Optimisation des bases de données MS SQL Server

Quatrième partie : Les requêtes

En théorie, quelque soit la façon d'écrire une requête, le SGBDR doit être capable de trouver le moyen le plus efficace de traiter la demande, grâce à l'optimiseur. Mais parce qu'il y a loin de la théorie à la pratique, différentes écritures et différents styles de résolution, aliés à la qualité de l'indexation peuvent donner des temps d'exécution très variés.
C'est pourquoi la maîtrise de la conception de requêtes est un des points clefs de la performance d'une base de données bien conçue.
Voyons quels en sont les principes basiques.

Par Frédéric Brouard - MVP SQL Server
Expert SQL et SGBDR, Auteur de :

- SQL, Développement, Campus Press 2001
- SQL, collection Synthex, Pearson Education 2005, co écrit avec Christian Soutou
- http://sqlpro.developpez.com (site de ressources sur le langage SQL et les SGBDR)
- Enseigant aux Arts & Métiers et à l'ISEN Toulon

Dans un cours que je donne aux Arts & Métiers, je montre comment une bonne indexation alliée à la qualité de l'écriture des requêtes, peut faire varier dans une proportion de plus de 300 le temps d'exécution d'une

requête. Je commence l'exercice par une demande simple : écrire une requête SQL permettant de répondre à une question basique, mais en prenant soin d'exprimer différentes solutions, même les plus bizarres. Outre la solution simpliste, certains y arrivent par une union, d'autres avec des sous requêtes, d'autres encore avec des jeux de CASE... L'exécution brute de toutes ces solutions, donne un coût d'exécution allant du simple au triple. La pose d'un premier index trivial ne donne rien, par manque de sélectivité. Après élimination de cette première tentative, la pose d'un nouvel index montre que les requêtes les plus coûteuses au départ deviennent les plus rapide, alors que celles qui étaient les moins coûteuses à l'origine n'ont rien gagné. Un nouvel essai d'indexation remet toutes les requêtes au même rang qu'au départ, chacune gagnant dans la même proportion et plus que l'indexation précédente. Enfin, la dernière tentative d'indexation étant la bonne, toutes les requêtes bénéficient d'un gain important, mais certaines bien plus que d'autres.
Le clou est enfoncé lorsque l'on choisit de dénormaliser avec une vue. Là le gain devient gigantesque. Il est voisin de 13 000. Mais il ne prend pas en compte l'effort supplémentaire à faire pour les mises à jours (INSERT, UPDATE, DELETE...).
Cet exercice nous apprend trois choses :

- différentes écritures d'une même requête ne donnerons pas forcément les mêmes performances, bien que dans l'absolue ce devrait être le cas[1]
- rien ne sert de poser un index s'il ne sert pas la requête
- une même requête écrite de différentes manières ne bénéficiera pas des mêmes gains lorsque l'on pose un index

J'ajouterais que tout ceci évolue en fonction de la volumétrie des données et des données mêmes ! Tant est si bien qu'il est difficile de trouver de prime abord ce qu'est l'écriture d'une bonne requête.
Affirmons cependant qu'une bonne requête est une requête qui sait tirer partie du moteur de requête pour le forcer à calculer un plan de requête dont les étapes sont les plus courtes à traiter.

Plan de requête et optimiseur

Mais au fait, qu'est-ce qu'un plan de requête ? Quel est sont importance ? Osons le dire immédiatement. Ce qui nous intéresse n'est pas tant le plan de requête que ce qu'il y a derrière : la consommation des ressources. Et là il faut bien dire que l'habillage graphique de Microsoft est plus un miroir aux alouettes qu'un outil de premier ordre... En effet beaucoup de développeurs sont aveuglés par les jolies icônes et ne réalise pas que ce qui compte n'est pas tant le dessin de ces dernières mais plus le dessein qui se trame derrière !
Ainsi la plupart des développeurs sont persuadés qu'entre deux plans de requête, le meilleur est celui qui a le moins d'étapes... Grave erreur !

