Optimisation des bases de données MS SQL Server

Troisième partie : Le modèle de données

Optimiser une base de données simplement par la qualité de son modèle est une chose simple, très efficace et et à coût nul... Voila qui devrait intéresser beaucoup de monde. Or c'est souvent l'inverse qui se passe : le peu d'attention apportée au modèle, le peu de précaution dans le choix des types de données, le peu de respect des formes normales sont autant d'icebergs qui ne vont pointer leurs nez qu'au moment ou la base commencera d'être volumineuse. Ces erreurs du modèle, aux couteuses contre performances ne montreront des effets pervers qu'au moment ou l'inertie des données sera telle que tout remaniement de la base deviendra une entreprise risquée, complexe et douloureuse.
*Il ne faut pas oublier que les SGBDR ont été conçu pour manipuler des relations. Croire qu'un SGBDR agit comme au bon vieux temps des applications consommatrices de fichiers est un non sens absolu. La lecture d'"**enregistrements**" au sein de "**fichiers**"n'a pas de sens dans un SGBDR. Et la transposition d'un modèle de données à base de fichiers en tables dans une base de données à raison d'une table pour un fichier est un échec assuré dont beaucoup d'éditeurs de solutions informatiques ont fait les frais.*
Ce nouvel article a donc pour but de vous faire comprendre ce que sont les données, les types de données et la

modélisation dans la perspective d'optimisation d'une base et donc d'un serveur.

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

Toute application avec une forte implication de SGBDR commence par une modélisation des données. La qualité d'un modèle de données, ne se fera sentir que lorsque ce dernier sera mis à l'épreuve du feu, qui dans l'univers des SGBDR consiste à farcir ses tables qu'une quantité phénoménale de données et jouer les requêtes les plus fréquentes afin d'en mesurer les temps de réponse. Or cette phase est rarement entreprise en test. Elle l'est généralement en production. C'est là qu'est l'os, hélas[1], car il est déjà trop tard !
Lorsqu'un modèle de données est établi, et que le poids du volume des données se fait sentir, alors tenter de le remodéliser pour gagner des performances est un chalenge difficile : les évolutions du schéma conduisent à des migrations de données importantes (donc risquées) et des modifications d'interfaces conséquentes (donc du code à récrire). Lorsqu'il s'agit d'une base de données volumineuse, l'inertie des données peut être telle que l'alternative est s'adapter avec un coût de modification élevé ou mourrir.
C'est pourquoi un modèle de données bâclé présente la particularité d'avoir un coût très élevé lorsqu'il doit être rectifié, alors qu'un modèle peaufiné présente un coût quasi nul si l'on utilise l'outil adéquat et l'homme d'expérience.
Malheureusement, les français ont beau avoir inventé une méthode de modélisation d'une grande simplicité (MERISE[2]) il n'en demeure pas moins

[1] J'avoue avoir emprunté cette tournure... La phrase authentique étant "*ce navire n'a pas d'hélice, hélas, c'est là qu'est l'os*" dans La Grande Vadrouille, film de Gérard Oury...
[2] On l'a beaucoup oublié mais la modélisation des données n'est qu'une modeste partie de la méthode MERISE, qui commence par la définition du schéma directeur informatique, à savoir QUI, QUE, QUOI COMMENT informatiser et à quelles échéances...

que peu d'informaticiens savent modéliser les données de manière intelligente.
Bref, ce sont de ces écueils que je veux aujourd'hui vous entretenir, et pour cela, j'ai découpé en différentes parties le présent article. La première traite des types de données, la seconde des clefs, la troisième des tables et la quatrièmes de la normalisation.

De l'influence du type de données

Lorsque j'aborde les cours sur le langage SQL et que nous en sommes aux types de données, je pose souvent l'innocente question "*quel intérêt y a t-il à avoir un type de chaîne de caractères à longueur variable (VARCHAR) et un autre à longueur fixe (CHAR) ?*" A ce jour, aucun de mes élèves ne m'a jamais donné la réponse attendue... Or l'intérêt, c'est le choix ! Tantôt je vais utiliser tel type pour telle raison et tantôt de sera un autre. Pour bien comprendre les caractéristiques des différents types de données il faut à nouveau se plonger au coeur du fonctionnement d'un SGBDR et tenter d'imaginer comment sont stockées les données dans les pages de la base.
La différence fondamentale entre un CHAR de longeur n et un VARCHAR de même longueur, c'est que les données stockées dans le premier sont complétées à droite par un caractère de remplisage (le blanc, référence 20 en code ASCII), tandis que pour le second on stocke la longueur vraie.
Mais si la ligne de la table comporte plusieurs colonnes de longueur variable, comment savoir ou commence et ou se termine chaque élément de données ? Avec une taille fixe, c'est facile, il suffit de compter les octets et l'on sait tout de suite quel est le découpage des données composant la ligne. Avec des éléments de taille variable, il faut rajouter une information supplémentaire afin d'indiquer ou se termine chaque donnée. En l'occurence dans SQL Server il faut ajouter deux octets pour chaque colonne de taille variable, car ces dernières peuvent aller jusqu'à 8000 octets.
Avantage du VARCHAR, son économie de place lorsque la longueur des données à y stocker peuvent varier dans d'importantes proportions, par exemple comme les lignes d'une adresse. Inconvénient, la nécessité d'un calcul supplémentaire pour trouver l'emplacement de la donnée, ainsi qu'une augmentation de la longeur des lignes pour y stocker l'information du nombre de caractères.
Une des caractéristiques perverse du VARCHAR est sa forte propension à fragmenter les structures de données lors des modifications de valeurs.
Pour obtenir de bonne performance et gagner de la place, les données d'une ligne sont structurées de la manière suivante : toutes les colonnes de taille fixe sont placées au début, celles de taille variables à la fin. En effet, avec cette organisation, seules les colonnes de taille variable ont besoin d'une information supplémentaire pour la longeur. Mais que se passe t-il si une donnée de longeur n est modifiée en une données de longueur $n + i$? Il n'est plus possible de stocker la donnée dans son emplacement original. C'est d'ailleurs le même phénomène qui s'est produit avec moi quand lors de mon premier été d'installation en Provence, je me suis laissé aller à la sieste, la

pétanque, le pastaga[3] et le rosé : à l'autonme, aucun de mes pantalons n'accueillait plus ma bedaine !

le syndrome du pantalon,
ou comment un varchar devenant obèse conduit à des lectures en zig-zag

Il en va de même dans SQL Server, le "pantalon" ligne ne peut plus recevoir la donnée et cette dernière est rejetée sur une autre page, provoquant des lectures en zig-zag. Seul moyen de correction : la réindexation. Seul moyen de l'éviter : adopter du CHAR !

Résumons nous : si le littéral est de longueur faible ou fortement mis à jour ou encore si la donnée est d'une taille proche du maximum, préférez une taille fixe. Dans le cas contraire : grande taille, peu de mise à jour, données de longueur très fluctuante, alors la taille variable vous fera économiser de la place, donc du volume, donc de la mémoire, donc plus de ligne seront traitées directement dans le cache...

Dans un genre différent, on trouve la problématique du stockage des données littérales de type UNICODE en comparaison à l'ASCII. En unicode, chaque caractère est représenté par deux octets permettant ainsi 65 536 symboles, ce qui permet de marier les alphabets latin, grec, cyrillique, hébreu, asiatiques et bien d'autres encores. En ASCII 8[4] bits, il n'y a bien évidemment qu'un seul octet par caractère ! D'ou une économie de coût de stockage de moitié si l'on travaille en ASCII (donc CHAR, VARCHAR) comparé à la version UNICODE (NCHAR, NVARCHAR, le N voulant dire *NATIONAL*).
Il y a donc une sacrée différence de performance entre 1 et 2 octets. Le double tout simplement. Or il est rare qu'une application soit à la fois internationale et internationalisée. Internationale au sens ou tout être humain quelque soit sa nationalité, sa culture et en particulier sa langue soit en mesure de l'utiliser, et internationalisée, au sens ou des utilisateurs de cultures différentes et donc de langues diférentes vont devoir utiliser la même application en même temps. Les rares cas que je connaisse consiste en des sites webs interactifs planétaires ou l'on s'est simplifié la vie en imposant finalement l'anglais comme langue commune !
Il n'y a donc pas d'intérêt à utiliser de l'UNICODE s'il n'y a pas une raison probante pour le faire.

