Université de NANCY I
U.E.R. de Sciences Mathématiques

Centre de Recherche
en Informatique de Nancy

EFFICACITE DU TRI DANS LE CONTEXTE DE MEMOIRE VIRTUELLE

THESE

présentée pour l'obtention du diplôme de

DOCTEUR INGENIEUR

(spécialité informatique)

par

Patrick SECHET

soutenue publiquement le 04 - janvier 1984

Jury:

Président ..... M. J.C.DERNIAME

Examinateurs..... M. P.Y.CUNIN
M. D.COULON
M. J.P.TREUIL

Patrick SECHET
EMBRAPA (DMQ/DSI)
Supercenter Venâncio 2000
70333 Brasília (DF)

SOMMAIRE

RESUME

Le présent travail a pour objet la recherche d'une méthode de tri, principalement destinée au classement selon une clé secondaire de grands fichiers non conventionnels. La recherche de l'efficacité du programme dans un contexte de mémoire virtuelle, a conduit à une étude détaillée du comportement des algorithmes de tri, tant interne qu'externe, vis à vis de cette configuration particulière.

La sélection de quelques modèles de classification, bien adaptés à priori, est opérée après un examen général des algorithmes connus comme étant les plus efficaces. Une étude comparative des méthodes sélectionnées s'appuie sur l'évaluation des durées de chaque étape du traitement, et en particulier du temps passé aux échanges de pages. Les résultats analytiques obtenus mettent en évidence un seuil, pour la taille du fichier d'entrée, au delà duquel le tri exécuté selon une philosophie de formation et fusion de listes partielles devient plus économique que la méthode basée sur une simple philosophie de tri interne.

Le logiciel, projeté pour offrir un maximum de possibilités à l'utilisateur (supports, blocages, formats, etc...), est opérationnel sur ordinateur IBM/370-135, sous DOS/VS.

Pour montrer l'efficacité du système obtenu, des comparaisons expérimentales, entre le choix retenu et les autres possibilités de tri mises à la disposition d'un utilisateur de fichiers non conventionnels, ont été réalisées, dans le cas particulier d'un fichier organisé selon une stratégie de type "B-tree".

Finalement, un rapide examen du comportement de la méthode de tri vis à vis des différentes méthodes existantes de gestion de mémoire, ainsi qu'un exposé des méthodes d'adaptations d'autres algorithmes à l'environnement particulier de mémoire virtuelle, permettent de démontrer la relative stabilité des conclusions de ce travail dans des situations variées.

LISTE DES FIGURES

Chapitre 2

Chapitre 3

Chapitre 4

Chapitre 5

1. INTRODUCTION

Le traitement de l'information par des procédés électroniques est progressivement devenu une nécessité de toutes les activités humaines, depuis les tâches commerciales les plus routinières jusqu'aux recherches scientifiques les plus sophistiquées. La croissante expansion de cette activité exige en particulier la formation, initiale et permanente, d'équipes de professionnels efficaces dans ce domaine, ce qui demande un investissement élevé.

Etant entendu que les pertes de temps destinées à la maintenance des systèmes d'application sont particulièrement importantes, on s'attache de plus en plus à développer des logiciels de gestion de données à la fois plus complets et plus universels, dans le but de réduire le temps passé aux travaux de codification les plus fastidieux, ceci en utilisant, en particulier, la ressource de l'indépendance des données.

L'élaboration de notre travail a pour principal objectif la définition d'une méthode de tri, s'appuyant sur l'utilisation d'index, adaptée à un tel logiciel de gestion de données. Ajoutée à l'étude du logiciel lui-même, présentée par ailleurs (Costa, 1980), cette étude apparaît comme l'un des aspects d'un ample projet, dont la finalité est d'offrir à l'utilisateur les services d'un ensemble d'outils appropriés pour l'administration des données, dans toutes les applications du traitement informatique.

La recherche de la performance pour la méthode de tri, principalement dans le cas de grands fichiers, reste l'objectif fondamental lors du choix des processus et de la valeur des paramètres. Cette performance est évaluée en terme de temps de calcul dépensé en fonction de la taille du fichier à ordonner. Tout au long de cette étude, il s'avère nécessaire de formuler des hypothèses restrictives en ce qui concerne le contexte auquel est destiné l'implantation réelle. Ainsi l'accroissement de l'efficacité du programme développé est généralement obtenu par l'adaptation des modèles de traitement aux conditions matérielles de l'exécution, ce qui rend en compensation le programme plus dépendant du matériel pour lequel il a été conçu.

Il est apparu indispensable, dans une première partie, de présenter la majorité des algorithmes qui ont été étudiés. On distingue alors très vite deux philosophies différentes pour le tri, en fonction de ce que le fichier peut, ou non, être entièrement contenu dans la mémoire centrale. Le premier sous-chapitre est consacré à l'étude du groupe des algorithmes de tri interne; l'examen des algorithmes du second groupe est exposé immédiatement après. Une étude comparative des algorithmes proposés dans les deux cas a servi à orienter le choix des méthodes retenues, pour lesquelles des améliorations de détail ont été étudiées.

La deuxième partie contient l'élément le plus important de ce travail: l'examen du comportement des algorithmes choisis dans un contexte de mémoire virtuelle, par le biais d'une schématisation originale.

Dans cet objectif, le contexte dans lequel le programme a été développé est décrit en insistant plus particulièrement sur le phénomène de pagination, originaire du système de mémoire virtuelle. Le rôle prépondérant joué par les échanges de pages dans le calcul des temps nécessaires aux étapes successives du tri, conduit à réviser le choix des algorithmes. La fin de ce chapitre est consacrée au calcul empirique du plafond d'efficacité maximale du classement réalisé selon une méthodologie de tri interne, en tenant compte des échanges de pages. Pour ce dernier travail, il aura été nécessaire d'étudier minutieusement les stratégies possibles de distribution des "runs" sur les zones de mémoire auxiliaire.

L'implantation fait l'objet du chapitre suivant: plus qu'une présentation exhaustive du travail réalisé, on a cherché à donner une vision globale du processus complet, en soulignant les points les plus intéressants de la réalisation, tout en justifiant, dans la mesure du possible, les options prises au cours de l'analyse ou de la codification. Ainsi cette troisième partie constitue un véritable résumé du manuel d'analyse du système.

Des conclusions sont tirées d'après les comparaisons expérimentales réalisées entre la méthode choisie et les autres ressources de tri disponibles pour l'utilisateur de fichiers non conventionnels, et montrent que les résultats obtenus sont assez satisfaisants. De plus, des suggestions sont données pour des implantations postérieures qui auraient pour but d'améliorer, d'augmenter ou de rendre plus souples les ressources offertes par le système développé.

Une dernière partie enfin, permet de généraliser les résultats obtenus au cours de cette étude sur un algorithme particulier (de tri), dans une configuration particulière (de mémoire virtuelle), à un éventail plus large de situations.

Dans une première phase on a passé en revue les différents systèmes possibles de gestion de mémoire, en examinant l'incidence correspondante sur le choix de l'algorithme de tri: il apparaît alors que le choix réalisé reste parmi les meilleurs dans la plupart des contextes.

Dans un second temps, le comportement d'autres algorithmes vis à vis de la mémoire virtuelle a été examiné. L'exemple choisi d'un Système de Gestion de Bases de Données montre que le contexte de mémoire hiérarchisée impose l'adoption de méthodes déterminées (référenciation linéaire, indexs à plusieurs niveaux, structure modulaire, etc...) plutôt que d'autres (chaînage, par exemple), sous peine de voir les performances se dégrader par une pagination excessive.

2. LE PROBLEME DU TRI

La réorganisation d'un ensemble donné d'élements selon un ordre croissant ou décroissant, est une opération extrêmement fréquente dans le traitement des données. Bien que cette opération constitue le plus souvent une étape préliminaire d'un processus de recherche, on la retrouve dans les tâches les plus diverses, telles que création ou fusion de fichiers, mise à jour de fichiers séquentiels, édition d'états, etc....

Dans les applications scientifiques le tri revêt une importance moindre, dans la mesure où les collections de données généralement manipulées sont de dimensions plus réduites. On peut toutefois citer la construction de compilateurs ou encore la statistique descriptive, comme exemples de domaines qui utilisent ces techniques.

On peut enfin noter que l'étude du problème de tri est intéressante en soi, compte tenu de ce que les algorithmes de tri constituent de bons exemples de l'utilisation des méthodes de l'algorithmique.

2.1 DEFINITIONS ET NOTATIONS

Les divers concepts présentés dans ce sous-chapitre précisent la méthodologie adoptée, et servent de critères de comparaison entre les méthodes présentées.

2.11 Définitions générales

Etant donné un ensemble, appelé fichier ou table, de N objets:

$R(i), \quad i = 1, 2, 3, \ldots, N$,

appelés enregistrements, et en supposant que chaque enregistrement est associé à, au moins, une clé:

$K(i), \quad i = 1, 2, 3, \ldots, N$,

pour laquelle est définie une relation d'ordre quelconque; l'objectif du tri, par ordre croissant des clés, est de trouver une permutation:

$p(1), p(2), \ldots, p(N)$, des enregistrements, telle que:

$K(p(1)) < K(p(2)) < \ldots < K(p(N))$.

On convient d'appeler inversion dans la table un couple d'indices i et j, tel que $i < j$ alors que $K(i) > K(j)$. Dans ces conditions, on peut dire que trier un fichier consiste à en éliminer les éventuelles inversions rencontrées dans la comparaison des clés entre elles, en opérant la permutation des enregistrements correspondants.

2.12 Tri interne et tri externe

Vis à vis de l'occupation en mémoire centrale, un tri sera dit externe quand il n'est pas possible de stocker la totalité du fichier en mémoire et, par conséquent, le fichier est stocké sur des mémoires auxiliaires extérieures. Le tri est dit interne dans le cas contraire.

Cette distinction est fondamentale, tant en termes de philosophie de travail qu'en ce qui concerne les problèmes d'optimisation. En effet, dans le cas du tri externe, le fichier est divisé en lots, ou "runs", formés en mémoire centrale et provisoirement stockés en mémoire auxiliaire. Une étape supplémentaire est alors nécessaire pour réaliser la fusion des différents lots. En conséquence, le nombre des entrée-sorties devient un facteur prédominant dans la recherche de l'efficacité de l'algorithme, alors que l'on ne se préoccupe que de réduire le nombre de comparaisons et d'échanges d'enregistrements dans le cas du tri interne.

2.13 Tri direct et indirect

La majorité des algorithmes fait intervenir la permutation des éléments de la table de telle sorte que, lorsqu'il s'agit d'enregistrements importants, leur déplacement conduit à perdre un temps d'exécution non négligeable.

Une première méthode, tri par listes, consiste à conserver dans chaque enregistrement un pointeur vers l'enregistrement consécutif, de façon à organiser ainsi le fichier initial en une liste linéaire simple (voir figure 2.1). Les échanges sont alors réalisés avec les pointeurs, et non avec les enregistrements, tandis que la perte d'espace additionnel due à ce champ supplémentaire reste généralement insignifiante, en regard de la dimension d'un enregistrement.

Une autre méthode, tri avec utilisation de table d'adressage, fait intervenir un vecteur V, dont l'élément générique V(i) représente l'adresse i de l'enregistrement R(i) (voir figure 2.2). De cette façon les échanges se font seulement sur les entrées de la table d'adresses.

Il est important d'observer ici que l'association, dans la mémoire centrale, des clés des enregistrements avec les adresses de ceux-ci stockés sur des dispositifs de mémoire auxiliaire (à accès direct), doit permettre, dans de nombreux cas, d'éviter un tri externe (voir figure 2.3).

2.14 Efficacité du tri

Le concept d'efficacité est associé à celui de coût, lequel pour le traitement de l'information, est basé sur les temps d'exécution et l'espace-mémoire requis. On sait que ces deux aspects sont généralement en conflit, et donc qu'il convient de trouver le compromis le plus adéquat à répondre à l'objectif désigné.

Etant donné un algorithme, le temps nécessaire pour réaliser un tri dépend simultanément du nombre d'enregistrements à classer, de la disposition initiale des enregistrements et du programme qui implante

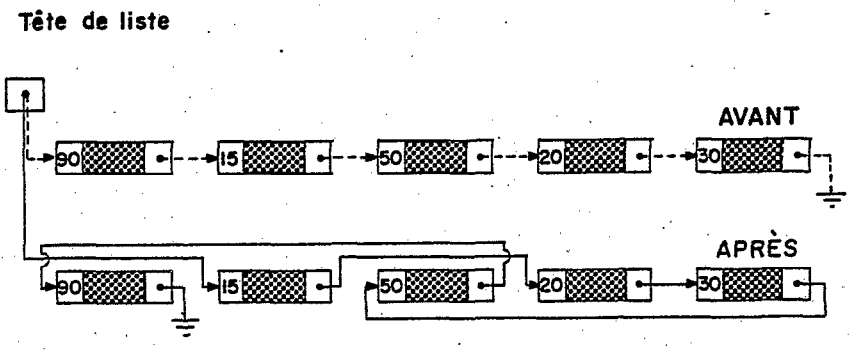


Fig. 2.1 Tri par liste linéaire

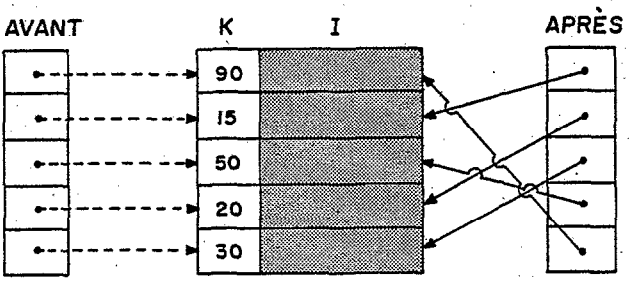


Fig. 2.2 Tri par table d'adresse

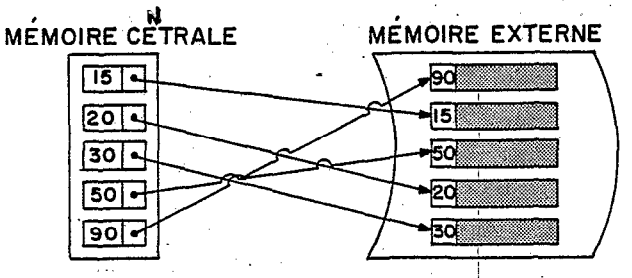


Fig. 2.3 Méthode pratique de tri indirect (APRÈS)

cet algorithme. En effet, il existe des algorithmes très rapides lorsque le fichier à classer est très peu désordonné, c'est à dire ne contient que peu d'inversions, tandis que d'autres algorithmes atteignent leur maximum d'efficacité lorsque les clés sont distribuées aléatoirement.

En pratique, l'efficacité en terme de temps d'exécution est conditionnée par le nombre de comparaisons à effectuer, dans la mesure où les mouvements des enregistrements peuvent être significativement réduits grâce aux méthodes indirectes. Le passage d'un algorithme trivial, de la classe N^2 (environ N^2 comparaisons), à un algorihtme efficace, exigeant à peine NlogN comparaisons, et exploitant la même idée de base, nécessite une modification fondamentale. En effet, pour meilleur que soit programmé un algorithme de la classe N^2, il sera plus lent qu'un algorithme de la classe NlogN, dans la mesure où N est suffisamment grand.

Les idées essentielles adoptées pour rendre les algorithmes efficaces, consistent à réduire le tri d'une table de N clés, à celui de deux sous-fichiers de respectivement p et N - p clés (principe de division, ou de récursivité), et de rendre p le plus près possible de N / 2 (principe d'équilibrage).

2.15 Stabilité du tri

Un algorithme de tri sera dit stable s'il restitue toujours, à la fin de l'opération, un couple de clés égales dans le même ordre relatif d'entrée.

Pour certaines applications, il peut s'avérer important de garantir cette propriété. Par exemple, si l'on désire ordonner une liste, par service, du fichier de personnel d'une entreprise déjà classé selon l'ordre alphabétique des employés, il suffira de trier selon le code du service si l'on utilise un algorithme stable, tandis qu'il serait nécessaire d'ordonner en fonction du code du service et du nom de l'employé si l'on emploie un algorithme non stable.

En raison de ce qu'ils peuvent exiger une comparaison supplémentaire dans la boucle interne, les algorithmes les plus efficaces ne sont généralement pas stables.

2.2 LA CODIFICATION DES ALGORITHMES

Il existe plusieurs façons de présenter un algorithme: celui-ci peut être écrit dans un langage naturel comme l'anglais (Knuth, 1973 & Tremblay et al., 1976), ou le français (Meyer et al., 1978); il peut aussi être directement traduit dans un langage de programmation de haut niveau, comme "Fortran", "Algol W" ou "PL/I" (Meyer et al., 1978), voire de bas niveau comme MIX (Knuth, 1973).

On a choisi de présenter les algorithmes en LA ("Linguagem para escrever Algoritmos"), langage proche de l'Algol mais avec les mots

clés en portugais, ce qui répond à une préoccupation de clarté et d'accessibilité.

Les principales caractéristiques de ce langage sont sommairement décrites ci-après: pour de plus amples détails, la lecture du texte original de l'auteur est recommandée (Queiroz, 1978).

2.21 Caractères de base de LA

En plus des lettres de l'alphabet, tant majuscules que minuscules, et des chiffres du système numérique décimal, on dispose de trois autres types de caractères de base. Les premiers sont les opérateurs, tant arithmétiques (+, -, *, /, **) comme de relation, ou booléens (<, <=, =, <>, >= et >). On utilise ':=' comme opérateur d'attribution. Les délimiteurs sont les parenthèses (ouvertes et fermées), les crochets (ouverts ou fermés) et les commentaires (ouverts et fermés), ces derniers représentés par les symboles '/*' et '*/', en plus de la virgule, du point du point-virgule, de l'apostrophe et du mot-clé 'FIM'.

Finalement les derniers caractères de base de LA sont les mot-clés, pour lesquels on utilisera les lettres minuscules dans les algorithmes et les lettres majuscules dans le texte. Il s'agit de: ALGORITMO (algorithme), ARRAY, ATE (jusqu'à), ENQUANTO (tant que), ENTAO (alors), FACA (fait), IMPRIMA (écrit), LEIA (lit), PARA (pour), PARA:, PARE. (termine), PASSO (incrément), RETORNE. (retourne), SE (si) et VA (va).

2.22 Eléments de base de LA

A partir des caractères de base de LA, on construit les éléments de base, qui sont les nombres, les variables, les "ARRAY", les fonctions de base et les expressions. Il existe trois types d'expression en LA: les expressions arithmétiques, les expressions logiques et les expressions alphanumériques, ces dernières font intervenir des chaînes de caractères.

2.23 Instructions de LA

En plus de l'instruction d'attribution, y compris l'attribution multiple, et des instructions d'entrée-sortie (à partir des mot-clés LEIA et IMPRIMA), le langage contient diverses instructions de contrôle.

L'instruction d'arrêt est normalement PARE.. On utilise RETORNE., dans le cas d'un algorithme appelé par un autre. Le "GOTO" inconditionnel est codifié "VA PARA", et on utilise des nombres pour la représentation des étiquettes. Le "GOTO" conditionnel est codifié "SE ... ENTAO", et réalise la rupture de séquence si le résultat de l'expression logique consécutive au mot "SE" est vrai. Enfin, il existe deux formes d'instructions itératives en LA: la première est codifiée "ENQUANTO ... FACA ..." et la deuxième est codifiée: "PARA: ... ATE ... PASSO ... FACA ...".

2.24 Autres caractéristiques de LA

On peut utiliser le groupage des instructions (bloc) au moyen du mot-clé "FACA". Ainsi, un algorithme écrit en LA, est constitué d'une séquence finie de commandes étiquetées, terminées par le délimiteur "FIM" commencée par un en-tête contenant le mot-clé "ALGORITMO", suivi d'un nom et, éventuellement, d'une liste de paramètres.

2.25 Exemples: PERMUTE et COMPARE

Comme on l'a vu (voir §2.11), la majorité des algorithmes réalise la classification d'une table par élimination progressive des inversions rencontrées. Il est donc nécessaire de disposer de deux subroutines applicables sur un couple de clés (K(i),K(j)), pour réaliser respectivement les opérations de comparaison et de permutation, algorithmes codés "COMPARE(I,J)" et "PERMUTE(I,J)".

Dans la codification des algorithmes de tri, on admet que:

- l'appel de la routine PERMUTE(i,j) effectue l'échange des valeurs des clés et des enregistrements dont les indices sont i et j;

- l'appel de la routine COMPARE(i,j) retourne une variable entière "COMP", qui vaut:

0 si K(i) < K(j),

1 si K(i) = K(j),

2 si K(i) > K(j).

Dans la pratique, la clé K(i) sera le plus souvent constituée de plusieurs champs de l'enregistrement R(i), obéissant à différents formats (alphanumérique, entier, réel, etc...). Par ailleurs, en fonction du type de tri considéré, la permutation de deux enregistrements pourra se concrétiser, soit par un simple échange de pointeurs, soit par l'échange des propres enregistrements.

Dans ces conditions, l'élaboration des subroutines COMPARE et PERMUTE devra être adaptée à chaque cas spécifique de tri: la codification de l'algorithme principal restera toutefois indépendante de ces dernières conditions. Le cas le plus simple de codification de ces algorithmes est lorsque la clé est numérique et le tri est direct:

```
algoritmo COMPARE(I,J); /* le résultat est COMP */
 01. se (K(I) < K(J)) então faça (COMP := 0; retorne.);
 02. se (K(I) = K(J)) então faça (COMP := 1; retorne.);
 03. faça (COMP := 2; retorne.) fim
```

```
algoritmo PERMUTE(I,J); /* les enregistrements R sont globaux */
 01. AUX  := R(I);
 02. R(I) := R(J);
 03. R(J) := AUX;
 04. retorne. fim
```

2.3 LE TRI INTERNE

Cette partie est entièrement consacrée à la présentation comparative des algorithmes de tri interne. On ne prétend pas ici traiter exhaustivement du sujet: au contraire, on tente plutôt de souligner la philosophie générale de chaque méthode, en montrant chaque fois comment l'on peut aboutir à un algorithme performant.

2.31 Les idées de base

Il existe plusieurs philosophies qui peuvent être mises en oeuvre pour réaliser le classement d'une table. Chacune fournit, d'immédiat, un algorithme trivial qui peut généralement être transformé en un algorithme efficace au moyen d'artifices tels que la récursivité ou la structuration des données, en arbre binaire, par exemple. On peut citer:

- le tri par énumération: chaque clé est comparée avec toutes les autres, le nombre de clés plus petites qu'une clé déterminée, donne la position définitive de celle-ci;

- le tri par insertion: une clé est séparée tandis que les autres sont ordonnées, la clé isolée est alors insérée dans l'ensemble ordonné;

- le tri par sélection: on recherche la plus petite (ou la plus grande) clé de l'ensemble, que l'on positionne en tête de la collection. On sélectionne ensuite la seconde plus petite clé et on itère le procédé;

- le tri par échanges: dans cette méthode on réalise, par étapes successives, l'élimination des inversions, en effectuant la permutation de tous les couples d'éléments rencontrés en désordre. Le processus se poursuit jusqu'à ce qu'il n'existe plus d'inversions;

- le tri par fusion: dans ce procédé on cherche à diviser le fichier en sous-fichiers, qui sont ordonnés plus rapidement, et on réalise ensuite la fusion de ces sous-fichiers;

- le tri par répartition: cette méthode n'utilise pas de comparaisons de clés entre elles, mais suppose que les clés sont formées d'un ensemble fini d'éléments. Les clés sont alors classées successivement en accord avec les composants d'ordre supérieur (chiffres les plus à droite), avec un algorithme stable.

2.32 Tri par énumération

Dans cette méthode on compare chaque paire de clés, tout en mémorisant, pour chaque clé, le nombre de clés qui lui sont inférieures. Lorsque toutes les comparaisons ont été effectuées, ce nombre (augmenté d'une unité) indique la position finale de la clé dans la collection ordonnée.

On observe que cette méthode ne fait intervenir aucun mouvement d'enregistrements, dans la mesure où la liste des "compteurs"

fonctionne exactement comme une table d'adresses. Cependant chaque clé est comparée avec les N - 1 autres, et en conséquence l'algorithme nécessite:

N * (N - 1) / 2

comparaisons, et se trouve ainsi être de la classe de N^2.

En admettant quelques restrictions sur les clés, on aboutit finalement à un algorithme performant: en effet, si l'on suppose que les clés prennent seulement des valeurs entières comprises dans un intervalle (u,v) d'amplitude raisonnable, on peut compter le nombre de clés égales à u, u + 1, u + 2, ..., v d'où la position d'une clé K = u + i est obtenue en sommant le nombre de clés égales à u, u + 1, u + 2, ... , u + (i - 1).

```
algoritmo M(N,U,V); /* tri par énumération */
 01. array CONT(N);
 02. para: I := 1 até N faça (CONT(I) := 0);
 03. para: J := 1 até N faça (J := K(I), CONT(J) := CONT(J) + 1);
 04. para: I := U + 1 até V faça CONT(I) := CONT(I) + CONT(I - 1);
 05. para: J := N até 1 passo (-1) faça (I := CONT(K(J)),
                                         S(I) := R(J),
                                         CONT(K(J)) := I - 1);
 06. pare. fim
```

La seconde instruction réalise l'initialisation des compteurs: la troisième totalise le nombre de clés K(I) dont la valeur est J. Le nombre de clés plus petites qu'une clé donnée est obtenu au cours de la quatrième instruction, en sommant les valeurs des compteurs; il suffit alors, pour terminer, de positionner les clés dans le fichier de sortie conformément à la séquence des compteurs.

Cette méthode de classement peut être très efficace, étant donné qu'elle ne fait intervenir aucune comparaison de clés, et que le nombre des transferts d'enregistrements est minimal. Cependant, les restrictions imposées pour l'intervalle de définition des clés limitent beaucoup l'application de cette méthode.

2.33 Tri par insertion

Dans cette méthode les clés sont prises en compte chacune séparément, en les insérant une par une dans le sous-fichier constitué par les clés déjà ordonnées, lequel croît ainsi d'une unité.

L'inconvénient majeur dé cette méthode d'insertion directe vient de ce qu'il est nécessaire de déplacer (d'une position en moyenne) J / 2 enregistrements, au moment d'inclure l'enregistrement R(J + 1), dans l'ensemble ordonné des J précédents. Ainsi l'utilisation des méthodes indirectes devient-elles particulièrement intéressantes dans ce cas.

Néanmoins, on observe que pour l'opération élémentaire ci-dessus définie, J / 2 comparaisons sont nécessaires en moyenne (J dans le pire des cas, c'est à dire lorsque le fichier d'entrée est classé dans

l'ordre exactement inverse de celui recherché), et pour l'ensemble du fichier on obtient:

$$(1 + 2 + \ldots + N - 1) / 2 = N * (N - 1) / 4$$

comparaisons en moyenne.

Dans le cas idéal du fichier initial ordonné, le nombre de comparaisons est réduit à N, étant donné que chaque clé n'a besoin que d'une comparaison pour être insérée dans sa position définitive. Le tri par insertion est en conséquence peu performant en général, mais devient cependant très efficace pour la classification d'une table déjà partiellement ordonnée. Un facteur essentiel de perte de temps dans le classement par insertion, vient de ce que chaque enregistrement est déplacé d'une seule position à chaque opération, et va ainsi tarder à rejoindre sa position définitive; on peut alors penser à insérer l'élément d'indice J dans la sous-liste des éléments d'indices respectifs:

$$I - H,\ I - 2H,\ I - 3H,\ \ldots$$

au lieu du sous-fichier des J - 1 enregistrements précédents.

On obtient de cette manière une table de H suites disjointes ordonnées: cette étape est alors répétée en choisissant une valeur H' inférieure à H, avec laquelle le tri s'accélère, compte tenu de la propriété caractéristique du tri par insertion. La méthode de SHELL emploie une séquence décroissante:

$$H(I),\ H(I - 1),\ \ldots\ ,\ H1$$

avec $H1 = 1$ et $H(L + 1) = 3 * H(L) + 1$

la valeur maximale L_{max} de L étant choisie de telle façon que L_{max} soit le premier élément de la séquence définie précédemment, qui soit égal ou supérieur à N / 9 (calculé par défaut).

La figure 2.4 illustre ce processus, bien que le couple de valeurs choisies de H (1 et 3) ne satisfasse pas la relation ci-dessus, pour convenance de dessin.

```
algoritmo S(N); /* tri par insertion: SHELLSORT */
 01. faça (L := 1, enquanto (L < N / 9) faça L := 3 * L + 1);
 02. enquanto (L > 0) faça ( para: J := L + 1 até N
                             faça (I := J - L, COMPARE(I,I + L),
                                   enquanto (I > 1 & COMP = 2)
                                      faça (PERMUTE(I,I + L),
                                            I := I - L,
                                            COMPARE(I,I + L))),
                                   L := L / 3);
 03. pare. fim
```

La première instruction permet de déterminer le plus grand incrément de liste N_{max}; la deuxième instruction est constituée d'une double boucle, dont l'objectif est de positionner l'enregistrement R(j), dans la sous-liste, une fois fixée la valeur de l'incrément L.

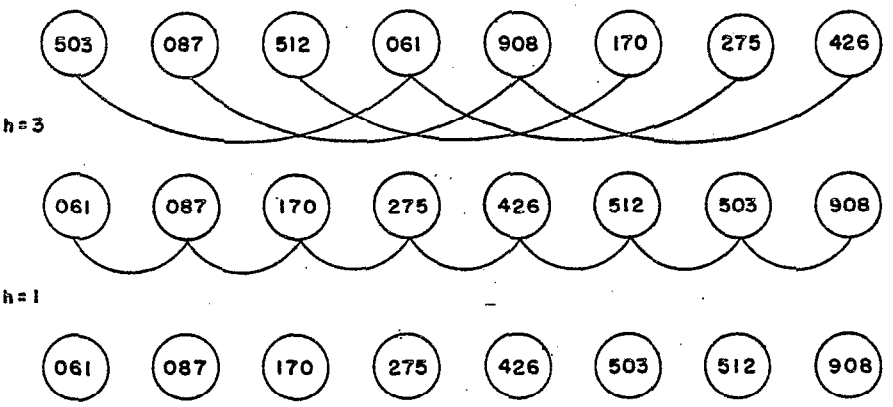
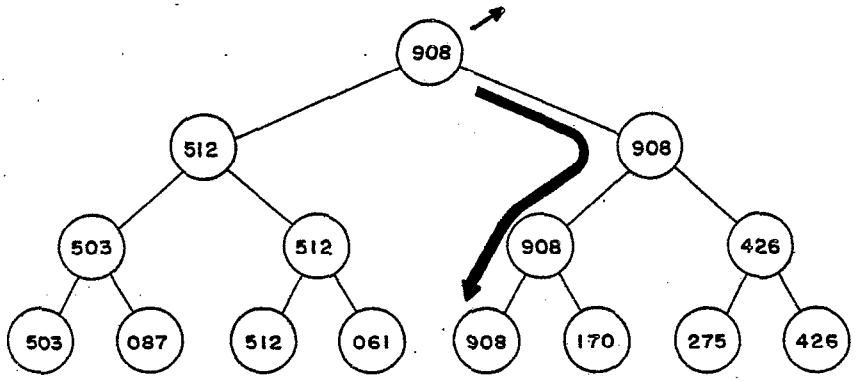
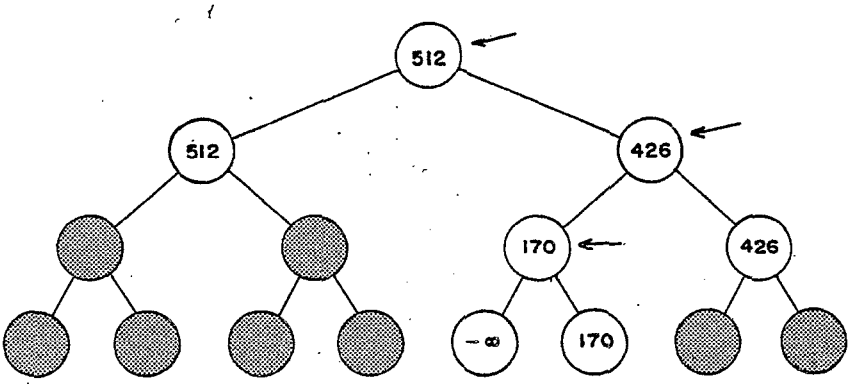


Fig. 2.4 Tri par la méthode de Shell



a) Sélection de la plus grande clé (après N-1 comparaisons)



b) Sélection de la deuxième clé (après $\lceil \log_2 N \rceil$ comparaisons)

Fig. 2.5 TRI PAR SELECTION EN ARBRE BINAIRE

On démontre que le nombre de mouvements entraînés est proportionnel à $N^{1,25}$, de telle forme que cet algorithme peut être considéré comme performant, jusqu'à 100 000 enregistrements environ.

Il est important toutefois de noter qu'un algorithme avec cette philosophie de comparaison d'éléments éparses dans une table, devra faire intervenir de nombreux échanges de pages, pour un système utilisant la mémoire virtuelle. Il y a lieu de remarquer également que cet algorithme n'est pas stable.

2.34 Tri par échanges

La méthode consiste à éliminer, par étapes, les inversions rencontrées dans la permutation initiale du fichier. L'application triviale de la méthode, qui élimine successivement les inversions constatées entre clés consécutives, est connue sous le nom de "Bubble sort", ainsi appelée en raison de l'image qu'elle donne de faire 'flotter' les clés les plus 'légères'.

Ce premier algorithme, malgré les quelques raffinements qui peuvent y être apportés, ne permet pas de fournir une implantation efficace de tri, en raison de la manière très locale d'aborder le problème: en effet, chaque clé se trouve comparée avec ses voisines immédiates, et nécessite ainsi de nombreuses étapes pour rejoindre sa position définitive dans la table ordonnée.

L'algorithme présenté ci-après (Hoare, 1961) élimine cet inconvénient, en réalisant une partition du fichier initial en deux sous-fichiers à partir d'une clé pivot, de telle sorte que toutes les clés situées à gauche soient plus petites, et que toutes les clés situées à droite soient plus grandes que la clé pivot. Bien évidemment, la clé pivot occupe alors sa position finale dans le fichier, et le tri avec cet algorithme devient récursif.

Pour réaliser cette division du fichier, il suffit de parcourir la table à partir de la droite, jusqu'à ce que l'on rencontre un élément dont la clé est supérieur à la clé pivot: une fois rencontré, on parcourt la table à partir de la gauche cette fois, jusqu'à ce que l'on rencontre un élément dont la clé soit supérieur à la clé pivot. En permutant les deux éléments ainsi rencontrés on élimine une inversion; le procédé est alors poursuivi jusqu'à ce que les positions parcourues à droite et à gauche se croisent (à l'endroit de la clé pivot), après un nombre de comparaisons effectuées égal à la dimension du fichier.

La codification proposée ci-après réalise l'implantation de cet algorithme, en choisissant comme clé pivot la première clé du fichier.

```
algoritmo Q(N); /* tri par échanges: QUICKSORT */
 01. array K(N), STK(20); /* pile double */
 02. faça (TOP := 0, ITEM := (1,N), INSTK);
 03. enquanto (TOP > 0) faça (OUTSTK, (LB,UB) := ITEM,
                              enquanto (UB > LB) faça PARTICAO);
 04. pare. fim
```

```
algoritmo PARTICAO; /* réalise la séparation en deux sous-fichiers
                       et empile le plus grand */
 01. faça (I := LB, J := UB, T := K(I)); /* T est le pivot */
 02. enquanto (T < K(J)) faça J := J - 1; /* à droite */
 03. se (J < I) então faça (K(I) := T, và para 8);
 04. faça (K(I) := K(J), I:= I + 1);
 05. enquanto (K(I) < T) faça I := I + 1; /* à gauche */
 06. se (J.> I) então faça (K(J) := K(I), J := J - 1, và para 2);
 07. faça (K(J) := T, I := J); /* nouveau pivot */
 08. se (I - LB < UB - I) então faça (ITEM := (I + 1, UB), INSTK,
                                      UB := I - 1, và para 10);
 09. faça (ITEM := (LB,I - 1), INSTK, LB := I + 1);
 10. retorne. fim
```

```
algoritmo INSTK; /* insertion d'un item dans la pile */
 01. se (TOP > 20) então faça (imprima ('STACK OVERFLOW'),
                               retorne.);
 02. faça (TOP := TOP + 1, STK(TOP) := ITEM);
 03. retorne. fim
```

```
algoritmo OUTSTK; /* récupération d'un item de la pile */
 01. se (TOP < 0) então faça (imprima ('STACK UNDERFLOW'),
                              retorne.);
 02. faça (ITEM := STK(TOP), TOP := TOP - 1);
 03. retorne. fim
```

Dans cette version, on utilise une pile double pour stocker les limites inférieures et supérieures des sous-fichiers à ordonner. En choisissant toujours de trier en premier le plus petit des deux fichiers obtenus, on peut garantir que la taille de la pile ne sera pas supérieure à $\log_2 N$.

Codifié de cette façon, "Quicksort" est près de deux fois plus rapide que ses concurrents, pour un grand fichier dont les éléments sont distribués aléatoirement (Knuth, 1973). Cependant, il existe une possibilité de dégradation de l'algorithme lorsque le fichier est déjà partiellement ordonné dans sa configuration initiale. Cette dégradation peut aller jusqu'à conduire à un algorithme de la classe de N^2, quand le fichier d'entrée est totalement ordonné.

Il n'existe aucune méthode générale qui permette d'éliminer totalement le risque de dégradation (on pourra en effet toujours construire une table qui nécessite une durée proportionnelle à N^2 pour être triée). Néanmoins, il existe diverses possibilités (Sedgewick, 1975) de rendre le pire cas hautement improbable de telle sorte que "Quicksort" constitue une méthode performante dans tous les cas.

On observe finalement que cet algorithme n'est pas stable, et qu'il n'existe aucun moyen pour le rendre stable.

2.35 Tri par sélection

La méthode la plus simple se ramène à une séquence de N - 1 pas, où le pas élémentaire 'j' consiste à sélectionner l'élément dont la clé est la plus grande de l'ensemble:

K(1), K(2), ... , K(j - 1), K(j) ,

et de positionner celui-ci au rang j.

Bien évidemment, le pas j requiert j - 1 comparaisons, et, en conséquence, le tri de la collection des N clés nécessitera un nombre de comparaisons égal à:

$$(N - 1) + (N - 2) + \ldots + 2 + 1 = N * (N + 1) / 2 .$$

Afin que cet algorithme passe de la classe N^2 à la classe NlogN, il convient de mieux exploiter l'information collectée par les comparaisons à chaque étape. On peut, par exemple, garder un pointeur vers la position de l'avant dernière clé rencontrée au cours du pas 'j'. Cette information pourra alors être exploitée au cours de l'étape suivante, pour économiser le nombre de comparaisons (près de la moitié).

Pour obtenir une amélioration plus sensible, on doit utiliser une structure de données telle que soit conservée le résultat des comparaisons successives. L'idée d'un arbre binaire peut se déduire d'une analogie avec une compétition par élimination directe. Chaque 'match' (comparaison de deux clés) fournit un 'vainqueur' (élément dont la clé est supérieure à celle de l'autre), qui pourra alors continuer la compétition (monter dans l'arbre). Une fois établi l'arbre (voir figure 2.5), N - 1 comparaisons suffisent à séparer la plus grande clé du fichier; celle-ci est ensuite substituée par une clé ("Low value"), plus petite que toutes les autres clés présentes, de telle sorte que la réorganisation de l'arbre ne nécessite pas plus de comparaisons que celles faites sur une seule branche (c'est à dire $\log_2 N$).

Un tel algorithme fait ainsi intervenir N + NlogN comparaisons, et appartient donc à la classe des algorithmes efficaces, en terme de durée. Malheureusement, il nécessite un espace double pour stocker les N - 1 noeuds et les N feuilles de l'arbre.

L'idée présentée par FLOYD en 1962 (Knuth, 1973), et perfectionnée plus tard (Williams, 1964), consiste à simuler l'arbre binaire sur le seul vecteur formé par les clés distribuées séquentiellement dans la mémoire.

Dans l'algorithme correspondant ("Heapsort"), le 'père' d'un noeud d'indice K est le noeud d'indice K / 2 (par défaut), et, naturellement, les 'fils' d'un noeud situé dans la position K sont les noeuds qui occupent les positions 2K et 2K + 1.

La codification présentée ci-dessous utilise une subroutine (REORG) pour réorganiser l'arbre après une sélection: cette même subroutine est appelée dans la première partie du programme principal,

lors de la crêation de l'arbre. A la fin du travail les enregistrements se trouvent ordonnés 'in situ', selon une séquence ascendante (ce qui est obtenu par la permutation, à chaque pas, des éléments extrêmes).

```
algoritmo H(N); /* tri par sélection: HEAPSORT */
 01. array K(N);
 02. para: I := entier (N / 2) até 1 passo (-1) faça REORG(I,N);
 03. para: I := N até 2 passo (-1) faça (PERMUTE(1,I),
                                         REORG(1,I));
 04. pare. fim
```

```
algoritmo REORG(I,J); /* réorganise entre les indices i et j */
 01. L := I;
 02. ESQ := 2 * L;
 03. se (ESQ > J) então vá para 9;
 04. faça (LMAX := ESQ, DIR := ESQ + 1);
 05. se (DIR > J) então vá para 7;
 06. faça (COMPARE(ESQ,DIR), se (COMP = 0) então LMAX := DIR);
 07. faça (COMPARE(LMAX,L), se (COMP < 1) então vá para 9);
 08. faça (PERMUTE(L,LMAX), L := LMAX, vá para 2);
 09. retorne. fim
```

Il est important de souligner qu'il n'a été fait aucune hypothèse concernant la permutation initiale des clés et, en conséquence, l'algorithme reste de la classe NlogN dans tous les cas. Toutefois on perçoit facilement que cet algorithme ne sera pas performant dans une configuration de mémoire virtuelle, en raison du caractère pseudo-aléatoire de la sélection des couples de clés qui doivent être comparées , et éventuellement échangées. Par ailleurs cet algorithme n'est pas stable non plus.

2.36 Tri par fusion

Etant donné deux listes ordonnées d'enregistrements:

$$R_1, R_2, \ldots, R_n \quad \text{et} \quad R'_1, R'_2, \ldots, R'_m ,$$

le processus de fusion ("merge") de ces deux listes permet d'obtenir une nouvelle liste ordonnée:

$$R''_1, R''_2, \ldots, R''_{n+m}.$$

Ce procédé peut se généraliser pour effectuer la fusion multiple de K listes ("K-way merge"); de plus, en réalisant l étapes successives, on peut obtenir une liste ordonnée de lkn enregistrements, à partir de lk sous-listes élémentaires de n enregistremments. Etant entendu qu'une table contenant un élément unique est intrinsèquement ordonnée, on obtient ainsi un procédé original de tri.

Cet algoritme est de la classe NlogN, mais possède l'inconvénient de nécessiter un espace double pour stocker les fichiers intermédiaires. Pour tourner cette difficulté, on peut utiliser une méthode de tri indirect, comme le tri par liste linéaire, par exemple.

Dans l'algorithme présenté ci-après, la fusion est réalisée à partir de sous-listes rencontrées dans la table: on recherche ainsi à profiter du degré initial d'ordre du fichier. Chaque enregistrement contient un pointeur L et il existe deux pointeurs auxiliaires L(0) et L(N + 1), qui fonctionnent comme des têtes de listes déjà fusionnées. Un pointeur négatif marque la fin d'une sous-liste connue comme étant ordonnée; le pointeur 0 marque la fin de liste entière. Enfin, une fois le tri terminé, L(0) est l'indice de l'enregistrement dont la clé est la plus petite.

```
algoritmo L(N) /* tri par fusion: LIST MERGE SORT */
 01. array K(N + 2), L(N + 2);
 02. faça (L(0) := 1, L(N + 1) := 2,
          para: I := 1 até N - 2 faça L(I) := -(I + 2),
          (L(N - 1), L(N)) := 0);
 03. faça (S := 0, T := N + 1, P := L(S), Q := L(T),
          se (Q = 0) então vá para 11);
 04. faça (COMPARE(P,Q), se (COMP = 2) então vá para 8);
 05. faça (/L(S)/ := P, S := P, P := L(P),
          se (P < 0) então vá para 4);
 06. faça (L(S) := Q, S:= T);
 07. faça (T := Q, Q := L(Q),
          enquanto (Q > 0) faça (T:= Q, Q := L(Q)), vá para 10);
 08. faça (/L(S)/ := Q, S := Q, Q := L(Q),
          se (Q > 0) então vá para 4);
 09. faça (L(S) := P, S := T, T := P, P := L(P),
          enquanto (P > 0) faça (T := P, P := L(P)));
 10. faça (P := -P, Q := -Q,
          se (Q = 0) então faça (/L(S)/ := P, /L(T)/ := 0,
                                 vá para 3),
          vá para 4);
 11. pare. fim
```

Dans cette codification, on utilise la teminologie /L(Y)/, avec la signification suivante:

se (L(Y) < 0) então /L(Y)/ := -(valeur absolue de X);
se (L(Y) > 0) então /L(Y)/ := +(valeur absolue de X).

L'algorithme ainsi codé possède une efficacité intermédiaire entre celle de "Quicksort" et celle de "Heapsort" et, dans la mesure où il ne présente aucune possibilité de dégradation eu égard à la permutation initiale du fichier à ordonner, c'est un algorithme très intéressant. Il pourra par conséquent être mis en oeuvre comme un excellent algorithme de tri interne: il possède de surcroît l'avantage d'appartenir au groupe restreint des algorithmes stables.

2.37 Tri par distribution

L'idée de tri par distribution a cela d'originale qu'elle ne requiert aucune comparaison des clés entre elles: la propre valeur de la clé est utilisée pour déterminer une zone dans laquelle l'enregistrement correspondant sera placé.

Bien entendu, ces algorithmes nécessiteront d'un espace supplémentaire pour stocker les diverses classes d'équivalence ainsi formées: c'est un autre exemple du fameux compromis espace-temps, et dans ce cas également les méthodes de tri indirectes seront utiles.

On suppose que chaque clé K_i est formée de p composantes:

a(p - 1), a(p - 2), ... , a(2), a(1);

(les a(i) pourront être des lettres ou des chiffres, par exemple), chaque composante pouvant prendre une valeur quelconque dans l'intervalle (0,M), (M = 25, dans le cas des lettres de l'alphabet).

Dans ces conditions, la philosophie utilisée consiste à placer chacune des clés dans une des M files auxiliaires, en fonction de la valeur de la première composante de la clé (a(p)). En rendant les clés à leur espace initial, dans la séquence croissante des files, on termine la première étape avec la table triée d'après la première composante des clés.

L'algorithme termine après p étapes et utilise l'espace intermédiaire nécessaire pour stocker (M + i) * N enregistrements.

La version présentée par Seward en 1954 (Knuth, 1973) utilise seulement l'espace correspondant à 2N enregistrements et M compteurs, en opérant une étape supplémentaire pour déterminer le nombre d'éléments qui seront placés dans chaque file et en allouant l'espace auxiliaire en fonction du résultat obtenu.

L'utilisation de l'allocation en chaîne permet de réduire la réquisition d'espace à:

(R + p) * N + 2pM ,

où R est l'espace occupé par un enregitrement et p celui requis par un compteur (p << R). L'algorithme présenté ci-dessous ("Radix list sort") se compose de deux routines. La première distribue une file initiale en plusieurs sous-files, tandis que la seconde concatène diverses files enchaînées, pour obtenir une file unique.

```
algoritmo R(N); /* tri par distribution */
 01. array K(N), L(N), A(N); /* L_i = compteur, A_i = adresse */
 02. para: I := 1 até P passo 1 faça DISTRIBUE(I).
 03. pare. fim
```

```
algoritmo DISTRIBUE(I); /* réalise une étape de distribution */
 01. array RI(M + 1), RF(M + 1);
 02. Q := A(K(N)); /* Q pointe le dernier enregistrement */
 03. para: J := 0 até M passo 1 faça (RI(J) := A(RF(J)),
                                      RF(J) := *);
 04. S := AI(Q); /* AI est le Ième "digit" de la clé K(Q) */
 05. faça (L(RI(S)) := Q, RI(S) := Q);
 06. se (I = 1) então faça (para: J := 2 até N passo 1
                            faça (se (Q = A(K(J))) então
                                  faça (Q := A(K(J - 1)),
                                        vá para 4)));
 07. se (I > 1) então faça (Q := L(Q), se (P <> *) então vá para 4);
 08. faça (CONCATENA, Q := RF(0));
 09. retorne. fim
```

```
algoritmo CONCATENA; /* réorganise les files en une file unique */
 01. I := 0;
 02. Q := RI(I);
 03. faça (I := I + 1, L(Q) := *, vá para 6);
 04. se (RF(I) = *) então vá para 3;
 05. faça (L(Q) := RF(I), vá para 2);
 06. retorne. fim
```

La figure 2.6 permet d'accompagner l'algoritme, c'est à dire la construction des files, pour la série des clés de D.E.KNUTH (Knuth, 1973). L'algorithme utilise deux pointeurs RI et RF pour le début et la fin de chaque zone. Un pointeur vide est noté *. On peut observer que la méthode de distribution des enregistrements dans les files est réalisée à partir du "digit" le moins significatif en premier: ceci garantit un nombre maximal de M files qui seront liées p fois.