[1] En fait nous croyons souvent et naïvement que certaines écritures de requêtes sont identiques. Mais nous oublions souvent l'influence du marqueur NULL, dont le comportement particulier dans différents prédicats oblige le moteur SQL à des constructions parfois fort différentes.

Commençons donc par définir ce qu'est un plan de requête : le plan de requête est la décomposition en étapes élémentaires du traitement de données nécessaire à la résolution de la requête. Les plans de requêtes se présentent sous la forme d'arbres dont les feuilles sont différents points de départs indépendants et la racine le point final de collecte des lignes à afficher.
En premier lieu ce sont les feuilles de l'arbre « plan de requête » auxquelles il convient de porter toute son attention. Comme ces dernières sont des points de départ, et qu'un point de départ consiste à lire des données, plus on lira vite les données et mieux ce sera !
En second lieu il faut s'intéresser aux algorithmes de jointures. Comme il faut bien en faire, plus ces algorithmes seront efficaces en fonction des données à mettre en relation et mieux cela sera.
Enfin il faut se préoccuper des autres étapes : groupements, tris et autres algorithmes sont-ils bien nécessaires ou pas ?

deux requêtes identiques écrites dans deux casses différentes sont considérées comme deux entités distinctes et ne bénéficie pas de la mise en cache

Mais à y voir de plus près, la première mesure de l'efficacité d'un plan par rapport à un autre est en fait la consommation des données. Moins le moteur lira de pages de données pour un même traitement, plus la requête sera efficace et rapide. En second lieu et si deux requêtes arrivent ex æquo en consommation de pages, il faudra préciser la mesure par la consommation de CPU. Ces deux mesures sont simples et facile à mettre en œuvre dans l'analyseur de requêtes.[2]

Finalement la grande question est : comment le moteur SQL calcule t-il le plan de requête ? C'est là, nous l'allons voir, du grand Art…

Confronté à une requête qu'on lui lance, le moteur relationnel tente de savoir s'il n'a pas déjà rencontré cette forme de requête, aux paramètres près. Pour cela il transforme chaque donnée de la requête en paramètre. Puis il compare la chaîne de caractères ainsi obtenue avec les autres éléments qu'il conserve en mémoire. S'il trouve une correspondance absolue alors il peut se resservir du plan de requête précédemment établis. Mais, même si le plan a déjà été établis, et a donc été repris depuis le cache des procédures, il est possible que ce dernier évolue. En effet des changements de dernière minute dans le plan de requête, peuvent intervenir dans différentes conditions : valeurs très déséquilibrée de paramètres, volumétrie de données ayant subit un grand changement depuis la dernière utilisation du plan, obsolescence des statistiques disponible sur les index…
Il n'y a pas longtemps, un internaute postait dans un forum SQL Server sa stupéfaction de voir sa requête passer de quelques minutes à plusieurs

[2] SET STATISTICS IO ON pour la consommation des pages pour une requête et SET STATISTICS TIME ON pour la consommation CPU

heures et ne trouvait aucune explication… Les statistiques de ses index ayant été calculées il y a fort longtemps, SQL Server avait considéré – sans doute à tort – qu'on ne pouvait plus s'y fier, et décidé de modifier le plan afin que les accès aux données lisent séquentiellement toutes les données…

La comparaison d'une requête avec celles qui ont déjà été mise en cache s'effectue toujours sur le binaire du motif. Il en résulte que deux requêtes identiques écrites dans deux casses différentes seront considérées comme deux entités distinctes. De même si l'on inverse les membres d'un prédicat, par exemple en mettant tantôt la colonne en premier, tantôt la valeur. Dans ces deux cas, cela provoquera deux mises en caches, deux plans de requêtes… Avec la conséquence que le cache se saturera plus vite, et comme il sera plus encombré, il vieillira plus vite et ces éléments sortirons donc du cache plus rapidement en vertu de la fameuse Loi LRU…

