Techniques D’optimisation RMS

Métamorphosez vos temps de réponse
par une meilleure maîtrise de RMS

Mathieu LECOMPTE-BOINET

DECYLOG

6 Juin 1991

Objectifs

- Cet exposé vous propose une synthèse des techniques d'optimisation RMS.
- Cet exposé est centré essentiellement sur l'utilisation des fichiers séquentiels indexés.
- Cet exposé est fondé sur l'expérience et sera émaillé d'exemples représentatifs.
- Le but est de donner des indications concrètes sur les méthodes à suivre pour optimiser des fichiers RMS.

Plan

Cet exposé est découpé en 5 parties:

- Présentation de la structure interne des fichiers séquentiels indexés, dans le but de bien comprendre la nature des problèmes posés.

- Paramétrage: description et méthodologie pour régler au mieux l'ensemble des paramètres d'un tel fichier.

- Maintenance d'un fichier séquentiel indexé: quelques règles et conseils.

- Optimisations "run-time": utilisation de buffers pour réduire le nombre d'IOs.

- Performances: réflexion sur l'organisation des traitements pour obtenir les meilleures performances possibles.

Structure interne

Caractéristiques

Un fichier séquentiel indexé est caractérisé par les éléments principaux suivants:

- Les enregistrements sont de longueur fixe ou variable, et peuvent être ajoutés, modifiés ou supprimés à volonté.

- Une clé primaire avec ou sans doublons, non modifiable.

- 0 à 254 clés secondaires avec doublons et modifications.

- Les enregistrements sont triés dans l'ordre de la clé primaire.

- Prologue 3: compression données/index.

Principes généraux

L'organisation interne d'un fichier séquentiel indexé s'appuie sur les entités suivantes:

- Le bucket:
  - -- c'est la plus petite unité d'information manipulée par RMS -> taille d'IO.
  - -- un bucket contient des enregistrements de données ou d'index, ce qui permet de séparer ces deux types d'entités.
  - -- pas d'enregistrement à cheval sur deux buckets.

- L'area:
  - -- permet de regrouper des buckets de même nature: données, index primaire, autres index.
  - -- allocation d'espace disque et extensions par area, ce qui permet de minimiser la tendance au désordre interne.

Organisation des buckets

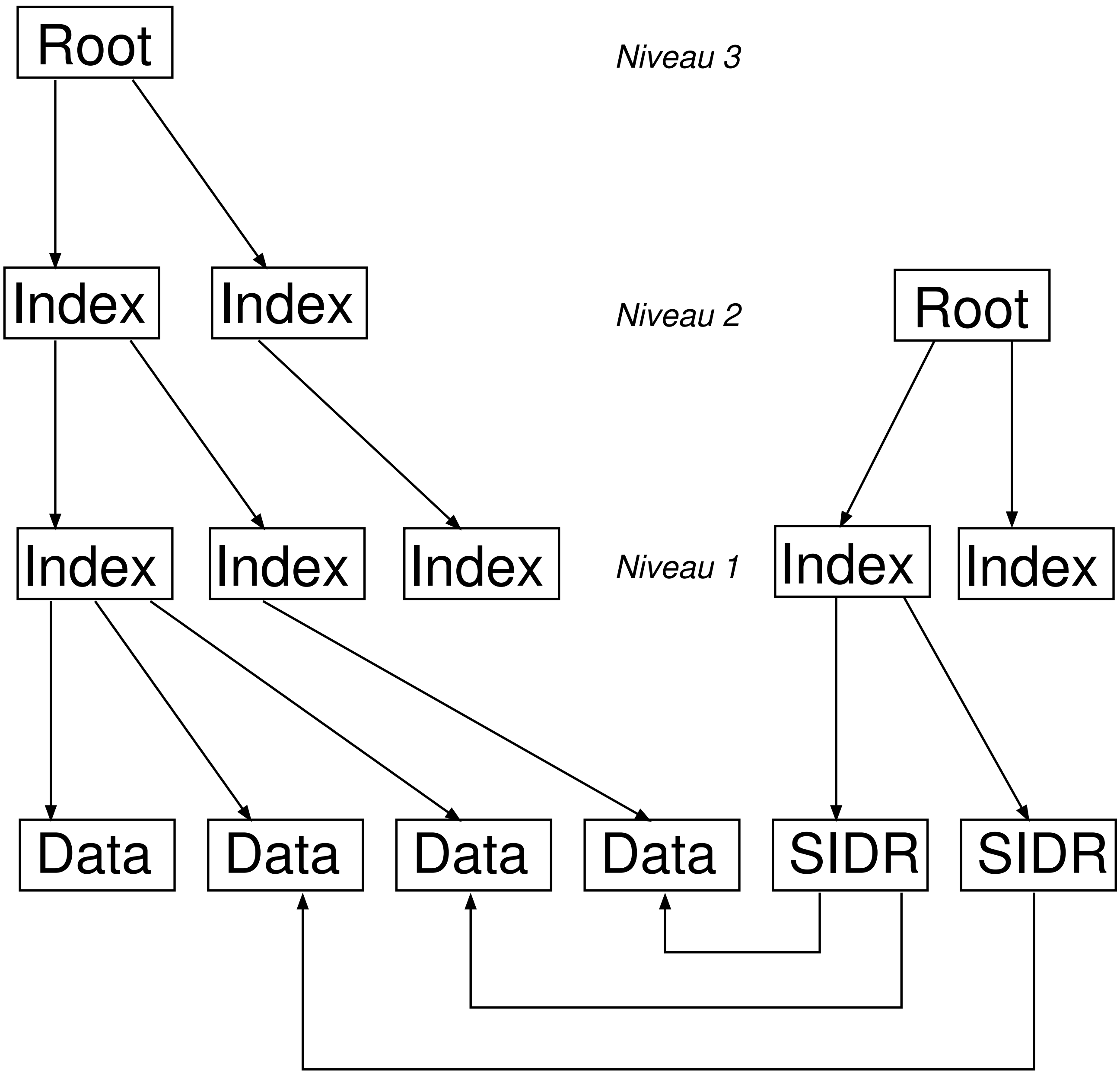
L'organisation interne des buckets repose sur une structure arborescente en niveaux:

- Le niveau 0 (le plus bas) contient les données, triées dans l'ordre de la clé primaire.

- Un ou plusieurs niveaux contenant l'index primaire, selon le principe suivant:
  - -- pour chaque bucket de niveau i, stockage de la valeur de clé la plus haute dans un bucket de niveau i+1.
  - -- si le nombre de buckets d'un niveau est supérieur à 1, alors un niveau supplémentaire est créé de sorte que l'arbre démarre à partir d'un bucket unique, appelé "root bucket".

- Le ou les index secondaires sont organisés de façon analogue, mais le niveau 1 contient des SIDR (clé + liste de pointeurs).

Schéma de principe



Fonctionnement

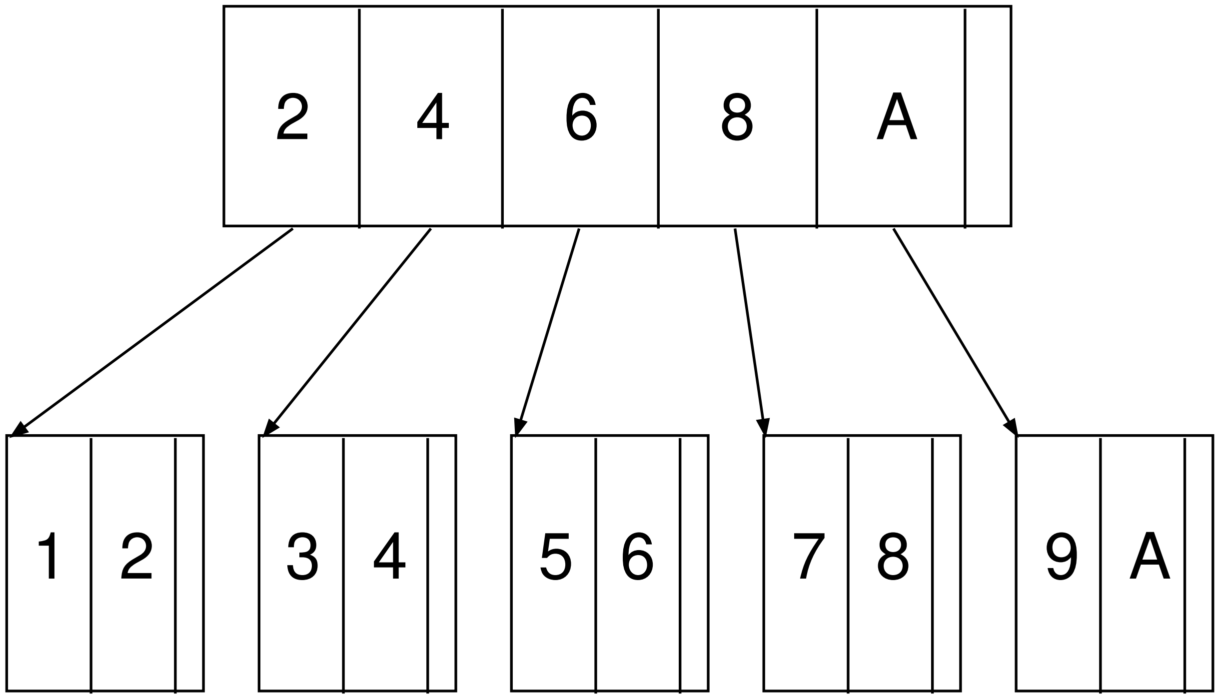
Pour retrouver un enregistrement de clé "x", le principe est le suivant:

- lecture du root bucket et recherche de la première clé supérieure ou égale à "x". Associé à celle-ci figure un pointeur sur le bucket de niveau inférieur.

- lecture de ce bucket et recherche séquentielle selon le même principe: si le bucket courant contient des clés, on recherche la première supérieure ou égale à celle demandée.

- au niveau le plus bas, le bucket contient les enregistrements eux-mêmes. La recherche se fait là aussi de façon séquentielle.

Exemple



Pour retrouver le record de clé "5", RMS lit le "root bucket", le parcourt jusqu'à trouver la première clé supérieure ou égale à 5 (ici: 6), lit le bucket pointé, lequel contient les enregistrements de clés 5 et 6, et le parcourt jusqu'à trouver le record cherché.

Conséquences pratiques

On peut déduire quelques conséquences de ce mécanisme:

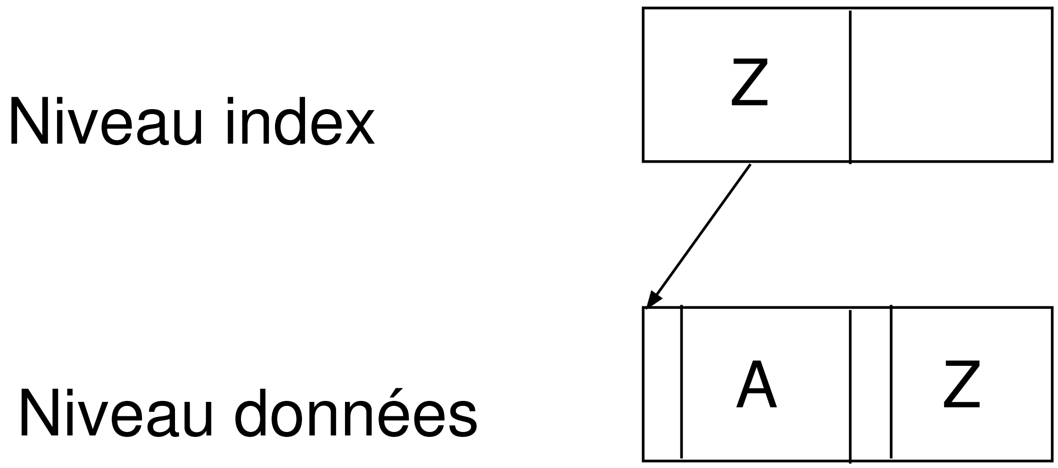
- dans le cas d'une recherche par clé, le nombre d'IO est directement proportionnel au nombre de niveaux d'index.
- plus la taille de bucket est élevée, moins grand sera le nombre de niveaux d'index.
- la contrepartie est un temps de recherche dans le bucket plus long, ainsi que le risque de charger fréquemment en mémoire des informations inutiles.
- les opérations de mise à jour conduisent à un nombre d'IO qui dépend du nombre de clés.
- les suppressions d'enregistrements conduisent à de la perte d'espace.