Avec quelques améliorations de détail dans l'implantation, la méthode peut atteindre un temps de tri proportionnel à N, si les clés sont uniformément réparties, proportionnel à N^2 dans les pires cas. Elle possède l'avantage d'être stable, et l'inconvénient de faire quelques restrictions concernant la nature des clés à ordonner.

2.38 Comparaison des diverses méthodes

Il existe d'autres méthodes performantes de tri qui n'ont pas été décrites dans ce chapitre, parce qu'elles sont très semblables à celles qui sont décrites. Par exemple, "Adress calculation sort" ou "Multiple list insertion sort", sont deux méthodes hybrides de tri par insertion et par distribution, dans la classe desquels le "Radix list sort" (voir §2.37) peut être considéré comme le meilleur exemple. De la même façon, "Treesort" est très semblable à "Heapsort" (voir §2.35), comme le "Radix exchange sort" utilise les mêmes idées de tri par échanges, données pour "Quicksort" (voir §2.34).

Divers auteurs donnent des études comparatives de certaines de ces méthodes, basées sur une expérimentation (Meyer et al., 1978; Scowen, 1965 & Loeser, 1974), ou sur une analyse théorique.

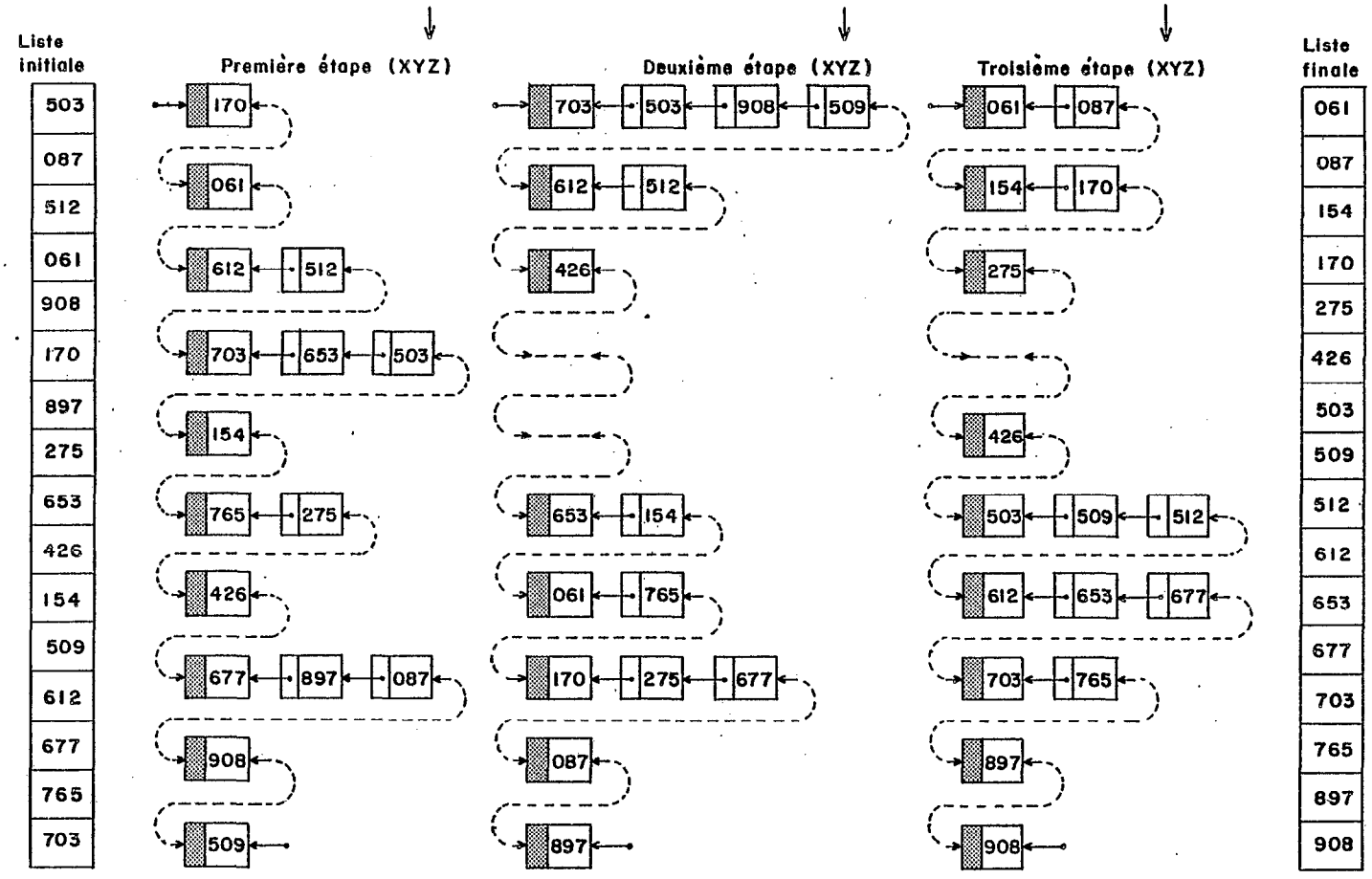


Fig. 2.6 EXEMPLE DE TRI PAR DISTRIBUTION ('RADIX LIST SORT')

Le tableau 2.1 ci-après, lequel utilise en particulier les résultats de D.E.KNUTH, a pour objectif de mettre en évidence les critères les plus relevants de comparaison entre les méthodes, dans l'optique de l'implantation de l'une d'entre elles comme méthode performante et universelle. En particulier les deux dernières colonnes de ce tableau, c'est à dire les restrictions imposées par la méthode au type de la clé et la possible adaptation de la méthode pour un système de mémoire virtuelle, sont primordiales pour le choix de la méthode de tri interne. Le seul inconvénient de la méthode "Quicksort" apparaît dans la colonne du temps maximal de tri, ce que l'on peut parfaitement négliger, eu égard au peu d'influence des opérations internes à la mémoire en comparaison de celles d'entrée-sorties engendrées par les échanges de pages, par exemple (voir chapitre 3). De plus, de nombreuses modifications sont connues (Sedgewick, 1975, en particulier) pour réduire la probabilité d'apparition du cas de dégradation de l'algorithme.

2.39 Amélioration de l'algorithme de tri interne

Les améliorations qui peuvent être apportées à l'algorithme "Quicksort", tel qu'il est codifié au paragraphe 2.34, concernent la durée du tri, particulièrement dans le pire des cas de fichier initial déjà classé. En effet, une économie substantielle d'espace paraît difficile, étant entendu que l'algorithme, dans sa présentation originale (c'est à dire avec l'empilage du plus grand sous-fichier), garantit déjà que l'espace nécessaire n'est pas supérieur à $N + 2\mu \log N$ fois l'espace requis pour stocker un seul enregistrement (μ, très petit, est le rapport entre l'espace nécessaire au stockage d'un pointeur et celui nécessaire au stockage d'un enregistrement).

En ce qui concerne l'élimination du pire cas, diverses méthodes ont été testées, toutes ayant pour objectif de choisir un pivot susceptible de réaliser une bonne partition de la table, autrement dit un élément dont la clé soit représentative de la valeur moyenne des clés de l'ensemble. Un compromis entre une approximation raisonnable, obtenue après une rapide recherche, ou une estimation plus fine au prix d'une implantation plus dispendieuse au point de vue temps, conduit à choisir comme clé pivot pour l'ensemble:

$K(i), K(i + 1), \ldots, K(j - 1), K(j)$;

la valeur médiane de l'échantillon:

$K(i), K(i / 2 + j / 2), K(j)$.

Tant cette dernière recherche que la propre 'architecture' du "Quicksort" (et en particulier la gestion de la pile double), rend la méthode peu performante pour les petits fichiers. Ceci peut apparaître comme un défaut grave, étant donné que la récursivité impose que la méthode soit appliquée à de nombreux fichiers petits. Une méthode plus performante devra alors être utilisée pour les sous-fichiers dont le nombre d'éléments est inférieur à une valeur M, déterminée comme une limite au delà de laquelle "Quicksort" devient plus efficace que cette dernière méthode. On choisit généralement le tri par insertion directe (voir §2.33), mais une classification par fusion pourrait également être choisie, avec M relativement faible.

METHODE DE TRI	STABLE ?	TEMPS		ESPACE	RESTRICTIONS	ADAPTATION À LA MEMOIRE VIRTUELLE
		MOYEN	MAXIMUM			
MATH SORT (§ 2.32)	OUI	EXCELLENT	EXCELLENT	2 N	OUI	MAUVAIS
SHELL SORT (§ 2.33)	NON	MOYEN	MOYEN	N+E log N	NON	MAUVAIS
QUICK SORT (§ 2.34)	NON	EXCELLENT	MAUVAIS	N+E log N	NON	BON
HEAP SORT (§ 2.35)	NON	MOYEN	MOYEN	N	NON	MAUVAIS
LIST MERGE SORT (§ 2.36)	OUI	MOYEN	MOYEN	N+EN	NON	MOYEN
RADIX LIST SORT (§ 2.37)	OUI	EXCELLENT	MOYEN	N+EN	OUI	MOYEN

TABLEAU 2.1 - ÉTUDE COMPARATIVE DE QUELQUES MÉTHODES EFFICACES DE TRI INTERNE

Dans la pratique la valeur de M n'est pas très relevante, pourvu qu'elle soit dans l'intervalle de 5 à 20; la valeur optimale dépend de l'implantation du programme. La valeur 12 en donne, sans aucun doute, une bonne approximation.

On peut également observer que, plutôt que traiter immédiatement les petits fichiers (j - i < M), ils peuvent être ignorés, évitant ainsi des opérations d'insertion et de déplacement, en plus des appels aux routines de tri par insertion. Après le traitement du fichier entier par le programme de "Quicksort", on aboutit à une configuration du fichier avec tous les éléments de partage en leur place et séparés par des groupes de, au plus, M éléments non ordonnés. Une étape séparée de tri par insertion sur tout le fichier permet de compléter l'opération d'une façon efficace, étant donnée la propriété caractéristique du tri par insertion (efficace pour les fichiers déjà presque ordonnés).

Une fois implantés ces raffinements de la méthode, on peut chercher à améliorer la réalisation pratique. Pour atteindre une amélioration sensible, il est indispensable d'utiliser l'assembleur comme langage de programmation, étant entendu que les compilateurs produisent une interprétation près de 300 à 400% plus lente que la codification directe (Sedgewick, 1975). Il est en particulier nécessaire d'examiner rigoureusement les "loops" les plus internes, en essayant de réduire au maximum le nombre des instructions qui seront exécutées environ NlogN fois. Par exemple, on rencontre souvent la boucle interne codée de la façon suivante:

```
01. enquanto (J > I & K(J) > T) faça J := J - 1;
02. enquanto (I < J & K(I) < T) faça I := I + 1;
03. faça (se (J = I) então retorne.; PERMUTA(I,J); và para 1);
```

On observe qu'avec cette manière de coder, les boucles internes (01 et 02) comportent un double test, alors que le test sur les indices sert seulement lorsque ceux-ci s'égalent, à la dernière exécution de la boucle. Ce test peut alors être éliminé du "loop", en le réalisant seulement au moment de l'échange des clés (voir §2.34), moyennant quelques précautions en ce qui concerne les cas singuliers (comme celui de toutes les clés égales, par exemple).

2.4 LE TRI EXTERNE

Pour ce qui est du tri externe, la littérature spécialisée est aussi pauvre qu'elle est riche pour le tri interne: on peut seulement s'appuyer sur l'oeuvre monumentale de D.E.KNUTH (Knuth, 1973) pour la réalisation de ce sous-chapitre. Le principal motif de cette disproportion vient probablement de ce que le tri interne est plus facile à étudier, pour pouvoir s'affranchir des hypothèses restrictives inhérentes à la configuration concrète de l'implantation (périphériques et systèmes d'exploitation, en tête).

2.41 La philosophie

Rappelant la définition donnée au paragraphe 2.12, le tri d'un fichier est dit externe lorsque celui-ci ne peut pas être chargé dans la mémoire centrale en une seule fois. Bien entendu, dans certains cas de tri indirect, le tri externe pourra se transformer en tri interne, grâce au stockage de la clé et d'un pointeur (voir figure 2.3), au lieu de l'enregistrement tout entier.

En supposant que le fichier à ordonner soit stocké sur une aire de mémoire externe à accès direct, l'une quelconque des méthodes présentées pour le tri interne pourrait être utilisée, au prix d'un nombre d'opérations d'entrée-sorties ('I/O') proportionnel au nombre de comparaisons et de déplacements requis par la méthode. Evidemment, dans le cas d'un fichier de taille raisonnable, cette méthode devient vite prohibitive: on cherche par conséquent une philosophie différente, avec comme objectif la minimisation du nombre de 'I/O'.

L'idée la plus immédiate, et la plus utilisée, consiste à diviser le fichier d'entrée en sous-listes successives, appelées "runs", de telle sorte que chacune d'elles tienne individuellement dans la mémoire centrale, et puisse ainsi être triée et sauvegardée sur un espace de mémoire auxiliaire. Une étape complémentaire de fusion de ces "runs" devient alors nécessaire pour générer le fichier final désiré.

Le tri par distribution peut également conduire à un processus de tri externe, dans la mesure où le nombre de valeurs possibles des clés est limité.

Le prochain paragraphe traite de la partie de formation des "runs", tandis que le paragraphe suivant traite de la fusion de ceux-ci, en vue de la reconstituion du fichier complet. Le paragraphe 2.44 présente les autres alternatives du tri externe. Finalement le dernier paragraphe prétend conclure sur une étude comparative des diverses méthodes, en se placant dans une configuration usuelle des dispositifs périphériques.

2.42 La formation des "runs"

Soit N le nombre d'enregistrements du fichier original et P le nombre maximum d'enregistrements que peut contenir la mémoire interne. Il est clair que chacun des processus présentés (voir §2.3) peut être utilisé dans le tri d'une succession de "runs" de P enregistrements, avec un nombre total de "runs" formés égal à:

$$n = \text{Partie entière } (N / P) + 1 .$$

L'avantage d'un tel procédé est d'obtenir des "runs" de longueur fixe (excepté le dernier), ce qui constitue un facteur naturel d'optimisation de la phase de fusion. En compensation le nombre n de "runs" peut être grand (de l'ordre de plusieurs dizaines, par exemple), ce qui porte préjudice à la phase de fusion, et qui peut conduire à la morceler en plusieurs étapes successives.

Aussi doit-on chercher à diminuer le nombre n de "runs", ce qui peut seulement être réalisé par augmentation du dénominateur P. Un

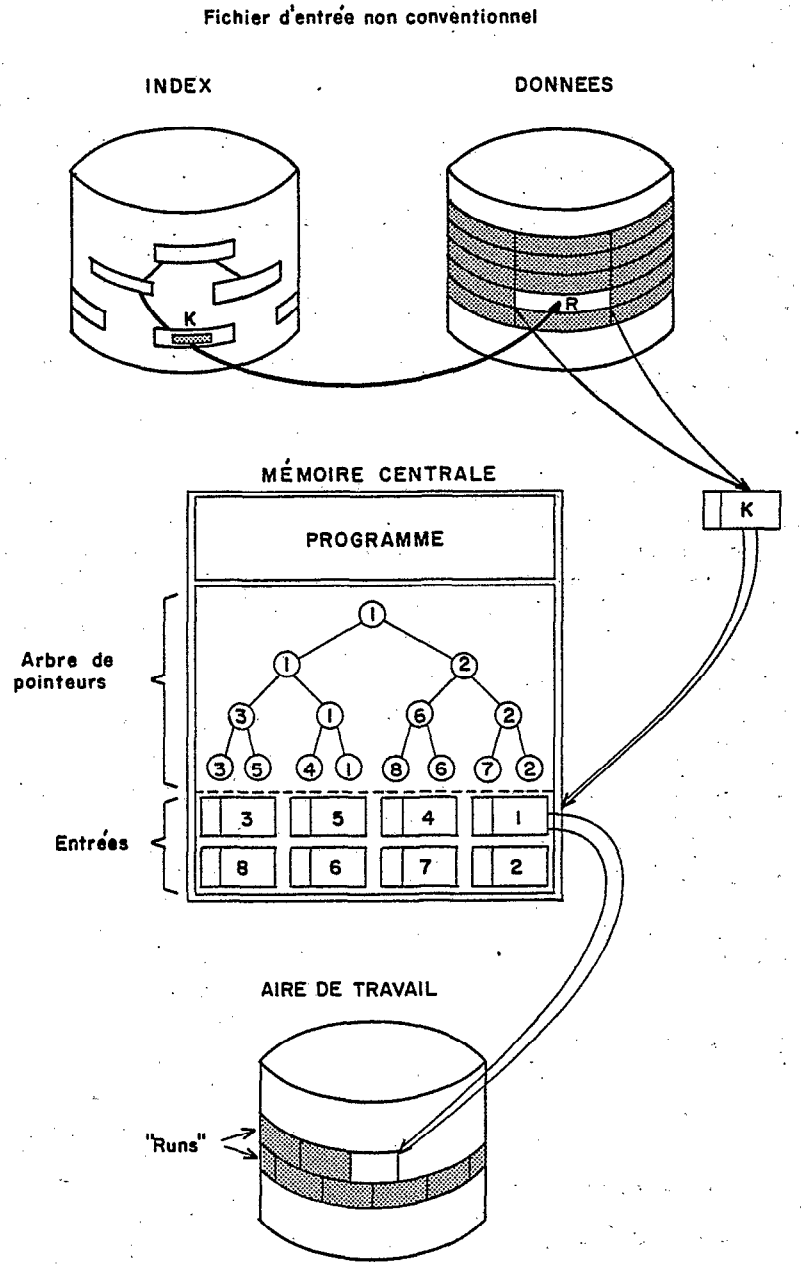


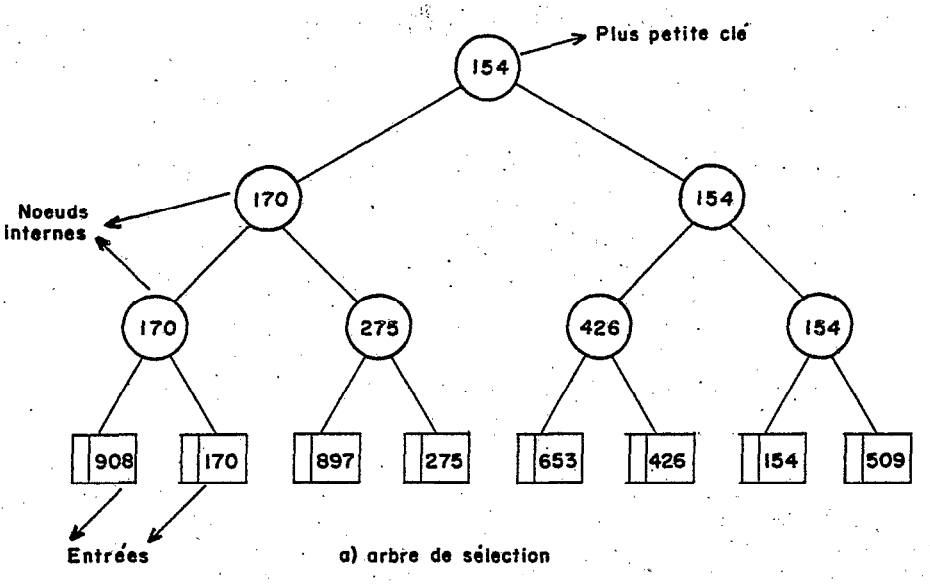
Fig. 2.7 PROCESSUS DE FORMATION DE 'RUNS' ('REPLACEMENT SELECTION')

procédé qui peut être adopté consiste à stocker P enregistrements dans la mémoire, à en sélectionner le plus petit (qui sera par conséquent le premier élément du "run" formé) et à susbstituer ce dernier par le $P + 1^{ème}$ élément du fichier initial (voir figure 2.7). Si cet élément est plus petit que celui qui vient de 'sortir' (la probabilité d'un tel évènement n'est que de 1 / (P + 1)), il ne pourra pas faire partie du même "run", et sera donc marqué comme ne pouvant intervenir dans les comparaisons suivantes. Dans tous les cas, le nouveau plus petit élément est sélectionné et placé comme élément suivant du "run", lequel termine seulement lorsque la mémoire est remplie d'éléments marqués. Un nouveau "run" est alors commencé, à partir des premiers éléments réorganisés. Ce processus ("Replacement selection") permet la formation de "runs" d'au moins P enregistrements (cas du fichier d'entrée ordonné dans la séquence exactement inverse de celle requise), de jusqu'à N enregistrements (cas du fichier déjà ordonné), et d'en moyenne 2P enregistrements. Avec cette augmentation moyenne de la taille des "runs", leur nombre est réduit de moitié, ce qui peut diminuer sensiblement le nombre d'étapes de fusion à réaliser.

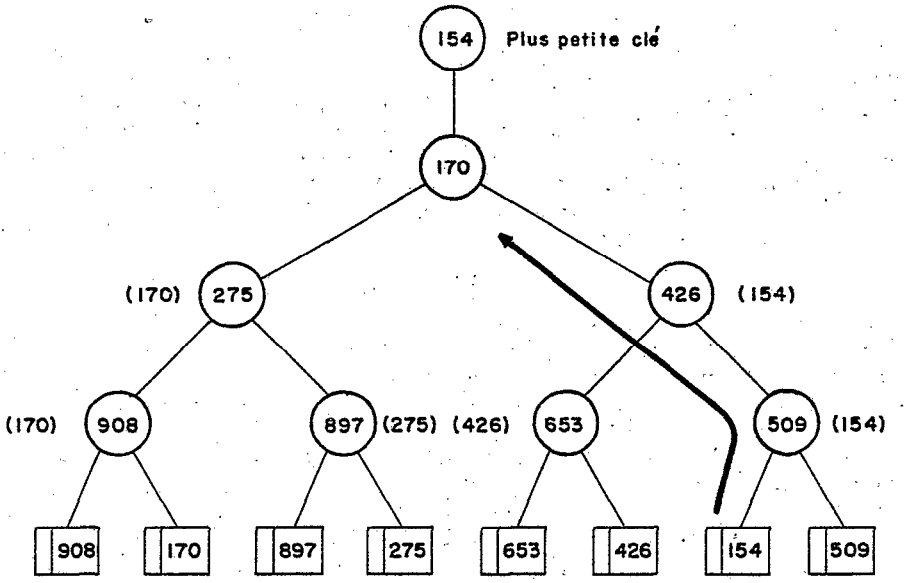
Pour l'implantation de cette méthode, n'importe quel algorithme de tri par sélection convient (voir §2.35), dans la mesure où le nombre de comparaisons effectivement réalisées pour la sélection du plus petit élément importe peu. Par exemple, la méthode de sélection directe devra exiger P comparaisons à chaque étape (entrée d'une nouvelle clé), tandis que la méthode type "Heapsort" devra requérir à peine logP comparaisons (une fois l'arbre monté).

La méthode présentée par J.R.WALTER, J.PAINTER et M.ZALK en 1958 (Knuth, 1973) utilise un arbre de sélection, d'une façon très semblable à la méthode de Williams (voir §2.35). Si l'on observe (voir figure 2.8) que pour déterminer la nouvelle structure de l'arbre quand la plus petite clé doit être substituée, il suffit d'examiner les clés qui ont été comparées avec cette plus petite (les clés 'perdantes' des comparaisons avec elle), il est naturel de stocker dans les noeuds internes (résultat de la comparaison) les clés les plus grandes au lieu des plus petites.

La méthode de sélection naturelle, proposée par W.D.Frazer et C.K.Worg en 1972 (Knuth, 1973), s'appuie sur le même arbre de 'perdants', et cherche à augmenter la taille des "runs" en utilisant un espace de mémoire auxiliaire (aire "d'overflow") pour stocker les enregistrements inactifs (marqués) de la mémoire centrale. En effet, dans la méthode précédente les clés marquées (plus petites que la dernière clé du "run" courant), restent dans la mémoire centrale (voir figure 2.9) et, en conséquence, l'arbre devient de plus en plus étroit, ce qui a pour résultat final la limitation de la taille des "runs". Si au contraire ces mêmes éléments sont stockés sur une aire de mémoire externe (qui peut recevoir P' éléments), l'espace ainsi libéré dans la mémoire centrale permet de conserver la dimension initiale de l'arbre, ce qui est un facteur de croissance pour la dimension des "runs". En se limitant au cas P' = P, ce qui permet d'éviter la possibilité de transbordement au moment du propre déchargement de ce dernier espace, la taille espérée des "runs" atteint eP, avec e = 2,72.

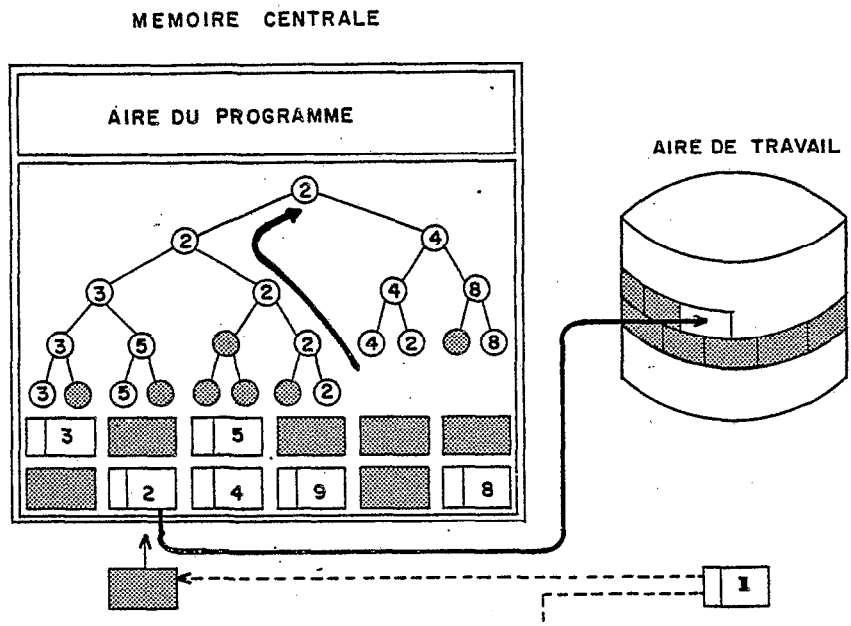


a) arbre de sélection

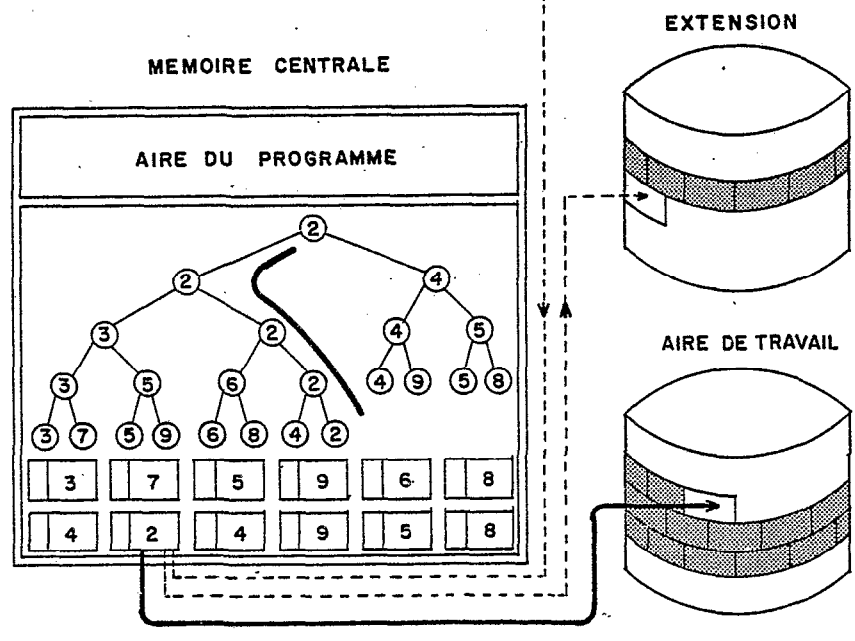


b) arbre de "perdants" correspondant

Fig. 2.8 ARBRE DE SÉLECTION ET ARBRE DE PERDANTS



a) 'Replacement Selection'



b) Sélection naturelle

Fig. 2.9 COMPARAISON DES MÉTHODES DE 'REPLACEMENT SELECTION' ET SELECTION NATURELLE

Cette dernière méthode peut être implantée à partir de l'algorithme codifié ci-après. On utilise P noeuds internes pour stocker les perdants ("Loser") et le numéro du "run" auquel appartiendra l'enregistrement pointé par "Loser". Q est le pointeur de la clé sélectionnée. L'algorithme procède sans routine spéciale, ni pour le montage de l'arbre, ni pour les 'vidages' successifs de la mémoire.

```
algoritmo SN(P); /* formation de "runs": SELECTION NATURELLE */
 01. array K(P + 1), RN(P), LOSER(P);
 02. faça (RMAX := RC := RQ := Q := CHAVE := 0, LASTKEY := *);
 03. para: J := 0 até P - 1 passo 1 faça (RN(J) := K(J):= 0,
                                          LOSER(J) := J);
 04. se (RQ = RC) então vá para 6; /* test de fin de "run" */
 05. faça (se (RQ > RMAX) então vá para 14, RC := RQ);
 06. se (RQ <> 0) então faça (SAIDA(K(Q)), LASTKEY := K(Q));
 07. se (FIMENTRADA or OVERFLOWFULL)
                  então faça (se (CHAVE = 1) então vá para 14,
                              RQ := RMAX + 1, vá para 10);
 08. ENTRADA(K(Q));
 09. se (K(Q) < LASTKEY)
        então faça (se (RQ = 0)
                    então faça (RQ := RQ + 1,
                                se (RQ > RMAX) então RMAX := RQ,
                                vá para 10);
              faça (PUSHOVERFLOW(K(Q)), se (OVERFLOWFULL)
                                           então K(Q) := *,
                                           vá para 7));
 10. T := entier( P / 2 + Q / 2); /* processus de substitution */
 11. se (RN(T) < RQ or (RN(T) = RQ & K(LOSER(T)) < K(Q))
       então faça (LOSER(T) :=: Q, RN(T) :=: RQ);
 12. se (T = 1) então vá para 4;
 13. faça (T := entier(T / 2), vá para 11);
 14. se (OVERFLOWEMPTY & CHAVE = 0) então pare.;
 15. faça (CHAVE := 1 - CHAVE, ENTRADA :=: OVERFLOW);
 16. se (CHAVE = 0) então vá para 7;
 17. vá para 2; fim
```

Présenté de cette façon, l'algorithme laisse libre la codification de certaines procédures. C'est le cas de ENTRADA(K), SAIDA(K), PUSHOVERFLOW(K), SETOVERFLOWEMPTY. Par ailleurs on utilise un certain nombre de variables booléennes: FIMENTRADA, OVERFLOWFULL, OVERFLOWEMPTY.

Ce même algorithme peut être utilisé pour l'implantation de la méthode de "Replacement selection", moyennant , en particulier, l'élimination des instructions labellées 15, 16 et 17, et la transformation de l'instruction 14 en:

```
14. pare. fim
```

2.43 La fusion des "runs"

La dernière étape du processus de tri externe consiste à intercaler entre eux les enregistrements appartenant aux diverses sous-listes triées. Dans le cas où l'on a seulement deux "runs", le moyen le plus simple d'obtenir le résultat désiré est de comparer les deux éléments les plus petits (un de chaque "run"), et d'en tirer le plus petit. Il suffit alors de répéter ce procédé jusqu'à épuisement de l'une des files; les enregistrements qui restent (de l'autre file) seront alors copiés, dans le même ordre sur le fichier de sortie. L'algorithme suivant réalise un tel travail:

```
algoritmo M; /* fusion des "runs": TWO-WAY MERGE */
 01. array X(M), Y(N), Z(M + N);
 02. (I, J, K) := 1; /* initialisation */
 03. se (X(I) > Y(J)) então vá para 7; /* comparaison */
 04. faça (Z(K) := X(I), K := K + 1, I := I + 1,
          se (I < M) então vá para 3);
 05. para: L := 0 até M + N + K passo 1 faça Z(K + L) := Y(J + L);
 06. pare.;
 07. faça (Z(K) := Y(J), K := K + 1, J := J + 1,
          se (J < N) então vá para 3);
 08. para: L := 0 até M + N + K passo 1 faça Z(K + L) := X(I + L);
 09. pare. fim
```

Il est clair que l'opération de fusion fait intervenir un nombre d'instructions exécutées, globalement proportionnel à n + m et, par conséquent, c'est une opération plus simple que celle de tri. Ainsi, lorsque l'on a P "runs";, la sélection de la plus petite clé pourra être réalisée à partir des P - 1 comparaisons des premières clés de chaque "run". Si P n'est pas très petit, on pourra avoir avantage à utiliser un arbre de sélection, d'une manière un peu semblable à la méthode présentée au paragraphe précédent. De la même façon, le nombre des comparaisons nécessaires pour la sortie d'un élément sera réduit à logP (par excès), une fois l'arbre établi.

Lorsque la valeur de P est grande (supérieure à 20, par exemple), la quantité d'information qui doit être stockée simultanément pour effectuer les comparaisons, et en particulier les P blocs correspondants aux P "runs", devient supérieure à l'espace disponible en mémoire interne. Dans ces conditions la fusion des "runs" ne peut pas être réalisée en une seule étape: la solution consiste alors à exécuter plusieurs étapes de fusion de P' "runs", en "runs" intermédiaires plus grands, stockés sur d'autres zones de travail, jusqu'à n'avoir plus que Q (Q < P') "runs", qui pourront être intercalés en une seule fois.

Il existe plusieurs possibilités de décomposition de ce travail, même à P' constant. Plus petit sera le nombre de lectures et d'écritures des enregistrements, plus rapide sera la phase de fusion: c'est dans ce sens que l'on cherche à optimiser le modèle adopté.

La méthode développée par R.L.Gilstad et présenté en 1960 sous le nom de "Polyphase merge" (Knuth, 1973), utilise la série des nombres de Fibonacci:

(X(i): X(0) = 0, X(1) = 1, X(n + 1) = X(n) + X(n - 1));

pour minimiser le nombre des copies des "runs" lorsque des bandes magnétiques sont utilisées comme aires de travail. En effet, dans le cas où l'on dispose par exemple de tois bandes magnétiques, après une étape de fusion des "runs" contenus sur les unités 1 et 2, il est nécessaire de distribuer à nouveau les "runs" résultants (étape de copie), avant de réaliser une nouvelle étape de fusion. Dans le cas où le nombre des "runs" à fusionner par ce procédé est un nombre de Fibonacci, soit X(n + 1), on peut distribuer X(n) "runs" sur l'unité 1 et X(n - 1) "runs" sur l'unité 2, de telle sorte que le premier pas produise X(n - 1) "runs" sur la bande 1 à intercaler avec les X(n - 1) "runs" de la bande 3, et ainsi de suite.

L'algorithme présenté ci-dessous permet de réaliser ce travail avec un nombre d'unités quelconque T (supérieur ou égal à 3, toutefois), en utilisant le "P-way merge" (P = T - 1). Comme le nombre initial de "runs" n'est pas toujours un nombre de Fibonacci, des "runs" fictifs doivent être créés en nombre convenable.

```
algoritmo PM(T); /* fusion des "runs": POLYPHASE MERGE */
 01. array A(T), D(T), FITA(T);
 02. para: J := 1 passo 1 até T - 1 faça (A(J) := D(J) := 1,
                                          FITA(J) := J);
 03. faça (A(T) := D(T) := 0, FITA(T) := T, L := J := 1);
 04. faça (GRAVACORRIDA(J), D(J) := D(J) - 1);
 05. se (FIMENTRADA) então faça (para: J := 1 até T passo 1
                                  faça REWINDFITA, vá para 11);
 06. se (D(J) < D(J + 1)) então faça (J := J + 1, vá para 4);
 07. se (D(J) = 0) então vá para 9;
 08. faça (J := 1, vá para 4);
 09. faça (L := L + 1, AUX := A(1), para: J := 1 até T - 1 passo 1
                       faça (D(J) := AUX + A(J + 1) - A(J),
                             A(J) := AUX + A(J + 1)));
 10. faça (J := 1, vá para 4);
 11. se (L = 0) então faça (FITASAIDA := 1, pare.);
 12. FUSAOCORRIDAS(T); /* fusion des "runs" sur l'unité T */
 13. faça (L := L - 1, REWINDFITA(T), REWINDFITA(P),
           TP(1) := TP(T), D(1) := D(T));
 14. para: J = 2 até T passo 1 faça (FITA(J) := FITA(J - 1),
                                     D(J) := D(J - 1),
                                     vá para 11); fim
```

La codification fait apparaître clairement la décomposition en deux étapes: les instructions labellées 2 à 10 exécutent la distribution des "runs" sur les unités 1 à T - 1, avec la génération sur chaque bande j de D(j) "runs" fictifs, jusqu'à épuiser le fichier d'entrée. Les commandes 11 à 14 réalisent la fusion des "runs", en utilisant la routine FUSAOCORRIDAS(T) pour intercaler les "runs" de fita(1), fita(2), ... , fita(T - 1), jusqu'à épuisement des listes (tant fictives que réelles) contenues sur la bande fita(T - 1). La fusion des "runs" fictifs dans ce cas se traduit par:

```
faça (para: J = 1 até T - 1 passo 1 faça (D(J) := D(J) - 1;
                                          D(T) := D(T) + 1).
```

On utilise également une routine appelée REWINDFITA, avec un paramètre pour désigner quelle est l'unité à réembobiner. L'absence de paramètres doit être interprétée comme le réembobinage de toutes les bandes. Enfin, le paramètre FITASAIDA, toujours égal à 1, sert de pointeur pour l'unité sur laquelle le fichier définitif est chargé.

Il existe d'autres modèles de fusion des "runs" sur bandes magnétiques: par exemple le "Cascade merge" présenté par B.K.Betz et W.C.Carter en 1969 (Knuth, 1973) permet d'obtenir des résultats semblables à ceux obtenus avec le modèle précédent. On peut observer également que ces procédés de fusion peuvent être adaptés à la lecture 'à reculons' des bandes magnétiques, moyennant la formation alternative de séquences d'enregistrements ascendantes et descendantes. Cette dernière ressource permet d'éliminer le temps de réembobinage des bandes, en écrivant dans un sens et en lisant (pour la fusion) dans l'autre.

Dans le cas où l'on peut utiliser des aires de travail sur disques magnétiques, le problème créé par la limitation du nombre d'unités de bandes physiquement disponibles pour le travail de fusion disparaît, la valeur de P étant alors seulement limitée par l'espace de mémoire interne consacré au stockage des "buffers" de lecture et écriture au cours de l'étape. Bien entendu, la fusion de S "runs" sera réalisée avec un nombre minimal d'étapes, en utilisant des étapes intermédiaires de "P-way merge". Le modèle de fusion pour la totalité du processus peut être représenté par un arbre 'P-aire', qui pourra être complété par l'addition de "runs" fictifs, comme le montre l'exemple de la figure 2.10.

Le nombre d'étapes de fusion est la hauteur de l'arbre, c'est à dire:

$$q = \log_P S \text{ (par excès).}$$

Ainsi pour réaliser la fusion des 13 "runs" de la figure 2.10, on obtient trois étapes dans le cas P = 3 ($\log_3 13 > 2$), alors qu'il suffirait de deux étapes avec P = 4 ($\log_4 13 < 2$). Une fois donné le nombre S de "runs", la valeur de P est choisie comme étant le plus petit entier qui rend $\log_P S$ inférieur à un nombre q, déterminé avec S et la valeur maximale P_{max} de P permise par l'espace disponible en mémoire interne.

L'algorithme présenté ci-dessous réalise la fusion des "runs" en plusieurs étapes, en accord avec ce modèle.

```
algoritmo V(P,S); /* fusion de S "runs" */
 01. array R(S + 1); /* indices des "runs" */
 02. faça D := P - mod(S,P - 1); /* nombre des "runs fictifs */
 03. faça (K := P - D, R(S + 1) := FUSAOCORRIDAS(R(1), ... , R(K)),
          D := 0);
 04. para: I := 1 até K passo 1 faça R(I) := R(I + K);
 05. faça (S := S - K, se (S = 1) então vá para 6, vá para 3);
 06. pare. fim
```

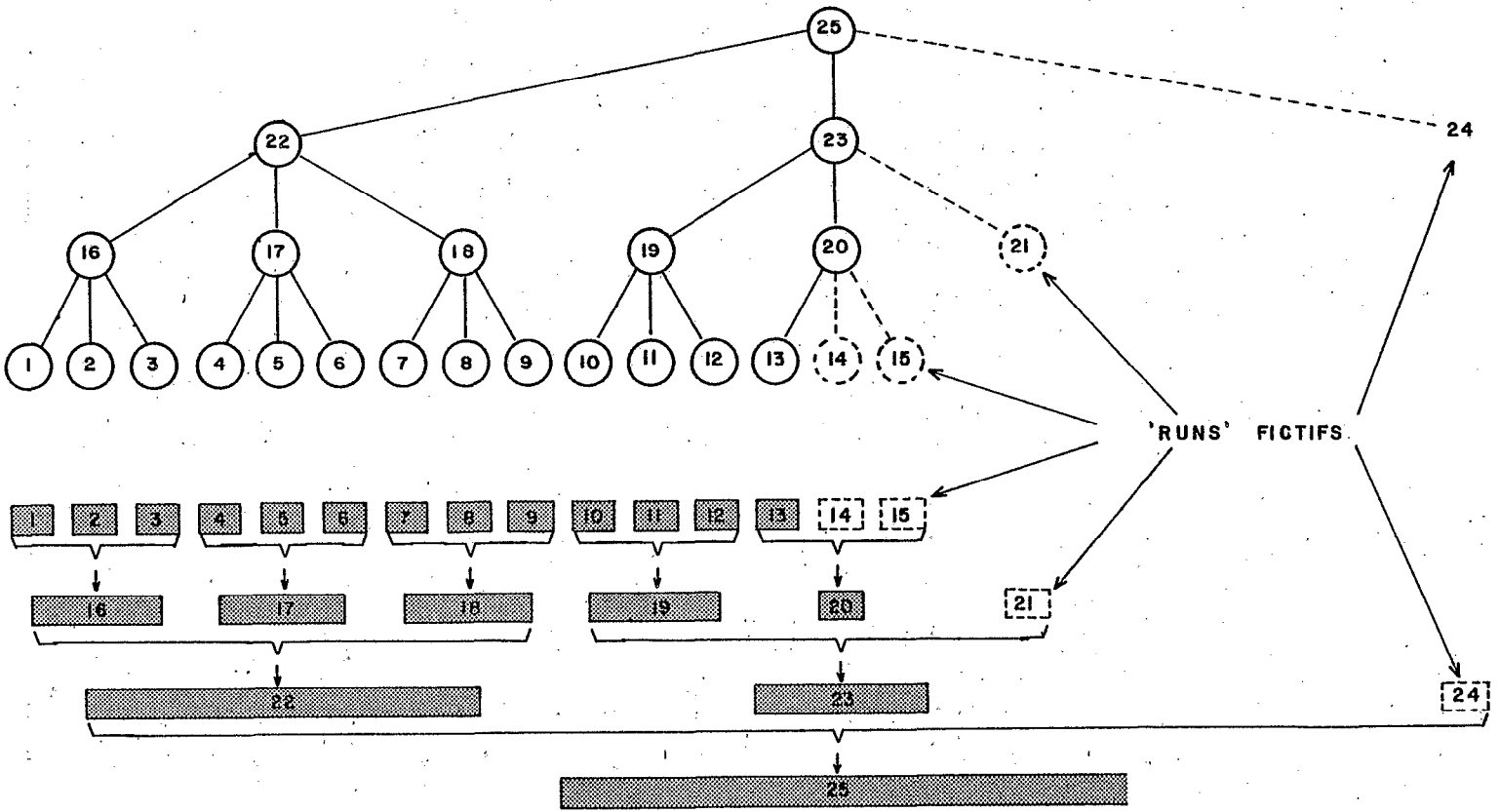


Fig. 2.10 MODÈLE DE FUSION EN PLUSIEURS ETAPES REPRESENTÉ PAR UN ARBRE TERNAIRE

Avec cette codification, la procédure FUSAOCORRIDAS restitue comme résultat l'adresse du nouveau "run". L'algorithme termine quand il ne reste plus qu'un "run", dont l'adresse sera R(1).

2.44 Les autres méthodes de tri externe

L'idée de tri par distribution se concrétise dans les antiques 'trieuses de cartes'. Dans le cas où les valeurs des clés appartiennent à un intervalle de:

0 à 2^k - 1, le fichier peut être ordonné avec k étapes de distribution en deux paires de fichiers auxiliaires, en accord avec les 'digits' binaires successifs. Cette méthode, appelée "External radix sort", nécessite dans sa dernière étape la copie d'environ la moitié des enregistrements d'un fichier à l'autre, de façon que soit reconstitué le fichier initial. Un tel processus peut être intéressant lorsque le domaine de valeur des clés est restreint, et quand l'utilisation d'un algorithme stable est jugé nécessaire (voir §2.15).

Des algorithmes intéressants existent et ont été développés pour effectuer le tri d'un fichier à partir de deux bandes magnétiques (ou de deux zones de disque), étant entendu que le tri par fusion demande au moins trois unités. L'un d'eux, présenté par H.B.Demuth en 1956 (Knuth, 1973), suit la philosophie de "Replacement selection". P enregistrements sont lus sur l'unité d'entrée et servent à établir un arbre de sélection à partir duquel la substitution du plus petit élément par les enregistrements lus successivement, permet la génération d'une file sur la bande de sortie, avec un nombre d'inversions plus faible que le nombre initial. Une fois épuisé le fichier d'entrée, les P - 1 enregistrements présents dans la mémoire possèdent les P - 1 plus grandes clés du fichier et sont déchargés en séquence ascendante. Après réembobinage des deux bandes, le procédé peut être répété, avec permutation des unités d'entrée et de sortie, pour éliminer d'autres inversions en restituant au moins P - 1 autres enregistrements dans leur place définitive. Le fichier devient complètement ordonné lorsque l'on peut vérifier qu'il n'y a aucune inversion, ce qui peut être testé au moment de l'écriture de chaque enregistrement en sortie en le comparant au précédent. Evidemment l'efficacité du procédé dépend des valeurs relatives de P et N (nombre total d'enregistrements du fichier): le nombre maximal d'étapes nécessaires est (N - 1) / (P + 1), par excès. Cette méthode est très semblable au procédé du "Bubble sort" (voir §2.34).

De la même façon les autres méthodes de tri interne par échanges, comme le "Quicksort" ou le "Radix exchange sort" peuvent être adaptées au tri externe avec deux unités de bandes magnétiques ("Two tape sorting"). Les résultats obtenus sont très voisins, étant entendu que les opérations les plus dispendieuses au point de vue du temps sont celles de lecture et d'écriture sur les bandes, communes à toutes les méthodes.

Une dernière méthode présentée par S.Sobel en 1962, et perfectionnée par D.L.Bencher en 1966 (Knuth, 1973), apparaît comme une alternative à la méthode globale de tri par fusion (voir §§2.42 & 2:43), dans la mesure où ce procédé ("Oscillating sort") alterne les

étapes de fusion de "runs" avec les étapes de formation des "runs". La première étape de fusion est effectuée en lisant 'à reculons' les "runs" ascendants pour former de nouveaux "runs" descendants, en laissant les unités prêtes à recevoir d'autres "runs" qui seront de nouveau intercalés. Le tableau 2.2 a) donne un exemple de la méthode avec T = 5 dérouleurs de bandes magnétiques.