Maintenant que se passe t-il si la requête n'est pas trouvée dans le cache des procédures ? Il convient au moteur relationnel de calculer un plan de requête et le meilleur possible !
Pour cela la technique est simple, mais très raffinée. Le moteur est censé évaluer toutes les combinaisons possibles pour lire, traiter les données et enchaîner les différentes étapes aboutissant à la résolution de la requête. Mais comme la combinatoire peut rapidement devenir gigantesque, le programme qui calcule le plan de requête doit faire des impasses sur des solutions à priori peu rentables. Pour cela il commence par évaluer la façon d'attaquer la lecture des données : peut-il se servir d'un index ou doit-il lire la table ? L'utilisation de l'index peut-elle mettre en œuvre un algorithme de recherche ou est-il condamné à lire séquentiellement toutes les données ? La réponse à ses interrogations se trouve dans les statistiques afférentes aux index, car derrière chaque index il y a des statistiques qui permettent d'estimer le volume de données qui va être manipulé. Cela se calcule en fonction de la valeur du paramètre, et de la dispersion des données dans l'index ou la table, le but étant de lire le moins possible de données dès le départ.
A présent que l'on connaît la façon d'attaquer les données, on peut ensuite estimer la façon de les traiter. Ainsi une jointure utilisera tel ou tel algorithme (tri fusion, boucle imbriquées, clef de hachage…) en fonction des volumes respectifs des données à mettre en relation et de la cardinalité[3] de la jointure.
Même avec toutes ces impasses, le temps de calcul du plan de requête peut vite devenir très long et par exemple conduire à une situation absurde ou le calcul du plan optimal plus de temps que le traitement de la requête. C'est pourquoi l'optimiseur, car c'est son nom, utilise un algorithme de backtracking et une fenêtre de temps.
Le backtracking est un moyen efficace d'explorer un arbre de combinaison probable en valuant progressivement les branches à chaque birfucation. Cela permet de s'arrêter dès que la descente dans une branche revêt un coup supérieur au meilleur plan déjà calculé.

[3] Rappelons que la cardinalité n'est autre que le nombre de liens entre deux éléments : un à un, un à plusieurs, plusieurs à plusieurs.

La fenêtre de temps, car il serait absurde que l'établissement du plan ne dure trop longtemps.
Ainsi l'optimiseur ne calcule pas systématiquement le plan optimal, mais un plan de requête qu'il a jugé le moins coûteux compte tenu de différentes contraintes préalables.

Bien entendu on peut forcer l'optimiseur à se comporter différemment. On peut aussi l'aider on lui donnant quelques impératifs… Mais cette façon de faire n'est généralement pas la plus intelligente, car elle enferme l'optimiseur dans un carcan qui peut se révéler tôt ou tard plus handicapant qu'avantageux. Non, il existe d'autres moyens plus efficace encore et ces autres moyens sont au nombre de deux : l'écriture de la requête et l'implantation judicieuse des index…

L'écriture des requêtes

Il est assez surprenant de constater avec quelle désinvolture certains développeurs dédaigne l'écriture des requêtes SQL. Il y a sans doute une explication à cela. Dans des temps immémoriaux, les pionniers de l'informatique furent payés à la ligne de code. Plus ils tartinaient les listings, plus ils étaient payés. La mesure de la productivité était basée sur la faconde et certains langages permettaient de bien gagner sa vie parce que très verbeux, comme le fameux CoBOL… Il est probable, tel un atavisme, que ce travers soit resté de nos jours. Pondre un programme de 2 000 lignes d'un code, même modernisé semble plus productif à bien des actuels tâcherons que d'optimiser une seule requête SQL sur la même période de temps et alors qu'elle remplit le même but !
« je ne sais pas si mon patron apprécierait que je passe une journée sur la même requête » me disent certains, alors que j'intervient au prix fort pour rectifier une sauce qu'ils auraient pu réussir du premier coup à moindre frais…

Passé le cap de la culture, il y a aussi celui de la formation. Bien des développeurs sont capable d'écrire des algorithmes de gestion d'arbres sous forme récursive et avec un langage des plus moderne et ignorent encore que les jointures entre les tables s'effectuent dans une clause JOIN dont la syntaxe remonte à 1992 !
Pauvres universités, pauvres écoles d'ingénieurs où l'on concède à l'apprentissage des bases de données quelques maigres heures dans un cursus de quelques années… Bref, il faut bien le dire, à part quelques passionnés, peu d'informaticiens savent écrire de vraies requêtes. Alors de là à parler de techniques d'écriture…

Mais c'est pourtant ce que nous allons faire. Citons en trois : l'approche ensembliste, la résolution à petits pas et la technique d'ajout d'information.