Et pour obtenir de la souplesse dans l'éventualité d'un changement de type, on peut utiliser des domaines SQL[5] au sein du modèle de données. Alié à un outil de modélisation qui gère cette particularité de la norme SQL, la transformation d'un type SQL en un autre dans un modèle comportant de très nombreux attributs en sera grandement facilité, notamment pour ce qui est des littéraux. D'autant que SQL Server intégre de manière intelligente

[3] pour les nordistes, le Pastis !
[4] Avant l'ASCII 8 bits le 7 bits était le premier standard d'alphabet pour la communication par voie automatisée, utilisée notament pour les téléscripteurs, les telex et les télétypes.
[5] Exemple, norme SQL : CREATE DOMAIN D_POURCENT AS REAL CHECK (VALUE BETWEEN 0 AND 100). Dès lors n'importe quelle table ou vue peut utiliser directement un domaine SQL en lieu et place d'un type.

et performante la notion de domaine à travers ses *user types*, ses *rules* et ses *defaults*[6].

Puisque nous avons parlé culture, parlons des collations. Non il ne s'agit pas de casse-crouter. Le terme collation possède un sens ancien plus profond : il s'agit de comparer des manuscrits, des textes... C'est ce que fait le pilote de l'Airbus A380 qui vous transporte au septième ciel lorsqu'il confirme dans sa voie l'ordre qu'il vient de recevoir du contrôleur aérien : "*Vol 714 pour Sydney, veuillez monter à six mille pieds", "OK, ici vol AF 714, nous montons à dix mille pieds"*. Remarquez que dans le présent dialogue, la collation est mauvaise. Le but étant bien évidemment d'obtenir confirmation de l'ordre à exécuter en faisant répéter le destinataire. Dans notre exemple, le contrôleur doit immédiatement réagir pour informer le pilote de son erreur et lui intimer de corriger sa trajectoire. De l'utilité de la collation.
Encore un bon point pour SQL Server. Il implémente parfaitement bien la notion de collation notamment depuis la version 2000. Le seul hic : peu de gens le savent et très peu s'en servent !

collation : l'oublié du littéral

Une collation au sens SQL du terme est un objet qui permet de définir le comportement de la chaîne de caractères qu'elle qualifie. Ce comportement piloté par la collation peut être le respect ou non de la casse[7], le respect ou non de caractères diacritiques[8], le respect ou non de la largeur d'encodage[9], le respect ou non des différents types de caractères japonais Kana[10] (Hiragana et Katakana) et pour finir l'ordre du tri des littéraux[11]...
Or la collation a un coût ! La comparaisons d'éléments dissemblables n'est pas magique. Elle repose sur des algorithmes et des structures de données dont l'exécution est plus ou moins performante selon la collation adoptée.
De plus les données littérales sont stockées d'une manière spécifique en fonction de la collation choisie, tant est si bien que tout changement de collation "on the fly" est impossible sans une migration des données.
En revanche, il est clair que l'absence de collation, c'est à dire la comparaison binaire de chaînes de caractères (collation forte) est plus rapide qu'une collation faible (faisant confusion de la casse ou des accents par

[6] Il me faut hélas le dire, depuis la version 2005, Microsoft considère comme "deprecated" ces éléments et ne propose pas pour autant un code de remplacement. Cependant la communauté des développeurs MS SQL Server insiste pour que soit créé la notion de domaine SQL parmi d'autres revendication. Notez que malgré cette dépréciation, SQL utilise toujours en interne le type sysname pour la qualification des noms des objets... Quelle contradiction !
[7] le nom de *casse*, c'est à dire les majuscules (haut de casse) et les minuscules (bas de casse) viennent du fait que dans la préhistoire de l'imprimerie moderne, celle qui a commencé avec Gutemberg, les compositeurs de texte avaient devant eux un bac organisé en de multiples petit casiers remplis de caractères en plomb et appelé casses. Les majuscules étaient situées en haut et les minuscules en bas pour des raisons d'ergonomie, d'ou leurs noms.
[8] de *dia* deux et *critique* différence, c'est à dire un caractère de base (lettre) différencié par un symbole supplémentaire comme une accent, une ligature, une cédille, etc.. D'ailleurs savez vous que & est une ligature des lettres *e* et *t* et @ une ligature de *ad*. En matière de ligature il y a aussi l'*e* dans l'*o* (cœur, bœuf...) et l'*e* dans l'*a* comme dans Lætitia (merci Gainsbourg) ou encore ex æquo... Et pour finir il y a des ligatures technique, pour la beauté de l'impression, tel que *fi*, *ft* ou pire la ligature à trois lettres *ffi* !
[9] ASCII ou UNICODE
[10] Je n'ai absolument aucune idée de ce qui se cache derière le Kana, n'étant pas japonophile. Si quelque lecteur peut m'informer à ce sujet, j'en serais ravi !
[11] Ordre linguistique du dictionnaire

exemple). Or cette collation sans collation a un nom, c'est la collation binaire !
En dehors de cette collation binaire[12], toute collation provoquera plus de traitement dans toutes les manipulations de données littérales. Il semble donc intéressant de définir une collation binaire le plus souvent possible.
Mais ou faut-il la mettre ? SQL Server accepte de définir la collation à différents endroits : le serveur, les bases, les colonnes et l'expression. Pour des raisons de performances, vous êtes invités à spécifier la collation au niveau du serveur et n'en changer que dans certains cas exceptionnels.
Mais une collation binaire au niveau serveur est-elle bien adaptée à la majorité des cas ? Non seulement je dirais oui, mais l'usage d'une collation binaire à l'installation du serveur nous réserve une surprise de taille...
Comme nous l'a enseigné le bon Docteur Codd (règle n°4) *la description des objets contenus dans une base de données doit être représentée, comme pour les données ordinaires, c'est à dire par des valeurs dans des tables*. La conséquence est que si vous installez votre serveur avec une collation binaire, donc sensible à la casse, tous les noms des objets des bases de données doivent être écrits exactement de la manière dont ils ont été créés c'est en dire en respectant la casse et les éventuels caractères diacritiques. Loin de nuire à l'écriture du code et en particulier des requêtes, cette façon de faire minimise le cache des procédures. En effet l'entrée et la recherche dans ce cache se fait par un tableau des chaînes de caractères décrivant les procédures et les requêtes. Pour aller le plus vite possible, les données de ce cache sont stockées en binaire et la comparaison s'effectue en hexadécimal. Si les noms des objets d'une même requête sont écrit tantôt en majuscules, tantôt en minuscules, tantôt avec une combinaison des deux, vous allez vous retrouver avec un cache encombré de multiples entrées pour une même requête avec les plans de requêtes associés, donc une augmentation considérable de la taille du cache, mais aussi un vieillissement prématuré de ces entrées, donc une perte plus rapide des plans préétablis du fait de l'algorithme LRU...
Bref il est donc bénéfique à de nombreux niveaux d'utiliser une collation binaire, même si vos développeurs trouvent ridicule l'idée que l'écriture des requêtes SQL devienne sensible à la casse !

Faut-il utiliser un INTEGER, un SMALLINT, un TINYINT ou un BIGINT ?
De manière similaire à la problématique des littéraux, losque l'on a le choix, il faut considérer les différentes options. Mais là on doit compter avec un paramètre supplémentaire : la taille du mot du processeur. En effet, l'optimisation ne sera pas la même si SQL tourne sur un OS 32 bits ou sur du 64... Dans un environnement 32 bits, un entier de 64 bits (BIGINT) coûte un peu plus de 2 fois plus en traitement qu'un 32 bits (INT). En 64 bits un entier 32 bits coûte un tout petit peu plus en traitement qu'un 64 bits. La raison en est simple : le seul moyen, pour le processeur, de tester si le 32 bits en est réellement un (et donc qu'il n'y a pas d'overflow[13]) est de le contrôler algorithmiquement à l'aide d'une passe dans un circuit

[12] Attention dans SQL Server 2005 il existe deux collations binaires identifiées par exemple pour l'ordre du dictionnaire français en French_BIN et French_BIN2... La plus intéressante étant la toute dernière (BIN2).
[13] Je devrais dire dépassement de capacité, mais c'est beaucoup plus long !

