines mises à jour (`UPDATE`) où une contrainte d'unicité doit être vérifiée. Un `UPDATE` seul sur une colonne avec contrainte unique pourrait déboucher sur l'insertion de doublons, parce que la mise à jour est exécutée ligne par ligne par le moteur de stockage. Il faut donc vérifier la totalité des lignes mises à jour en une fois. Cela se fait par la séquence d'opérateurs reproduite en figure 8.9.

Figure 8.9 — Split et collapse

Le *split* est la séparation d'un `UPDATE` en un `DELETE` suivi d'un `INSERT`. Des opérateurs de filtre puis de tri organisent les lignes supprimées puis insérées dans l'ordre de la clé unique, afin de vérifier qu'il n'y a pas deux insertions qui portent la même valeur, le *collapse* réunit ensuite les lignes sur la clé unique (vous trouvez une propriété `GROUP BY` qui indique sur quelles colonnes le *collapse* est effectué)[1].

Compute Scalar – calcul d'une valeur scalaire à partir d'une expression, et de tout type de source. Par exemple un `COUNT(*)` à partir d'un *scan* d'index est implémenté en tant que *Stream Aggregate*, puis *Compute Scalar*. Un *Compute Scalar* est visible partout où une colonne doit contenir une valeur calculée, concaténée, etc.

Concatenation – copie des lignes d'un jeu de résultats vers un autre, typiquement utilisé pour un `UNION` ou `UNION ALL`. Dans le cas d'un `UNION`, si les entrées sont triées, un `Merge Join` sera souvent préféré, pour permettre un dédoublonnage plus rapide.

Constant Scan – introduit des valeurs constantes comme nouvelles lignes. Un *Compute Scalar* est souvent utilisé ensuite pour ajouter des colonnes à cette ligne. Vous le verrez par exemple lorsque vous voudrez générer un jeu de résultats vide (par exemple pour créer une structure de table vide à l'aide de `SELECT INTO`). Des requêtes telles que celles-ci :

1. http://blogs.msdn.com/craigfr/archive/2007/09/06/maintaining-unique-indexes.aspx

```
SELECT TOP 0 * FROM Person.Address;
SELECT * FROM Person.Address WHERE 1 = 0;
```

créent un plan très simple, composés d'un seul *Constant Scan*.

Eager Spool et **Lazy Spool** – les opérateurs de *spool* créent une table de travail dans `tempdb`. Leur différence vient de leur « enthousiasme » à le faire. Le *spool* enthousiaste (*eager*) insère toutes les lignes en une seule fois dans la table de travail, alors que le *spool* paresseux (*lazy*) le fait une ligne après l'autre : lorsque l'opérateur dépendant du *Lazy Spool* lui demande une ligne, celui-ci en fait la demande à son opérateur enfant, et stocke la ligne dans le *spool*.

Filter – filtre un jeu de lignes selon le critère présent dans sa propriété `Argument`.

Hash Aggregate – calcul d'agrégation utilisé plutôt sur des volumes importants, et uniquement quand la requête utilise un `GROUP BY`. Une table de hachage est créée pour permettre un calcul sur tous les sous-ensembles en même temps. Cette méthode utilise plus de mémoire et verrouille plus que le *Stream Aggregate*.

Hash Match – test de présence de ligne sur une table de hachage bâtie à partir de l'entrée de l'opérateur enfant (la liste des colonnes faisant partie du hachage est dans le prédicat `HASH:()` de la propriété `Argument`). Les valeurs de hachage sont calculées pour chaque ligne testée et comparées à la table de hachage, pour tester leur présence. C'est un opérateur physique qui dépend de trois types d'opérateurs logiques :

- *join* – l'opérateur enfant du haut est celui sur lequel la table de hachage est basée, l'opérateur du bas est celui à partir duquel chaque ligne est testée ;
- *aggregate ou distinct* – la table de hachage est créée à partir de l'entrée, les doublons sont supprimés, ou les agrégations calculées, puis la comparaison est faite avec un *scan* de la même entrée pour conserver les *match* ;
- *union* – la table de hachage est calculée à partir de l'opérateur enfant du haut. Les doublons sont éliminés. Les lignes de l'entrée du bas sont testées, et les non-*match* sont retournés.

Key Lookup – recherche de lignes dans une table *clustered*, à partir d'une clé d'index *clustered* présente dans un index *nonclustered*, après parcours ou recherche de cet index. C'est une de deux formes particulières du *bookmark lookup*.

Merge Interval – normalise plusieurs intervalles pour en produire un seul, qui sera utilisé dans une recherche d'index.

Nonclustered Index Spool – un index *nonclustered* est créé sur une table de travail dans `tempdb`, pour améliorer les performances de recherche dans cette table.

Les opérateurs **Remote** (`Delete`, `Insert`, `Query`, `Scan`, `Update`) indiquent des opérations sur un serveur distant ou un serveur lié (le plus souvent à travers un serveur lié, *linked server*, les serveurs distants, *remote server*, étant obsolètes).

RID Lookup – recherche de lignes dans une table *heap*, à partir d'un identifiant de ligne (*Row ID*) présent au niveau feuille d'un index *nonclustered*, après parcours

ou recherche de cet index. C'est une de deux formes particulières du *bookmark lookup* ;

Vous verrez souvent des opérateurs de **Segment**, **Sequence** et **SequenceProject** dans les requêtes impliquant des fonctions de fenêtrage (une syntaxe de la norme SQL implémentée en SQL Server depuis la version 2005). Elles servent à calculer le résultat de ces fonctions à l'intérieur du jeu de résultats. SQL Server 2008 améliore les performances des fonctions de fenêtrages.

Sort – tri des lignes d'entrée. La propriété `ORDER BY` indique la ou les colonnes sur lesquelles s'opère le tri.

Spool et **Table Spool** – *Spool* sauve le résultat intermédiaire dans une table de travail dans `tempdb`, *Table Spool* fait de même à partir des lignes parcourues et sélectionnées dans l'opérateur enfant.

Stream Aggregate – calcul d'agrégation sur un ensemble ou sous-ensemble provenant d'une source triée. Si la source n'est pas déjà triée (elle l'est par exemple si elle provient d'un *scan* d'index, puisque la clé de l'index est triée), un opérateur *Sort* devra précéder celui-ci (ou un *Hash Aggregate* sera utilisé, selon la taille de la table). Vous trouvez dans les propriétés de l'opérateur l'expression de calcul[1].

Table Scan – *scan* de table sans index *clustered*, donc de table *heap*.

Table-valued Function – une fonction utilisateur retournant une table est appelée. La table retournée est stockée dans `tempdb` (c'est une variable de type table), elle sera traitée dans `tempdb`.

UDX – une opération XML, comme un `FOR XML` ou une requête XQuery.

8.1.2 Algorithmes de jointure

SQL Server implémente trois algorithmes physiques pour effectuer des jointures. Nous allons les présenter.

La **boucle imbriquée** (*nested loop*) est l'algorithme de base, qui est le plus simple et le plus utilisé dans les plans d'exécution. Il consiste à extraire des lignes d'une première table (la table externe), et à rechercher leur correspondance dans une seconde table (la table interne), comme si on cherchait une liste de références dans un premier livre, pour aller chercher chaque référence dans un second. L'algorithme peut se schématiser ainsi :

```
pour chaque ligne de la table externe
    pour chaque ligne de la table externe
        joindre les lignes
        retour des lignes
```

1. http://blogs.msdn.com/craigfr/archive/2006/09/13/752728.aspx

D'où le nom de boucles imbriquées. Cet algorithme fonctionne très bien pour les petits volumes, mais son coût augmente proportionnellement aux nombres de lignes à traiter.

Les *bookmark lookups* sont exprimés en boucles imbriquées dans les plans depuis SQL Server 2005.

La **fusion** (*merge join*) consiste à prendre deux tables triées selon les mêmes colonnes, et à extraire les correspondances entre la première table et la seconde. Comme elles sont triées, le parcours se fait toujours vers l'avant, chaque correspondance est conservée, et les lignes suivantes sont testées. C'est donc un algorithme très rapide, mais il implique un tri préalable, et une clause d'équijointure (on ne peut comparer de cette façon que des égalités). Le tri étant très coûteux, la jointure de fusion est donc utilisée principalement sur les tables ordonnées par un index *clustered*, ou sur les nœuds feuilles d'index.

Le **hachage** (*hash join*) est privilégié sur les jeux de résultats importants, car il permet, après préparation, de réaliser de grosses jointures relativement rapidement. Il fonctionne en deux phases : il crée premièrement une table de hachage sur les colonnes de la condition de recherche sur la première table. Ensuite, il parcourt la deuxième table, crée un hachage pour chaque condition de recherche et effectue la comparaison avec la table de hachage. Cet algorithme ne fonctionne qu'avec des clauses d'équijointures. Il souffre de quelques désavantages : il est « bloquant », car la première étape doit être complètement terminée avant de passer à la seconde (alors que le *nested loop* et le *merge* peuvent travailler au fil de l'eau), et il est très consommateur de mémoire. SQLOS lui réserve une quantité de mémoire estimée pour son travail, mais si cette estimation était trop optimiste, il doit, pendant l'exécution, « baver » sur `tempdb` (on parle de *spilling*). Cela ralentit bien sûr nettement l'opération. Le hachage est utilisé aussi pour les calculs d'agrégations, et ce débordement peut aussi se produire dans ce cas.

Débordements de hachages

Vous avez un événement de trace qui vous permet de détecter les débordements dans `tempdb` dus à des hachages (*Hash Aggregate* et *Hash Join*) : `Errors and Warnings : Hash Warning`. Vous pouvez utiliser des indicateurs de requête pour forcer un algorithme de calcul d'agrégation :

- `option(order group)` – force le *Stream Aggregate* ;
- `option(hash group)` – force le *Hash Aggregate*.

Attention à l'option `hash group` : un *Hash Aggregate* n'est possible qu'en présence d'un `GROUP BY`. Si vous utilisez cet indicateur dans une agrégation scalaire (un calcul d'agrégat sans regroupements, donc sans clause `GROUP BY`), une erreur vous sera renvoyée à la compilation : « *Query processor could not produce a query plan because of the hints defined in this query…* ».[1]

8.2 GESTION AVANCÉE DES PLANS D'EXÉCUTION

Vous pouvez modifier le plan d'exécution généré par l'optimiseur de deux façons : soit en ajoutant à la requête des indicateurs spécifiques forçant certaines stratégies logiques ou physiques de résolution de la requête, soit en y appliquant un plan d'exécution « fait main » complet, à l'aide des guides de plan (*plan guides*). Nous allons passer les deux solutions en revue. Mais avant cela, il nous faut répéter encore que ces fonctionnalités ne sont à considérer qu'en dernier ressort, et lorsque vous savez ce que vous faites. Soyez notamment conscient des répercussions lors de changement de structure ou de distribution des valeurs dans les tables. Par exemple, un guide de plan qui force l'utilisation d'un index qui a disparu, génère une erreur.

8.2.1 Indicateurs de requête et de table

Vous avez à disposition des **indicateurs** (*hints*) permettant d'indiquer à l'optimiseur une stratégie préférée, voire de le forcer à utiliser une méthode, un index, et jusqu'à un plan d'exécution prédéfini. Certains de ces indicateurs sont intéressants, mais ils ne devraient être utilisés qu'en cas de réel besoin. Le moteur d'optimisation de SQL Server est très performant, et le forcer à se comporter d'une certaine manière est en général contre-indiqué. Cette mise en garde faite, nous allons présenter ici ces indicateurs, et tâcher d'expliquer quand ils peuvent être utilisés, et quand ils ne devraient pas l'être.

On utilise un indicateur à même la requête, soit à la fin de celle-ci avec la clause `OPTION()` pour les indicateurs de requête (*query hints*), soit après une table avec la clause `WITH()` pour les indicateurs de tables (*table hints*). Voici un exemple d'utilisation d'un indicateur de requête forçant un certain algorithme de jointure :

```
SELECT *
FROM Sales.Customer c
JOIN Sales.CustomerAddress ca
       ON c.CustomerID = ca.CustomerID
WHERE TerritoryID = 5
OPTION (MERGE JOIN);
```

Indicateurs de requête

Les indicateurs de requête disponibles sont les suivants :

```
{ HASH | ORDER } GROUP
{ CONCAT | HASH | MERGE } UNION
{ LOOP | MERGE | HASH } JOIN
```

Ces trois indicateurs permettent de forcer les algorithmes d'agrégation (voir section 8.1.1), d'`UNION` et de jointure. Évitez de forcer ce choix, à de rares exceptions près, l'optimiseur sait ce qu'il fait.

1. Référence sur le Hash Aggregate : http://blogs.msdn.com/craigfr/archive/2006/09/20/hash-aggregate.aspx

```
FAST nombre_de_lignes
```

Précise que la requête doit être optimisée pour renvoyer d'abord un nombre spécifique de lignes au client, avant de poursuivre. Cette clause peut être utile pour un affichage rapide sur une première page de pagination. Elle permet, comme la clause `TOP`, de profiter de la fonctionnalité d'objectif de lignes (*row goal*), qui peut modifier le plan d'exécution en conséquence. C'est une bonne chose pour le retour du nombre de lignes indiqué, mais potentiellement au détriment des performances de la requête entière. Il s'agit donc d'une option à double tranchant, qui peut faire empirer les choses[1]. Il est préférable d'utiliser la clause `TOP`, qui garantit le retour du nombre de lignes indiqué.

```
FORCE ORDER
```

Spécifie que l'ordre des tables dans la déclaration de jointure doit être conservé. Cette option est très rarement utile, mais si vous avez un soupçon de mauvaise génération de plan, vous pouvez essayer différents ordres d'apparition des tables dans votre clause `FROM`, avec cette option, pour voir quelle est la combinaison gagnante. Une fois de plus, un changement structurel de tables, et la modification de la stratégie d'indexation, peut changer le plan optimal, qui ne pourra plus alors être automatiquement choisi par l'optimiseur.

```
MAXDOP nombre_de_processeurs
```

Vous permet d'indiquer un nombre de processeurs maximum impliqué dans une parallélisation de la requête. Rappelons que la parallélisation maximum peut être définie au niveau du serveur par l'option « maximum degree of parallelism ». Cet indicateur vous permet d'attribuer une autre valeur à une requête spécifique, ce qui peut se révéler utile soit pour désactiver le parallélisme sur une requête coûteuse (`MAXDOP 1`), soit pour l'activer sur une requête alors qu'il est désactivé au niveau du serveur. Rappelons également que cette option ne signifie pas que SQL Server va nécessairement paralléliser la requête. On lui en donne la possibilité, et il fait son choix selon le coût de la requête (selon l'option de serveur « cost threshold for parallelism ») et la charge actuelle des processeurs de la machine.

```
OPTIMIZE FOR ( @variable_name = literal_constant [ , ...n ] )
```

Indique à l'optimiseur de générer un plan optimisé pour une valeur de paramètre donnée, ce qui est très utile pour déjouer les pièges du *parameter sniffing* (voir section 9.2.1). Il vous suffit d'indiquer une valeur distribuée de façon représentative dans votre colonne, pour éviter la compilation avec des plans extrêmes. SQL Server 2008 ajoute plus de souplesse à cet indicateur :

- `OPTIMIZE FOR UNKNOWN` : tous les paramètres sont considérés avec une valeur de distribution moyenne de la colonne.
- `OPTIMIZE FOR ( @variable_name = UNKNOWN)` : une valeur de distribution moyenne est considérée pour ce paramètre.

1. Voir http://blogs.msdn.com/queryoptteam/archive/2006/03/30/564912.aspx

Cela permet d'éviter dans tous les cas les valeurs extrêmes. En général, pour les colonnes qui ont une distribution très hétérogène (quelques valeurs très sélectives, d'autres avec beaucoup de doublons), une stratégie de *scan* sera choisie. Il est inutile de spécifier cet indicateur pour des colonnes avec une contrainte ou un index unique. Dans ce cas le plan n'a pas de risque d'être mal estimé.

```
RECOMPILE
```

Force la recompilation systématique de l'instruction. Utile à l'intérieur d'un objet de code. Nous le verrons plus en détail dans la section 9.2.1.

```
PARAMETERIZATION { SIMPLE | FORCED }
```

Force un auto-paramétrage. Nous verrons cet indicateur plus en détail dans la section 9.3.

```
KEEP PLAN, KEEPFIXED PLAN
```

permet de diminuer le nombre de recompilations produites sur une instruction dans un objet de code. Au fil des versions, SQL Server provoque de moins en moins de recompilation en cachant de mieux en mieux les instructions individuellement. `KEEP PLAN` diminue le nombre de recompilations dues à des changements de cardinalité (utile pour une table temporaire, qui va provoquer une recompilation après une insertion première de six lignes), `KEEPFIXED PLAN` empêche une recompilation due à des changements de statistiques de distribution. Nous le verrons plus en détail dans la section 8.3.1.

```
EXPAND VIEWS
```

Indique que les index d'une vue indexée ne doivent pas être utilisés pour satisfaire la requête, mais que le plan doit se calculer sur les tables sous-jacentes. L'indicateur de table `WITH(NOEXPAND)` provoque l'effet inverse : seuls les index de la vue sont considérés.

```
MAXRECURSION nombre_de_récursions
```

Limite le nombre d'appels récursifs dans une expression de table (*common table expression*, CTE) récursive.

```
USE PLAN N'xml_plan'
```

Vous permet de « coller » un plan d'exécution en format XML. Cet indicateur ne peut être utilisé que dans un `SELECT`. C'est l'indicateur le plus dangereux, car il fixe tout le plan d'exécution dans le marbre. Il suffit qu'un élément de la structure de table qui y est référencé soit supprimé, pour provoquer une erreur d'exécution. De même, une fonctionnalité de plan modifiée dans une version ultérieure invalide aussi le plan. N'utilisez cette fonctionnalité qu'en dernier ressort, quand par exemple vous connaissez un meilleur plan, généré par une version antérieure de SQL Server, et que la mise à jour a dégradé les performances. Vous pouvez récupérer le plan à l'aide de `SET SHOWPLAN_XML`, et le passer à la requête. Envoyez ce plan en UNICODE, avec le N devant le littéral, pour éviter des conversions de caractère indésirables à l'enregistrement du plan. Le plan doit pouvoir être validé par le schéma

Showplanxml.xsd, disponible dans le répertoire d'installation de SQL Server, ou sur le site de Microsoft.

Une option de session : SET FORCEPLAN { ON | OFF }, force également les plans d'exécution de deux façons : les instructions exécutées dans la session respecteront l'ordre d'apparition des tables dans la clause FROM, et l'opérateur de jointure en boucle imbriquée (*nested loop*) sera partout utilisé, à moins qu'un indicateur force un autre algorithme. Autant dire que cette option est à laisser à OFF.

Indicateurs de table

Les indicateurs de table s'ajoutent après la déclaration de table, dans une clause FROM, comme ceci :

```
SELECT FirstName, LastName
FROM Person.Contact WITH (READUNCOMMITTED);
```

Nous en listons ici les principaux. La plupart concernent les verrouillages, nous ne les abordons que succinctement, entrant plus en détail sur le sujet dans le chapitre 7.