Lorsque le nombre total S de "runs" à former est une puissance de P = T - 1, la méthode conduit au plus petit nombre d'étapes sur les données, c'est à dire:

$$\log_P S \text{ (par excès)} + 1 \quad ,$$

résultat normalement atteint avec 2P bandes magnétiques. Toutefois, lorsque S est de la forme $P^k + 1$, il faut une étape supplémentaire sur la totalité des données. Cet inconvénient peut être réduit avec le raffinement proposé par Bencher ("Criss cross merge"), dont le schéma tend à retarder les étapes de fusion, jusqu'à obtenir une meilleure connaissance de S. Le tableau 2.2 b) donne un exemple du même cas T = 5; il est intéressant d'observer que les "runs" descendants D_4 obtenus dans la phase 9 seront fusionnés seulement dans la phase 15 (à moins que le fichier d'entrée ne soit épuisé avec 16 "runs"). De la même manière les étapes 1 à 15 sont répétées pour obtenir le deuxième "run" A_{16} à l'étape 23, et ainsi de suite. L'algorithme présenté ci-après incorpore cette modification.

```
algoritmo O(P); /* tri externe: OSCILLATING SORT et
                              CRISS-CROSS MERGE   */
 01. array D(T), A(L,T);
 02. para: J := 0 até T - 1 passo 1 faça D(J) := 0;
 03. para: J := 1 até T - 1 passo 1 faça (DISTRICORRIDA(J));
 04. faça (A(0,0) := -1, L := Q := 0);
 05. se (NAOFIMENTRADA) então vá para 8;
 06. para: J := 0 até T - 1 passo 1
        faça (se (J <> Q & mod(A(0,J),2) = 1) então REWINDFITA(J));
 07. faça (MERGECORRIDAS(Q)), pare.); /* sortie unité Q */
 08. faça (L := L + 1, R := Q, S := 0,
           Q := mod(Q + 1, T)); /* incrémente d'un niveau */
 09. para: J := 1 até T - 2 faça (DISTRICORRIDA(mod(Q + J,T)));
 10. A(L,Q) := A(L,R) := -1;
 11. se (A(L - 1,Q) <> S) então vá para 8;
 12. faça (MERGECORRIDAS(R), S := S + 1, L := L - 1,
           A(L,R) := S, A(L,Q) := -1);
 13. R := mod(2 * Q - R,T);
 14. se (L = 0) vá para 5;
 15. para: J := 0 até T - 1 passo 1
        faça (se (A(L,J) <> S & J <> Q & J <> R) então vá para 8);
 16. vá para 11; fim
```

Cet algorithme utilise les deux procédures suivantes:

	OPERATION	T1	T2	T3	T4	T5
PHASE 1	DISTRIBUTION	A^1	A^1	A^1	A^1	–
PHASE 2	FUSION	–	–	–	–	D^4
PHASE 3	DISTRIBUTION	–	A^1	A^1	A^1	$D^4 A^1$
PHASE 4	FUSION	D^4	–	–	–	D^4
PHASE 5	DISTRIBUTION	$D^4 A^1$	–	A^1	A^1	$D^4 A^1$
PHASE 6	FUSION	D^4	D^4	–	–	D^4
PHASE 7	DISTRIBUTION	$D^4 A^1$	$D^4 A^1$	–	A^1	$D^4 A^1$
PHASE 8	FUSION	D^4	D^4	D^4	–	D^4
PHASE 9	FUSION	–	–	–	A^{16}	–
- - - -	- - - - - - -	- - -	- - -	- - -	- - -	- - -

a) Méthode simple

	OPERATION	T1	T2	T3	T4	T5
PHASE 1	DISTRIBUTION	–	A^1	A^1	A^1	A^1
PHASE 2	DISTRIBUTION	–	A^1	$A^1 A^1$	$A^1 A^1$	$A^1 A^1$
PHASE 3	FUSION	D^4	–	A^1	A^1	A^1
PHASE 4	DISTRIBUTION	$D^4 A^1$	–	A^1	A^1 A^1	A^1 A^1
PHASE 5	FUSION	D^4	D^4	–	A^1	A^1
PHASE 6	DISTRIBUTION	$D^4 A^1$	$D^4 A^1$	–	A^1	A^1 A^1
PHASE 7	FUSION	D^4	D^4	D^4	–	A^1
PHASE 8	DISTRIBUTION	$D^4 A^1$	$D^4 A^1$	$D^4 A^1$	–	A^1
PHASE 9	FUSION	D^4	D^4	D^4	D^4	–
PHASE 10	DISTRIBUTION	D^4	$D^4 A^1$	$D^4 A^1$	$D^4 A^1$	A^1
PHASE 11	FUSION	$D^4 D^4$	D^4	D^4	D^4	–
PHASE 12	DISTRIBUTION	$D^4 D^4 A^1$	D^4	$D^4 A^1$	$D^4 A^1$	A^1
PHASE 13	FUSION	$D^4 D^4$	$D^4 D^4$	D^4	D^4	–
PHASE 14	DISTRIBUTION	$D^4 D^4 A^1$	$D^4 D^4 A^1$	D^4	$D^4 A^1$	A^1
PHASE 15	FUSION	$D^4 D^4$	$D^4 D^4$	$D^4 D^4$	D^4	–
PHASE 16	FUSION	D^4	D^4	D^4	–	A^{16}
- - - - -	- - - - - - -	- - -	- - -	- - -	- - -	- - -
PHASE 22	FUSION	$D^4 D^4$	$D^4 D^4$	D^4	–	$A^{16} D^4$
PHASE 23	FUSION	D^4	D^4	–	A^{16}	A^{16}
- - - -	- - - - - - -	- - -	- - -	- - -	- - -	- - -

b) Avec le raffinement du 'criss-cross merge'

A^i = 'Run' ascendant de taille i fois le 'run' élémentaire
D^i = 'Run' descendant de taille i fois le 'run' élémentaire
TI = Bande i

TABLEAU 2.2 TRI EXTERNE PAR LA METHODE "OSCILLATING SORT"

```
algoritmo DISTRICORRIDA(J); /* écrit un "run" initial */
 01. faça A(L,J) := 0;
 02. se (FIMENTRADA) então faça (D(J) := D(J) + 1, vá para 4);
 03. GRAVACORRIDA(J);
 04. pare. fim
```

```
algoritmo MERGECORRIDAS(J); /* fusion des "runs" unité J */
 01. para: I := 0 até T - 1 passo 1
       faça (se (D(I) = 0 & I <> J) então vá para 4);
 02. para: I := 0 até T - 1 passo 1
       faça (se (I <> J) então D(I) := D(I) - 1);
 03. faça (D(J) := D(J) + 1, vá para 5);
 04. FUSAOCORRIDAS(J);
 05. pare. fim
```

Les trois routines GRAVACORRIDA, FUSAOCORRIDAS et REWINDFITA qui subsistent dans la codification sont celles qui sont présentées par ailleurs, pour l'algorithme "Polyphase merge" (voir §2.43). En particulier le rôle de FUSAOCORRIDAS(J) est d'intercaler sur l'unité J un "run" de chaque unité I, telle que I <> J et D(I) = 0, tout en décrémentant d'une unité le nombre D de "runs" fictifs des autres unités.

La matrice A(L,J) est utilisée pour représenter l'état de chacun des L "runs" écrits au niveau L sur l'unité J. Quand A(L,J) est pair, le "run" correspondant est descendant, ascendant dans le cas contraire. La valeur A(L,J) = - 1 indique que le niveau L n'utilise pas le niveau J.

L'algorithme principal est constitué de telle forme que l'exécution de la boucle de niveau L (instructions labellées 08 à 16) s'achève avec l'unité Q vide et au maximum un "run" en chacune des autres bandes magnétiques. L'instruction 7 réalise alors la fusion des "runs" en lisant 'en avant' les "runs" ascendants et 'à reculons' les "runs" descendants.

2.45 Comparaison des différentes méthodes

Il existe bien d'autres raffinements des méthodes présentées dans ce dernier sous-chapitre. Au contraire du cas du tri interne, une étude comparative de l'efficacité de chacune d'entre elles est impossible, dans la mesure où la meilleure méthode à utiliser dépend essentiellement de la disponibilité en périphériques de chaque installation. Par exemple il serait absurde d'utiliser les méthodes du type "Two tape sorting" lorsque l'on dispose d'aires de disque suffisantes pour réaliser un tri par distribution et fusion hautement efficace.

Cependant quelques observations peuvent être faites dans le but de choisir une méthode efficace et universelle de tri externe. On peut noter par exemple que le tri par "External radix sorting" est généralement inférieur au tri par distribution et fusion, pour

produire des sous-fichiers plus petits dans le premier cas que dans le second.

La majorité des algorithmes présentés dans les paragraphes précédents cherchent à minimiser les opérations de copies de "runs" d'une unité à l'autre, ainsi que les opérations de réembobinage de bandes. Dans le cas le plus fréquent d'utilisation de zones de disque, qui autorisent l'accès direct, les problèmes créés par la limitation du nombre d'unités physiques disparaissent, ce qui évite tant les copies que les réembobinages. En conséquence l'optimisation de la méthode fait intervenir d'autres aspects du traitement.

2.46 L'amélioration du procédé de tri externe

En se limitant au cas spécifique d'utilisation d'aires de mémoire auxiliaire sur disque, on doit chercher à établir une méthode de tri externe, réalisée à partir de trois subroutines:

- routine de formation des "runs";

- routine de réalisation d'une étape de fusion d'au plus P_{max} "runs";

- routine pricipale de groupement des "runs" pour la fusion, coforme à un modèle en arbre 'P-aire' complet.

Evidemment, les conditions physiques de la réalisation jouent un rôle essentiel dans le choix de la méthode. Par exemple, en ce qui concerne la formation des "runs", l'efficacité de la méthode dans une configuration de mémoire virtuelle est le facteur le plus important à considérer. D'un autre côté la distribution des "runs" successifs, les considérations de blocage, de structures de stockage, etc..., ont une influence primordiale dans la phase de fusion des "runs".

Dans le chapitre 3 on examine, dans une configuration particulière, les possibilités d'optimisation des ressources disponibles, dans le but d'obtenir un modèle physique efficace de tri externe. Pour développer le modèle logique, on peut observer que l'algorithme V(P,S) de fusion de S "runs" par "P-way merge" (voir §2.43), utilise l'artifice des "runs" fictifs, dans la mesure où la procédure FUSAOCORRIDAS prend un nombre fixe de "runs" à intercaler. Par ailleurs, comme la valeur P_{max} du nombre de "runs" à fusionner simultanément est seulement limitée par l'espace de mémoire interne disponible dans cette phase, celle-ci peut être choisie relativement grande (de l'ordre de plusieurs dizaines, en fonction du blocage). En conséquence, l'établissement d'un arbre de sélection, comme il a été suggéré au paragraphe 2.43, peut être intéressant pour réduire le nombre de comparaisons.

La méthode présentée ci-après effectue la fusion des S "runs" initiaux à partir d'étapes successives de fusion d'un nombre variable (inférieur à P_{max}) "runs", conformément au modèle d'arbre 'P-aire' complet (seule la dernière étape prend moins de P_{max} "runs"). Une file des indices des "runs" à fusionner est gérée (voir figure 2.11); à

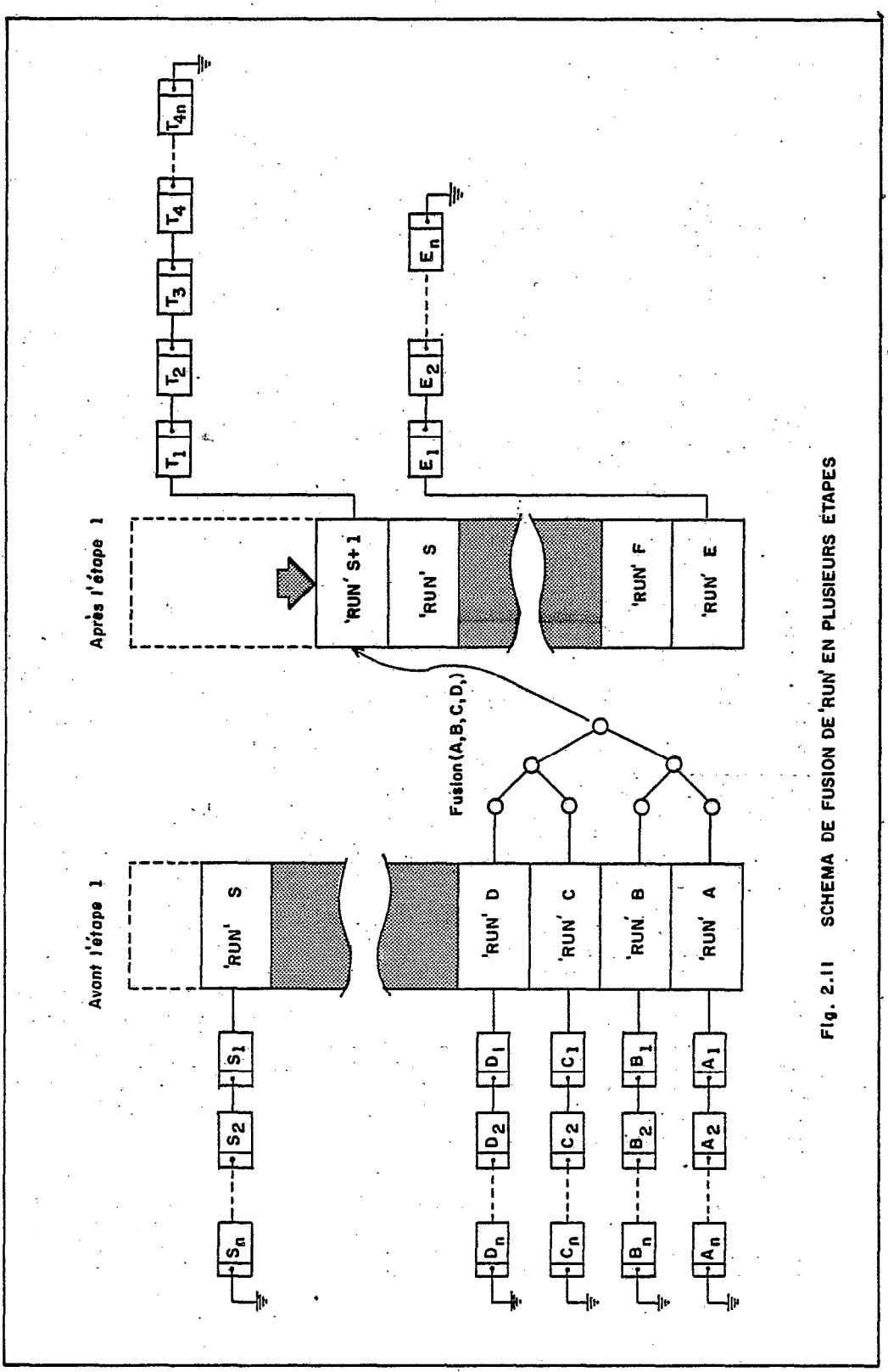


Fig. 2.11 SCHEMA DE FUSION DE 'RUN' EN PLUSIEURS ÉTAPES

chaque étape les P_{max} "runs" du début de la file sont fusionnés, en un "run" dont l'adresse est placée à la fin de cette même file.

La procédure FUSAOCORRIDAS suit la philosophie de "Replacement selection" à partir d'un arbre de sélection de la plus petite clé (d'au plus P_{max} noeuds, réorganisé par nouvelle initialisation à chaque fois qu'un "run" est épuisé). Une variable booléenne globale ULTIMOPASSO est utilisée; selon la valeur de cette variable la procédure GRAVACORRIDA est substituée par la procédure GRAVASAIDA.

```
algoritmo FC(P,S); /* fusion de S "runs" en plusieurs étapes */
 01. array R(S + 1); /* adresses des "runs" */
 02. faça (K := P, ULTIMOPASSO := false);
 03. se (S < P) então faça (ULTIMOPASSO := true, K := S);
 04. faça (R(S + 1) := FUSAOCORRIDAS(K), S := S - K + 1);
 05. se (S = 1) então pare.;
 06. para: I := 1 até S passo 1 faça R(I) := R(I + K);
 07. và para 3; fim
```

```
algoritmo FUSAOCORRIDAS(K); /* N-WAY MERGE par sélection */
 01. array LOSER(P), KEY(P), RN(P);
 02. faça (Q := RQ := 0, COR := 1);
 03. para: J := 0 até K - 1 passo 1 faça (KEY(J):= RN(J) := 0,
                                          LOSER(J) := J);
 04. se (RQ = 0) então và para 6;
 05. faça (se ULTIMOPASSO então faça (GRAVASAIDA(KEY(Q),
                                      và para 6),
          GRAVACORRIDA(KEY(Q)));
 06. faça(LEITURA(KEY(Q),COR), COR := COR + 1,
          se FIMCORRIDA então và para 12);
 07. se (RQ = 0) então RQ := 1;
 08. faça T := entier (K / 2 + Q / 2);
 09. se (RN(T) < RQ or (RN(T) = RQ & KEY(LOSER(T)) < KEY(Q)))
        então faça (AUX := Q, Q := LOSER(T), LOSER(T) := AUX,
                    AUX := RQ, RQ := RN(T), RN(T) := AUX);
 10. se (T = 1) então và para 4;
 11. faça (T := entier(T / 2), và para 9);
 12. faça (K := K - 1, REORGANIZA);
 13. se (K <> 1) então và para 2;
 14. se (ULTIMOPASSO) então faça (enquanto NAOFIMCORRIDA
                                  faça GRAVASAIDA(KEY(Q), pare.);
 15. se (NAOULTIMOPASSO) então faça (enquanto NAOFIMCORRIDA
                           faça GRAVACORRIDA(KEY(Q)), pare.); fim
```

La procédure LEITURA(KEY(Q),COR) autorise l'entrée de l'enregistrement courant du "run" COR, dans la position Q, et active la variable booléenne FIMCORRIDA lorsque le "run" est épuisé. Dans ce dernier cas, la routine REORGANIZA est appelée pour réinitialiser le procédé de sélection avec un arbre de K - 1 noeuds au lieu de K. Quand K devient égal à 1, les enregistrements qui restent du dernier "run"

sont copiés sur le fichier de sortie, ou sur le "run" résultant si l'étape est intermédiaire.

2.5 CONCLUSIONS

Au cours de ce chapitre, les pricipaux concepts et techniques du tri ont été présentés. En particulier, les notions de tri interne et externe ont été mises en évidence, les méthodes correspondantes étant fondamentalement différentes (bien que le même résultat soit recherché), eu égard au principe de minimisation des opérations d'entrée-sorties dans ce dernier cas.

De toutes façons, une fois la configuration de l'ordinateur et ses ressources en périphériques fixées, le critère essentiel d'efficacité devient, sans aucun doute, le temps passé à la classification désirée. Il convient toutefois d'observer qu'il existe souvent un compromis à respecter entre la minimisation du temps passé et une pénalisation référente à tout autre paramètre. Ainsi, le sacrifice de la stabilité imposé par les meilleurs algorithmes peut se traduire dans certains cas, par une augmentation du nombre de comparaisons à réaliser (en raison de l'addition d'un certain nombre de champs à la clé), et en conséquence peut augmenter sensiblement le temps nécessaire à l'opération. De la même manière, l'objectif d'économie d'espace est très important, dans la mesure où un tri interne est toujours préférable à un tri externe, même très efficace.

Ces considérations ont conduit au choix d'une méthode de tri interne, le "Quicksort", et, moyennant l'hypothèse d'utilisation de zones de mémoire auxiliaire à accès direct, d'une méthode de tri externe par distribution et fusion de "runs". Finalement quelques améliorations ont été trouvées et analysées (voir §§2.39 & 2.46), dans le but de perfectionner les procédés sélectionnés. Pour choisir une méthode efficace de formation de "runs", il est nécessaire de restreindre encore le domaine d'application de ce processus d'optimisation.

La finalité du chapitre 3, ci-après, est essentiellement l'étude de l'évolution des concepts présentés dans un environnement de mémoire virtuelle. De plus, l'adoption d'hypothèses restrictives en ce qui concerne les ressources périphériques disponibles, conduit à proposer une systématique de distribution des "runs".

3. TRI ET MEMOIRE VIRTUELLE

La plupart des systèmes actuels de grande capacité utilise les ressources de la mémoire virtuelle, en offrant la possibilité d'écrire des programmes comme si il existait une mémoire interne disponible quasi-illimitée. Avec cette technologie, les méthodes de tri externe peuvent paraître obsoletes, dans la mesure où la classification peut être réalisée très simplement selon un abordage propre au tri interne, tout en libérant le programmeur de la charge de gestion de mémoire.

Dans la pratique, la zone d'adressage virtuel est obtenue à partir d'aires de mémoire auxiliaire à accés direct, par exemple par implantation d'un système réel via demande de 'pages': la gestion de mémoire est alors réalisée automatiquement par le système, au moyen d'un logiciel appelé 'de pagination'.

Cependant, il est important de savoir jusqu'à quel point le système utilise efficacement la mémoire, étant entendu que le temps gâché lors d'une opération de tri par les échanges de pages n'est certainement pas négligeable, puisque ceux-ci font intervenir des opérations de transfert de la mémoire externe à la mémoire interne, et vice-versa.

Evidemment, le modèle de tri choisi reste un paramètre important pour la fonction de pagination, en plus de la disponibilité en mémoire réelle et du besoin en mémoire virtuelle: toutes les méthodes de tri interne ne seront pas forcément adéquates à une 'machine virtuelle'. De la même façon qu'il existe des règles à observer lorsque l'on écrit un programme de tri sur un système conventionnel, il existe de nouvelles règles à suivre pour écrire un programme qui doit fonctionner sur un ordinateur muni d'un système de mémoire virtuelle (Brawn et al., 1970).

Dans les paragraphes suivants, après avoir fixé une configuration déterminée d'ordinateur (celle pour laquelle ce travail a été réalisé), on étudiera le comportement des méthodes choisies pour chacune des tâches à effectuer dans l'environnement de mémoire virtuelle. En particulier le choix de la méthode de formation des "runs" sera précisé, tandis qu'il était resté ouvert au chapitre précédent (voir §§2.45 & 2.46).

Pour aboutir à une optimisation de la méthode qui sera finalement implantée, il est nécessaire d'étudier les diverses possibilités de structure de stockage des "runs" sur disque, en cherchant à minimiser les temps passés, tant dans la phase de distribution que dans celle de fusion des "runs".

3.1 LE SYSTEME DE CALCUL

Le présent travail se destine avant tout à l'implantation d'une méthode efficace de tri pour un fichier non conventionnel (voir chapitre 4). Bien que l'on ait cherché à paramétrer le programme pour obtenir une souplesse maximale par rapport à la configuration du système, l'optimisation de la méthode est naturellement obtenue

moyennant certaines hypothèses en ce qui concerne les composants périphériques de celui-ci.

Les paragraphes suivants fournissent une description générale du système de calcul réel pour lequel la méthode a été développée. Tout d'abord, une description résumée des équipements disponibles est présentée, en détaillant les caractéristiques estimées des temps d'accès de mémoire auxiliaire sur disque, la plus utilisée dans cette configuration. Dans le paragraphes suivants, les ressources du système d'exploitation qui permettent la réalisation de la multiprogrammation et de la mémoire virtuelle sont examinées. Finalement, les autres ressources du système qui interviennent dans les diverses phases du travail, sont également discutées.

3.11 L'équipement

Le système de calcul sur lequel on prétend développer une méthode efficace de tri, est essentiellement constitué d'un ordinateur IBM, série 370 modèle 135, de 256K de mémoire réelle. La performance de l'unité centrale en terme de temps, est évaluée en Mips, millions d'instructions par seconde, soit 0,185 Mips dans le cas spécifique du modèle 135.

En plus des unités périphériques classiques lentes (une lectrice de cartes, modèle 2501, de 600 cartes par minute, une imprimante, modèle 1403, de 1100 lignes par minute et une console-système, modèle 135), on dispose de quatre unités de dérouleur de bandes magnétiques, modèle 3420, permettant une densité d'écriture de 1600 bpi ("bytes per inch").

En ce qui concerne la mémoire auxiliaire à accès direct, on dispose d'environ 200 Mégabytes, répartis en sept unités de disques magnétiques, modèle 2314. La figure 3.1 a) représente le schéma de n'importe quel "disk-pack". Les données sont stockées sur les vingt faces internes de onze disques concentriques, tournant à la vitesse de 2400 rotations par minute. Un bras d'accès muni de vingt têtes de lecture-écriture, et capable de se mouvoir dans le sens radial des disques, est utilisé pour le transfert des données. Chaque face de disque est divisée en 200 pistes concentriques. On appelle cylindre l'ensemble des vingt pistes qui peuvent être lues, ou écrites, non simultanément, sans déplacement du bras. Chaque piste permet le stockage de 7294 bytes, de telle sorte que la capacité maximale de stockage d'un "disk-pack" est supérieure à 29 Mégabytes.

Dans ce type d'unité périphérique, le temps d'accès et de transfert des données dépend de quatre opérations:

- mouvement du bras ("seek");

- sélection de la tête de lecture correspondant à la piste recherchée;

- rotation du disque jusqu'à ce que la position initiale de transfert des données soit atteinte;

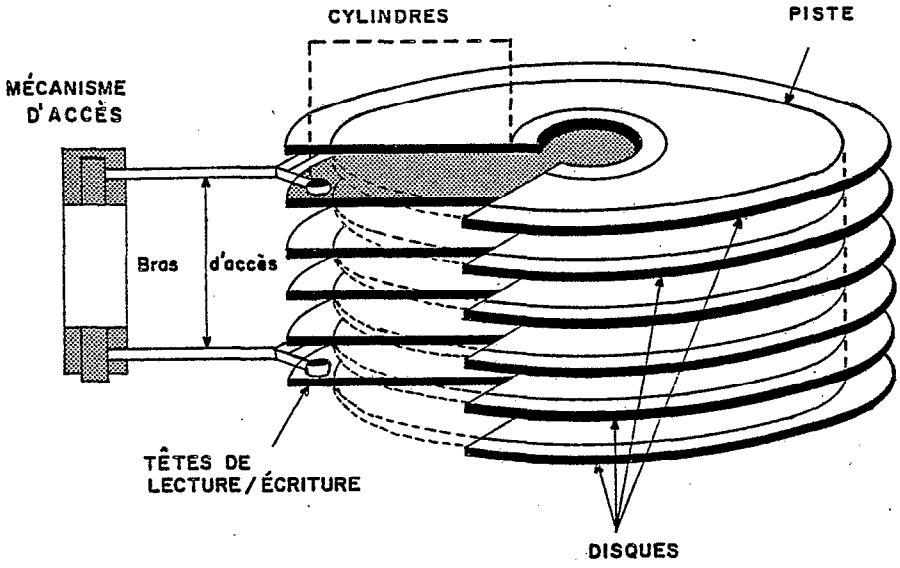


Fig.3.1a) UNITÉ DE DISQUE

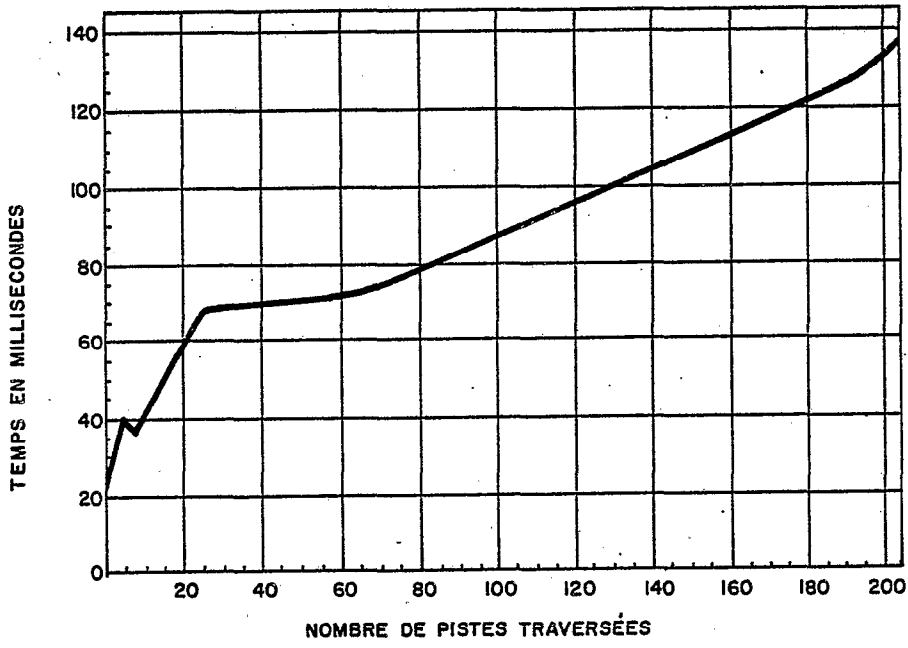


Fig.3.1b) TEMPS DE DÉPLACEMENT DU BRAS ('SEEK-TIME')

- propre transfert du bloc de données, lu ou écrit. La première opération est généralement celle qui consomme le plus de temps, et dépend essentiellement du nombre de cylindres à traverser (voir figure 3.1 b)), bien que l'accéleration est une certaine influence. Le temps de "seek" varie d'un minimum de 25 millisecondes (ms) à un maximum de 130 ms, avec une moyenne proche de 60 ms. La sélection de la tête, réalisée à partir de procédés purement électroniques, consomme un temps toujours négligeable dans la pratique. Le temps moyen de rotation pour atteindre le début de la zone des données correspond évidemment à une demi-rotation, soit 12,5 ms. Enfin le temps nécessaire pour la transmission des données est le temps de passage du bloc de données devant la tête de lecture. Il est usuel d'exprimer ce dernier en fonction de la vitesse de transmission, soit 312 Kilobytes par seconde dans le cas du disque, modèle 2314.

On peut donner un exemple de calcul de temps passé à une opération d'entrée-sortie: supposons que l'on désire lire un enregistrement physique de 2048 bytes, localisé en une position quelconque d'un fichier occupant seulement 6 cylindres, pour le ré-écrire en une autre position de ce même fichier. Sans formuler aucune hypothèse en ce qui concerne la position initiale du bras d'accès, on a un "seek" moyen pour atteindre le cylindre de lecture (60 ms), puis le temps de sélection de la tête (négligeable), puis le temps moyen de rotation (12,5 ms) et le temps de transmission du bloc (6,6 ms). Le temps nécessaire pour l'écriture du même bloc diffère seulement en raison du temps de "seek" inférieur (26,5 ms), dû au fait que les deux cylindres de lecture et d'écriture sont séparés en moyenne de près de deux positions (valeur moyenne de la différence i-j, prise en valeur absolue, pour i et j appartenant à l'ensemble des entiers compris entre 0 et 5). En conséquence, le temps total passé dans l'exemple considéré est de:

$$60 + 12,5 + 6,6 + 26,5 + 6,6 + 12,5 = 125 \text{ ms.}$$

3.12 Les ressources de multiprogrammation et de mémoire virtuelle

La caractéristique principale d'un système opérationnel est d'offrir, ou non, la ressource de multiprogrammation (voir chapitre 5). Cette technique permet l'exécution concommitante, sur un système unique, de plusieurs programmes, situés sur des espaces d'adressages différents, appelés partitions. L'objectif essentiel de cette méthode est de compenser la différence de vitesse de traitement, très importante, existant entre la mémoire centrale et les unités périphériques. En effet, alors qu'un programme en cours d'exécution requiert une opération d'entrée-sortie, le processeur central, au lieu de rester inactif, est libéré pour l'exécution d'un programme dans une partition de priorité plus faible, ce qui permet de diminuer le temps total d'exécution. Dans le système considéré il existe trois partitions, dénommées BG, F2 et F1, dans l'ordre normal de priorité croissante; la partition de plus grande priorité (F1) est réservé à l'activité permanente du programme "spooler" ("POWER/VS"), dont le rôle consiste à gérer pour les autres partitions les opérations d'entrée-sorties à partir des unités lentes ("Spooling"). Ce dernier programme permet de réduire le temps global d'exécution de plusieurs programmes présents dans les autres partitions, en satisfaisant toutes

les opérations d'entrée-sorties sur unités lentes, à une vitesse supérieure qui est celle de l'accès aux files gérées par lui-même sur disque. La lecture des cartes et l'édition des listings sont alors des opérations réalisées en parallèle par le "POWER", durant l'exécution des autres programmes.

A un moment déterminé, seule une partie de l'espace de mémoire allouée à une partition est réellement utilisée par le programme en cours d'exécution. De plus, la taille de chaque partition est fixée au moment de la génération du système, de telle sorte que les programmes qui requièrent une partition supérieure ne peuvent être chargés.

Le système de gestion de mémoire par pagination, bien que non inhérent à la multiprogrammation, est une ressource additionnelle qui se marie fort bien avec cette dernière, pour améliorer l'efficacité globale du système.

La philosophie de base de cette dernière technique (Mongiovi, 1976), consiste à charger seulement une partie des programmes qui doivent être exécutés, étant donné que la plupart des programmes utilise à peine une partie de leur espace adressable à un instant donné de leur exécution. Pour implanter cette méthode, il est indispensable de diviser chaque espace adressable (programme et données, ou "job") en parcelles égales, lesquelles sont appelées 'pages', et de diviser la mémoire réelle en parties de même taille (2K = 2048 bytes), appelés blocs, de telle façon que n'importe quelle page pourra être localisée en n'importe quel bloc disponible.

Au moment de charger le programme, et les données éventuelles, les pages correspondantes sont transférées dans les blocs de la mémoire principale, tant qu'il y a de l'espace disponible. Quand le nombre de blocs devient insuffisant pour contenir toutes les pages à charger, celles-ci sont transférées sur une aire de disque, le "Page data set" (fichier de pages), jusqu'à épuiser le chargement du programme. Il existe une zone spéciale de la mémoire virtuelle, appelée 'SVA' (voir figure 3.2), dans laquelle certaines subroutines fréquemment utilisées, sont chargées de façon permanente, de telle sorte qu'une routine qui fait partie de cet ensemble puisse être exécutée, dans n'importe quelle partition, sans la nécessité d'être chargée à chaque nouvel appel.

Evidemment, pour qu'une instruction donnée soit exécutée, il est indispensable qu'elle soit présente dans la mémoire réelle au moment de l'exécution. Au cas où elle ne s'y trouve pas, l'exécution est interrompue, jusqu'à ce que soit chargée dans la mémoire la page correspondante, grâce à la libération d'un bloc quelconque de mémoire, par élimination d'une autre page, présentement inutile. En conséquence, les blocs qui contiennent les pages d'un même "job", ne restent pas contigus, et le système utilise un algorithme de transformation d'adresses afin que les pages soient logiquement contiguës.

A chaque "job" en cours d'exécution est associée une table de copie de pages, ainsi qu'une table de blocs de mémoire, de telle sorte que l'on sait à chaque instant à quelle partition est affecté un bloc déterminé (voir figure 3.3). Les tables de copie de pages ont une

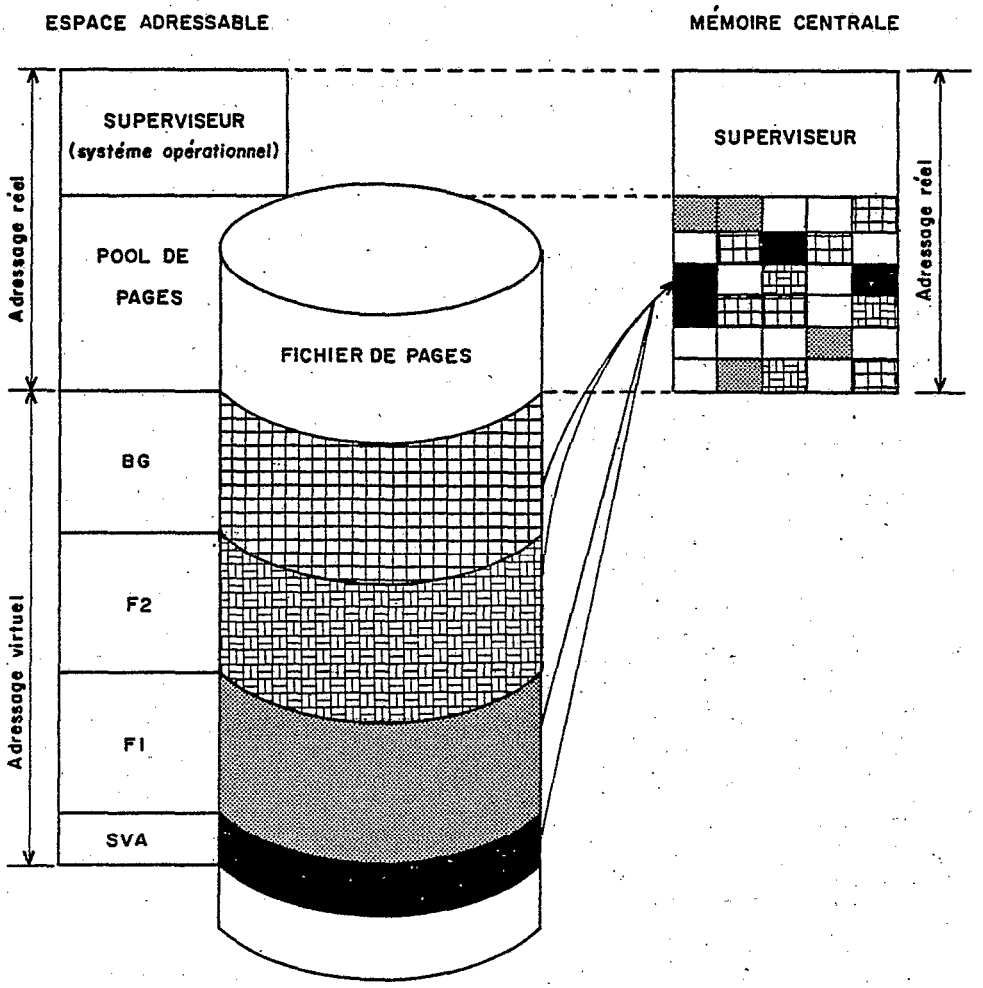


Fig. 3.2 MÉTHODE D'ATTRIBUTION DE MÉMOIRE PAR DEMANDE DE PAGE

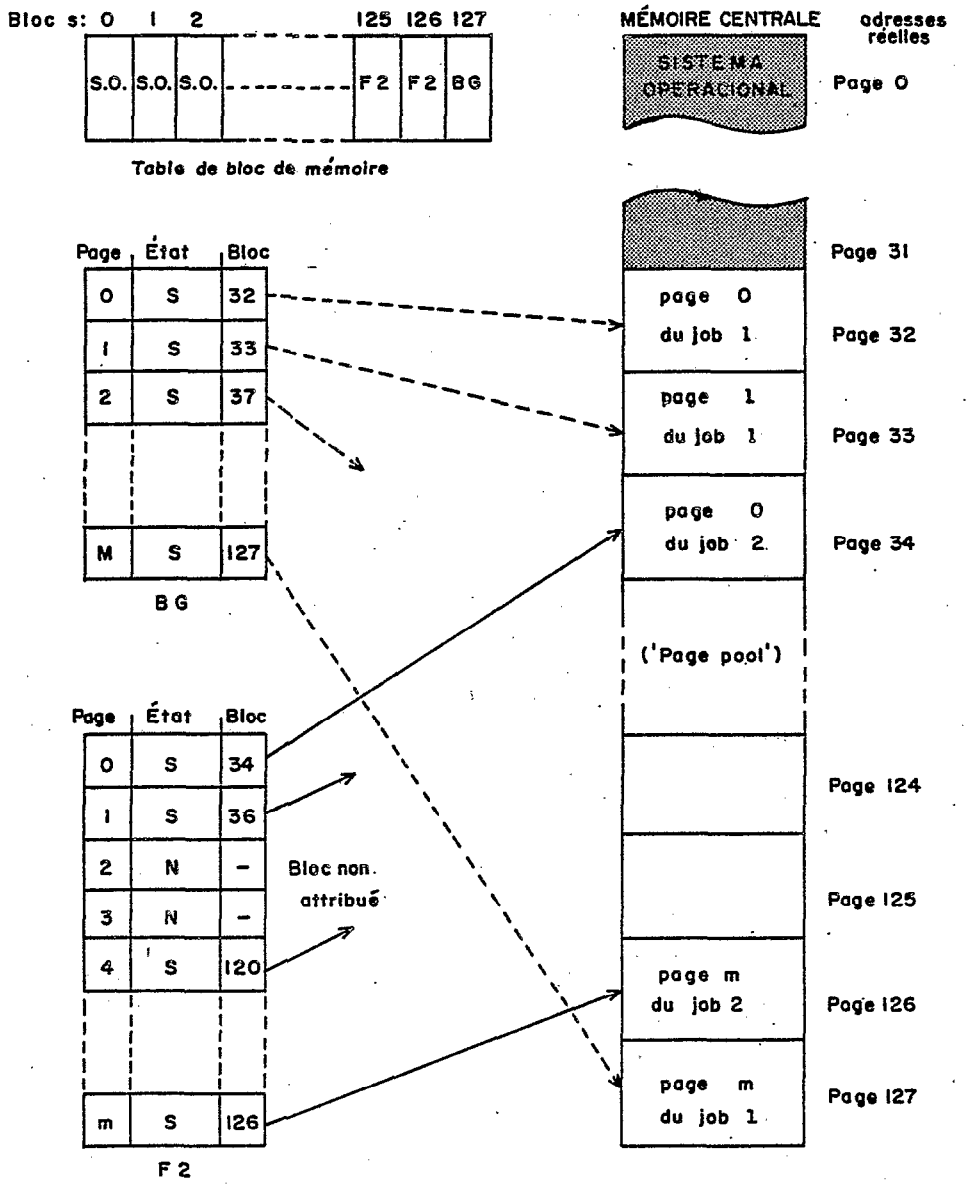


Fig. 3.3 RÔLE DES TABLES DE CONTRÔLE DE PAGES

entrée pour chaque 2K bytes d'espace adressable, laquelle contient diverses données comme l'adresse du bloc de mémoire auxiliaire d'où cette page a été tirée et éventuellement l'adresse du bloc de mémoire principale où cette page a été chargée. D'autres données stockées sous la forme de bits informent le système:

- si la page est réellement présente dans le bloc de mémoire principale considéré (bit d'occupation);

- si la page a fait l'objet d'altérations depuis qu'elle a été chargée. Dans le cas contraire elle pourra être détruite au moment de son exclusion éventuelle, puisque son image existe en mémoire auxiliaire (bit d'altération);

- si la page a été consultée récemment. Dans le cas contraire elle devient susceptible d'être éliminée au moment où un nouveau bloc de mémoire sera nécessaire (compteur de fréquence de consultations).

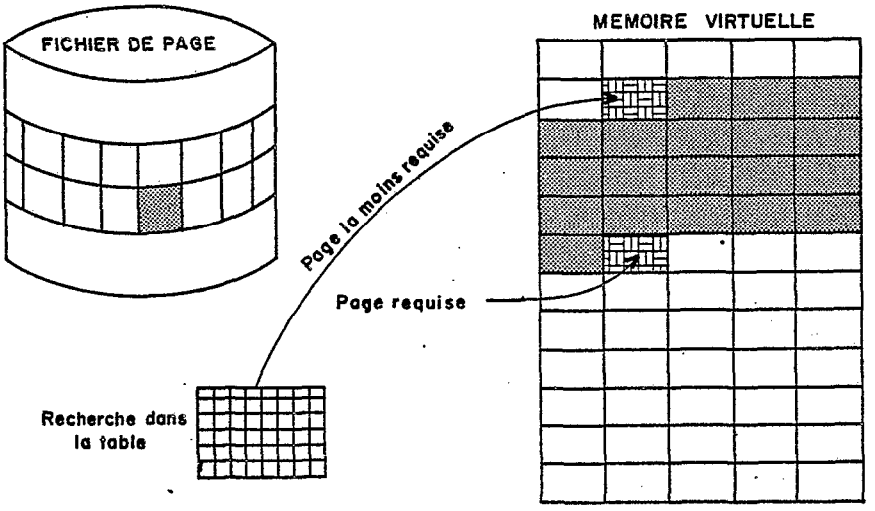
Une demande de pages, généré par un appel à une instruction ou à une donnée contenue dans cette page, implique souvent l'élimination d'une autre page (échange de pages), excepté dans le cas où il existe encore au moins un bloc de mémoire disponible pour recevoir la page requise.

La recherche de la page qui sera éliminée est effectuée à partir de la consultation des tables de copie de pages, sur les bits de plus haut niveau de l'ensemble 'bits d'altération et bits du compteur de fréquence de consultation'.

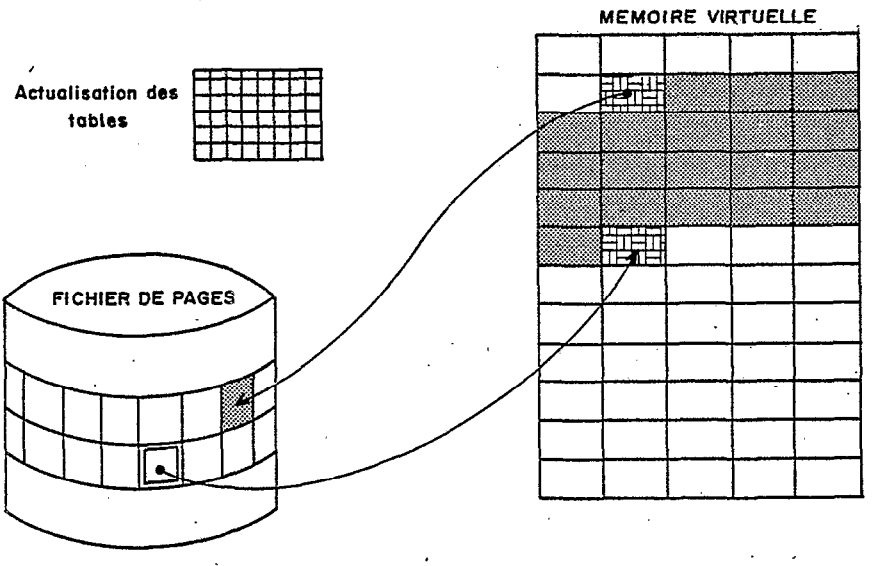
Aux yeux de l'utilisateur ces opérations sont, bien entendu, totalement transparentes: le programmeur a ainsi l'illusion de disposer d'une mémoire quqsi-illimitée. Toutefois, pour l'analyste très préoccupé d'utiliser efficacement la mémoire, le système peut encore être schématisé sous une forme simple. En effet, tout se passe comme si la page contenant les données appelées était échangée avec celle dont l'utilisation est la plus ancienne. Comme il a déjà été signalé, les blocs de mémoire ne sont pas spécifiquement réservés à une partition déterminée et, par conséquent, le nombre R de blocs disponibles pour l'exécution d'un "job" n'est pas constant, mais dépend principalement de l'activité des autres partitions. Cependant, si l'on suppose une activité sensiblement constante des "jobs" exécutés concurremment dans les autres partitions, le phénomène de pagination peut être regardé comme le mouvement d'un ensemble de R 'pages réelles', à l'intérieur d'un autre ensemble de P 'pages virtuelles' (voir figure 3.4), où P * 2K est la dimension en bytes de la partition. Ce mouvement est provoqué par les échanges de pages, eux-mêmes imposés par les appels successifs d'information dans l'espace adressable.

Dans le système de calcul choisi pour l'implantation du programme, on a la distribution suivante de l'espace de mémoire:

- la mémoire réelle est divisée en trois parties, avec 62K réservés pour la zone du système d'exploitation ("Supervisor"), 24K en pages fixées pour la partition F1 du "Power" et les 170K restants pour l'espace commun aux trois partitions ("Page pool");



a) Réquisition d'échange



b) Réalisation de l'échange

Fig. 3.4 Schématisation d'un échange de pages

- la mémoire virtuelle totalise 720K, dont la majeure partie est réservée aux partitions de production, BG et F2, avec 242K chacune. Les 236K qui restent sont répartis entre la 'SVA' (64K) et la partie de mémoire virtuelle exigée par le programme "Power".

Avec le pool de pages de 170K qui doit être réparti pour les trois partitions BG, F2 et F1, le nombre de pages réelles R que l'on peut espérer avoir pour l'exécution d'un programme dans une partition déterminée, peut être estimé à 32 (64K), en supposant une activité moyenne des autres partitions (exécution d'un programme moyen de 60K, par exemple).

Dans ces conditions, il devient important de déterminer, avec la meilleure précision possible, le temps passé à un échange de pages, dans la mesure où le pourcentage des pages réelles par rapport aux pages virtuelles sera en moyenne de 25 à 30%.

Pour obtenir les 720K de mémoire virtuelle, le fichier de pages a été dimensionné avec 120 pistes de disque 2314, étant donné qu'une piste peut contenir jusqu'à trois pages. Un échange de pages est donc réalisé par écriture d'un enregistrement de 2048 bytes, suivi de la lecture d'un enregistrement équivalent, en une autre position d'un fichier dont l'extension totale est de six cylindres consécutifs, étant entendu que l'on néglige toutes les opérations correspondantes en mémoire interne (recherche dans les tables de copie de pages, en particulier). Comme on l'a vu (voir §3.11), le temps passé à cette opération est voisin de 125 ms. Toutefois, en prenant en compte le fait que le fichier de pages jouit d'une situation privilégiée sur le disque qui contient les bibliothèques du système (voir §3.13 et figure 3.5), on peut s'attendre à ce que, au cours d'une exécution de "job", l'essentiel de l'activité du bras d'accès de ce disque reste dédiée au propre fichier de pages, de telle sorte qu'il apparaît raisonnable d'estimer à seulement 45 ms le temps passé au "seek" initial, au lieu de 60 ms en moyenne. De cette manière, l'estimation du temps passé à un échange de pages complet se réduit à 110 ms (et seulement 65 ms, dans le cas où la page éliminée ne requiert pas de "back-up").