électronique microprogrammé. D'ou un surcoût de traitement de l'ordre de quelques pourcents. Autrement dit, pour des jointures entre clefs auto incrémentées, soit l'OS est 32 bits et ces données doivent être définies en INT, soit l'OS est 64 bits et il faut songer à mettre du BIGINT. Quant à l'argument qui consiste à dire que le 64 bits est deux fois plus volumineux que du 32 bits, il est parfaitement concevable, mais va peser peu en regard du volume d'une ligne de la table. Songez donc si ce cas vous tourmente à réduire quelques uns de vos CHAR d'un ou deux caractères et le tour sera joué !

La ligne de table

On passe souvent sous silence la notion de longueur des lignes d'une table. Pourtant la connaissance de cet élément peut s'avérer précieuse pour certains cas d'optimisation. En effet, SQL Server stocke les informations dans des pages de données dont la taille est de 8 Ko[14]. L'espace réellement utilisable pour les données est de 8060 octets, c'est à dire 8 Ko moins quelques petits octets techniques comme l'identifiant numérique de la page, les références aux pages suivantes et précédentes, l'indication de la nature de la page et la clef de l'objet (table ou index) auquel appartient la page. Par principe, une ligne doit tenir dans une page. Ainsi, aucune ligne d'aucune table, hormis les colonnes de type BLOB, ne peut dépasser 8060 octets.
Or il arrive de temps en temps que certaines lignes obèses, dimensionnées de manière un peu trop arbitraire grèvent les performances en lecture. Que pensez vous donc d'une ligne d'une table qui mesurerait 4050 octets ? En tout état de cause, vous ne pouvez pas en placer plus d'une par page. Ce qui signifie que près de 50% des pages sont vides ou que le coût de stockage de la table est le double d'une table ayant des lignes juste un tout petit peu moindre !
Pensez donc à auditer la longueur moyenne des lignes des tables ayant les colonnes les plus volumineuses...

Qualité d'une bonne clef

S'il est une race de donnée entre toutes qui doit mériter l'attention la plus extrême, c'est bien celle qui se cache derrière la ou les colonnes constituant la clef d'une table. A l'école, on apprend bêtement que la clef d'une relation doit être choisie parmi les attributs de cette dernière. Dès lors s'il nous faut identifier une personne, et cela avec l'interdiction cnilesque[15] d'utilisation du NNI[16], il est concevable que tout un chacun utilise les attributs NOM +

[14] Une taille située entre celle de la granule système de stockage (généralement 1 Ko) et la quantité minimale optimale de lecture d'un disque, voisine de 64 Ko.
[15] Adjectif inventé signifiant *de la cnil*, c'est à dire provenant de cette Commission Nationale Informatique et Liberté, créée à la suite de l'affaire SAFARI (tappez dans Google les mots clefs Giscard et "chasse aux français") et qui s'est récemment brillament reniée en autorisant notre état à interconnecter tous les fichiers informatiques par croisement sur le numéro de sécurité sociale.
[16] NNI : Numéro National d'Identité attribué à chaque français par l'INSEE et qui se compose de différents éléments tels que sexe, année, mois, département et commune de naissance et enfin rang.

PRENOM + DATE_NAISSANCE, voire quelques autres éléments si statistiquement le volume des données à traiter risque d'être élevé.
Comme cet exemple est en matière de performances, le pire de tous, voyons quels en sont les inconvénients.

Une clef stable ne doit revêtir aucune sémantique

D'abord, c'est long, et contrairement au dicton, plus c'est long, plus c'est mauvais ! Une clef composée de 40 caractères par exemple provoque des calculs beaucoup plus complexes qu'un nombre, notamment lorsqu'il faut faire des jointures ou des tris[17].
Ensuite, c'est mouvant[18]. En effet le nom d'une personne peut changer, notamment lorsqu'elle se marie. Cela va donc entraîner une quantité phénoménale de modifications, et risque de créer des faux[19].
Enfin, c'est lourd. Écrire des requêtes avec des jointures qui nécessitent 3 colonnes mise en relation... Quelle fatigue !

Nous pouvons donc en déduire quelques éléments concernant la qualité d'une bonne clef. Elle doit être :

- la plus courte possible;
- stable dans le temps;
- composée d'une seule colonne.

A l'évidence une clef dont la longueur ne dépasse pas la longueur du mot du processeur[20] est parfaite au niveau des calculs machiniques.
A l'évidence, la stabilité dans le temps ne peut être vraie que si la clef ne revête aucune sémantique[21] dans l'univers modélisé.
Se pose alors la question : quel type de données est le plus apte à représenter une clef ?
Une dernière condition va vous faire toucher du doigt la solution : plus le type considéré peut représenter différentes valeurs, plus il sera facile à utiliser pour des tables volumineuses. Ainsi, l'utilisation d'une clef littérale composée de 4 caractères est battue à plate couture par l'entier 32 bits : les caractères inférieurs à 20 (espaces) sont réputés non imprimables, donc quasiment impossible à saisir, de plus certaines caractères au dessus de 128 sont difficiles à frapper au clavier. On ne peut donc compter que sur environs 100 caractères différents, ce qui fait du littéral de 4 caractères une clef ne pouvant représenter que 100 millions de tuples différents. Alors que notre simple entier limité aux valeurs positives permet déjà plus de 2 milliards de combinaisons...
D'ou l'intérêt d'une clef purement numérique et attribuée au hasard à chaque nouvelle insertion.

[17] La plupart du temps une jointure exige un tri préalable !
[18] Je n'ai pas choisi cet adjectif au hasard. J'ai pensé aux sables mouvants dans lesquels s'enlisent les concepteurs de modèles de données.
[19] J'ai montré comment l'utilisation d'une telle clef pouvait entraîner la création de fausses factures punissables au pénal !
[20] Donc 4 octets pour les OS/machines 32 bits et 8 octets pour les OS/machines 64 bits.
[21] Autrement dit et contrairement à ce que l'on vous a appris à l'école sur les fumeuses dépendances fonctionnelles, n'utilisez jamais un attribut ou une combinaison d'attributs comme clef, ajoutez votre clef et considérez là comme un attribut à part entière. Notez que c'est ce que l'INSEE à fait avec le NNI ou le ministère du transport avec l'immatriculation des véhicules, qu'ils soient terrestres à moteurs ou même aéronefs !

Au hasard ? Nous y reviendrons... Car le hasard ne fait pas bien les choses. En revanche, l'auto incrément, avec le bon index, si !

Mais il y a parfois quelques idées fumeuses qui traînent en matières de conception de données et de modélisation de clefs. Pour anecdote, il y a quelques années un internaute m'apostrophe par un mail[22] en me demandant de lire un e-paper concernant un truc génial sur les clefs... Un informaticien avait eu la bonne idée de préconiser l'utilisation du GUID (vous savez cet identifiant universel généré au hasard et qui pèse 32 octets). Dès lors, plus besoin de clef primaire, plus besoin de contraintes d'unicité ! La merveille des merveilles... Sauf en ce qui concerne le coût et les performances. Calculer un GUID est très complexe et 32 octets, nécessite 8 passes dans le processeur à chaque traitement. Bref ce génie avait inventé la clef qui tue !

On notera donc qu'une bonne clef est un entier de la longeur du mot du processeur et par défaut attribuée automatiquement pas auto incrément.

une clef auto incrémentée minimise la fanêtre des données

Seule ombre au tableau, l'insertion massive de lignes dans une table avec une clef auto incrémentée provoque un phénomène bien connu des quinquagénaire ayant commu les rames Thompson du Métropolitain de Paris : ça se bouscule au portillon !
Néanmoins sachez-le, la contention liés à ce problème s'est nettement améliorée depuis les toutes dernières version de SQL Server.
En revanche, un des avantages considérables de l'utilisation des clefs sur auto incrément est qu'il minimise la fenêtre des données. Comment cela est-il possible ? Simplement par le fait que, statistiquement, nous utilisons beaucoup plus fréquemment les lignes les plus récemment insérées, donc, celles situées vers la fin de l'index plutôt que les plus anciennes. Cela est vrai pour les données comptables, les salaires, les commandes mais aussi dans une moindre mesure pour les biens à vendre ou encore les clients...