- FORCESEEK : exclusivement en SQL Server 2008 : force une stratégie de *seek* d'index plutôt qu'un *scan*.
- NOEXPAND : force l'utilisation des index sur une vue indexée.
- INDEX (index_val [,...n]) :force l'utilisation d'un index (même s'il n'a rien à voir avec le critère de filtre de la clause WHERE.
- HOLDLOCK : force le niveau d'isolation SERIALIZABLE pour la table.
- NOLOCK : force le niveau d'isolation READ UNCOMMITTED pour la table.
- NOWAIT : désactive l'attente de verrous. Si la table est verrouillée à l'exécution de la requête, SQL Server renvoie une erreur immédiatement.
- PAGLOCK : force le choix d'une granularité de verrous par pages, au lieu d'une granularité par ligne ou par table.
- READCOMMITTED : force le niveau d'isolation READ COMMITTED pour la table (niveau d'isolation par défaut de SQL Server), utilise le *row versioning* si la base est en mode READ_COMMITTED_SNAPSHOT.
- READCOMMITTEDLOCK : force le niveau d'isolation READ COMMITTED pour la table, sans utiliser le row versioning même si la base est en mode READ_COMMITTED_SNAPSHOT.
- READPAST : force la lecture d'une table sans attendre la libération des verrous incompatibles. Cela peut donc entraîner une lecture incomplète, mais rapide (on n'obtient donc que l'argent du beurre, mais pas forcément le beurre). Par exemple, le code suivant ne va lire que quatre lignes sur cinq (les nombres 1, 2, 4, 5) :

```
CREATE TABLE dbo.ReadPast (nombre int not null)

INSERT INTO dbo.ReadPast (nombre) VALUES (1)
INSERT INTO dbo.ReadPast (nombre) VALUES (2)
```

```
INSERT INTO dbo.ReadPast (nombre) VALUES (3)
INSERT INTO dbo.ReadPast (nombre) VALUES (4)
INSERT INTO dbo.ReadPast (nombre) VALUES (5)

BEGIN TRAN
UPDATE dbo.ReadPast SET nombre = -nombre
WHERE nombre = 3

-- dans une autre session
SELECT *
FROM dbo.ReadPast WITH (READPAST)
```

- `READUNCOMMITTED` : force le niveau d'isolation `READ UNCOMMITTED` pour la table, strictement équivalent à `NOLOCK`.
- `REPEATABLEREAD` : force le niveau d'isolation `REPEATABLE READ` pour la table. Ce n'est donc intéressant que dans une transaction explicite.
- `ROWLOCK` – force le choix d'une granularité de verrous par ligne (granularité par défaut de SQL Server), empêche donc normalement l'escalade. Sans garantie. Par exemple, en niveau d'isolation `SERIALIZABLE`, des verrous plus larges doivent de toute façon être posés.
- `SERIALIZABLE` : force le niveau d'isolation `SERIALIZABLE` pour la table, strictement équivalent à `HOLDLOCK`.
- `TABLOCK` : force le choix d'une granularité de verrou par table. Pose un verrou partagé (S).
- `TABLOCKX` : force le choix d'une granularité de verrou par table. Pose un verrou exclusif (X).
- `UPDLOCK` : force un verrouillage de mise à jour (U).
- `XLOCK` : force un verrouillage exclusif (X).

8.2.2 Guides de plan

Les guides de plan permettent de forcer des indicateurs de requêtes sur une instruction sans la modifier directement. Ils sont utiles pour appliquer des indicateurs à des requêtes sur lesquelles vous n'avez pas la main. Vous créez simplement un **guide de plan** à l'aide de la procédure stockée système `sp_create_plan_guide`, et vous modifiez, désactivez ou activez ce guide par la procédure `sp_control_plan_guide`. Vous pouvez créer trois types de guides :

- un guide pour une instruction dans un **OBJET** de code : procédure stockée, fonction utilisateur, déclencheur ;
- un guide **SQL**, pour des requêtes ad hoc ;
- un guide de type **TEMPLATE**, pour modifier le comportement d'autoparamétrage d'une classe de requêtes. Ce type ne permet que de modifier l'indicateur `PARAMETERIZATION` de la requête.

Vous spécifiez le type dans le paramètre `@type` à la création du guide. Voici un exemple d'utilisation de guide de plan :

```
CREATE PROCEDURE dbo.GetContactsForPlanGuide
   @LastName nvarchar(50) = NULL
AS BEGIN
   SET NOCOUNT ON

   SELECT FirstName, LastName
   FROM Person.Contact
   WHERE LastName LIKE @LastName;
END
GO

EXEC dbo.GetContactsForPlanGuide 'Abercrombie'

EXEC dbo.GetContactsForPlanGuide '%'
-- pas bon
GO

EXEC sys.sp_create_plan_guide
   @name = N'Guide$GetContactsForPlanGuide$OptimizeForAll',
   @stmt = N'SELECT FirstName, LastName
             FROM Person.Contact
             WHERE LastName LIKE @LastName',
   @type = N'OBJECT',
   @module_or_batch = N'dbo.GetContactsForPlanGuide',
   @params = NULL,
   @hints = N'OPTION (OPTIMIZE FOR (@LastName = ''%''))'

EXEC dbo.GetContactsForPlanGuide '%'
-- mieux !
GO
-- suppression
SELECT * FROM sys.plan_guides
EXEC sys.sp_control_plan_guide
   @Operation = N'DROP',
   @Name = N'Guide$GetContactsForPlanGuide$OptimizeForAll'
```

Dans cet exemple, nous avons créé une procédure stockée qui est exécutée la première fois avec un paramètre provoquant une compilation de plan d'exécution utilisant un *seek* d'index, par *parameter sniffing* (voir section 9.2.1). Le deuxième appel utilisera un plan d'exécution très défavorable. Nous créons ensuite un guide de plan qui utilise l'indicateur `OPTIMIZE FOR` pour forcer une compilation prenant en compte une valeur de paramètre générique (un scénario du pire, qui sera désagréable pour les valeurs de paramètre qui profiteraient d'un index, à cause de leur grande sélectivité, mais qui limitera les dégâts en cas d'envoi de paramètre trop peu sélectif). Pour supprimer le guide de plan, nous utilisons le paramètre `@Operation = N'DROP'` en appel de `sp_control_plan_guide`. Nous pourrions aussi désactiver le plan par la valeur `'DISABLE'`.

En SQL Server 2008, les événements de trace `Performance : Plan Guide Successful Event` et `Performance : Plan Guide Unsuccessful Event` vous indiquent les plans d'exécutions créés à l'aide d'un guide, ou si la création du plan a échoué. De même, dans SSMS, vous trouvez la liste des guides de plan dans l'explorateur d'objets, sous le nœud « Programmability ».

La vue système `sys.plan_guides` liste les guides de plan enregistrés dans la base courante.

8.3 OPTIMISATION DU CODE SQL

Un SGBDR est un serveur dont le travail est d'assurer un stockage des données optimal et cohérent. Il est maître de ses données. SQL est un langage de requête (le nom est bien choisi) dont l'expression est déclarative : *via* une requête SQL, le programmeur décrit le résultat désiré. Par exemple, une requête comme celle-ci :

```
SELECT FirstName, LastName
FROM Person.Contact
WHERE Title = 'Mr.'
ORDER BY LastName, FirstName;
```

est purement « descriptive » : on y demande une liste des prénoms et noms de contacts, lorsque leur titre est « Mr. », triés par nom et prénom. À aucun moment, on indique au serveur *comment* produire cette liste. Tout à fait comme lorsque vous allez dans une librairie commander un livre, vous demandez simplement à votre libraire de passer commande d'un ouvrage, vous ne lui dites pas : « pourriez-vous faire une recherche dans votre catalogue de livres distribués, pour noter le numéro ISBN du titre que je souhaite obtenir, pour le copier-coller dans votre programme de commande, et y inscrire mon nom. Ensuite, merci d'imprimer un bon de commande pour le faire parvenir par voie postale au distributeur de ces éditions. »

Votre libraire est bien plus qualifié que vous pour effectuer cette tâche, et d'ailleurs, si vous le forciez à utiliser votre méthode, la commande prendrait peut-être plus de temps, ou même ne serait-elle jamais faite correctement. SQL Server fonctionne de la même façon. Si la syntaxe du langage est déclarative, son exécution sera finalement procédurale : le moteur d'optimisation, qui fait partie du moteur relationnel, est un bijou d'algorithmique. La stratégie de plan d'exécution se base sur les connaissances dont dispose le moteur d'optimisation sur la structure et le contenu des tables et des vues. Le travail du programmeur SQL est d'écrire la requête la plus descriptive possible, pour donner l'information précise de ce que la requête doit obtenir.

Durant la phase de compilation, le moteur relationnel a une grande capacité à normaliser les syntaxes équivalentes pour produire l'arbre d'exécution. Par exemple, ces requêtes produisent (presque) le même plan (sur SQL Server 2005) :

```
SELECT c.FirstName, c.LastName
FROM Person.Contact c
CROSS JOIN HumanResources.Employee e
```

```
WHERE c.ContactId = e.ContactId
GO
SELECT c.FirstName, c.LastName
FROM Person.Contact c
JOIN HumanResources.Employee e ON c.ContactId = e.ContactId
GO
SELECT c.FirstName, c.LastName
FROM Person.Contact c
WHERE EXISTS (SELECT *
FROM HumanResources.Employee e WHERE c.ContactId = e.ContactId)
GO
SELECT c.FirstName, c.LastName
FROM Person.Contact c
WHERE c.ContactId IN (SELECT e.ContactId
FROM HumanResources.Employee e)
GO
```

Les deux premiers plans sont identiques, les deux suivants aussi, qui incluent un tri (*distinct sort*). Le nombre de reads est identique, et considérant la faible volumétrie (290 lignes retournées), les performances sont, dans les faits, les mêmes, même si les deux derniers plans sont plus coûteux. Il n'est donc souvent pas utile de choisir une syntaxe légèrement différente d'une autre pour améliorer les performances. Testez les plans d'exécution, et choisissez la syntaxe la plus intuitive et la plus lisible.

De même, l'ordre d'apparition des colonnes dans le `SELECT`, des tables dans la clause `FROM` et des expressions dans les clauses `WHERE` et `HAVING`, n'a pas d'importance. La requête sera de toute manière décomposée et récrite durant la phase de compilation. Sur certains SGBDR, l'ordre des tables dans la jointure influe sur les performances. Ce n'est pas le cas en SQL Server. Par contre, l'ordre des colonnes dans les clauses `GROUP BY` et `ORDER BY` peut modifier les performances, en influant sur les tris, opérations coûteuses.

Écriture ensembliste

Nous l'avons dit, le langage SQL est déclaratif. Il est toutefois aussi ensembliste : toutes les instructions définissent et travaillent sur des ensembles de données, avec parfois des entorses à l'algèbre relationnelle (la clause `ORDER BY` par exemple est une hérésie pour le modèle relationnel, où les *tuples* d'une relation n'ont aucun ordre précis. SQL permet de générer un jeu de résultat trié, ce qui viole la théorie relationnelle, mais est très utile). Il est donc important d'éviter le plus possible des syntaxes non déclaratives et non ensemblistes, le plus triste exemple étant le curseur. Les boucles `WHILE` entrent dans ce cas de figure.

Nous avons également dit que, bien que le langage soit ensembliste, l'exécution est procédurale : le moteur de stockage doit finalement parcourir les lignes ou les clés d'un index une à une. Ce simple exemple suffit à la démontrer :

```
DECLARE @i int;
SET @i = 0;
SELECT @i = @i + 1 FROM Person.Contact;
SELECT @i;
```

après exécution, `@i` vaut 19973, ce qui est la cardinalité de la table `Person.Contact`.

Pour des considérations de performances, cela signifie une chose : même si la requête est écrite en pur SQL, elle peut être plus ou moins efficace selon la stratégie décidée par l'optimiseur. Certaines constructions SQL permettent des stratégies optimales, d'autres forcent le plan à utiliser des opérations proches d'un fonctionnement de curseur. C'est notamment le cas de syntaxes introduites dans SQL Server 2005, comme la clause `PIVOT` et les fonctions de fenêtrage. Leur utilité est donc à balancer avec leur coût, et elles sont à bannir autant que possible, surtout lorsqu'il existe une syntaxe plus traditionnelle. Prenons un exemple. Nous voulons obtenir une liste des contacts avec, pour chaque ligne, le nombre total de personnes portant le même nom de famille. Nous pouvons utiliser une sous-requête ou une fonction de fenêtrage :

```
-- SELECT avec sous-requête
SELECT
    t.FirstName,
    t.LastName,
    (SELECT COUNT(*)
     FROM Person.Contact
     WHERE LastName = t.LastName) cnt
FROM Person.Contact t
GO
-- SELECT avec fonction de fenêtrage
SELECT
    t.FirstName,
    t.LastName,
    COUNT(*) OVER (PARTITION BY LastName) as cnt
FROM Person.Contact t
GO
```

Nous voyons le résultat de la trace sur la figure 8.10. La requête utilisant la sous-requête a nécessité environ 1 200 lectures, alors que celle utilisant la fonction de fenêtrage a lu 47 000 pages, et en a écrit 12. La raison de ces lectures et écriture est la création d'une table de travail (opérateur *table spool*). À l'heure actuelle, dans SQL Server 2005, la première requête est donc bien plus performante.

EventClass	TextData	CPU	Reads
SQL:BatchCompleted	-- SELECT avec sous-requête SELECT...	31	1192
SQL:BatchCompleted	-- SELECT avec fonction de fenêtrag...	187	47270

Figure 8.10 — Différence de performances

Ce choix entre une syntaxe ou une autre n'a pas de valeur absolue, et c'est pour cette raison que nous disons « à l'heure actuelle ». Comme l'optimiseur SQL évolue, il est possible que le plan d'exécution de la fonction de partitionnement s'améliore dans le futur. Il paraît toutefois peu probable qu'il finisse par dépasser en performances la syntaxe utilisant la sous-requête. La règle de base qu'on peut en déduire est

celle-ci : lorsque c'est possible, choisissez la syntaxe la plus « intuitive » et la plus « traditionnelle ». La clarté aide l'optimiseur, et les syntaxes plus anciennes, bien connues, ont plus de chance de bénéficier d'optimisations solides.

Comparaisons et filtres

Dans la clause `WHERE` d'une requête, comme dans la clause `ON` d'une jointure, peuvent êtres exprimées des expressions de filtre. Ces expressions retournent un résultat en logique ternaire : VRAI, FAUX ou INCONNU. Elles sont composées d'un opérateur et de deux opérandes (à gauche et à droite de l'opérateur). La plupart du temps, un des opérandes au moins est une colonne de table. L'autre opérande est la valeur recherchée. Par exemple :

```
SELECT LastName, FirstName
FROM Person.Contact
WHERE LastName LIKE 'Al%';
```

Si la colonne recherchée est indexée, l'optimiseur va peut-être choisir d'utiliser l'index dans le plan d'exécution, selon la sélectivité de celui-ci. Si les estimations de l'optimiseur lui montrent qu'un *scan* de la table ou d'un index sera moins coûteux en lectures qu'une recherche dans les clés de l'index, le *scan* sera choisi. Cela dépend du nombre de lignes retournées par la requête. Quoi qu'il en soit, en l'absence d'un index utile, SQL Server sera obligé d'exécuter un *scan*. Considérez donc la création d'index sur les colonnes présentes dans vos expressions de filtre, et testez après création si l'index est utilisé par la requête. Cette règle est bien entendu valable pour les expressions de jointure :

```
SELECT c.LastName, c.FirstName, a.AddressLine1, a.PostalCode, a.City
FROM Person.Contact c
JOIN HumanResources.Employee e ON c.ContactId = e.ContactId
JOIN HumanResources.EmployeeAddress ea ON e.EmployeeId = ea.EmployeeId
JOIN Person.Address a ON ea.AddressId = a.AddressId;
```

Dans cet exemple, les expressions de la clause `ON` des jointure effectuent des recherches sur les colonnes tout comme la clause `WHERE`. Elles bénéficient donc de la présence d'index. Les clés des tables mères sont déjà indexées : en règle générale, une jointure est effectuée entre une ou plusieurs colonnes liées par une contrainte de clé étrangère (bien que rien dans le langage SQL ne force la jointure à s'appuyer sur une telle définition de modèle). La clé étrangère ne peut s'appuyer que sur une clé primaire ou unique de la table mère. Ces clés créent nécessairement des index. Par exemple, La colonne `ContactId` de la table `Person.Contact` est clé primaire *clustered*. Par contre la création de clé étrangère en SQL Server ne force en rien la création d'un index sur la clé déportée dans la table fille. C'est à vous de le faire, et c'est pratiquement une étape nécessaire pour assurer la bonne performance des requêtes de jointure.

Pour qu'une expression de filtre utilise l'index, il est impératif que la colonne soit présente comme opérande, sans aucune modification. C'est ce qu'on appelle un SARG (*Search ARGument*). Un *seek* d'index ne peut être évalué comme possibilité que sur un opérande de type SARG, c'est-à-dire qui correspond exactement à ce qui

est stocké dans la clé de l'index. Toute modification de colonne dans l'expression, comme par exemple une concaténation, un passage dans une fonction, ou un changement de collation à la volée, empêchera toute utilisation d'index, et forcera par conséquent le *scan*. C'est donc à éviter. Exemples de choses à ne pas faire :

```
SELECT FirstName, LastName
FROM Person.Contact
WHERE LastName COLLATE Latin1_General_CI_AI = 'Jimenez';

SELECT FirstName, LastName
FROM Person.Contact
WHERE LEFT(LastName, 2) = 'AG';

SELECT FirstName, LastName
FROM Person.Contact
WHERE LastName + FirstName = 'AlamedaLili';
```

écrivez donc vos filtres plutôt comme ceci :

```
SELECT FirstName, LastName
FROM Person.Contact
WHERE LastName LIKE 'Jim[eé]nez';

SELECT FirstName, LastName
FROM Person.Contact
WHERE LastName LIKE 'AG%';

SELECT FirstName, LastName
FROM Person.Contact
WHERE LastName = 'Alameda' AND FirstName = 'Lili';
```

L'opérateur `LIKE` permet l'utilisation de l'index, à condition que le début de la chaîne soit fixe. Il correspond ainsi au début de la clé de l'index, et est donc SARGable, comme on peut chercher efficacement dans un annuaire en ne connaissant que les quelques premières lettres d'un nom.

Cette règle est valable aussi pour les opérations arithmétiques. SQL Server n'est pas capable de modifier une expression contenant des opérations sur des constantes pour la simplifier, une opération connue en compilation sous le nom de *constant folding*. Prenons un exemple :

```
SELECT *
FROM Person.Contact
WHERE ContactId + 3 = 34;

SELECT *
FROM Person.Contact
WHERE ContactId = 37;
```

Il n'est pas très difficile pour un compilateur de transformer automatiquement la première clause en la seconde. Pourtant, SQL Server ne le fait pas, comme nous le voyons dans les plans d'exécution en figure 8.11.

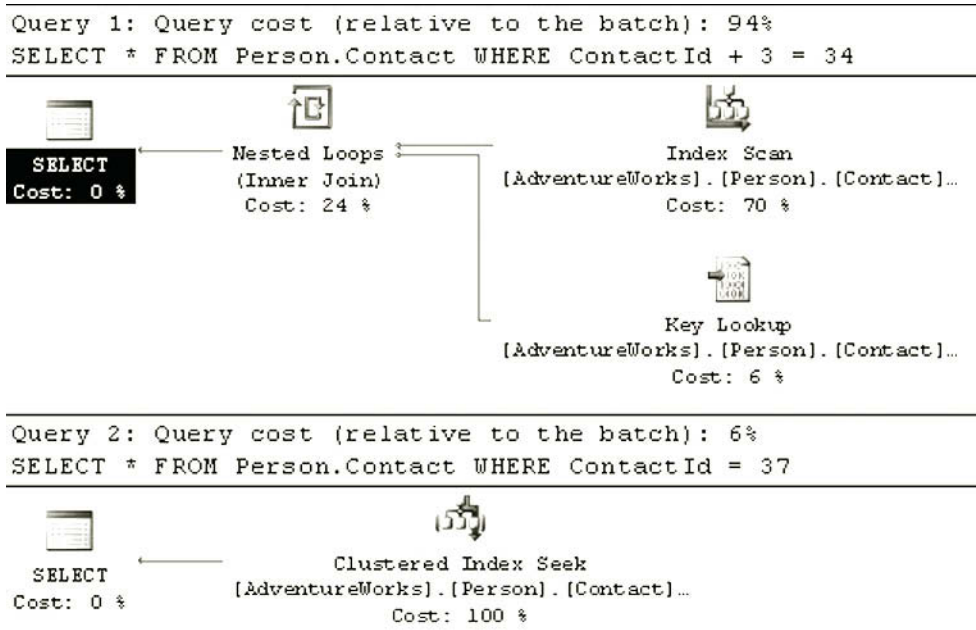

Figure 8.11 — Différence de plans d'exécution

La règle à suivre est donc celle-ci : une des deux opérandes doit être la colonne, seulement la colonne, et rien que la colonne.