3.13 Les bibliothèques du système

Le système dispose d'un certain nombre de bibliothèques (IBM (2), 1973 & IBM (7), 1977 et figure 3.5), dont la finalité est de stocker et pouvoir restituer rapidement toutes les routines qui peuvent être exécutées périodiquement dans le système. En fonction, en particulier, de la périodicité d'exécution de ces routines, on pourra utiliser une, ou plusieurs, des formes de stockage permises par l'ensemble suivant de bibliothèques:

- bibliothèque source ("SOURCE LIBRARY"), qui contient les instructions dans leur langage d'origine, et qui nécessitera par conséquent à chaque récupération (pour exécution), une compilation préalable du programme considéré;

- bibliothèque réallocable ("RELOCATABLE LIBRARY"), qui contient les modules en langage objet et en adressage relatif, et qui

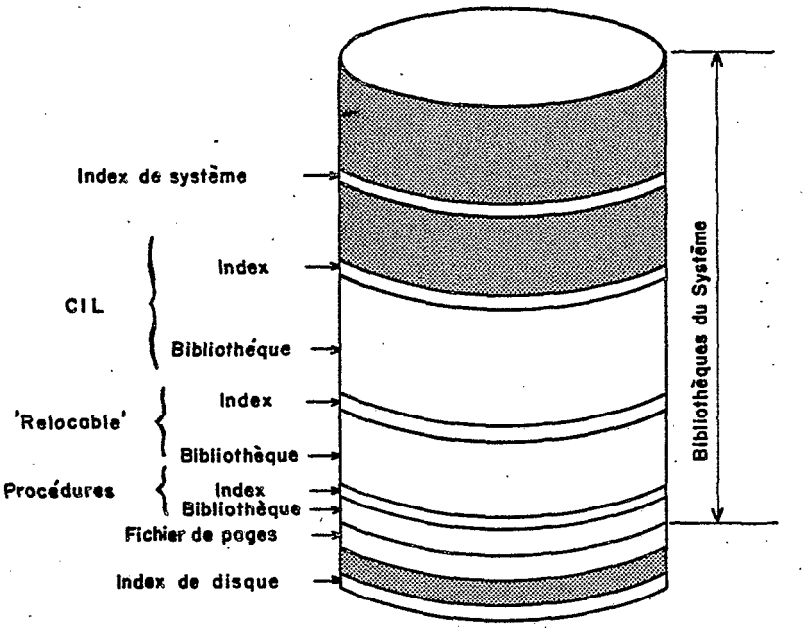


Fig. 3.5 LOCALISATION DES BIBLIOTHÈQUES DU SYSTÈME

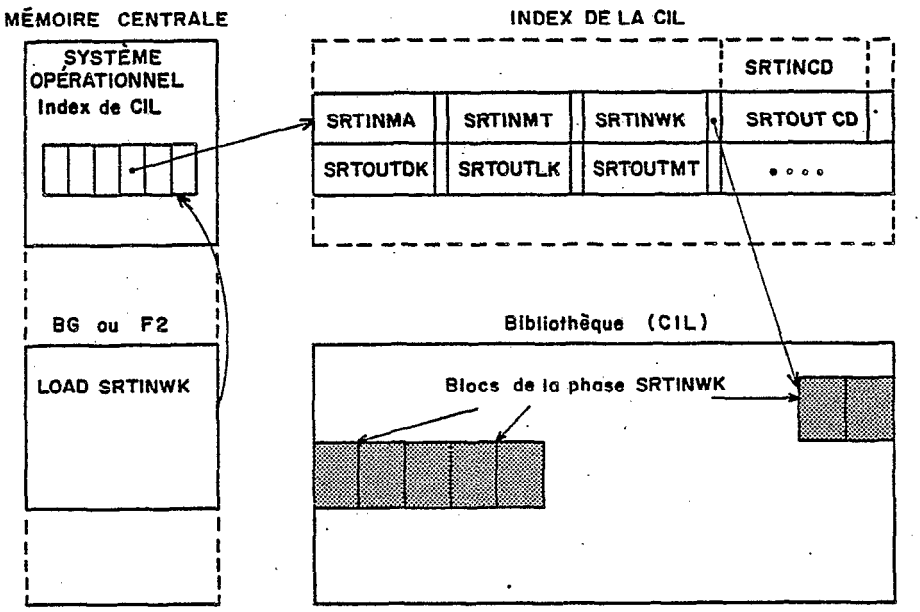


Fig. 3.6 RÉCUPÉRATION D'UNE PHASE DE PROGRAMME

nécessitera par conséquent à chaque récupération (avant exécution), l'édition de liens des modules désirés;

- bibliothèque objet ("CORE-IMAGE LIBRARY", ou CIL), qui contient les phases de programme traduites et en adressage absolu, et qui nécessitera seulement un chargement dans la mémoire pour l'exécution des phases désirées.

Un index est associé à chaque bibliothèque et est situé dans les premières pistes allouées à la bibliothèque considérée. Les entrées de cet index contiennent l'information afférente à chaque élément stocké (voir figure 3.6), telle que le nom, l'adresse sur le disque, la longueur, etc.... De plus, les premières entrées renseignent sur l'espace disponible dans la bibliothèque.

Un programme utilitaire, comme celui de tri, est exécuté fréquemment. Pour cette raison, il est stocké sous une forme qui permet son chargement rapide, c'est à dire en un ensemble de phases cataloguées dans la 'CIL'. En effet, l'aptitude de ce programme a admettre plusieurs types d'entrées et de sorties différentes (voir chapitre 4), est obtenue à partir d'une structure modulaire du programme, grâce à laquelle seules les phases correspondantes au type d'entrée ou de sortie désiré sont chargées successivement, étant donné que celles-ci sont les plus dispendieuses au point de vue de l'espace de mémoire, puisqu'elles contiennent les procédures de 'I/O' ("buffers", en particulier). L'exécution du programme de tri nécessite ainsi un certain nombre de chargements de phases dans la mémoire.

La bibliothèque objet contient généralement un grand nombre de phases de programmes (de l'ordre de 1000, par exemple pour fixer les idées). Pour diminuer le temps de recherche d'une phase déterminée, les entrées de l'index sont classées par ordre alphanumérique: de plus, dans l'espace du "supervisor", un index des derniers noms de phases de chaque piste de l'index de la 'CIL' est maintenu. Ainsi, avec une structure de fichier indexée sur deux niveaux, et en négligeant les temps passés en mémoire centrale; pour localiser une phase, seule une recherche séquentielle sur la piste correspondante de l'index de la 'CIL' sera nécessaire. C'est à dire, en moyenne, le temps de lire neuf enregistrements consécutifs, étant entendu que les entrées sont groupées par bloc de 256 bytes (17 blocs par piste de 2314).

Une fois localisée la position de début de la phase dans la bibliothèque, les blocs (1024 bytes) constitutifs de cette phase sont lus séquentiellement et chargés dans la mémoire, moyennant l'élimination éventuelle d'autres pages. Le temps total passé au chargement d'une phase de 24K, par exemple, sera alors la somme du:

- temps de recherche dans l'espace du "supervisor" (négligeable);

- temps de recherche sur la piste de l'index de la 'CIL', c'est à dire: $60 + 9 * (25 + 0,8) = 290$ ms;

- temps de récupération des 24 blocs (six par piste) qui forment la phase considérée, c'est à dire:

70 + 6 * (25 + 3,3) + 3 * 26 = 310 ms;

- temps de chargement dans la mémoire, négligeable quand on dispose de blocs disponibles (chargement initial d'un programme, par exemple), mais qui fait intervenir le temps des échanges de pages dans le cas contraire (quand la phase qui doit être chargée doit coexister avec des données en place en mémoire centrale, en particulier), c'est à dire: 12 * 110 = 1320 ms.

Ainsi, on arrive à un total de 0,6 seconde dans le premier cas, et près de 2 secondes dans ce dernier cas, ce qui n'est pas du tout négligeable lorsque l'on a à faire à des chargements répétitifs, bien que l'on puisse réduire ces temps par une meilleure situation de la phase, soit dans la 'CIL' (en lui attribuant un nom commençant par la lettre 'A'), soit encore dans la 'SVA' (voir §3.12). En effet, le fait d'appeler les phases fréquemment chargées par un nom commençant par la lettre 'A', permet d'économiser une grande partie du temps de "seek" entre le cylindre contenant l'index de la 'CIL', et le cylindre de la bibliothèque contenant la phase considérée. Cependant, le fait de placer cette dernière dans la 'SVA' serait encore plus économique, étant donné que toutes les phases contenues dans cette zone sont disponibles en permanence dans le "pool" de pages, puisqu'elles sont chargées au moment du début des opérations journalières ("IPL TIME"). De plus, cette localisation conduit à un autre avantage, car ces phases sont disponibles pour n'importe quelle partition, ce qui dispense de cataloguer celles-ci dans les bibliothèques 'CIL' de chaque partition.

Néanmoins, l'aire réservée à la 'SVA' doit être maintenue rigoureusement limitée à la génération du système, afin de ne pas porter préjudice à l'ensemble des "jobs" exécutés, étant entendu que chaque page supplémentaire allouée à la 'SVA', devra être inévitablement tirée du "pool" de pages (voir §3.12). Aussi, généralement les seules routines du fabricant sont-elles cataloguées dans cette zone, notamment celles de gestion des opérations d'entrée-sorties.

Une dernière solution serait de renoncer simplement à une structure segmentée du programme de tri (voir §3.22), moyennant une diminution sensible de l'espace réservé aux données, et en conséquence, de la taille des "runs" formés. Il est clair qu'un compromis doit être recherché entre le gain de temps ou celui d'espace, dans la structure de ce programme.

3.2 REVISION DES ALGORITHMES

Pour atteindre l'objectif principal de cette étude, c'est à dire implanter une méthode efficace de tri pour un fichier non conventionnel, il est indispensable d'analyser minutieusement les conditions spécifiques du système pour lequel ce travail est réalisé. La ressource de la pagination, offerte par la plus grande partie des systèmes importants de calcul, est certainement un exemple parfait d'outil qui doit être manié avec précaution, sous peine de dégradation importante des résultats attendus. En effet, l'application directe

d'une méthode, même réputée très efficace comme le "Heapsort", peut conduire à des résultats qui ne seront pas meilleurs que si l'on avait implanté une méthode triviale, comme celle du "Bubblesort" ou encore de l'insertion directe, méthodes qui sont avantagées dans le contexte de mémoire virtuelle, étant entendu que la différence de classe entre ces algorithmes (en terme de nombre de comparaisons effectuées) pèse peu devant quelques dizaines d'échanges de pages.

En conséquence, les algorithmes qui seront utilisés doivent être révisés, pour être adaptés au nouveau contexte envisagé. Il est clair que la mesure de la performance de la méthode à choisir, passe par l'évaluation du temps passé dans chacune des étapes de l'exécution. Dans l'impossibilité de déterminer expérimentalement ces quantités, ce qui non seulement exigerait un travail prohibitif de programmation, mais encore ne garantirait pas la représentativité des tests, compte tenu de la grande variabilité des conditions d'exécution influencées, par exemple, par l'activité des autres partitions, une étude analytique paraît être la solution.

En principe, la sélection des raffinements à appliquer s'appuie sur le comptage des échanges de pages qui interviennent dans chaque cas, facteur le plus dispendieux de temps. Néanmoins, le choix des structures de stockage sur support auxiliaire est également important, et sera commenté à la fin de ce sous-chapitre.

3.21 Les règles de programmation

Pour mettre en chantier un programme qui doit être exécuté sur une machine à mémoire virtuelle, deux cas se distinguent d'emblée:

- ou bien le programme, et les données correspondantes, tiennent à tout moment de l'exécution dans l'ensemble des pages réelles disponibles de la partition ("Working-set"). Dans ce cas, il est évident que l'exécution ne sera en rien affectée par le phénomène de pagination, en conséquence de quoi les règles d'optimisation des programmes dans un système traditionnel peuvent être conservées;

- ou bien le programme excède la quantité de mémoire réelle allouée, à un moment quelconque de son exécution. Dans ces conditions, un certain nombre d'échanges de pages devront être réalisés, nombre évidemment croissant en même temps que le nombre de pages qui manquent, d'où une dégradation de performance dans l'opération du programme.

Un programme utilitaire de tri mérite un effort de programmation substantiel, pour qu'il soit rendu efficace (IBM (7), 1977). En effet, il s'agit d'un programme qui doit être exécuté très fréquemment, qui requiert un temps de traitement important dans le cas du tri des grands fichiers et qui utilise un espace de mémoire important, principalement pour stocker des "runs" de grande taille. Aussi ce programme possède-t'il toutes les caractéristiques nécessaires pour être analysé avec soin, du point de vue des échanges de pages.

La dimension du "Working-set" détermine si le programme sera, ou non, exécuté efficacement, c'est à dire si celui-ci exigera une

augmentation sensible de temps de calcul par rapport à l'exécution en mémoire réelle. La réduction du "Working-set" à une valeur la plus voisine possible du nombre R de pages réelles disponibles (voir §3.12), est l'objectif de toutes les règles d'optimisation de la programmation dans le contexte de mémoire virtuelle.

Certainement, le moyen le plus évident de satisfaire à cette règle est de réduire l'espace adressable du "job", en réduisant le dimensionnement des zones de données contenues simultanément dans la mémoire. En général, ceci peut conduire à un traitement fractionné des données, et par conséquent à d'autres pertes de temps en opérations annexes (fusion par exemple). On constate ainsi qu'un compromis devra être trouvé entre le traitement de la totalité des données en une seule fois, moyennant des échanges de pages, et le traitement en plusieurs lots, moyennant une opération finale de fusion.

Une seconde technique, pour réduire sensiblement l'ensemble des pages nécessaires à l'exécution d'un programme, consiste à regrouper les appels successifs, de telle sorte que, dans un même intervalle logique de codification, on puisse utiliser le maximum d'information contenue dans une même page. Par exemple, on devra éviter les opérations de branchement, en placant le plus possible les instructions logiquement successives dans la séquence physique du programme. Un nombre important d'échanges de pages pourra être économisé si l'on prend garde de faire appel aux données indexées (vecteurs, matrices) dans l'ordre dans lequel elles apparaissent dans la mémoire. Un appel dans une séquence aléatoire pourra avoir comme effet d'engendrer autant d'échanges de pages qu'il y a d'éléments appelés. Par ailleurs, l'initialisation des paramètres juste avant d'être utilisés, au lieu du début du programme, permet de retarder la réquisition de la page correspondante jusqu'au moment le plus propice. Des sections de programme utilisées fréquemment peuvent être regroupées, de façon à réduire le nombre de pages nécessaires, durant la plus grande partie du temps d'exécution. En effet, une page a une capacité d'environ 500 instructions (chaque instruction élémentaire occupant, en moyenne, quatre bytes), et par conséquent trois routines, même d'encombrement réduit, localisées dans des endroits séparés du programme, peuvent exiger la présence en mémoire réelle de trois pages, alors qu'il aurait peut-être suffit d'une seule page, si elles avaient été placées en séquence physique dans le programme.

Les autres techniques qui peuvent être utilisées pour optimiser la programmation sur un ordinateur à mémoire virtuelle, sont seulement accessibles à qui programme dans un langage de bas niveau. Elles consistent en premier lieu à stocker les constantes fréquemment utilisées dans des zones proches des instructions qui les utilisent, au lieu de la pratique traditionnelle qui consiste à réserver une aire de données à la fin du programme. On doit par ailleurs essayer de charger dans des zones séparées les parties de codification susceptibles d'être modifiées au cours de l'exécution, des autres parties, de manière à réduire le nombre de pages altérées (et ainsi le nombre de "back-ups" dans le fichier de pages).

L'alignement des adresses des "buffers" par rapport aux limites de pages (multiples de 2K), doit également économiser les échanges. L'utilisation de certaines routines permet d'influencer le mécanisme

de pagination, en particulier en fixant certaines pages virtuelles dans les blocs de mémoire réelle, réservés dans la partition d'exécution. De cette façon, le programmeur peut conserver en mémoire réelle une partie de son programme, dont la libération serait inopportune.

3.22 L'organisation générale du programme de tri

En fonction du type des fichiers d'entrée ou de sortie à mettre en oeuvre dans l'opération de tri, certaines instructions et données doivent être modifiées. Ce sont non seulement les zones de "buffers" allouées (par exemple une entrée sur cartes perforées nécessitera d'un seul "buffer" de 80 positions, tandis qu'une entrée d'un fichier indexé sur disque peut nécessiter plusieurs zones de jusqu'à 7294 bytes) qui sont différents, mais aussi les propres déclarations de description des fichiers ("DTF Declarative Macros"), et jusqu'à la forme des commandes de lecture et d'écriture ("Imperative Macros"). Pour éviter un chargement dans la mémoire d'un programme très encombrant, et avec une grande part de sa codification inutile, on cherchera à le segmenter avec une phase d'entrée de données et une autre phase de sortie des données (voir figure 3.7), ces phases étant récupérées en temps d'exécution.

L'utilisateur doit inévitablement instruire le système sur un certain nombre de valeurs de paramètres, pour définir l'opération qu'il désire réaliser (type d'entrée, type de sortie, champs de tri, etc...). Ces paramètres doivent être validés, c'est à dire que les incohérences éventuelles doivent être détectées: l'opération ne se poursuivra que si toutes les valeurs de paramètres ont été reconnues valables. En même temps ces paramètres doivent être interprétés, pour que soit déterminé le type de tri à effectuer (tri interne ou tri externe), ainsi que la méthode à choisir (cas du tri externe). Toute cette partie de codification, assez importante, n'a pas besoin de coexister avec les données dans la mémoire, et, en conséquence, le programme peut être amputé de cette partie, traitée comme une phase, également chargée (puis effacée) en temps d'exécution.

Finalement, l'isolement d'une phase de travail destinée à la classification proprement dite avec les données 'in situ', permet de profiter pleinement du gain d'espace créé par l'utilisation (par ailleurs indispensable) de techniques de segmentation de programmes. En effet, en substituant la taille mémoire nécessaire à la somme des espaces adressables des phases d'entrée, de travail et de sortie, par celle de la plus grande de ces trois phases, on libère un espace plus important pour le stockage des données qui peuvent être ordonnées en une seule fois, de telle sorte que la possibilité de tri interne devient plus importante.

Dans le cas du tri externe, le problème d'économie d'espace n'est pas si important, étant donné que la phase de fusion demande une durée pratiquement indépendante du nombre de "runs" à intercaler (voir §§2.43 & 2.44). Aussi, au prix d'une diminution encore raisonnable de la taille des "runs" (de l'ordre de 20K), les chargements successifs des phases d'entrée, de travail (tri) et de sortie des données (distribution des "runs" sur les aires de travail), pourront être

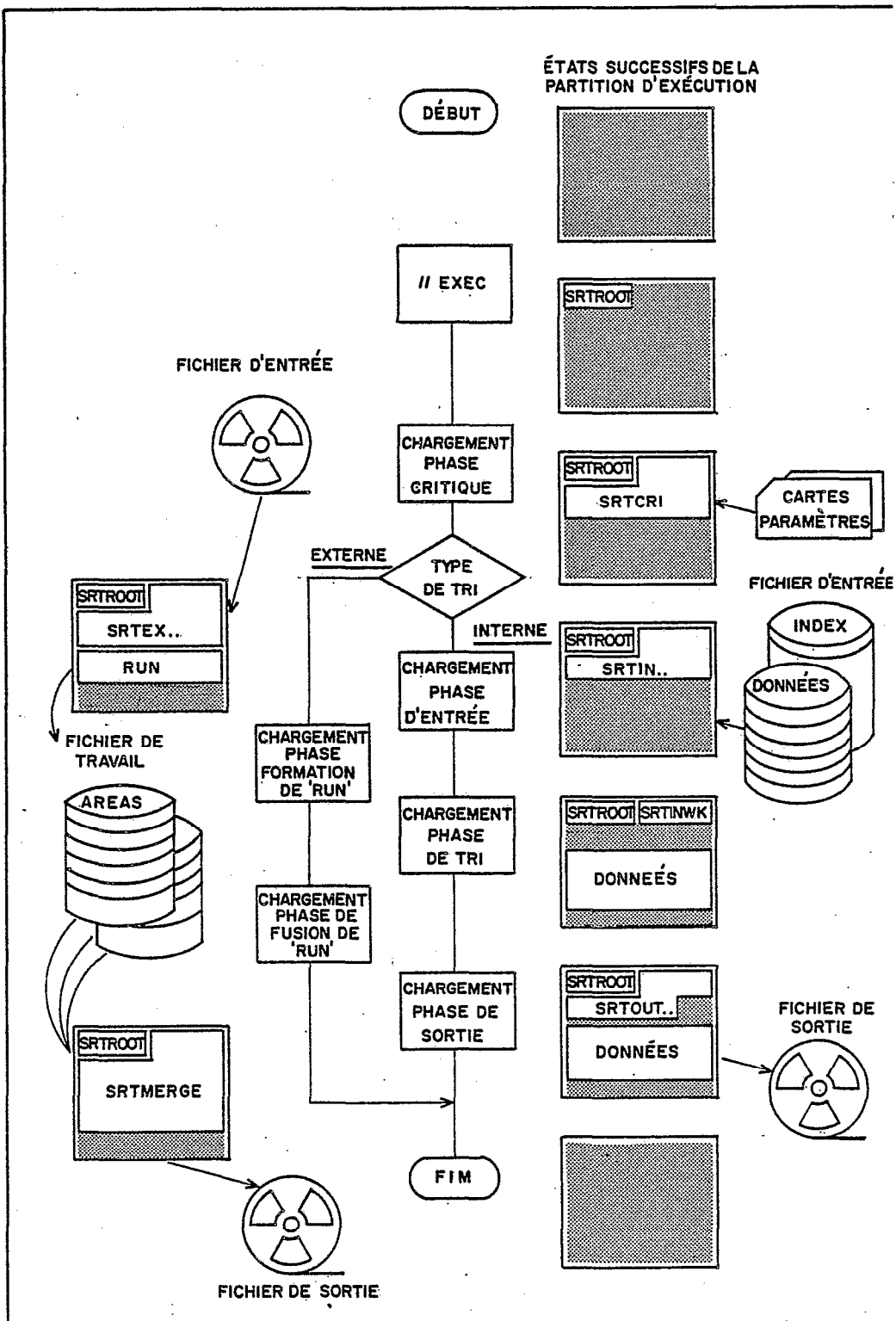


Fig. 3.7 STRUCTURE DU PROGRAMME DE TRI

économisés en joignant le tout en une phase unique. De cette façon, le tri externe est exécuté à partir de deux phases, seulement:

- la phase de distribution des "runs", chargée après une recherche du type d'entrée;

- la phase de fusion des "runs", chargée après une recherche du type de sortie.

L'utilisation de cette segmentation de programme, pour construire un certain assemblage de phases au cours de l'exécution, ne peut être effectivement programmé qu'en utilisant un langage de bas niveau. Le prix payé pour l'utilisation de cette ressource, qui est la fixation de l'équipement pour lequel ce programme est développé, est compensé par l'augmentation de l'efficacité et de l'adaptabilité obtenues avec l'utilisation de ressources de programmation plus puissantes (Brawn et al., 1970), comme on le verra au chapitre suivant.

L'idée de tri indirect déjà présentée (voir §2.13 & figure 2.3), appliquée au seul stockage des clés dans la mémoire, avec un pointeur pour l'information correspondante, conduit à l'élaboration du graphique 3.8. Avec la longueur T des clés en abscisses (exprimée en bytes), et le nombre N d'enregistrements à trier en ordonnées, le plan cartésien peut être divisé en deux parties, par la courbe représentative de l'hyperbole:

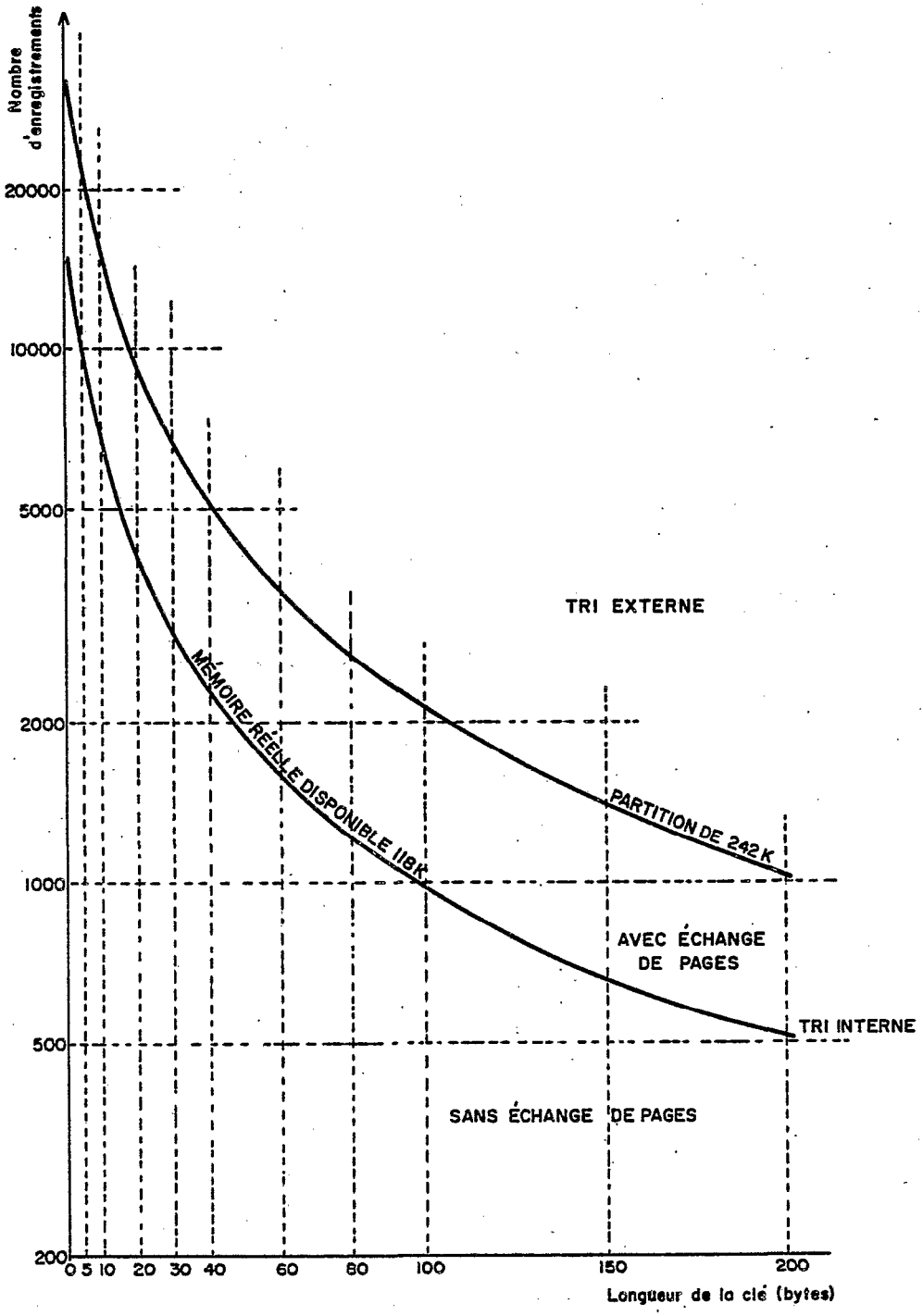
$N * (T + p) = P$, où p représente le nombre de bytes nécessaire au pointeur (p sera généralement une adresse d'enregistrement stocké sur disque), et P la taille de mémoire disponible dans la partition pour stocker les données (c'est à dire la taille totale de la partition, diminuée de l'espace occupé par la plus grande des phases qui doivent coexister avec les données).

Evidemment, une autre hyperbole peut être établie avec la valeur R du nombre de pages réelles disponibles, et sépare le domaine du tri interne en deux zones: sans échange ou avec échanges de pages. Le graphique 3.8 a été établi avec les valeurs:

$P = 242K$ et $R = 118K$, lesquelles correspondent à l'exécution du programme de tri lorsque l'activité de l'autre région de production est nulle. Pour donner un exemple d'utilisation de cet abaque, on peut observer que si 2000 enregistrements avec des clés de 40 positions peuvent être triés sans échange de pages, jusqu'à 5000 de ces mêmes enregistrements pourront l'être, mais avec échanges de pages; au-delà, le tri devient obligatoirement externe.

3.23 Hypothèses pour les calculs d'échanges de pages

Les paragraphes suivants sont essentiellement consacrés à la définition finale des méthodes de tri, tant interne qu'externe, qui seront utilisées pour une efficacité maximale du programme projeté. Aussi, dans la mesure du possible, les temps d'exécution pour chaque opération doivent être évalués, et en particulier les temps consacrés aux échanges de pages, desquels les temps de traitement sont directement proportionnels. Les calculs des nombres d'échanges de



GRAPHIQUE - 3.8
DÉTERMINATION DU TYPE DE TRI

pages, étant donnée la complexité du phénomène de pagination dans un contexte de multiprogrammation, pourront seulement être possibles moyennant un ensemble d'hypothèses générales pour simplifier la détermination analytique.

Les premières hypothèses qui doivent être formulées concernent l'algorithme d'échanges de pages lui-même. Il existe plusieurs procédés pour déterminer quelle doit être la page à éliminer (Madnick et al., 1974, Mongiovi, 1976), comme on le verra au chapitre 5, lesquels dépendent du choix du constructeur. On s'appuie souvent sur un algorithme capable de déterminer quelle est la page qui est restée le plus longtemps dans la mémoire sans souffrir d'altération. On admet également que l'algorithme ne permet pas de reconnaître, lors d'un nouveau chargement de programme ou de données, si la page qui devra recevoir les données est, ou non, présente dans la mémoire, de telle sorte que l'altération correspondante est faite via échange de pages, et non directement dans la mémoire réelle à partir de la recherche dans la table de copie de pages du "job" en cours d'exécution. Ces hypothèses sont valables dans le cas expérimental considéré: pour un autre type d'équipement, un algorithme certainement différent pourra conduire à une divergence sensible dans les résultats obtenus, en termes de nombres d'échanges de pages.

Comme il a déjà été vu (voir §3.12),les échanges de pages peuvent être complets ou incomplets, en fonction de ce que la page à éliminer a été altérée, ou non, et par conséquent nécessite, ou non, d'être recopiée dans le fichier de pages. En toute rigueur, le comptage des échanges de pages dans les deux cas doivent être séparés, étant entendu que les temps passés dans les deux cas sont sensiblement différents (respectivement 110 et 65 ms). Toutefois, il paraît raisonnable de ne pas maintenir cette distinction, vu que la complexité additionnelle résultante ne compenserait pas le gain de précision obtenu, dans les conditions générales de précision de ces calculs empiriques. La moyenne de 100 ms, adoptée pour le temps passé à un échange de pages, est évaluée d'une façon approximative, en considérant que le nombre d'échanges incomplets, dans la mesure où le programme serait 'bien fait' (c'est à dire, tel qu'une page de données appelée est, le plus souvent, effectivement altérée lors de son passage en mémoire réelle), et que le nombre de pages réelles n'est pas trop faible, est sensiblement inférieur au nombre d'échanges complets.

Une autre hypothèse générique concerne l'interaction entre les diverses partitions. L'ensemble des pages réelles n'est pas réparti de forme fixe pour chaque partition (voir figure 3.4): c'est le système opérationnel lui-même qui choisit un bloc de mémoire disponible pour y allouer une 'page virtuelle' appelée. Aussi, le nombre R de 'pages réelles' disponibles pour la partition d'exécution varie-t'il selon l'activité des autres régions. Statistiquement, en supposant que les autres régions maintiennent une activité sensiblement constante, tout se passe comme si une page requise dans la partition considérée, alors qu'il n'y a plus de bloc de mémoire disponible, provoquait l'élimination d'une page de la même partition. En condition normale d'exploitation du système (multiprogrammation), cette règle fondamentale est vraie globalement seulement: elle devient par contre

rigoureuse dans le cas où le programme est exécuté, en mode virtuel, en régime de monoprogrammation.

Les autres hypothèses sont intrinsèques au programme de tri. Elles peuvent être énoncées selon la séquence suivante:

- hypothèse sur la valeur de R: on suppose le nombre de pages réelles disponibles au moins supérieur au nombre requis pour la zone de programme (c'est à dire environ 20K bytes), de façon que le phénomène de pagination n'affecte essentiellement que la partie des données;

- hypothèse sur la longueur des clés: on suppose que la longueur des entrées dans la zone de données, constituées de la clé de tri concaténée avec le pointeur pour l'information, est faible par rapport à la taille d'une page, de manière que la probabilité que l'appel à une clé puisse conduire à l'appel de deux pages (logiquement contiguës) soit faible. Pour donner un exemple, avec une clé de 27 bytes et un pointeur de 5 bytes, la probabilité que la référence à une clé induise un double échange de pages est de:

32 / 2048 = 1 / 64 , soit environ 1,6%. Il est important de remarquer que cette probabilité peut aller jusqu'à s'annuler, si l'on prend soin d'aligner la zone des données avec les limites de pages;

- hypothèse de fixation des pages: pour convenance de calcul, on supposera que certaines pages sont fixées à certains moments (ce qui peut fort bien être programmé), de telle sorte que certains facteurs aléatoires annexes de mouvement de pages soient éliminés.

3.24 Révision de l'algorithme de tri interne

L'algorithme de tri interne choisi, le "Quicksort" (voir §§2.34 & 2.39), a été sélectionné, en grande partie, grâce à son efficacité présumée dans le contexte de mémoire virtuelle (voir tableau 2.1). Parmi les algorithmes efficaces présentés dans le sous-chapitre 2.3, le "Quicksort" est le seul à suivre une philosophie de recherche séquentielle, et par conséquent le seul à respecter les règles fondamentales de programmation dans ce même contexte. Le paragraphe ci-après permet de justifier quantitativement cette affirmation. Pour l'instant on se bornera à déterminer le nombre d'échanges de pages qui interviennent dans une opération de tri réalisée selon cette méthode.

La figure 3.9 montre les diverses images de la partition durant l'exécution de cette étape, conformément à la schématisation donnée antérieurement (voir §3.12 & figure 3.4). L'exemple choisi présente un ensemble de R = 7 pages réelles (foncées), dans une partition de P = 12 pages virtuelles. Si l'on suppose que la phase de chargement des données nécessite de trois pages, alors que la phase de travail n'en a besoin que d'une, neuf pages subsistent pour la zone de données, que l'on suppose entièrement remplie.

Dans l'état a), après le chargement, les pages réelles se rencontrent dans les dernières positions virtuelles, vu que les données sont stockées séquentiellement à partir du début de la zone.

Dans la réalité, ceci signifie que les dernières pages appelées sont celles dont l'adresse est la plus grande, et en conséquence celles qui restent, à la fin de l'étape, dans les blocs de mémoire réelle. A ce moment, la phase de tri est chargée dans la première page, et les deux pages suivantes ne seront plus référencées dans cette étape. L'état b) est atteint après la recherche de la clé-pivot par application de la méthode du "median", à partir de l'appel de la première page de la zone de données (clé K(1)), de la page du milieu de cette zone (clé K(N+1)/2), et de la page finale (clé K(N)).

Cette opération nécessite l'appel aux pages A2 et B3, qui seront allouées, moyennant l'élimination des pages dont l'appel est le plus ancien, c'est à dire les pages C3 et A4. L'état c) est obtenu durant le premier pas du "Quicksort", à la fin d'une première phase. En effet, rappelons que la méthode réalise une lecture de la liste à partir des deux extrémités: dans un premier temps les référenciations successives du début de la zone (pages A2, B2,...), imposent des échanges de pages, qui sont obtenues à partir de la seconde partie de la zone et, en moindre quantité, de la zone du programme jusqu'à ce que les propres appels dans la seconde partie de la zone empêchent la continuation de ce processus. Cette condition est atteinte lorsque l'on a le même nombre de pages de chaque côté de la zone des données, c'est à dire R/2.

La fin du procédé de double lecture provoque le transfert des pages du début et de la fin de la partition (aussi anciennes dans la mémoire), vers la partie médiane de la zone, où les deux indices doivent se rejoindre, quand la configuration de la région est celle présentée en d).

Au moment de commencer le second pas, on suppose que les pages réelles allouées à la zone de programme au moment de la phase d'entrée, et non utilisées depuis le début de la phase de tri, ont été récupérées. En admettant également que la liste des clés ait été idéalement séparée en deux sous-listes de même taille, grâce à un choix heureux de la clé-pivot, la situation e) de la mémoire au début du second pas est très semblable à la situation a) initiale, vu que, la zone ayant été réduite de moitié, la moitié des pages réelles sont localisées à la fin de cette nouvelle zone. Cependant, une différence essentielle vient de ce que l'on dispose de la même quantité additionnelle de pages réelles de l'autre côté de la zone, et qui ne seront pas immédiatement appelées.

L'exécution successive des étapes suivantes devra rapidement conduire à un état de la mémoire, tel que f), dans lequel l'exécution des opérations est entièrement réalisée dans la mémoire réelle, étant donné que la taille des sous-fichiers correspondants est inférieure à R. Ce fait est important, dans la mesure où chaque étape devra donner lieu à un certain nombre d'échanges de pages et, par conséquent, la croissance du rapport entre la taille Q du "run" en cours de tri et le nombre R de pages réelles, devra provoquer une augmentation plus grande encore du nombre total d'échanges de pages, étant donné que le tri devra nécessiter chaque fois plus d'étapes.

Avant de commencer les calculs analytiques, on peut observer que:

- le raffinement de la méthode du "Quicksort" proposé par Sedgewick en 1975 (Meyer et al., 1978 & Sedgewick, 1975), comme on l'a vu au paragraphe 2.39, c'est à dire de laisser les sous-fichiers de petite dimension (qui seront ordonnés par la méthode d'insertion directe) pour une étape finale sur la totalité du fichier, devra être abandonnée dans le cas de l'utilisation de la mémoire virtuelle. En effet, en les traitant au moment où ils sont rencontrés, il n'y aura pas de réquisition supplémentaire de pages, alors qu'une étape complémentaire induit un mouvement complet des R pages réelles dans la zone Q;

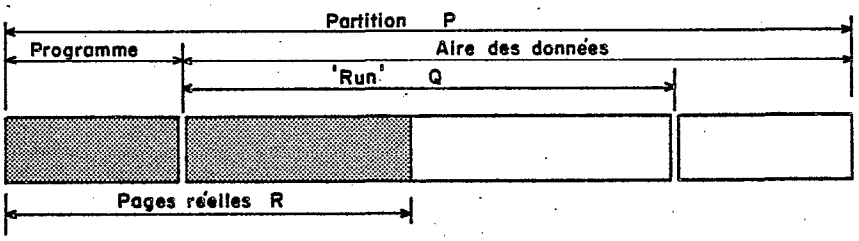
- le choix, après une division du fichier, du sous-fichier le plus grand pour placer dans la pile, conduirait à une séquence aléatoire de translations du groupe de pages réelles dans l'aire des données. L'empilage systématique du sous-fichier situé dans la partie la plus voisine du début de la zone des données, garantit une translation continue de ce groupe, de sorte qu'à la fin de l'opération les pages réelles se trouvent forcément au début de la zone de données. Ceci devra favoriser la phase suivante de déchargement, en réduisant le nombre d'échanges nécessaires. Pour compenser l'augmentation éventuelle du nombre de pas provoquée par la suppression de ce raffinement, il est possible d'augmenter la taille de la pile double (voir §2.34), d'augmenter également la taille de l'échantillon réalisé par recherche du "median", et d'augmenter enfin la taille des sous-fichiers qui doivent être triés par insertion directe.

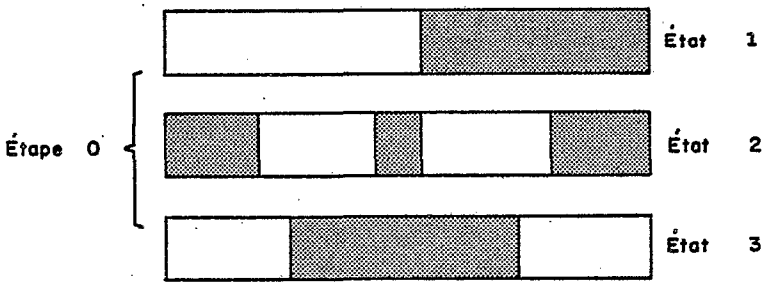
Le calcul théorique du nombre d'échanges de pages qui interviennent dans le tri, par la méthode du "Quicksort", d'un "run" de dimension Q alors que l'on ne dispose que de R pages réelles, peut être réalisé, moyennant les hypothèses formulées antérieurement (voir §3.23), à partir de la situation de la partition à la fin du chargement de ces données dans la zone, jusqu'à une situation définie avec précision (les R pages au début de la zone), juste avant le déchargement des données.

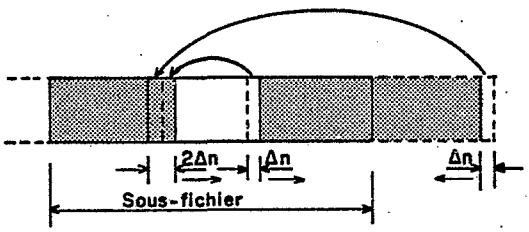
La figure 3.10, qui utilise une représentation linéaire de l'espace adressable, permet d'accompagner le développement des calculs. Dans la mesure où le programme de chargement a nécessité de 20K (10 pages), la quantité réelle de pages allouées pour la zone des données est:

R - 10 , lors du début du pas 0 du "Quicksort".

Bien sûr, le nombre d'échanges qui interviennent durant la division d'un sous-fichier de taille q quelconque, dépend seulement de la position initiale des 2r pages réelles, à l'intérieur de l'espace adressable défini par ce sous-fichier, et de l'ancienneté relative de chaque page de cet ensemble, une fois fixées les valeurs relatives de r et q. Compte tenu de la systématique d'empilage des sous-fichiers successifs, à chaque pas, et pour le traitement de la routine 'PARTICAO' de chaque sous-fichier, on peut déterminer sans ambiguïté ces deux caractéristiques. Pour préciser les valeurs obtenues pour q, la formulation de l'hypothèse d'équidivision devient nécessaire, ce qui apparaît d'ailleurs convenable, étant entendu que pour d'autres







(Séquence de traitement des sous-fichiers)

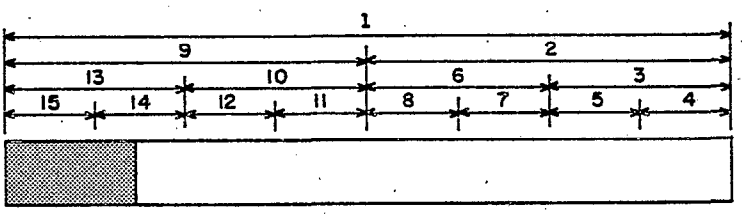


Fig. 3.10 CALCUL DU NOMBRE D'ÉCHANGES DE PAGES DU QUICKSORT

raisons (voir §2.39) on s'efforce d'obtenir ce cas idéal. En conséquence, les valeurs successives de q seront:

$$q = Q / 2^i \quad , \text{ où } i = 0, 1, 2, \ldots.$$

Dans la pratique, la valeur de Q par rapport à R est limitée, sous peine d'une extrême dégradation de l'efficacité du système. Pour convenance des calculs, la valeur limite de Q, Q_{max} telle que:

$$Q_{max} = 8 * (R - 2) \quad ,$$

paraît surdimensionnée (à partir du moment où l'on dispose de 32 pages réelles, Q_{max} atteint 480K).

On observe, dans la séquence des opérations d'empilage illustrée à la fin de la figure 3.10, qu'il existe trois cas distincts à considérer, pour ce qui est de la position des pages réelles avant chaque exécution de la routine 'PARTICAO':

- les pages réelles sont entièrement inclues dans la zone à diviser. C'est seulement le cas pour l'étape 0 (fichier 1), pour laquelle l'état initial est parfaitement connu, vu que cette étape est consécutive au chargement des données;

- les pages sont situées pour moitié dans la zone (au début, de toutes façons). L'autre moitié, à l'extérieur, est constituée de pages aussi anciennes. Cette situation s'obtient quand on vient de terminer le traitement d'un grand sous-fichier, qui incluait celui qui va être traité. En effet, à la fin du traitement du fichier 1, par exemple (fin de l'étape 0), les pages réelles sont situées dans la partie centrale du fichier 1, et en conséquence, pour moitié au début du sous-fichier 2 et pour moitié à la fin du sous-fichier 9. Ce cas est aussi celui du traitement des fichiers: 3, 4, 7, 10, 11 et 14;

- les pages sont situées partiellement (ou totalement) en dehors du fichier qui va être traité, et les pages éventuellement contenues à l'intérieur sont les plus anciennes et, par conséquent, les premières à être éliminées. Ceci survient lorsque l'on termine le traitement d'un fichier situé à droite de celui qui va être traité. C'est le cas des fichiers: 5, 6, 8, 9, 12, 13 et 15.

Dans l'étape 0 il y a:

(R - 10) / 2 échanges de pages pour atteindre l'état 2 (voir figure 3.10), et encore:

Q - (R - 10) échanges pour atteindre l'état 3, d'où un total d'échanges de pages de:

$$T = Q - (R - 10) / 2.$$

En supposant qu'à la fin de l'étape les 8 pages correspondant à la différence de taille entre les phases d'entrée et de travail du programme ont été libérées instantanément, au cours du pas j pour un sous-fichier quelconque on dispose de:

$2r = R - 2$ pages réelles pour traiter le fichier de dimension:

$q = Q / 2^j$. Les mouvements des pages réelles au cours de cette étape (réquisition des pages du début de la zone, aux dépens des pages de fin de zone) conduit à examiner deux cas, en fonction des valeurs respectives de q et r. En effet, ou bien le fichier sera entièrement 'couvert' par les pages réelles avant que la lecture séquentielle à droite ne réquisitionne les pages qui ont été rendues pour satisfaire la lecture séquentielle à gauche, ou bien, au contraire, il reste partiellement 'découvert'.

Dans le premier cas, le nombre d'échanges $2D_n$ vérifie l'équation:

$$2D_n + r - D_n = q \quad ,$$

qui traduit le fait que la somme des pages contenues à gauche et à droite 'couvre' complètement les q pages requises. Dans la second cas, on a une première étape avec:

$$2D_n = 2r / 3$$

échanges pour atteindre la situation 2, lorsque le nombre des pages situées à droite et à gauche de la zone q est le même. Avec $2r / 3$ échanges supplémentaires, on obtient les 2r pages réelles dans la zone, et finalement $q - 2r$ échanges sont encore nécessaires pour terminer l'étape.

Evidemment, la limite entre les deux cas est atteinte quand le total des q pages virtuelles sont allouées à la fin de la première étape (situation 2), c'est à dire quand D_n vérifie le système d'équations:

$$D_n = q - r$$

$$D_n = r / 3$$

qui conduit à la solution: $q = 4r / 3$. En conclusion, le nombre d'échanges T_j qui ont lieu à l'étape j, réalisée sur un fichier de type 1 est:

- si $q < r$, alors $T_j = 0$;
- si $r < q < 4r / 3$, alors $T_j = 2 * (q - r)$;
- si $q > 4R / 3$, alors $T_j = q - 2r / 3$.

Dans la même étape j, pour un fichier de type 2, on suppose que les 2r pages réelles sont réparties en x pages dans le fichier lui-même, et y dans le sous-fichier situé à droite, qui vient d'être traité. Il est important de noter ici que c'est seulement pour les sous-fichiers plus petits que la valeur de x est non nulle. Naturellement, ces x pages sont celles qui ont été laissées au cours de l'étape antérieure (par exemple, lors du traitement du fichier 5, ces pages sont celles laissées après le traitement du fichier 4, dans les positions atteintes à la fin du traitement du fichier 3), et qui

restaient du traitement du fichier adjacent, ce qui seulement peut arriver dans la dernière étape.

Pour x = 0, c'est à dire quand la zone du fichier q ne contient aucune page réelle, le nombre d'échanges nécéssaires au cours de l'étape est q. Lorsque x est différent de 0, les conditions suivantes sont vérifiées:

- y = q, parce que le fichier adjacent, à droite, vient d'être traité;

- x + q = 2r, nombre de pages réelles disponibles dans la zone des données;

- x > q, vu que cette dernière étape a été nécessaire.

En vertu de ces conditions, et en rappelant que ces x pages sont les plus anciennes, la 'couverture' du fichier q sera atteinte seulement après un nombre d'échanges de pages égal à:

$$x / 2 + q - x = q - x / 2.$$

Ce résultat se retrouve facilement si l'on considère seulement les pages réelles qui peuvent être utilisées en place.

La succession des discontinuités dans la pente du graphique représentatif de la fonction T = f(R,Q), résulte de l'intervention successive des nouveaux pas qui réquièrent des échanges de pages. Les limites de chaque intervalle appartiennent à l'un des deux groupes suivants:

- $Q_{1i} = (R - 2) * 2^i$; $i = 0, 1, 2, 3$,

au-delà duquel l'étape i commence à demander des échanges de pages;

- $Q_{2i} = (4 / 3) * (R - 2) * 2^i$; $i = 0, 1, 2, 3$,

au-delà duquel les fichiers de type 1 demandent plus d'échanges de pages.