L'approche ensembliste consiste à considérer le problème sous l'angle des ensembles, c'est-à-dire avec les outils des mathématiques modernes que

sont les diagrammes de Venn. Vous savez, les fameuses « patates » ! Cette technique consiste à penser ensemble et non ligne à ligne, à considérer le problème de manière globale et non comme à une série d'éléments ordonnés. Je prend souvent l'exemple introductif suivant : « une table est un sac de bille. Pouvez vous en extraire la dernière ? » La réponse fuse : impossible ! Mais si les billes ont été horodatées, alors la solution du problème saute aux yeux !
Considérez dès lors les ensembles et leurs opérations en regard des tables et des opérateurs de l'algèbre relationnel. Tiens… rien ne ressemble plus à une intersection qu'une jointure…
Bref commencez par dessiner les ensembles en jeu, faites les s'interpénétrer, s'exclure, se compléter et alors au beau milieu du graphe, la solution apparaîtra, bête, évidente… Mais il fallait y penser !

L'approche par petit pas consiste à partir d'une requête simple qui approche vaguement la solution puis de la compléter par petites touches. Là où les choses s'enveniment, c'est lorsque le code de la requête grossit à cause de toutes les jointures, des différents prédicats des filtres WHERE et HAVING et de clauses plus barbares telles que les GROUP BY… Au fur et à mesure il devient difficile d'y retrouver ses petits. Mais que cela ne tienne. Transformez cette première étape en un objet mille fois plus simple : une vue par exemple ! Dès lors vous pourrez à nouveau la farcir de quelques filtres et jointures supplémentaires, la placer en sous requête ou lui intimer d'en user… C'est bien ce que vous faites en programmations objet ? Non ? En emboîtant vos objets dans d'autres et en nommant avec prétention le tout sous le nom générique de « classes » !
Qu'à cela ne tienne, utilisez les vues comme les expressions de table afin de parvenir au but. Si l'ensemble vous déplait, alors réimbriquez la requête composant chaque vue dans celle qui abrite toute ces vues et peut être aurez vous la chance de trouver au final quelques simplifications.
De plus, depuis l'apparition des expressions de tables, les fameuses CTE, l'imbrication des requêtes les unes dans les autres est encore facilité par la volatilité et la souplesse inhérente à cette construction. Usez-en, abusez-en !

L'approche par ajout d'information, part du constat bien simple que SQL ne saurait inventer des données qui n'existent pas. Combien de fois ais-je vu des solutions alambiquées et foireuses, destinées à livrer sous forme de tables, des données absentes de la base, alors qu'il aurait été si simple d'ajouter ces données : ajouter une table, des tables, des vues afin de résoudre le problème élégamment et non dans une logorrhée quéristique !
Je n'ai qu'un seul exemple à donner : imaginons qu'il faille imprimer un nombre variable d'étiquettes portant le nom du patient. Par exemple suivant la table ci-dessous, de nom *impressions* :

Nom	Etiquettes
MARTIN	3
DURAND	2
MULLER	5
LEGRAND	1

C'est à dire imprimer 3 étiquettes pourn martin, 2 pour Durand, etc…

Autrement dit, comment générer la table suivante en sortie :

Nom
MARTIN
MARTIN
MARTIN
DURAND
DURAND
MULLER
MULLER
MULLER
MULLER
MULLER
LEGRAND

Peu de développeurs parviennent à résoudre le problème en une seule requête. Tout simplement parce que dans le modèle de données de la base manque l'élément clef : une table des nombres, une table d'une seule colonne avec tous les nombres depuis 0 jusqu'à… suffisamment !
Ainsi en rajoutant une telle table :

```
CREATE TABLE T_NUM (I INT)
INSERT INTO T_NUM VALUES (0)
INSERT INTO T_NUM VALUES (1)
INSERT INTO T_NUM VALUES (2)
INSERT INTO T_NUM VALUES (3)
INSERT INTO T_NUM VALUES (4)
INSERT INTO T_NUM VALUES (5)
…
```