De l'intérêt des index

On ne le dira jamais assez, une des optimisations les plus faciles consiste à choisir les bons index. En cette matière, j'ai l'habitude de dire que l'indexation d'une base ne doit pas représenter une part trop supérieure à 1/3 du volume global de la base, sauf le cas particulier des données statiques. En effet si le volume des index double celui de la base, le gain apporté par ces derniers est diminué par l'encombrement du cache. Il y a donc un équilibre à trouver. En d'autres termes, trop d'index tue l'index[23] !
En effet si l'index est intéressant en lecture parce qu'il va réduire drastiquement les temps de réponse, il s'avère coûteux en mise à jour (INSERT, UPDATE, DELETE) car il oblige chaque ligne manipulée à autant d'entrées à ajouter, supprimer ou modifier suivant le nombre d'index. Par

[22] Sur http://www.developez.com
[23] En référence à Edouard Balladur qui a dit "trop d'impôts tue l'impôt" phrase démagogique qui ne l'a pas empêché d'en ajouter de nouveaux...

exemple une table dotée de quatre index dans laquelle nous faisons l'insertion d'une ligne induit fatalement quatre données à insérer dans les quatre index en sus de l'insertion de la ligne dans la table.

Mais au fait, qu'est-ce qu'un index ? Imaginez que je vous invite chez moi à dîner et que je vous affirme habiter aux Champs Elysés à Paris[24]. Avec cette seule indication, vous avez intérêt à venir de bonne heure car il va vous falloir aller d'immeuble en immeuble, sonner à toutes les portes afin de savoir si un certain Brouard habite ici.
Il est bien certain que si je vous avais indiqué le numéro dans la rue, vous auriez trouvé plus facilement. Or comme un fait exprès, les immeubles des rues de nos villes ont été numérotés en séquence[25], dans le sens de la marche. Tant est si bien que trouver le bon immeuble s'avère un jeu d'enfant et économise du temps et des souliers. Et bien c'est cela un index : une information supplémentaire qui permet de retrouver une donnée le plus rapidement possible. Nous utilisons couramment une quantité phénoménale d'index sans le savoir : le répertoire de votre téléphone mobile, le calendrier, le numéro des chaînes de TV, un dictionnaire, le bottin des pages jaunes, la table des matière d'un livre... sont autant d'index !

Dans une base de données, un index consiste à redonder une ou plusieurs colonnes en les triant. Mais il faut faire référence à l'origine de ces données c'est à dire la table. Une structure d'index contient donc à la fois les données indexées et une référence à la ligne de la table dans laquelle on trouve cette donnée. En fait un index est une structure de données annexe à la table, devant être en permanence synchrone avec les données de la table.
La structure de stockage qui est aussi la méthode de tri permettant des recherches rapide, est basée sur ce que l'on apelle un arbre équilibré, c'est à dire une arborescence ou toutes les données (autrement dit les feuilles de l'arbre) sont situées au même niveau. Ainsi, que l'on cherche Amandine ou Zoé, l'effort de pacours de l'index doit être le même. Cet arbre est donc constitué d'une racine, c'est à dire du point d'entrée dans l'index, de pages de navigation, c'est à dire d'aiguillages et enfin de pages de données que sont les feuilles de l'arbre[26]. A noter que la racine est aussi une page de navigation, donc d'aiguillage. Il y a donc deux natures de pages dans un index : celles qui permettent de naviguer et celles qui contiennent les données.

Rendons nous compte à quel point un index bien formé peut minimiser le temps de recherche d'une donnée. Imaginons une table contenant des employés avec tout un tas de rubriques comme la clef primaire de type entier auto incrémenté, un nom, un prénom, un matricule, un numéro de sécurité sociale, une date de naissance, un adresse, des moyens de contacts...

[24] Rassurez-vous, je n'en ai pas encore les moyens !
[25] En camargue où les routes de campagne sont longues et les maisons peu fréquentes, la numérotation est au kilomètre. Ainsi, si l'on vous donne rendez-vous au n° 3.821, cela qui signifie que vous devez parcourir 3e km + 821 m à partir de l'origine de la numérotation.
[26] Contrairement aux arbres terrestres, les arbres informatiques se représentent avec la racine en haut et les feuilles en bas... Sans doute encore une lubie de nos cousins anglo-saxons qui roulent à gauche et placent les adjectifs avant les substantifs.

Assurons nous que la clef, comme c'est souvent le cas, ait généré un index qui a permis de trier la table sur cette donnée.
Admettons que dans cette table la longueur de la ligne soit de 200 octets, qu'en particulier la clef soit de 4 octets, et le nom de 16 caractères donc 16 octets en moyenne. Suggérons qu'il y ait 100 000 lignes dans cette table.
Pour rechercher toutes les lignes dont le nom est Brouard, il faudra parcourir toute la table soit plus de 19 Mo. En fait il faut lire 2 500 pages[27].

Un index bien choisi doit diviser par cent voire mille les temps de réponse

Créons maintenant un index sur cette colonne. Il faudra 100 000 x (16 + 4) octets[28], soit 1,9 Mo. Nous avons donc déjà minimisé la recherche au 10e de ce qu'elle était à l'origine. Cependant, nous savons que les données dans un index sont ordonnées. Autrement dit le coût d'une recherche n'est pas linéaire mais logarithmique... Par exemple chacune de nos pages de données concernant cet index comporte environ 366 lignes[29]. Il faut donc 274 pages pour stocker toutes les entrées de cet index[30]. Pour les pages de navigation de cet index une seule page suffit. En effet une ligne d'une page de navigation d'index est constitué par la référence du premier nom de la page de données, associé au numéro de cette page de données, soit à nouveau 22 octets. Or la page racine peut adresser jusqu'à 366 pages de données. Tant est si bien que la recherche de notre nom dans l'index ne nécessite jamais que la lecture de 2 pages et non plus 274 comme si l'on devait lire tout l'index. C'est bien évidemment radicalement moindre que le balayage de toutes les données de la table soit 2 500 pages ! Pour notre cas de figure un rapport 1 250 fois moindre...
Dans les faits, on a l'habitude de dire qu'un index bien choisit doit divisier par cente à mille les temps de réponse.

Bien entendu SQL Server pose sans vous le dire des index lorsque cela s'avère nécessaire. Chaque fois que vous placez une contrainte de clef primaire, SQL Server créé un index de type *cluster* afin de minimiser la durée de vérification de l'unicité. Chaque fois que vous définissez une contrainte d'unicité SQL Server pose, pour les mêmes nécessité, un index non cluster.
Reste qu'aucun autre index n'est automatiquement posé par SQL Server sur les intégrités référentielles alors qu'il est bon de poser de manière quasi systématique des index sur toutes les contraintes de clefs étrangères et sur les colonnes les plus fréquemment recherchées. Par exemple dans notre table des employés, les colonnes nom, matricule et numéro de sécurité sociales devraient avoir chacune un index.

J'ai parlé de *cluster*, mais qu'est-ce qu'un index cluster ? C'est un index qui mélange la table et l'index. Puisqu'un index constitue fatalemet une redondance des données, pourquoi ne pas éviter cette redondance pour au moins un des index de la table ? D'où l'idée de l'index cluster, c'est à dire de

[27] 100 000 (nombre de lignes) / 8060 (taille de la page) * 200 (taille de la ligne), les calculs étant effectués en nombres entiers.
[28] Les quatre octets surnuméraires afin de faire référence dans l'index à la clef permettant de retrouver la ligne dans la table.
[29] 8060 (taille de la page) divisé par 22 (taille de la ligne + offset de ligne)
[30] 100 000 (entrées) divisé par 366 (nombre de lignes dans une page)

trier les lignes de la table dans l'ordre des colonnes formant l'index. Dans ce cas de figure, seules les pages de navigation de l'index cluster sont surnuméraires.
Il y a un inconvénient à l'index cluster. En effet si cet index cluster est bâti sur une colonne dont les données arrivent chronologiquement, à l'insertion, dans le désordre le plus complet, il faut en permanence retrier toutes les lignes de la table. En revanche si cet index est bâtit sur une colonnes chrono ordonnées, alors l'effort de tri est quasi nul ! Or quels types de colonne s'avèrent naturellement chrono ordonnées ? Les clefs auto incrémentées[31] !