Dans la liste des opérateurs disponibles, l'égalité (=) est le plus utilisé. C'est celui qui offre le plus de possibilité d'optimisation par un index. L'opérateur de différence (<>) est souvent le moins efficace, non pas parce qu'il empêche en tant que tel l'utilisation d'un index, mais parce qu'il provoque souvent la sélection d'un nombre important de lignes. En réalité, les opérateurs qui provoquent une recherche de plage, sont à manier avec précaution, car ils peuvent entraîner le retour d'un grand nombre de lignes. Ces opérateurs sont `<`, `>`, `<=`, `>=`, `BETWEEN` et `<>`.

Grandeur et misère des fonctions utilisateur

Les **fonctions utilisateur** (*User Defined Function*, UDF) sont des objets de code, au même titre que les procédures stockées et les déclencheurs. Leur travail est de retourner une valeur scalaire ou une table au code T-SQL appelant. Elles peuvent sembler très pratiques au premier abord, car elles permettent de réaliser le rêve de tout programmeur : la diminution du code redondant par modularisation. Avons-nous besoin de faire une recherche répétitive sur une table, pourquoi ne pas encapsuler cette recherche dans une fonction ? Parce que ce qui paraît à première vue une bonne idée pour le programmeur se révèle une très mauvaise idée pour un SGBDR.

Nous pourrions songer à modulariser notre code de la façon suivante : au lieu d'utiliser des sous-requêtes ou des jointures, nous serions tentés d'aller chercher les

valeurs ou les calculs qui nous intéressent à l'aide d'une fonction. En reprenant notre exemple du compte de noms de famille, essayons de modulariser le code, pour réutiliser ce calcul dans toutes circonstances. Nous créons alors la fonction suivante :

```
CREATE FUNCTION Person.GetCountContacts (@LastName nvarchar(50))
RETURNS int
AS BEGIN
   RETURN (SELECT COUNT(*)
           FROM Person.contact
           WHERE LastName LIKE @LastName)
END;
```

Nous profitions ainsi d'une fonction qui retourne le compte des contacts pour toute la table (paramètre '%'), par début de nom (paramètre 'A%' par exemple, ou par nom (paramètre 'Ackerman' par exemple) :

```
SELECT Person.GetCountContacts('%');
SELECT Person.GetCountContacts('A%');
SELECT Person.GetCountContacts('Ackerman');
```

Pratique, non ? Peut-être dans ce contexte-là, mais gardons-nous d'inclure cette fonction dans la clause `SELECT` d'une requête. Observons les différences de performances entre deux façons différentes d'obtenir les mêmes résultats :

```
-- SELECT avec utilisation de la fonction
SELECT
    t.FirstName,
    t.LastName,
    Person.GetCountContacts(t.LastName) as cnt
FROM Person.Contact t
GO
-- SELECT avec sous-requête
SELECT
    t.FirstName,
    t.LastName,
    (SELECT COUNT(*)
     FROM Person.Contact
     WHERE LastName = t.LastName) cnt
FROM Person.Contact t
GO
```

Nous avons tracé ces requêtes. Voyons le résultat sur la figure 8.12.

EventClass	TextData	CPU	Reads	Duration	Writes
SQL:BatchCompleted	-- SELECT avec utilisation de la fonction SELEC...	2687	46282	8632	0
SQL:BatchCompleted	-- SELECT avec sous-requête SELECT t.FirstNam...	125	1206	1053	0

Figure 8.12 — Trace des requêtes

La requête utilisant la fonction dure huit secondes et lit près de 46 000 pages, alors que la solution avec sous-requête, une seconde (125 millisecondes de temps CPU) pour 1 200 pages lues. La différence est sans appel. Pourquoi cela ? Parce

qu'avec la fonction, nous forçons SQL Server à utiliser la stratégie de requête que *nous* avons décidée. Pour chaque ligne de la table `Person.Contact` impliquée dans la requête, SQL Server doit entrer dans la fonction et exécuter la requête qui s'y trouve à nouveau. Puisque nous avons séparé le code en deux objets, l'optimiseur n'a aucun moyen de lier les deux requêtes pour comprendre ce qu'on veut obtenir. Ainsi la nature déclarative du langage SQL est violée. En revanche, en incluant avec une sous-requête les deux opérations dans la même requête, l'optimiseur a tout en main pour la décortiquer, la comprendre, et l'optimiser. Regardons le plan d'exécution (un peu nettoyé pour faciliter sa lecture) de l'ordre utilisant la sous-requête :

```
|--Compute Scalar
     |--Hash Match(Right Outer Join ...)
          |--Compute Scalar ...
          |    |--Stream Aggregate( … Count(*)))
          |         |--Index Scan(OBJECT:(...
                        [nix$Person_Contact$LastName]), ...)
          |--Clustered Index Scan(OBJECT:(...
             [PK_Contact_ContactID] AS [t]))
```

La conclusion est simple : méfiez-vous des fonctions.

Comptage des lignes d'une table

Pour compter le nombre total de lignes d'une table, vous n'êtes pas obligé de faire un `COUNT`. Vous pouvez vous appuyer sur les informations de métadonnées, qui sont, autant que nous avons pu en faire l'expérience, toujours à jour. Évidemment, cette astuce n'est valable que si vous voulez la cardinalité de toute la table, non filtrée. Démonstration :

```
SELECT COUNT(*)
FROM Person.Contact;
GO

SELECT SUM(row_count) as row_count
FROM sys.dm_db_partition_stats
WHERE
    object_id=OBJECT_ID('Person.Contact') AND
    (index_id=0 or index_id=1);
GO
```

Résultat en *reads* :

```
Table 'Contact'. Scan count 1, logical reads 46, … -- COUNT(*)
Table 'sysidxstats'. Scan count 2, logical reads 4, …
    -- sys.dm_db_partition_stats
```

Plan d'exécution : figure 8.13.

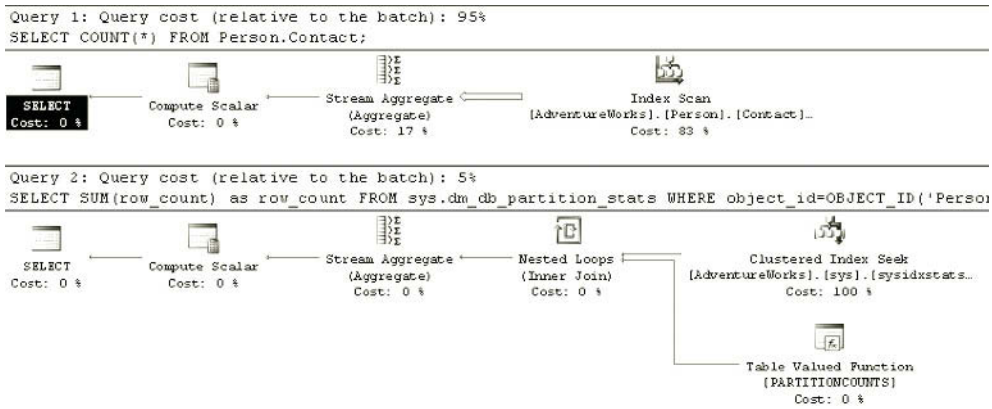

Figure 8.13 — Plans d'exécution

8.3.1 Tables temporaires

Les **tables temporaires**, bien qu'elles soient déclarées dans `tempdb`, restent en mémoire, dans le *buffer*, jusqu'à ce qu'elles atteignent une taille qui nécessite leur écriture sur disque.

À propos des variables de type table

SQL Server 2000 a introduit la variable de type `table`, c'est-à-dire la possibilité de stocker une table dans une variable Transact-SQL. Cette facilité est utile pour renvoyer une table à partir d'une fonction utilisateur. Elle peut aussi être utilisée à l'intérieur d'une procédure stockée pour remplacer une table temporaire. Vous entendrez parfois dire que la variable de type `table` est bien plus légère qu'une table temporaire car elle n'est pas écrite dans `tempdb` et ne vit qu'en mémoire. C'est erroné. La variable de type `table` est elle aussi stockée dans `tempdb`, comme une table temporaire. SQL Server lui assigne un nom interne. Il est très simple de le démontrer :

```
SELECT * FROM tempdb.INFORMATION_SCHEMA.TABLES
GO

DECLARE @t TABLE (id int)

SELECT * FROM tempdb.INFORMATION_SCHEMA.TABLES
GO
```

Vous verrez dans le deuxième apparaître une nouvelle référence de table dans les métadonnées de `tempdb`, avec un nom ressemblant à ceci (exemple lors de notre exécution) : #04CA11FE.

Notez que si vous ne séparez pas les deux tests par un `GO`, vous verrez la table dans les deux `SELECT`, le `DECLARE` étant placé en premier par la compilation du code SQL.

Comme pour une table temporaire, la table contenue dans la variable réside en mémoire jusqu'au moment où elle atteint une certaine taille. Elle est ensuite écrite dans `tempdb`. Le gain apporté par les variables de type `table` est donc limité. Y a-t-il alors une différence entre les deux options ? Oui, la variable de type `table` est plus rapide principalement parce qu'elle consomme moins de ressources pour la maintenir : il n'y a pas de calcul de statistiques de colonnes sur une variable de type `table` par exemple, le verrouillage des lignes est moindre, et ne dure que le temps de l'instruction de modification, de même que la transaction de modification, même si elle est journalisée (il suffit pour le constater d'utiliser la fonction `fn_dblog()` que nous avons déjà utilisée), n'est pas enrôlée dans une transaction explicite, elle s'exécute et est validée automatiquement dans sa propre transaction « privée ». Ces avantages peuvent aussi se révéler des inconvénients, selon l'utilisation qu'on veut faire de la variable de type `table`. L'absence de calcul de statistiques sur les colonnes peut être contre-productive sur des variables de type `table` volumineuses. Comme l'optimiseur n'a aucune statistique, son plan d'exécution va se baser sur l'estimation du retour de zéro ou d'une seule ligne, quelle que soit la réalité. On peut le vérifier simplement, par exemple en regardant le plan d'exécution estimé de cette requête (qui retourne en réalité 1 763 lignes sur ma version de la base resources) :

```
DECLARE @t TABLE (Name SYSNAME)
INSERT @t SELECT name FROM sys.system_objects

SELECT * FROM @t
```

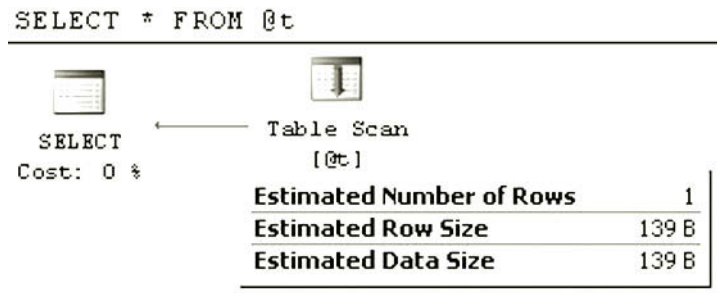

Figure 8.14 — Estimation requête table variable
Le plan d'exécution généré peut donc se révéler bien moins performant qu'avec une table temporaire.

Il est de même impossible de créer des index sur une variable (à part un index lié à une contrainte : déclaration de clé primaire ou de clé unique, possibles unique-

ment à la création de la table), comme il est impossible d'y appliquer après création toute commande DDL. Ainsi, les variables ne doivent pas être utilisées pour remplacer des tables temporaires volumineuses sur lesquelles on effectue beaucoup d'opérations de recherche.

Nous l'avons dit, les opérations sur la variable ne sont pas enrôlées dans une transaction. La différence est simple à expérimenter :

```
USE tempdb
GO

CREATE TABLE #t (Name SYSNAME)

BEGIN TRAN
INSERT #t SELECT name FROM [master].[dbo].[sysobjects]

ROLLBACK

SELECT * FROM #t

GO

DECLARE @t TABLE (Name SYSNAME)

BEGIN TRAN
INSERT @t SELECT name FROM [master].[dbo].[sysobjects]
ROLLBACK

SELECT * FROM @t

GO
```

Dans la première partie du code, nous utilisons une table temporaire, dans laquelle nous insérons des lignes à l'intérieur d'une transaction explicite. Après l'annulation de la transaction (`ROLLBACK`), la table est vide : l'insertion a été annulée. En revanche, l'application du même code à la variable de type `table` montre que, même après le `ROLLBACK`, les lignes insérées sont toujours dans la variable. Bien que bénéfique pour les performances, ce comportement est évidemment dangereux : si vous codez des transactions impliquant des variables de type `table`, vous risquez d'obtenir des résultats inattendus.

En conclusion, Microsoft recommande de remplacer les tables temporaires par des variables autant que possible. Précisons donc qu'elles ne sont intéressantes que lorsqu'elles sont de petite taille, pour des opérations simples, et bien entendu, lorsqu'on ne peut les remplacer par des jointures, des sous-requêtes ou des expressions de table.

8.3.2 Pour ou contre le SQL dynamique

Le sujet fait débat. Que penser du SQL dynamique ? C'est ainsi qu'est appelée la possibilité de générer une instruction SQL, plus ou moins complexe, dans une

variable de type VARCHAR, pour ensuite l'évaluer et l'exécuter dynamiquement à l'aide du mot-clé EXEC(UTE). Par exemple :

```
DECLARE @sql varchar(8000)
SET @sql = 'SELECT * FROM Person.Contact'
EXEC (@sql)
```

Cette syntaxe particulière est fréquemment utilisée pour construire, à l'intérieur de procédures stockées, des requêtes complexes, notamment pour permettre une combinaison de multiples critères de recherches dans la clause WHERE. Imaginons le cas d'un site web qui permet, dans une page de formulaire, d'effectuer une recherche multicritères sur des produits. Chaque argument de recherche correspond à une colonne, le tout sur une ou plusieurs tables de notre base. L'internaute peut saisir ou non une valeur pour chaque argument. Une procédure stockée reçoit en paramètre la liste des arguments, qu'elle utilise pour filtrer la requête. Il n'y a pratiquement que deux choix : écrire une requête traditionnelle qui prend en compte tous les cas de figure, comme ceci :

```
SELECT FirstName, MiddleName, LastName, Suffix, EmailAddress
FROM Person.Contact
WHERE
   (LastName = @LastName OR @LastName IS NULL) AND
   (FirstName = @FirstName OR @FirstName IS NULL) AND
   (MiddleName = @MiddleName OR @MiddleName IS NULL) AND
   (Suffix = @Suffix OR @Suffix IS NULL) AND
   (EmailAddress = @EmailAddress OR @EmailAddress IS NULL)
ORDER BY LastName, FirstName
```

ou générer son code SQL de façon dynamique, avec une construction comme celle-ci :

```
DECLARE @sql varchar(8000)

SET @sql = '
   SELECT FirstName, MiddleName, LastName, Suffix, EmailAddress
   FROM Person.Contact
   WHERE '
IF @FirstName IS NOT NULL
   SET @sql = @sql + ' FirstName = ''' + @FirstName + ''' AND '
IF @MiddleName IS NOT NULL
   SET @sql = @sql + ' MiddleName = ''' + @MiddleName + ''' AND '
IF @LastName IS NOT NULL
   SET @sql = @sql + ' LastName = ''' + @LastName + ''' AND '
IF @Suffix IS NOT NULL
   SET @sql = @sql + ' Suffix = ''' + @Suffix + ''' AND '
IF @EmailAddress IS NOT NULL
   SET @sql = @sql + ' EmailAddress = ''' + @EmailAddress + ''' AND '

SET @sql = LEFT(@sql, LEN(@sql)-3)
EXEC (@sql)
```

Le **SQL dynamique** a ses défauts : intégré dans une procédure stockée, il annule en quelque sorte les avantages de précompilation de la procédure. En effet, la chaîne

exécutée ne fait plus partie de la procédure stockée. Son évaluation et son exécution dynamique sont prises en charge par le moteur relationnel comme les autres requêtes ad hoc.

Sécurité

Une autre contrainte vient de la sécurité : le SQL dynamique n'obéit plus aux règles de chaînage de propriétaires. En d'autres termes, les privilèges doivent être vérifiés sur les objets contenus dans le SQL dynamique, et l'utilisateur exécutant la procédure doit donc être autorisé sur ces objets. Ceci pour éviter les risques d'injection SQL, c'est-à-dire d'envoi de code malicieux. Vous pouvez créer votre procédure avec la clause `EXECUTE AS` pour l'exécuter dans un contexte d'utilisateur ayant des privilèges sur ces objets. Soyez néanmoins très prudents sur les risques d'injections (voyez par exemple cet article, pour vous prévenir contre elles : http://fr.wikipedia.org/wiki/Injection_SQL).

À quoi bon faire une procédure stockée si c'est pour la faire agir comme un code client ? Mais dans le cas de figure du formulaire de recherche, cet inconvénient se révèle plutôt un avantage. La non-réutilisation du plan d'exécution permet d'éviter l'écueil dans lequel nous tombons dans le cas de l'exemple de requête statique utilisant le bricolage (`LastName = @LastName OR @LastName IS NULL`). Nous avons créé deux procédures stockées dont vous trouverez le source sur le site d'accompagnement du livre. Elles sont nommées `Person.SearchContactsStatique` et `Person.SearchContactsDynamique`, et elles implémentent la même requête selon les deux solutions proposées. Essayons deux appels :

```
EXEC Person.SearchContactsStatique @LastName = 'Adams'
GO
EXEC Person.SearchContactsDynamique @LastName = 'Adams'
GO

EXEC Person.SearchContactsStatique
      @LastName = 'Adams', @MiddleName = 'S'
GO
EXEC Person.SearchContactsDynamique
      @LastName = 'Adams', @MiddleName = 'S'
GO
```

Sur la figure 8.15, nous voyons la différence de performance entre les deux solutions, lors du deuxième appel. Elle est très parlante. La solution dynamique gagne haut la main. Nous voyons aussi un CacheHit : le code ad hoc de la requête dynamique s'est tout de même inséré dans le cache de plans.

Sur les figures 8.16 et 8.17, nous voyons respectivement le plan de la première procédure – qui n'a pas changé, et le plan de la seconde – qui s'est adapté.