Si l'on observe que pour les valeurs de Q inférieures à la quantité (R - 10), le nombre d'échanges est nul, étant donné que les blocs de mémoire disponibles sont en nombre suffisant pour recevoir toutes les pages virtuelles, tant dans la zone du programme que dans celle des données, on peut compléter le tableau 3.1. Il est important de noter que pour chaque sous-fichier de taille q, le nombre d'échanges qui interviennent dans un pas du "Quicksort" sur ce fichier, est inférieur à la valeur de q et, évidemment, le nombre total d'échanges qui interviennent pour i pas, est inférieur à i * q.

Le graphique 3.11 a été établi pour un nombre de blocs de mémoire disponibles à la région d'exécution (R) égal à 32, et montre qu'avec une taille de "run" (Q) de 111 pages (valeur maximale qui peut être obtenue dans la partition de 242K, compte tenu des 20K réservés pour le programme), le nombre d'échanges de pages atteint 290. En rappelant

	ETAPE Ø	ETAPE 1		ETAPE 2		ETAPE 3		NOMBRE TOTAL D'ECHANGES
		FICHIER TYPE 1	FICHIER TYPE 2	FICHIER TYPE 1	FICHIER TYPE 2	FICHIER TYPE 1	FICHIER TYPE 2	
$0 \leq Q \leq R-10$	Ø	—	—	—	—	—	—	Ø
$R-10 < Q \leq R-2$	$Q-\frac{R}{2}+5$	—	—	—	—	—	—	$Q-\frac{R}{2}+5$
$R-2 < Q \leq \frac{4}{3}(R-2)$	$Q-\frac{R}{2}+5$	$Q-R+2$	$Q-R+2$	—	—	—	—	$3Q-\frac{5R}{2}+9$
$\frac{4}{3}(R-2) < Q \leq 2(R-2)$	$Q-\frac{R}{2}+5$	$\frac{Q}{2}-\frac{R}{3}$	$Q-R+2$	—	—	—	—	$\frac{5Q}{2}-\frac{11R}{6}+7$
$2(R-2) < Q \leq \frac{8}{3}(R-2)$	$Q-\frac{R}{2}+5$	$\frac{Q}{2}-\frac{R}{3}$	$\frac{Q}{2}$	$2(\frac{Q}{2}-R+2)$	$2(\frac{Q}{2}-R+2)$	—	—	$4Q-\frac{29R}{6}+13$
$\frac{8}{3}(R-2) < Q \leq 4(R-2)$	$Q-\frac{R}{2}+5$	$\frac{Q}{2}-\frac{R}{3}$	$\frac{Q}{2}$	$2(\frac{Q}{4}-\frac{R}{3})$	$2(\frac{Q}{2}-R+2)$	—	—	$\frac{7Q}{2}-\frac{21R}{6}+9$
$4(R-2) < Q \leq \frac{16}{3}(R-2)$	$Q-\frac{R}{2}+5$	$\frac{Q}{2}-\frac{R}{3}$	$\frac{Q}{2}$	$2(\frac{Q}{4}-\frac{R}{3})$	$2(\frac{Q}{4})$	$4(\frac{Q}{4}-R+2)$	$4(\frac{Q}{4}-R+2)$	$5Q-\frac{57R}{6}+21$
$\frac{16}{3}(R-2) < Q \leq 8(R-2)$	$Q-\frac{R}{2}+5$	$\frac{Q}{2}-\frac{R}{3}$	$\frac{Q}{2}$	$2(\frac{Q}{4}-\frac{R}{3})$	$2(\frac{Q}{4})$	$4(\frac{Q}{8}-\frac{R}{3})$	$4(\frac{Q}{4}-R+2)$	$\frac{9Q}{2}-\frac{41R}{6}+13$

TABLEAU 3.1 NOMBRE D'ÉCHANGES DE PAGES DANS LE TRI INTERNE

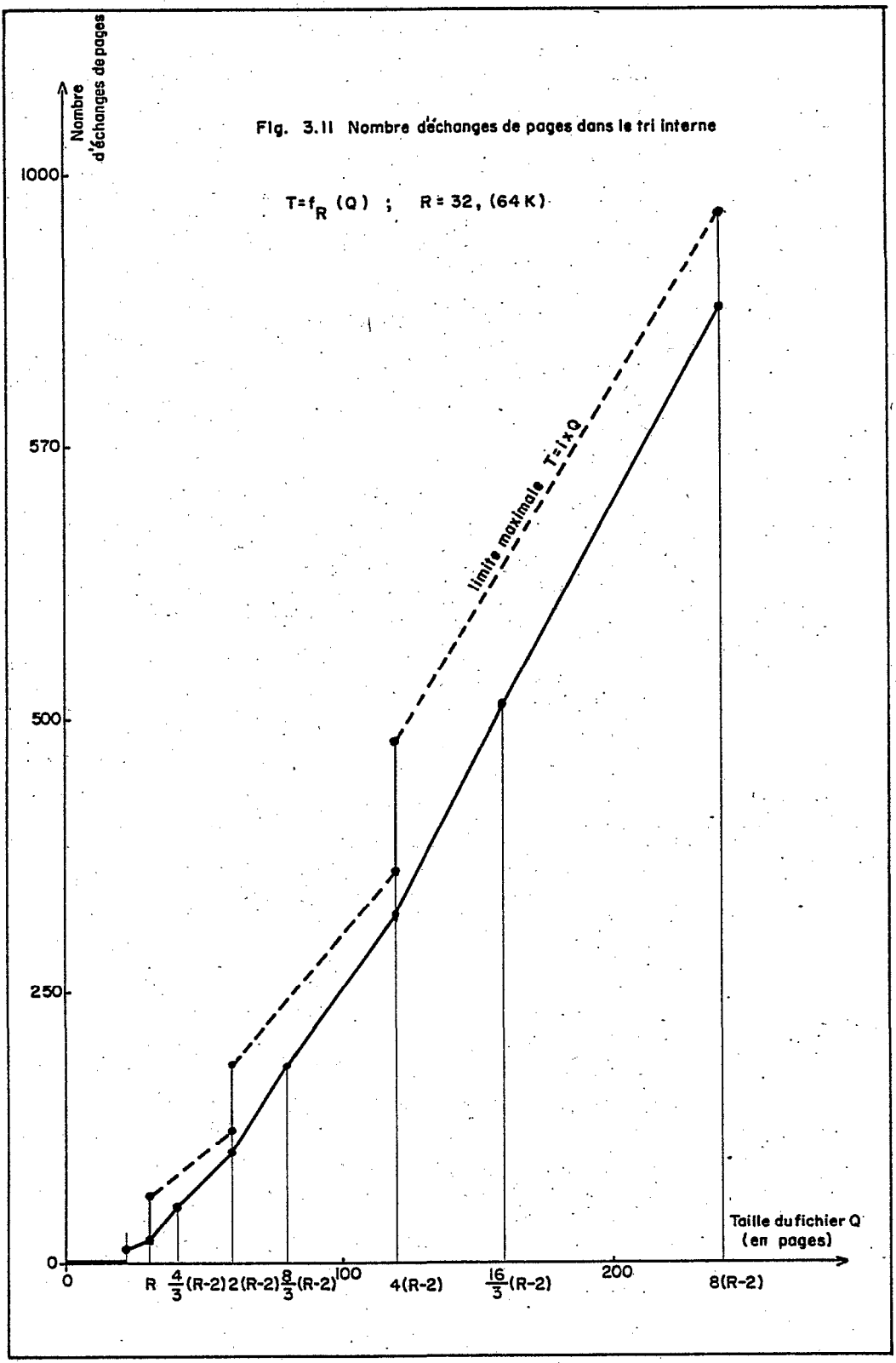


Fig. 3.11 Nombre d'échanges de pages dans le tri interne

$T = f_R(Q)$; $R = 32$, (64 K)

qu'un échange de pages dure environ 110 ms (voir §3.12), le temps total passé, seulement pour les échanges, serait de 32 secondes. Dans ces conditions, on pressent qu'il pourra être plus économique de segmenter le fichier Q en sous-fichiers plus petits, qui seraient triés séparément, stockés provisoirement sur disque et finalement fusionnés.

3.25 Définition de l'algorithme de formation des "runs"

Aucun des algorithmes présentés pour la formation des "runs" (voir §2.42) n'a été sélectionné au cours de l'étude comparative du paragraphe 2.45. Le calcul des échanges de pages au cours du tri interne, réalisé antérieurement, constitue un exemple particulièrement clair de l'influence du contexte de l'implantation dans le choix du propre algorithme utilisé dans le programme.

La philosophie de formation des "runs" consiste à utiliser tout l'espace de mémoire principale disponible pour créer des sous-fichiers qui devront être fusionnés ultérieurement. En principe, plus le nombre des "runs" formés est petit, plus le nombre de pas de fusion à exécuter après sera faible. Par conséquent, on cherchera à augmenter la taille des sous-listes qui résultent de la division d'un fichier déterminé.

Les méthodes de formation de "runs" qui ont été étudiées (voir §2.42), sont au nombre de trois:

- la simple utilisation de l'espace P de mémoire principale disponible, à partir d'une méthode de tri interne, créant ainsi des "runs" de taille fixe, égale à P;

- la méthode de "Replacement Selection", utilisant une structure d'arbre, qui permet de créer des "runs" de dimension variable, égale en moyenne à 2P;

- la méthode de sélection naturelle, semblable à la méthode antérieure, qui utilise de surcroît une zone d'extension capable de stocker le même nombre d'enregistrements que celui contenu dans la mémoire principale, et qui permet de créer des "runs" de dimension variable, égale en moyenne à 2,7P.

Certainement, en première approximation, le meilleur modèle sera celui qui provoque le moins d'échanges de pages dans la création de "runs" de même taille fixée Q. Pour que cette affirmation soit totalement vraie, il est nécessaire de supposer que les temps passés dans les autres opérations qui interviennent au cours de cette étape sont négligeables; en particulier, les opérations de 'I/O' entre la mémoire centrale et la zone d'extension.

La distribution de l'espace de mémoire disponible dans l'implantation d'un programme de formation de "runs", basé sur l'algorithme de "Replacement selection" (RS) ou de sélection naturelle (SN), peut être fait dans la configuration prise en exemple (voir §§3.11, 3.12 & 3.13). En réservant 28K pour la zone allouée au programme, qui dans ce cas doit maintenir en permanence les "buffers"

d'entrée et de sortie, on dispose de 214K pour stocker les données constituées des entrées (clé et pointeur vers l'enregistrement) stockées dans la séquence de lecture, et de 'l'arbre de perdants' (voir figure 2.8), utilisé par la méthode.

Pour dimensionner la zone qui sera réservée à l'arbre, on doit connaître le nombre de clés qui résideront simultanément dans la mémoire durant la création d'un "run" de taille Q = 222K (par exemple). Ceci, évidemment dépend de la propre dimension de la clé de tri: on peut en obtenir une valeur maximale, avec la plus petite entrée possible de 6 bytes (clé d'un byte, associée à un pointeur de 5 bytes). En rappelant que 'l'arbre de perdants' contient, pour chaque noeud interne, le numéro du "run" (RN) et le "loser", qui peuvent être contenus dans un espace minimal de trois bytes, le dimensionnement de l'arbre sera donné par l'équation suivante:

$$\text{dim(arbre)} = 3 * N_{max} = 3Q / (m * (3 + 5 + 1)) \quad ,$$

où m = 2 pour RS et 2,7 pour SN.

Avec la taille choisie Q = 222K, la dimension de l'arbre atteint 37K pour RS et 28K pour SN. En observant que l'arbre devra être maintenu en permanence dans la mémoire réelle, de la même forme que la totalité du programme, la somme de ce terme avec les 28K de la zone programme (soit respectivement 65K et 56K), est très voisine de l'espace de mémoire réelle alloué, R = 64K.

Dans une hypothèse, au demeurant très favorable, d'une clé de sept bytes, par exemple, l'espace mémoire occupé par celles-ci serait donné par l'expression:

12Q / 9m , soit 148K pour RS et 110K pour SN, ce qui termine la définition de la distribution d'espace adressable dans ces opérations.

Une fois établi l'arbre, et une fois placées séquentiellement toutes les premières clés dans la mémoire, opération qui ne devrait pas faire intervenir plus d'échanges de pages que celle de chargement des données dans le tri interne (voir §3.22), la méthode fait appel au transfert, une par une, de toutes les clés à destination de la zone du "buffer" de sortie (qui est présent en permanence dans le "Page Pool"). Etant donné que les clés sont encontrées dans l'ordre défini par le fichier d'entrée, l'appel provoqué par le transfert de chacune d'elles est totalement aléatoire dans cette dernière zone.

Pour la méthode RS, une fois fixées par artifices de programmation (voir §3.21) les pages réservées tant pour le programme que pour l'arbre, la sortie de chaque clé nécessitera un échange de pages, c'est à dire pour la totalité d'un "run", plus de 12 000, en plus de celles qui interviennent lors du déchargement des Q / 2 dernières entrées.

Pour la méthode SN, on dispose de seulement 4 pages réelles pour couvrir une aire de 54 pages virtuelles: aussi, la probabilité que la sortie d'une clé provoque un échange de pages est de plus de 90%, ce qui conduit à plus de 8 000 échanges de pages pour la totalité du

"run". De plus, quoique cette méthode soit apparemment plus économique que la précédente (puisqu'elle crée des "runs" sensiblement plus longs), on observe ici que la nécessité d'allouer en permanence un "buffer" supplémentaire pour la zone d'extension (ce qui doit diminuer encore la disponibilté des blocs de mémoire réelle) d'une part, les temps passés dans les opérations de 'I/O' pour la zone d'extension d'autre part, sont des facteurs qui peuvent certainement inverser ces conclusions.

Ainsi, l'importance des activités de pagination qui y interviennent, rendent ces deux dernières méthodes inadéquates à la formation des "runs", dans le contexte considéré. Toutefois, étant donné que la construction et la maintenance d'un arbre de sélection sont apparues comme les facteurs prédominants de la dégradation de performance, on peut penser à l'utilisation d'une méthode de sélection directe (voir §2.35) comme algorithme de détermination de la plus petite clé dans le programme de "Replacement Selection". En effet, une telle méthode respecte la règle de référenciation linéaire, fondamentale dans le contexte de mémoire virtuelle, et de plus économiserait l'espace occupé par l'arbre, en restituant ainsi les pages réelles correspondantes à la zone des clés. Cependant cette méthode apparaît rapidement pire encore, dans la mesure où la sortie de chaque clé nécessitera une lecture, même partielle, de cette dernière zone, créant ainsi plusieurs échanges de pages pour chaque clé.

En conclusion, la méthode plus simple de formation des "runs" par tri interne dans la partition, apparaît comme étant celle qui économise le plus d'échanges de pages pour créer un "run" de taille Q déterminée. De plus, un avantage secondaire est d'obtenir des "runs" de taille fixe, excepté le dernier, ce qui favorisera l'étape postérieure de fusion des "runs", examinée dans le prochain paragraphe.

3.26 Révision de l'algorithme de fusion des "runs"

La méthode choisie pour la fusion des "runs" (voir §2.46) est la plus convenable quand on peut utiliser des zones auxiliaires sur disque. La propre routine de fusion utilise un arbre de sélection dont le nombre de noeuds internes est égal au nombre de "runs" créés, ce dernier relativement petit. De plus, conformément à l'organisation choisie pour le programme dans la partie de tri externe (voir §3.22 & figure 3.7), la phase de fusion des "runs" ne coexiste jamais avec les données, et en conséquence on dispose de la totalité de la partition pour le programme, principalement pour les zones de "buffers" associés aux "runs" et aux fichiers de sortie.

Ainsi, l'objectif de l'étude présentée dans ce paragraphe est-elle de trouver le meilleur compromis entre la tendance à augmenter le nombre des "runs" qui seront fusionnés simultanément (ce qui diminuera le nombre de pas de fusions à effectuer), et la tendance à diminuer ce même nombre pour réduire le phénomène de pagination excessive provoqué par la présence de nombreux "buffers" dans la mémoire principale. Dans ce but, il devient nécessaire de mieux préciser la codification de l'algorithme de fusion (voir figure 3.12): ce programme utilise une

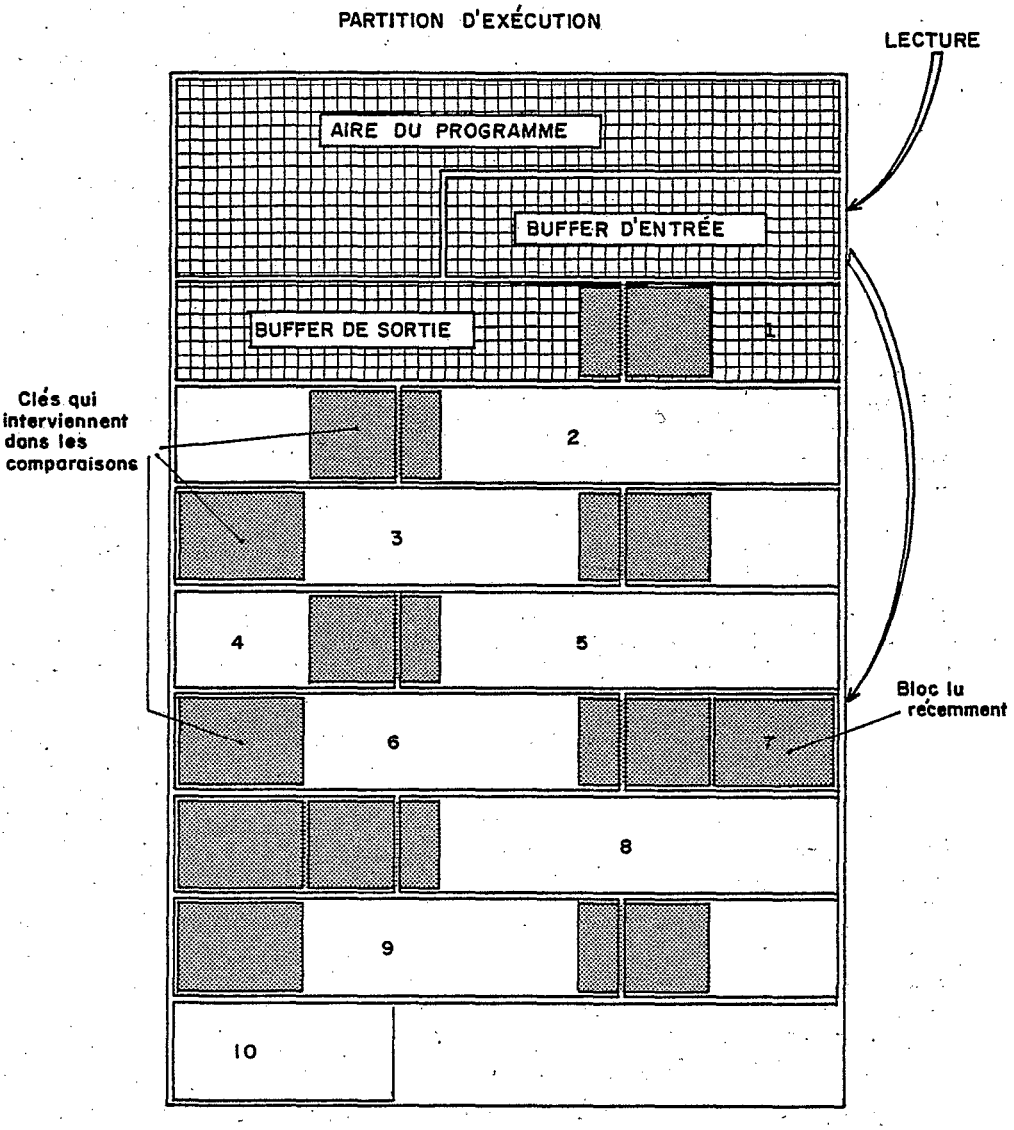


Fig. 3.12 ÉTAT DE LA MEMOIRE DURANT LA PHASE DE FUSION ('10-Way merge')

zone pour la lecture de tous les "runs" d'entrée (zone programme), les données lues sont ensuite transférées à une zone allouée pour ce "run" (zone données). Un "buffer" différent est utilisé, soit pour le "run" résultant si le pas est intermédiaire, soit pour le fichier de sortie dans le cas du dernier pas. En raison de la présence de ces deux zones, la partie programme occupe approximativement 20K et, évidemment, 222K restent disponibles pour stocker jusqu'à 31 zones de 7294 bytes (taille de bloc choisie pour les "runs" sur disque, et qui minimise les temps passés aux opérations d'entrée-sorties).

Lorsqu'on dispose de R pages réelles dans la partition pour réaliser un "N-way merge", les (R - 10) pages réelles disponibles pour la zone de données iront 'couvrir' totalement cette zone dans le cas où:

$$7n < 2 * (R - 10).$$

Naturellement, le bon fonctionnement de cette phase exige l'existence d'au moins une page réelle disponible pour chaque "run", vu que chaque sélection de clé dans l'arbre provoquera l'appel par lecture de la prochaine clé du "run" correspondant. Ainsi, la valeur n = R - 10 constitue la limite à ne pas dépasser sous peine de dégradation sensible de cette phase. Cependant, le calcul du nombre d'échanges de pages est intéressant, seulement dans le cas de la fusion de n "runs", avec n tel que:

$$2 * (R - 10) / 7 < n < (R-10) ,$$

c'est à dire pour 64K de mémoire réelle disponible (R = 32):

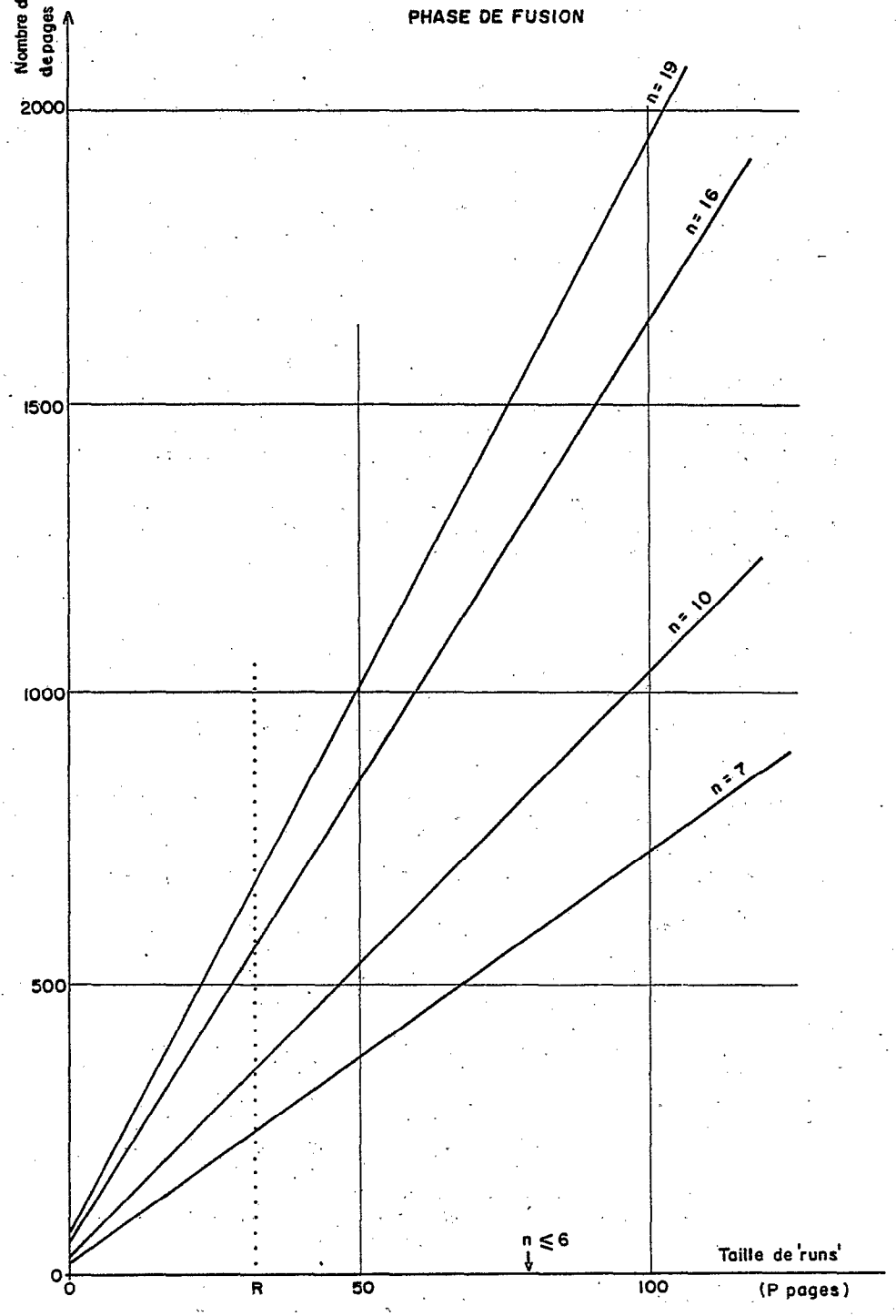
$6 < n < 22$. Dans ce cas on dispose de (R - 10) pages disponibles pour 'couvrir' une zone de Q = 7n / 2 pages virtuelles.

Au moment de la lecture des premiers blocs de chaque "run" (chargement de la zone des données), le nombre d'échanges de pages est, c'est certain, inférieur à Q. Immédiatement après, la première comparaison exige la présence des pages initiales de chaque "run", et ainsi le nombre d'échanges exigés pour cette opération est sûrement inférieur à n. Enfin, pour la fusion des n "runs", chacun de P pages, n * (P -1) échanges supplémentaires vont être exigés.

En considérant qu'au cours du processus, une situation telle que celle montrée sur la figure 3.12 se présente à la lecture de chaque bloc de données, on perçoit intuitivement que la présence dans la zone de données de trois pages additionnelles devra réduire le nombre d'échanges de pages occasionnés dans ces opérations, et alors la valeur de n = (R - 10) - 3, apparaît comme convenant bien dans cette phase, étant entendu que pour d'autres raisons (voir §2.43), on recherche la valeur maximale pour n (ici n=19, pour R=32).

Le graphique 3.13 présente le nombre maximal d'échanges de pages créés dans la phase de fusion, pour diverses valeurs de n; les résultats très élevés doivent être examinés avec attention. Ainsi la limite de 200 échanges de pages (temps passé de 22 secondes) est seulement atteinte par la fusion de 19 "runs" de 100 pages chacun, ce qui représente un fichier total de près de 200 000 enregistrements

GRÁFICO-3.13 NOMBRE MAXIMUM D'ECHANGES DE PAGES DANS LA PHASE DE FUSION



avec des clés de quinze positions. On peut noter également que, tant que le nombre n de "runs" qui doivent être fusionnés simultanément fait partie de l'intervalle précédemment défini, la quantité d'échanges de pages dépend seulement de la taille du fichier initial, c'est à dire nP.

3.27 Conclusions

Dans les paragraphes de ce dernier sous-chapitre, on a mis en évidence l'influence du phénomène de pagination, tant dans le propre choix de l'algorithme de tri, que dans le champ d'application de chacun d'eux. Bien entendu, tous les algorithmes qui font intervenir une référenciation non séquentielle (méthode utilisant des listes chaînées, arbres, etc...), doivent être évitées dans le contexte de mémoire virtuelle. De la même façon, si l'on utilise la méthode de Shell, l'incrément h devra être maintenu suffisamment petit pour limiter le nombre de pas qui doivent être réalisés sur la totalité de la liste (et ainsi la pagination).

La spécification des dispositifs périphériques employés dans la réalisation a permis de quantifier la pénalisation par laquelle se traduit l'utilisation de la mémoire virtuelle. Pour arriver à la détermination, pour chaque situation, d'une méthode optimale, il est nécessaire de quantifier de la même manière les temps passés dans les autres opérations, et en particulier les opérations de transfert entre la mémoire externe et la mémoire interne.

En effet, selon ce qui a été vu au paragraphe précédent, c'est à dire que la fusion d'au plus six "runs" peut se faire sans provoquer d'échanges de pages, on conçoit que pour une taille de fichier initiale déterminée, la division en sous-listes plus petites qui pourront être triées selon une méthodologie de tri interne, même avec un peu d'échanges de pages, pourra devenir finalement plus rapide.

Dans la mesure où la structure générale du programme a été projetée pour éviter les pertes de temps provoquées par de trop nombreux chargements de phases, les opérations qui restent les plus dispendieuses seront celles d'écritures et de lectures des "runs" sur disque.

L'objectif principal du prochain sous-chapitre est d'étudier les stratégies possibles de stockage et de récupération de l'information sur disque, de façon à optimiser les étapes de distribution et fusion des "runs" et d'évaluer le temps passé à ces opérations.

3.3 STRATEGIE DE DISTRIBUTION DES "RUNS"

Pour évaluer l'influence relative de l'utilisation de la mémoire virtuelle dans le processus de tri, il est indispensable de mesurer les temps passés au cours des autres opérations et, en particulier, les opérations de 'I/O', les plus dispendieuses.

L'objectif du présent sous-chapitre est d'examiner les diverses stratégies possibles pour le stockage des "runs" en zones de mémoire

auxiliaire sur disque. Compte tenu de ce que la majeure partie du temps gâché dans les opérations de transfert entre ces unités périphériques et la mémoire centrale est dûe aux mouvements du bras d'accès, la recherche d'un schéma optimisé d'allocation des "runs" est pleinement justifiée. Cette stratégie dépend tant des paramètres inhérents aux propres données qui doivent être stockées (organisation initiale du fichier, par exemple), que de la division des aires de travail réservées. De plus, il faut rechercher à minimiser les temps de "seek", tant dans la phase de distribution des "runs", que dans celle de leur fusion.

Dans un premier paragraphe, les stratégies de stockage, examinées par rapport au nombre d'aires de travail disponibles, sont passées en revue; ensuite, les conditions dans lesquelles ces stratégies sont optimales sont étudiées, moyennant des hypothèses restrictives concernant l'organisation initiale du fichier d'entrée. L'avant dernier paragraphe est consacré aux évaluations de temps passé dans les opérations de fusion, avec les hypothèses retenues pour l'allocation des "runs": les conclusions qui sont tirées font l'objet du paragraphe 3.34.

3.31 Les types d'allocation

A la suite des résultats obtenus dans la recherche de l'algorithme optimal de fusion des "runs" (voir §3.26), il s'avère intéressant de fusionner de nombreux "runs" en une seule fois (jusqu'à 19, pour R = 32 pages, par exemple). Dans la mesure où il n'est pas possible d'utiliser un nombre aussi élevé d'aires de disques séparés, il y aura inévitablement des mouvements du bras d'accès ("seek"), pour que soient réalisées des lectures en des "runs" différents, mais écrits sur la même unité de disque.

La recherche d'une systématique optimale pour le stockage de l'information sur les disques n'a pas une importance prépondérante dans la phase initiale de distribution des "runs". En effet, on peut rappeler que l'organisation générale du programme dans la phase de tri externe (voir §3.22 & figure 3.7) impose plusieurs étapes, chacune constituée de la lecture d'une partie du fichier initial d'entrée, du tri interne de ces données préalablement chargées dans la mémoire principale et de l'écriture finale de ce "run" sur une aire de mémoire auxiliaire. Aussi, durant la phase de tri proprement dit, des artifices de programmation permettent de changer la position du bras d'accès, jusqu'à le positionner dans la situation désirée pour l'écriture du "run" résultant, ce qui éliminera la plus grande partie des "seeks" requis lors de cette phase (dans la mesure où, bien sûr, les unités de disque sur lesquelles sont stockés les "runs" sont exclusivement réservées pour l'exécution du programmme de tri).

Le premier facteur de minimisation des temps passés est, sans aucun doute, le blocage maximal des données contenues dans les "runs", c'est à dire des entrées (clés associés aux pointeurs) triées. En effet, en utilisant des blocs dont la taille est celle d'une piste de disque (ici, 7294 bytes), le temps de rotation (voir §3.11) est minimisé: en contrepartie, on doit supporter l'inconvénient inhérent au dimensionnement plus important des "buffers" correspondants.

Si l'on considère alors seulement la phase de fusion, le schéma d'allocation dépendra en premier lieu du nombre des aires de travail sur disques séparés disponibles. Ainsi, lorsque l'on dispose de deux disques, il est clairement raisonnable d'utiliser le premier pour la lecture et le second pour l'écriture, étant donné que (globalement) à chaque bloc lu correspond un bloc gravé. Dans le cas où l'on dispose de trois, ou plus, disques séparés, une allocation judicieuse des "runs" qui doivent être fusionnés pourra réduire encore le temps total passé dans les "seeks", en éliminant également des mouvements du bras d'accès nécessités par les lectures.

Le cas où l'on ne dispose que d'un seul disque comme zone de travail conduit à une recherche plus poussée de la stratégie de stockage. La première idée qui mérite d'être examinée est d'utiliser l'allocation dite orthogonale. La figure 3.14 présente les deux types d'allocation qui peuvent être adoptés. Dans l'allocation séquentielle le "run", constitué de plusieurs blocs, est stocké sur des pistes consécutives de telle sorte que, au cours de la distribution, un seul "seek" est nécessaire pour chaque ensemble de 20 pistes écrites, et le "run" (i + 1) débute à la première piste laissée libre par le "run" i. Dans l'allocation orthogonale, les "runs" sont stockés de telle manière que le "run" 1 soit stocké sur les pistes 0 des premiers cylindres; le "run" 2 sur les pistes 1 des premiers cylindres, et ainsi de suite. Avec ce schéma, l'écriture d'un "run" i quelconque nécessite un "seek" (minimal, car d'un seul cylindre), pour chaque bloc (piste), mais en compensation la phase de fusion devra tirer avantage de ce type de distribution, puisque celle-ci sera réalisée en partie avec plusieurs pistes du même cylindre (c'est à dire sans "seek").

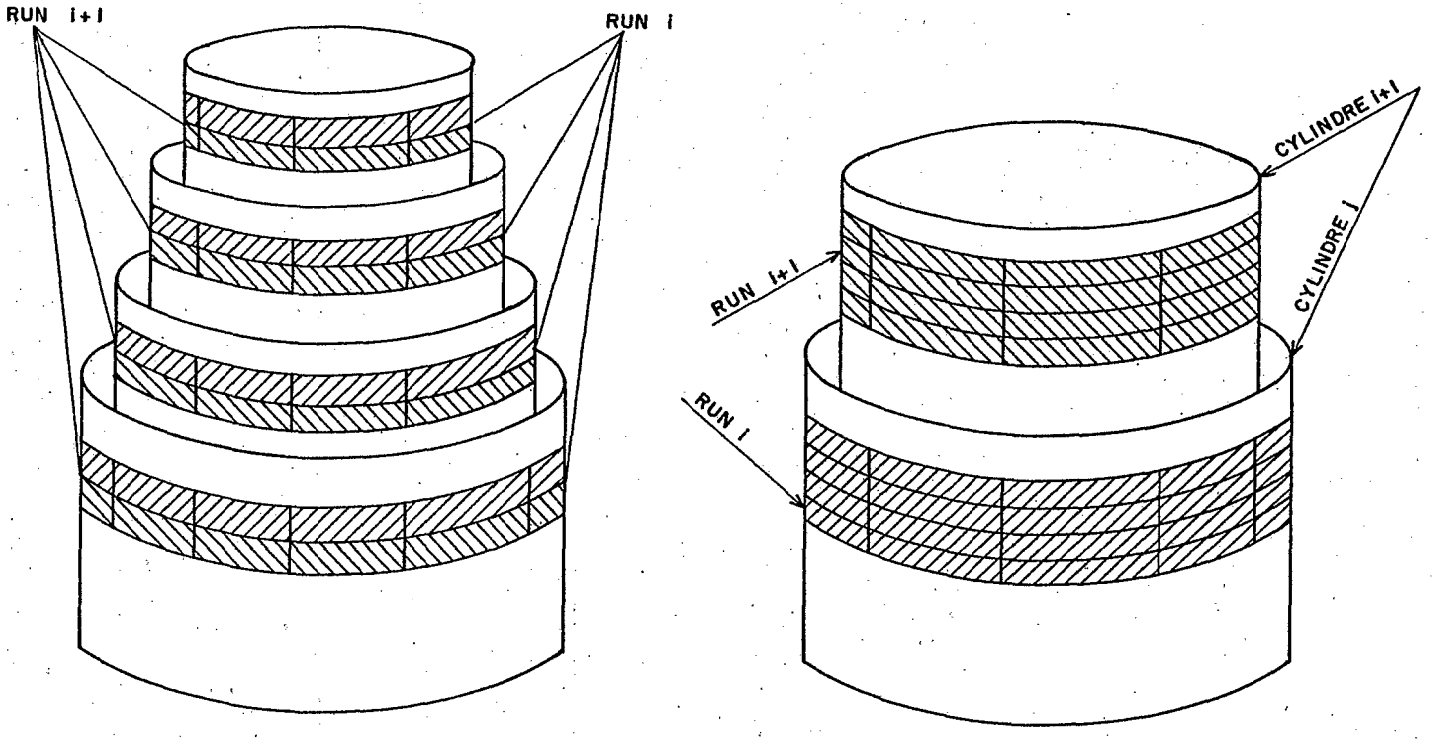
3.32 Les conditions d'optimisation

Pour analyser les conditions dans lesquelles ces deux types d'allocation s'avèrent les plus adéquates, on peut examiner les deux situations extrêmes suivantes:

- le fichier est totalement ordonné par rapport à la clé pour laquelle on prétend faire le tri;

- le fichier est complètement désordonné par rapport à cette même clé, de telle sorte que la fonction de répartition de cette clé est du type uniformément répartie (la probabilité d'apparition d'une clé C, dont la valeur K appartient à l'intervalle (k,k + dk), est directement proportionnelle à dk).

Dans la première situation, plutôt fréquente en traitement de l'information, seront créés des "runs" successifs tels que la dernière clé du "run" i est plus petite que la première clé du "run" j (i < j). Aussi la phase de fusion dégénère en une succession de copies des divers "runs" d'entrée dans le "run" de sortie de telle sorte que, après la lecture de la première piste de chaque "run", tous les autres blocs du "run" 1 seront lus avant tous ceux du "run" 2, etc.... Bien entendu, dans cette situation, une allocation séquentielle des "runs" apparaît comme optimale, vu que seule sera nécessaire un nombre de



a) allocation orthogonale

b) allocation séquentielle

Fig. 3.14 Distribution de 'runs'

"seeks" requis pour effectuer la lecture totale de la partie occupée de la zone de travail.

Dans la seconde situation, des "runs" sont créés dont les clés auront des valeurs uniformément réparties sur l'ensemble du domaine de définition. Dans ces conditions, dans n * j pas élémentaires de fusion de n "runs" de N clés chacun, un nombre voisin de j clés de chaque "run" aura été sélectionné. En d'autres termes, le transfert des blocs de chaque "run" d'entrée vers le "run" de sortie est réalisé parallèlement. Naturellement, dans une telle situation, une allocation orthogonale est hautement recommandée, vu que les lectures des pistes d'un même cylindre seront effectuées consécutivement, et par conséquent le nombre de "seeks" requis sera seulement celui nécessaire à la lecture complète de la partie occupée de la zone de travail.

Bien entendu, les situations réelles ne seront jamais strictement conformes à l'un de ces deux cas: toutefois, le degré d'ordre initial du fichier est généralement connu par l'utilisateur comme étant faible ou élevé, de sorte que le passage de cette information au système peut orienter le choix, en temps d'exécution, de la routine d'allocation la plus appropriée.

3.33 L'évaluation des temps passés dans la fusion des "runs"

De la même façon que l'allocation séquentielle des "runs" est optimale dans le cas d'un fichier d'entrée totalement ordonné, elle devient mauvaise dans le cas d'un fichier totalement aléatoire, vu que dans cette dernière situation on a les "seeks" les plus grands en permanence d'un "run" à l'autre. Pour ce qui est de l'allocation orthogonale, la meilleure dans ce dernier cas, les conclusions sont inversées, et l'on a les plus grands mouvements du bras dans la fusion des "runs" qui proviennent d'un fichier totalement ordonné.

Dans l'impossibilité d'évaluer le temps passé dans la phase de fusion des "runs" générés à partir d'un fichier quelconque, on peut tout au moins obtenir un intervalle dans lequel cette valeur est incluse, par estimation des temps passés dans les conditions extrêmes de fonctionnement de cette opération. Etant donné la valeur q (pages) de la taille commune des "runs", le nombre de pistes nécessaire pour stocker un "run" est:

$$m = 2q / 7 \text{ (par excès)}.$$

En rappelant les caractéristiques des disques qui servent d'exemple dans cette étude (voir §3.11 & figure 3.1), la lecture d'un bloc de 7 294 bytes, une fois positionné le bras sur le cylindre adéquat, nécessite un temps de révolution de 12,5 ms et un temps de transmission de 23,5 ms, soit au total 36ms.

En considérant seulement la phase de fusion des "runs", la première opération réalisée est la lecture du premier bloc de chacun des n "runs". Ensuite, les comparaisons vont requérir une succession de lectures des blocs de chaque "run", selon une séquence qui ne dépend que de l'organisation du fichier d'entrée. Les temps consommés

dans ces opérations varient en accord avec le type d'allocation choisie:

- dans l'allocation orthogonale, on a (36n + 60) ms pour lire les n blocs '1' de tous les "runs", à partir d'une position initiale quelconque du bras d'accès. Dans le cas idéal (fichier d'entrée totalement aléatoire), les n blocs requis à chaque fois appartiennent au même cylindre, et en conséquence, la fin de la phase va demander:

(36n + 25,5) * (m - 1) millisecondes ,

où 25,5 ms est le temps correspondant au "seek" d'un seul cylindre; finalement, le temps total consommé dans ce cas sera:

T_1 = 36mn + 25,5m + 34,5 millisecondes.

Dans le pire cas (fichier d'entrée totalement ordonné), on a (m - 1) * (25,5 + 36) ms pour lire chaque "run" et finalement le temps total consommé sera:

T_2 = 61,5mn + 36n + 60 - 61,5 = 61,5mn + 36n - 1,5 ms ;

- dans l'allocation séquentielle, à partir d'une position initiale quelconque du bras d'accès, on a:

(36n + 60 + (25 + m / 40)) * (m - 1) millisecondes

pour lire les blocs '1', où t = 25 + m / 40 ms est le temps de mouvement du bras entre deux cylindres séparés par m pistes (m / 20 cylindres). Dans le cas optimal (fichier d'entrée totalement ordonné), les opérations consécutives ne nécessiteront qu'au plus mn / 20 "seeks" élémentaires de 25,5 ms, en plus du temps nécessaire à la lecture des mn - 1 blocs restants. Aussi le temps total sera:

T_3 = 37mn + 25n millisecondes.

Dans le pire cas (fichier d'entrée complètement aléatoire), les lectures successives exigent un mouvement du bras entre deux blocs aléatoirement distribués dans un groupe de mn - n blocs, c'est à dire:

25 + (mn - n) / 120 ms. Compte tenu des valeurs attendues de m et n (les deux inférieures à 20), le dernier terme de cette somme sera négligeable et, en conséquence, le temps total consommé dans ce cas pourra être évalué à:

T_4 = 61mn + 35 millisecondes.

En assimilant la quantité m = 2q / 7n (par excès) à 2q / 7n, et en ne prenant en compte que les termes du second degré, le produit mn peut être substitué par 2q / 7, et les formules qui donnent T_1, T_2, T_3 et T_4 peuvent se résumer à:

$T_1 = T_3$ = 10,4 * Q millisecondes ;

$T_2 = T_4$ = 17,6 * Q millisecondes .

Etant donné la symétrie existant entre les conditions d'utilisation de chaque type d'allocation, et dans la mesure où le degré d'ordre du fichier initial est connu, même grossièrement, on peut supposer que le temps T nécessaire à la lecture des "runs" dans la phase de fusion ne devra pas dépasser la valeur 14Q ms, correspondant à:

$$\min((T_1 + T_2) / 2 , (T_3 + T_4) / 2).$$

Les valeurs relativement faibles de temps obtenues (26,6 secondes pour l'exemple du fichier de 200 000 enregistrements avec des clés de 15 positions et segmenté en 19 "runs" de 100 pages chacun, à comparer avec les 22 secondes obtenues pour les échanges de pages dans la même phase) ne doivent pas surprendre, étant entendu qu'elles ont été obtenues avec une valeur élevée du blocage (chaque piste contient 364 entrées), et en admettant que l'écriture du "run" de sortie ne consomme pas de temps (cas de l'utilisation d'un second disque pour une étape intermédiaire, ou cas du dernier pas).

3.34 Conclusions

L'examen des résultats obtenus dans le paragraphe précédent peut suggérer l'abandon de l'une des deux méthodes d'allocation, en raison du peu de différence relative entre le temps passé dans le pire et dans le meilleur des cas (approximativement 70%). Immédiatement, l'allocation séquentielle paraît la plus attrayante en raison de la simplicité avec laquelle la zone de travail est référencée, avec "l'EXTENT" fourni par l'utilisateur, c'est à dire les deux limites "Lower" et "Upper" de la zone. En effet, dans ce type d'allocation, le "run" 1 sera stocké sur les pistes:

Lower, Lower + 1, Lower + 2,..., Lower + (m - 1) ,

et juste après, le "run" 2 sera stocké à partir de la piste Lower + m. En même temps, on pourra testé, avant l'écriture sur la piste Lower + i, si i est supérieur à la différence Upper - Lower, auquel cas on vérifie que la zone réservée est insuffisante pour contenir tous les "runs".

Dans le cas de l'allocation orthogonale, le traitement est plus complexe en raison de la nécessité d'utiliser une zone "d'overflow" pour contenir les blocs i de chaque "run", qui ne tiennent pas sur la zone définie par la piste i - 1 de l'allocation et "l'EXTENT" de l'utilisateur (voir figure 3.15). Dans la solution proposée, on utilise un chaînage des blocs de chaque "run", avec une aire "d'overflow" dynamique, contenue dans la propre zone principale, et respectant la philosophie de l'allocation orthogonale.

Toutefois, pour certains cas de processus de fusion avec plusieurs pas, l'allocation orthogonale devient intuitivement bien meilleure que l'allocation séquentielle, dans la mesure où les "runs" de sortie peuvent être stockés sur les dernières pistes des mêmes cylindres que les "runs" d'entrée. Pour donner un exemple, supposons que 16 "runs" ont été distribués sur les pistes 0 à 15 de m cylindres: on pourra alors utiliser le "6-way merge" (qui ne fait intervenir aucun échange

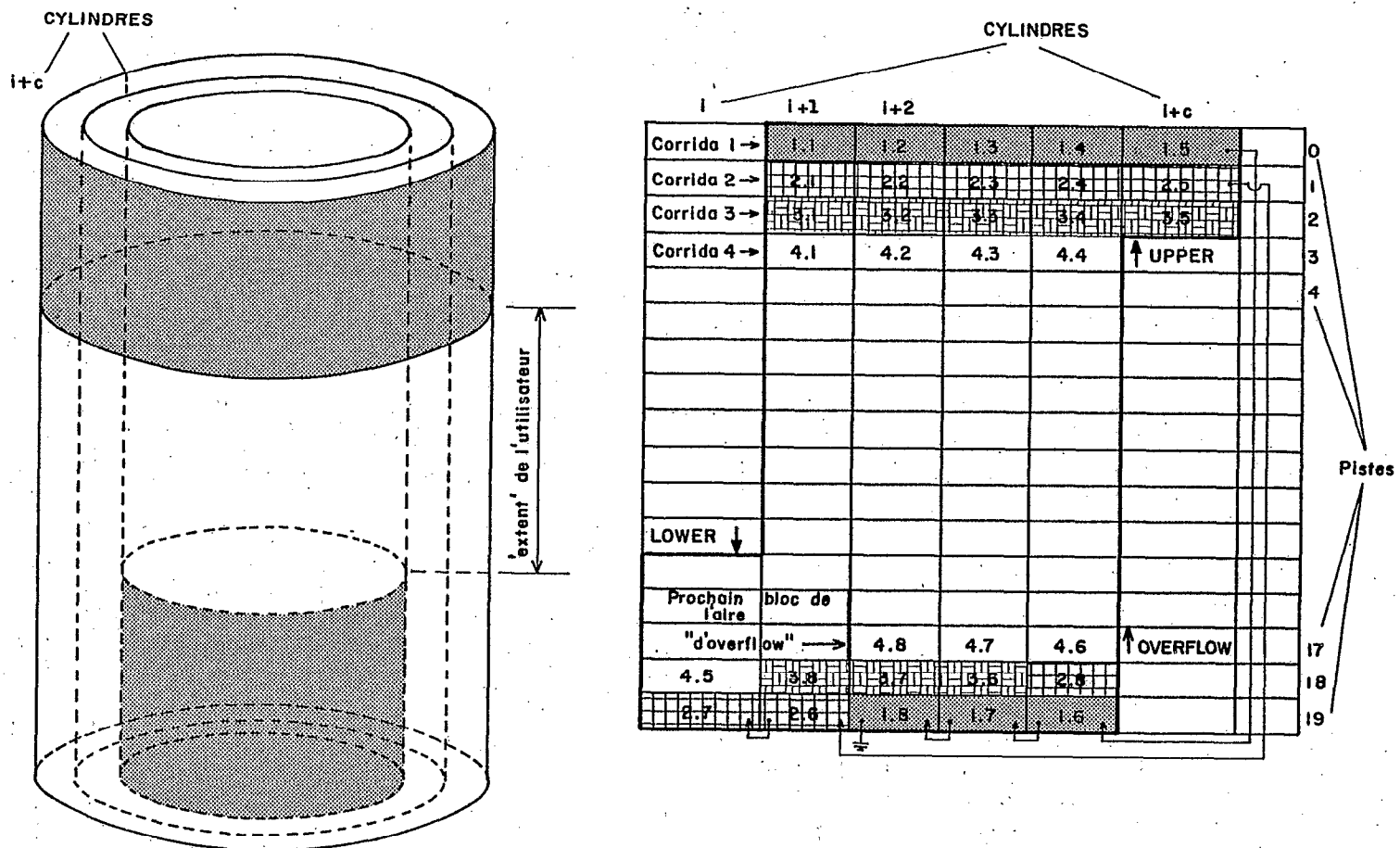


Fig. 3.15 PROCESSUS D'ALLOCATION ORTHOGONALE DE 'RUNS' AVEC CHAÎNAGE ET AIRE "D'OVERFLOW" DYNAMIQUE

de pages), en écrivant le "run" 17, résultat de la fusion des "runs" 1 à 6, sur les pistes 16 de 4m cylindres. De la même façon, le "run" 18, résultat de la fusion des "runs" 7 à 12, sera écrit sur les pistes 16 des mêmes cylindres. Finalement, en associant les pistes 13 à 16 avec ces deux dernières, l'étape finale génère un nombre minimal de "seeks", alors que l'allocation séquentielle aurait engendré beaucoup plus de "seeks" à chaque pas, pour déplacer le bras entre les "runs" d'entrée et de sortie.