La solution triviale apparaît d'une grande simplicité :

```
SELECT nom
FROM   impressions
       INNER JOIN T_NUM
             ON Etiquettes < I
```

C'est ainsi que je voit dans la plupart des modèles de données, dans la plupart des bases que j'audite, cette absence navrante de table de nombre, de table de dates, etc.
Une explication m'est venue un jour d'un technicien : « *mais nous n'avons pas le droit de rien rajouter à la base ! Vous vous rendrez compte ce qui risque de se passer si l'on y ajoute quelque chose ?* » Aussitôt je l'ai regardé d'un air navré : « *relisez donc les règles de Codd mon ami… On y affirme que l'ajout d'un objet à la base ne perturbe en rien le fonctionnement de celle-ci. Et quand bien même vous n'auriez pas le droit moral de faire cela, que diable ! Créez vos tables dans une autre base, une base de travail que vous utiliserez pour l'occasion. Ce sera en outre portable et sans danger !* »

Utilisation des index

C'est d'abord la qualité de l'indexation qui fait la rapidité d'exécution des requêtes. Il faut donc bien comprendre ce qu'est un index, à quoi il sert et comment il est structuré.

Commençons par une métaphore. Imaginez que je vous invite à dîner ce soir chez moi et, en vous quittant, je vous donne rendez-vous à 20 heures et vous signale habiter aux Champs Élysés à Paris[4]. Comme je ne vous ais pas précisé l'immeuble, il vous sera difficile de me trouver. Le jeu pour vous, va consister à rentrer dans tous les halls, frapper à toutes les portes, palabrer quelques instants afin de trouver le bon appartement.
Vous venez de comprendre à quoi sert un index ! Les numéros des immeubles dans les rues, parce que les immeubles y ont été ordonnés dans le sens croissant[5], constituent un excellent exemple de ce qu'est un index.
Avant que n'entre en vigueur le téléphone[6] automatique, rentrer en communication avec un interlocuteur nécessitait de demander à l'opératrice le nom de la personne ou de l'entreprise et la ville de domiciliation. Petit à petit, avec les cas d'homonymie dans les grandes villes, on se rendit compte que cela n'était pas pratique et on commença d'attribuer des numéros aux abonnés, tout en maintenant la sectorisation par ville. C'est ainsi que Fernand Raynaud se rendit célèbre avec son "22 à Asnières"... Puis on enrichit le numéro d'une appelation de standard. C'est ainsi que le commissaire Maigret était joignable à BREtagne 38 96 (BREtagne était le nom du standard situé rue de Bretagne et dont on composait les 3 premières lettres à titre d'indicatif. Il desservait quelques dizaines d'artères parisiennes, dont le boulevard Richard Lenoir ou habite Jules Maigret). Puis, le 25 octobre 1985, on rajouta un chiffre en tête. Pour paris, ce fut le 4. Enfin, le 18 octobre 1996 on rajoute un nouvel indicatif de zone : 01 pour Paris et sa région, 02 à 05 pour les différents secteurs d'une France coupé en 4. Enfin, si vous téléphonez depuis l'étranger l'indicatif de pays est le 33 ! Résumons... Par un simple n° de téléphone, on sait à quelques pâtés de maisons, ou vous habitez. Dans mon village les autochtones se communiquent leurs coordonnées téléphoniques avec les simples 4 derniers chiffres, sachant que tout le monde est à la même enseigne pour le reste.
N'est-ce pas là un maginfique exemple d'index ? Un tel numéro sert à "aiguiller" la recherche du destinataire...

C'est donc cela un index : une information redondante dont le but est d'accélérer les recherches. Cependant il y a quelques limites à l'index...