Nous en concluons qu'un index cluster est très intéressant sur une clef primaire auto incrémentée, mais qu'il peut s'avérer néfaste pour d'autres types de données composant la clef. Autrement dit, réfléchissez au comportement par défaut de SQL Server qui propose un index cluster pour toute clef primaire, ce n'est pas toujours la panacée...

Nous avons dit qu'un index faisait référence à la clef de la table. Mais qu'en est-il d'une table sans clef, dans laquelle on créé quelques index ? Dans ce cas il est nécessaire que SQL Server retrouve la ligne de la table qui peut d'ailleurs être un doublon. Contrairement à quelques SGBDR qui numérote les lignes à l'insertion, SQL Server agit sur une logique parfaitement ensembliste et considère que c'est à vous d'agir ! Mais pour résoudre ce cas de figure SQL Server utilise une couteuse combinaison d'informations : le numéro du fichier, le numéro de la page et le numéro de la ligne soit 12 octets. Une telle information doit aussi se retrouver dans toutes les entrés de tous les index secondaires...
Ce qui fait dire à certains auteurs anglo-saxons "*every table should have a key*". Sous entendu, un index cluster unique et not null !

Pousons nous maintenant la question des index multicolonnes. Sont-ils intéressants ? Oui et non. En fait dans un index multicolonne, l'information est *vectorisée*, c'est à dire que chaque colonne affine la précision de la précédente. Pour réaliser cela, l'information est concaténée. Ceci fait que des recherches ne peuvent pas s'effectuer par le parcours de l'index si ces informations sont à l'intérieur de l'index.
Par exemple si l'index est composé des colonnes NOM, PRENOM et DATE_NAISSANCE, alors toute recherche efficace sur cet index ne peut utiliser que les combinaisons suivante : NOM ou NOM + PRENOM ou NOM + PRENOM + DATE_NAISSANCE. Ainsi la recherche d'un prénom ou d'une date de naissance seule ne pourra bénéficier de l'apport que constitue un index.

Si un index multicolonne n'est pas efficace dans certains cas, il peut cependant s'avérer payant du fait de sa couverture... Nous savons qu'un index est redondant. Il porte en lui toutes les informations des colonnes indexées et s'il s'agit d'un index secondaire sur une table dotée d'une clef primaire, ce même index contient aussi la valeur de la clef.

[31] Mais aussi et tout bêtement, les colonnes de type DATETIME d'horodatage...

Dès lors toute requête portant sur des données uniquement contenues dans l'index n'a pas besoin de lire les informations de la table. Nous avons là un index dit "couvrant" pour la requête considérée.
SQL Server 2005 a étendu cette notion en permettant lors de la définition d'un index d'ajouter des colonnes d'informations qui, bien que ne participant pas à la définition proprement dite de l'index, permet de couvrir certaines requêtes... Autrement dit il est possible d'ajouter à un index une ou plusieurs colonnes informatives de manière à assurer une couveture artificielle !

On pourrait encore parler des heures des index. Alors, continuons, le sujet n'est pas inépuisable, mais presque...

Une idée pour aller plus vite dans la recherche des index, est de réduire la taille des données. Mais comment faire ? Un vieil outil mathématico informatique permet de transformer une donnée littérale en un code numérique. C'est ce que l'on apelle clef de hachage. SQL Server, et en particulier depuis la version 2005, offre la possibilité d'utiliser différents algorithmes pour calculer une correspondance numérique avec un littéral. Pour avoir mis cela en place dans une base de données comportant quelques millions d'adresses e-mail, je peu vous affimer que le gain peut être impressionnant ! Dès lors il suffit de combiner une colonne calculée dans la définition de la table et d'indexer cette colonne. Voire d'externaliser la clef de hachage dans une table en relation avec la table originale.

l'arme absolue de l'indexation est constituée par la vue indexée

D'autres systèmes permettant de réduire la masse des informations à chercher sont possibles, comme le partitionnement, aidé en cela par des fonctions particulières. Ainsi en matière de nom de personnes, on peut utiliser des mécanismes de consonnance (Soundex, Metaphone, Phonex...) ou encore des tables de Cutter Sanborn.

Mais il y a mieux. L'arme absolue en matière de performance apportée par l'indexation est constituée par la vue indexée. Dans un cours d'optimisation que je donne de manière régulière, je montre comment passer de requêtes dont l'exécution provoque la lecture de 98 000 pages de données à 2 en allant de la table non indexée, aux index, simples puis composites pour aboutir à la vue indexée, clou du spectacle !

Dernier élément dont je voudrais vous parler, en matière d'indexation. Je vous ai dit que l'inconvénient de l'index est son coût en matière de mise à jour des données (INSERT, UPDATE, DELETE). Ce coût peut être notablement amoindri en utilisant un facteur à la création de l'index. Ce paramètre s'apelle le *fill factor*. Un facteur de remplissage de 100 % qui signifie que lors de la création les pages de l'index sont remplies au maximum, implique que toute nouvelle insertion provoquera une réorganisation de l'index, et en particulier la césure d'une page en deux, avec mise à jour de l'ensemble des pointeurs faisant référence à l'ancienne page. Ce réorganisation a un coût exhorbitant... C'est elle qu'il faut éviter. Cela s'apelle un *split* de page. En jouant sur le facteur de remplissage et en créant

des index dont les pages ne seront pas remplies à plus de 80% lors de la création de l'index, on se réserver 20% d'insertions qui ne devrait pas entraîner de split de page, et cela au prix d'un index dont le surcroit en volume n'est que de 20%... Mais au bout du compte il faudra bien un jour reconstruire ces index car le facteur de remplissage ira inexorablement en diminuant au fur et à mesure des insertions. Cela est une autre histoire, elle s'apelle maintenance !

Je m'aperçoi que j'ai oublié de vous parler de l'indexation textuelle, sujet d'une richesse incalculable... Comment trouver un mot rapidement dans un texte ? Et pour, les pluriels ? Les conjugaisons ? Les synonymes ? Les homonymes ? Comment proposer à l'utilisateur un mot qui existe alors qu'il à commis une fautre de frappe, une erreur orthographique ? Des solutions efficaces existent pour toutes ces questions. Et en matière de proposition de correction aorthographique, il existe des solutions plus ou moins efficaces qui s'apellent algorithme du dictionnaire de Knuth, différence de Hamming, distance de Levenshtein, inférence directe, index rotatifs...

Normalisez, normalisez encore, normalisez toujours...

La normalisation des données, c'est à dire la construction d'un modèle de données sans redondance et dans lequel aucune anomalie de mise à jour n'est possible, devrait être la règle. Hélas la plupart des personnes qui modèlisent s'affranchissent en cours d'élaboration du modèle de certaines de ses règles de base.
Reprenons les une par une et critiquons les...

Règle n°1 : *Une entité est en première forme normale si toutes ses propriétés sont élémentaires et s'il existe au moins une clef caractérisant chaque occurrence de l'entité.*

Outre l'obligation de clef, l'atomicité d'un attribut doit être étudié, non pas sous l'angle du pur modèle des données, mais sous l'angle des traitements qui devront être opérés sur cette donnée.
Étudions par exemple, la collecte des informations sur des personnes. Un attribut NOM peut comporter le titre, le nom et le prénom d'une personne si aucun traitement particulier n'est envisageable sur ce conglomérat de données (données purement informationnelles). En revanche si l'on doit chercher une personne par son nom de famille ou établir des statistiques sur le sexe des personnes, alors il faudra séparer ces éléments. Et si le traitement consiste à envoyer une offre promotionnelle au moment de la fête (Saint Claude, Saint Pierre...) alors il faudra probablement scinder les prénoms composés afin d'éviter une requête trop complexe ou l'envoi en double du bon de réduction ! Enfin, si le volume des données est très important il peut être intéressant d'externaliser les informations les plus redondantes comme le titre (M., Mme., Mlle. ...) ou le prénom. Cela économisera de nombreuses occurrences des Claude, Jean, Marc ou Luc...