3.4 INTERVALLES DE PLUS GRANDE EFFICACITE

Au cours de ce chapitre on cherche à évaluer les temps passés lors des opérations les plus dispendieuses, celles qui font intervenir des transferts d'information entre la mémoire principale et les unités périphériques. Les opérations d'échanges de pages, inhérentes à l'utilisation de la mémoire virtuelle, se sont avérées comme étant celles qui jouent le rôle le plus important dans les pertes de temps, si bien que l'on peut se demander jusqu'à quel point le tri interne d'un fichier de taille Q donnée, est réalisé plus rapidement que le tri externe de ce même fichier, divisé en deux parties égales, par exemple.

Pour répondre à cette question, il est indispensable d'examiner tous les autres facteurs de consommation de temps: c'est l'objectif de premier paragraphe de ce sous-chapitre.

En poussant plus loin ce raisonnement, il est intéressant d'examiner s'il existe un intervalle des valeurs de Q, indépendamment de n'importe quel autre paramètre (du moins en première approximation), tel que le tri par division en n sous-listes, est simultanément plus rapide que celui par division en n + 1 sous-listes (parce que l'on économise le temps nécessaire au traitement d'une sous-liste), et que celui par division en n - 1 sous-listes (parce que l'on économise des échanges de pages, tant au cours du tri que, éventuellement, dans la phase de fusion).

Bien sûr, en continuant encore cette étude, on peut comparer l'utilisation de plusieurs pas de tri au lieu d'un pas unique, de façon que soit réduit le nombre d'échanges de pages qui interviennent dans la phase de fusion, en réduisant le nombre de "runs" simultanément intercalés.

Les paragraphes ci-après fournissent des arguments pour répondre à ces questions.

3.41 Les divers facteurs de consommation de temps

L'évaluation des temps passés dans les opération de calculs sur ordinateur est toujours un travail délicat, en raison des nombreux paramètres qui peuvent influencer le déroulement d'une étape déterminée. On peut citer l'utilisation de la mémoire virtuelle, la ressource de multiprogrammation, l'intervention des programmes du système comme le "Job Control" et le "Spooler", le phénomène d'exécution simultanée de certaines opérations de 'I/O' avec les

calculs ("Overlap") et les interruptions provoquées par la réquisition du processeur, comme exemples d'interférences les plus importantes qui perturbent les mesures de temps passé à la réalisation d'une tâche déterminée. Aussi, n'atteint-on généralement qu'une approximation, plus ou moins rigoureuse, de la valeur désirée.

Dans le cas particulier étudié, on a cherché à déterminer les temps correspondants, en éliminant, quand c'est possible, les temps considérés comme négligeables. Le fait que les évaluations soient relatives, parce que menées à des fins de comparaison, permet d'éviter le calcul de certaines valeurs particulièrement difficiles à estimer, et qui, au moment du bilan, seraient de toutes façons exclues, puisque présente dans les deux membres de l'inéquation.

En considérant la totalité du temps consommé par l'exécution du programme de tri, pour un fichier d'entrée de taille Q et constitué de N clés de longueur T, avec:

$$Q = N * (T + 5) \quad ,$$

on doit additionner aux temps précédemment définis, les termes suivants:

- le temps passé au chargement de la phase 'ROOT' (voir figure 3.7) et de la phase de critique, l'exécution de cette dernière phase et la détermination du type de tri choisi, et finalement le temps de déchargement de la phase de tri externe ou interne (on supposera cette dernière unique, pour convenance de calcul). Cette valeur n'a pas besoin d'être déterminée, dans la mesure où ce travail devra être réalisé, quel que soit le type de tri qui devra être effectué;

- le temps nécessaire à la lecture séquentielle et au chargement dans la mémoire de la liste de taille q. Pour toutes les divisions réalisées, la totalité du fichier devra être lue et, si l'on suppose que l'augmentation de ce temps due à la lecture de Q en n parties de dimension q = Q / n est négligeable, ce terme se réduit à la durée de pagination provoquée par le chargement de q pages dans la zone de données, c'est à dire:

q - (R - 14) échanges de pages. En effet, on observe que des R pages réelles disponibles, 14 sont fixées initialement par le programme (voir §3.25), et en conséquence R - 14 sont allouées à l'aire des données;

- le temps réellement passé aux opérations internes à la mémoire dans l'étape de tri proprement dite des:

$$N' = N * q / Q \quad \text{clés.}$$

Cette valeur est très difficile à estimer, du fait qu'elle dépend du degré d'ordre initial du fichier, vu que les pas élémentaires de tri par insertion sont plus efficace dans le cas d'un fichier déjà presque ordonné. En utilisant le nombre moyen d'instructions exécutés par seconde (dans l'exemple choisi du modèle /370-135: 0,185 Mips, voir §3.11), le temps total T passé au tri de N' clés est proportionnel à N'logN', et au nombre d'instructions

contenues dans la boucle la plus interne (en principe, la routine de comparaison de deux clés). Si l'on se contente d'examiner seulement la valeur relative du temps T_1 passé dans le tri du fichier Q entier (T_1 = AQlogQ), par rapport au temps T_n passé au tri des n sous-fichiers de taille q = Q / n :

$T_n = n * A * (Q / n)$, on a:

$T_1 - T_n = AQlogQ - AQ * (logQ - logn)$, ou encore:

$T_1 - T_n = AQlogn = T_1 * (logn / logQ)$.

Dans la mesure où n est petit (dans le cas du paragraphe 3.42, n = 2), et Q est grand (supérieur à R, au moins), le terme x:

x = logn / logQ

est, certainement, inférieur à 20%, ce qui rend la différence $T_1 - T_n$ négligeable, et par là dispense de prendre en compte la durée des opérations internes dans les comparaisons;

- le temps nécessaire au chargement de la phase de fusion, seulement dans le cas du tri externe. Il a déjà été vu (voir §3.13) que le temps de chargement dans la mémoire de cette phase (24K), localisée dans la 'CIL', est de 2 secondes;

- enfin, le temps nécessaire à l'écriture du fichier de sortie, quand les données qui doivent être stockées sont prêtes dans la mémoire principale, soit à la fin du tri interne, soit au cours de l'opération de fusion (tri externe, pas final). Hormi les échanges de pages nécessaire à la lecture de toute la liste de données présentes dans la mémoire au moment du déchargement, c'est à dire q - (R - 14), cette valeur est évidemment la même, quelle que soit la division du fichier initial.

3.42 Limite d'efficacité du tri interne

On appelle limite d'efficacité du tri interne la valeur 1 du rapport Q / R, telle que pour Q > 1 * R , il devient plus lent de trier le fichier en une seule fois (en raison du nombre plus élevé d'échanges de pages), que de trier succcessivement les deux moitiés du fichier, pour les intercaler ensuite. Evidemment, lorqu'il est plus lent de trier le fichier en deux parties plutôt qu'en une seule fois, il sera encore plus lent de trier le fichier en trois parties, ou plus. Par ailleurs, la division en deux parties égales est celle qui gâchera le moins de temps, en raison, entre autres choses, du caractère non linéaire du nombre d'échanges de pages (ou du temps consommé) de la phase de tri proprement dite, par rapport à la taille du fichier.

Aussi la limite 1, si elle existe, est la valeur de Q / R telle que T_1 (qui représente le temps total passé au tri d'un fichier de taille Q dans une partition de taille P, lorsque l'on ne dispose que de R pages réelles), soit exactement égal à T_2, qui représente le temps total passé, dans les mêmes conditions, pour le tri des deux

sous-fichiers obtenus, par équidivision du fichier initial. L'équation correspondante fait intervenir les deux sommes suivantes:

$$T_1 = T_{initial} + T_{lecture} + T_{1chargement} + T_{1tri} + T_{1déchargement} + T_{final};$$

$$T_2 = T_{initial} + T_{lecture} + 2 * (T_{2chargement} + T_{2tri} + T_{2déchargement} + T_{2fusion} + T_{2final}).$$

En utilisant les résultats du paragraphe précédent pour simplifier l'équation $T_1 = T_2$, on aboutit à:

$$T_{1chargement} + T_{1tri} + T_{1déchargement} = T_{2fusion} + 2 * (T_{2chargement} + T_{2tri} + T_{2déchargement});$$

où tous les termes, excepté $T_{2fusion}$, sont proportionnels aux nombres d'échanges de pages qui interviennent dans les opérations. On peut observer par ailleurs que ces opérations de pagination, lors du chargement et du déchargement des données, sont incomplètes, dans le sens défini au paragraphe 3.12, et ainsi ne requièrent que 65 millisecondes, alors qu'il en faudrait 110 pour un échange complet (cas du tri proprement dit). Dans ces conditions, on a:

$$T_{ichargement} + T_{idéchargement} = 2 * (Q / i - (R - 14)) * 65$$

millisecondes, alors que:

$$T_{itri} = 110 * E(tri) \text{ millisecondes},$$

où E(tri) est le nombre d'échanges de pages de la phase de tri. En rappelant que le temps $T_{2fusion}$ se compose du temps passé au chargement de la phase correspondante (c'est à dire 2 secondes, voir §3.13), et du temps passé aux opérations nécessaires à la fusion des "runs" (c'est à dire 14Q ms, voir §3.3), on a finalement:

$$T_2 - T_1 = T_{2fusion} - \text{Dif.}(T(tri)) - \text{Dif.}(T(charg./déchar.)).$$

On perçoit rapidement que la valeur 1 de Q / R doit être très voisine de 1, vu que le temps passé aux échanges de pages dans les seules phases de chargement et déchargement des données (dans le cas où Q / i est supérieur à R - 14), compense rapidement le temps perdu au cours de la phase de fusion.

Pour: $Q < R - 14$, il n'y a pas d'échanges de pages dans aucun des deux cas, et par conséquent:

$$T_2 - T_1 = T_{2fusion} > 0.$$

Aussi, initialement, c'est à dire pour les fichiers très petits, la division du fichier d'entrée n'est pas recommandée, ce qui se savait à priori.

Pour: R - 14 < Q < R - 10 (et en supposant que R est supérieur à 18, de telle sorte que pour cet intervalle de Q, on ait Q / 2 < R - 14), on a seulement Q - (R - 14) échanges de pages dans les phases de chargement et déchargement du fichier unique Q, si bien que la quantité:

$$T_2 - T_1 = 14Q + 2\,000 - 130 * (Q - (R - 14))$$

se maintient positive, quelle que soit la valeur de R, tant que Q appartient à l'intervalle défini.

Pour: R - 10 < Q < R - 2 (et en supposant que R est supérieur à 26, de telle sorte que dans cet intervalle de Q on ait Q / 2 > R - 14), le tri du fichier unique commence à faire intervenir des échanges de pages (voir tableau 3.1), et ainsi l'on a:

$$T_2 - T_1 = 14Q + 2\,000 - 130Q - 1\,820 - 110 * (Q - R / 2 + 5)$$

soit encore:

$$T_2 - T_1 = 185R - 226Q - 370.$$

Le graphique 3.16 présente, dans le plan cartésien (Q,R), l'évolution de la limite d'efficacité du tri interne, représentée par le segment de droite d'équation $T_2 - T_1 = 0$, et délimité par les droites d'équations respectives R = 26, Q = R - 10 et Q = R - 2. On constate alors que la limite 1 existe pour toute valeur de R appartenant à l'intervalle (26,46) en pages, c'est à dire entre 52 et 92K.

Au delà de cette limite de 92K de mémoire réelle disponible, la limite d'efficacité est donnée par Q = R - 10. Pour les valeurs de R inférieures à 26 pages, on aurait une autre courbe représentative de la limite d'efficacité. Cependant, non seulement ces valeurs de R et Q ne présentent que peu d'intérêt dans les conditions de l'exploitation, mais en plus un certain nombre des hypothèses qui ont été formulées au cours du chapitre 3 (voir, en particulier, §3.23) ne sont plus vérifiées, ce qui rend d'autant plus imprécise une éventuelle détermination. Ainsi, l'équation $T_2 - T_1 = 0$ donne dans ce dernier cas une valeur limite pour 1. Les valeurs obtenues pour ce coefficient, dans une gamme usuelle des valeurs de R (de 50 à 100K) est de seulement 75% (1 = 0,767 pour l'exemple choisi de R = 32 pages) ce qui certainement peut surprendre.

En évaluant autrement cette limite par le coefficient 1' calculé avec le même rapport entre Q et les r pages réelles effectivement disponibles dans la zone de données (r = R - 14), on obtient la valeur 1' = 1,364 (c'est à dire, tri interne plus efficace pour les fichiers de dimension Q inférieure à 1,36 fois le nombre R de pages réelles), sans aucun doute plus satisfaisante pour l'esprit (mais aussi moins commode pour les calculs).

3.43 La généralisation

De la même forme qu'il a été montré qu'à partir d'une valeur déterminée pour la taille Q du fichier d'entrée, le tri devenait plus

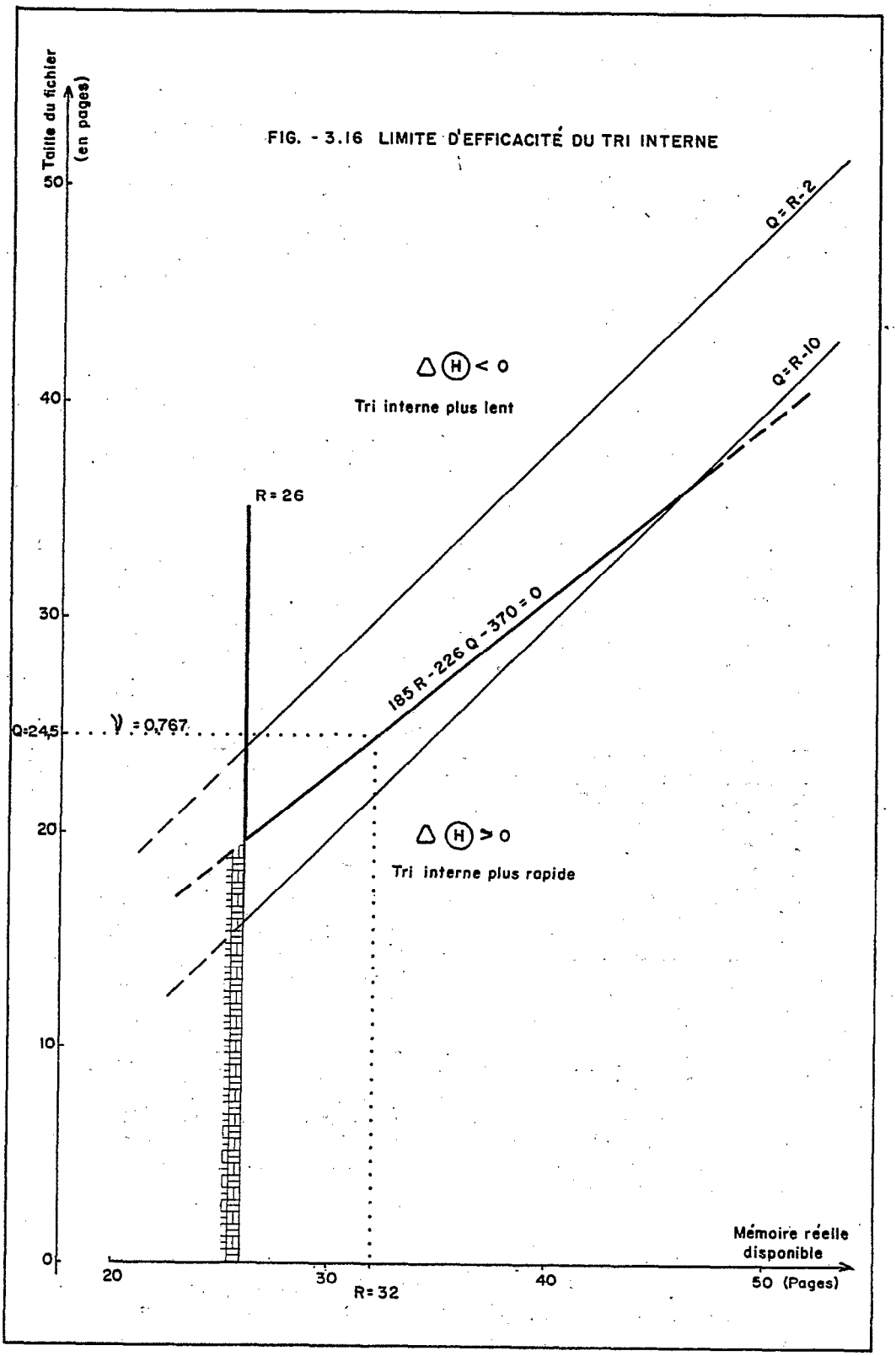


FIG. - 3.16 LIMITE D'EFFICACITÉ DU TRI INTERNE

rapide par division de ce fichier en deux parties, on peut démontrer qu'à partir d'une valeur encore supérieure, la division en trois parties devient encore plus avantageuse.

Afin de simplifier les calculs, on peut observer que dans le domaine le plus intéressant des valeurs q de taille d'un "run", c'est à dire entre R - 2 et 4 * (R - 2) (de 60K à 256K, pour R = 32 pages), le nombre E d'échanges de pages qui interviennent dans le tri varie presque linéairement entre les valeurs:

(R / 2) + 3 et 21 * (R / 2) - 19 (voir tableau 3.1), soit encore conformément à l'équation:

$$E = 10q / 3 - 17R / 6 + 31 / 3 - 2q / (3 * (R - 2)).$$

Ce dernier terme varie, pour les valeurs de q considérées, à l'intérieur de l'intervalle (-8/3,-2/3), si bien que l'on aboutit à la formule approximative suivante:

$$E = 10q / 3 - 17R / 6 + 9 ,$$

et en conséquence à T = 110E, pour le temps correspondant, exprimé en millisecondes. On doit sommer à cette dernière valeur:

- le temps passé aux chargements et déchargements des "runs", soit:

$T_i = 0$, si Q / i (=q) est inférieur à R - 14 et:

$T_i = 130 * i * (Q / i - (R - 14))$ au delà de cette limite;

- le temps passé à la fusion, indépendant (en première approximation) du nombre de "runs" qui doivent être intercalés:

$T_{fusion} = 14Q + 2\,000.$

On peut alors construire, pour chaque modèle de division du fichier initial (jusqu'à six parties), la courbe représentative du temps total consommé dans le tri, en fonction de la taille Q du fichier initial.

Le graphique 3.17 présente les courbes obtenues (pour R = 64K et P = 256K), en prenant en compte, seulement qualitativement, un décalage de chaque courbe à l'origine, pour représenter les pertes de temps très difficiles à évaluer analytiquement, telles que le temps passé au redémarrage de la phase de lecture du fichier d'entrée pour le chargement d'un "run" supplémentaire, tâche qui fait intervenir des échanges de pages et des opérations internes à la mémoire.

Il est cependant intéressant de mettre en évidence l'existence des points d'intersection entre ces courbes, qui constituent les limites des intervalles dans lesquels le temps de traitement est minimal, pour une division particulière du fichier d'entrée.

La division en plus de six sous-fichiers conduit à examiner les deux possibilités de traitement. En effet, comme on l'a déjà vu (voir

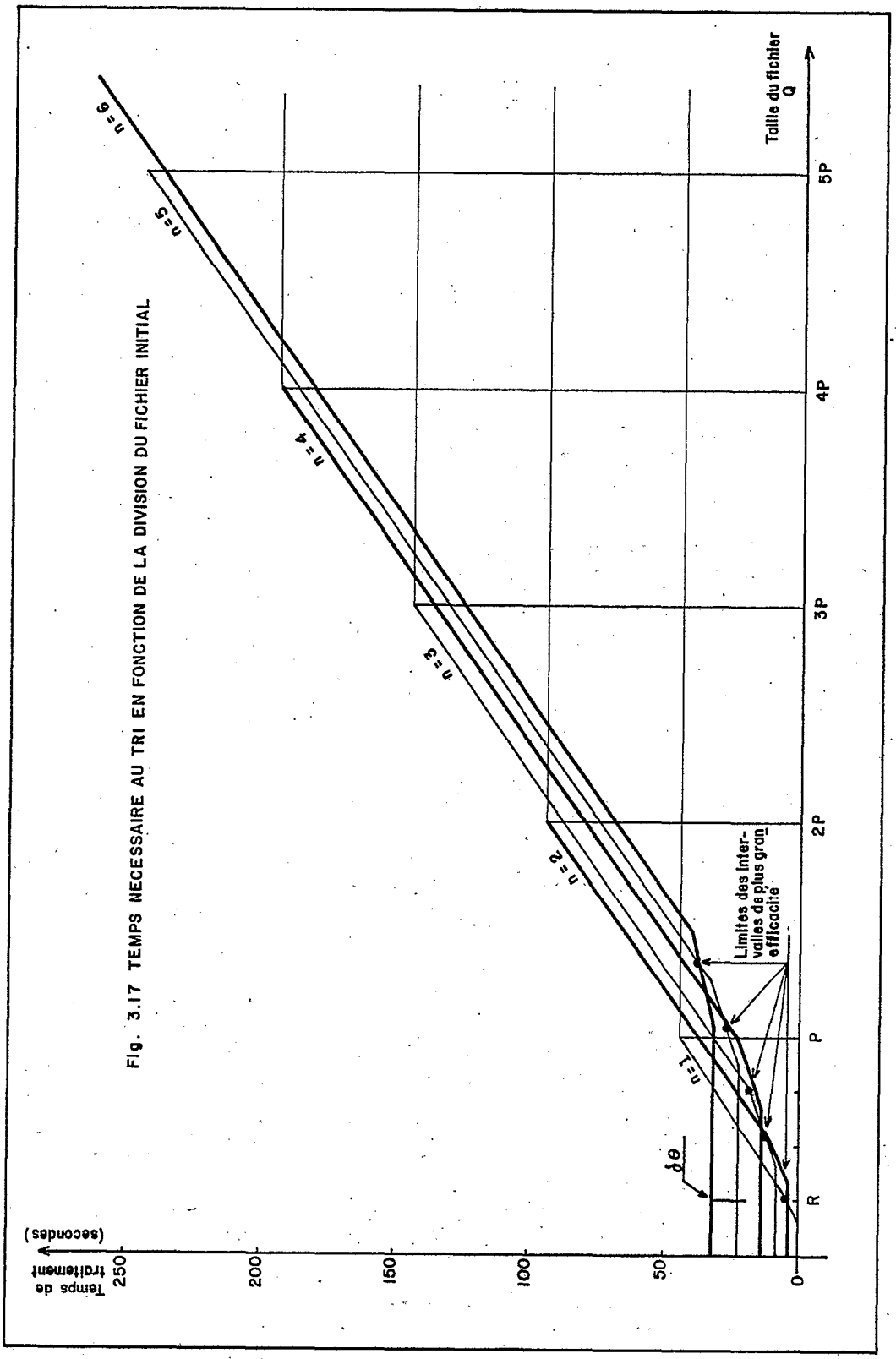


Fig. 3.17 TEMPS NECESSAIRE AU TRI EN FONCTION DE LA DIVISION DU FICHIER INITIAL

§3.26 & figure 3.13), au delà de six "runs" fusionnés simultanément, la phase de fusion demande un certain nombre d'échanges de pages, en raison de ce que tous les "buffers" ne peuvent être entièrement contenus dans la mémoire réelle. En conséquence, la fusion de 11 "runs", par exemple, serait plus rapide lorsqu'elle est réalisée en deux étapes de "6-way merge", qu'en une seule fois, avec le "11-way merge".

Pour prendre un exemple de la comparaison des deux méthodes, dans le cas déjà cité du fichier de 200 000 enregistrements munis de clés de 15 positions et divisé en 19 "runs" de 100 pages chacun, on a 220 secondes perdues en échanges de pages dans le cas du "19-way merge" (voir §3.26), tandis que le "6-way merge" utilisé pour générer les "runs" 20, 21 et 22 à partir respectivement des "runs" 1 à 6, 7 à 12 et 13 à 16, donne une perte équivalente (due à la distribution et à la fusion des 16 "runs" du pas intermédiaire), égale à:

2 * 14 * 16 * 100 millisecondes, soit 45 secondes.

Ce cas extrême, choisi pour être particulièrement défavorable à la méthode de l'étape unique (forte pagination), montre l'inefficacité de l'utilisation d'un nombre élevé de "runs", à intercaler simultanément dans le contexte de mémoire virtuelle.

3.44 Détermination du type de tri

Dans les paragraphes précédents on a mis en évidence l'existence d'un intervalle d'efficacité optimale pour chacun des schémas de division du fichier d'entrée, schéma pour lesquels on obtient, dans un intervalle déterminé, le meilleur compromis entre le phénomène de pagination excessive provoqué par l'occupation d'une zone trop importante de mémoire virtuelle, et la croissance des opérations de transfert entre mémoire interne et mémoire externe, inhérente à la réduction de la taille des "runs". En deçà de la première limite L = 1 * R, le tri est interne, externe dans le cas contraire, c'est à dire quand l'espace Q demandé pour les données est supérieur à L.

La figure 3.18 a) esquisse le procédé de détermination, que la schématisation de la partition permet d'accompagner avec plus de facilité. Une zone appelée 'COMMON' réside au début de la partition durant toute l'exécution du "job", et contient les paramètres interprétés et stockés au cours de la phase initiale. La détermination du type de tri exige:

- le calcul de la zone requise Q, à partir de la taille de l'entrée (clé + pointeur), et du nombre maximal d'enregistrements prévus;

- le calcul, en temps d'exécution, de l'aire P disponible pour stocker les données, à partir de l'adresse limite de la partition, de l'adresse de chargement du programme et de la taille de la plus grande des phases qui devra coexister avec les données;

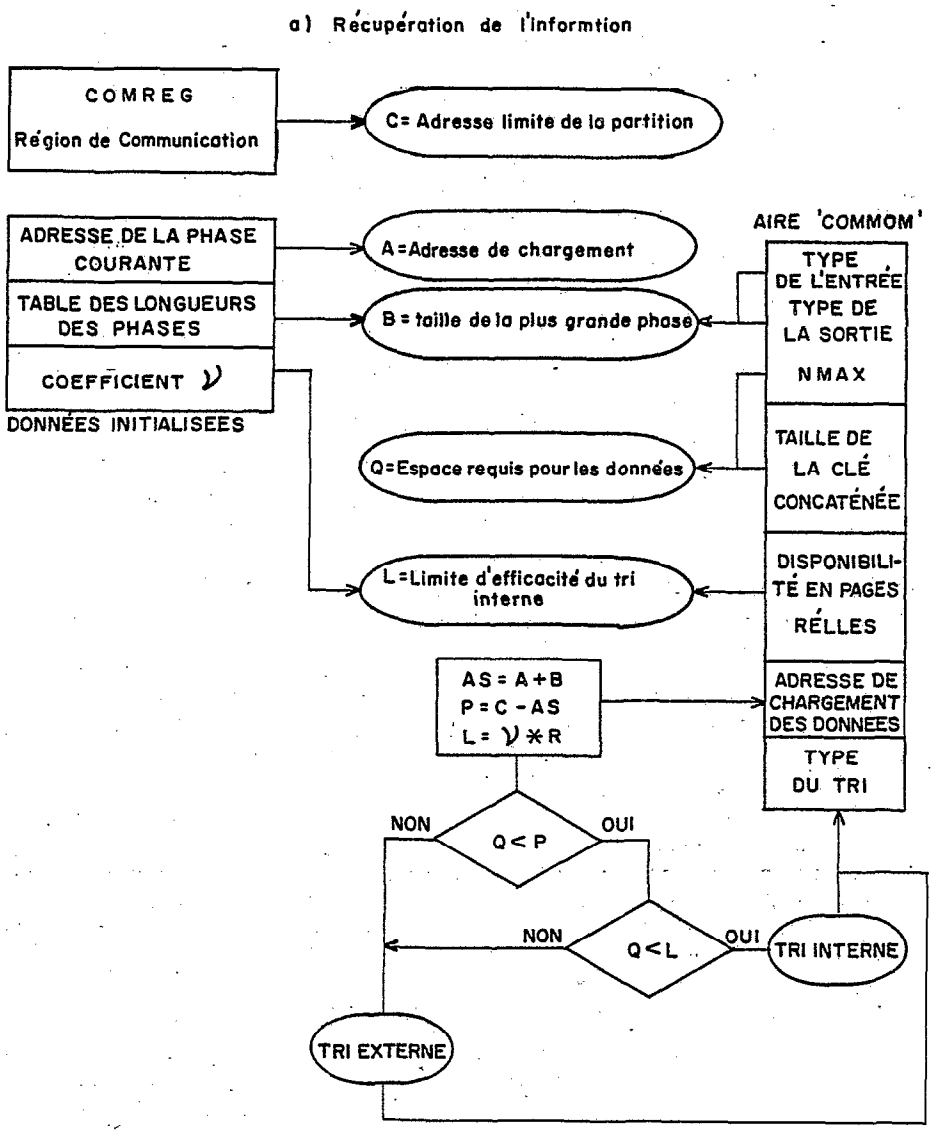


Fig. 3.18 DÉTERMINATION DU TYPE DE TRI

- le calcul de la limite d'efficacité du tri interne, en fonction du nombre de pages réelles disponibles, optionnellement fourni par l'utilisateur (64K assumés par défaut).

On peut observer en passant que l'utilisateur peut forcer le tri interne, en donnant une valeur grande pour R (ce qui justifie le test Q < P, voir figure 3.18).

Une fois déterminé le type de tri à réaliser, la phase correspondante est chargée dans la mémoire en vue de son exécution immédiate.

3.5 CONCLUSION

La spécification, au début de ce chapitre, des ressources, tant en termes de "hardware" que de "software" qui seront utilisées dans l'implantation du programme de tri projeté, a permis de préciser les méthodes définitives qui seront utilisées pour obtenir la meilleure efficacité du système développé.

Les plus divers aspects ont été examinés sous l'angle de l'efficacité: à commencer par le langage à utiliser et la structure à adopter pour le programme, jusqu'à la stratégie de stockage des "runs" sur disque, en passant par les algorithmes de tri, tant internes qu'externes, à choisir. Dans la mesure où le temps passé dans l'exécution de chaque tâche mesure l'efficacité atteinte, on a cherché à évaluer les pertes de temps inhérentes à chaque opération, ce qui a été possible moyennant une étude détaillée des mécanismes utilisés dans le système de calcul.

Par l'importance du temps passé aux échanges de pages, dont le nombre peut être très élevé, le phénomène de pagination afférent à l'utilisation de la mémoire virtuelle, est le paramètre qui a le plus retenu l'attention. Malgré de nombreuses modifications effectuées dans les algorithmes qui sont normalement optimisés pour la mémoire réelle, les estimations de temps obtenues à partir du calcul du nombre d'échanges de pages montrent que la situation critique est atteinte avec une utilisation d'à peine 30% d'excédent de mémoire virtuelle, par rapport à la mémoire réelle.

Curieusement, dans les dernières conclusions, il s'est avéré que la mémoire virtuelle ne constitue pas un outil bien adapté au tri interne, étant donné, en particulier, le nombre élévé de référenciations (comparaisons) qui doivent être opérées. De plus, elle n'est pas très efficace dans le procédé de fusion en une seule fois d'un nombre important de "runs" (pour économiser plusieurs étapes de fusion), en raison du nombre accru d'échanges de pages.

Le chapitre 4 à suivre est principalement consacré à l'implantation d'un programme de tri, projeté pour un type de fichier spécifique, conformément aux normes et concepts étudiés dans ce travail.

4.LA REALISATION

On prétend donner, dans le présent chapitre, une brève description de la réalisation du programme pour le contexte informatique pris en exemple. On cherchera à justifier, dans la mesure du possible, les options prises, tant dans la partie de l'analyse que dans la codification elle-même, dans le but de respecter les conditions d'optimisation étudiées dans les premiers chapitres de ce travail.

L'apparition d'un nouvel aspect, la souplesse du système, c'est à dire l'aptitude du programme développé au tri de plusieurs types de fichiers d'entrée, en restituant l'information classée avec plusieurs options de sortie, conduit à faire certaines concessions en ce qui concerne l'efficacité du programme dans certains cas spécifiques.

Un premier sous-chapitre est consacré à un bref examen de la structure du fichier pour lequel cette méthode a été développée: pour plus de détails on se reportera au travail original de l'auteur de la méthode d'administration de données (Costa, 1980).

Les parties suivantes présentent les méthodes adoptées pour le développement du système: tout d'abord, la tâche de passage des paramètres est examinée, en soulignant les aspects les plus importants pour l'utilisateur. Ensuite, seuls quelques points particulièrement intéressants de l'implantation des procédés de tri interne et externe ont été décrits.

Le sous-chapitre 4.5 contient une rapide narration de certains tests réalisés, dans le souci de comparer le système offert avec les autres ressources de tri à la disposition d'un utilisateur de fichiers non conventionnels.

On termine par quelques suggestions proposées pour incrémenter ce projet, à l'occasion d'éventuels développements ultérieurs.

4.1 STRUCTURE DES FICHIERS A TRIER

Un travail récent (Costa, 1980), propose un système de gestion de fichiers, qui permet l'indépendance des données. La récupération séquentielle, par rapport à la clé primaire, de l'information contenue est une ressource intrinsèque du système initial; pour obtenir une récupération du fichier selon la séquence définie par une autre clé, il est nécessaire de trier celui-ci en accord avec la séquence désirée.

L'utilisation d'un programme utilitaire de tri, fourni par le fabricant, nécessite forcément un travail additionnel de la part des utilisateurs, vu que le programme en question n'accepte que des fichiers conventionnels en entrée. Fournir une solution à ce dernier problème constitue l'objectif de l'étude réalisée.

4.11 Organisation générale du système

Dans le travail considéré, l'implantation de la ressource d'indépendance des données est obtenue par le biais de l'utilisation

d'une bibliothèque des enregistrements, dont l'objectif est d'offrir aux utilisateurs du système la facilité de récupérer seulement les champs qu'ils manipulent avec des programmes déterminés. En effet, lors de la création de son fichier de base, l'utilisateur dispose de la faculté de cataloguer dans une bibliothèque les descriptions de tous les champs de l'enregistrement, de sorte que tous les programmes d'exploitation postérieurs soient affranchis du contrôle des caractéristiques physiques du fichier.

Tant les fichiers de base du système, propres à chaque utilisateur de celui-ci, que le fichier (unique) qui sert de support à la bibliothèque, présentent une organisation de fichier indexé (à deux niveaux), dont l'index est monté selon une structure appelée "B-tree".

Cette stratégie de stockage possède les avantages suivants:

- les opérations de lecture séquentielle et de lecture séquentielle sélective (c'est à dire à partir d'une position choisie), sont autorisées;

- les opérations de récupération aléatoire, insertion et exclusion d'enregistrements, sont effectuées dans un laps de temps proportionnel au logarithme du nombre de clés existantes dans le fichier.

4.12 Organisation de chaque fichier

Dans la configuration choisie, les clés et les enregistrements sont stockés dans des zones distinctes: la figure 4.1 décrit les entrées des deux zones ('INDICE' et 'DADOS'), qui composent l'un quelconque de ces fichiers:

- la zone de l'index ('INDICE') est divisée en pages contenant un nombre variable de champs. Chaque champ est constitué, en plus de la clé K et d'un pointeur P vers l'enregistrement correspondant (adresse dans la zone de données), d'un pointeur p vers une autre page dans laquelle toutes les clés stockées sont supérieures à K;

- la zone de données ('DADOS') est gérée à partir d'une liste d'espace disponible (champ 'chaînage') et contient, en plus des enregistrements, un bloc de contrôle constitué d'une chaîne de bits, utilisée comme 'masque' pour dénoter les enregistrements de ce bloc qui sont (logiquement) exclus.

Toutes les informations indispensables à l'administration du fichier sont groupées dans la première partie ("header") de la zone réservée pour stocker l'index, selon un format illustré par la figure 4.1 a). Le paramètre k joue un rôle important dans la mesure où il constitue la limite inférieure du nombre d'entrées contenues dans une page, tandis que 2k constitue la limite supérieure de ce même nombre.

Il arrive, dans certaines applications, que la taille de la clé utilisée pour l'établissement du "B-tree" (clé contenue dans la zone 'INDICE') représente un pourcentage sensible de la taille totale de l'enregistrement (plus de 20%, par exemple), auquel cas l'utilisateur

ADRESSE DE LA RACNE	K	TAILLE DE L'ENREGISTRE - MENT	LISTES D'ESPACE DISPONIBLE		FACTEUR DE BLOC	
			ÍNDICE	DADOS		

a) *Première piste de l'aire de 'ÍNDICE'*

CHAÎNAGE	NOMBRE DE CLÉS DE LA PAGE	p_0	E_1			E_2			E_3				E_{2k}		
			p_1	P_1	K_1	p_2	P_2	K_2	p_3	P_3	K_3		p_{2k}	P_{2k}	K_{2k}

b) Piste générique du fichier 'ÍNDICE' (piste pleine)

CHAÎNAGE	BLOC DE CONTRÔLE	BLOC					
		Enregistrement	Eregistrement	Eregistrement	Eregistrement		Eregistrement

c) Piste générique du fichier "DADOS"

Fig. 4.1 FORMAT DES PISTES DES AIRES 'ÍNDICE' ET 'DONNÉES'

pourra trouver convenable de stocker dans la zone 'DADOS', seulement l'information satellite, c'est à dire l'enregistrement amputé des champs constitutifs de la clé. De cette façon, des procédures spéciales devront être élaborées pour, tant lors de la récupération qu'à la génération du fichier, concaténer la clé avec l'information satellite, de mode à reconstituer l'enregistrement initial.

4.13 La récupération de l'information

N'importe quel mode de lecture peut être utilisé pour récupérer le fichier en vue de son tri, dans la mesure où l'information est récupérée entièrement, bien entendu. Apparemment, le moyen le plus économique de faire ce travail consite à lire seulement la zone 'DADOS', en récupérant séquentiellement chaque enregitrement, tout en négligeant ceux qui sont logiquement exclus, bien que physiquement présents, après le test correspondant du bit de contrôle.

Trois raisons importantes conduisent à préférer un algorithme de récupération séquentielle du fichier, bien que ce dernier fasse intervenir deux opérations de transfert à chaque récupération (une lecture dans 'INDICE' et une autre dans 'DADOS'):

- la zone de 'DADOS', pour être administrée à partir d'un index, n'est pas construite comme une simple file, c'est à dire qu'elle ne possède pas de marque de fin de fichier, ce qui rend difficile sa récupération séquentielle;

- évidemment, dans le cas où l'utilisateur a opté pour la non répétition des champs constitutifs de la clé (information contenue dans la zone 'INDICE') dans la zone 'DADOS', on a forcément les deux accès;

- la récupération du fichier, par lecture séquentielle de la zone de données, suivrait l'ordre de création du fichier (en première approximation, étant entendu que les exclusions et inclusions postérieures à la génération sont réalisées avec utilisation de la liste chaînée d'espace disponible). Dans ces conditions, l'utilisateur ne pourrait pas connaître le degré d'ordre initial de son fichier, et on perdrait ainsi une possibilité d'optimisation de l'opération de tri (voir §3.32).

La récupération séquentielle du fichier démarre, après la localisation de la plus petite clé obtenue à partir de la racine (via le pointeur p0), par la lecture de toutes les clés de cette même page. Après avoir épuisé toutes les clés d'une page déterminée, on passe à la 'page-soeur', par le biais de la 'page-mère', en récupérant lors de ce passage la clé dont le pointeur associé adresse cette 'page-soeur'. Bien sûr l'algorithme utilise une pile pour stocker l'adresse de la page à lire et le pointeur vers la prochaine clé, à l'intérieur de cette page.

La récupération d'une clé déterminée dans 'INDICE' est suivie de la récupération de l'enregistrement dans la zone 'DADOS'.

4.2 LE PASSAGE DES PARAMETRES

Il est certain que pour un programme utilitaire, la tâche de transmission des paramètres entre l'utilisateur, qui les fournit, et le système, qui les utilise, est très importante et mérite une attention particulière. En effet, la définition des paramètres conditionne la souplesse du programme; la codification de ceux-ci, dans un langage cohérent et clair pour l'utilisateur, détermine l'utilisation correcte du système. Enfin, l'interprétation, l'analyse et la validation rigoureuse des paramètres, sont indispensables pour le développement satisfaisant du traitement suivant.

La nécessité d'une phase préliminaire réalisant l'interprétation des paramètres fournis et orientant le développement consécutif du programme, a été mise en évidence dans le chapitre précédent (voir §§3.22 & 3.44). En particulier, le regroupement dans cette phase de toutes les routines préparatoires, est justifié par le fait que celle-ci n'est pas limitée en terme d'espace.

Les deux premiers paragraphes du présent sous-chapitre sont essentiellement consacrés à une brève analyse syntaxique et sémantique des commandes de contrôle, en soulignant les paramètres les plus originaux de cette implantation.

La validation des données constitue le sujet du paragraphe 4.23, alors que les paragraphes de la fin de ce sous-chapitre ont pour objet de décrire les méthodes adoptées pour l'exécution des tâches les plus délicates de cette phase (montage de la routine 'COMPARE', par exemple).

4.21 Les cartes de contrôle

L'élaboration d'un programme de tri représente un effort d'analyse et de programmation suffisamment important pour tenter d'offrir un système apte à accepter plusieurs types d'entrée. De la même manière, l'utilisateur pourra opter pour un type de sortie du fichier classé, différent de la traditionnelle copie séquentielle sur bande magnétique.

Parallèlement, au cours de ce travail il a été vu que la connaissance de certains paramètres pouvait utilement orienter le traitement, dans le sens d'une plus grande efficacité: la taille de mémoire réelle disponible, le nombre de zones de travail allouées et le degré initial d'ordre du fichier en sont les exemples les plus importants.

Pour ces raisons, il est indispensable de créer un langage commun entre l'utilisateur qui doit fournir les valeurs des divers paramètres et le système qui doit les recevoir et les interpréter, afin d'adapter le traitement à exécuter au travail requis. Cette interface est réalisée par le moyen d'un ensemble d'instructions obligatoirement fournies sous la forme de cartes de contrôle.

La forme générale d'une commande est la suivante:

```
.COMANDO=(PARAMETRO1=VAL1,PARAMETRO2=SUB1/SUB2, commentaire
```

..PARAMETROn=VALn) commentaire

où:

- la zone d'entrée de la carte est constituée de la totalité des 80 colonnes, avec les deux premières colonnes obligatoirement remplies;

- le symbole '.' dans la première colonne d'une carte est l'identificateur d'une carte de contrôle du système 'SRT'. Un second point (dans la deuxième colonne) identifie la carte de continuation de la carte immédiatement précédente;

- le symbole 'COMANDO' est l'identificateur de l'instruction choisie, et peut appartenir à l'ensemble suivant: ENTRADA, SAIDA, OPCAO, CHAVE, ROTINA et FIM (voir tableau 4.1);

- le signal '=' est utilisé comme séparateur des zones commande et opérande, ou encore des zones identificateur de paramètre et valeur de paramètre;

- la zone opérande est délimitée par une paire de parenthèses et est constituée d'une ou plusieurs zones de définition de paramètre, séparées par le symbole ',' (séparateur de deuxième niveau);

- la valeur du paramètre peut être simple ou composée de plusieurs sous-paramètres, séparés par le symbole '/' (séparateur de troisième niveau);

- un espace termine la zone d'entrée de la carte, et marque le début de la zone commentaire. La possibilité d'une carte de continuation pour chaque commande a été réservée: cette dernière devra toujours commencer par une zone paramètre (ce qui revient à dire qu'une carte de commande peut seulement se terminer par ')' ou ',');

- les cartes de contrôle peuvent être présentées dans n'importe quel ordre, toutefois la dernière carte doit contenir l'instruction 'FIM'.

4.22 La définition des paramètres

On rencontre les définitions de paramètres, dans un ordre quelconque, à l'intérieur de la zone opérande des cartes de commandes auxquelles ces paramètres se réfèrent. Le tableau 4.1 présente un résumé des paramètres et valeurs possibles.

Les paramètres définis dans la zone opérande de l'instruction 'ENTRADA' sont:

- le paramètre 'TIPO', obligatoire, permet de spécifier le type du fichier d'entrée, parmi quatre options possibles: FITA, CARTAO, DISCO ou MASST. La première option correspond à un fichier d'entrée sur bande magnétique, sur cartes perforées pour la seconde option. La troisième se réfère à un fichier séquentiel sur disque, tandis que la dernière correspond au fichier présenté antérieurement (voir §4.12);

ID	INSTRUCTION	PARAMÈTRE	VALEUR DU PARAMÈTRE	OBLIGATOIRE OU OPTIONNEL	VALEUR (DÉFAUT)	OBSERVATIONS
●	OPÇÃO	LABEL	{SS | US | SU | UU}	O	SS	—
		TESTE	{S | N}	O	N	—
		MEMÓRIA	exprimé en K bytes	O	64	maximum = 256 K
		ÁREA	{1 | 2 | 3}	O	1	—
		RANDOMICO	{S | N}	O	S	—
	ROTINA	NOME		R	—	1er caractère alphabètique
		TAMANHO	exprimé en bytes	R	—	maximum = 242 K
		ENDEREÇO	exprimé en bytes	R	—	maximum = 72ØK
	CHAVE	1, 2, 3, 4, 5, 6, 7, 8, 9	{initial/taille du champ/format/séquence}	R	—	séquence = {A D}
	ENTRADA	TIPO	{FITA | CARTÃO | DISCO | MASST}	R	—	—
		REGISTRO	taille de l'enregistrement	R	—	maximum=724Ø
		BLOCAGEM	taille du bloc	R	—	maximum=724Ø
		NMAX	estimation du nombre d'enregistrements	R	—	—
		CONCATENA	{S | N}	O	N	—
		FECHE	{Ø | 1 | 2}	O	Ø	—
	SAÍDA	TIPO	{FITA | CARTAO | DISCO | ROTINA}	R	—	—
		BLOCAGEM	taille du bloc	R	—	maximum = 724Ø
		FECHE	{Ø | 1 | 2}	R	Ø	—
	FIM	—	—	R	—	—

TABLEAU 4.1 RESUMÉ DES CARTES DE PASSAGE DE PARAMÈTRES

- le paramètre 'REGISTRO', obligatoire, donne la valeur de la dimension totale de l'enregistrement d'entrée, exprimée en bytes, même dans le cas où seule l'information satellite est stockée dans la zone 'DADOS'. Le système n'accepte que des enregistrements de longueur fixe;

- le paramètre 'BLOCAGEM', obligatoire, donne la valeur de la taille du bloc d'entrée, limitée à 7 240 bytes (utilisation de disques modèle 2314 ou 2319, dont les pistes sont de 7 294 bytes). Ce paramètre est requis même dans le cas où le fichier à un facteur de bloc égal à 1;

- le paramètre 'NMAX', obligatoire, donne une estimation du nombre maximal d'enregistrements: il n'est pas limité, si ce n'est par la valeur maximale (2 147 483 647) autorisée par une représentation de 4 bytes. Dans le cas où le nombre d'enregistrements à trier n'est pas connu, une limite supérieure doit être fournie pour NMAX (choisie la plus voisine possible de la valeur réelle supposée), afin de déterminer le type de tri (voir §§3.44 & 4.25);

- le paramètre 'FECHE', optionnel, permet de spécifier l'option de clôture du fichier d'entrée (dans le cas d'une bande magnétique) à la fin du traitement: Les valeurs 0, 1 et 2 correspondent respectivement au réembobinage simple ("REWIND"), réembobinage complet ("UNLOAD"), ou au non réembobinage ("NOREWIND"). Quand ce paramètre n'est pas spécifié par l'utilisateur, le système assume automatiquement le réembobinage de la bande d'entrée à la fin de la lecture;

- le paramètre 'CONCATENA', optionnel, permet de spécifier si l'enregistrement complet est stocké dans la zone 'DADOS' (CONCATENA=N), ce qui est le cas normal (assumé par défaut). La valeur CONCATENA=S (de 'sim') indique au système que seule l'information satellite est stockée dans cette zone et, dans ce cas, la convention de concaténation de la clé à gauche de l'information est validée. Bien entendu, ce paramètre n'est utilisé que dans le cas du fichier 'MASST'.

