

Optimisation des bases de données MS SQL Server

par Frédéric Brouard

Date de publication : 30 mai 2008

Dernière mise à jour :

On a beau répéter que l'optimisation de bases de données ne relève pas d'outils ni d'automatismes, mais du simple artisanat, il y a toujours quelques personnages pour prétendre qu'il suffit de faire ceci ou cela, pour obtenir de bonnes performances.

Si les choses sont plus complexes qu'il n'y parait, il n'en reste pas moins vrai que certains principes simples et des règles d'une grande évidence qui devraient guider l'équipe en charge du développement d'un projet informatique, sont souvent ignorées voire sciemment bafouées.

Cette série d'article a pour but de présenter l'optimisation des bases de données sous toutes ses facettes. Il ne s'agit pas d'un cours technique (pour cela la place manquerait), mais plus globalement d'une réflexion sur les erreurs à ne pas commettre, celles à rectifier et les mesures à prendre dans le cadre de l'exploitation courante d'une base de données.

Copyright et droits d'auteurs : La Loi du 11 mars 1957 n'autorisant aux termes des alinéas 2 et 3 de l'article 41, d'une part que des copies ou reproductions strictement réservées à l'usage privé et non [...] à une utilisation collective, et d'autre part que les analyses et courtes citations dans un but d'illustration, toute reproduction intégrale ou partielle faite sans le consentement de l'auteur [...] est illicite. Le présent article étant la propriété intellectuelle conjointe de Frédéric Brouard et de SQL Server magazine, prière de contacter l'auteur pour toute demande d'utilisation, autre que prévu par la Loi à SQLpro@SQLspot.com

I - Fonctionnement de SQL Server : Règles de base de l'optimisation

Ma pratique de l'audit, comme les cours que j'écris et enseigne au sujet de l'optimisation SQL Server m'a permis de constater à quel point l'ignorance est grande chez les techniciens de l'informatique : le modèle de données, le choix des types, le codage des procédures, la concurrence des transactions...

Autant de sujet mal maîtrisés parce que vus uniquement sur le plan scolaire. Il ne faut jamais oublier que la base de données est le point critique de toute application. *Nul ne saurait se passer de données*, tel devrait être le premier commandement de la Loi informatique... et pourtant nombreuses sont les entreprises qui n'ont aucune conscience que leur capital, leur patrimoine informatique, réside essentiellement dans leur données, la qualité de ces données, et les performances que l'on peut exiger d'elles. La plupart des organisations sont obnubilées par l'aspect "sapin de Noël" de leurs merveilleux applicatifs graphiques, mais dans les faits ces logiciels constituent en quelques sortes les maigres arbres qui masquent une forêt : celle des données. A l'aire du *trash* logiciel, à l'époque du décisionnel, les entreprises devraient se raccrocher encore plus à leurs données qui sont le seul point commun entre les différentes refontes de leurs programmes et qui leur permettent de piloter le navire entreprise. Or c'est rarement le cas. Les performances demandées sont loin d'être atteintes du fait de modèles déficients, d'écritures tarabiscotées, d'une piètre qualité de données, quand ce ne sont pas des serveurs mal exploitées et des bases peu ou pas administrées...

Ce profond décalage entre ce qu'il faudrait faire et la réalité à deux origines distinctes : un manque de formations scolaires et professionnelles et le discours marketing tout azimut.

Sur le manque de formation scolaire, force est de constater que les programmes en matière de bases de données relationnelles - réduits aujourd'hui à peau de chagrin - ont été calés sur la version 1992 du langage SQL et la plupart du temps sur le dialecte abscond et peu normatif du SGBDR Oracle ⁽¹⁾. Il en résulte à ce point une méconnaissance de SQL, que, récemment, dans le cadre de tests de recrutement, la plupart des candidats, confrontés à une jointure interne à la norme SQL de 1992, considérait cela comme un élément spécifique du langage propre à un SGBDR particulier ! Or nous sommes passés de la norme de 1992 dite SQL 2 aux normes SQL:1999 ⁽²⁾ et SQL:2003 ⁽³⁾, chacune ayant apporté des éléments aujourd'hui traduits dont les plus importants figurent aujourd'hui dans la version 2005 du SGBDR SQL Server.

Comme il suffit la plupart du temps de mettre sur son CV "*langage SQL*" pour que le recruteur suppose que le candidat maîtrise les technologies des bases de données relationnelles, on en arrive au fait que l'informaticien devient de fait un bon pisseur de requêtes SQL et qu'il acquiert assez rapidement le statut d'architecte des données par le simple fait qu'il sait utiliser l'interface graphique *machin* pour créer une table.

Du côté du discours mercatique, tous sont coupables... ! A commencer par Microsoft qui fait croire à la simplicité ⁽⁴⁾ d'un SQL Server 2005, alors que même les meilleurs experts s'arrachent les cheveux actuellement pour contrôler une bête dont l'étendue des possibilités nécessite au bas mot une vingtaine de jours de formation pour en absorber la plupart des aspects...