EventClass	TextData	CPU	Reads	Duration
Trace Start				
SP:CacheMiss	EXEC Person.SearchContactsStatique @LastName = 'Adams', @Mi...			
SP:CacheHit				
Showplan XML	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/...			
SQL:BatchCompleted	EXEC Person.SearchContactsStatique @LastName = 'Adams', @Mi...	0	357	21
SP:CacheMiss	EXEC Person.SearchContactsDynamique @LastName = 'Adams', @M...			
SP:CacheHit				
SP:CacheHit	SELECT FirstName, MiddleName, LastName, Suffix, EmailAd...			
Showplan XML	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/...			
SQL:BatchCompleted	EXEC Person.SearchContactsDynamique @LastName = 'Adams', @M...	0	12	7

Figure 8.15 — Différence de performances

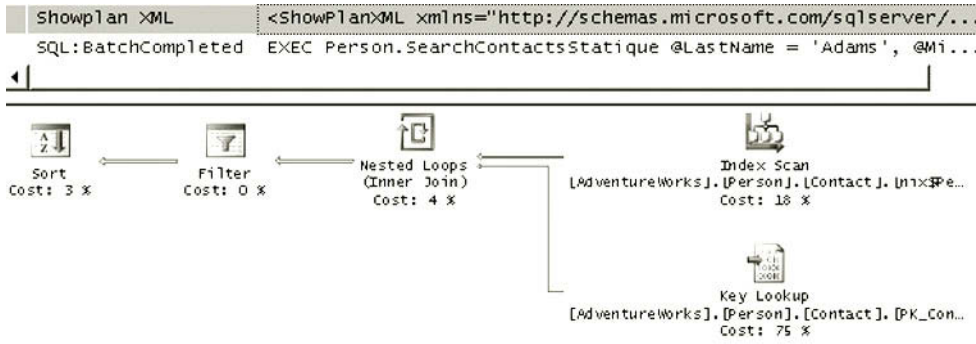

Figure 8.16 — Plan de la procédure statique

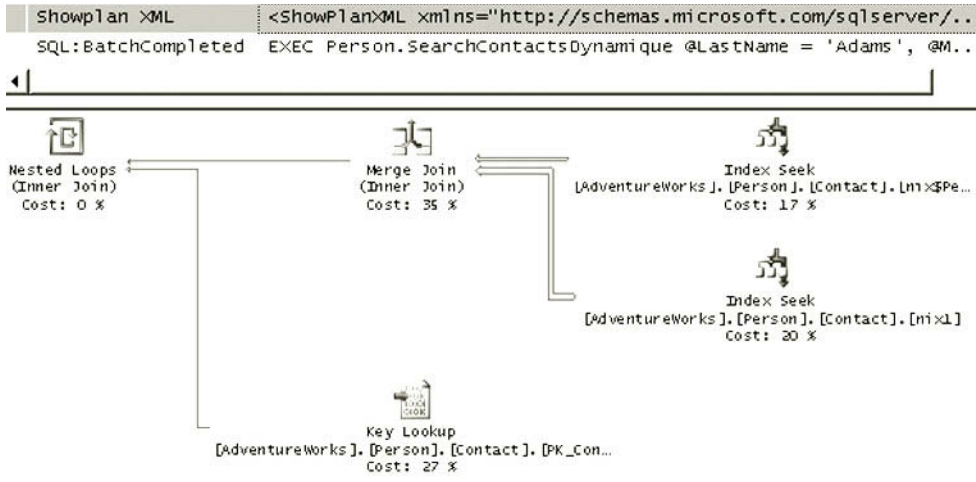

Figure 8.17 — Plan de la procédure dynamique

Nous pourrions conserver la syntaxe de la première procédure, et forcer la recompilation, mais la requête, comportant de multiples variables, reste difficile à analyser. Parfois, la syntaxe dynamique, malgré ses défauts, en terme de sécurité et de lisibilité, est intéressante pour les performances.

8.3.3 Éviter les curseurs

Le curseur est une construction T-SQL permettant de traiter séquentiellement un jeu de résultat côté serveur. Parce qu'il viole la syntaxe ensembliste du langage SQL, et ainsi court-circuite toute possibilité d'optimisation par SQL Server, il est l'ennemi public numéro un des performances du code SQL. En présence de **curseurs**, la meilleure décision d'optimisation à prendre est de les remplacer par du code ensembliste, ce qui est la plupart du temps possible. Bien que les curseurs soient en eux-mêmes « optimisables » (par le choix de curseurs en type `READ ONLY` ou `FAST_FORWARD` notamment), nous pensons préférable de présenter des moyens de s'en passer.

La plupart des curseurs qu'on peut rencontrer dans du code de production sont créés par manque de connaissance des possibilités du langage SQL. Le code d'administration est un problème différent. On y rencontre des curseurs pour automatiser des maintenances sur les objets, ce qui est acceptable, compte tenu de la fréquence d'utilisation, et du choix de fenêtres d'administration favorables.

Souvent, les curseurs sont utilisés pour effectuer des traitements ou des vérifications complexes sur les lignes. Ils peuvent en général être remplacés par de simples jointures, ou un passage par une table temporaire. Le but est de les récrire en code ensembliste. Si le traitement ligne par ligne est néanmoins nécessaire, il est souvent plus intéressant d'effectuer une boucle `WHILE` avec des `SELECT`, qui se révèlent plus efficaces. Voici un exemple simple de curseur, et son remplacement par une boucle `WHILE` :

```
-- curseur
DECLARE cur CURSOR FAST_FORWARD
FOR SELECT ContactId FROM Person.Contact ORDER BY ContactId

DECLARE @CurrentContactID int
OPEN cur

FETCH NEXT FROM cur INTO @CurrentContactID
WHILE (@@fetch_status <> -1)
BEGIN
   IF (@@fetch_status <> -2)
      PRINT @CurrentContactID

   FETCH NEXT FROM cur INTO @CurrentContactID
END

CLOSE cur
DEALLOCATE cur
GO
```

```
-- boucle
DECLARE @CurrentContactID int

SELECT TOP 1 @CurrentContactID = ContactId
FROM  Person.Contact ORDER BY ContactId

WHILE 1 = 1 BEGIN
    PRINT @CurrentContactID

    SELECT TOP 1 @CurrentContactID = ContactID
    FROM  Person.Contact
    WHERE ContactID > @CurrentContactID

    IF @@ROWCOUNT = 0 BREAK

END -- WHILE
GO
```

Mais les boucles ne sont pas toujours nécessaires. SQL est un langage déclaratif et ensembliste, mais l'exécution de la requête par le moteur d'exécution est au final séquentielle : il faut bien finir par parcourir les lignes l'une après l'autre. Cette caractéristique peut être mise à profit. Voici par exemple comment réaliser une concaténation :

```
DECLARE @str VARCHAR(MAX)

SELECT @str = COALESCE(@str + ', ', '') + LastName
FROM Person.Contact
GROUP BY LastName
ORDER BY LastName

SELECT @str
```

À l'aide des fonctions SQL et des jointures, vous pouvez aussi réaliser des séparations et des pivots. Par exemple :

```
CREATE TABLE #citations (
    auteur varchar(50),
    phrase varchar (1000)
)

INSERT INTO #citations
SELECT 'Guitry', 'Il y a des gens sur qui on peut compter. Ce sont généralement
des gens dont on n''a pas besoin' UNION ALL
SELECT 'Cioran', 'Un homme ennuyeux est un homme incapable de s''ennuyer' UNION
ALL
SELECT 'Talleyrand', 'Les mécontents, ce sont des pauvres qui réfléchissent'

SELECT auteur,
    NullIf(SubString(' ' + phrase + ' ' , id ,
    CharIndex(' ' , ' ' + phrase + ' ' , id) - ID) , '') AS mot
FROM (SELECT ROW_NUMBER() OVER (ORDER BY NEWID()) as id
FROM sys.system_views) tally
CROSS JOIN #citations
```

```
WHERE id <= Len(' ' + Phrase + ' ') AND SubString(' ' + Phrase + ' ' , id - 1, 1)
= ' '
```

Cette requête place chaque mot rencontré dans la citation sur une nouvelle ligne, dans la colonne mots. Elle se base sur une jointure avec une table de nombres, que nous avons ici générée à la volée. Une table de nombre est toujours utile, faites-en une, à l'aide de `ROW_NUMBER()`. Pour plus de précision sur cette méthode de découpage, lisez cet article : http://www.sqlteam.com/article/parsing-csv-values-into-multiple-rows.

Un autre besoin courant est de produire des totaux cumulés. Cette opération peut être plus rapide effectuée par un curseur que par une requête ensembliste, c'est donc un contre-exemple[1]. Le code ensembliste est à essayer toutefois sur des petits totaux cumulés. Voici un exemple de code :

```
SELECT
    th1.TransactionID,
    th1.ActualCost,
    SUM(th2.ActualCost) AS TotalCumule
FROM Production.TransactionHistory th1
JOIN Production.TransactionHistory th2 ON th2.TransactionID <= th1.TransactionID
    AND th2.TransactionDate = '20030901'
WHERE th1.TransactionDate = '20030901'
GROUP BY th1.TransactionID, th1.ActualCost
ORDER BY th1.TransactionID
```

Récursivité

Avant SQL Server 2005, la récursivité ensembliste n'était pas possible (en tout cas sur des recherches récursives dont on ne connaissait pas la profondeur). L'expression de table (*Common Table Expression*, CTE) résout ce problème. Exemple d'appel récursif pour descendre dans une hiérarchie classique employé/manager :

```
WITH employeeCTE AS
(
    SELECT e.EmployeeId, c.FirstName, c.LastName, 1 as niveau,
        CAST(N'lui-même' as nvarchar(100)) as boss
    FROM HumanResources.Employee e
    JOIN Person.Contact c ON e.ContactID = c.ContactID
    WHERE e.ManagerID IS NULL

    UNION ALL

    SELECT e.EmployeeId, c.FirstName, c.LastName, niveau + 1,
        CAST(m.FirstName + ' ' + m.LastName as nvarchar(100))
    FROM HumanResources.Employee e
    JOIN Person.Contact c ON e.ContactID = c.ContactID
    JOIN employeeCTE m ON m.EmployeeId = e.ManagerId
)
SELECT FirstName, LastName, niveau, boss FROM EmployeeCTE;
```

1. Voir cette entrée de blog d'Adam Machanic, dont l'exemple de code est tiré : http://sql-blog.com/blogs/adam_machanic/archive/2006/07/12/running-sums-redux.aspx

En SQL Server 2008, vous disposez du type de données `HierarchyID`, qui est un type complexe implémenté en classe .NET. Il permet de stocker une information de hiérarchie, et de retrouver ces informations à travers de méthodes. Malgré le fait que ce soit un type .NET, il est plus rapide que l'utilisation de CTE. De plus, on peut indexer ses propriétés si nécessaire, en passant par une colonne calculée. Pour vous donner une idée de l'utilisation de ce type, voici un exemple de requête, qui fait à peu près la même chose que la CTE précédente :

```
SELECT o1.EmployeeName, o1.EmployeeID.GetLevel() as level,
    o2.EmployeeName as boss
FROM HumanResources.Organization o1
JOIN HumanResources.Organization o2
    ON o2.EmployeeID = o1.EmployeeID.GetAncestor(1)
WHERE hierarchyid::GetRoot().IsDescendantOf(o1.EmployeeId)= 1[1]
```

Vous trouvez des exemples d'utilisation dans les BOL 2008, sous l'entrée « *Working with hierarchyid Data* ».

Une dernière méthode, très efficace, implique une structuration de la table spécifique, en représentation intervallaire. Vous en trouvez la description dans cet article de Frédéric Brouard : http://sqlpro.developpez.com/cours/arborescence/.

Mises à jour

Un `UPDATE` s'effectue lui aussi ligne par ligne, même si l'instruction est ensembliste. Comme vous pouvez affecter une valeur à une variable dans un `UPDATE`, et mélanger cette affectation avec une mise à jour de colonne, vous pouvez générer une forme détournée de boucle, qui vous évitera un curseur. Exemple très simple :

```
CREATE TABLE dbo.testloop (
   id int NOT NULL IDENTITY (1,1) PRIMARY KEY CLUSTERED,
   nombre int NULL,
   groupe int NOT NULL
)
GO

INSERT INTO dbo.testloop (groupe)
SELECT TOP (10) 1 FROM sys.system_objects  UNION ALL
SELECT TOP (20) 2 FROM sys.system_objects  UNION ALL
SELECT TOP (15) 3 FROM sys.system_objects
GO

DECLARE @i int
SET @i = 0

UPDATE dbo.testloop
SET @i = @i + 1,
    nombre = @i
GO
```

1. La méthode IsDescendantOf s'appelait IsDescendant en version beta (CTP) 6. Elle a été renommée en IsDescendantOf.

```
SELECT * FROM dbo.testloop
```

Cette méthode est relativement peu sûre, car elle se base sur une implémentation physique non documentée, et potentiellement non consistante au fil des versions. Elle est néanmoins pratique. Depuis SQL Server 2005, les fonctions de fenêtrage permettent de réaliser la même opération, avec en plus la capacité de gérer précisément un ordre, et un regroupement par la valeur d'autres colonnes (c'est pour cette raison que nous avons créé la colonne `groupe` dans la table). Le problème est que nous ne pouvons utiliser une fonction de fenêtrage que dans un `SELECT`. Nous pouvons contourner cette limitation à l'aide d'une expression de table :

```
WITH cteTestLoop AS (
   SELECT
      id,
      ROW_NUMBER() OVER(ORDER BY id) as rownumber
   FROM dbo.testloop
)
UPDATE tl
SET nombre = ctl.rownumber
FROM dbo.testloop tl
JOIN cteTestLoop ctl ON tl.id = ctl.id;
```

Cet exemple effectue exactement la même mise à jour, mais avec garantie de l'ordre par la colonne `ID`. Cette méthode est officielle et sûre, par contre vous pourriez encore utiliser la solution à base de variable dans les cas où le tri ne vous importe pas, pour des raisons de performance pure. Voyez la différence de plan d'exécution sur la figure 8.18. La version CTE doit créer un plan compliqué, avec une jointure sur la même table, un tri et une table de travail (*table spool*).

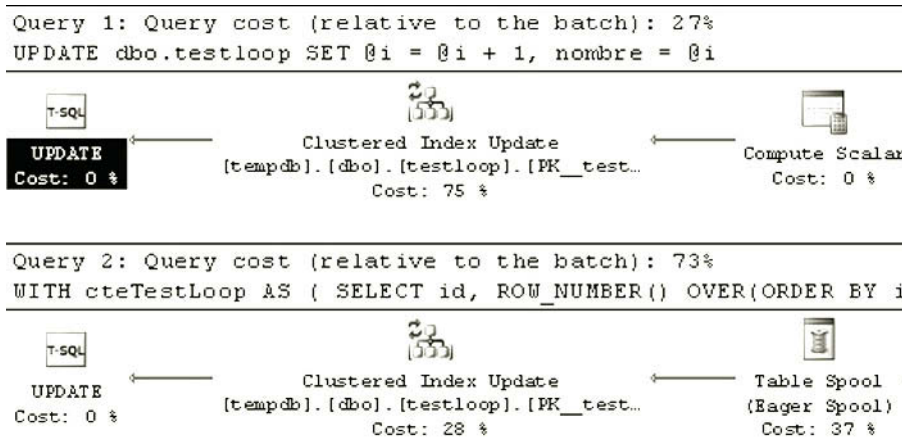

Figure 8.18 — Plan des requêtes de mise à jour

Mais il faut aussi dire qu'avec la fonction de partitionnement il est facile de recommencer la numérotation à chaque groupe, en modifiant la fonction `ROW_NUMBER()` dans la CTE ainsi :

```
WITH cteTestLoop AS (
   SELECT
      id,
      ROW_NUMBER() OVER(PARTITION BY groupe ORDER BY id) as rownumber
   FROM dbo.testloop
) …
```

Ce qui serait difficile à imiter dans notre « bricolage », surtout pour garantir un tri correct.

> La fonction `ROW_NUMBER()` demande obligatoirement un `ORDER BY`. Vous pouvez simplement utiliser n'importe quelle colonne, ou, si vous n'avez pas de colonne pour générer cet ordre, vous pouvez utiliser l'astuce suivante : `ROW_NUMBER() OVER(ORDER BY (SELECT 0))`.

8.3.4 Optimisation des déclencheurs

Les **déclencheurs** (*triggers*) sont des blocs de code SQL qui sont exécutés au déclenchement d'une opération DML sur une table : `INSERT`, `UPDATE` ou `DELETE`. Depuis SQL Server 2005, existent aussi des déclencheurs DDL, qui s'exécutent lors de toute modification de structures. Ils n'entrent pas dans le cadre de l'optimisation du système, nous ne les aborderons donc pas ici.

Du point de vue de la compilation, les déclencheurs se comportent comme des procédures stockées : leur plan d'exécution est mis en cache dans le cache de procédure à leur premier rappel, et réutilisé ensuite. Il n'y a donc pas recompilation systématique, sauf si le code lui-même provoque des recompilations en modifiant le contexte. Il est important de garder le code à l'intérieur du déclencheur aussi court et simple que possible : en effet, il s'exécute dans la transaction de l'instruction qui l'a déclenché, et la prolonge donc d'autant, ce qui augmente la durée des verrous.

En SQL Server, les déclencheurs sont uniquement ensemblistes (par opposition à certains SGBDR qui implémentent des déclencheurs « *PER ROW* »), cela veut dire qu'ils se déclenchent une seule fois par instruction, quel que soit le nombre de lignes affectées. Il est donc essentiel de penser en terme de requêtes ensemblistes dans un déclencheur. Il est très commun de voir du code tel que celui-ci, dans un *trigger* :

```
CREATE TRIGGER atr_d$sales_currency$archive
ON sales.currency
AFTER DELETE
AS BEGIN
   DECLARE @CurrencyCode NCHAR(3),
           @Name NVARCHAR(50),
           @DeletedDate smalldatetime
```

```
    SELECT  @CurrencyCode = CurrencyCode,
            @Name = Name,
            @DeletedDate = CURRENT_TIMESTAMP
    FROM Deleted

    INSERT INTO sales.currencyArchive
       (CurrencyCode, Name, DeletedDate)
    VALUES
       (@CurrencyCode, @Name, @DeletedDate)
END
```

Bien entendu, ceci ne fonctionne correctement que si une seule ligne est supprimée. Si toute la table `sales.currency` est vidée, une seule ligne sera inscrite dans la table `sales.currencyArchive`. C'est une erreur fréquente, méfiez-vous en.

Dans le contexte du déclencheur, deux pseudo-tables sont disponibles, pour retrouver les lignes affectées par l'instruction : `DELETED` contient les lignes supprimées par un `DELETE` ou un `UPDATE`, et `INSERTED` les lignes ajoutées par un `INSERT` ou un `UPDATE`. L'`UPDATE` est considéré logiquement comme un `DELETE` suivi d'un `INSERT`. Ces pseudo-tables sont gérées en interne par des versions de ligne stockées dans le *version store* de `tempdb` (voir section 4.4). De ce fait, le déclenchement d'un *trigger* provoquera une copie de version et de l'activité dans `tempdb`. De même, ces pseudo-tables n'ont pas d'index, et les recherches se feront toujours par *scan* (vous verrez dans le plan d'exécution des opérateurs particuliers, avec leur coût relatif : « *Inserted scan* » et « *Deleted scan* »). Évitez donc les déclencheurs sur des tables massivement modifiées.

Le plan d'exécution du déclencheur est visible *via* l'affichage des plans dans SSMS, que ce soir le plan estimé (donc aussi *via* `SET SHOWPLAN_XML`) ou le plan réel. Par contre, `SET STATISTICS IO ON` n'affiche pas les statistiques de lectures et d'écriture des pseudo-tables.

Depuis SQL Server 2005, le `ROLLBACK` à l'intérieur d'un *trigger*, pour annuler la transaction de l'instruction déclenchante, envoie une erreur 3609 dans la session, afin d'avertir l'utilisateur. Vous pouvez placer le `ROLLBACK` dans un bloc `TRY CATCH` pour éviter cette erreur.

Bonne pratique

En réalité, l'erreur 3609 en envoyée si `@@TRANCOUNT` est égal à 0 à la sortie du déclencheur. Elle apparaît donc aussi si un `COMMIT TRANSACTION` sans `BEGIN TRANSACTION` est exécuté dans le *trigger*, ce qui est sans intérêt. Déclencher l'erreur 3609 a aussi un sens en cas de `ROLLBACK`. Il est relativement dangereux d'exécuter un `ROLLBACK` dans un *trigger*, car cette instruction annule toute la chaîne des transactions, jusqu'à la transaction la plus externe. Ce n'est peut-être pas ce que vous souhaitez, dans la mesure où vous n'avez jamais de garantie que le code déclenchant le *trigger* ne gère pas de transaction explicite. De même, un `ROLLBACK` fermera les éventuels curseurs créés dans le code appelant, sauf s'ils sont `STATIC` ou `INSENSITIVE` et si `CURSOR_CLOSE_ON_COMMIT` est à `OFF`. Enfin, si vous faites un `ROLLBACK`, n'oubliez pas de le faire suivre par un `RETURN`, pour éviter d'exécuter ensuite du code qui serait en dehors de la transaction, et donc validé.