Bref, rien qu'à ce niveau, une seule table, mais déjà quatre modèles de données !

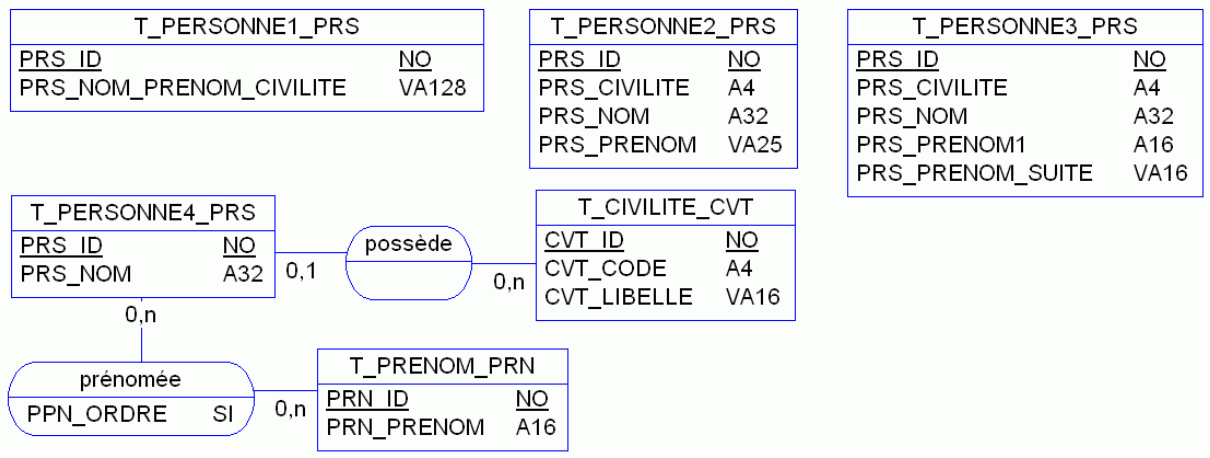

Figure 1 : *une entité, quatre modèles différent suivant les traitements demandés et le volume expecté des données*

Autre exemple : faut-il décomposer ou non un numéro national d'identité[32] ? Encore une fois, tout dépend du traitement. Dans une table d'employé il y a peu d'intérêt de le faire. Mais dans une application de gestion de données médicales, cela peut revêtir une importance particulière, car ce numéro comporte le sexe, le lieu de naissance et indirectement donne une idée de l'âge, toute données qui peuvent avoir un intérêt magistral dans bien des cas ou l'on fouillera les données à le recherche de coorélations entre l'âge, le sexe et la survancne de telle ou telle maladie !
A titre d'exercice, posez vous la question de la modélisation d'une adresse IP...[33]

Règle n° 2 : *Une entité est en deuxième forme normale si et seulement si elle est en première forme, et que tout attribut n'appartenant pas à une clef ne dépend pas que d'une partie de cette clef.*
Tout ce charabia pour nous dire que certaines informations anodines d'une table peuvent s'avérer être en partie redondante avec les informtions contenues dans la clef. Dans la pratique, cette règle a deux implications : soit l'un de vos attribut peut dépendre de la clef, mais plus vicieux, si votre clef est décomposable, alors décomposez-la et constatez que parfois certaines informations n'ont pas bedoin de figurer en sus de la clef.
Prenons un exemple. Notre numéro national d'identité constitué de 13 caractères est en fait une clef composée de 6 éléments (en sus de la clef permettant de contrôler la saisie). Dès lors comment modéliser correctement un patient pour lequel on aura prévu d'utiliser comme clef primaire de la table ce fameux numéro INSEE ? Le premier caractère donnant le sexe inutile de le redonder. Les caractères 6 à 10 donnant le département et la commune, inutile de saisir en sus le lieu de naissance, utilisons une table de références des communes par codification INSEE...

[32] Communément appelé numéro de sécu.
[33] Pensez à l'IPV6 et au fait qu'il peut être intéressant de savoir quel sont les machines de tel ou tel sous réseau...

Voici résumé dans différentes étapes, un tel affinage du modèle (voire figure 2, 3 et 4)...

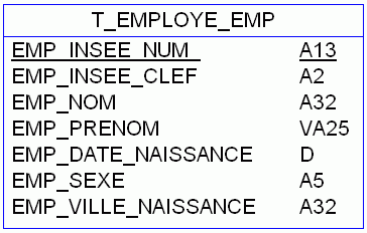

Figure 2 : modèle trivial la clef de la table, n° INSEE dans une seule colonne

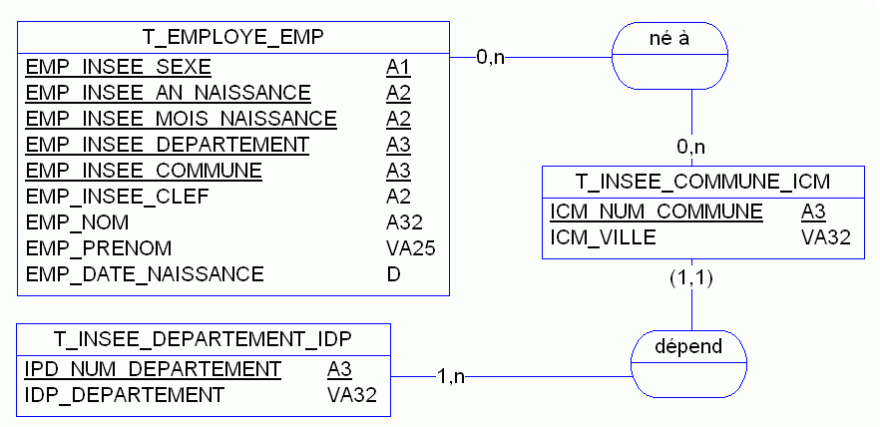

Figure 3 : affinage du modèle précédent,
mais redondance des informations de commune et département de naissance

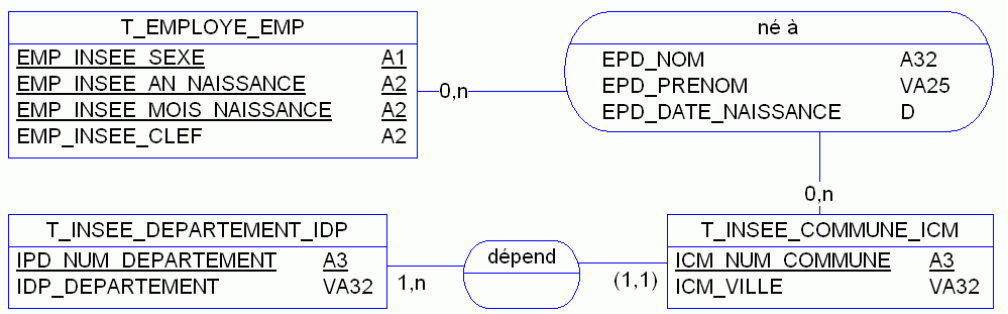

Figure 4 : modèle optimum, plus de redondance des informations de lieu de naissance

Dans ce cernier modèle (figure 4) parfait du point de vue normalisation, plus aucune redondance n'est visible. Il a fallut créer un lien identifiant entre les entités *insee_commune* et *insee_departement* et reporter les informations de l'employé dans l'association ! Pas évident de parvenir à un tel modèle sans un certain savoir faire et une bonne pratique de la modélisation de données.
En pratique, ce modèle conceptuel aboutira à la base de données suivante (figure 5).