Coupables aussi les éditeurs de solution de développement en tout genre qui vous font croire que plus n'est besoin de mettre les mains dans le cambouis SQL parce que leur atelier, Ô combien intelligent, calcule automatiquement les bonnes structures et écrivent les bonnes requêtes...

De fil en aiguille, ces outils et ces hommes construisent une base, une application et la livre. Le tout est alors mis en exploitation. Passé les quelques mois de la garantie ou le client peut réclamer - et ou, par bonheur, il n'y a presque pas de données dans la base - le système semble donner toute satisfaction. Puis le temps passe et joue en défaveur

- (1) Oracle a attendu la version 9 sortie en 2001 soit 9 ans de retard pour se mettre à faire des jointures à la norme SQL 2 datant de 1992. Pour autant Oracle a été l'un des acteurs les plus impliqués dans l'élaboration de cette norme !
- (2) Introduisant notamment le relationnel objet, les requêtes récursives et les fonctions de fenêtrage?
- (3) Introduisant notamment l'auto incrément et le type XML.
- (4) L'IHM est sensée masquer la complexité au commun des mortels utilisateurs. Pour autant, peut-on considérer que l'informaticien et qui plus est le développeur, soit si un mortel si commun ? Les interfaces MS de SQL Server 2005 sont si bien pensées qu'il est plus que difficile de se rendre réellement compte de ce que fait la machine derrière un simple clic !

du client : le volume de données s'accroît et les performances se mettent à chuter, souvent brutalement ⁽⁵⁾, alors qu'aucune alerte ne s'était préalablement produite.

Le client mécontent tance l'auteur du logiciel qui se met à incriminer la plupart du temps le SGBDR mais le plus souvent la machine : "votre serveur n'a pas assez de mémoire..., de disques..., de processeurs..."

Il est bien difficile pour une SSII ou un éditeur d'admettre que sa faible compétence l'a conduit à une impasse. C'est difficile par ce que reconnaître cela suppose des compétences que l'on n'a pas et que faire appel à un prestataire externe pour auditer la qualité de ce que l'on a fait n'est pas franchement coutume dans notre mauvaise habitude de négliger l'importance du service, la nécessité de la critique, le regard extérieur.

Pour étayer mes propos, je citerais trois exemples.

Le premier est le cas de cet éditeur d'informatique médicale qui visait l'administration de l'hôpital et avait bâti l'architecture de stockage de ses informations en reprenant les structures de données hérité de son vieux logiciel à base de fichiers. Il en résultait un salmigondis de rubriques éparses et redondantes. De surcroît il avait commis la faute magistrale d'utiliser le type REAL au lieu du type DECIMAL pour stocker ses données comptables. Chaque année, à cause des cumuls, les erreurs d'écart d'arrondis propre au type REAL empêchaient les balances comptables d'être équilibrées et le bilan de s'avérer exact. Il manquait toujours quelques centimes, à droite, à gauche, au milieu et sur les cotés ! N'ayant toujours pas compris pourquoi, il publiait chaque année une "rustine" destinée à corriger ces écarts. Ce processus enfin intégré à sa chaîne de maintenance, il commença de s'attaquer au marché des gros hôpitaux constitué par les CHU... Mais là les volumes de données n'étant pas les mêmes, les performances commencèrent à se révéler lamentables par la faute du modèle : la redondance comme l'absence de normalisation ayant conduit à l'écriture de requêtes gigantesques, illisibles, ampoulées, inextricables et, compte tenu du volume, d'une durée exceptionnellement longue. Finalement, l'éditeur était coincé : soit il indiquait à son client que celui-ci devait prendre un serveur surdimensionné pour pallier le déficit de performance lié à la structure de la base - et dans ce cas il n'arrivait pas à vendre sa solution - soit il la vendait quand même mais se retrouvait en exploitation avec le mécontentement grandissant de ses clients, mécontentement qui alla jusqu'au procès avant qu'il n'abandonnât finalement ce marché pourtant prometteur.

Le second exemple est du même tonneau et curieusement dans le même secteur. On prend une application qui marche, écrite pour Access et on utilise le "wizard", ce magicien d'assistant pour porter son application en SQL Server. Il a suffi de moins d'une heure suivi de deux ou trois corrections et voilà un produit que ses auteurs croient capable des performances nécessaires et suffisantes pour attaquer les marchés les plus gros, c'est à dire les plus juteux... Hélas c'était sans compter sur la perversité de cet assistant - en fait de magicien, une sorcière déguisée - qui a traduit toutes les requêtes à envoyer au serveur, en élément si parcellaires que pour certains écrans, il fallait envoyer plus d'une centaine de requêtes unitaires ⁽⁶⁾. De plus, le logiciel avait été conçu pour scruter certaines données : toute les secondes pour les unes, toutes les vingt secondes pour les autres, des requêtes étaient lancées sur le serveur pour se renseigner sur l'état de telle ou telle donnée. Avec une simple projection sur un nombre d'utilisateur potentiel de 50, on arrivait au chiffre pharamineux de 750 000 requêtes lancée en pure perte pendant la période d'exploitation journalière du logiciel sans qu'aucune action humaine sur l'interface n'ait été entreprise !