Conséquences pratiques (suite)

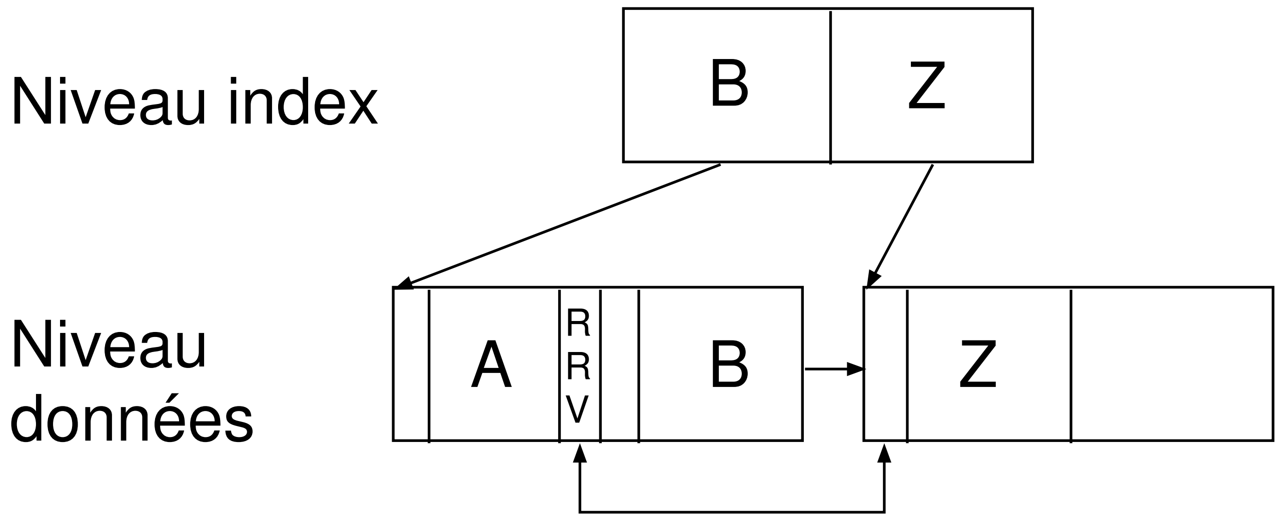
- l’ajout d’un enregistrement au sein d’un bucket plein (rappelons que les enregistrements sont triés dans l’ordre de la clé primaire) déclenche une opération dite de "bucket split":
  - -- un nouveau bucket est créé.
  - -- la moitié du contenu du bucket initial est recopiée dans le nouveau, afin de ménager de la place pour le nouvel enregistrement.
  - -- pour mémoriser le chaînage, un système de pointeurs (RRV) est utilisé.
  - -- l’ajout d’un nouveau bucket implique la mise à jour du bucket de niveau supérieur (ajout dune clé), laquelle peut également déclencher un "bucket split".
  - -- dans le pire des cas, on peut aboutir à la création d’un niveau supplémentaire.
  - -- ceci concerne tous les index.

Bucket split: schéma

Avant Split



Après Split



Conséquences d’un bucket split

L’opération de split de bucket a de fâcheuses conséquences en termes de performances:

- Un coût CPU et IO supplémentaire.
- Le chaînage des buckets entraîne un certain désordre interne, générateur d’IOs supplémentaires et dispersées.
- Il n’y a pas de mise à jour des SIDR: en cas de recherche par clé secondaire, le nombre d’IO est également augmenté (le SIDR ne pointe plus forcément sur le bon bucket).
- L’obligation de créer de nouveaux buckets conduit à étendre l’area correspondante, donc à fragmenter le fichier.

Il faut donc jouer sur le taux de remplissage des buckets et préallouer des areas de taille suffisante.

En résumé

Ce rapide exposé permet de dégager quelques points fondamentaux:

- le rôle que joue la taille de bucket: le problème sera de trouver un compromis entre une taille importante, qui favorise la réduction du nombre d'IO, et le temps de traitement de celles-ci.

- le rôle que joue le taux de remplissage des buckets: en fonction du type d'application, on jouera sur ce paramètre pour minimiser la tendance au désordre interne.

- le choix des deux paramètres précédents dépend également du nombre et de la taille des enregistrements et des clés, ainsi que du taux de compression.

Comment choisir ? Il faut un outil: FDL

Paramétrage

FDL: principe

FDL (File Description Language) est un outil qui permet de décrire de façon symbolique à l'aide de directives les caractéristiques d'un fichier RMS quel qu'il soit:

- FILE: attributs généraux tels que le nom, l'UIC, les protections, etc...