Il est donc plus prudent d'effectuer le `ROLLBACK` dans le code appelant, ou d'utiliser le mécanisme des points de sauvegarde, à l'aide d'un `SAVE TRANSACTION nom_du_point;`, et ensuite d'un `ROLLBACK TRANSACTION nom_du_point;`, qui effectivement n'annule que cette partie de la transaction.

Un *trigger* est déclenché quel que soit le nombre de lignes affectées. Une instruction DML avec une clause de filtre qui ne retourne aucun *match* fera donc exécuter le déclencheur. Comme ceci par exemple :

```
DELETE FROM sales.currency WHERE CurrencyCode = 'FRF';
-- ou plus simplement :
DELETE FROM sales.currency WHERE 1 = 0;
```

Il est bon de tester cela au début du *trigger*, pour éviter l'exécution du code en pure perte. La variable `@@ROWCOUNT` est disponible à l'intérieur du *trigger* pour cela :

```
ALTER TRIGGER atr_d$sales_currency2$archive
ON sales.currency2
AFTER DELETE
AS BEGIN
   IF @@ROWCOUNT = 0 RETURN;
   ...
```

Pour éviter d'exécuter le code de votre déclencheur si les colonnes qui vous intéressent ne sont même pas mises à jour, servez-vous des fonctions `UPDATE()` et `COLUMNS_UPDATED()`, qui vous indiquent quelles colonnes ont été mentionnées dans l'instruction qui déclenche le *trigger*. Référez-vous au BOL pour la syntaxe de ces fonctions, mais notez qu'elles n'indiquent pas si une colonne est réellement mise à jour (si sa valeur a changé), mais si elle est *mentionnée* dans l'instruction DML.

Un `SET NOCOUNT ON` est aussi une bonne idée.

Un déclencheur, comme tout bon événement, interagit le moins possible avec la session, et principalement ne renvoie pas de jeu de résultat. C'est inutile et dangereux pour l'application cliente, qui peut être parasitée par un recordset indésirable. Vous pouvez désactiver le renvoi de `SELECT` malencontreusement présents à l'intérieur d'un *trigger* à l'aide de l'option de serveur `'disallow results from triggers'` :

```
EXEC sp_configure 'show advanced options', 1;
RECONFIGURE;
EXEC sp_configure 'disallow results from triggers', 1;
RECONFIGURE;
EXEC sp_configure 'show advanced options', 0;
RECONFIGURE;
```

Si le but est d'afficher ou de rediriger le résultat d'une opération, songez à utiliser la clause `OUTPUT`, plus légère, plutôt qu'un déclencheur : disponible dans les instructions `INSERT`, `UPDATE` et `DELETE`, cette clause rend visible les pseudo-tables `DELETED` et `INSERTED`. Exemple pour l'archivage :

```
DELETE FROM sales.currency2
OUTPUT deleted.CurrencyCode, deleted.Name, CURRENT_TIMESTAMP
INTO sales.currencyArchive;
```

Sans la partie INTO, OUTPUT retourne le jeu de résultat à la session. Attention, dans ce cas il ne peut y avoir de déclencheur pour la même instruction.

Nous avons pour l'instant traité de déclencheur de type AFTER, c'est-à-dire s'exécutant *après* la modification de données. Il existe un autre type de déclencheur : INSTEAD OF. Celui-ci s'exécute non pas avant l'instruction (il n'existe pas de *trigger* BEFORE en SQL Server), mais *à la place*. Si vous voulez que l'instruction qui a déclenché le *trigger* s'exécute réellement, *vous devez la réécrire* dans le corps du *trigger*, en vous basant sur le contenu des pseudo-tables. Ce type de déclencheur est utile pour réorienter le résultat d'une instruction, ou ne traiter qu'une partie de celle-ci. Un de ses intérêts principaux est qu'il peut être placé sur une vue, permettant alors une gestion fine des mises à jour à travers la vue.

Enfin, ajoutons que souvent les déclencheurs sont utilisés pour effectuer des contrôles de cohérence. Autant que possible, préférez les contraintes CHECK aux *triggers*. Les contraintes CHECK sont trop souvent sous-estimées. Elles peuvent contenir des expressions complexes portant sur toutes les colonnes de la ligne. Si vous devez effectuer des contrôles à partir de données hors de la table, vous pouvez tricher en passant par une fonction utilisateur. Si vous devez faire des vérifications sur un ensemble de lignes de la table, par contre le déclencheur sera probablement plus optimal, de par son comportement ensembliste. La fonction utilisateur, elle, devra être appelée ligne par ligne. Voici, par exemple, un code de *trigger* permettant de vérifier que des dates de validité d'historique ne se chevauchent pas (attention, ce code présuppose que les colonnes fromDate et toDate n'acceptent pas les NULL) :

```
ALTER TRIGGER atr_iu$sales_CurrencyHistory$checkConsistency
ON sales.CurrencyHistory
FOR INSERT, UPDATE
AS BEGIN
   IF @@ROWCOUNT = 0 RETURN
   SET NOCOUNT ON

   IF EXISTS (
      SELECT 1
      FROM sales.CurrencyHistory ch WITH (READUNCOMMITTED)
      JOIN inserted i
         ON ch.CurrencyCode = i.CurrencyCode AND
            ch.fromDate <> i.fromDate
      WHERE
         ( ch.fromDate BETWEEN i.fromDate AND i.toDate OR
           ch.toDate   BETWEEN i.fromDate AND i.toDate ) OR
         ( i.fromDate  BETWEEN ch.fromDate AND ch.toDate OR
           i.toDate    BETWEEN ch.fromDate AND ch.toDate )
         )
   BEGIN
      RAISERROR ('Des dates d''historique se chevauchent !', 16, 10)
      RETURN
   END
END
```

Limiter les *pollings* avec les notifications de requête

Si vous utilisez le client natif SQL (*SQL Native Client*, ou SNAC - SQLNCLI), vous pouvez profiter de la fonctionnalité de notifications de requêtes (*Query Notifications*). Elle est très utile pour mettre à jour automatiquement les données de référence dans votre application ou code site web. Souvent, des données de références sont chargées dans des objets de l'application client. Ces données de références changent rarement, mais pour être sûr d'être à jour, le code client effectue des requêtes régulières de vérification, au cas où le contenu de la table aurait changé. Query Notifications est fondé sur Service Broker, et permet aux applications de s'abonner à des événements qui lui seront envoyés par le serveur, à travers le client natif. Le code client doit simplement utiliser un objet `SqlDependency`, sur lequel il place un gestionnaire d'événement. Vous trouverez plus d'informations dans les BOL, sous l'entrée « *Using Query Notifications* ».

9

Optimisation des procédures stockées

Objectif

Il existe deux types de code SQL : la requête élaborée par le client et envoyée au serveur est appelée une requête ad hoc, souvent une requête saisie manuellement ou construite dynamiquement, et l'objet de code stocké, dont le plus courant est la procédure stockée. La procédure stockée offre de nets avantages de performances. Nous allons expliquer pourquoi, et vous donner les outils pour optimiser vos procédures.

La procédure stockée est un objet de code stocké sur le serveur, créé à l'aide de la commande `CREATE PROCEDURE`. Il est hors du domaine de ce livre de vous en présenter la syntaxe : nous allons simplement nous concentrer sur quelques éléments avancés qui vous permettront d'optimiser leur utilisation.

Autant que possible, préférez des **procédures stockées** à du code SQL généré du côté client, même pour des requêtes aussi simples qu'un unique `SELECT`. Les procédures stockées permettent d'encapsuler et de centraliser du code sur le serveur, qui n'aura besoin d'être modifié qu'à un seul endroit en cas de changement de logique ou de structure de données. Elles permettent également de gérer la sécurité : grâce au principe de chaînage de propriétaire (*ownership chaining*, voir ce terme dans les BOL), vous pouvez ne donner que les privilèges d'exécution sur une procédure, sans autoriser l'accès aux objets sous-jacents, ce qui permet de contrôler précisément les points d'entrée et de lecture des données. L'appel d'une procédure stockée est également plus concis que l'envoi d'un long code SQL, ce qui diminue le trafic réseau.

Enfin, la procédure stockée est précompilée et conservée dans un cache de procédures, en mémoire, ce qui évite des recalculs de plans d'exécution. Nous détaillerons cet élément.

9.1 DONE_IN_PROC

Par défaut, SQL Server retourne un message au client après chaque exécution d'instruction SQL, indiquant le nombre de lignes affectées. Vous le voyez dans l'onglet « messages » de SSMS, lorsque vous exécutez la moindre commande. Ce message prend la forme : « `(14 row(s) affected)` » (pour une langue de session us_english). Ce message est indépendant des jeux de résultats eux-mêmes, il est envoyé dans un paquet TDS à travers le réseau à chaque exécution d'une instruction, qu'elle soit dans un *batch* ou qu'elle fasse partie d'une procédure stockée. Cela provoque des allers-retours réseau inutiles. Dans la plupart des cas ce message `done_in_proc` n'est pas utilisé par le client (la fonction ODBC `SQLRowCount()` l'utilise, mais pas la propriété `RecordCount` de l'objet `ADODB.Recordset`). Vous pouvez donc obtenir un gain de performance réel en désactivant l'envoi des *tokens* `done_in_proc`, surtout pendant l'exécution de procédures stockées complexes.

Pour cela, vous avez plusieurs solutions. La plus simple est de placer, en début de vos procédures stockées, l'instruction `SET NOCOUNT ON`, comme ceci :

```
CREATE PROCEDURE dbo.DoSomethingUseful
AS BEGIN
    SET NOCOUNT ON
       ...
```

Ce qui désactive le renvoi des messages `done_in_proc` pour le reste de la session. Vous pouvez également désactiver globalement ces messages pour toutes les sessions, en modifiant le paramètre de serveur `'user options'` :

```
EXEC sp_configure 'user options', 512
RECONFIGURE
```

ou à l'aide du drapeau de trace 3640 à ajouter à la ligne de commande de démarrage du serveur SQL, dans les paramètres du service (propriété « *Startup Parameters* » du service Dans *SQL Server Configuration Manager* (voir section 7.5.2).

Malgré cela, c'est une bonne idée d'écrire systématiquement la commande `SET NOCOUNT ON` au début de toutes vos procédures stockées, car l'option a pu être remise à `OFF` dans la session (vous pouvez notamment configurer SSMS pour initialiser cette option à l'ouverture de session, voyez la fenêtre de propriétés de la requête, dans la page « *advanced* »).

Enfin, vous pouvez tester quel est l'état de cette option de la façon suivante (exemple d'utilisation) :

```
IF @@OPTIONS & 512 = 512
   PRINT 'SET NOCOUNT est à ON';
```

9.2 MAÎTRISE DE LA COMPILATION

Un avantage important de la procédure stockée, est la réutilisation de son plan d'exécution. Nous allons détailler ce mécanisme. Ci-après, nous ne parlerons que de procédures, mais gardez à l'esprit que le mécanisme est le même pour les fonctions utilisateur (UDF) et pour les déclencheurs, qui sont précompilés de la même manière.

Lorsque la procédure est créée *via* une instruction `CREATE PROCEDURE`, son code source est stocké dans une table système de métadonnées de la base courante. Vous pouvez retrouver ce code grâce à la vue système `sys.sql_modules` :

```
SELECT definition
FROM sys.sql_modules
WHERE object_id = OBJECT_ID('dbo.uspGetBillOfMaterials')
```

Cette requête retourne la définition de la procédure nommée `dbo.uspGetBillOfMaterials`. La vue `sys.sql_modules` interroge des tables systèmes qui ne peuvent plus être requêtées directement depuis SQL Server 2005. Si vous êtes curieux de savoir comment ces tables systèmes s'appellent, vous pouvez retrouver la définition de cette vue (comme de tous les objets système) à l'aide d'un de ces commandes :

```
SELECT OBJECT_DEFINITION(OBJECT_ID('sys.sql_modules'))
```

ou

```
SELECT definition FROM sys.system_sql_modules
WHERE Object_Id = OBJECT_ID('sys.sql_modules')
```

Ce stockage n'implique en rien une compilation, ou une optimisation. Il faut distinguer en SQL Server la phase de compilation, à proprement parler la vérification des privilèges de l'utilisateur et de l'existence des objets, de l'optimisation. L'optimisation est la génération d'un plan d'exécution pour le code SQL. Lorsque la procédure est créée, rien de tout cela n'est fait, même pas la vérification de l'existence des objets : on peut créer une procédure qui référence des tables qui n'existent pas, par les vertus de la fonctionnalité de « résolution de nom différée » (*Deferred Name Resolution*). Cette résolution différé ne s'applique qu'aux tables (et aux vues, qui sont syntaxiquement et conceptuellement identiques aux tables), tout autre objet référencé, comme une procédure stockée appelée par un `EXECUTE`, ou une colonne d'une table, doivent exister.

Ainsi, à la création de la procédure, on peut dire que seule une vérification syntaxique du code SQL est effectuée. La vérification de l'existence des tables, comme l'optimisation, ne seront réalisées que lors de la première exécution de la procédure stockée. Par « première exécution », nous entendons première exécution depuis le démarrage de l'instance SQL. Le plan d'exécution généré pour la procédure lors de sa première exécution sera stocké en mémoire vive, dans ce qu'on appelle le cache de plans, ou cache de procédures. Lors des exécutions ultérieures de la procédure, ce plan sera utilisé, ce qui économise l'étape d'optimisation. Le plan reste dans le cache jusqu'au redémarrage du service. Il est également possible que, à cause d'une pression

sur la mémoire, le cache se nettoie. Il ne contient au maximum que deux versions d'un plan pour une procédure : la version « sérielle » (monoprocesseur) et éventuellement la version parallélisée. De plus, ce plan est stocké sans information d'utilisateur, nous verrons dans la section 9.1.2, ce que cela implique.

Le cache de procédure peut être examiné à l'aide d'une vue de gestion dynamique : `sys.dm_exec_cached_plans`. Alliée aux fonctions `sys.dm_exec_sql_text()` et `sys.dm_exec_query_plan()`, elle vous permet d'observer en détail ce qui réside dans le cache :

```
SELECT cp.usecounts, cp.size_in_bytes, st.text,
    DB_NAME(st.dbid) as db,
    OBJECT_SCHEMA_NAME(st.objectid, st.dbid) + '.'
        + OBJECT_NAME(st.objectid, st.dbid) as object,
    qp.query_plan, cp.cacheobjtype, cp.objtype
FROM sys.dm_exec_cached_plans cp
CROSS APPLY sys.dm_exec_sql_text(cp.plan_handle) st
CROSS APPLY sys.dm_exec_query_plan(cp.plan_handle) qp
```

Cette requête ne fonctionne qu'à partir du Service Pack 2 de SQL Server 2005, qui introduit la fonction `OBJECT_SCHEMA_NAME()`, et améliore la fonction `OBJECT_NAME()`, lui permettant de passer l'identifiant de base de données.

En exécutant cette requête, vous constaterez que la colonne `cp.objtype` contient différents types de requêtes, et pas seulement des procédures stockées. Nous reviendrons sur la capacité de SQL Server à cacher d'autres plans d'exécution plus loin.

Vous pouvez constater en observant cette requête que la vue `sys.dm_exec_cached_plans` retourne les colonnes `usecounts` et `size_in_bytes`, fort utiles pour juger de la taille et de l'utilité du cache. Bien entendu `usercounts` indique le nombre de réutilisation du plan d'exécution, et `size_in_bytes` sa taille en mémoire. Le cache d'une procédure complexe peut prendre plusieurs mégaoctets. La taille indiquée ne dépend d'ailleurs pas seulement de la complexité du plan d'exécution : elle varie aussi selon le nombre d'exécutions simultanées de la procédure ou du batch, car, comme nous le verrons, des plans en rapport avec le contexte d'exécution sont générés à l'exécution à partir de ce plan compilé, et eux-mêmes cachés. On aura compris une chose : un serveur SQL fortement sollicité, et qui exécute des requêtes complexes, gagnera fortement à avoir le plus de mémoire de travail possible. L'architecture 64 bits apporte un réel plus dans ce cas de figure.

Dès que le plan d'exécution de la procédure est en cache, il sera réutilisé à chaque appel ultérieur, économisant ainsi le calcul coûteux du plan d'exécution. Vous pouvez facilement observer les différences de temps d'exécution entre le premier appel d'une procédure et les suivantes avec le profiler, et notamment sur la valeur en temps CPU, qui inclut le temps de compilation.

Normalement, la procédure va rester dans le cache. Par contre, comme sur les systèmes 32 bits, la mémoire virtuelle est limitée, et comme le cache de procédure

réside dans cette portion de la mémoire (rappelons que la mémoire au-delà des 4 Go ne peut être attribuée qu'au cache de données), il peut arriver qu'une pression sur cette mémoire oblige SQL Server à nettoyer le cache[1], pour faire de la place pour la mémoire de travail. Dans ce cas, SQL Server va tout de même s'efforcer de conserver les plans les plus coûteux à recréer. Vous pouvez vous faire une idée du coût d'un plan à l'aide des colonnes `original_cost` et `current_cost` de `sys.dm_os_memory_cache_entries`. Voici un exemple de requête :

```
SELECT
    DB_NAME(st.dbid) as db,
    OBJECT_SCHEMA_NAME(st.objectid, st.dbid) + '.'
        + OBJECT_NAME(st.objectid, st.dbid) as object,
    cp.objtype, cp.usecounts, cp.size_in_bytes,
    ce.disk_ios_count, ce.context_switches_count,
    ce.pages_allocated_count, ce.original_cost, ce.current_cost
FROM sys.dm_exec_cached_plans cp
JOIN sys.dm_os_memory_cache_entries ce
    on cp.memory_object_address = ce.memory_object_address
CROSS APPLY sys.dm_exec_sql_text(cp.plan_handle) st
```

Pression sur le cache de procédures

N'oubliez pas qu'un verrou de compilation est posé sur une procédure lors de sa compilation. Il ne peut y avoir qu'une seule compilation de la même procédure à la fois, ce qui signifie que si la compilation est longue, tous les utilisateurs devront attendre la fin de la compilation. Pour éviter ce type de problèmes, n'écrivez pas de procédures trop longues ou complexes, modularisez au besoin, et assurez-vous de disposer de suffisamment de mémoire de travail (en dessous de la limite des 4 Go en 32 bits).

Si vous souhaitez vider le cache, par exemple pour réaliser des tests consistants de performances, vous disposez de la commande `DBCC FREEPROCCACHE`, qui vide entièrement le cache de procédures. Elle est en général utilisée conjointement avec `DBCC DROPCLEANBUFFERS`, pour obtenir un état de la mémoire proche d'un démarrage de l'instance SQL.

```
CHECKPOINT
DBCC DROPCLEANBUFFERS
DBCC FREEPROCCACHE
```

Il vaut mieux bien entendu éviter de lancer ces commandes sur un serveur de production, qui verrait immédiatement ses performances se dégrader jusqu'à ce que le cache soit de nouveau rempli.

Deux autres commandes DBCC, moins connues permettent de vider le cache : pour les plans qui s'appliquent à une base de données spécifique : `DBCC FLUSHPROCINDB (DBID)` :

```
SELECT DB_ID('AdventureWorks')
```

1. À travers un *memory sweep*, lancé par un algorithme de planification, dont on peut voir l'état dans la vue sys.dm_os_memory_cache_clock_hands.

```
DBCC FLUSHPROCINDB (5)
```

Pour vider plus généralement les caches de SQL Server : `DBCC FREESYSTEMCACHE('ALL')`.