Les paramètres définis dans la zone opérande de la commande 'SAIDA' sont:

- le paramètre 'TIPO', obligatoire, permet de spécifier le type de sortie entre quatre options FITA, CARTAO, DISCO ou ROTINA. La dernière option permet de restituer à l'utilisateur le fichier ordonné dans la mémoire interne, de telle sorte qu'une subroutine d'application (dont les caractéristiques sont spécifiées dans la commande 'ROTINA') puisse être chargée, pour éditer un état par exemple. On évite ainsi, dans un tel cas, l'écriture du fichier de sortie (ordonné) sur un support externe, pour le lire de nouveau par le programme d'édition. Cette ressource, qui doit être développée, pourra permettre l'utilisation de ce programme comme procédure spécifique interne pour un langage de haut niveau, comme le 'SORT INTERNE' de Cobol ou PL/I, par exemple;

- les paramètres 'BLOCAGEM' et 'FECHE' ont la même signification et les mêmes attributs que dans le cas de l'instruction 'ENTRADA'.

Les paramètres définis dans la zone opérande de l'instruction 'ROTINA' sont:

- le paramètre 'NOME', obligatoire, définit une phase contenue dans la "Core Image Library", qui devra être chargée à la fin du programme de tri, avec les données présentes dans la mémoire. Aussi s'agit-il d'une chaîne d'au plus 8 caractères alphanumériques, dont le premier est obligatoirement alphabétique;

- le paramètre 'TAMANHO', obligatoire, informe sur la dimension de la routine qui doit être chargée. Cette valeur est exprimée en bytes, et a été limitée à 247 808 bytes dans les conditions d'exécution du programme, ce qui correspond à la taille totale de la partition virtuelle (242K bytes);

- le paramètre 'ENDERECO', obligatoire, informe sur l'adresse absolue de charge de cette routine. Exprimée en bytes, la valeur permise, dans les mêmes conditions d'exploitation, va de 0 à 737 280 bytes (720K bytes). Au moment de spécifier ce dernier paramètre, l'utilisateur doit prendre garde de choisir une valeur suffisamment élevée (à l'intérieur de sa région d'exécution), pour ne pas effacer la partie finale des données classées.

Les paramètres définis dans la zone opérande de l'instruction 'OPCAO' sont:

- le paramètre 'LABEL', optionnel, qui permet de spécifier si il existe, ou non, une zone de label dans les fichiers d'entrée et de sortie. Pour chaque fichier, le code 'S' signifie "Standard label", tandis que le code 'U' est utilisé pour notifier l'absence de label. Les valeurs possibles pour ce paramètre sont donc: SS, SU, US ou UU. La première de ces deux lettres se réfère au fichier d'entrée. Dans le cas où la valeur de ce paramètre n'est pas spécifiée, le système assume la valeur 'SS';

- le paramètre 'TESTE', optionnel, permet de réaliser une simple critique des cartes de contrôle du programme, jusqu'à la détermination du type de tri, sans exécution de celui-ci. La valeur de 'TESTE' peut être 'S' ou 'N', avec cette dernière comme valeur assumée par défaut, correspondant à l'exécution complète du programme;

- le paramètre 'MEMORIA', optionnel, spécifie la quantité de mémoire réelle disponible (exprimée en K octets), pour la partition d'exécution du "job". Dans le cas où ce paramètre n'a pas été spécifié, la valeur normalement assumée est 64 (soit encore R=32 pages). L'utilisateur peut néanmoins fournir n'importe quelle valeur de l'intervalle 0 à 242 (taille de la partition d'exécution). Rappelons que la limite d'efficacité du tri interne est proportionnelle à R (voir §3.42), aussi l'utilisateur pourra-t'il forcer un tri externe en spécifiant une valeur faible pour ce paramètre, et inversement une valeur forte si l'autre partition est inactive par exemple;

- le paramètre 'RANDOMICO', optionnel, spécifie, sous une forme booléenne, quel est le degré initial d'ordre du fichier. La valeur 'S', qui est prise par défaut, correspond à un fichier désordonné. Le

cas d'un fichier d'entrée ordonné est noté 'N'. La spécification de ce paramètre permet d'orienter le choix d'une stratégie de stockage des "runs" sur disque (voir §3.32), dans le cas du tri externe;

- le paramètre 'AREA', optionnel, donne le nombre de zones de travail qui peuvent être utilisées par le programme, pour distribuer les "runs" dans la phase de tri externe. Les valeurs possibles sont 1, 2 ou 3, la valeur 1 étant celle assumée par défaut, en cas de non spécification de ce paramètre.

L'instruction 'CHAVE' peut compter dans sa zone opérande jusqu'à neuf paramètres, symbolisés par les chiffres de 1 à 9. Ces paramètres sont utilisés pour caractériser les champs de la clé choisie. La forme générale de définition est la suivante:

n = n1 / n2 / format / séquence

où:

- n est un chiffre de 1 à 9, qui désigne l'ordre de priorité du champ considéré dans la constitution de la clé;

- n1 représente la position initiale de ce champ, repérée par rapport au début de l'enregistrement (clé + information satellite, dans le cas 'CONCATENA = SIM' en particulier). La valeur maximale pour n1 a été limitée à 4 095, pour convenance de programmation;

- n2 représente la taille, en octets, du champ considéré, et est limité selon le format dudit champ. Il convient de noter que dans le cas 'CONCATENA = SIM', aucun champ ne peut être situé pour partie dans la zone 'INDICE', et pour le reste dans la zone 'DADOS': il devra alors être considéré comme étant constitué de deux champs distincts;

- le sous-paramètre format spécifie le type de champ à traiter, parmi les sept options suivantes:

  * AN pour un champ alphanumérique d'au plus 256 octets,
  * NM pour un champ numérique d'au plus 256 octets,
  * BI pour un champ binaire d'au plus 256 octets,
  * DC pour un champ 'décimal compacté' d'au plus 16 octets,
  * FX pour un champ entier de 2 ou 4 octets,
  * PF pour un champ réel de 4 ou 8 octets,
  * DZ pour un champ 'décimal zoné' d'au plus 16 octets;

- le sous-paramètre séquence donne la hiérarchie à adopter, représentée par la valeur 'A' (séquence ascendante) ou 'D' (séquence descendante).

Etant entendu que les champs constituent des paramètres différents, l'ordre dans lequel on les rencontre sur la carte 'CHAVE' est indifférent, la hiérarchie étant intégralement définie par les chiffres utilisés.

La dernière carte de contrôle contient la commande 'FIM', et n'autorise aucun paramètre.

La figure 4.2 illustre la codification des cartes de contrôle pour un exemple particulièrement simple: le tri d'un fichier organisé comme il a été décrit antérieurement (voir §§4.11 & 4.12), selon un champ 'nom' alphanumérique de 50 octets, générant une bande magnétique en sortie, avec un label standard et un facteur de bloc égal à 10. Par l'absence de carte 'OPCAO', on suppose que le degré d'ordre initial du fichier est faible et que 32 pages de mémoire réelle sont utilisées dans la partition, ainsi qu'une seule zone de mémoire auxiliaire dans le cas de tri externe.

4.23 La validation des paramètres

Quel que soit le type d'erreur rencontré au cours de la phase de critique, un message d'erreur est édité sur le listing de sortie, précédé du code:

SR0nmX

où:

- le préfixe SR caractérise le système de tri (SRT);

- le chiffre 0 caractérise la phase de critique;

- n et m sont les deux chiffres qui composent le numéro d'ordre du message édité;

- X est une lettre de l'ensemble I, A ou F, qui caractérise le niveau de l'erreur rencontré et la qualification du message correspondant: Informatif, Avertissement ou Fatal.

Après la validation de toutes les cartes de contrôle, l'exécution du "job" est arrêtée, soit si la valeur du paramètre 'TESTE' est 'SIM', soit s'il y a occurrence d'une erreur de type 'F'. La validation des cartes de contrôle est réalisée selon plusieurs niveaux. Localement les conditions suivantes sont vérifiées:

- la carte de contrôle est identifiable (symbole '.' dans la première colonne);

- l'identificateur de commande est valable (appartient bien à l'ensemble défini);

- une carte de continuation apparaît seulement après une carte commande;

- les séparateurs sont correctement utilisés, et au niveau approprié;

- l'identification d'un paramètre est valable (appartient à l'ensemble des paramètres de l'instruction dans laquelle il a été rencontré);

- la valeur d'un paramètre numérique est réellement numérique;

- la valeur d'un paramètre littéral appartient à l'ensemble des codes permis;

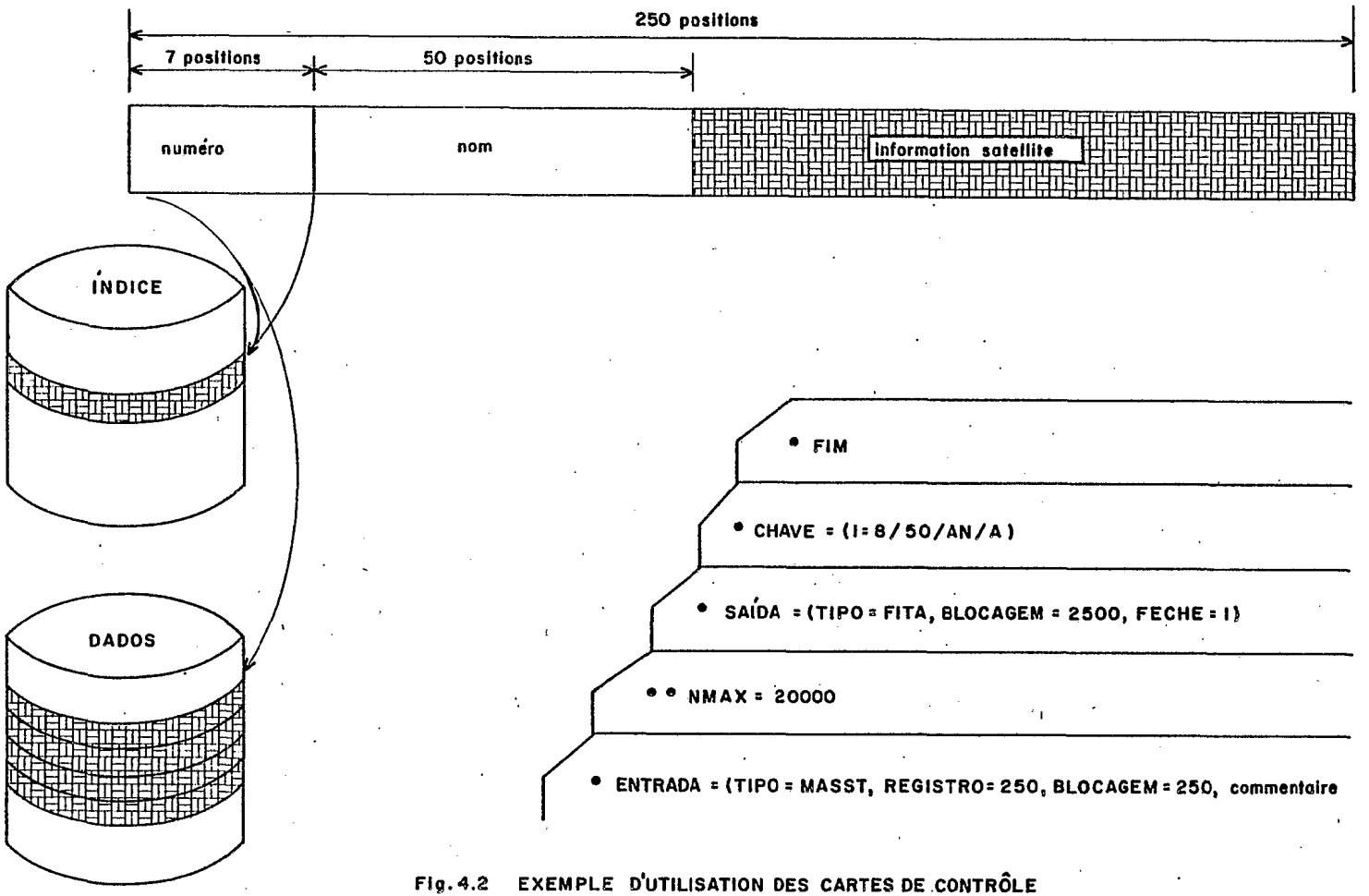


Fig. 4.2 EXEMPLE D'UTILISATION DES CARTES DE CONTRÔLE

- la valeur d'un paramètre numérique est incluse dans l'intervalle de définition de celui-ci (entre la valeur minimale et la valeur maximale).

Au fur et à mesure que ces paramètres et commandes sont rencontrés, validés et décodifiés, ils sont stockés dans une zone "COMMON" (voir §4.24), pour être utilisés au moment opportun dans le cours d'exécution du programme. Lorsque l'on rencontre la commande 'FIM', un second niveau de validation est activé. Globalement les conditions suivantes sont vérifiées:

- toutes les commandes requises ont été rencontrées;

- aucune commande n'a été encontrée plus d'une fois;

- tous les paramètres requis ont été rencontrés;

- aucun paramètre n'a été encontré plus d'une fois;

- tous les sous-paramètres d'un champ de clé ont été rencontrés, et seulement une fois;

- il existe au moins un champ de clé pour le tri.

Le dernier niveau d'analyse et validation des paramètres se préoccupe de la compatibilité des paramètres entre eux. Les conditions suivantes sont alors vérifiées:

- la carte 'ROTINA' est rencontrée dans le cas où le paramètre 'TIPO' de la carte 'SAIDA' possède la valeur 'ROTINA', et dans ce cas seulement;

- le paramètre 'CONCATENA' étant présent, le type d'entrée en est compatible ('MASST');

- si le type d'entrée est 'MASST' ou 'DISCO', le label d'entrée doit être standard;

- si le type d'entrée est 'CARTAO', le label correspondant doit être omis et la taille de l'enregistrement est identique au blocage d'entrée, égal à 80;

- si le type d'entrée est 'MASST', 'DISCO' ou 'CARTAO', le paramètre 'FECHE' est omis (message d'avertissement);

- on a une situation semblable pour la sortie;

- la dimension de chaque champ de clé est inférieure à la taille globale de l'enregistrement;

- les positions définies pour chaque champ de clé 'tombent' bien à l'intérieur de l'enregistrement;

- la taille spécifiée pour un champ de clé est compatible avec les limites imposées pour son type (voir §4.22).

4.24 La zone "COMMON"

Une des fonctions importantes de la phase initiale du programme est d'interpréter certains paramètres fournis par l'utilisateur, en élaborer d'autres et les transmettre à celles des autres phases du programme qui auront à s'en servir. Ces données interprétées sont placées dans une zone spéciale, localisée au début de l'espace adressable de la partition (voir figure 3.18) et protégée vis à vis des chargements successifs des phases. La figure 4.3 présente le format de cette zone. Les paramètres qui y sont contenus sont de trois types distincts:

- les paramètres directement fournis par l'utilisateur et qui sont situés pour la plupart en début de zone. Ils sont stockés sous la forme de bytes ou sous la forme décimal condensé, et seront utilisés au fur et à mesure des nécessités au cours de l'exécution du programme;

- les paramètres obtenus au cours d'une autre phase et qui doivent être communiqués à une troisième: par exemple, la table des adresses des "runs", générée lors de la phase de distribution et utilisée pendant la phase de fusion dans le tri externe (voir §§4.41 & 4.42). Dans cet ensemble, la taille du bloc de contrôle, dernier champ de la zone de 'COMMON', constitue un cas particulier puisqu'il est obtenu à partir de la phase d'entrée dans le cas du fichier 'MASST', alors qu'il est élaboré au cours de la phase de critique pour les autres types d'entrée (voir §4.31);

- les paramètres élaborés dans la propre phase de passage des paramètres. En plus de l'adresse de début de la zone de données et de la valeur du type de tri (obtenus dans la routine finale de détermination du type de tri), deux paramètres sont également obtenus et méritent une attention particulière.

La zone 'DOKEY' de l'aire de 'COMMON' est en réalité constituée d'un ensemble d'instructions destinées à placer la clé de tri dans la mémoire au moment de la lecture de chaque enregistrement d'entrée. On conçoit facilement que, en fonction des champs de clé définis par l'utilisateur à l'aide de la commande 'CHAVE', ces instructions devront être modifiées, en particulier lorsqu'il existe un ou plusieurs champs de type décimal zoné, à compacter avant d'inclure dans la clé concaténée (voir §4.31 et fig. 4.4 & 4.5). Aussi la routine 'DOKEY' est-elle construite en temps d'exécution à partir d'une succession d'instructions correspondantes à chacun des (jusqu'à 9) champs de la clé. Chaque instruction réalise le transfert du champ considéré de l'enregistrement d'entrée vers une zone de formation de l'entrée, elle-même transférée, une fois complète, vers la zone de données de la mémoire. La dimension obtenue pour la clé concaténée est stockée dans un autre champ de la zone 'COMMON'.

Le traitement de la zone 'COMPARE' est très semblable à celui de la zone 'DOKEY'. En effet, de la même façon que le positionnement et la taille des champs de la clé conditionnent le module de concaténation de l'entrée, le format et la séquence choisie conditionnent le module de comparaison de deux clés. On perçoit aisément par exemple, que l'instruction de comparaison pour une zone décimal condensé soit différente de l'instruction correspondante pour

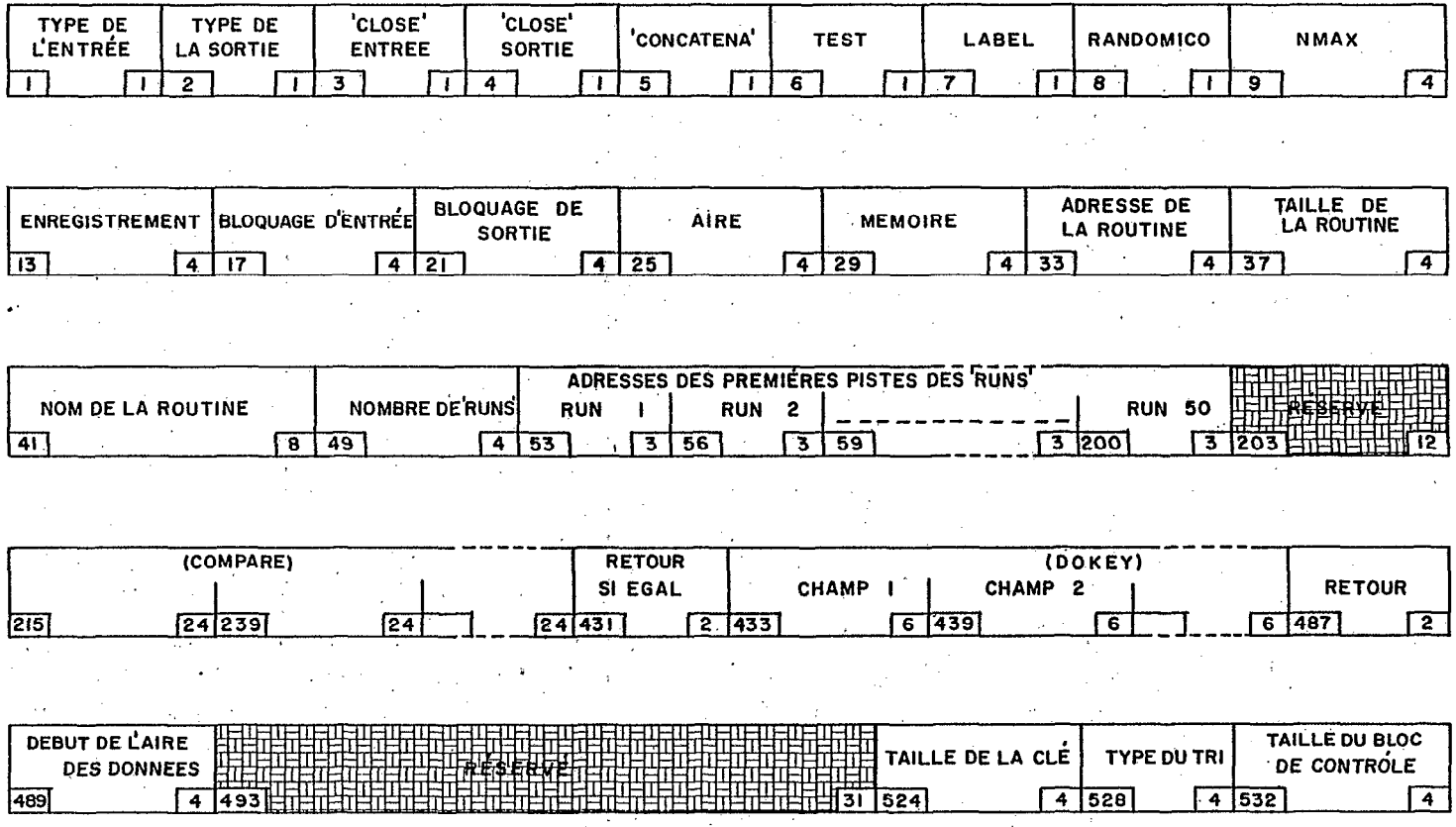


Fig. 4.3 FORMAT DE L'AIRE DU 'COMMON' (Taille en bytes)

une zone de caractères, même de longueur identique. Aussi, la routine 'COMPARE' est-elle également élaborée, en temps d'exécution, dans la phase initiale, pour être utilisée dans les phases de tri (interne ou externe) et de fusion. En conséquence, à n'importe quel moment du programme, la comparaison de deux clés sera réalisée, après le positionnement adéquat des deux registres utilisés comme pointeurs pour ces clés, avec un branchement inconditionnel vers la routine 'COMPARE' dans la zone 'COMMON', le résultat étant restitué à la sortie à travers des deux bits du "Condition Code", comme n'importe quelle instruction élémentaire de comparaison.

4.25 La détermination réelle du type de tri

Dans l'implantation, la détermination du type de tri à exécuter est effectuée en accord avec la théorie élaborée au cours du chapitre 3 de cette étude. Il convient à peine de souligner dans ce paragraphe le calcul, en temps d'exécution, de l'aire disponible pour stocker les données.

L'adresse limite de la partition (C) est obtenue par utilisation d'une macro-instruction ("COMRG"), qui récupère cette information gardée dans l'aire réservée au système d'exploitation, plus particulièrement dans une zone spéciale de communication avec les programmes d'application. L'adresse de charge A (voir fig. 3.18), commune à toutes les phases du tri interne, est la même que l'adresse de charge de la phase de validation des paramètres. En utilisant cette propriété on obtient facilement cette adresse absolue, là encore en temps d'exécution.

Enfin, la taille (B) de la plus grande des phases d'entrée et de sortie (étant entendu que la phase de tri est la plus petite de toutes) est obtenue par recherche dans la table des tailles de phases, à partir du type d'entrée et du type de sortie fournis par l'utilisateur.

Finalement, la taille maximale (P) de la zone des données est obtenue par la formule:

P = C - AS, où AS = A + B, est l'adresse du début de la zone des données.

4.3 LES PHASES DU TRI INTERNE

L'organisation générale du programme est conforme à celle qui a été définie dans la partie théorique de notre étude. Une fois choisie la voie du tri interne au cours de la phase initiale du traitement, le calcul est segmenté en trois phases distinctes, de sorte qu'un espace maximal soit obtenu pour le fichier d'entrée, qui sera stocké en une seule fois dans la zone de mémoire disponible pour les données.

Le premier paragraphe de ce sous-chapitre présente la phase de lecture et de stockage des données dans la mémoire principale, dans le cas particulier du type de fichier d'entrée pour lequel ce système a été initialement développé. A cette occasion, les étapes

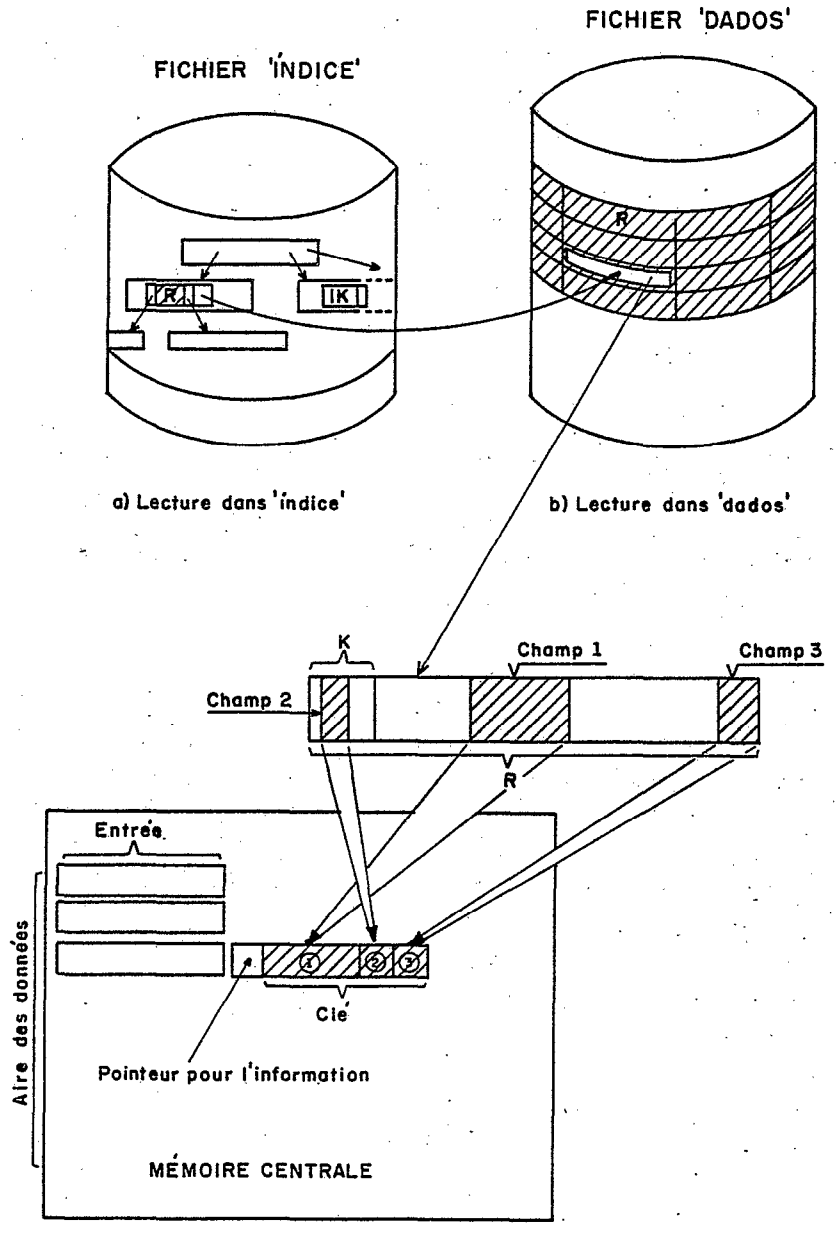
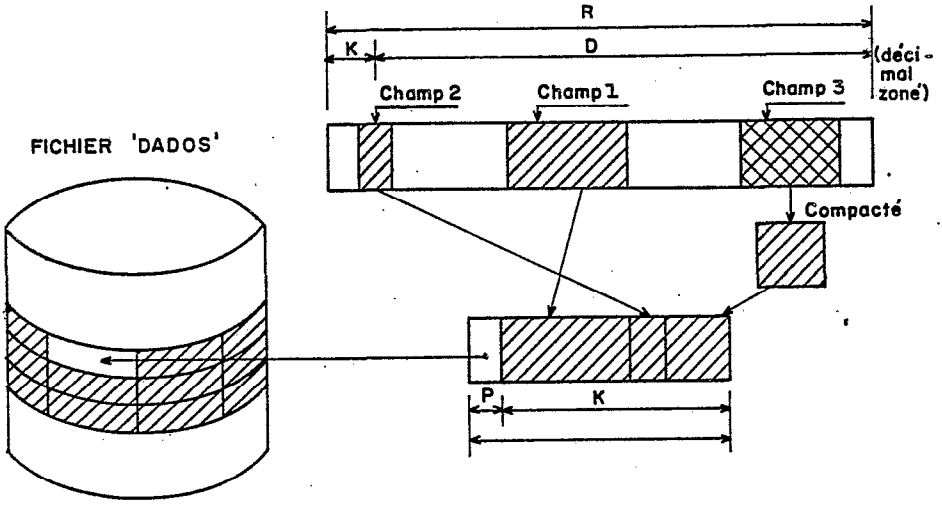
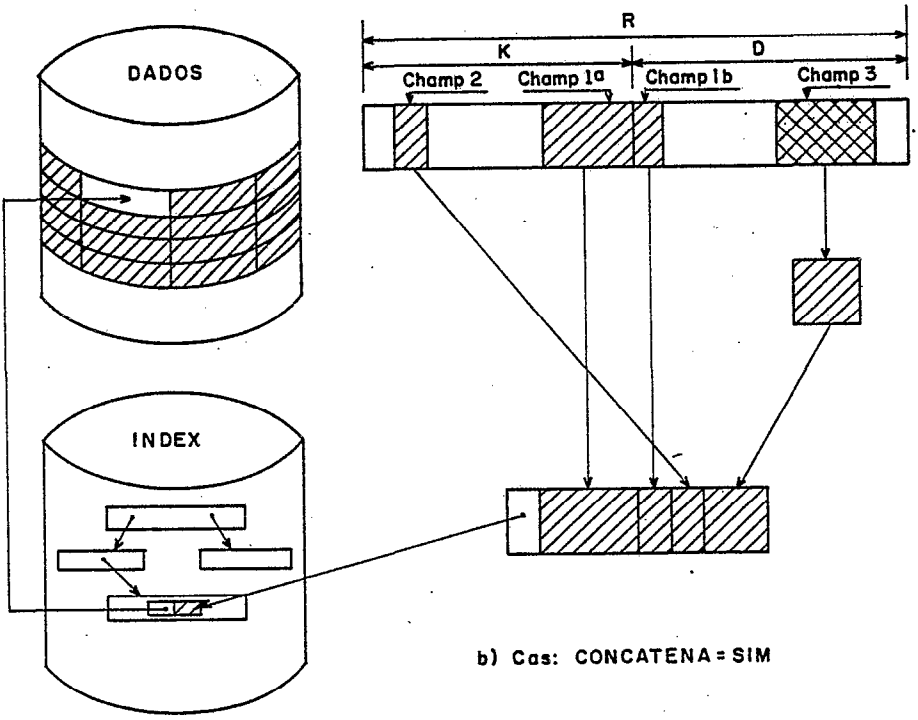


Fig. 4.4 Formation des entrées à trier



a) Cas: CONCATENA = NÃO



b) Cas: CONCATENA = SIM

Fig. 4.5 Rôle du paramétre CONCATENA dans la formation de l'entrée

supplémentaires qui devront être réalisées pour la codification des phases correspondantes à d'autres types de fichiers sont proposées en suggestion.

Dans le deuxième paragraphe, la méthode utilisée pour réaliser la phase de tri interne proprement dite (en accord avec les normes définies dans la partie théorique) est abordée. Certainement, avec les études réalisées et rapportées dans le deuxième (voir §§2.34, 2.38 & 2.39) et le troisième chapitre (voir §§3.24 & 3.42), cette phase a déjà été définie avec précision; aussi souligne-t'on seulement dans ce chapitre les points les plus significatifs de la codification.

On termine finalement la description de l'implantation du tri interne par la présentation de la phase de déchargement et écriture de la liste ordonnée en un fichier de sortie séquentiel sur bande magnétique, seule phase de sortie qui ait été réalisée. De la même forme que pour la phase d'entrée, les modifications nécessaires en vue de l'élaboration d'autres processus de sortie sont brièvement examinées.

4.31 La phase d'entrée

Selon le type d'entrée stipulé par l'utilisateur, et après la phase de critique des cartes de contrôle, une phase de lecture des données est chargée dans la mémoire. En effet, chaque type d'entrée nécessite une phase de lecture propre, dont les caractéristiques sont les suivantes:

- celle-ci contient les modules de définition logique et physique du fichier qui doit être lu ('DTF'), et le (ou les) "buffer(s)" correspondant(s). La taille de ce "buffer", tout comme certains paramètres de la 'DTF', devront être altérés en temps d'exécution, en fonction des paramètres fournis par l'utilisateur ('REGISTRO', 'BLOCAGEM', 'FECHE');

- en principe, après l'ouverture du fichier, la phase de lecture est calquée sur l'algorithme de récupération séquentielle du fichier d'entrée considéré. Selon que ce fichier est stocké en mémoire externe à accès direct (TIPO = 'MASST' ou 'DISCO'), ou non (TIPO = 'CARTAO' ou 'FITA'), la lecture de chaque enregistrement pourra induire son écriture sur une zone de disque ('DADOS');

- elle réalise également la concaténation des champs de clé dans une zone unique, par l'intermédiaire du module 'DOKEY' (voir §4.24), une fois compactés tous les champs du type décimal zoné (s'il y en a). L'entrée qui doit être stockée en mémoire interne est constituée d'un pointeur vers l'information et de la clé, éventuellement concaténée (voir fig. 4.4). Le pointeur est formé de 5 bytes, dont les trois premiers constituent l'adresse relative de la piste d'écriture de l'enregistrement dans la zone de 'DADOS', et les deux derniers précisent la position de celui-ci à l'intérieur de cette piste (utilisation du bloc de contrôle). Pour convenance de codification, la clé est située à la droite et le pointeur (de longueur fixe dans tous les cas) à la gauche de cette entrée. Il est intéressant de remarquer que ce pointeur est obtenu par simple lecture, vu qu'il figure déjà dans la zone 'INDICE' du fichier de type

'MASST'. Il pourra être élaboré directement à partir de la position courante de l'enregistrement dans le fichier (TIPO = 'DISCO', ou organisation séquentielle indexée), ou après le stockage de l'enregistrement dans la zone 'DADOS' créée par la propre phase.

On a cherché à optimiser la réalisation de la phase de lecture dans le cas du fichier de type "B-tree" (TIPO = 'MASST'). Ainsi par exemple, en observant qu'il est très possible que deux enregistrements qui doivent être lus consécutivement appartiennent à la même piste, l'adresse de la piste qui doit être lue est toujours comparée avec celle de la piste qui vient de l'être, de façon à économiser un accès additionnel à la zone 'DADOS', dans le cas où ces deux pistes sont identiques.

Le rôle joué par le paramètre 'CONCATENA' dans la formation de l'entrée est intéressant. La figure 4.5 a) montre que dans le cas 'CONCATENA' = 'NAO', le pointeur mémorisé dans l'entrée indique directement l'enregistrement correspondant dans la zone 'DADOS'. De plus la concaténation des champs sélectionnés pour former la clé de tri est simple. Dans la figure 4.5 b), correspondant au cas 'CONCATENA' = 'SIM', on peut observer que le pointeur stocké se réfère à la clé, l'information satellite étant elle-même récupérée par le biais du pointeur propre à la structure du fichier (voir §4.12 et figure 4.1). Enfin la propre formation de l'entrée à trier dans la mémoire fait intervenir des opérations plus complexes de la subroutine 'DOKEY' (information qui provient de 'INDICE' et 'DADOS', au lieu de 'DADOS' seulement).

4.32 La phase de tri

En raison de l'organisation générale adoptée pour le programme (voir fig. 3.7), la phase de tri est unique, c'est à dire indépendante du type d'entrée ou de sortie, comme la phase de critique des paramètres et la phase de fusion des "runs" dans le tri externe. De plus, une caractéristique principale de cette phase est d'occuper très peu d'espace de mémoire (à peine 2K), en raison de l'absence de "buffers" et de 'DTF'. Lors de la codification, on s'est simplement soucié de l'efficacité, au point de vue du temps, de cette opération. Les raffinements suivants ont finalement été retenus pour le "Quicksort":

- choix du "median" entre trois pour la clé-pivot (méthode proposée par Singleton);

- tri en une seule étape finale séparée, par insertion directe, des petits fichiers de moins de 12 clés (méthode proposée par Sedgewick);

- empilage systématique du sous-fichier situé le plus à droite, et étape finale d'insertion d'arrière vers l'avant (de droite à gauche).

Les deux dernières conventions répondent à des préoccupations apparemment différentes. En réalité, en raison des faibles valeurs rencontrées pour la limite L d'efficacité du tri interne, le programme devra toujours utiliser un faible pourcentage de la mémoire virtuelle,

ce qui justifie le fait de l'optimiser dans un contexte de mémoire réelle (point 2), tout en se souciant de ne pas pénaliser le cas extrême d'utilisation (malgré tout modérée) de la mémoire virtuelle (dernier point).

De la même façon que la routine 'COMPARE' utilise deux registres pour indiquer la position des clés qui doivent être comparées, l'échange de ces mêmes clés (méthode "Quicksort") et le mouvement de celles-ci (méthode d'insertion directe), sont réalisés à partir de modules qui utilisent les mêmes techniques de codification. L'utilisation des instructions dites 'de registre à mémoire', voire de celles dites 'de registre à registre', doit favoriser l'optimisation du programme, en diminuant la durée des opérations qui utilisent ces indices.

Soit 261 bytes la taille maximale de l'entrée à trier (cinq bytes du pointeur, suivis d'une clé d'au plus 256 positions), le mouvement de l'entrée peut être réalisé avec une ou deux instructions, selon la valeur relative de la taille de la clé par rapport à la valeur maximale de l'argument permis dans ces instructions.

4.33 La phase de sortie

La phase de sortie qui a été réalisée permet d'obtenir une copie du fichier trié sur une bande magnétique. En première approximation, cette dernière phase dépend seulement du type de sortie choisie par l'utilisateur. Toutefois le paramètre 'CONCATENA' joue un rôle important dans cette dernière phase, dans le cas où le fichier d'entrée est du type 'MASST'. La figure 4.6 montre comment l'information est récupérée dans ce cas, à partir d'une succession de deux accès sur disque, l'un dans la zone 'INDICE' et l'autre consécutif dans l'aire de 'DADOS'. Bien sûr, dans le cas 'CONCATENA' = 'NAO', un seul accès est nécessaire, vu que l'information classée dans la mémoire adresse directement l'enregistrement correspondant. Aussi, la facilité offerte pour le stockage de l'information satellite à la place de l'enregistrement complet, devra-t'elle être utilisée parcimonieusement, après un examen minutieux des bénéfices que l'on peut espérer en tirer.

L'élaboration d'une phase de sortie, dans le cas du type de sortie 'ROTINA', n'a pas été réalisée, et constituerait un travail complémentaire intéressant, d'autant que l'on peut en espérer un gain de temps important, étant donné l'élimination dans ce cas d'une opération additionnelle de copie du fichier en mémoire externe.

Il est clair que le travail essentiel de cette phase consiterait à charger le module de l'utilisateur dans le mémoire, en informant de certains paramètres (en nombre et en type à définir par convention), et finalement donner le contrôle à cette routine (utilisation de la macro-instruction 'FETCH', par exemple).

4.4 LES PHASES DU TRI EXTERNE

Au vu des résultats établis au chapitre 3 de cette étude, il existe plusieurs points communs entre le tri externe et le tri

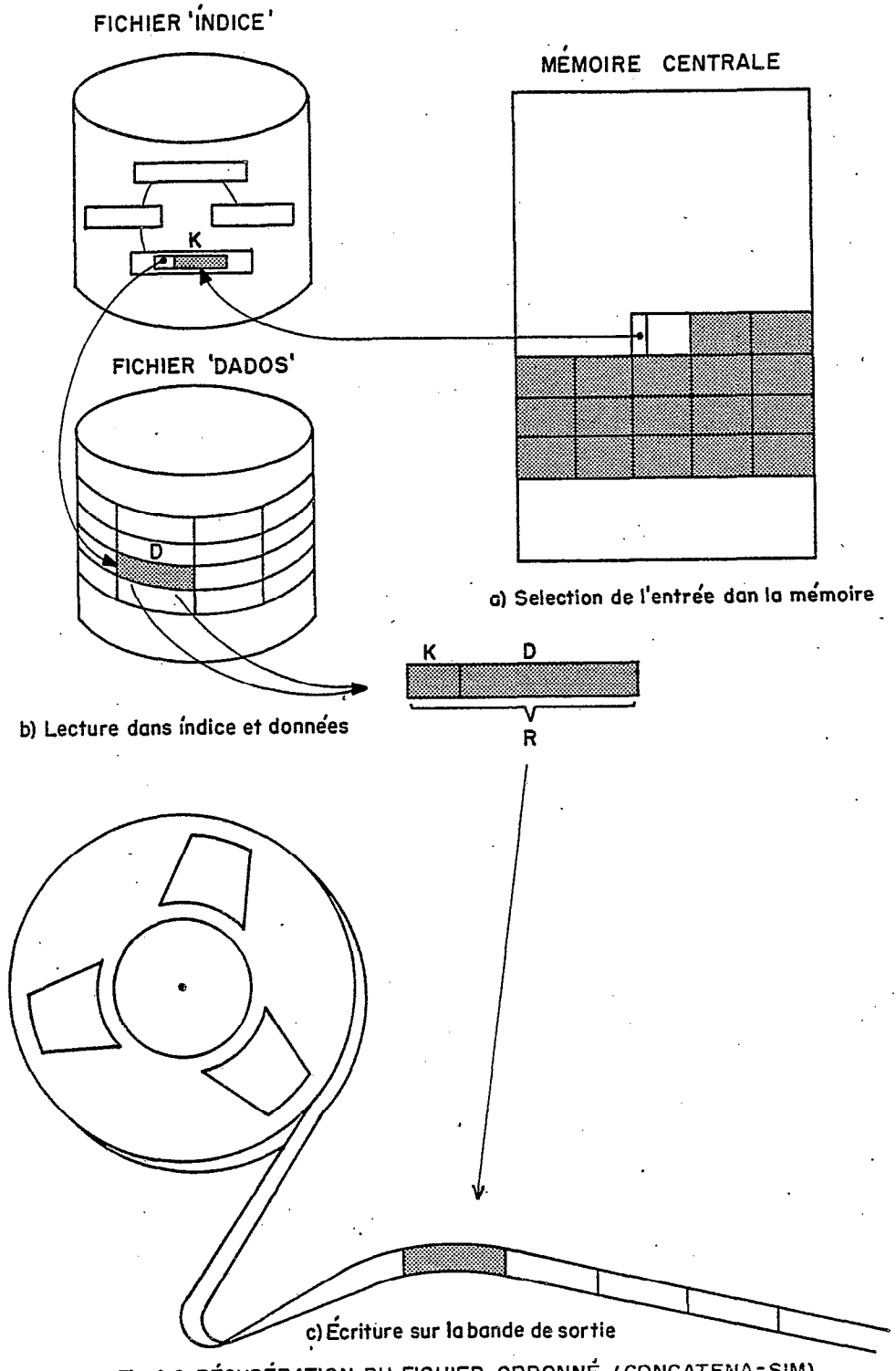


Fig.4.6 RÉCUPÉRATION DU FICHIER ORDONNÉ (CONCATENA=SIM)

interne. En particulier, la phase de formation des "runs" n'est rien d'autre que la fusion des trois phases du tri interne en une phase unique, seulement modifiée pour permettre le tri de plusieurs "runs" successifs, et permettre également l'écriture des "runs" sur disque, au lieu du fichier de sortie, selon une stratégie qui dépend de paramètres fournis par l'utilisateur.

En ce qui concerne la phase finale de fusion des "runs", la codification et la réalisation pratique de l'algorithme élaboré (voir §§2.43 & 2.46), et amélioré (voir §3.26) sont les points les plus intéressants qui seront mis en évidence dans la phase finale de ce chapitre.

4.41 La phase de formation des "runs"

La phase de formation des "runs" réalise la succession des trois étapes qui ont été présentées séparément dans le cas du tri interne:

- la lecture des données et le chargement des entrées séquentiellement dans la mémoire;

- le tri interne de ces données dans la mémoire;

- l'écriture finale du "run" formé dans la zone de travail.

La première tâche réalisée diffère de la tâche correspondante du tri interne, en ce qui concerne les procédés de début et de fin de "run". En effet, alors que la lecture des données dans le tri interne est effectuée par le biais d'un simple algorithme de récupération séquentielle du fichier d'entrée (voir §4.31), dans ce dernier cas c'est la sélection d'un "run" qui est réalisée. Aussi, le chargement du $i^{ème}$ "run" de n enregistrements débute avec l'enregistrement: $n * i + 1$ et se termine avec l'enregistrement: $n * (i + 1)$ ou avec le dernier enregistrement du fichier ($N^{ième}$), si, et seulement si on a:

$$n * i + 1 < N < n * (i + 1).$$

En conséquence, ce procédé nécessite que soit possible la récupération séquentielle sélective du fichier d'entrée. La réalisation, triviale dans le cas d'un fichier organisé séquentiellement (TIPO='FITA', 'CARTAO' ou 'DISCO'), serait un peu plus élaborée dans le cas de l'implantation éventuelle de la méthode pour un fichier séquentiel indexé. Dans le cas d'un fichier organisé selon la structure "B-tree" (TIPO='MASST'), la récupération séquentielle sélective utilise une pile (voir §4.13), dont la finalité est de maintenir un pointeur pour l'enregistrement courant et le chemin suivi pour l'atteindre. La phase de distribution des "runs" implantée maintient cette pile, de telle mode qu'à chaque instant un pointeur repère le début du prochain "run" à ordonner.

La stratégie de stockage des "runs" sur disque dépend des paramètres 'AREA' et 'RANDOMICO', fournis par l'utilisateur par le moyen des cartes de contrôle (ou assumés par défaut par le système), et que l'on trouve dans la zone 'COMMON':

- dans le cas 'AREA'=3, la routine d'allocation implantée doit écrire les "runs" alternativement dans les deux premières zones, pour minimiser les opérations de "seek" à effectuer dans la phase ultérieure de fusion;

- dans le cas 'AREA'=1 ou 2, une seule zone sera utilisée pour l'écriture initiale des "runs". Toutefois la valeur du paramètre 'RANDOMICO' détermine le type d'allocation à choisir dans cette zone: allocation orthogonale pour 'RANDOMICO'=SIM, allocation séquentielle pour 'RANDOMICO'=NAO.

Dans tous les cas, au début de chaque bloc (piste) de chaque "run", on a le nombre des entrées contenues dans ce bloc ainsi que l'adresse relative de la prochaine piste du même "run" (code 'HIGH VALUE', dans le cas du bloc final de chaque "run"). Ce système de chaînage permet de résoudre tant les problèmes d'"overflow" rencontrés dans le cas de l'allocation orthogonale (voir §3.34 et figure 3.15), que ceux du stockage dans le cas de plusieurs aires (inclusion d'un byte dans cette adresse pour déterminer quelle est la zone considérée).

Lors de la distribution des "runs", le système se charge de l'écriture dans la zone 'COMMON', d'une table séquentielle qui informe la phase de fusion des "runs", de l'adresse du premier bloc de chaque "run". Finalement le nombre des "runs" générés est également stocké dans la zone 'COMMON' (voir figure 4.3).

4.42 La phase de fusion des "runs"

Cette dernière phase doit être codifiée pour chaque type de fichier de sortie, étant entendu que l'écriture de celui-ci est réalisée au cours de cette phase. L'algorithme qui a été implanté (seulement dans le cas de sortie sur bande magnétique) correspond au modèle de fusion en plusieurs étapes, déjà présenté (voir figure 2.10). Étant donné le nombre maximal n_x de "runs" qui peuvent être intercalés simultanément, le nombre d'étapes de fusion qui devront être réalisées peut être déterminé en fonction de la valeur de Q par rapport à R fois les puissances successives de n_x, où R est le nombre de pages réelles disponibles dans la partition d'exécution (voir figure 4.7).

Un point intéressant de l'algorithme choisi, est qu'il utilise un arbre de sélection dont le nombre de noeuds internes est égal au nombre de "runs" à intercaler. Aussi, à chaque fois qu'un "run" est épuisé, le noeud externe correspondant doit être éliminé, ce qui pourra être effectué, soit par l'introduction d'une clé de valeur "LOW VALUE", soit par la propre réorganisation de l'arbre avec une nouvelle exécution du processus initial de montage de l'arbre.