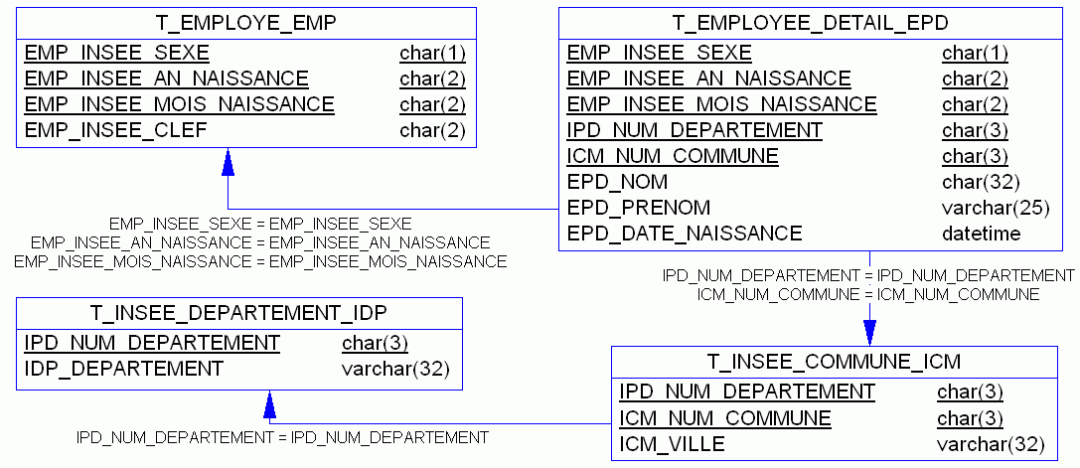

Figure 5 : base de données du modèle affiné

Reste qu'un tel modèle possède une clef peu efficace. On trouvera donc plus juste de rajouter dans toutes les entités des clefs informatiques auto incrémentées et de transformer les clefs sémantiques en contraintes d'unicité. On bénéficiera alors du meilleur des deux mondes : concision de la clef donc performance des jointures et non redondance absolue donc moindre volume des données.

modèle diachronique et modèle synchronique s'affrontent sur le terrain de la modélisation

Règle n° 3 : *Une entité est en troisième forme normale si elle est en deuxième forme normale et que tout attribut n'appartenant pas à une clef ne dépend pas d'un attribut non clef.*
Ceci pour dire qu'il faut se méfier de certaines données : elles peuvent induire une redondance, et tout particulièrement en ce qui concerne les clefs étrangères. Par exemple dans une table ou le n° INSEE de personne figuren en tant qu'attribut non clef, le simple fait de rajouter le sexe ou le lieu de naissance suffit à violer cette règle. Mais il y a un cas ou cette règle parait être violée et ou elle ne l'est pas. C'est celui des modèles dont les données doivent être "*photographiées*" à un instant précis de la chronologie applicative... Autrement dit le problème de la réprésentation du temps dans un modèle de données. A ce stade, deux techniques s'affrontent : le modèle synchronique et le modèle diachronique... Lequel va gagner ?
Étudions encore une fois un exemple parlant, celui de la gestion des commande de produits marchands. Le modèle que tout un chacun fait est le suivant (voire figure 6).

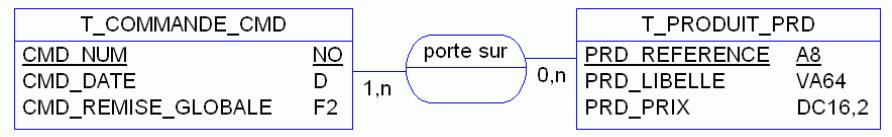

Figure 6 : modèle classique de gestion de commandes de produits

Mais ce modèle possède un inconvénient majeur : si l'on doit changer le libellé ou le prix d'un produit, alors la réédition des commandes portant sur les produits chanés génère des documents différents des originaux. D'où

l'idée de dupliquer prix et libellé dans l'association "porte sur" afin de prendre un cliché, un instantané, une copie des valeurs des données des produits concernant chaque commande. Voici donc notre modèle plus orthodoxe (figure 7). Il est appelé modèle synchronique dans le sens ou l'information est piqué à l'istant de la commande.

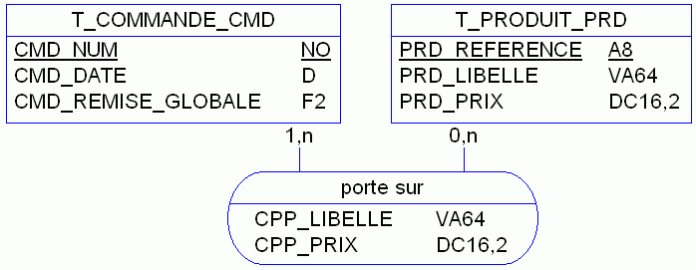

Figure 7 : modéle de gestion des commandes de produits dit "synchronique"

Mais ce modèle introduit une redondance néfaste au volume des données puisqu'il est probable (et souhaitable) que de nombreuses commandes portent sur les mêmes produits... De plus il perd certaines informations ! En effet, imaginez que vous avez dans votre offre produit une savonette fort cher et qui sent très mauvais. Il est probable qu'aucun client ne l'achète jamais. Le directeur commercial s'en rend compte et modifie le prix à la baisse. La savonette dès lors commence à se vendre malgré sa puanteur... Comment savoir que vous avez eu un produit que vous n'avez pas vendu ? Cette information a disparu du fait du défaut de ce modèle !

Pour contrer ces problèmes, l'idée est alors de gérer une version des produits dans le temps. A chaque changement opéré sur la ligne, on conserve l'ancienne version avec une date d'obsolèscense. Ceci suppose de rajouter à la clef de l'entité produit un attribut de type DATE. Cette technique dite diachronique, conduit au modèle suivant (voire figure 8) :

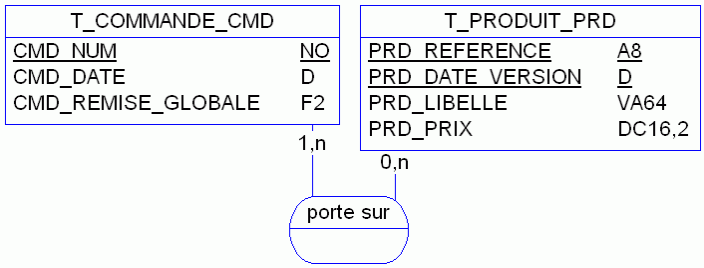

Figure 8 : modéle de gestion des commandes de produits dit "diachronique"

Mais véhiculer une date dans une clef n'est pas très performant. Alors, pour conserver toute sa beauté à un tel modèle, on peut utiliser une clef non sémantique, par exemple un auto incrément. Dès lors on bénéficie du meilleur des deux mondes, mais attention à garder l'unicité du couple d'attribut référence + date version.
Ce dernier modèle se schématise comme suit (voire figure 9) :

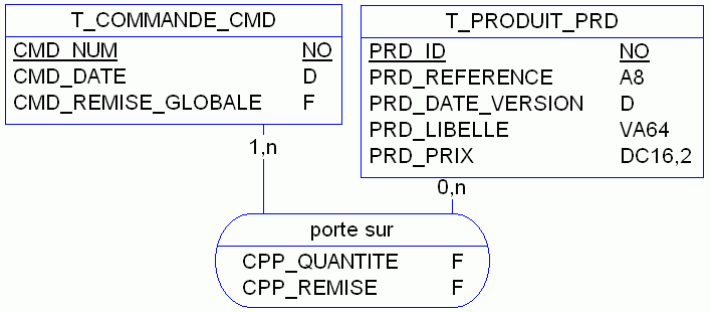

Figure 9 : modèle diachronique optimisé, une contrainte d'unicité doit figurer pour le couple d'attribut PRD_REFERENCE + PRD_DATE_VERSION

Arrêtons là notre introspection des formes normales et de l'optimisation des modèles. Il y aurait encore beaucoup à dire... Intéressons nous rapidement à quelques techniques particulières.
Peu d'informaticiens pensent à utiliser les techniques d'héritage des données. Pourtant cela résoud bien des problèmes et permet d'économiser souvent un masse significative de données. De même pour la modélisation par méta modèles, c'est à dire le possibilité de garder un modèle évolutif de stockage de l'information sans pour autant restructurer en permanence le schéma de la base. Enfin, sachez qu'il existe des modèles très particuliers pour résoudre très élégammant des problèmes complexes de données. Par exemple pour la modélisation des arborescenses et autres hiérarchies, il existe un outil très efficace : le mode intervallaire.