> Plus d'informations sur cette entrée de blog : http://sqlblog.com/blogs/kalen_delaney/archive/2007/09/29/geek-city-clearing-a-single-plan-from-cache.aspx

Il est également à noter que le cache se vide intégralement lors de quelques opérations, qu'il faut donc éviter d'exécuter inutilement sur un serveur de production :

- détachement d'une base de données (`sp_detach_db`) ;
- utilisation de la commande `RECONFIGURE` pour appliquer les changements d'une option de serveur ;
- lorsqu'une vue est créée avec `CHECK OPTION`, toutes les entrées du cache qui référencent la base de données dans laquelle se trouve la vue, sont vidées ;
- avant SQL Server 2005 Service Pack 2, `DBCC CHECKDB` vidait le cache. Ce n'est plus le cas.

Si vous êtes confrontés à des nettoyages de cache intempestifs, consultez le journal d'erreur (ERRORLOG) de SQL Server. Depuis le Service Pack 2 de SQL Server 2005, un message d'information y est enregistré lorsque le cache se vide (aussi à l'issue d'un `DBCC FREEPROCACHE`). Vous pouvez également tracer l'événement SQL Trace `Errors and Warnings : ErrorLog`. L'événement `Security Audit : Audit DBCC Event` se déclenche également à l'exécution de toute commande DBCC.

9.2.1 Paramètres typiques

Sur quelle base le plan d'une procédure est-il calculé ? Lorsque nous passons des paramètres, leurs valeurs peuvent différer et générer des plans d'exécution différents. Qu'en est-il alors du comportement de la procédure stockée ? Malheureusement, il n'y a en ce domaine pas de miracle : toutes les instructions de la procédure sont optimisées en tenant compte des valeurs de paramètres envoyées. Le premier appel impose donc la qualité du plan d'exécution. Si la valeur des paramètres est « typique », le plan sera de bonne qualité. Par contre, si des paramètres « extrêmes » sont passés, cela peut produire de mauvaises performances lors des appels ultérieurs. Prenons l'exemple de ces deux requêtes :

```
--créons un index pour aider la recherche
CREATE NONCLUSTERED INDEX [nix$Person_Contact$LastName]
ON [Person].[Contact] (LastName)
GO

-- 2 lignes à retourner
```

```
SELECT FirstName, LastName, EmailAddress
FROM Person.Contact
WHERE LastName LIKE 'Ackerman'

-- 911 lignes à retourner
SELECT FirstName, LastName, EmailAddress
FROM Person.Contact
WHERE LastName LIKE 'A%'
```

À l'évidence, le plan d'exécution sera différent : la sélectivité de l'index est excellente pour répondre à la première requête, et SQL Server choisira un *seek* ; par contre, la deuxième requête sera résolue par un *scan*, probablement moins coûteux. Vous comprenez déjà le problème : si nous créons une procédure stockée de ce type :

```
CREATE PROCEDURE Person.GetContactByLastName
    @LastNameStart nvarchar(50)
AS BEGIN
    SET NOCOUNT ON

    SELECT FirstName, LastName, EmailAddress
    FROM Person.Contact
    WHERE LastName LIKE @LastNameStart
END
```

Nous pouvons aisément vérifier que le plan d'exécution mis en cache dépend du paramètre envoyé lors du premier appel de la procédure. Exécutons une première fois la procédure, et examinons le plan en cache :

```
EXEC Person.GetContactByLastName 'A%'
GO

SELECT cp.size_in_bytes, qp.query_plan
FROM sys.dm_exec_cached_plans cp
CROSS APPLY sys.dm_exec_sql_text(cp.plan_handle) st
CROSS APPLY sys.dm_exec_query_plan(cp.plan_handle) qp
WHERE
  st.dbid = DB_ID('Adventureworks') AND
   st.objectid = OBJECT_ID('Adventureworks.Person.GetContactByLastName')
```

Dans le plan XML, nous trouvons l'opérateur de *scan* d'index *clustered*, donc de table :

```
<RelOp NodeId="0" PhysicalOp="Clustered Index Scan" LogicalOp="Clustered Index
Scan" EstimateRows="911.466" EstimateIO="0.540903" EstimateCPU="0.0221273"
AvgRowSize="126" EstimatedTotalSubtreeCost="0.56303" Parallel="0"
EstimateRebinds="0" EstimateRewinds="0">
```

Si nous appelons la procédure avec le paramètre `'Ackerman'`, et que nous examinons le plan d'exécution dans SSMS, nous constatons que l'opérateur est toujours un *scan* : le plan a été réutilisé, alors qu'il n'est de loin pas, dans ce cas, le plus efficace par rapport au paramètre envoyé.

La procédure est compilée en utilisant une fonctionnalité appelée *parameter sniffing* (littéralement « flairage de paramètre ») : le moteur d'optimisation détecte la

valeur du paramètre passé. Le plan sera donc calculé selon le paramètre envoyé lors du premier appel de la procédure. Si c'est `'Ackerman'`, la procédure fera toujours un *seek*, si c'est `'A%'`, elle scannera toujours la table. Le *parameter sniffing* est une bonne chose lorsque le premier appel est passé avec des paramètres représentatifs des futurs appels de la procédure. Par contre, c'est une mauvaise chose lorsque le premier appel est un cas particulier. Nous allons en faire la démonstration, ce qui nous permettra de démontrer une autre particularité du *parameter sniffing* :

```
-- procédure avec utilisation directe du paramètre
CREATE PROCEDURE dbo.GetContactsParameter
   @LastName nvarchar(50) = NULL
AS BEGIN
   SET NOCOUNT ON

   SELECT FirstName, LastName FROM Person.Contact
   WHERE LastName LIKE @LastName;
END
GO
-- procédure avec variable locale
CREATE PROCEDURE dbo.GetContactsLocalVariable
   @LastName nvarchar(50) = NULL
AS BEGIN
   SET NOCOUNT ON

   DECLARE @MyLastName nvarchar(50)
   SET @MyLastName = @LastName

   SELECT FirstName, LastName FROM Person.Contact
   WHERE LastName LIKE @MyLastName;
END
GO

-- utilisation
EXEC dbo.GetContactsParameter @LastName = 'Abercrombie'
GO
EXEC dbo.GetContactsLocalVariable @LastName = 'Abercrombie'
GO

EXEC dbo.GetContactsParameter @LastName = '%'
GO
EXEC dbo.GetContactsLocalVariable @LastName = '%'
GO
```

Avant d'expliquer la raison pour laquelle nous avons créé deux procédures, observons les résultats de l'exécution, présentés figure 9.1. Nous y voyons les statistiques d'exécution des quatre appels de procédure, ainsi que le plan d'exécution généré du premier appel.

EventClass	TextData	CPU	Reads	Duration
Showplan XML	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserv...			
SQL:BatchCompleted	EXEC dbo.GetContactsParameter @LastName = 'Abercrombie'	0	24	11
Showplan XML	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserv...			
SQL:BatchCompleted	EXEC dbo.GetContactsLocalVariable @LastName = 'Abercromb...	15	1148	32
Showplan XML	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserv...			
SQL:BatchCompleted	EXEC dbo.GetContactsParameter @LastName = '%'	63	59976	602
Showplan XML	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserv...			
SQL:BatchCompleted	EXEC dbo.GetContactsLocalVariable @LastName = '%'	15	1135	588

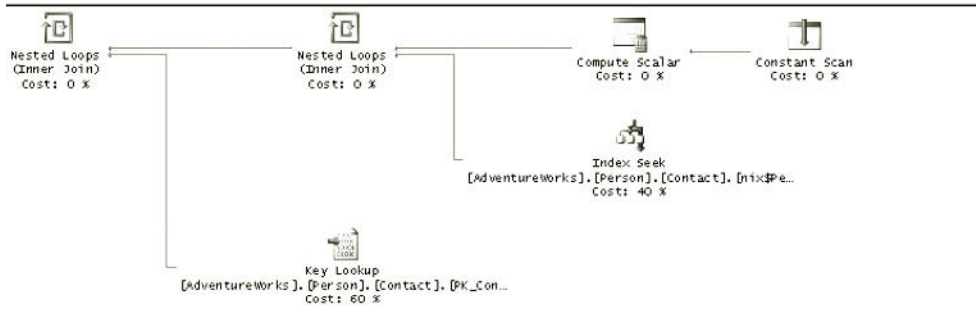

Figure 9.1 — Recompilations

Nous avons passé d'abord le nom `'Abercrombie'` : la compilation de la procédure a donc produit un plan basé sur un *seek* d'index. Ce plan sera réutilisé tout au long des exécutions futures de la procédure, ce que nous voyons plus loin : un appel avec le paramètre `'%'` génère près de 60 000 reads et le moteur de stockage a été forcé de parcourir près de 20 000 fois l'index, pour trouver l'emplacement de chaque ligne de la table.

Et qu'en est-il de la deuxième procédure ? Nous sommes passés par une variable locale, à laquelle nous avons affecté la valeur du paramètre, puis avons utilisé cette variable locale comme opérande de l'expression de filtre. Ce que nous voulions montrer, c'est que cette syntaxe ne permet pas le *parameter sniffing*. Dans ce cas, le plan d'exécution généré prend en compte une distribution moyenne des valeurs dans la colonne, et non pas la valeur actuellement passée à la requête, ceci simplement parce que cette valeur n'est pas connue lors de l'optimisation. Dans notre cas, la distribution moyenne incite SQL Server à choisir un *scan*. Lors du premier appel, cette stratégie est plus coûteuse que le *seek*. Par contre, lors du second appel, le résultat est bien plus consistant en terme de temps CPU et de reads. Dans le deuxième cas, le *scan* est la bonne stratégie.

Il est donc important que vous preniez ce comportement en compte lorsque vous créez des procédures stockées. Souvent, les paramètres sont assez consistants, et vous n'avez pas à vous soucier du premier plan d'exécution généré, mais dans certains cas, vous devez gérer ces différences, soit en écrivant vos procédures stockées différemment (en les modularisant, par exemple), soit en les forçant à se recompiler.

Forçage de la recompilation

Que faire pour résoudre le problème des appels de procédures stockées avec des paramètres s'appliquant à des colonnes dont la distribution est très variable ? Une solution est de provoquer manuellement la recompilation. Cela peut être fait de plusieurs façons. L'option `WITH RECOMPILE` peut être indiquée dans le corps de la procédure stockée ou indépendamment à chaque appel :

```
ALTER PROCEDURE Person.GetContactByLastName
    @LastNameStart nvarchar(50)
WITH RECOMPILE
AS BEGIN …
-- ou :
EXEC Person.GetContactByLastName 'Ackerman' WITH RECOMPILE
```

La première méthode, qui force la recompilation à chaque appel, est utile pour une procédure où les valeurs de paramètres sont très différentes chaque fois, la seconde pour forcer la recompilation dans des cas particuliers. Si votre procédure n'est appelée qu'avec des paramètres atypiques, préférez la première solution. La seconde est d'une utilité particulière : un `EXEC … WITH RECOMPILE` génère un plan d'exécution qui ne sera pas mis en cache, et qui **ne remplacera donc pas le plan caché existant**. Cette méthode sera donc précieuse si vous avez besoin d'appeler ponctuellement une procédure avec un paramètre atypique.

La recompilation est une opération coûteuse, c'est pourquoi il vaut mieux limiter l'usage du `WITH RECOMPILE` à des procédures de petite taille, qui en ont vraiment besoin, c'est-à-dire où le coût de la recompilation est moindre que celui de l'exécution avec un plan inefficace.

Pour les procédures qui comportent de multiples instructions, vous pouvez sélectionner une recompilation instruction par instruction, à l'aide de l'indicateur de requête `OPTION (RECOMPILE)`. Par exemple :

```
CREATE PROCEDURE dbo.GetContactsParameter
   @LastName nvarchar(50) = NULL
AS BEGIN
   SET NOCOUNT ON

   SELECT FirstName, LastName
   FROM Person.Contact
   WHERE LastName LIKE @LastName
   OPTION (RECOMPILE);
END
```

Une meilleure solution est de forcer, toujours par indicateur de requête, sur quelle valeur l'optimiseur doit se baser :

```
SELECT FirstName, LastName
FROM Person.Contact
WHERE LastName LIKE @LastName
OPTION OPTIMIZE FOR (@LastName = '%'));
```

L'inconvénient de cette approche est évidemment de coder en dur une valeur de la colonne, qui peut évoluer à travers le temps, ou disparaître.

> **En SQL Server 2008**, l'indicateur OPTIMIZE FOR est complété de la façon suivante :
> `OPTION (OPTIMIZE FOR (@variable = UNKNOWN))`
> ou `OPTION (OPTIMIZE FOR UNKNOWN)`
> qui permettent, comme à travers l'utilisation de variables locales, de faire une optimisation générique à partir d'une distribution moyenne des valeurs des colonnes, et donc d'assurer la stabilité du plan, soit pour une variable, soit pour toutes les variables utilisées dans la requête.

Ces options sont très utiles dans les procédures stockées complexes, où le problème du *parameter sniffing* se fait le plus aigu, notamment lors de branchements conditionnels. Prenons le cas de cette procédure « modulaire » :

```
CREATE PROCEDURE Person.GetContactsByWhatever
    @NamePartType tinyint,
    @NamePart nvarchar(50)
AS BEGIN
    SET NOCOUNT ON

    IF (@NamePartType = 1)
        SELECT FirstName, LastName, EmailAddress
        FROM Person.Contact
        WHERE FirstName LIKE @NamePart
    ELSE IF (@NamePartType = 2)
        SELECT FirstName, LastName, EmailAddress
        FROM Person.Contact
        WHERE LastName LIKE @NamePart
    ELSE IF (@NamePartType = 3)
        SELECT FirstName, LastName, EmailAddress
        FROM Person.Contact
        WHERE EmailAddress LIKE @NamePart
END
```

Le but, louable au premier abord, est de créer une procédure générique. Malheureusement, la première valeur envoyée dans `@NamePart` conditionne le plan d'exécution des trois instructions, pour trois colonnes différentes ! Dans ce cas, la recompilation sélective est très intéressante, car elle ne générera une compilation que de l'instruction utilisée (dans laquelle on se branche), au lieu d'une recompilation générale comme avec un `WITH RECOMPILE`. Notons tout de même que la meilleure solution reste d'éviter la création de procédures génériques.

sp_recompile

La procédure stockée système sp_recompile permet de marquer une procédure ou un déclencheur pour être recompilée. Dans les faits, elle supprime la procédure du cache. Vous pouvez aussi indiquer une table ou une vue. Dans ce cas, toutes les procédures référençant cet objet seront recompilées à leur prochaine exécution. Cette procédure est aujourd'hui peu utile, SQL Server faisant ce travail lui-même,

notamment invalidant automatiquement les procédures d'un objet qui vient d'être modifié.

9.2.2 Recompilations automatiques

Les recompilations ne sont pas toutes déclenchées volontairement. Pour plusieurs raisons, il ne serait pas raisonnable de conserver un plan d'exécution intact en cache tout le long de la vie d'une instance (qui peut rester active des années sans être redémarrée). Le serveur SQL vit : non seulement les structures sont susceptibles de changer, ce qui invalide le plan d'exécution, mais le contenu des tables évolue, entraînant des recalculs de statistiques, le contexte d'exécution des utilisateurs peut lui aussi changer, etc. Tous ces événements entraînent un changement potentiel de plan d'exécution. SQL Server est attentif à ces modifications, et peut, lorsque le besoin s'en fait sentir, déclencher des recompilations automatiques, lors de l'exécution du code. Depuis SQL Server 2005, ces recompilations s'effectuent par instruction (*statement-level recompilation*) et n'affectent donc pas la procédure ou le *batch* tout entier.

Elles se produisent pour deux types de raisons :

- exactitude des données – la modification des structures sous-jacentes, comme l'ajout ou la suppression de colonnes de table, de contraintes, la suppression d'un index utilisé dans le plan, etc. ainsi que la modification d'options de session (notamment à l'intérieur de la procédure stockée). les options qui peuvent modifier le résultat des requêtes ou la valeur des constantes, comme par exemple `SET ANSI_NULLS`, `SET CONCAT_NULL_YIELDS_NULL`, `SET DATEFORMAT`, etc., peuvent provoquer des recompilations systématiques. Elles peuvent changer le contexte d'exécution, donc forcer le plan d'exécution à s'adapter à ce nouvel environnement ;
- optimisation du plan – principalement lors d'un recalcul de statistiques. Le plan compilé comporte une valeur de seuil de recompilation, et teste à chaque exécution si le nombre de modifications dans la table dépasse ce seuil. Si c'est le cas, la requête est recompilée.

Les recompilations peuvent s'avérer coûteuses. Elles sont un mal (ou un bien) nécessaire, mais ralentissent parfois inutilement l'exécution de procédures stockées. Cela est généralement dû à de mauvaises pratiques de programmation, qui déclenchent des recompilations répétitives à chaque exécution de la procédure stockée, voire plusieurs fois par exécution. Nous avons parlé des options de sessions. D'autres éléments sont à considérer :

L'interpolation de code DML et de code DDL est susceptible de provoquer des recompilations : la modification de structure d'objets doit être répercutée dans le plan d'exécution des requêtes qui les référencent ;

La mauvaise utilisation de tables temporaires peut entraîner des recompilations. La situation depuis SQL Server 2005 est meilleure qu'en SQL Server 2000 : comme

les recompilations s'effectuent maintenant instruction par instruction, beaucoup de recompilations dues aux tables temporaires ne se produisent plus, car le plan de toute la procédure n'est pas invalidé et le cache de chaque instruction peut être réutilisé. Lorsque la table temporaire est créée, une recompilation de type « compilation déférée » (voir plus loin les événements de trace) est déclenchée. Ensuite, l'insertion, la mise à jour ou la suppression de lignes dans la table temporaire peut rapidement provoquer d'autres recompilations, pour prendre en compte les nouvelles cardinalités (la distribution des valeurs dans une colonne). Une recompilation est générée sur une table temporaire à partir de six insertions sur une table vide. Par exemple :

```
ALTER PROC dbo.testtemptable
    @nbInserts smallint = 1
AS BEGIN
    DECLARE @i smallint
    SET @i = 1

    CREATE TABLE #t (
        id int identity(1,1) PRIMARY KEY NONCLUSTERED,
        col char(8000) NOT NULL DEFAULT ('e'))

    WHILE @i <= @nbInserts BEGIN
        INSERT INTO #t DEFAULT VALUES
        SELECT * FROM #t WHERE col = 'e' OR id > 20
        SET @i = @i + 1
    END
END
GO

EXEC dbo.testtemptable
GO
```

Si vous tracez cette exécution avec le profiler, vous verrez un résultat ressemblant à la figure 9.2.