Comme un index constitue une information redondante et ordonnée, la recherche dans un index est rapide, si ce que l'on cherche va dans le sens du tri. Un index ne sert donc à rien dans certains cas. Prenons par exemple un numéro de téléphone et constatons que sa partie la plus intéressante n'est pas le début, mais la fin. En effet, vous français qui me lisez en ce moment, voici trois n° de téléphone identiques :

+ 33 1 42 92 81 00
00 33 1 42 92 81 00

[4] Ne m'y cherchez pas, je déteste les Champs Élysés. Ce doit d'ailleurs être une artère très boueuse vu le nombre incroyable de 4x4 qui s'y pavanent !
[5] A mon avis, ce doit être l'inverse, car déplacer un immeuble est plus couteux qu'une plaque numérotés... Mais il s'agit là d'un propos en avance sur notre prochain article consacré à la maintenance des performances...
[6] Lorsqu'on lui présenta cette terrible invention, Clémenceau dit du téléphone "*Quoi ? On vous sonne comme un laquais !*"

01 42 92 81 00

Or la partie significative de ces n° est la fin et non le début.
Imaginons maitenant que vous cherchez à savoir quel est l'interlocuteur qui vous a laissé un numéro de téléphone tel que 01 42 92 81 00. En général et compte tenu des différentes écritures possible (avec ou sans suffixe de pays) vous allez construire votre prédicat de recherche avec un LIKE comme ceci :

```
WHERE NUM_TEL LIKE '%1 42 92 81 00'
```

Malheureusement, même en plaçant un index sur ce type de colonne, vous n'aurez jamais aucune chance de trouver rapidement votre correspondant. En effet, les données de l'index étant trié par rapport aux premièrs caractères composant la chaîne, il faudra balayer toutes les données pour en trouver la correspondance.
Vous venez de toucher du doigt une notion importante, ce que les anglos saxons nomment "la sargeabilité" (en anglais *sargeable*[7] : Search ARGument ABLE). Il ne suffit pas de placer un index, encore faut-il que l'expression de filtrage puisse l'utiliser...

Dans bien des cas on peut rendre la requête "sargeable" avec quelques petites transformation, à la limite de la dénormalisation. En effet il suffit de stocker le n° de téléphone en inversant la chaîne de caractères lors des phases d'INSERT et d'UPDATE. Dès lors, la recherche devient rapide avec le truc suivant :

```
WHERE NUM_LET LIKE '001829241%'
```

De la même manière un prédicat contenant un opérateur logique OU n'est pas sargeable. Mais vous pouvez rendre sargeable la requête en la décomposant en deux requêtes réunies par l'opérateur ensembliste UNION.
Exemple :

```
SELECT ...
FROM   CLIENT AS C
       INNER JOIN EMPLOYE AS E
             ON C.CLI_ID = E.CLI_ID
WHERE  CLI_NOM = 'DUPONT'
   OR  EMP_NOM = 'DUPONT'
```

N'est pas sargeable, même si des index ont été placés sur les noms des clients et des employés...
En revanche, sa petite soeur est parfaitement sargeable :

```
SELECT ...
FROM   CLIENT
WHERE  CLI_NOM = 'DUPONT'
UNION
SELECT ...
FROM   EMPLOYE
WHERE  EMP_NOM = 'DUPONT'
```

Ne sont généralement pas sargeable :

- les LIKE en début de mot (sauf à inverser)

[7] Devrions nous dire "cherchable ?"