- RECORD: type, longueur, etc...

- AREAn: par area (de 0 à N), définition de l'espace alloué, technique d'allocation, taille de bucket, extension.

- KEYn: pour chaque clé, position, longueur, type, droit de mise à jour, doublons, taux de remplissage des buckets, etc...

Mise en oeuvre

FDL se met en oeuvre très simplement à partir d’une interface DCL permettant de:

- créer un fichier FDL sous éditeur (EDT ou EDIT/FDL).
- extraire le FDL d’un fichier existant, à l’aide d’un outil d’analyse:

  $ Analyze /Rms /Fdl SYSUAF.DAT

- optimiser les paramètres grâce à un outil spécialisé:

  $ Edit /Fdl /Script=Optimize SYSUAF.FDL

- convertir le fichier:

  $ Convert /Fdl=Sysuaf SYSUAF.DAT .NEW

Outil d'optimisation

L'analyseur produit un FDL qui outre les caractéristiques du fichier cible, contient également des informations statistiques utilisées par l'optimiseur (Edit /Fdl).

Celui-ci permet à travers une interface de type menus d'exécuter un script d'optimisation qui assiste l'utilisateur:

- lecture des données d'analyse.
- saisie de quelques critères tels que le nombre d'enregistrements que l'on compte ajouter, l'ordre d'ajout (random ou en fin), ...
- pour chaque clé, l'utilisateur peut définir en particulier le taux de remplissage des buckets et choisir une taille.

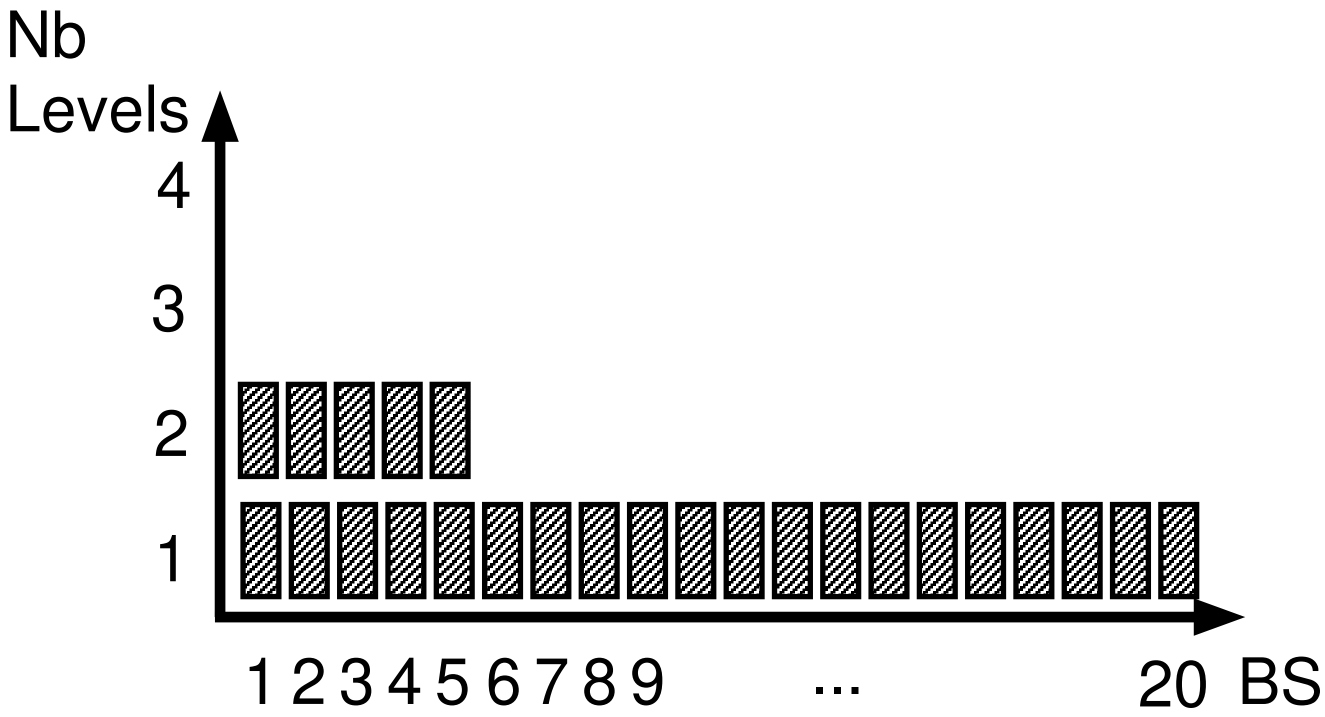
Outil d'optimisation (suite)

- L'outil calcule en fonction des différents paramètres un graphe représentant le nombre de niveaux d'index en fonction de la taille de bucket, et l'affiche.

