Le bêtisier de l'été

La-men-table ! ! !

Selon le principe de Peter, les employés très incompétents ne posent pas de problème, car ils sont rapidement détectés et mis "hors d'état de nuire". Voici quelques exemples prouvant que ce n'est malheureusement pas toujours vrai.

Dans certaines grosses entreprises, personne n'a le temps de rien, et surtout pas de vérifier que les programmeurs travaillent correctement, en suivant les procédures, etc.

Conséquence : le travail est uniquement évalué au vu des résultats : ils sont corrects, donc le programme est bon. Que l'individu ait écrit 3000 lignes alors qu'il aurait pu le faire en 500, qu'il y ait passé 3 semaines au lieu de 3 jours… ceci est secondaire.

Ce comportement est à rapprocher de certains hauts reponsables qui ne jugent le bon fonctionnement d'une entreprise qu'au travers d'un seul élément : le montant de la marge bénéficiaire.

Tous les exemples cités dans cet article sont rigoureusement authentiques… et proviennent tous de la même entreprise ! Simplement, comme il s'agit de programmes écrits en cobol dans un environnement IBM gros système (SGBD DB/2, transactionnel CICS…), j'ai transcrit le code en "pseudo-code" français, pour permettre à tout un chacun, même non informaticien, de comprendre.

Il suffit simplement de savoir que "`A ← B + C`" signifie "mettre en A la somme de B et C", et que "`A ← B`" signifie "copier le contenu de B dans A".

Pourquoi faire simple quand on peut faire compliqué ?

C'est le défaut des programmeurs qui se jettent sur le travail d'écriture du code sans analyser, et qui ont tendance à associer productivité et nombre de lignes de code écrites.

Je précise bien "de code", car en général, plus le programme est mal écrit, moins y il a de commentaires pour aider à la compréhension.

Premier cas

Trouvé plusieurs fois dans un seul et même programme, le code ci-dessous, en cobol, occupait six lignes (if … / then / … / else / … / end-if) !

Ce qui a été écrit	Ce qu'il suffisait de faire

`si A = 0`
`  alors B ← 0`
`  sinon B ← A`
`fin`
	`B ← A`

Deuxième cas

Ceci est un grand classique :

Ce qui a été écrit	Ce qu'il suffisait de faire

si condition
alors
  traitement 1
  traitement 2
sinon
  traitement 1
  traitement 3
fin
	traitement 1
	si condition
	  alors traitement 2
	  sinon traitement 3
	fin

Si traitement 1 ne fait qu'une seule ligne, ce n'est pas encore trop grave ; mais j'ai rencontré de tels cas avec plus de dix instructions bêtement répétées dans les deux termes de l'alternative.

Troisième cas

Une simple variante du précédent :

Ce qui a été écrit	Ce qu'il suffisait de faire

si condition1
  alors fin
si condition2
  alors fin
si ...
si conditionN
  alors fin
	si condition1
	ou condition2
	ou ...
	ou condition3
	  alors fin

Quatrième cas

Comment avoir un gros programme en évitant de varier le code :

Ce qui a été écrit	Ce qu'il suffisait de faire

fonction x(n)
début
  traitement (n)
fin fonction
	fonction x
	début
	  pour I variant de 1 à 10
	  traitement (I)
	  fin
	fin fonction

(40 fois dans le programme) : pour I variant de 1 à 10 exécuter x(I) fin	(40 fois dans le programme) : exécuter x

Il faut savoir que la boucle ci-dessus, écrite en cobol et bien aérée, occupait à chaque fois 6 lignes, et que la fonction appelée ne l'était jamais autrement que dans une telle boucle !

Cinquième cas

Le programmeur disposait d'une fonction envoyant l'information transmise vers l'imprimante. Ayant des espacements variables à faire, il a défini des fonctions lui permettant de créer divers espacements en une seule instruction. Après avoir défini une fonction `espace1` générant une ligne blanche, il a écrit deux autres fonctions générant respectivement 2 et 3 lignes blanches :

fonction espace2
début
  exécuter espace1
  exécuter espace1
fin fonction
	fonction espace3
	début
	  exécuter espace1
	  exécuter espace1
	  exécuter espace1
	fin fonction

Jusqu'ici, tout va bien.