- les LIKE en plein mot (sauf construction particulière[8])
- les expressions logiques contenant des OR
- les expressions calculées dont la colonne faite partie de l'expression
- les colonnes qui sont utilisées par une fonction
- des fourchettes d'exclusion
- le NOT
- …

Le pire vient sans doute d'expressions telles que celle-ci :

```
WHERE COL1 = COL2
```

A priori une expression sargeable… Et bien dans certains cas… Non ! En effet si les types SQL utilisés par COL1 et COl2 ne sont pas strictement les mêmes, le moteur SQL devra transformer les données d'une colonne dans l'autre à l'aide d'une fonction et de ce fait l'index ne pourra être utilisé !
Par exemple si COL1 est indexé et de type INTEGER et COL2 de type FLOAT, alors l'index ne sera pas utilisé car l'expression s'écrit en fait :

```
WHERE CAST(COL1 AS FLOAT) = COL2
```

Attention aussi aux index composés de plusieurs colonnes. Seule la vectorisation des colonnes[9] est sargeable. De plus la collecte des statistiques ne se fait que sur la première colonne. Veillez donc à ce qu'elle soit la plus discriminante, la plus sélective possible.
Par exemple l'index suivant :

```
CREATE INDEX X_PRS ON T_PERSONNE (PRS_CIVILITE, PRS_PRENOM, PRS_NOM)
```

…ne sera jamais utilisé dans les clauses suivantes :

```
WHERE PRS_PRENOM = 'Alain'
WHERE PRS_NOM = 'DUPONT'
WHERE PRS_PRENOM = 'Alain' AND PRS_NOM = 'Dupont'
```

Il sera faiblement exploité si le prédicat de recherche contient la colonne PRS_CIVILITE, car les valeurs de cette colonne ont une distibution très faible (M, Mme, Mlle).
A l'évidence, cet index aurait dû être constitué comme ceci :

```
CREATE INDEX X_PRS ON T_PERSONNE (PRS_NOM, PRS_PRENOM, PRS_CIVILITE)
```

…au moins les deux dernières requêtes en auraient bénéficiées.

Depuis la version 2005, SQL Server permet de rajouter des colonnes d'information à l'index. Ces colonnes ne font pas partie de l'index (et ne sont donc pas triées avec les données de l'index) mais peuvent servir à mieux couvrir une requête. Un index est dit couvrant si sa seule lecture permet de traiter entierement la requête.
A titre d'exemple intéressons nous à la table suivante :

```
CREATE TABLE T_EMPLOYE_EMP
(EMP_ID          INT NOT NULL IDENTITY PRIMARY KEY,
```

[8] Nous nous sommes amusés à rendre cette expression sargeable pour des besoins de manipulation de grands textes, mais cela nécessite une construction particulière à base d'indexation textuelle et d'index rotatifs.
[9] Cette vectorisation des colonnes fait référence à la technique du Row Value Constructor… En effet dans une liste de données dont chaque élément précise le précédent, la recherche d'un élément situé au milieu du vecteur n'a pas de sens. Par exemple dans le cadre d'une donnée temporelle dont la précision va de l'heure à la seconde, la recherche d'un minutage précis n'a pas de sens…

```
EMP_NOM         CHAR(32) NOT NULL,
EMP_PRENOM      VARCHAR(25),
EMP_MATRICULE   CHAR(8) NOT NULL UNIQUE,
EMP_CIVILITE    CHAR(4),
EMP_DATE_NAISS  DATETIME)
```

Cette table contient en fait deux index, et tous deux ont été créés de manière sous jacente aux contraintes.
Le premier index est de type "cluster". C'est celui de la clef primaire. Il s'agit en fait de la table toute entière triée sur la clef. SQL Server par défaut créé des index cluster pour chaque PRIMARY KEY.
Le second est un index secondaire, c'est à dire qu'il consiste en une copie triée des données indexées avec en sus la référence à la ligne originale, c'est à dire la clef de la table (à défaut de clef, la référence à la ligne de table est composée de trois éléments : le n° du fichier de stockage des données, le n° de la page dans le fichier, la position de la ligne dans la page...).
Créons maintenant un index de toute pièce comme ceci :

```
CREATE X_EMP1 ON T_EMPLOYE_EMP (EMP_NOM, EMP_PRENOM)
```

La requête suivante :

```
SELECT EMP_ID, EMP_PRENOM, EMP_NOM
FROM   T_EMPLOYE_EMP
WHERE  EMP_NOM LIKE 'M%'
```