Le troisième cas est encore plus révélateur : dans une compagnie aérienne, une application de planification était écrite avec un des outils de développement les plus fameux pour son aspect "sapin de Noël" : de clinquant écrans bariolés, agrémentés de multiples objets visuels surchargés de données, clignotant de messages surgissant, permettaient de réaliser le planning des équipages pour la rotation des avions. Mais les temps de réponse étaient parfois jugés anormaux dans leur durée et cela de manière aléatoire. En fait chaque fois que le pointeur de souris se déplaçait sur l'écran, des centaines de requêtes étaient envoyées au serveur. Le planning était représenté par de très petites cases dont le temps constituait l'ordonnée et le vol, l'abscisse. Pour se renseigner sur un équipage, les développeurs avaient eu la brillante idée de faire surgir la composition de l'équipage dans une info bulle en allant demander les informations au serveur. Mais cet appel se faisait à chaque fois que la souris parcourait le moindre pixel à l'intérieur de la case ! Bien évidemment en test cela s'avérait rapide et éblouissant... tout au moins sur un jeu de données plus que restreint. Mais sur une application en exploitation depuis des mois et avec des écrans 21 pouces, l'hallucinant clignotement informationnel mettait à genou un serveur pourtant des plus robustes. D'autant plus qu'encore une fois le modèle

(5) Voir la conclusion pour l'explication de ce phénomène

(6) Le fameux DBlookUp d'Access entre autres !

de données avait faiblement pris en compte la problématique des données temporelles, qui sont généralement les plus complexes à modéliser et à optimiser.

Tout ces exemples montrent bien comment un système qui a bien marché à un moment donné peu subitement basculer dans la plus sombre des performances sans que l'auteur par son manque de prévision n'arrive à comprendre pourquoi. Dans ces trois exemples, le modèle des données est fortement en cause. Comme il l'est dans tous les audits que j'ai réalisés. Et il est souvent très tard pour rectifier le tir, du fait de l'intense effort à entreprendre pour remanier un modèle de données et le code sous-jacent. Et souvent, au lieu d'entamer une nouvelle version en partant d'un modèle de données sain et performant, on préfère tenter l'aventure de la modification par petite touche, aventure qui finalement n'aboutit que très rarement et s'avère finalement bien plus couteuse qu'une refonte globale.

Je suis pour ma part effaré par le nombre d'entreprises qui conçoivent des logiciels sans avoir jamais réalisé une analyse et une modélisation de données sérieuse. Si vous avez le moindre doute sur votre éditeur ou sur votre prestataire, demandez-lui le modèle conceptuel de données de l'application. Si vous obtenez quelque chose, ce sera le plus souvent un modèle physique de la base, prouvant que l'analyse des données, c'est à dire l'étude du modèle conceptuel, n'a jamais été entreprise ⁽⁷⁾. C'est la première phase du basculement vers la médiocrité des performances, et la phase la plus cruciale puisqu'il est difficile de revenir sur un modèle bâclé.

Mes audits m'on conduit à montrer, qu'en dehors des problématiques matérielles, la quantité d'erreurs et les gains potentiel se situaient en premier dans le modèle de données, en second dans l'écriture du code et enfin dans l'exploitation (un pourcentage négligeable se situant au niveau du serveur SQL et de l'OS).

Ce que l'on peut approximativement synthétiser par le tableau ci après :

Cause	Erreurs	Gain global possible
modèle de données	50 à 80 %	40 à 90 %
écriture du programme, écriture des requêtes	40 à 70 %	20 à 60 %
exploitation	10 à 30 %	10 à 50 %
paramétrage serveur et OS	0 à 5 %	0 à 5 %

NOTA : pour mesurer l'erreur, je me suis servi de mes audits et des points que j'ai relevé, sachant qu'une même erreur située dans plusieurs programmes ne comptait que pour une seule erreur.

Assez de discours maintenant, venons en au vif du sujet. Je vous propose de naviguer en ma compagnie dans la matière propre à l'optimisation des bases de données avec SQL Server. Pour cette navigation, je vous propose différentes escales. D'abord comprendre le fonctionnement d'un serveur MS SQL Server. Ensuite de voir en quoi le matériel peut nous aider. Vous aurez compris qu'une étape importante sera constituée par le modèle des données. Puis nous irons farfouiller du côté du code pour enfin terminer par l'exploitation d'une base de données. Tout en sachant que malgré la verve que je compte mettre nous n'aurons fait qu'effleurer le sujet.

Mais en avant toute mettons le cap dès aujourd'hui sur les principes de base qui constitue le fonctionnement d'un serveur SQL...

En lisant les lignes suivantes, vous allez comprendre l'un des principaux mystères liés aux phénomènes de performance : pourquoi les performances ne sont pas linéaires et peuvent conduire brutalement à un point de rupture... Ou, pour paraphraser monsieur de La Palice, *pourtant, ça marchait bien avant* !

(7) Le Modèle Conceptuel de Données ou MCD se caractérise par la présence d'entité et d'association ou relations, alors que le modèle physique ne comprend que des tables qui représente un sur ensemble de l'entité (en fait l'entité et les éléments techniques de la relation).