Par contre, quand il a voulu écrire une fonction permettant de créer un nombre N de lignes blanches, il s'est quelque peu emmêlé les pinceaux. Jugez-en :

Ce qui a été écrit	Ce qu'il suffisait de faire

fonction espace(N)
début
  Triple ← partie entière (N/3)
  Reste ← reste (N/3)
  pour I variant de 1 à Triple
  exécuter espace3
  fin
  si Reste = 2
  alors exécuter espace2
  sinon
  | si Reste = 1
  | alors exécuter espace1
  | fin
  fin
fin fonction
	fonction espace(N)
	début
	  pour I variant de 1 à N
	  exécuter espace1
	  fin
	fin fonction

Sans commentaires…

Sixième cas

Ici, notre programmeur atteint de "codorrhée"[1] s'est surpassé, au point que je ne puis que vous expliquer le problème sans pseudo-code, ou bien alors réserver 8 pages de LA LETTRE rien que pour çà !

Le problème : il s'agissait de lire un fichier, de cumuler certaines valeurs en fonction de divers critères, et d'écrire les résultats cumulés dans un autre fichier. Le fichier en entrée contenait onze informations numériques (quantitée livrée en moins d'un jour, quantité livrée le lendemain, quantité livrée en 2 jours, …, quantité totale à livrer, quantité totale déjà livrée, etc.), toutes du même format, en deux groupes distincts (soit au total 22 valeurs). Cinq niveaux de cumul étaient à faire (par produit, par gamme, par vendeur, par secteur, par région).

La bonne réponse : il suffisait de définir, tant en entrée qu'en sortie, un tableau à deux dimensions (2 par 11), en interne un tableau à 3 dimensions (5 par 2 par 11) et d'utiliser des indices pour parcourir l'ensemble des données. Ceci se fait aisément en 200 lignes de cobol.

La programmation effectuée : il y a 11 informations ayant chacune un sens propre, donc on déclare 11 variables (que la sémantique des données soit sans importance lui échappe complètement). Il y en a 2 groupes, ce qui fait 22 variables pour le fichier en entrée, et autant pour le fichier en sortie. Les 5 niveaux de cumul nécessitent autant de structures de 22 variables, soit 110 zones de travail. Bien sûr, pour le traitement, il faut faire les assignations de cumul et de copie pour chaque variable. Bref, le programme fourni a ainsi atteint 800 lignes !

Quelle différence y a-t-il entre une table relationnelle et un fichier ?

Nombreux sont ceux qui ne le savent pas et créent des requêtes plus complexes que nécessaire et qui coûtent généralement beaucoup plus cher en exploitation.

---

[1] *Une personne atteinte de "logorrhée" (en grec, logos = la parole) ne peut s'empêcher de parler continuellement. Vous pouvez donc aisément en déduire le sens du mot "codorrhée".*

Premier cas

Nous avons deux tables A et B. A est identifié par une clé C, B par une clé D, et contient un champ X ayant une contrainte d'intégrité sur le champ C de A, c'est-à-dire qu'on ne peut créer ou mettre à jour une ligne de B qu'en mettant obligatoirement dans X une valeur correspondant à une ligne existante de A.C. En termes merisiens, on a une cardinalité (1,1) de B vers A.

Donc, quand on lit B, on est sûr qu'il existe dans A une ligne telle que A.C = B.X. Pourtant, on trouve parfois des requêtes de ce genre :

Ce qui a été écrit	Ce qu'il suffisait de faire
`lister D de B` `  dont B.X = A.C`	`lister D de B`

En d'autres termes, notre programmeur demande seulement les lignes de B qui correspondent à une ligne existante dans A. Alors que toutes les lignes de B sont dans ce cas… Le moteur du SGBD effectuera le contrôle (il n'utilise les contraintes d'intégrité qu'en mise à jour) et consommera donc plus de ressources.