... est non seulement sargeable, mais l'index utilisé est couvrant. En effet il contient bien le nom, mais aussi le prenom et la référence à la ligne qui n'est autre que la clef, en l'occurrence la colonne EMP_ID. Point n'est alors besoin pour le moteur SQL d'aller lire en sus des données complémentaire dans la table.
Mais si vous ne cherchez jamais sur la combinaison nom + prénom, alors préférez une couverture plus douce en utilisant l'option INCLUDE. Cela induira moins d'effort au moteur SQL lors des mises à jours :

```
CREATE X_EMP2 ON T_EMPLOYE_EMP (EMP_NOM) INCLUDE (EMP_PRENOM)
```

Avec cette nouvelle syntaxe, la colonne EMP_PRENOM sera stockée de manière purement informationnelle et conduira au même effet à moindre coûts !

Je ne vous ais pas encore tout dit sur l'indexation. Il reste encore beaucoup de choses à évoquer, comme les index sur les colonnes calculées ou les vues indexées qui sont en fait une forme élégante de dénormalisation.
Il reste aussi beaucoup de choses à élaborer dans les labos de recherche des université et des éditeurs informatiques.

Un peu de code pour terminer

Le code Transact SQL des routines est aussi un point dur de la perte de vélocité des traitements.

Pour les fonctions utilisateurs (UDF), évitez les fonctions tables. Les tables ainsi produites ne peuvent être optimisées..

Parce qu'elles sont susceptible d'être lancées sur des dizaines de miliers de lignes, le code des fonctions scalaires doit être étuidé à la loupe. On prendra soin par exemple d'éliminer tous les effets de bord avant de commencer à dérouler l'algorithme. Cela permettra de s'affranchir de lancer du code en pure perte.
Démonstration :

```
CREATE FUNCTION F_PIVOT (@data VARCHAR(8000), @pivot int)
RETURNS VARCHAR(8000)
AS
RETURN SUBSTRING(@data, @pivot+1, LEN(@data)-@pivot) +
       SUBSTRING(@data, @pivot, 1) +
       SUBSTRING(@data, 1, @pivot-1)
END
```

Voici une fonction permettant de pivoter deux parties d'une chaîne par rapport à un caractère de position donné. par exemple F_PIVOT('aperçurent', 7) donne 'entraperçu'... Étonnant non ?
Mais cette fonction oblige à dérouler le code lorsque les paramètres sont nuls ou inopérants. Appliquée à des tables composées de milliers de lignes, de telles fonctions perdent du temps, notamment s'il y a un fort taux de NULL, chaînes vides... Voici cette même fonction récrite pour améliorer ses temps de réponse :

```
CREATE  FUNCTION F_PIVOT (@data VARCHAR(8000), @pivot int)
RETURNS VARCHAR(8000)
AS
BEGIN
-- premier effet de bord : un des paramètres est manquant
IF @data IS NULL OR @pivot IS NULL
   RETURN NULL
-- second effet de bord, la chaîne est vide
IF @data = ''
   RETURN ''
-- troisième effet de bord, le pivot est supérieur à la longueur de
chaîne
IF @pivot > LEN(@data)
   RETURN @pivot
-- quatrième effet de bord, le pivot est négatif
IF @pivot <= 0
   RETURN @pivot
-- cas général
RETURN SUBSTRING(@data, @pivot+1, LEN(@data)-@pivot) +
       SUBSTRING(@data, @pivot, 1) +
       SUBSTRING(@data, 1, @pivot-1)
END
GO
```

On pourra prévoir de même pour certaines procédures.

Au fait, pour les procédures : banissez les curseurs. Il est très rare qu'un curseur soit plus rapide qu'une requête... Et dans leurs très grande majorité ces derniers peuvent être remplacés par des requêtes ou un code moins agressif. Un curseur interdit toute optimisation, tant est si bien qu'une requête même complexe va très souvent beaucoup plus vite qu'un curseur de quelques lignes.

Enfin, en matière de transaction, choisissez le bon niveau d'isolation (le plus faible possible) et surtout le code le plus court afin que la durée de la transaction soit la plus petite. Peut être n'avez vous pas besoin de tout transactionner !