II - fonctionnement d'un serveur SQL

A bien y réfléchir, un serveur SQL ⁽⁸⁾ que nous simplifierons à une base et un seul utilisateur, se compose de peu de choses : un fichier de données, un fichier de journal, un moteur relationnel, un moteur de stockage et un mécanisme d'écriture dans le journal qui assure la bonne marche des transactions.

A cela il faut ajouter la mémoire vive qui va jouer un rôle essentiel, les disques comptant pour l'instant dans notre histoire pour quantité négligeable ⁽⁹⁾ ...

Notre singulier utilisateur veut extraire des données et pour cela lance la commande SELECT... Aussitôt cet ordre SQL - qui en fait constitue une transaction - est enregistré par le journal de transaction. Puis le relais est donné au moteur transactionnel qui scrute la mémoire afin de savoir si les lignes de tables concernées par la requête sont en mémoire. Si elles le sont, la requête est exécutée en lisant les données en mémoire, l'extraction est produite et le résultat est envoyé au demandeur.

Si les données ne figurent pas en mémoire, une demande est envoyée au moteur relationnel d'avoir à les y placer. Tant que cette demande n'est pas entièrement satisfaite, la requête est mise en sommeil. Lorsque le moteur de stockage a effectué son travail, alors le moteur relationnel reprend le sien...

Une fois les données délivrées, une marque est effectuée dans le journal pour signaler l'achèvement (bon ou mauvais) de la transaction.

Autrement dit, règle N°1 :

- La lecture des données s'effectue toujours en mémoire.

Notre singulier utilisateur veut mettre à jour une donnée... Il utilise pour cela la commande UPDATE. La transaction est enregistrée dans le journal. La donnée est-elle présente en RAM ? Supposons que oui. On modifie directement la donnée en mémoire et on positionne un flag pour indiquer que celle ci a changé. La transaction opérée est marquée terminée dans le journal.

De temps en temps un processus particulier du moteur relationnel scrute la mémoire à la recherche des données modifiées. Il dispose pour cela des flags et va regrouper les écritures pour en optimiser l'enregistrement sur le disque.

On en conclut deux nouvelles règles N°2 et N°3 :

- Les mises à jour de données s'effectuent d'abord en mémoire (INSERT, UPDATE, DELETE)
- L'écriture des données est différée et regroupée

Notre utilisateur unique décide de renouveler le SELECT qu'il a entrepris au début de son aventure avec nous. Il est surpris par la rapidité avec laquelle les données lui parviennent cette fois-ci. Pas étonnant les données sont déjà en mémoire. Mieux, la requête est déjà préparée car elle aussi est mise en cache : la façon de la traiter - *son plan d'exécution* - a déjà été établi. Il n'y a donc plus que le temps de traiter et de renvoyer les données. La mise en cache de la requête consiste tout simplement à répertorier la chaîne de caractère qui symbolise l'exécution de l'ordre SQL, avec la référence de son plan d'exécution.

La règle N° 4 est donc :

- Le texte d'une requête est mis en cache avec son plan d'exécution

Ces quatre règles édictées, nous pouvons dès à présent constater que l'élément le plus important pour un serveur SQL est la mémoire ou sont mis en cache tant les données que les requêtes (et plus généralement les procédures, fonctions, triggers et règles... c'est à dire tous les objets codés en Transact SQL).

(8) Nous avons grandement simplifié le modèle afin qu'il soit compréhensible au commun des mortels, mais plus particulièrement à l'informaticien lambda.

(9) Pour ce qui est du processeur, inutile d'en parler puisqu'il est omniprésent !

Il nous faut donc nous intéresser au cache. Comment fonctionne-t-il ? Telle est la question, et pour la résoudre nous devons adopter deux points de vue différents : celui des données et celui des procédures, toute en sachant qu'en matière d'alimentation du cache : la règle est commune, c'est LRU qui dicte sa loi... LRU pour *Last Recent Use*, c'est à dire "moins récemment utilisé". Autrement dit pour faire place à une nouvelle mise en cache, l'espace mémoire le plus anciennement accédé doit céder sa place.

II-A - Cache et RAM

En fait LRU ne va se déclencher que si SQL Server manque de mémoire. Tant que SQL Server constate qu'une partie de la RAM n'est pas mis à sa disposition ⁽¹⁰⁾, il la prendra, quitte à soutirer toute la mémoire disponible. Il est programmé pour vampiriser toute la RAM, celle de l'OS exceptée, au mépris de toutes les applications. Seul l'OS s'il s'estime stressé est autorisé à lui demander d'en restituer... Plus il a de mémoire et plus la base est petite, moins LRU se déclenchera. Dans un tel cas, des données vieilles de plusieurs années peuvent persister en mémoire, jusqu'à l'arrêt sur serveur.

L'adéquation RAM/mise en cache peut être mesurée à l'aide des compteurs du moniteur de performance. Dès que cette adéquation commence à diminuer, les données doivent être lues sur le disque, et c'est là que le système bascule...