- L'utilisateur a la possibilité de modifier la valeur de certains paramètres, le plus intéressant étant naturellement le taux de remplissage des buckets.

- Dès que ces paramètres sont fixés, on peut alors choisir une taille de bucket. L'éditeur en recommande toujours une, mais il faut tenir compte des applications:
  - -- accès aléatoires (base articles par exemple): privilégier une taille faible.
  - -- accès plutôt séquentiels ou groupés: ne pas hésiter à grossir la taille.

Exemple de graphe



Dans cet exemple, on voit qu'une taille de bucket de 6 permet de ramener le nombre de niveaux d'index à 1, auquel il faut rajouter le niveau 0 des buckets de données.

Il faut noter que l'optimiseur recommande toujours des valeurs qui sont un multiple de la taille de cluster du disque cible.

Quelques conseils

L'optimiseur FDL génère automatiquement un nombre d'area judicieux (2 ou 3 selon les cas), avec pour chacune d'elle:

- une allocation initiale d'espace disque.
- l'attribut "contiguous best try", qui permet de minimiser la fragmentation lors d'une extension.
- un nombre de blocs d'extension proportionnel à la taille de l'area.
- une taille de bucket spécifique, ce qui permet d'affiner le comportement lorsqu'il est souhaitable de travailler avec des tailles différentes de bucket entre données et index.

Il faut donc laisser faire l'optimiseur.

En conclusion

Les valeurs par défaut utilisées par RMS sont rarement adaptées aux besoins réels. Il est donc indispensable d’utiliser l’optimiseur et de convertir régulièrement les fichiers sensibles d’une application.

Le problème qui se pose est de savoir sur quels critères décider qu’un fichier a besoin d’être paramétré, et à quel rythme le réorganiser.

Il faut donc définir les critères de maintenance d’un fichier.

Maintenance

Principes

Les règles de paramétrage et de gestion d'un fichier séquentiel indexé dépendent uniquement de l'usage qui en est fait, donc de l'application:

- fréquence des mises à jour, qui favorisent la désorganisation des index secondaires.

- fréquence et répartition des ajouts d'enregistrements, qui favorisent la désorganisation de l'index primaire (bucket split).

- accès majoritairement aléatoires (ex: fichier articles), aléatoires avec parcours séquentiel partiel (recherche sur clé incomplète), ou séquentiels (batch nocturne d'édition).

- pertes d'espace dues aux suppressions.

Réorganisations

La réorganisation est une opération normale et nécessaire de maintenance d’un fichier séquentiel indexé. Elle permet:

- de reconstruire complètement la structure interne du fichier, en respectant les taux de remplissage des buckets, et en reconstituant des areas logiquement contigües.
- de récupérer l’espace perdu suite aux suppressions d’enregistrements.
- de préallouer de l’espace disque dans le cas d’un fichier qui est en phase de croissance.

Le rythme de réorganisation pourra classiquement être la semaine, la quinzaine, le mois ou plus selon les cas.

Taux de remplissage des buckets

Un fichier étant une entité qui évolue, le taux de remplissage peut lui-même être adapté en fonction des besoins:

- Dans le cas d’un fichier en cours de peuplement, ou subissant des ajouts fréquents et aléatoires, il est judiieux de prévoir des taux de remplissage minimaux (50%) et des réorganisations fréquentes.

  De plus, il faut préallouer de l’espace disque afin de minimiser les opérations d’extension et la fragmentation qui en découle.

- A l’inverse, pour un fichier qui n’évolue plus ou qui est enrichi dans l’ordre de la clé primaire, un taux de remplissage maximal est recommandé.

Optimisations "run-time"

Principes

RMS offre plusieurs possibilités d'amélioration des temps de traitement par l'utilisation de techniques de "bufferisation" permettant de:

- minimiser le nombre d'IOs, partant du principe qu'une IO de N blocs est plus rapide à traiter que N IOs de 1 bloc.
- anticiper sur les accès ultérieurs: dans le cas de traitements plutôt séquentiels, il est judicieux de charger en mémoire plusieurs enregistrements en un seul accès disque.
- jouer le rôle de cache, en tâchant de conserver en mémoire les informations les plus fréquemment accédées.