Nota : ce contrôle aurait été pertinent si le champ D avait été facultatif ou si aucune contrainte d'intégrité n'avait été définie.

Deuxième cas

Un programme reçoit les paramètres P1 et P2, et doit lire une table T (contenant notamment un champ K), avec un filtre dépendant des paramètres :

- si P1 = 'A', lire les lignes telles que K = P2 ;
- si P1 = 'B', lire les lignes telles que K ≥ P2 ;
- si P1 = 'C', lire les lignes telles que K > P2.

Qu' a fait notre programmeur codorrhétique ? voyez vous-mêmes :

Ce qui a été écrit

```
définir liste1 curseur pour
   lister K, X, Y, Z de T
     dont K = P2 ;

définir liste2 curseur pour
   lister K, X, Y, Z de T
     dont K >= P2 ;

définir liste3 curseur pour
   lister K, X, Y, Z de T
     dont K > P2 ;

si P1 = 'A'
alors ouvrir liste1
   faire jusqu'à fin de liste1
   |  lire liste1
   |  exécuter traitement
   fermer liste1
fin

si P1 = 'B'
alors
   ouvrir liste2
   faire jusqu'à fin de liste2
```

(suite colonne 2)

Ce qu'il suffisait de faire

```
définir liste curseur pour
   lister K, X, Y, Z de T
     dont (K=P2 et P1≠'C')
       ou (K>P2 et P1≠'A') ;

ouvrir liste
faire jusqu'à fin de liste
|  lire liste
|  exécuter traitement
fermer liste ;
```

(suite de la colonne 1)

```
   |  lire liste2
   |  exécuter traitement
   fermer liste2
fin

si P1 = 'C'
alors
   ouvrir liste3
   faire jusqu'à fin de liste3
   |  lire liste3
   |  exécuter traitement
   fermer liste3
fin
```

À noter que dans ce cas, les performances ne sont pas dégradées, mais bonjour la complication du programme…

Troisième cas

Il faut récupérer des informations dans deux tables, sachant que pour chaque ligne de la table T1 correspondent de 0 à N lignes de la table T2. Chaque ligne de T1 est identifiée par le champ C et contient le champ E, chaque ligne de T2 est identifiée par les champs (C, D) et contient le champ F. Dans le pseudo-code ci-dessous, il suffit de savoir que pour chaque information lue, on sait si elle a une valeur ou pas.

Il existe en SQL une fonction appelée jointure externe qui permet de lier deux tables, et de récupérer les données de l'une même s'il n'y a pas de correspondance dans l'autre. En d'autres termes, si la table T1 contient les lignes [C1, C2, C3] et la table T2 les lignes [(C1,D1), (C1,D2), (C3,D2), (C4, D4)], une requête avec une jointure simple ne fournira que les combinaisons existantes entre les deux tables, soit :

```
C1 D1 E1 F1
C1 D2 E1 F2
C3 D2 E3 F3
```

…alors qu'une requête avec une jointure externe sur la table T1 fournira :

```
C1 D1 E1 F1
C1 D2 E1 F2
C2 -  E2 -
C3 D2 E3 F3
```

Malheureusement, bien que le programme en question ait été écrit bien après l'installation d'une version du SGBD permettant ce genre de jointure, le programmeur a complètement ignoré cette option, et a travaillé comme ceci :