Prenez conscience que le temps de lecture d'un bit dans une RAM est de l'ordre de 10 nano secondes tandis que celui d'un disque est de l'ordre de 10 ms. Entre les deux un écart de 1 000 000. Oui, j'ai bien dit UN MILLION. Une lecture disque est un million de fois moins rapide qu'une lecture mémoire. Cette différence doit toutefois être amendée par le fait que d'un côté, entre la mémoire et le processeur il existe tout un circuit électronique et des micro programmes pour charger et décharger le processeur, et que de l'autre côté, le disque lui même possède souvent un cache, ce qui fait que l'un dans l'autre l'écart effectif est plus proche d'un rapport de mille que du million...

En conclusion, dès que le ratio de mise en cache commence à diminuer singulièrement, les performances baissent de manière très sensible. Une baisse de ce ration de 20% est déjà très sensible. Au delà, la file d'attente des demandes de lecture du disque se met à s'allonger démesurément et l'on se retrouve dans une situation d'engorgement ⁽¹¹⁾ digne du périphérique un vendredi soir vers 17 heures avec chaussée extérieur un accrochage et chaussée intérieure une voie immobilisée par une panne d'essence !

Mais nous savons déjà comment optimiser notre base de données SQL : il suffit d'y ajouter de la RAM !

C'est si vrai que les pros de la chose, déterminent les caractéristiques des serveurs qu'ils vont devoir acheter en calculant ce qui doit être mis en cache. Non toute la base ne doit pas tenir en mémoire. Mais ce qui doit y être, c'est la "*fenêtre de données*", c'est à dire la frange des données réellement et couramment exploitée à l'instant T.

Par exemple pour une comptabilité, ce qui constitue la fenêtre des données c'est à 50% le mois en cours, 30% le trimestre en cours, 20% l'année en cours et 10 % le reste... Ce qui fait qu'une base de données comptable avec une moyenne de 4 Go par année et un historique de 6 ans, aura besoin de : $(4 / 12) * 0,5 + (4 / 4) * 0,3 + 4 * 0,2 + 4 * 6 * 0,1 = 3,67$ Go de RAM !

Bien entendu, il faut rajouter à cela le cache de procédures (généralement assez minime face au données), les accès client (qui peut être conséquent s'il y a de nombreux utilisateurs), et l'OS.

Par exemple dans un tel cas, avec un nombre d'utilisateur ne dépassant pas 200, j'aurais tendance à préconiser une RAM de 6 à 8 Go.

⁽¹⁰⁾C'est pourquoi SQL Server nécessite impérativement un serveur dédié car les autres applications risquent d'étouffer sauf à restreindre la mémoire allouée à ce dernier

⁽¹¹⁾On parle alors de contention. Nos grands-mères portaient des bas de contention pour éviter les effets des varices. Ce sont maintenant les PDG des grandes multinationales qui s'y mettent à cause des effets de phlébite dus aux voyages en avion !

Mais encore faut-il que le modèle de données soit en adéquation avec le fenêtrage! Nous verrons que la façon de modéliser, et en particulier la structure des clefs a une influence si importante que le calcul de fenêtrage va s'avérer très en dessous de la réalité...

Première commandement : la RAM doit être en adéquation avec le volume des données à traiter. Tout ce qui peut être fait pour augmenter la RAM et en diminuer son contenu, va dans le sens de l'amélioration des performances.

S'il suffisait de rajouter toujours de la RAM, mon papier se terminerait ici... Mais notons déjà que toutes les éditions et versions d'OS Windows Server et de MS SQL Server et en particulier celles en 32 bits ne permettent pas n'importe quelle quantité de RAM en ligne. Cela va de 2 Go à 64. Autrement dit il faut aussi l'économiser, c'est ce que nous allons voir ci après.

II-B - Le cache des procédures

S'il est un cache bien amusant, c'est celui des procédures. Il sert simplement à se souvenir que telle ou tel bout de code - en fait et plus précisément, telle ou telle requête - a déjà été joué et que l'on en a déduit un plan d'exécution.

Toute requête SQL est en fait une chaîne de caractères. Pour savoir si une requête a déjà été jouée, aux paramètres près ⁽¹²⁾, il suffit de comparer les chaînes de caractères des requêtes déjà présentes dans le cache avec la nouvelle arrivée. Si la phrase est identique, le plan de requête déjà établi est immédiatement rejoué. On économise ainsi un nouveau calcul du plan de requête.

Mais les caches de procédures, qu'ils soient Oracliens ou Microsofties portent tous le même défaut : pour être rapide ils se contentent d'analyser binaires les chaînes de caractères. Autrement dit une même requête écrite, l'une avec des noms d'objet en majuscule et l'autre, sa sœur jumelle, écrite avec un mélange de majuscule et de minuscule feront l'objet de deux entrées dans le cache donc de deux plans de requêtes. C'est à dire : deux fois plus de temps de calcul pour le plan mais surtout deux fois plus de volume pour le cache. Or plus de cache mangé pour les procédures, c'est moins de cache pour les données...