Mais pour ceux qui ne sont pas convaincus par l'intérêt d'un modèle parfaitement normalisé, je donnerais un seul exemple. Celui concernant la modélisation d'une personne et de ses coordonnées. Non seulement un modèle non normalisé entraîne de la redondance et donc un volume accru de données, mais il rend les requêtes les plus triviales très difficiles à écrire et donc catastrophiques en terme de performances. Voyons cela de près. Dans le modèle pas normalisé constitué d'une seule table (voir figure 10), comment retrouver efficacement une personne qui habite *rue Pasteur* ou dont le numéro de téléphone est *04 11 47 28 52* ou encore qui possède l'email *toto@microsoft.com* ?

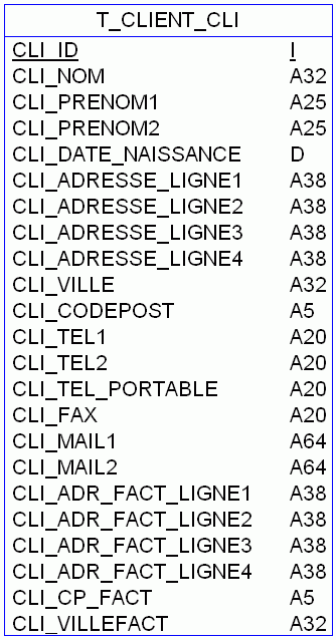

Figure 10 : modèle pas normalisé des coordonnées d'une personne

```
Requêtes SQL dans un modèle de coordonnées client pas normalisé. Exemple, recherche par
numéro de téléphone :
SELECT *
FROM   T_CLIENT_CLI
WHERE  CLI_TEL1 = '04 11 47 28 52'
   OR  CLI_TEL2 = '04 11 47 28 52'
   OR  CLI_TEL_PORTABLE = '04 11 47 28 52'
   OR  CLI_FAX = '04 11 47 28 52'
Cette requête balaye toute la table des clients en lisant 4 colonnes.
```

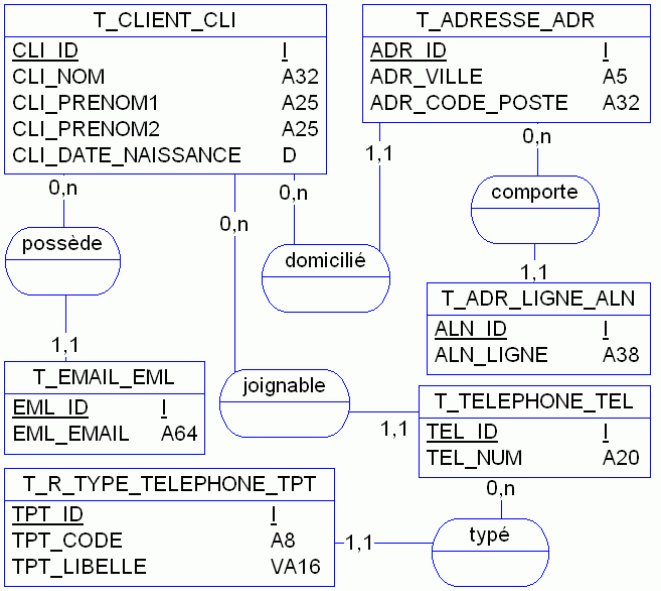

Figure 11 : modèle normalisé des coordonnées d'une personne

```
Requêtes SQL dans un modèle de coordonnées client bien normalisé. Exemple, recherche par
numéro de téléphone :
SELECT *
FROM   T_CLIENT_CLI AS C
       INNER JOIN T_TELEPHONE AS T
             ON C.CLI_ID = T.CLI_ID
WHERE  TEL_NUM = '04 11 47 28 52'
Cette requête peut profiter d'un index posé sur la colonne TEL_NUM et parcoure donc un
volume infime de données.
```

Dans un exercice que je donne dans le cadre d'un cours d'optimisation, je pose les conditions suivantes et demande de comparer le volume des deux modèles (figures 10 et 11) : Supposons qu'il existe 500 000 clients, dont la moitié a une adresse renseignée avec une moyenne de 2 lignes par adresses (incluant code potal et ville), et qu'en moyenne on trouve 1,5 n° de téléphone par client et que 50% ont un email, 10% deux emails... Quel est le modèle le plus performant ?

Je terminerais pas un constat. Dans son offre logicielle Microsoft ne présente pas d'outil performant de modélisation de données. C'est dommage. Mais recourir à un outil de modélisation comme Power AMC ou Win Design est indispensable. Certe le coût en est élevé, mais l'outil est plus pérenne que les environnements de développement et ne concerne que quelques personnes dans l'entreprise. Mais surtout je démontre à tous mes clients les gains spectaculaires que l'on obtient avec un tel outil. Et il est rare qu'il ne soit pas amorti en quelques mois !

Ne dénormalisez jamais, sauf...

A priori et compte tenu du fonctionnement d'un SGBDR, la dénormalisation, c'est à dire la redondance de l'information, nuit aux performances. Il y a cependant quelques rares cas ou elle apporte un certain bénéfice.
Or il est impossible de mesurer le bénéfice, et surtout la moins value apportée par la nécessaire augmentation des temps de traitement des mises à jour (INSERT, UPDATE, DELETE) sans avoir eprouvé le modèle avec un volume de données qui représente une masse proche de celle qui sera réellement exploitée.
Lisez la discussion que nous avons eu sur Internet à ce sujet[34]. Elle est pleine d'enseignement et riche de contradictions. Mais elle converge sur un point : aucune dénormalisation, sans avoir prouvé le bénéficie absolu que l'on peut en tirer !
N'ayez donc aucune crainte : un modèle performant est un modèle parfaitement normalisé. Il n'y a qu'à l'épreuve du temps, et donc à l'augmentation concommittente du volume que l'on pourra décéler les éléments candidats à une dénormalisation, en tester la pertinence et modifier en conséquence le schéma.

[34] En fait, deux discussions, la première sur la dénormalisation, l'autre sur l'unicité des clefs, le tout sur developpez.com aux URL : http://www.developpez.net/forums/showthread.php?t=6231 et http://www.developpez.net/forums/showthread.php?t=281221

Mais s'il faut dénormaliser, autant que ce soit avec le maximum de précaution : évitez la redondance dont le calcul s'effectue sur le client. Choisissez toujours un mécanisme de redondance qui garantisse l'intégrité de la base. En cette matière, le choix est large. Ma préférence, va dans l'ordre aux techniques suivantes : vues indexées, colonnes calculées indexées, procédures stockées remplaçant les requêtes directes (avec une gestion de la sécurité afin d'interdire ces dernières), trigger. Dans certains cas on pourra précalculer des données agrégées par exemple chaque nuit et compléter le résultat si besoin est avec les seules données du jour à l'heure d'exécution de la requête.
Dans tous les cas mesurez le gain de temps en lecture. S'il est inférieur à 2, il vaudrait mieux s'abstenir, car les pertes des temps de traitement des mises à jour risquent de devenir pénalisantes. A partir d'un gain de 10, allez-y franco[35]. Avec un gain de 1000, facilement obtenable avec une bonne vue indexée s'il y des calculs d'agrégats, vous aurez le statut de magicien !

Vous n'êtes pas encore convaincu de l'intérêt d'un modèle solide et bien normalisé ? Vous voulez discuter de la chose ? Alors n'hésitez pas à me joindre par courriel[36] à SQLpro@club-internet.fr

[35] Il semble que ce soit le général Massu, au moment de la dictature espagnole, qui à conseillé à De Gaulle qui se tatait de se présenter à l'election présidentielle "Allez-y Franco mon général !"
[36] pour les anglophone email ! ;-)