Ce qui a été écrit	**Ce qu'il suffisait de faire**

définir liste1 curseur pour
  lister C, E de T1 ;
 	
définir liste2 curseur pour
  lister C, D, F de T2
  dont C = T1.C ;
 	
ouvrir liste1
faire jusqu'à fin de liste1
| lire liste1
| ouvrir liste2
| si liste2 vide
| alors
| exécuter traitement A
| sinon
| faire jusqu'à fin de liste2
| | lire liste2
| | exécuter traitement B
| fin
| fermer liste2
fermer liste1 ;
	définir liste curseur pour
	  lister T1.C, T2.D, T1.E, T2.F
	  de T1,
	  joint à T2 si T2.C = T1.C ;
	
	ouvrir liste
	faire jusqu'à fin de liste
	| lire liste
	| si D a une valeur
	| alors exécuter traitement B
	| sinon exécuter traitement A
	| fin
	fermer liste ;

Bref, plutôt que de laisser le moteur du SGBD faire tout le travail en une seule fois, l'individu a géré un appareillage de fichiers, obligeant le moteur du SGBD à créer un nouveau curseur sur la deuxième table pour chaque ligne lue de la première table. Je ne vous dis pas les performances…

Quatrième cas

En simplifiant, il s'agissait d'afficher à l'écran une liste d'informations, l'affichage de chacune étant conditionnée par six paramètres (arrivées/mouvements, pour un responsable donné ou pas, pour une livraison donnée ou pas, pour un état d'en-cours ou pas, pour un fournisseur donné ou pas, pour un code pièce donné ou pas).

La complication[2] des tables du SGBD faisait que chaque combinaison des deux premiers paramètres nécessitait des requêtes différentes. Par contre, pour les quatre autres paramètres, il n'y avait pas de difficulté particulière. Voyons comment aurait pu être réalisée la recherche pour une combinaison donnée des deux premiers paramètres… et comment elle l'a été en réalité.

Nous appellerons P1, P2, P3, P4 les 4 paramètres, et V1, V2, V3, V4 les variables contenant les valeurs recherchées (la valeur "blanc" signifiant "pas de recherche").
Une condition `"V=blanc ou P=V"` est remplie (c-à-d considérée comme vraie) :
- si V est à blanc, la valeur de P important donc peu (pas de filtrage)
- ou si P a la même valeur que V (filtrage).

Ce qui a été écrit

```
définir liste1 curseur pour
lister P1, P2, P3, P4, A, B de T ;

définir liste2 curseur pour
lister P1, P2, P3, P4, A, B de T
  dont P1 = V1 ;

définir liste3 curseur pour
lister P1, P2, P3, P4, A, B de T
  dont P2 = V2 ;

définir liste4 curseur pour
lister P1, P2, P3, P4, A, B de T
  dont P1 = V1
    et P2 = V2 ;
```

...(je vous épargne le détail)

```
définir liste16 curseur pour
lister P1, P2, P3, P4, A, B de T
  dont P1 = V1
    et P2 = V2
    et P3 = V3
    et P4 = V4 ;

si V1=blanc
alors
|  si V2=blanc
|  alors
|  |  si V3=blanc
|  |  alors
|  |  |  si V4=blanc
|  |  |     alors ouvrir liste1
|  |  |     sinon ouvrir liste2
|  |  sinon
|  |  |  si V4=blanc
|  |  |     alors ouvrir liste3
|  |  |     sinon ouvrir liste4
|  sinon
|  |  si V3=blanc
|  |  alors
```

...(je vous épargne le détail)

```
faire jusqu'à fin de liste
|  si V1=blanc
|  alors
|  |  si V2=blanc
```

(suite colonne 2)

Ce qu'il suffisait de faire

```
définir liste curseur pour
   lister P1, P2, P3, P4, A, B de T
     dont (V1=blanc ou P1=V1)
       et (V2=blanc ou P2=V2)
       et (V3=blanc ou P3=V3)
       et (V4=blanc ou P4=V4) ;

ouvrir liste
faire jusqu'à fin de liste
|  lire liste
|  exécuter traitement
fermer liste ;
```

(suite de la colonne 1)

```
|  |  alors
|  |  |  si V3=blanc
|  |  |  alors
|  |  |  |  si V4=blanc
|  |  |  |     alors lire liste1
|  |  |  |     sinon lire liste2
|  |  |  sinon
|  |  |  |  si V4=blanc
|  |  |  |     alors lire liste3
|  |  |  |     sinon lire liste4
|  |  sinon
```