Vous pouvez donc en déduire une première règle de minimisation du cache : écrire toujours vos requêtes de la même façon, en particulier pour la casse.

Et pour cela vous disposez d'un atout simple avec SQL Server : la collation d'installation ! Préférez là binaire...

Second commandement : les requêtes devront toujours être écrites de la même façon (en particulier avec la même casse) afin de minimiser globalement le cache.

II-C - Le cache des données

Pour qu'un cache soit efficace il faut que les processus de mise en mémoire et de déchargement de la mémoire soient les plus optimisés possible.

Pour ce faire, on a longuement étudié les mécanismes de lecture et d'écriture des disques et des mémoires et l'on en a conclu la chose suivante : il ne sert à rien d'aller chercher une seule ligne d'une seule table sur un disque, car le temps perdu par les mécanismes matériels de déplacement des têtes de lecture seraient incommensurablement plus longs que l'envoi des quelques octets de la ligne. On en a donc conclu que le mieux était de lire un paquet de données que l'on a estimé à 64 Ko au minimum et on lui a donné le nom d'extension ⁽¹³⁾. Mais ces 64 Ko étant en fait trop grands pour constituer à eux seuls une ligne d'une table, on a parcellisé ces paquets en pages de données

(12) C'est-à-dire les expressions de valeurs figurant essentiellement dans les clauses WHERE, HAVING et SET (UPDATE)

(13) En anglais extent, en fait la granule de stockage de base est la page de 8 Ko mais la lecture comme l'écriture se fait par bloc de 8 pages de même nature. C'est la raison pour laquelle on a nommé ce bloc "extension".

de 8 Ko, pages pouvant contenir chacune une ou plusieurs lignes d'une table. Voilà pourquoi la longueur maximale d'une ligne d'une table a longtemps été de 8 Ko, moins quelques octets "techniques" ⁽¹⁴⁾.

Pour refléter cette contrainte, la mémoire cache en RAM est une reproduction fidèle de l'organisation en page du fichier disque contenant les données. Autrement dit, le cache de données est lui aussi constitué de pages de 8 Ko, et les pages sont copiées du disque à la RAM à l'identique de manière binaire.

La plupart du temps un page contiendra donc de multiples lignes... Mais lesquelles ?

C'est là qu'intervient la notion d'index...

- Soit la table n'a pas d'index et les lignes sont stockées dans l'ordre chronologique de leur insertion. Cela convient bien aux très petites tables ⁽¹⁵⁾.
- Soit la table possède un index de type CLUSTER ⁽¹⁶⁾ et les lignes sont triées dans l'ordre des données de cet index.
- Soit enfin la table ne possède pas d'index cluster et les lignes sont stockées dans l'ordre d'insertion, avec en sus une structure de données supplémentaire : l'index *heap* ⁽¹⁷⁾.

Si vous avez créé une clef primaire sur une table avec les options par défaut, alors la création de cette clef primaire a entraîné la création d'un index cluster et les lignes de la table sont physiquement stockées de manière triées.

Ceci n'est pas sans conséquence avec les problématiques de cache. Pour bien en comprendre l'enjeu, prenons un exemple criant de vérité. Notre développeur a créé une table des factures et pour modéliser sa clef a pris le parti de concaténer les quatre premières lettres du nom du client avec un numéro de comptage allant de 0000 à 9999.

Bref nos factures vont par exemple être stockées comme ceci dans la page de données : ...

...				
BRAS0017	2004-06-21	ESPECE	BRASSAC	MICHEL
BRAS0018	2004-08-11	ESPECE	BRASSAC	MICHEL
BROU0001	2003-07-23	CHEQUE	BROUARD	FREDERIC
BROU0002	2004-12-01	CARTE VISA	BROUARD	FREDERIC
BROU0004	2005-02-16	CHEQUE	BROUARD	FREDERIC
BROU0005	2006-07-20	ESPECE	BROUARD	FREDERIC
BRUN0001	1998-11-21	CHEQUE	BRUN	MARCEL
BRUN0002	1998-08-19	ESPECE	BRUN	MARCEL
BRUN0003	1999-12-21	ESPECE	BRUN	MARCEL
...				

... du fait du tri physique lié au cluster.

Or il se trouve que notre client BRUN est décédé l'an dernier et que sa dernière commande remonte à 2003... De même BRASSAC n'a pas commandé depuis 2004.

En fait, dans cette organisation de stockage, le fait de demander la dernière facture du client BROUARD va faire remonter une page en mémoire contenant une grande quantité de ligne portant sur d'anciennes factures.

Autrement dit, chaque fois que SQL Server va avoir à mettre en cache un lot de facture pour quelques clients encore actifs, ce modèle de données va ramener en mémoire de nombreuses factures obsolètes, en fait les lignes situées physiquement avant et après la seule ligne concernée dans la page de données.

Soyons encore plus précis et voyons ce qui va se passer dans un tel modèle si l'on effectue la requête suivante :

⁽¹⁴⁾Elle est maintenant de 2 Go, mais toujours au travers de pages de 8 Ko

```
SELECT *
FROM T_FACTURE
WHERE DATE_FACTURE BETWEEN DATEADD(DAY, -30, CURRENT_TIMESTAMP)
AND CURRENT_TIMESTAMP
```

Qui demande toutes les factures de ces trente derniers jours...