Buffers locaux

De façon standard, RMS utilise un ou plusieurs buffers locaux au process par fichier ouvert:

- Pour les fichiers séquentiels, 2 buffers suffisent, la taille étant choisie par la commande

  $ Set Rms /Block_count=N

  RMS utilise les techniques de "Read ahead" et "Write behind:" pendant que l'application lit ou écrit des enregistrements dans un buffer, l'autre sert au transfert avec le disque.

- Pour les fichiers indexés, depuis V5.4, le nombre de buffers utilisé est égal au nombre de niveaux d'index + 2. La taille quant à elle est égale à celle du bucket le plus gros.

Ces buffers sont imputés au Working Set du process.

Buffers globaux

Un buffer global est un buffer d'IO partagé entre tous les process accédant à un même fichier simultanément, et géré dynamiquement par RMS.

Lors du premier open, RMS crée une section globale dont la taille est égale au nombre de buffers globaux déclarés pour le fichier multiplié par la taille du bucket le plus gros. Cette section modifiable est mappée sur pagefile.

Tout autre process accédant au même fichier concurremment mappe la section dans son propre espace virtuel, comme n'importe-quelle section globale.

L'intérêt est que RMS utilise cette section comme un cache en y maintenant les informations les plus accédées, donc en particulier les buckets d'index, ce qui réduit considérablement le nombre d'accès disque.

Mise en oeuvre

Les buffers globaux sont déclarés soit dans le FDL, soit sous DCL:

$ Set File /Global_buffer_count=N

La mise en oeuvre de cette technique nécessite de régler quelques paramètres système:

- GBLSECTIONS et GBLPAGES
- GBLPAGFIL
- RMS_GBLBUFQUO
- Impact sur le nombre de locks -> surveiller SRPCOUNT, LOCKIDTBL

AUTOGEN se chargera de recalculer les autres paramètres liés à ceux-ci (SYSMWCNT, PAGEDYN, ...). La gestion de ces buffers se traduit par un coût supplémentaire d’EXEC.

Choix du nombre

- Le seul moyen de choisir le nombre de buff-ers globaux est de fixer une valeur à priori et de mesurer l'efficacité.
- L'optimiseur FDL donne une indication de la taille mémoire requise pour "cacher" chacun des index, mais ce n'est pas suffisant.
- Pour déterminer l'efficacité des buffers glo-baux, le mieux est d'utiliser Monitor:

```
$ Set File /Statistics  MYFILE.DAT
$ Monitor Rms /File=MYFILE.DAT -
                    /Item=Caching
```

- Il faut viser une efficacité minimale de 75%
- Dans le cas de fichiers tables de faible taille, on arrive ainsi à les charger en mémoire sans problème.

Performances

Création/Population d'un fichier

Les langages traditionnels disponibles sous VMS permettent aisément de créer un fichier séquentiel indexé et de le remplir. Cette facilité reste trompeuse, car rien ne permet de choisir de façon adéquate les paramètres.

Il est donc fortement recommandé de travailler avec FDL pour paramétrer et créer le fichier:

```
$ Create /Fdl=MYFILE   MYFILE.DAT
```

D'autre-part, lorsque cela est possible, il vaut souvent mieux créer un fichier séquentiel puis le convertir plutôt que de tenter de garnir un fichier indexé directement. Les temps de traitement vont du simple au décuple, voire plus sans problème.

Traitement typique d'historisation

Une historisation consiste à sélectionner dans un fichier des enregistrements pour les écrire dans un fichier historique, et les supprimer du fichier original.

Là encore, les langages évolués permettent d'écrire facilement une séquence du type:

```
lire enregistrement
si à_historiser alors
    écrire enregistrement (en historique)
    détruire enregistrement
fin si
```

Mais un tel traitement est évidemment une catastrophe sur le plan des performances.

La bonne solution consiste à générer des fichiers séquentiels et à les convertir. Le rapport des temps de traitement va de 1 à 5 ou 10 sans problème (selon le nombre d'index). Par contre, cette méthode exige de la place disque.

Questions

? ?

? ?