...(je vous épargne le détail)

```
|  exécuter traitement
fin

si V1=blanc
alors
|  si V2=blanc
|  alors
|  |  si V3=blanc
|  |  alors
|  |  |  si V4=blanc
|  |  |     alors fermer liste1
|  |  |     sinon fermer liste2
|  |  sinon
|  |  |  si V4=blanc
|  |  |     alors fermer liste3
|  |  |     sinon fermer liste4
|  sinon
```

...(je vous épargne le détail)

```
fin
```

[2] *et non pas "complexité".*

Et ceci pour chacun des quatre cas de figure. Notre adepte du "big is beautiful" a donc crée 64 requêtes au lieu de quatre, avec toutes les contraintes qui s'ensuivent, et a ainsi justifié plusieurs semaines de travail pour un monstrueux programme de plus de 4000 lignes.

La solution simple se faisait en 3 jours pour un programme de 700 lignes…

Les index à l'index

Dans un SGBD relationnel, il est primordial de définir un ou plusieurs index correspondant aux principaux types d'accès. Ces index doivent être optimisés, sinon l'utilisation des tables peut coûter très cher en temps machine.

Un exemple pour bien comprendre. Une table a un index sur plusieurs champs, et dans cet ordre : A, B, C. Si on fait une recherche avec une sélection sur A, C, l'index sera utilisé pour le premier champ. Si on fait une recherche avec une sélection sur B, C, l'index ne sera pas utilisé, et la recherche sera entièrement séquentielle.

Note pour mieux comprendre :
un index est dit "unique" si pour une valeur (ou un ensemble de valeurs pour une clé à plusieurs champs) donnée d'un index il ne peut exister dans la table qu'une seule ligne. A contrario, un index "multiple" permet d'avoir plusieurs lignes avec le même identifiant.

Bien sûr, plus il y a d'index, et plus il y a de champs dans un index, plus les mises à jour sont chères. Il faut donc judicieusement doser. Voici donc quelques exemples d'intempérance, qu'un administrateur de Base de Données compétent ne devrait jamais laisser passer... mais qui sont hélas passées !

Premier cas

Une table a été définie avec un index unique sur les champs A, B, C. Un deuxième index, également unique, a été défini sur les champs D, B, A, C, E.

Remarquez que les trois champs du premier index figurent dans le second. Cependant, puisque le premier champ est D, on peut quand même mettre les trois autres, puisqu'il s'agit d'une séquence différente, et que pour une valeur donnée de l'ensemble D, B, A il est possible d'avoir plusieurs valeurs de C.

Par contre, l'ensemble D, B, A, C est forcément unique, donnant donc une seule ligne possible, et donc une seule valeur de E. Avoir mis ce cinquième champ dans l'index est "non pertinent", puisqu'aucun affinage supplémentaire de séquence n'est possible. Par contre, toute mise à jour de E nécessitera une mise à jour de l'index, et donc un coût supplémentaire ; et ce sans aucune possibilité d'économie par ailleurs.

Deuxième cas

Comme ci-dessus, une table a été définie avec un index unique sur les champs A, B, C. Un deuxième index a été défini sur les champs D, B, A, C, mais multiple.

Il est évident que ce second index contenant les champs du premier est unique de fait. L'avoir défini multiple est non seulement stupide, mais peut occasionner également un coût supplémentaire lors des requêtes, puisque si on fait une recherche impliquant l'usage du second index, le moteur du SGBD prévoira la possibilité de récupérer plusieurs lignes pour un même identifiant.

Troisième cas

Une table a été définie, un de ses champs (A) pouvant avoir un nombre indéterminé de valeurs (A1,…An), mais dont deux d'entre elles (mettons A3 et A7) ont une signification importante, et servent souvent de critère de sélection. Le "concepteur" de la table a donc ajouté un champ supplémentaire B pouvant avoir trois valeurs : B=1 si A vaut A3, B=2 si A vaut A7, B=3 dans tous les autres cas.