Imaginons que la page contienne 250 lignes de facture en moyenne et que seul 3 factures par page sont réellement concernées. On considère en outre qu'il n'y a qu'une centaine de facture qui ont été émises ces 30 derniers jours...

Le calcul montre qu'il faudra monter en mémoire 34 pages de 8 Ko, soit 272 Ko. Ce n'est certes pas grand chose, mais comparé à une organisation de cluster dans laquelle on aurait pris pour composante de la clef la date de facturation ou encore si la clef avait été constituée d'un auto incrément, on serait passé au plus à la lecture de deux pages, soit une optimisation de la mémoire d'au moins 17 fois (1700 %...).

Voici comment on peut engorger un cache de données avec un modèle mal conçu.

Dans ce cas de figure, pour bénéficier d'une telle optimisation, il faudrait soit se débarrasser du mode cluster au profit du *heap* pour l'index sous jacent à la clef (si l'on veut conserver impérativement la composition de cette clef), sinon concevoir une clef plus astucieuse.

Partant d'ailleurs du même principe que le LRU, *les données les plus récentes* étant les plus utilisées, MS SQL Server préconise l'utilisation d'une clef auto incrémentée qui garantit un bon séquençement dans le temps. On voit tout de suite que dans un tel cas, les factures les plus récentes, donc les plus scrutées, sont situées toujours en fin de table, donc regroupées dans un minimum de pages, en fait les dernières de la table.

En ce sens le choix de la clef et particulièrement la préconisation d'utiliser l'auto incrément minimise le cache des données.

Troisième commandement : une bonne structure des clefs primaires et plus généralement, des index de la base, participe généralement à une économie drastique du cache des données.

II-D - Les écritures

A intervalles réguliers, établi par défaut à une minute, un processus délicieusement nommé " écrivain paresseux ⁽¹⁸⁾ " scrute les pages modifiées afin de les écrire physiquement dans le fichier des données. Bien entendu, l'intelligence de sa paresse, lui fait regrouper les pages à écrire en tenant compte de leur proximité au sein du fichier de données, par un savant calcul de contigüité. Mais comme cela ne suffit pas toujours, on peut même le réveiller si on le souhaite à l'aide de la commande CHECKPOINT qui force les écritures.

D'un autre côté le journal des transactions qui est sensé relater la vie de la base de données, s'oblige, pour cause d'acidité ⁽¹⁹⁾, à tracer toutes les informations sur les demandes de lectures et d'écritures faites dans la base ainsi que toutes les données se référant à ces demandes. Non seulement cela peut représenter un volume de données très important, mais lorsque les transactions durent longtemps - ce qui veut dire qu'elle concerne un fort volume de données - la mise à jour qui va en résulter, risque de saturer la mémoire cache !

Il n'est pas rare de voir un développeur se persuader que l'insertion d'un fichier de données dans une table, dans le cas d'une opération d'import de données, doit s'effectuer dans le périmètre d'une transaction. Le problème, c'est que dès que le fichier d'import devient conséquent, la transaction s'allonge et sature le journal comme la mémoire...

⁽¹⁸⁾ lazywriter

⁽¹⁹⁾ Il ne s'agit pas là de chimie mais bien de l'alchimie des bases de données, l'acronyme ACID signifiant Atomicité, Cohérence, Isolation et Durabilité, toutes caractéristiques obligatoires inhérentes à un véritable SGBDR.

Une idée plus astucieuse dans ce cas aurait été de découper ce fichier en petits lots et de gérer la mise à disposition des informations soit par de multiples petites transactions si le besoin impérieux d'une telle sécurité s'impose, soit par une logique fonctionnelle reposant sur une donnée la plus basique possible.

De fait, dans une grande transaction impactant la mise à jour de nombreuses lignes, une grande partie du cache est occupé par des données à écrire réduisant ainsi l'espace de celles à lire avant que la fatidique minute du *lazywriter* ne réveille cet écrivain afin qu'il entame sa scripturale tâche. A moins qu'on ne le force à agir en le frappant d'un vigoureux CHECKPOINT !