EventClass	TextData	EventSubClass
SQL:BatchStarting	EXEC dbo.testtemptable	
SP:StmtStarting	SET @i = 1	
SP:StmtStarting	CREATE TABLE #t (id int identity(1,1) PRIMAR...	
SP:StmtStarting	WHILE @i <= @nbInserts	
SP:StmtStarting	INSERT INTO #t DEFAULT VALUES	
SP:Recompile	INSERT INTO #t DEFAULT VALUES	3 - Deferred compile
SQL:StmtRecompile	INSERT INTO #t DEFAULT VALUES	3 - Deferred compile
SP:StmtStarting	INSERT INTO #t DEFAULT VALUES	
SP:StmtStarting	SELECT * FROM #t WHERE col = 'e' OR id > 20	
SP:Recompile	SELECT * FROM #t WHERE col = 'e' OR id > 20	3 - Deferred compile
SQL:StmtRecompile	SELECT * FROM #t WHERE col = 'e' OR id > 20	3 - Deferred compile
SP:StmtStarting	SELECT StatMan([SC0]) FROM (SELECT TOP 100 PERCE...	
SP:StmtStarting		
SP:StmtStarting	SELECT StatMan([SC0]) FROM (SELECT TOP 100 PERCE...	
SP:StmtStarting	SELECT * FROM #t WHERE col = 'e' OR id > 20	
SP:StmtStarting	SET @i = @i + 1	
SP:StmtStarting	WHILE @i <= @nbInserts	
SQL:BatchCompleted	EXEC dbo.testtemptable	

Figure 9.2 — Recompilations sur tables temporaires

C’est-à-dire deux recompilations, pour compilation déférée. Si nous exécutons ensuite `EXEC dbo.testtemptable 7`, voici le résultat en figure 9.3 :

EventClass	TextData	EventSubClass
SQL:BatchStarting	EXEC dbo.testtemptable 7	
SP:StmtStarting	SET @i = 1	
SP:StmtStarting	CREATE TABLE #t (id int identit...	
SP:StmtStarting	WHILE @i <= @nbInserts	
SP:StmtStarting	INSERT INTO #t DEFAULT VALUES	
SP:StmtStarting	SELECT * FROM #t WHERE col = 'e' OR...	
SP:StmtStarting	SET @i = @i + 1	
SP:StmtStarting	WHILE @i <= @nbInserts	
SP:StmtStarting	INSERT INTO #t DEFAULT VALUES	
SP:StmtStarting	SELECT * FROM #t WHERE col = 'e' OR...	
SP:StmtStarting	SET @i = @i + 1	
SP:StmtStarting	WHILE @i <= @nbInserts	
SP:StmtStarting	INSERT INTO #t DEFAULT VALUES	
SP:StmtStarting	SELECT * FROM #t WHERE col = 'e' OR...	
SP:StmtStarting	SET @i = @i + 1	
SP:StmtStarting	WHILE @i <= @nbInserts	
SP:StmtStarting	INSERT INTO #t DEFAULT VALUES	
SP:StmtStarting	SELECT * FROM #t WHERE col = 'e' OR...	
SP:StmtStarting	SET @i = @i + 1	
SP:StmtStarting	WHILE @i <= @nbInserts	
SP:StmtStarting	INSERT INTO #t DEFAULT VALUES	
SP:StmtStarting	SELECT * FROM #t WHERE col = 'e' OR...	
SP:StmtStarting	SET @i = @i + 1	
SP:StmtStarting	WHILE @i <= @nbInserts	
SP:StmtStarting	INSERT INTO #t DEFAULT VALUES	
SP:StmtStarting	SELECT * FROM #t WHERE col = 'e' OR...	
SP:StmtStarting	SET @i = @i + 1	
SP:StmtStarting	WHILE @i <= @nbInserts	
SP:StmtStarting	INSERT INTO #t DEFAULT VALUES	
SP:StmtStarting	SELECT * FROM #t WHERE col = 'e' OR...	
SP:Recompile	SELECT * FROM #t WHERE col = 'e' OR...	2 - Statistics changed
SQL:StmtRecompile	SELECT * FROM #t WHERE col = 'e' OR...	2 - Statistics changed

Figure 9.3 — Recompilations sur tables temporaires

Nous avons dû faire ici une requête `SELECT` sur la table temporaire un tout petit plus compliquée que nécessaire, parce que le comportement du cache sur des procédures stockées est bien meilleur qu’auparavant. La recompilation se déclenche au bout de la sixième insertion dans la table, ce qui est le comportement normal de SQL Server.

À la deuxième exécution de la procédure, il n’y aura pas recompilation, l’instruction étant déjà cachée avec un plan valable. SQL Server est donc capable de gérer assez bien les recompilations dues aux tables temporaires. Cela ne veut pas dire que vous devez les utiliser sans modération. Évitez donc le recours exagéré aux tables temporaires. Elles sont souvent inutiles et peuvent être remplacées par des sous-requêtes, des expressions de table (CTE, *common table expressions*), ou, au pire, des variables de table pour les petits volumes. Si vous êtes forcé de composer avec des tables temporaires, et que vous détectez par le profiler des recompilations fréquentes dues à des changements de cardinalité dans les tables, il vous reste deux options de

requêtes avec lesquelles vous pouvez modifier la fréquence des recompilations : `KEEP PLAN`, et `KEEPFIXED PLAN`.

KEEP PLAN et KEEPFIXED PLAN

l'indicateur de requêtes `KEEP PLAN` modifie les seuils de compilation des tables temporaires (premiers seuils à 6 lignes modifiées, puis 500 lignes modifiées) qui deviennent alors identiques à ceux des tables permanentes. Si les modifications apportées à des tables temporaires entraînent de nombreuses recompilations, essayez de placer cette option dans votre procédure, et vérifiez si cela diminue la fréquence de recompilation. Exemple de syntaxe pour notre procédure :

```
SELECT * FROM #t WHERE col = 'e' OR id > 20
OPTION (KEEP PLAN)
```

L'indicateur `KEEPFIXED PLAN` est encore plus contraignant : il empêche simplement toute recompilation de l'instruction pour raison d'optimisation (nouvelles statistiques, changement de cardinalité…). Appliquez-le aux requêtes que vous avez reconnues comme posant réellement un problème de recompilation.

Tracer les recompilations

Vous pouvez détecter les recompilations à l'aide d'une trace, et des événements `SP:Recompile` et `SQL:StmtRecompile`. Ces événements vous indiquent, dans la colonne `EventSubClass` de la trace, pour quelle raison la recompilation a été déclenchée. Les valeurs possibles sont celles du tableau 9.1.

Tableau 9.1– Valeurs

EventSubClass	Signification
1	La structure de l'objet a changé
2	Les statistiques ont changé
3	Compilation déférée : par exemple, lors de l'utilisation d'une table temporaire, les instructions de la procédure utilisant cette table ne peuvent être compilées au début de l'exécution de la procédure. Elles ne le pourront que lorsque la table sera effectivement créée. Cela déclenchera à ce moment une recompilation de ce type.
4	Une option de session a changé
5	Une table temporaire a changé
6	Un jeu de résultant distant (serveur lié) a changé
7	Une permission FOR BROWSE a changé.
8	L'environnement de Query Notification a changé
9	Une vue partitionnée a changé

EventSubClass	Signification
10	Les options de curseur ont changé
11	La recompilation a été demandée par l'option de requête RECOMPILE

9.3 CACHE DES REQUÊTES AD HOC

Le cache contient en réalité bien plus que les plans d'exécution des procédures stockées. Il existe trois parties principales du cache (*cache stores*) qui nous intéressent ici et qui stockent des résultats de compilation :

- **Object Plans** (`CACHESTORE_OBJCP`) : plans de procédures stockées, déclencheurs et fonctions.
- **SQL Plans** (`CACHESTORE_SQLCP`) : plans de *batches*.
- **Bound Trees** (`CACHESTORE_PHDR`) : arbres d'analyse d'une requête.

Elles comportent chacune une table de hachage qui permet de gérer les entrées du cache (une *hash table* composée de *hash buckets*). Vous pouvez obtenir des informations sur les caches par la vue de gestion dynamique `sys.dm_os_memory_cache_counters`, et examiner les tables de hachage par la vue `sys.dm_os_memory_cache_hash_tables` et enfin les hash buckets par `sys.dm_os_memory_cache_entries`. Il y a également plusieurs types d'objets dans le cache, les deux qui nous intéressent ici sont les plans compilés (*Compiled Plans*, CP) et les plans d'exécution (*Execution Plans*, MXC). La requête suivante vous donne la taille de ces caches :

```
SELECT
    Name,
    Type,
    single_pages_kb,
    single_pages_kb / 1024 AS Single_Pages_MB,
    entries_count
FROM sys.dm_os_memory_cache_counters
WHERE type in ('CACHESTORE_SQLCP', 'CACHESTORE_OBJCP',
               'CACHESTORE_PHDR')
ORDER BY single_pages_kb DESC
```

Les plans compilés représentent la compilation d'une procédure ou d'un *batch* de requêtes (c'est-à-dire d'ordres SQL envoyés en un seul lot depuis le client). S'il s'agit d'une procédure (procédure stockée, fonction, déclencheur), il est stocké dans `CACHESTORE_OBJCP`, s'il s'agit d'un batch, il ira dans `CACHESTORE_SQLCP`.

Les plans d'exécution sont en quelque sorte des instances des plans compilés, ils sont générés rapidement, à l'exécution, à partir d'un plan compilé. Il y a un plan d'exécution par utilisateur lançant la procédure ou le *batch*. Vous pouvez inspecter

ces plans à l'aide de la fonction `sys.dm_exec_cached_plan_dependent_objects`, à laquelle vous passez en paramètre un `plan_handle` venant de `sys.dm_exec_cached_plans`. Si la procédure est en cours d'exécution plusieurs fois simultanément, vous trouverez plusieurs références au même `plan_handle`.

Les plans compilés contiennent un tableau d'instructions SQL. Chaque ordre SQL dans la procédure ou le *batch* sont compilés séparément dans des Cstmt, ou *Compiled Statements*. Les plans d'exécution génèrent des Xstmts, les versions *runtime* des Cstmt[1]. Voici une requête pour voir la taille et l'utilisation des différents objets :

```
SELECT
    COUNT(*) as cnt,
    SUM(size_in_bytes) / 1024 as total_kb,
    MAX(usecounts) as max_usecounts,
    AVG(usecounts) as avg_usecounts,
    CASE GROUPING(cacheobjtype)
        WHEN 1 THEN 'TOTAL'
        ELSE cacheobjtype
    END AS cacheobjtype,
    CASE GROUPING(objtype)
        WHEN 1 THEN 'TOTAL'
        ELSE objtype
    END AS objtype
FROM sys.dm_exec_cached_plans
GROUP BY cacheobjtype, objtype
WITH ROLLUP
```

Le code de la procédure ou du *batch* est stocké dans un cache à part, le *SQL Manager Cache* (SQLMGR). Il s'agit d'un stockage différent du plan réellement utilisé à l'exécution. Vous pouvez voir les informations générales de ce cache à l'aide de cette requête :

```
SELECT *
FROM sys.dm_os_memory_objects
WHERE type = 'MEMOBJ_SQLMGR'
```

Pourquoi maintenir le code de la requête séparément du plan compilé ? Simplement parce que ce plan peut changer, selon l'état de la session, c'est-à-dire le contexte d'exécution. Il est important de s'y arrêter, car cela influe sur la possibilité qu'a SQL Server de réutiliser un plan en cache.

9.3.1 Réutilisation des plans

Imaginons que nous exécutions le même code, dans deux sessions qui comportent des options de session différentes (`SET…`, comme `DATEFORMAT`, `ANSI NULLS…`), ou dans la même session, avec un changement d'option intermédiaire. Par exemple :

1. Si vous voulez entrer plus en détail dans l'observation des éléments du cache, une série très complète d'articles a été publiée dans le blog de l'équipe Microsoft « SQL Programmability & API Development » : http://blogs.msdn.com/sqlprogrammability/

```
SET CONCAT_NULL_YIELDS_NULL ON
GO
SELECT TOP 10 FirstName + ' ' + MiddleName + ' ' + LastName
FROM Person.Contact
GO

SET CONCAT_NULL_YIELDS_NULL OFF
GO
SELECT TOP 10 FirstName + ' ' + MiddleName + ' ' + LastName
FROM Person.Contact
GO
```

Ou, imaginons que nous avons deux utilisateurs, Paul et Isabelle, qui sont déclarés dans la base avec deux schémas par défaut différents (par exemple `Person` et `HumanResources`). S'ils exécutent tous deux un code identique, qui ne référence pas le schéma de l'objet, un plan d'exécution différent doit être généré, même si l'objet qui sera touché est le même, simplement parce que SQL Server ne sait pas à l'avance quel va être le bon objet. En vertu du mécanisme de résolution de nom, un objet appartenant au schéma par défaut sera d'abord recherché, et s'il n'est pas trouvé, SQL Server cherchera un objet appartenant au schéma `dbo`. Exemple :

```
CREATE TABLE dbo.test (TestId int)
GO
ALTER USER isabelle WITH DEFAULT_SCHEMA = Person
ALTER USER paul WITH DEFAULT_SCHEMA = HumanResources
GO
CREATE TABLE dbo.test (TestId int)
GO
GRANT SELECT ON dbo.test TO paul
GRANT SELECT ON dbo.test TO isabelle
GO

EXECUTE AS USER = 'paul'
SELECT CURRENT_USER
GO
SELECT * FROM test
GO
REVERT
GO

EXECUTE AS USER = 'isabelle'
SELECT CURRENT_USER
GO
SELECT * FROM test
GO
REVERT
GO
```

Combien avons-nous de plans ? Vous trouverez sur la figure 9.4 le résultat de la requête

```
SELECT st.text, qs.sql_handle, qs.plan_handle
FROM sys.dm_exec_query_stats qs
```

```
CROSS APPLY sys.dm_exec_sql_text(qs.sql_handle) st
ORDER BY qs.sql_handle
```

exécutée après les deux exemples précédents.

	text	sql_handle	plan_handle
1	SELECT * FROM test	0x020000002F5CC820E0CC946DD76094543CF7AA299904C81A	0x060005002F5CC820B8814611000000000000000000000000
2	SELECT * FROM test	0x020000002F5CC820E0CC946DD76094543CF7AA299904C81A	0x060005002F5CC820B8A1FF1000000000000000000000000
3	SELECT TOP 10 FirstName + ' '...	0x02000000578082045F4F63242DD1BF4C47BCE5B3FADB8538	0x060005005780820 4B861B1170000000000000000000000
4	SELECT TOP 10 FirstName + ' '...	0x02000000578082045F4F63242DD1BF4C47BCE5B3FADB8538	0x060005005780820 4B8010211000000000000000000000

Figure 9.4 — Plusieurs plan_handle pour le même sql_handle

Vous voyez que pour le même `sql_handle`, vous avez chaque fois deux `plan_handle`. Vous pouvez aussi le constater en traçant avec le profiler l'événement `sp:CacheInsert`. Afin de vérifier pour quelle raison (quel attribut du plan) des plans différents ont été créés, vous pouvez vous baser sur la vue `sys.dm_exec_plan_attributes`, comme ceci par exemple (en passant un `sql_hande` trouvé) :

```
SELECT st.text, qs.sql_handle, qs.plan_handle, pa.attribute, pa.value
FROM sys.dm_exec_query_stats qs
CROSS APPLY sys.dm_exec_sql_text(qs.plan_handle) st
OUTER APPLY sys.dm_exec_plan_attributes(qs.plan_handle) pa
WHERE qs.sql_handle = 0x020000002F5CC820E0CC946DD76094543CF7AA299904C81A
and pa.is_cache_key = 1
ORDER BY pa.attribute
```

Ce qu'il faut comprendre ici, c'est que SQL Server cherche à réutiliser autant que possible les plans d'exécution en cache, car le recalcul de plans peut être très pénalisant. Nous venons de voir que certaines conditions invalident pourtant le plan en cache et obligent SQL Server à compiler à nouveau le batch ou la procédure. Il faut éviter autant que possible de provoquer de telles situations. Les deux cas que nous avons testés précédemment sont les plus courants :

- un changement d'options de session en cours de travail modifie le contexte d'exécution. Des options comme `ANSI_NULLS`, `ANSI_DEFAULTS`, `CONCAT_NULL_YIELDS_NULL`, `ARITHABORT`, `DATEFIRST`, `DATEFORMAT`, `LANGUAGE`, `QUOTED_IDENTIFIER`, etc. peuvent empêcher la réutilisation d'un plan, parce qu'elles influent sur les comparaisons, et qu'elles modifient potentiellement la valeur des littéraux exprimés dans la requête. SQL Server évalue très tôt dans la phase de compilation la valeur de ces littéraux (une fonctionnalité nommée *constant folding*). Si une option est modifiée, qui peut changer cette valeur déjà évaluée, le code doit être compilé à nouveau ;
- lorsqu'un objet n'est pas complètement identifié par son schéma, SQL Server ne peut pas garantir que l'appel par des utilisateurs dont le schéma par défaut est différent, va référencer le même objet, il doit donc recompiler.

Deux conseils s'imposent donc :
1. Maintenez des états de session consistants en affectant les options à la connexion et éviter de les changer en cours de route, surtout à l'intérieur des procédures stockées (le `SET NOCOUNT` n'entre pas dans cette catégorie, il ne provoque aucune recompilation, puisqu'il ne change pas le comportement des requêtes).
2. Préfixez toujours vos objets par leur nom de schéma.

En ce qui concerne les options de session, attention aux différentes variétés de bibliothèques client. Les anciennes méthodes de connexion, telles que ODBC, ne placent pas par défaut les mêmes valeurs d'option que les méthodes plus modernes, comme ADO.NET. Pour savoir quelles sont les valeurs des options de session, vous pouvez utiliser la commande `DBCC USEROPTIONS`, ou les événements `Session / ExistingConnection` et `Security Audit / Audit Login`.

La différence entre le plan compilé et le plan d'exécution peut être observée *via* la vue `sys.dm_exec_query_stats`, qui référence le plan compilé dans la colonne `sql_handle` et le plan d'exécution dans la colonne `plan_handle`. Les handles sont des hachages MD5 générés à partir du plan entier, ils sont donc garantis uniques par plan. Ils peuvent être passés à la fonction `sys.dm_exec_sql_text` pour voir le contenu du plan.

Lorsque nous avons extrait dans les requêtes précédentes des plans d'exécution du cache, à l'aide de la vue `sys.dm_exec_cached_plans` et de la colonne `plan_handle`, nous avions accès à la totalité du plan compilé, et non aux Cstmt individuels. Cette vision était celle d'un cache particulier, vous pouvez en faire l'expérience avec cette requête d'exemple :

```
DBCC FREEPROCCACHE
GO

SET CONCAT_NULL_YIELDS_NULL ON
GO
SELECT TOP 10 FirstName + ' ' + MiddleName + ' ' + LastName
FROM Person.Contact
GO

SET CONCAT_NULL_YIELDS_NULL OFF
GO
SELECT TOP 10 FirstName + ' ' + MiddleName + ' ' + LastName
FROM Person.Contact
GO

-- plusieurs plan_handle pour le même sql_handle
SELECT
    cp.usecounts,
    cp.size_in_bytes,
    st.text
FROM sys.dm_exec_cached_plans cp
OUTER APPLY sys.dm_exec_sql_text (cp.plan_handle) st
JOIN sys.dm_exec_query_stats qs ON qs.plan_handle = cp.plan_handle
GO
```

Cache des requêtes ad hoc

Nous l'avons vu, les procédures ne sont pas seules à être cachées, tout plan d'exécution d'une requête isolée est potentiellement réutilisable. Une première réaction serait de penser que cette fonctionnalité rend la procédure stockée moins intéressante, puisque toutes les requêtes peuvent profiter d'un cache de leur plan d'exécution. Ce n'est pas vraiment le cas, la procédure reste nettement plus performante, non seulement par les avantages que nous avons déjà abordés (diminution du trafic réseau, centralisation du code, sécurité facilitée), mais aussi parce que le cache de requêtes est soumis à quelques contraintes. Nous allons le voir.