Ceci pourrait être utile si on voulait lister cette table en commençant par A3 et A7, même si leur valeur les place en séquence n'importe où dans la table.

Dans la réalité, il s'avère que tous les programmes utilisant cette table, soit s'intéressent exclusivement aux lignes contenant A3 ou aux lignes contenant A7, ou bien traitent tout sans donner d'importance particulière à A3 et A7. Bref, le seul programme utilisant le champ B… est celui qui met à jour la table ! De plus, le champ B ne figure dans aucun des index définis pour cette table, ce qui le rend totalement inutile.

En guise de conclusion

On pourrait continuer longtemps ainsi, parler des tables "dupliquées" parce qu'une partie de leur contenu a un usage spécifique, et qu'il a été "jugé" plus simple de dupliquer le contenu plutôt que d'ajouter un champ différenciateur, avec comme résultante autant de programmes de mise à jour que de versions de table, avec 90% du code identique (mais pas commun, ce serait trop astucieux !)… etc.

Mais il faut bien s'arrêter quelque part, et je le ferai sur un dernier exemple.

Deux programmeurs avaient été chargés de modifier des programmes transactionnels existants, afin de permettre aux utilisateurs de définir des critères de sélection (comme Domaine, Famille de produit, Fournisseur, Période…) et d'éditer des listes correspondant à ces critères. Selon les fonctions, il y avait chaque fois entre 8 et 12 critères (dont plusieurs se retrouvaient dans toutes les fonctions), chacun d'entre eux pouvant être, au choix :

- laissé à blanc (= non fourni) ;
- '*', demandant de filtrer sur une liste de valeurs prédéfinies ;
- rempli avec une valeur exacte ;
- pour quelques critères, rempli avec un préfixe (ex: 'A' renverra toutes les informations dont la valeur commence par 'A').

Bref, rien de bien compliqué. On avait donc en gros à chaque fois 10 paramètres, chacun pouvant avoir 3 ou 4 types de valeurs.

Comment a procédé le premier ?
Simplement en créant un premier algorithme généralisé permettant de remplir la table de paramètres destinée à SQL, quatre algorithmes correspondant à chaque type de valeur, puis en les appliquant (par appel de fonction) à chaque paramètre. Dans un langage évolué tel que PL/1, Pascal, C ou même Qbasic, ceci aurait aisément tenu en une centaine de lignes.
Le cobol n'étant malheureusement pas aussi souple, quelque 200 lignes ont été nécessaires. Comme malheureusement l'implémentation locale n'autorise pas les requêtes dynamiques (c'est-à-dire construites dans le programme à l'exécution) en production – qui auraient pu être générées avec 20 lignes supplémentaires –, il a utilisé des requêtes statiques (c'est-à-dire écrites dans le source du programme et compilées avec lui), beaucoup plus complexes puisque non optimisables. Ceci a entraîné un supplément de 200 autres lignes.

En résumé, il a ajouté en moyenne 400 lignes à chaque programme, mis 3 jours pour mettre au point le premier, puis seulement ½ journée pour chacun des suivants. Soit 8 programmes modifiés et testés en à peine plus d'une semaine.

Et comment a travaillé l'autre ?
Eh bien, lui a vu non pas 4 fonctions à appliquer à 10 variables, mais 10 variables ayant chacune 3 ou 4 traitements possibles. Bref, il a dupliqué à mort, sans algorithme commun.
Le résultat ?

1.0 à 1.200 lignes de plus pour chaque programme, une semaine de travail pour chacun, soit un mois pour traiter seulement 4 programmes… et bien sûr des bogues résiduels.

Théorème

Une personne très incompétente ne peut être neutralisée que si ses supérieurs sont suffisamment compétents pour s'en apercevoir et agir en conséquence. ▲

Jean-Luc Blary