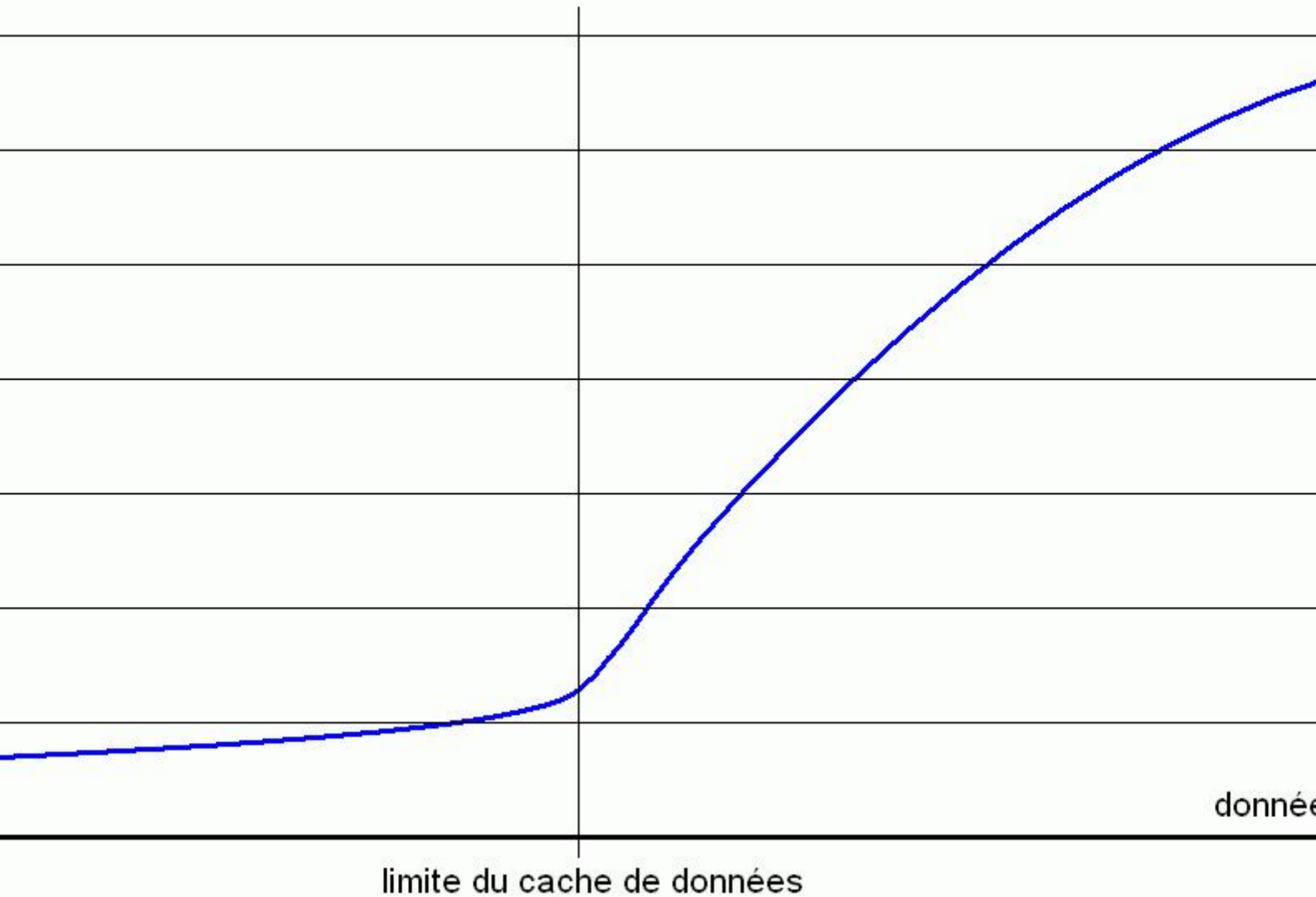
On en conclut donc notre quatrième et dernier commandement : plus les transactions sont petites, plus elles sont rapidement écrites, moins le cache est mobilisé.

II-E - Conclusion

Au début de cet article, je vous ai parlé de *l'effet cache* : tant que les données restent modestes, les performances sont excellentes. Dès que le volume s'accroît de manière importante, il se peut qu'à un moment donné les performances chutent de manière spectaculaire. Pourquoi ? Tout simplement parce que le volume des données à traiter dépasse sensiblement la capacité du cache. Dès lors, les appels au disque commencent à devenir fréquents, voir continus. Or entre le disque et la RAM nous avons vu que la différence de temps d'accès était dans un rapport d'au moins 1 à 1000 voire plus.

La courbe ci dessous est une tentative d'explication du phénomène. Veuillez noter que l'axe temps de réponse est logarithmique. Ce qui signifie qu'entre deux traits il existe un rapport de 1 à 10. Sur l'axe des données à traiter, tant que le cache fait son effet le temps de réponse varie peu. En revanche dès que le cache fait défaut, il faut lire le disque et mettre en cache, c'est là que le temps d'accès au disque, incommensurablement plus long que l'accès mémoire, se fait sentir et que le temps de réponse global chute de manière dramatique.

ponse



On pourrait dire encore beaucoup de choses sur le sujet et je vais sans doute être violement critiqué pour mon simplisme. Mais je préfère le didactisme à l'absolu vérité, la pédagogie à la technicité et vous engage à approfondir le sujet si le cœur vous en dit avec les quelques références que vous trouverez en fin d'article.

Dans les prochains mois, nous continuerons notre introspection de l'optimisation des bases de données sous MS SQL Server par les articles suivants : 2 - le matériel, 3 - le modèle de données, 4 - l'écriture des programmes et des requêtes, 5 - L'exploitation.

D'ici là n'hésitez pas à me joindre par courriel ⁽²⁰⁾ à SQLpro@SQLspot.com

(20) Pour les anglophones, email ! ;-)