Lorsqu'une requête ad hoc (c'est-à-dire un ordre SQL composé par le client, par opposition à du code stocké sur le serveur comme une procédure) est envoyée au serveur, SQL Server essaie de trouver une correspondance de texte de requête dans le cache SQLMGR. Cette correspondance est recherchée en comparant le hachage généré par la requête entrante avec les hachages présents dans le cache, puisque ces chaînes de hachages représentent un résumé de toute la requête cachée. Cela permet d'effectuer une recherche très rapide, mais cela signifie aussi que la moindre différence de syntaxe invalide la recherche, puisqu'elle produit un résultat de hachage différent. Une espace de plus, une différence de casse dans la requête (même sur un serveur dont la collation par défaut est insensible à la casse, cela n'a pas de rapport avec la gestion du cache) suffit à générer une nouvelle entrée dans le cache. Vérifions-le simplement :

```
DBCC FREEPROCCACHE
GO
SELECT * FROM dbo.test
GO
SELECT * FROM  dbo.test
GO
SELECT * FROM dbo.TEST
GO
select * from dbo.test
GO
SELECT
    cp.usecounts,
    cp.size_in_bytes,
    st.text
FROM sys.dm_exec_cached_plans cp
OUTER APPLY sys.dm_exec_sql_text (cp.plan_handle) st
JOIN sys.dm_exec_query_stats qs ON qs.plan_handle = cp.plan_handle
GO
```

Le code suivant provoque l'insertion de quatre entrées de cache, comme nous pouvons le constater dans la figure 9.5.

	text	sql_handle	plan_handle
1	SELECT * FROM Person.Contact	0x0200000055C87120D7557120B4ECA68E2D55FE12C9A8A19A	0x0600050055C87120B861B31500000000000000000000000000
2	SELECT * FROM Person.CONTACT	0x020000007768A2D235E5635EE42911E5265157D93B89F311D	0x0600050076 8A2D23B8A1441200000000000000000000000000
3	SELECT * FROM Person.Contact	0x0200000CCDF664292F90BB1E83B8A7E9D463892AB9C32427	0x0600050CCDF66429B8216412000000000000000000000000
4	select * from Person.Contact	0x0200000CDB8ACD262580086F64731CAA3D608852B1937804	0x0600050DB8ACD2688E13F10000000000000000000000000

Figure 9.5 — Plusieurs plans sur des différences syntaxiques

Attention – La correspondance avec un plan ad hoc caché n'est possible que si les objets sont préfixés par leur schéma.

Comme pour le cache de procédure, ces plans d'exécution ne sont libérés qu'en cas de besoin mémoire.

Mais ce fonctionnement est valable même en considérant des valeurs constantes différentes, passées dans les clauses `WHERE` pour la recherche ? En d'autres termes, des requêtes comme :

```
SELECT * FROM Person.Contact WHERE ContactId = 20
GO
SELECT * FROM Person.Contact WHERE ContactId = 40
GO
```

sont-elles cachées séparément ? Cela dépend. Dans cet exemple, un examen de sys.dm_exec_query_stats. Montre ceci :

```
(@1 tinyint)SELECT * FROM [Person].[Contact] WHERE [ContactId]=@1
```

Ce qui signifie que SQL Server a transformé la requête avant de la mettre dans le cache, pour remplacer les valeurs littérales par un paramètre. Dans certains cas, SQL Server peut donc « paramétrer » la requête, et transformer la valeur passée comme critère de recherche en paramètre, en interne. Cela permet de réutiliser le plan si d'autres paramètres sont passés. Ce mécanisme s'appelait paramétrage automatique (*auto-parameterization*) en SQL Server 2000, et est maintenant nommé paramétrage simple (*simple parameterization*).

Le comportement par défaut de SQL Server fait qu'un nombre limité de requêtes peuvent être ainsi paramétrées. Par exemple, les deux requêtes suivantes sont cachées séparément sur SQL Server 2005 sp2 :

```
SELECT * FROM Person.Contact WHERE LastName = 'Allen'
GO
SELECT * FROM Person.Contact WHERE LastName = 'Ackerman'
GO
```

Nous avons vu que la valeur de `LastName` peut ici fortement influencer le plan d'exécution, déclenchant soit un *seek*, soit un *scan* de l'index. L'exemple précédent, où le paramétrage avait eu lieu, était plus facile : `ContactId` est la clé primaire de la table, donc sa valeur est garantie unique. Une recherche d'égalité est donc garantie de retourner une seule ligne, et qui plus est dans une recherche par un index *clustered*

(option par défaut à la création d'une clé primaire). Le plan d'exécution ne changera pas, quelle que soit la valeur passée en paramètre. SQL Server ne cache donc qu'un plan qu'il considère « *safe* », c'est-à-dire qui va pouvoir répondre à pratiquement tous les cas de valeur du paramètre. De plus, il n'essaie même pas de paramétrer des requêtes qui comportent des constructions particulières, pour certaines très courantes, comme des jointures ou des sous-requêtes[1]. Vraiment pas de quoi concurrencer une procédure stockée...

Notons que la réutilisation du cache de requêtes paramétrées est plus souple avec la syntaxe, car elle ne base plus sa recherche sur le même mécanisme de comparaison de hachage. Une requête paramétrée sera alors « rationalisée « pour correspondre à la même requête comportant des espaces ou des retours chariot différents. Elle reste toutefois sensible à la casse.

Pour généraliser le paramétrage, vous disposez éventuellement d'une option de base de données qui force SQL Server à paramétrer toutes les requêtes :

```
ALTER DATABASE AdventureWorks SET PARAMETERIZATION FORCED

-- pour revenir à l'état par défaut :
ALTER DATABASE AdventureWorks SET PARAMETERIZATION SIMPLE
```

En passant en paramétrage forcé (*Forced Parameterization*), toutes les valeurs littérales rencontrées dans une requête sont converties en paramètres. La requête précédente sera alors stockée dans le cache avec la syntaxe suivante :

```
(@0 varchar(8000))select * from Person.Contact where LastName = @0
```

Apparemment utile, cette option n'est pas à activer à la légère, car le paramétrage implique un effort plus important de SQL Server, aussi bien à la génération du plan d'exécution qu'à la recherche de correspondances dans le cache. Elle peut aussi provoquer la conservation et la réutilisation de plans non optimaux, puisque les limites du paramétrage simple servent justement à générer des plans différents lorsque c'est nécessaire. Elle n'est donc pas la panacée. Rien ne remplace la simplicité d'une procédure stockée.

Vous noterez au passage que le paramètre généré est de type `varchar(8000)`, alors que la colonne recherchée est un `nvarchar(50)`. Le type de la colonne n'est pas vérifié par SQL Server, qui applique le type de la constante, et la taille maximum du type de données. Cette opération est appelée *bucketization*. Elle est forcée par le serveur dans ce cas, mais vous avez la possibilité de paramétrer votre requête dans votre code client (par exemple en utilisant la collection `Parameters` de l'objet `SQLCommand` en ADO.NET avec des types de données explicites), et ainsi d'optimiser le type de données du paramètre, pour une commande que vous allez réutiliser plusieurs fois.

1. Vous en trouvez une liste non exhaustive dans cette entrée de blog : http://blogs.msdn.com/sqlprogrammability/archive/2007/01/11/4-0-query-parameterization.aspx

SQL Server 2008 intègre une option de serveur nommée « `optimize for ad hoc workloads` », qui vous permet de diminuer la consommation de mémoire en cache de plan, lorsque votre serveur est principalement utilisé avec des requêtes dynamiques, non paramétrées. Pour l'activer, exécutez ce code :

```
EXEC sp_configure 'show advanced options',1
RECONFIGURE
EXEC sp_configure 'optimize for ad hoc workloads',1
RECONFIGURE
```

9.3.2 Paramétrage du SQL dynamique

Une facilité est offerte pour l'exécution paramétrée du **SQL dynamique**. Au lieu d'utiliser la commande `EXECUTE()`, qui se contente d'évaluer et d'envoyer l'instruction SQL telle quelle, la procédure stockées système `sp_executesql` permet de déclarer des paramètres dans la chaîne et de les envoyer à chaque appel, paramétrant ainsi explicitement le plan d'exécution. Voici un exemple qui présente les deux solutions :

```
DBCC FREEPROCCACHE
GO
DECLARE @sql varchar(8000)

SET @sql = 'SELECT * FROM Person.Contact WHERE LastName = ''Allen'''
EXECUTE (@sql)
SET @sql = 'SELECT * FROM Person.Contact WHERE LastName = ''Ackerman'''
EXECUTE (@sql)
GO

DECLARE @sql nvarchar(4000)

SET @sql = 'SELECT * FROM Person.Contact WHERE LastName = @LastName'
EXECUTE sp_executesql @sql, N'@LastName NVARCHAR(80)', @LastName = 'Allen'
EXECUTE sp_executesql @sql, N'@LastName NVARCHAR(80)', @LastName = 'Ackerman'
GO

SELECT
       cp.usecounts,
       cp.size_in_bytes,
       st.text
FROM sys.dm_exec_cached_plans cp
OUTER APPLY sys.dm_exec_sql_text (cp.plan_handle) st
JOIN sys.dm_exec_query_stats qs ON qs.plan_handle = cp.plan_handle
GO
```

Bien sûr, autant que possible, préférez la syntaxe avec `sp_executesql`.

Il est à noter qu'une fonctionnalité similaire est proposée par les bibliothèques d'accès client, qui permettent de « préparer » le code SQL à envoyer, pour favoriser la réutilisation du plan d'exécution. OLEDB expose l'interface `IcommandPrepare` pour ce faire. N'utilisez pas systématiquement cette fonctionnalité, elle est souvent

plus coûteuse qu'utile. Réservez-la à des exécutions multiples – au moins plus de trois fois – de la même requête.

9.4 À PROPOS DU CODE .NET

Depuis SQL Server 2005, vous pouvez écrire des objets de code SQL Server dans un **langage .NET**, compilé en langage intermédiaire (MSIL). Après avoir ajouté l'assembly compilée dans les métadonnées de votre base de données (à l'aide de la commande `CREATE ASSEMBLY`), vous pouvez utiliser les classes et méthodes de votre assembly pour créer des procédures stockées, déclencheurs, fonctions utilisateur, types de données ou fonctions d'agrégation. À la sortie de SQL Server 2005, certaines personnes se posaient la question de passer du langage T-SQL vers .NET pour la plupart de leurs besoins en requête, les développeurs ayant souvent une connaissance plus poussée de C# ou VB.NET que du langage SQL. Cette question, bien sûr, n'a pas eu de suite. Il est inutile, et d'ailleurs impossible, de vouloir remplacer T-SQL. SQL Server est un SGBDR qui respecte autant que possible les règles de Codd[1], et les principes du client-serveur : les données ne sont accessibles qu'à travers le langage de requête, ce langage étant SQL. Ainsi, même dans du code .NET, l'accès au contenu des tables ne peut être fait qu'à travers une connexion ouverte sur le serveur (une méthode de connexion *in-process* rapide est disponible, appelée la « connexion de contexte »), et le passage de commandes T-SQL. Pour manipuler les objets de base, le code .NET *in-process* dispose de… ADO.NET 2.0. Il n'y a donc aucun moyen de remplacer la saisie de code SQL. Cette « contrainte » « évolue » avec SQL Server 2008, dans lequel il est théoriquement possible de créer des objets .NET qui utilisent la technologie de mapping relationnel-objet nommée LINQ (*.NET Language Integrated Query*). Il devient ainsi possible de se passer de saisir du code SQL, les bibliothèques .NET s'occupant pour vous de traduire votre prose objet en requêtes. Il est clair que cette voie est en opposition avec le sujet de ce livre. Si vous êtes soucieux de performances, fuyez à tout jamais cette voie.

Utilisez la technologie à bon escient. Le langage de requêtes natif de SQL Server, Transact SQL, doit rester la solution de choix pour toute requête. Éventuellement, comme SQL est très peu souple en ce qui concerne les traitements en boucle et l'algorithmique, et comme le code procédural est en général plus lent en T-SQL que dans un langage compilé, dans des cas comme la manipulation de chaînes de caractères ou des opérations mathématiques, l'intégration du CLR peut s'avouer un allié utile. Nous parlons évidemment ici de la tentation de déplacer du code s'exécutant

1. Edgar F. Codd a publié en 1985 dans le magazine ComputerWorld deux articles restés célèbres, qui listent treize règles, ou lois, essentielles d'un système de bases de données relationnelles. La règle n° 12, notamment, connue sous le nom de règle de non-subversion, stipule que si le système dispose d'un langage de bas niveau, ce langage ne peut pas contourner ou remettre en cause les contraintes de sécurité et les règles d'intégrité énoncées au plus haut niveau. Voir http://www.sqls-pot.com/Les-regles-de-CODD-pour-un-SGBD-relationnel.html.

traditionnellement dans une couche métier, dans le moteur de base de données, ce n'est pas forcément non plus une bonne idée. D'un côté vous gagnez en diminution de transfert réseau entre la couche de données et la couche métier, d'un autre, vous grignotez avidement les ressources d'une machine dédiée normalement à la manipulation de données brutes. Ne vous y lancez qu'en cas de réelle nécessité.

Bibliographie

Ouvrages utiles

Itzik BEN-GAN, *Inside Microsoft SQL Server 2005 : T-SQL Querying*, Microsoft Press, 2006.

William BOSWELL, *Inside Windows Server 2003*, Addison Wesley, 2003.

Louis DAVIDSON, Kurt WINDISCH, et Kevin KLINE, *Pro SQL Server 2005 Database Design and Optimization*, Apress, 2006.

Kalen DELANEY, *Inside Microsoft SQL Server 2005 : Query Tuning and Optimization*, Microsoft Press, 2007.

Kalen DELANEY, *Inside Microsoft SQL Server 2005 : The Storage Engine*, Microsoft Press, 2006.

Edward L. HALETKY, *VMware ESX Server in the Enterprise : Planning and Securing Virtualization Servers*, Prentice Hall PTR, 2008.

Ken HENDERSON, *SQL Server 2005 Practical Troubleshooting : The Database Engine*, Addison-Wesley Professional, 2006.

Ken HENDERSON, *The Guru's Guide to SQL Server Architecture et Internals*, Addison-Wesley Professional, 2004. (*traite de SQL Server 2000, mais de nombreux passages sont utiles à la compréhension du fonctionnement de SQL Server 2005 et 2008*).

Ralph KIMBALL et Margy ROSS, *Entrepôts de données. Guide pratique de modélisation dimensionnelle, 2e édition*, Vuibert Informatique, 2002.

Steven WORT, Christian BOLTON, et Justin LANGFORD, *Professional SQL Server 2005 Performance Tuning*, Wrox Press, 2008.

Papiers techniques et livres blancs Microsoft

(à chercher sur le site de Microsoft, les liens étant parfois changés)

Database Engine Tuning Advisor (DTA) in SQL Server 2005 – document Word.

Références web

- SQL Server Query Optimization Team, *Distributed/Heterogeneous Query Processing in Microsoft SQL Server*, http://citeseer.ist.psu.edu/732761.html
- *Physical Database Storage Design* - http://www.microsoft.com/technet/prodtechnol/sql/2005/physdbstor.mspx
- Travaux de recherches sur lesquels est basé le *Database Engine Tuning Advisor* – http://research.microsoft.com/research/dmx/autoadmin/
- *SQL Server I/O Basics*, Chapter 2 – http://www.microsoft.com/technet/prodtechnol/sql/2005/iobasics.mspx
- *Working with tempdb in SQL Server 2005* – http://www.microsoft.com/technet/prodtechnol/sql/2005/workingwithtempdb.mspx

Sites

- GUSS : http://www.guss.fr/
- developpez.com : http://sqlserver.developpez.com/
- forums developpez.com : http://www.developpez.net/forums/
- SQL Server Performance – http://www.sql-server-performance.com/
- SQL Server Central – http://www.sqlservercentral.com/
- SQL Server Magazine (payant) – http://www.sqlmag.com/
- Site Microsoft Technet – http://technet.microsoft.com/fr-fr/sqlserver/
- SQL Team – http://www.sqlteam.com/
- SQL Server Community – http://www.sqlcommunity.com/
- Simple Talk (site communautaire de RedGate, un éditeur d'outils tiers pour SQL Server) – http://www.simple-talk.com/sql/
- Newsgroups Microsoft (USENET) : microsoft.public.fr.sqlserver, microsoft.public.sqlserver.programming. Cherchables via Google Groups : http://groups.google.com/

Blogs

- SQL Server Storage Engine – http://blogs.msdn.com/sqlserverstorageengine/
- SQL Server Manageability Team Blog – http://blogs.msdn.com/sqlrem/
- Microsoft SQL Server Development Customer Advisory Team – http://blogs.msdn.com/sqlcat/
- Microsoft SQL Server Support Blog – http://blogs.msdn.com/sqlblog/
- SQL Programmability & API Development Team Blog – http://blogs.msdn.com/sqlprogrammability/

- SQL Protocols – http://blogs.msdn.com/sql_protocols/
- Solid Quality Learning Weblogs – http://solidqualitylearning.com/blogs/
- Blogs by Current and Former Microsoft Most Valuable Professionals – http://msmvps.com/tags/SQL+Server/default.aspx
- SQL Server Community Blogs – http://sqlblogcasts.com/blogs/
- Tibor Karaszi – http://www.karaszi.com/SQLServer/
- SQLJunkies Weblogs – http://www.sqljunkies.com/WebLog/
- Australian SQL Server User Group Blogs – http://blogs.sqlserver.org.au/blogs/
- SQL Skills team blogs – http://www.sqlskills.com/blogs.asp
- Christian Robert, MVP – http://blogs.developpeur.org/christian/
- Brian Knight – http://www.whiteknighttechnology.com/cs/blogs/
- Liste des blogs des équipes SQL Server de Microsoft – http://blogs.msdn.com/sqlserverstorageengine/archive/2007/02/14/wow-lots-of-blogs-from-the-sql-product-team.aspx
- sqlblog.com – http://sqlblog.com/blogs

Index

S

T

INFOPRO

TYPE D'OUVRAGE

L'ESSENTIEL	SE FORMER	RETOURS D'EXPÉRIENCE

MANAGEMENT DES SYSTÈMES D'INFORMATION

APPLICATIONS MÉTIERS

ÉTUDES, DÉVELOPPEMENT, INTÉGRATION

EXPLOITATION ET ADMINISTRATION

RÉSEAUX & TÉLÉCOMS

Rudi Bruchez

OPTIMISER SQL SERVER

Dimensionnement, supervision, performances du moteur et du code SQL

Cet ouvrage s'adresse aux développeurs, administrateurs de bases de données (DBA), consultants et professionnels IT qui ont la responsabilité d'une base SQL Server.

Un système comme SQL Server joue un rôle central dans l'informatique d'une entreprise, et il est indispensable de s'assurer que l'on en obtient les meilleures performances possibles, surtout sur des volumes importants.

Cet ouvrage va vous aider à tirer le maximum de SQL Server dans ses **versions 2005 et 2008**. En comprenant l'architecture et le fonctionnement du moteur de ce gestionnaire de base de données, vous saurez comment choisir votre matériel et surveiller votre serveur. Vous apprendrez à interpréter les résultats des **compteurs de performances**, et à utiliser les **traces SQL**. Vous saurez comment utiliser les index de façon optimale. Vous comprendrez les problématiques de **qualité** du modèle de données et du code SQL. Vous maîtriserez en détail le mécanisme des transactions, le verrouillage, les blocages et les verrous mortels qui peuvent s'ensuivre. Vous apprendrez enfin à **optimiser** vos procédures stockées.

RUDI BRUCHEZ

est consultant et formateur spécialisé sur SQL Server depuis 2001. Certifié Microsoft (MCDBA, MCT et MCITP) et MVP (*Most Valuable Professional*) sur SQL Server, il est aussi rédacteur/modérateur sur le forum SQL Server du site communautaire developpez.com, principal site francophone pour les développeurs. Il répond également régulièrement aux questions posées sur le *newsgroup* officiel de Microsoft consacré à SQL Server (microsoft.public.fr.sqlserver).

MVP Microsoft® Most Valuable Professional

SUPPLÉMENTS en ligne

Retrouvez sur *www.babaluga.com* et sur *www.dunod.com* le code source des exemples de l'ouvrage

ISBN 978-2-10-053750-1

www.dunod.com

DUNOD