Quand le nombre de "runs" qui restent à fusionner est inférieur à n_x, le "run" résultant est écrit sur le propre support de sortie.

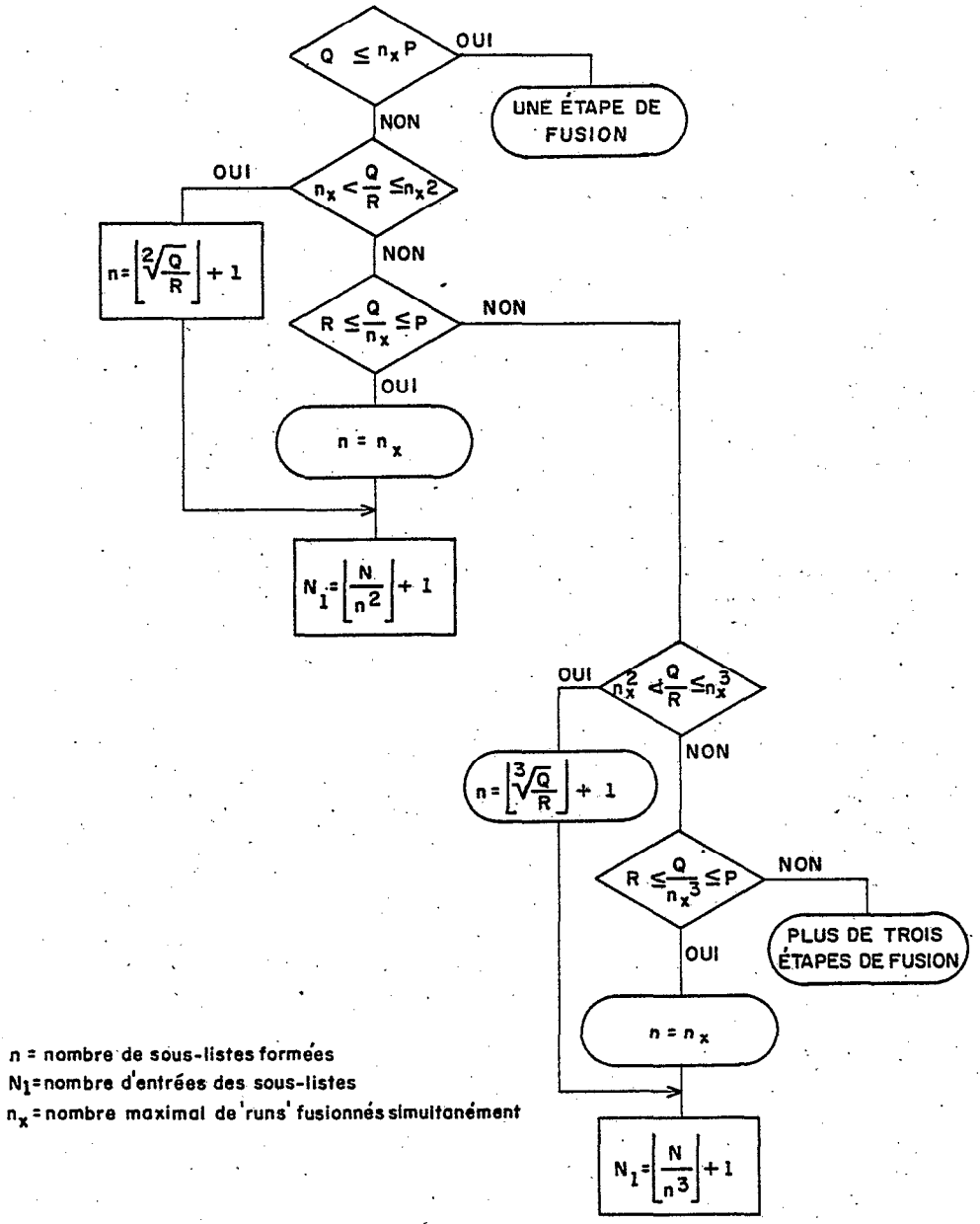


Fig. 4.7 DETERMINATION DU NOMBRE DE SOUS-LISTES ET DU NOMBRE DE CLÉS DE CHAQUE SOUS-LISTE (DEUX OU TROIS ETAPES DE FUSION)

4.5 UN TEST COMPARATIF

Pour obtenir une notion de l'efficacité de la méthode développée, un test comparatif a été réalisé avec les autres possibilités d'exécuter cette même tâche de tri, pour un utilisateur de fichier 'MASST'. La comparaison avec l'utilisation du programme fourni par le constructeur constitue un test particulièrement sévère, vu que les méthodes que celui-ci utilise doivent être particulièrement bien adaptées à l'équipement fabriqué.

Dans les résultats de ces tests, on ne doit pas seulement considérer les temps de traitement obtenus pour chaque méthode, sachant que ceux-ci sont beaucoup plus caractéristiques des opérations de lecture initiale et écriture finale, que du tri proprement dit. En effet l'utilisateur n'est bien entendu pas insensible à la tâche de codification que chaque méthode doit lui imposer, et ce dernier paramètre n'est pas quantifiable, bien que très important.

Dans un premier paragraphe, une présentation des principales caractéristiques du fichier d'entrée est réalisée.

Les trois paragraphes suivants sont consacrés à la description des solutions qui seraient adoptées respectivement par un programmeur en langage FORTRAN, un programmeur en langage COBOL, et l'utilisateur de notre système, ce dernier indépendamment du langage utilisé.

Le dernier paragraphe présente les résultats et les conclusions qui peuvent être tirées de ce premier test comparatif.

4.51 Le fichier généré pour le test

Pour ce test, on a cherché à utiliser un fichier qui ne favorise ni le système développé, ni le programme utilitaire du fabricant. Ainsi par exemple, vu que ce dernier utilise la fixation de 64K bytes de mémoire réelle, la valeur 64, assumée par défaut, a été choisie pour le paramètre 'MEMORIA' (voir §4.22). De la même façon, un enregistrement petit, de 250 positions, favorise le programme du fabricant (qui ne sépare pas la clé du reste de l'enregitrement), mais, en compensation, une clé de tri petite (7 positions) favorise le système SRT (en réalité, ce qui importe c'est le rapport de taille entre clé et enregistrement).

Finalement les dernières options ont été dictées, soit par l'intérêt pratique du test (application à un fichier 'MASST', par exemple), soit par la nécessité d'utiliser les phases réellement implantées (sortie sur bande magnétique, par exemple).

Un fichier totalement désordonné a été généré dans le format pris en exemple (voir figure 4.2), de la manière suivante:

- calcul d'une série de nombre pseudo-aléatoires de sept chiffres (champ 'numero');

- un champ 'nome' de 50 positions est obtenu par la répétition (7 fois) de la sélection, dans la table 'RIWLQEONCJ', composée aléatoirement, des 7 lettres correspondantes aux chiffres de la clé;

- les positions 57 à 250 de l'enregistrement (information satellite), sont remplies avec le caractère blanc.

La génération d'un fichier type 'MASST', construit à partir du champ 'nome', est faite avec les routines du système de S.O.COSTA (Costa, 1980), et garantit un fichier d'entrée complètement désordonné lors de sa récupération séquentielle, c'est à dire dans l'ordre lexicographique de la clé de montage (champ 'nome'), par rapport à la clé de tri choisie (champ 'numero').

Ce fichier de N enregistrements une fois généré, on obtient un fichier de N + n enregistrements, par inclusion de n nouveaux enregistrements élaborés à partir du calcul de n nombres aléatoires consécutifs, par le biais du même programme de génération (moyennant seulement la transmission de la dernière clé générée).

4.52 L'utilisateur du langage FORTRAN

Etant donné que le programme utilitaire fourni par le fabricant ne permet pas l'utilisation en entrée d'un fichier non conventionnel, il est nécessaire de récupérer séquentiellement ce fichier avant de pouvoir procéder à son tri. De plus le langage FORTRAN n'offre pas la possibilité d'utilisation de ce programme de tri comme une subroutine appelée à l'intérieur du programme principal, et en conséquence, il devient indispensable d'utiliser une seconde étape d'exécution du programme utilitaire.

Aussi, la solution qui pourra être adoptée par le programmeur FORTRAN consiste-t'elle à codifier un programme pour lire séquentiellement le fichier d'entrée, et l'écrire sur une bande magnétique. Dans une seconde étape, il s'avère nécessaire d'exécuter le programme utilitaire, en fournissant les cartes de contrôle conformes aux normes du fabricant, pour informer des valeurs des paramètres du tri.

4.53 L'utilisateur du langage COBOL

Bien évidemment, cet utilisateur aura la même nécessité d'exécuter la récupération séquentielle du fichier, pour le fournir au programme utilitaire sous forme d'une copie à accès séquentiel, apte à servir d'entrée à ce dernier.

Toutefois, la différence essentielle réside dans la ressource de 'Sort Interne' de ce dernier langage, qui autorise à ce que la phase de classification du fichier soit emboutie dans le propre programme de récupération.

Aussi la solution adoptée par le programmeur COBOL pourra-t'elle consister à codifier un seul programme pour appeler la procédure "SORT", en utilisant le module de récupération séquentielle pour la sélection du prochain enregistrement à se présenter. On économise de cette manière les opérations d'écriture et de lecture intermédiaires, qui étaient intervenues dans le cas précédent.

4.54 L'utilisateur du système développée

Il suffit dans ce cas de codifier les cinq cartes de contrôle, telles qu'elles sont présentées en exemple sur la figure 4.2, étant entendu que le système a essentiellement été développé pour accepter ce type d'entrée.

4.55 Les résultats obtenus

Les trois méthodes précédemment décrites ont été implantées pour être testées avec le fichier généré comme il a été indiqué au paragraphe 4.51. Les conditions normales d'exécution ont été respectées, le programme de tri étant exécuté dans la partition BG, alors que les partitions F1 et F2 restaient actives.

Le tableau 4.2 montre les temps obtenus pour divers nombres d'enregistrements du fichier d'entrée.

Les tests dans le cas du FORTRAN n'ont pas été poursuivis pour les fichiers les plus importants, en raison du temps consommé. Les tests pour 2 000, 5 000 et 7 500 enregistrements ont été réalisés dans la partition de basse priorité, alors que ceux pour 354, 1 326 et 2 960 enregistrements ont été réalisés dans la partition de plus forte priorité.

On constate à l'examen de ces résultats que:

- l'utilisation de ce système est toujours avantageuse pour le programmeur FORTRAN, en raison de la copie intermédiaire nécessaire sur un fichier magnétique;

- en ce qui concerne la comparaison avec la méthode accessible à l'utilisateur du langage COBOL, le système 'gagne' pour les petits fichiers, mais perd rapidement dès que la taille du fichier dépasse 500K.

Lors de la réalisation de ces tests, on a pu percevoir que la dernière phase de récupération des enregistrements dans le cas du système développé est très lente, vu que la lecture est faite enregistrement par enregistrement, ce qui pratiquement implique un "seek" à chaque fois, du moins dès que le fichier dépasse 500 enregistrements (c'est à dire un cylindre dans l'aire de 'DADOS').

4.6 CONCLUSIONS

Lors de la réalisation pratique de ce travail, nous nous sommes trouvé confrontés à de très nombreux problèmes de codification. L'implantation des algorithmes présentés n'est pas la tâche la plus difficile du programmeur, l'utilisation au mieux des registres de la machine rendant cette réalisation rapidement très efficace. L'élaboration des modules d'entrée et de sortie est en fait un problème autrement plus difficile, en raison de ce que l'on a cherché à développer un système très souple. Ainsi les diverses options proposées à l'utilisateur en ce qui concerne les supports, blocages et

nombre d'enregistrements	méthode de tri		
	COBOL	SISTEMA	FORTRAN
354 (86,4K)	1'28	1'01	1'56
1326 (323,7K)	3'46	3'12	5'36
2000 (488,3K)	4'31	6'09	7'30
2960 (722,7K)	4'16	4'41	9'58
5000 (1221 K)	11'50	14'22	>20'
7500 (1831 K)	14'18	26'40	—

Tableau 4.2 Comparaisons de temps entre les 3 méthodes

formats, tant d'entrée que de sortie, exigent l'utilisation de techniques qui permettent la modification du programme selon les paramètres fournis par l'utilisateur, en temps d'exécution (chargement de modules de traitement différents, altération des processus de I/O,...).

Le résultat, relativement peu probant, des tests réalisés, c'est à dire le fait que le programme élaboré n'ait pas été plus rapide (du moins pour le tri des grands fichiers), que la méthode de tri qui utilise le programme fourni par le fabricant comme routine appelée par un programme codifié par l'utilisateur dans un langage de haut niveau, donne sans doute une des leçons les plus importantes de ce travail. En effet, dans la mesure où l'on peut attribuer la pénalisation en temps de la méthode développée à la récupération du fichier après le tri, cette étude conduit à reconsidérer la propre idée initiale du tri indirect, pourtant très séduisante en théorie. Il apparaît en effet qu'il serait plus efficace de trier le fichier en stockant la totalité des enregistrements dans la mémoire interne (dans la limite de l'espace disponible offert par les pages réelles de la partition), ce qui économiserait un second accès dans la zone 'DADOS' au moment de la dernière phase, très consommatrice de temps, surtout quand l'espace occupé par ce dernier fichier, dépasse quelques cylindres. La réduction considérable de la taille des "runs" formés qui résulterait de ce changement, est facilement compensée par le gain de temps inhérent à la récupération séquentielle, au lieu de la récupération aléatoire, des enregistrements dans la phase finale.

Cependant, on peut remarquer que cet inconvénient disparaît lorsque l'on peut s'affranchir de la dernière phase de récupération du fichier, c'est à dire dans le cas de branchement direct sur le programme utilisateur, à l'issue du tri. Dans ces conditions, l'emploi du programme devient très attrayante pour l'utilisateur.

De nombreux points ont été mis en évidence au cours de cette étude et mériteraient d'être étudiés plus en détail, à l'occasion de travaux postérieurs:

- détermination, avec une plus grande précision, des intervalles de plus grande efficacité (voir §3.43), et étude de l'évolution de ceux-ci selon différentes configurations de mémoire virtuelle;

- examen des conditions de viabilité relative du tri direct et du tri indirect, en considérant la taille de l'enregistrement comme variable principale;

- étude des possibilités d'extension des ressources offertes par le système, et en particulier du développement des phases nécessaires au tri de fichiers séquentiels indexés, et aussi de l'option 'ROTINA'.

5. EXTENSION A D'AUTRES SITUATIONS

Afin de donner plus d'ampleur à ce travail, dont l'objectif jusqu'alors restait limité à la recherche de l'optimisation d'une opération déterminée, le tri, dans un contexte également fixé, la mémoire virtuelle, il a été jugé opportun d'entreprendre un essai de généralisation d'après les résultats obtenus lors de cette étude.

Deux voies distinctes de généralisation s'offrent immédiatement à l'esprit à partir du sujet traité. On peut les exprimer sous la forme suivante:

- l'opération restant fixée, que se passe-t'il si l'on fait varier le contexte? Autrement dit, les conclusions tirées sur l'algorithme de tri restent-elles valables dans le cas d'autres techniques de gestion de mémoire;

- le contexte étant fixé cette fois, que se passe-t'il lorsque l'on fait varier l'opération? Autrement dit, dans le contexte de mémoire virtuelle qui a été décrit et étudié, comment peut-on envisager l'adaptation de systèmes tels que ceux de gestion de bases de données, par exemple.

Le présent chapitre propose une réflexion pour chacune des deux voies explorées.

5.1 LE TRI ET LA GESTION DE MEMOIRE

Dans ce premier sous-chapitre, on se propose de réfléchir sur le choix optimal d'une méthode de tri d'un grand fichier, en fonction de divers modes possibles de gestion de mémoire.

Dans un premier temps, une brève description des divers schémas existants de gestion de mémoire et de leurs caractéristiques propres s'est avérée nécessaire. Bien que l'évolution actuellement considérable des micro-ordinateurs remette au goût du jour des schémas simples de gestion de mémoire (pour une raison évidente de coût), on a insisté particulièrement sur les techniques de pagination avec demande de pages, universellement adoptées à présent, en explorant les nombreux algorithmes de substitution de pages dans la mémoire.

L'étude du comportement des algorithmes de tri dans chacun des cas fait alors l'objet du dernier paragraphe et permet de conclure par l'établissement d'un tableau présentant le choix à effectuer pour le tri en fonction de la technique de gestion de mémoire considérée.

5.11 Allocation séquentielle simple

C'est le schéma le plus simple d'allocation de mémoire, pour lequel l'ensemble de la mémoire disponible est attribuée à un "job" unique. Comme le montre la figure 5.1, la mémoire principale est divisée en trois régions contiguës. Une partie de la mémoire est allouée en permanence au système d'exploitation, le code du programme utilisateur est chargé dans le restant. Son exécution nécessite une

zone de travail: l'ensemble code et zone de données n'utilisant généralement qu'une partie de la mémoire non allouée au système, il reste une part non utilisée.

Une telle méthode de gestion de mémoire est utilisée actuellement pour les micro-ordinateurs dits mono-tâches; historiquement on le rencontre sur le système IBM OS/360 Primary Control Program.

Ce mode d'allocation, en raison de sa simplicité, a pour avantage de n'utiliser que très peu de mémoire pour le système opérationnel: en contrepartie il n'offre pas de possibilité de multiprogrammation.

5.12 Allocation de mémoire partitionnée

C'est une des techniques les plus simples qui permettent la multiprogrammation. Elle consiste à diviser la mémoire non occupée par le système d'exploitation en plusieurs parties (régions ou partitions), chacune d'entre elles étant destinée à recevoir l'espace adressable d'un "job" (voir figure 5.2).

Lorsque les tailles des partitions sont fixées au moment de l'initialisation du système, la division est dite statique. Elle est dynamique lorsque la taille de chaque région est précisée au moment de recevoir le "job" qui doit y être exécuté.

Dans ce dernier cas, il y a perte d'espace due à la surestimation que l'utilisateur fait de la place nécessaire à l'exécution de son programme. Dans le premier cas, s'ajoute à cette première perte de place celle qui résulte de l'obligation de choisir pour un "job" une partition au moins aussi grande. De toutes les façons il y a création de trous (fragmentation), au fur et à mesure que l'on remplace un "job" par un autre, de taille au maximum égale.

La méthode de gestion de mémoire par allocation partitionnée a été utilisée dans le système IBM OS/360 MFT (Multiprogramming with a Fixed number of Tasks) sous la forme statique et IBM OS/360 VMS (Virtual Memory System) sous la forme dynamique.

Ce mode d'allocation permet donc la multiprogrammation en ne demandant que peu de mémoire supplémentaire pour le système d'exécution; il présente toutefois l'inconvénient d'être soumis à l'effet de fragmentation de l'espace mémoire non occupé.

5.13 Allocation de mémoire avec réallocation

Ce mode de gestion de mémoire apporte une solution au problème de la fragmentation en compactant périodiquement toutes les zones libres en une région unique, grâce à la translation de toutes les partitions occupées (voir figure 5.3).

Une telle technique a été mise en oeuvre par plusieurs constructeurs: Burroughs pour les séries 5500 et 6700, Digital pour son PDP-10, Univac sur le 1108, etc....

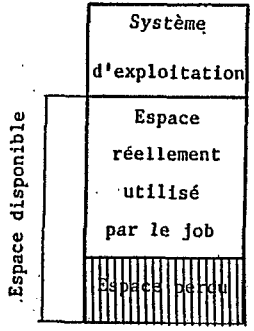


Fig. 5.1 Allocation séquentielle simple

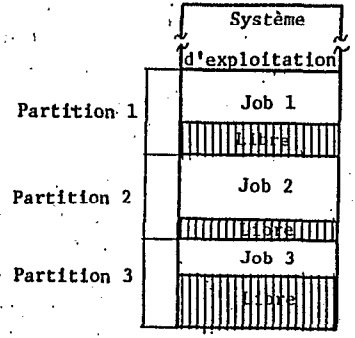


Fig. 5.2 Allocation de mémoire partitionnée

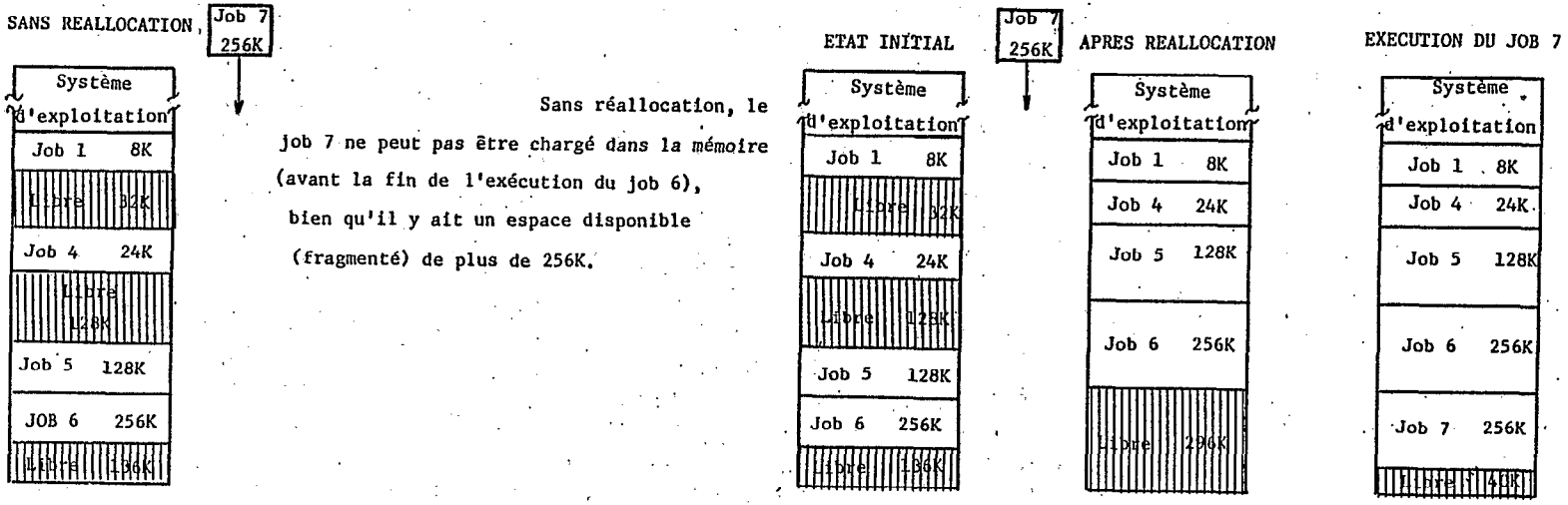


Fig. 5.3 Allocation de mémoire avec réallocation

Le degré de multiprogrammation ainsi obtenu est meilleur puisque la fragmentation a été éliminée: cependant, le temps passé à la compactation peut devenir très important, ce qui diminue globalement les performances au point de vue temps d'un système, et le système opérationnel devient également plus volumineux.

5.14 Allocation de mémoire par pages

C'est une autre solution au problème de la fragmentation qui présente l'avantage de ne pas faire appel au déplacement physique des partitions dans la mémoire à chaque compactation. Cette technique consiste à diviser l'espace adressable de chaque "job" en parcelles égales, lesquelles sont appelées "pages" (voir §3.12), et à diviser la mémoire en parties de même taille, appelées "blocs", de telle façon que n'importe quelle page pourra être localisée en n'importe quel bloc disponible. La maintenance de tables de copies de pages (voir figure 5.4) permet que les pages d'un programme soient logiquement contiguës, alors que les blocs qui les contiennent ne le sont pas nécessairement.

Le choix de la dimension des pages doit être judicieux: elles ne doivent être ni trop grandes pour éviter le phénomène de fragmentation interne, ni trop petites pour éviter la croissance démesurée des tables de copies de pages. Les constructeurs qui utilisent cette technique (Control Data sur la série 3300, par exemple) choisissent des dimensions de pages de 1, 2 ou 4K.

L'élimination du problème de la réallocation des partitions est le principal avantage de ce mode de gestion de mémoire par rapport au précédent. En contrepartie, le système d'exploitation prend de plus en plus d'ampleur et en même temps une partie de la mémoire principale est utilisée pour stocker les tables de copies de pages. Ce dernier inconvénient peut être toutefois éliminé grâce à l'utilisation de mémoires dites associatives, encore très récentes et dont le coût reste prohibitif.

5.15 Allocation de mémoire par pages, avec demande de pages

C'est la méthode qui a été considérée dans cette étude, et qui a été précédemment décrite (voir §3.12 et figures 3.2, 3.3 et 3.4). Le choix provient de ce que cette technique est très largement utilisée dans les systèmes actuels: OS/VS1, OS/VS2, DOS/VSE et OS/VM370 sur ordinateurs IBM 370, mais aussi MULTICS d'Honeywell, VMOS d'Univac, etc....

Toutefois les systèmes peuvent différer par les critères à partir desquels est sélectionnée la page qui se verra éliminée lors d'une demande de nouvelle page. Plusieurs algorithmes ont été présentés, parmi lesquels on peut citer:

- substitution au hasard ("Random replacement"), qui peut fréquemment retirer des pages utiles;

- algorithme FIFO ("First In - First Out"), simple à réaliser mais pas nécessairement efficace dans la mesure où la page qui est

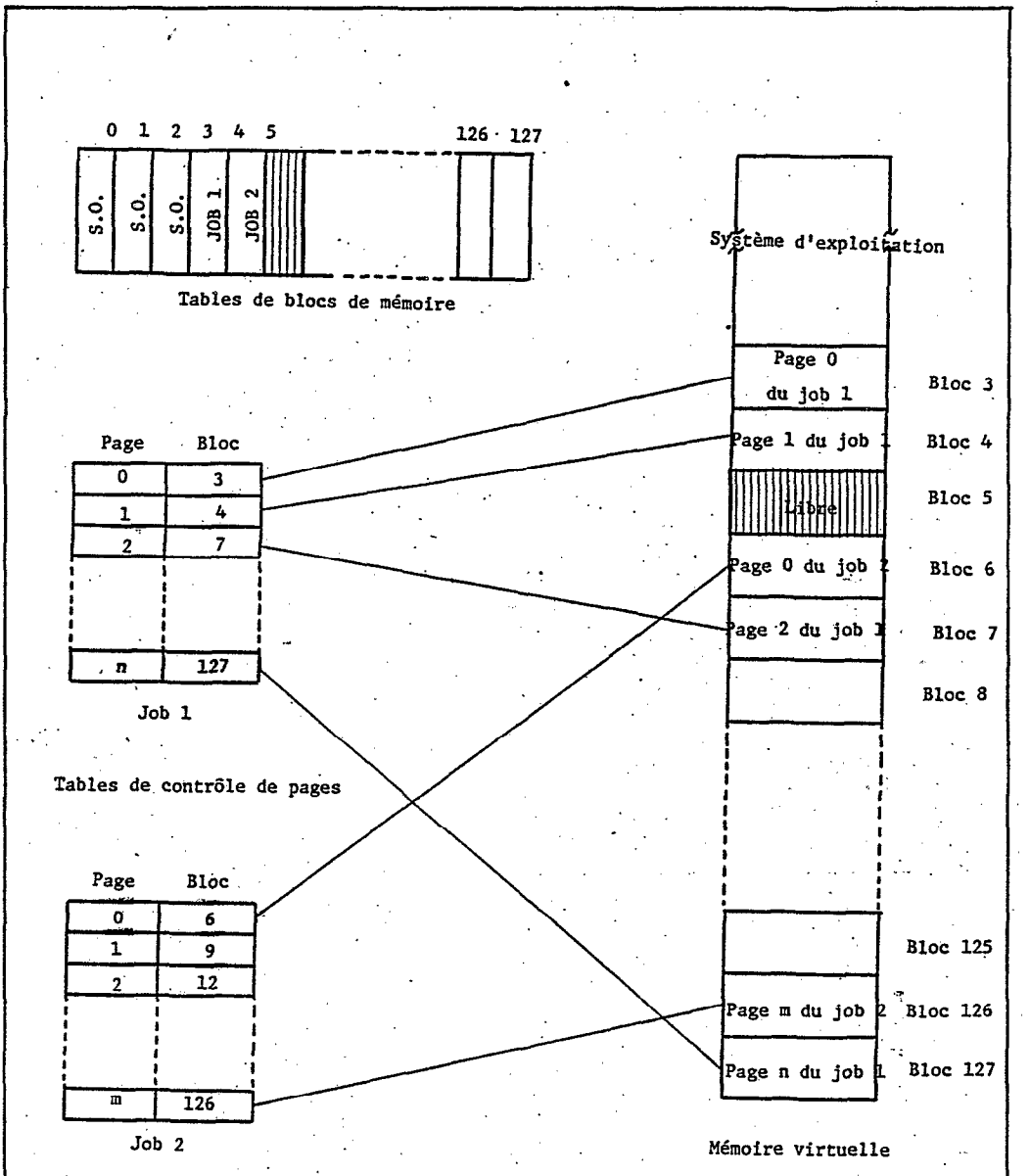


Fig. 5.4 Allocation de mémoire par pages

présente dans la mémoire depuis plus longtemps sera éliminée, alors qu'elle peut être à nouveau utile;

- algorithme LRU ("Least Recently Used"), c'est celui qui a été exposé au §3.12: plus complexe à réaliser il est aussi plus efficace;

- algorithme LFU ("Least Frequently Used"), c'est la page la moins fréquemment utilisée qui est remplacée: cet algorithme s'apparente au précédent;

- algorithme WS ("Working Set control"), dont la philosophie consiste à maintenir dans la mémoire les pages qui ont été nécessaires à l'exécution du programme durant les dernières T millisecondes (T choisi). Toutes les autres pages deviennent disponibles, ce qui permet une allocation de mémoire par partition variable;

- algorithme PFF ("Page Fault Frequency"), pour lequel l'idée de base est de contrôler le taux de demande de pages durant l'exécution de chaque "job" et de le réguler autour d'une valeur fixe K, par adjonction ou libération de blocs de mémoire entre partitions différentes;

- algorithme CPFR ("Controlled Page Fault Rate"), dont la philosophie est semblable au précédent, en ce qu'on y libère des blocs de mémoire lorsqu'un temps limite est atteint, même lorsqu'il n'y a pas eu une interruption par défaut de page.

La mémoire virtuelle obtenue à partir d'un système d'allocation de mémoire avec demande de pages possède de nombreux avantages. Ceux-ci ont fait adopter cette technique dans la plupart des systèmes actuels. Toutefois un inconvénient majeur consiste dans le risque de rencontrer des situations telles qu'un important pourcentage de temps du processeur est consacré aux échanges de pages ("Trashing"). Ces situations sont difficilement contrôlables.

5.16 Segmentation

La technique de segmentation, utilisée dans cette étude pour le programme de tri (voir §3.22 et figure 3.7), qui consiste à diviser le "job" à exécuter en segments (ou phases) chargés séparément dans la mémoire, peut être considérée comme une technique de gestion de mémoire (Madnick et al., 1974). Elle a l'inconvénient de n'être pas transparente pour l'utilisateur. Qui plus est, la taille des segments est limitée à celle de la mémoire centrale. Ce dernier inconvénient peut néanmoins être contourné avec l'utilisation simultanée de l'allocation avec demande de pages, si bien que ce mode de gestion est actuellement utilisé dans les grands systèmes.

5.17 "Swapping"

Cette méthode qui consiste à vider sur une mémoire auxiliaire la totalité d'un "job", après une courte période d'exécution, afin d'autoriser le chargement et l'exécution, pendant un même laps de temps, d'un autre programme, est utilisée par certains systèmes de

"Time-sharing" (TSO sur OS/360, par exemple). Le temps perdu aux transferts entre la mémoire centrale et la mémoire auxiliaire n'est pas trop significatif si les programmes en cours d'exécution ne sont pas trop importants (cas des "jobs" d'utilisation de l'éditeur de texte, par exemple). On peut remarquer de plus que, si le code est réentrant, il n'est pas nécessaire de le recopier plusieurs fois sur une mémoire auxiliaire.

5.18 Implantation du programme de tri

Pour chacun des modes de gestion de mémoire qui ont été passés en revue, et afin de réaliser une comparaison sur des bases concrètes et homogènes, on supposera que l'on dispose d'un ordinateur qui répond à la même configuration que celle qui a été décrite au cours de cette étude (voir §3.11), c'est à dire avec 256K de mémoire réelle. On peut alors imaginer que l'on implante dans cette configuration les différents modes de gestion de mémoire précédemment décrits, et essayer de déterminer quelle est la meilleure méthode de tri pour chaque cas, et ce en fonction de la taille du fichier à ordonner:

- allocation séquentielle simple: dans ce schéma, le système d'exploitation sera très simple, n'occupant par exemple que 32K. Les 224K restants pourront alors être utilisés pour stocker le programme, que l'on suppose non segmenté et qui pourrait atteindre 60K, compte tenu de "buffers" importants pour minimiser les opérations d'entrée-sorties. Dans ces conditions, on pourra utiliser tout algorithme de la classe NlogN, et en particulier celui du "Quicksort", pour trier un fichier de moins de 160K (tri interne);

- allocation de mémoire partitionnée: le schéma est analogue, avec une limite de mémoire plus faible, compte tenu de ce qu'il y a possibilité d'exécution concommitante dans d'autres partitions. Il faut noter que dans le cas de la division dynamique il sera quelquefois possible d'adapter la taille de la partition au tri à exécuter, alors que pour la division statique il y a le plus souvent mauvaise utilisation de la mémoire;

- allocation de mémoire avec réallocation: là encore, le choix de l'algorithme de tri interne n'a pas d'influence sur les performances globales du système. On peut seulement espérer avec cette technique une meilleure gestion de la mémoire grâce à l'élimination du phénomène de fragmentation, avec en contrepartie une perte de temps due aux compactations périodiques;

- allocation de mémoire par pages: avec cette technique, on peut espérer obtenir un meilleur ajustement (à une page près) de la dimension de la partition à l'espace adressable requis pour effectuer le tri, tout en éliminant l'inconvénient d'avoir à réaliser des réallocations à chaque chargement de nouveau job. Les performances globales du système restent néanmoins indépendantes du choix de l'algorithme de tri interne choisi;

- allocation de mémoire avec demande de pages: c'est avec ce mode de gestion de mémoire que l'on doit choisir judicieusement l'algorithme de tri, sous peine de voir les performances du système se

dégrader par une croissance excessive du phénomène de pagination. Pour l'algorithme LRU utilisé dans ce travail il en a été fait une étude exhaustive (voir chapitre 3). On étudie dans le prochain paragraphe si les conclusions qui en ont été tirées se maintiennent lors de l'utilisation des autres philosophies de substitution de pages;

- allocation de mémoire avec segmentation: ce mode de gestion de mémoire permet de fragmenter le programme de tri en quatre parties (voir §3.22 et figure 3.7), qui sont chargées successivement dans la mémoire. Qui plus est, les phases nécessaires au tri externe ne seront chargées que si le tri doit être réalisé de cette façon, en lieu et place des phases de tri interne. Dans les conditions qui ont été assignées pour cette étude, on peut évaluer à environ 40K l'économie d'espace correspondante, soit les deux tiers de l'espace adressable du programme. Bien entendu, cette économie d'espace se traduira par la possibilité de trier plus de données en une seule fois et par conséquent aboutit à une économie de temps pour la totalité de l'opération;

- allocation de mémoire par "swapping": avec cette méthode, on retrouve l'indépendance de l'algorithme de tri par rapport au mode de gestion de mémoire, du moins lorsque c'est le "job" dans sa totalité qui est déplacé.

5.19 Influence de l'algorithme de substitution de pages

La recherche de l'efficacité du tri dans le contexte de mémoire virtuelle, qui fait l'objet de cette étude, a été réalisée en utilisant l'algorithme LRU de substitution de pages. En effet, lorsqu'il y a réquisition d'un bloc de mémoire réelle pour y loger une page nécessaire à l'exécution d'un programme, c'est celui qui contient la page qui est restée le plus longtemps inutilisée qui est choisi. Ceci conduit à des règles de programmation (voir §3.21), et en particulier à celle de référenciation linéaire, qui a conduit au choix de l'algorithme "Quicksort".

Il est toutefois intéressant d'examiner les autres algorithmes de remplacement de pages pour déterminer la validité de cette conclusion:

- substitution au hasard: dans ce cas, quel que soit l'algorithme de tri choisi, toute page présente dans la mémoire réelle est susceptible d'être éliminée, y compris d'ailleurs celles qui correspondent à la partie de programme, si l'on a pas pris la précaution de fixer ces dernières par une instruction spéciale. En conséquence, aucun algorithme n'est meilleur qu'un autre dans ce contexte;

- algorithme FIFO: avec ce premier algorithme, se trouve éliminée la page qui est depuis le plus longtemps dans la mémoire réelle, indépendamment de l'utilisation qui en est faite. On doit alors considérer deux cas, selon que l'on a pu fixer les pages du programme dans la mémoire par un artifice de programmation, auquel cas ce sont seulement les pages correspondantes aux données qui seront affectées par les substitutions: on est alors ramené à un cas analogue à ce qui a été étudié pour l'algorithme LRU, à savoir que la

référenciation séquentielle est favorisée. Si l'on a pas pu fixer les pages du programme dans la mémoire réelle, on rencontrera un ensemble d'échanges de pages 'parasites' qui risque de masquer les conclusions précédentes, qui pourtant seraient toujours valables;

- algorithme LRU: la philosophie de cet algorithme est que si une page a été référenciée récemment dans la mémoire, elle le sera probablement de nouveau un peu plus tard, alors que celles qui ne l'ont pas été dans le même temps ne le seront probablement pas non plus dans un futur immédiat. Dans ces conditions on évite pratiquement les échanges de pages parasites qui pourraient intervenir dans la zone du programme, tout en favorisant les algorithmes de type "Quicksort" qui utilisent essentiellement la référenciation linéaire;

- algorithme LFU: avec cette technique les échanges de pages parasites sont encore moins probables. De la même manière que précédemment, les algorithmes de tri favorisés sont ceux pour lesquels les références successives à une page déterminée sont groupés dans le temps;

- algorithme WS: l'idée de base de cet algorithme est assez proche de celle des deux précédents et favorise également le "Quicksort" aux dépens de ses concurrents. Qui plus est, la possibilité que ce dernier algorithme offre de faire varier la taille des partitions en cours d'exécution est un facteur d'amélioration des performances globales du système en multiprogrammation;

- algorithme PFF: cet algorithme permet également une allocation dynamique de la mémoire. Lorsque le taux de demandes de pages atteint une valeur telle qu'il n'est plus possible d'en allouer de nouvelles sans procéder à une élimination, il faut alors faire appel à l'un des algorithmes précédents, l'algorithme WS par exemple. En ce qui concerne le programme de tri les conclusions sont donc les mêmes que précédemment;

- algorithme CPFR: pour les mêmes raisons que ci-dessus, on aboutit aux mêmes conclusions.

En conclusion, on peut souligner que si pour des modes de gestion de mémoire qui ne font pas intervenir une hiérarchie de stockage en deux niveaux le choix de l'algorithme de tri interne est pratiquement indifférent (voir tableau 5.1), il n'en va pas de même dans le contexte de mémoire virtuelle, où l'on doit choisir un algorithme qui respecte au mieux le principe de référenciation linéaire, "Quicksort", par exemple, au détriment d'algorithmes qui accèdent aléatoirement aux données, tel que "Heapsort".

Enfin il convient de rappeler ici que l'utilisation des méthodes de tri indirect, où seules les clés et une adresse par enregistrement sont stockées et manipulées dans la mémoire principale (voir §2.13), contribuent à une gestion efficace de la mémoire. Toutefois, c'est au moment de la récupération du fichier complet que l'on risque d'avoir à réaliser une séquence d'accès aléatoires (voir §4.6).

Les tendances actuelles de la gestion de mémoire par hiérarchie entre plusieurs niveaux, qui permettent à l'utilisateur d'obtenir une mémoire à un coût légèrement supérieur à celui du plus bas niveau (stockage de masse), tout en profitant d'un temps d'accès légèrement supérieur à celui du plus haut niveau ("cache-memory"), rendent d'autant plus nécessaire l'adoption des règles de programmation qui ont été énoncées (voir §3.21).

5.2 BASES DE DONNEES ET MEMOIRE VIRTUELLE

On a vu, dans le cas particulier de la recherche de l'efficacité pour une méthode de tri, que l'utilisation de la mémoire virtuelle avait une incidence extrêmement importante sur le choix de l'algorithme. On peut alors se demander comment d'autres programmes importants (par leur fréquence d'utilisation) sont adaptés pour leur utilisation dans ce contexte. C'est l'objet de ce second chapitre.

Le choix des systèmes de gestion de bases de données se justifie non seulement par l'importance que ceux-ci revêtent dans nombre d'applications du traitement informatique des données, mais aussi par la variété des opérations et des constituants qu'ils font intervenir.

Dans un premier temps, après avoir brièvement définit ce que l'on entend par système de gestion de bases de données, on passera en revue succintement les différents éléments d'un système de gestion de bases de données, leurs fonctions et interactions, pour finir par l'étude des conditions d'adaptation de ce système à la gestion de mémoire par pages, avec demande de pages.

5.21 Définition d'une base de données

On peut définir une base de données sur un sujet comme étant un ensemble de renseignements sur ce sujet, stockés sur un support physique dans un espace unique. Une base de données se conçoit à travers un système d'informations qui répond aux objectifs suivants:

- rendre les données contenues facilement accessibles à plusieurs applications;

- éliminer au maximum toute redondance inutile entre ces informations;

- rechercher l'exhaustivité sur le sujet considéré dans les conditions prédéterminées.

Une base de données se différencie d'un système traditionnel de gestion de données par fichiers indépendants, en ce qu'elle contient en plus la description des informations (et de leurs relations) qu'elle renferme. Pour rendre possible les divers traitements, il est nécessaire que les informations contenues dans la base soient logiquement structurées selon un modèle mathématique qui représente au mieux la réalité observée.

Du point de vue théorique, il existe essentiellement trois approches différentes pour représenter les structures de données dans une base:

- le modèle hiérarchique qui se contente de représenter des relations entre les entités du type père-fils (hiérarchie);

- le modèle "network" apparait comme une généralisation du modèle hiérarchique, en ce qu'il autorise la représentation des relations plus complexes (un fils peut alors avoir plusieurs pères);

- le modèle relationnel, plus récent, qui permet de voir les fichiers comme des tableaux où sont représentées de la même manière les relations d'attributs et les associations d'entités. Chaque colonne d'un tableau représente un attribut (une variable), tandis que chaque ligne représente un individu (une observation).

Pour créer et faire fonctionner une base de données il est nécessaire d'utiliser un logiciel appelé "Système de Gestion de Bases de Données" (SGBD).

5.22 Rôle d'un Système de Gestion de Bases de Données

Un SGBD sert à gérer des collections de données. Il comporte donc des fonctions qui permettent de décrire, de manipuler, de mettre à disposition et de traiter des données. Cet ensemble de fonctions réunies dans un seul logiciel apporte des solutions à un certain nombre de problèmes qui ne manquent pas de se poser dans un système de gestion classique. Aussi, avant de décrire les fonctions réalisées par un SGBD, il est intéressant d'examiner les diverses catégories de problèmes posés par la gestion classique des données:

- interdépendance des fichiers et des programmes, puisque la description des fichiers est contenue dans les programmes;

- difficultés d'extraction des données, en particulier lorsqu'il faut faire intervenir des critères de sélection multiples dont la formulation peut varier en fonction des traitements désirés;

- redondance des données, car les fichiers traditionnels sont le plus souvent conçus pour contenir un ensemble d'informations correspondant à une exploitation particulière;

- mauvaises performances des accès, dont la gestion reste à la charge des utilisateurs;

- difficulté de partage des données, qui suppose un effort de coordination et d'information, tant pour la consultation que pour la mise à jour.

Le SGBD joue ce rôle en s'appuyant sur un certain nombre d'opérations fondamentales, complétées ou non selon les systèmes par l'adjonction de fonctions complémentaires.

5.23 Fonctions réalisées par le SGBD

Quel que soit l'emploi et la structure d'une Base de Données, le système de gestion doit réaliser les opérations fondamentales suivantes:

- création physique initiale de la base;

- inclusion, exclusion et substitution d'enregistrements logiques dans la base;

- correction partielle d'un enregistrement logique, c'est à dire modification de la valeur d'un champ déterminé d'un enregistrement logique choisi à priori;

- sélection, c'est à dire extraction d'un sous-ensemble d'informations répondant à un certain nombre de critères formulés par l'utilisateur.

Parmi ces quatre opérations fondamentales, la dernière est la plus caractéristique des SGBD, car elle oblige généralement à faire appel à des organisations de fichiers particulières. De plus, on peut remarquer que pour toutes ces opérations il est indispensable de rechercher tout d'abord les enregistrements qui seront affectés par l'opération considérée.

D'autres opérations peuvent également être intégrées dans un SGBD, soit parce qu'elles sont requises pour améliorer les performances d'accès: c'est le cas des réorganisations à la suite de suppressions d'enregistrements dans certains logiciels; soit encore pour assurer des fonctions non réalisées par les opérations fondamentales.

C'est le cas des fonctions d'intégrité et de sécurité, qui sont des caractéristiques essentielles de ces systèmes, et sont généralement confiées à des programmes utilitaires; c'est aussi celui de l'accessibilité aux informations pour les utilisateurs, qui pourront se servir d'un langage conversationnel ou d'un éditeur d'états pour utiliser ces données sans avoir à en connaître l'organisation physique; c'est enfin celui du partage des données, assuré par un moniteur qui autorise l'utilisation concurrente d'une même base par plusieurs utilisateurs.

Enfin, des fonctions spéciales peuvent également être implantées dans un SGBD. Citons, par exemple, celles réalisant les fonctions suivantes:

- phonétisation, grâce à laquelle il est possible de répondre à des requêtes dans lesquelles l'orthographe exacte d'un mot-clé n'est pas connue;

- encryptage, qui autorise l'encodage des données contenues dans la base, de façon à ce qu'elles soient illisibles pour un utilisateur non autorisé;

- fonction d'audit, qui permet de suivre toutes les transactions réalisées sur la base dans un environnement multi-usage.

5.24 Structure physique d'une base de données

Ce qui distingue une gestion de type "base de données" d'une gestion classique, au point de vue de la structure physique, c'est généralement l'utilisation de techniques d'organisations de fichiers basées sur plusieurs clés d'accès, dites secondaires. La figure 5.5 situe les différentes organisations de fichiers, dont les plus connues sont évidemment celles basées sur une seule clé, dite clé primaire.

En effet, quel que soit le modèle selon lequel une base de données est structurée, celle-ci peut être perçue comme un ensemble de fichiers logiques reliés entre eux par des couplages, réalisés sur les champs communs à plusieurs fichiers, parfois appelés descripteurs. Le couplage entre deux fichiers consiste, pour chaque enregistrement de l'un d'eux, à connaître la liste des enregistrements de l'autre fichier qui possèdent la même valeur pour les descripteurs de couplage. Ces couplages entre fichiers sont traditionnellement réalisés par la création de fichiers structurés en listes inverses, multilistes ou leurs variantes, dont la propriété fondamentale est de permettre d'accéder à tous les enregistrements qui possèdent une caractéristique déterminée (critère exprimé en fonction de certaines valeurs de descripteurs), sans avoir à examiner l'ensemble des enregistrements.

Plus les données contenues dans la base répondent à une structure complexe, plus il y aura de couplages à réaliser entre les fichiers, et plus ces organisations de fichiers prennent de l'importance, surtout dans un contexte de mémoire virtuelle, où le coût non négligeable d'un échange de pages impose un soin particulier au choix de l'organisation.

5.25 Organisations de fichiers basées sur plusieurs clés d'accès

Pour présenter les diverses possibilités de représentation d'un fichier à plusieurs clés d'accès, on peut s'appuyer sur l'exemple d'un fichier d'étudiants, comprenant les informations suivantes: matricule, nom, adresse de résidence, matière étudiée. On dispose alors des possibilités suivantes:

- organisation multiliste: un index est créé pour chaque descripteur (par exemple la ville de résidence de l'étudiant). Chaque entrée dans cet index est constituée par la valeur de l'attribut considéré, un pointeur vers le premier enregistrement du fichier primaire qui vérifie cette valeur de l'attribut, et le nombre total d'enregistrements de la liste des enregistrements correspondants (voir figure 5.6). Dans le fichier principal il existe une zone qui contient un pointeur réservé à cette liste (chaque enregistrement indique le prochain de la liste). Par ailleurs, le champ correspondant à l'attribut indexé devient facultatif dans le fichier primaire;

- organisation par listes inverses: dans ce cas également on crée un index pour chaque descripteur. Chaque entrée de cet index est constituée par la valeur de l'attribut considéré, du nombre d'enregistrements du fichier primaire qui vérifient cette valeur de l'attribut et de la liste des adresses des enregistrements correspondants (voir figure 5.7). Dans ce cas, le fichier principal ne possèdent pas de zones complémentaires pour les chaînages;

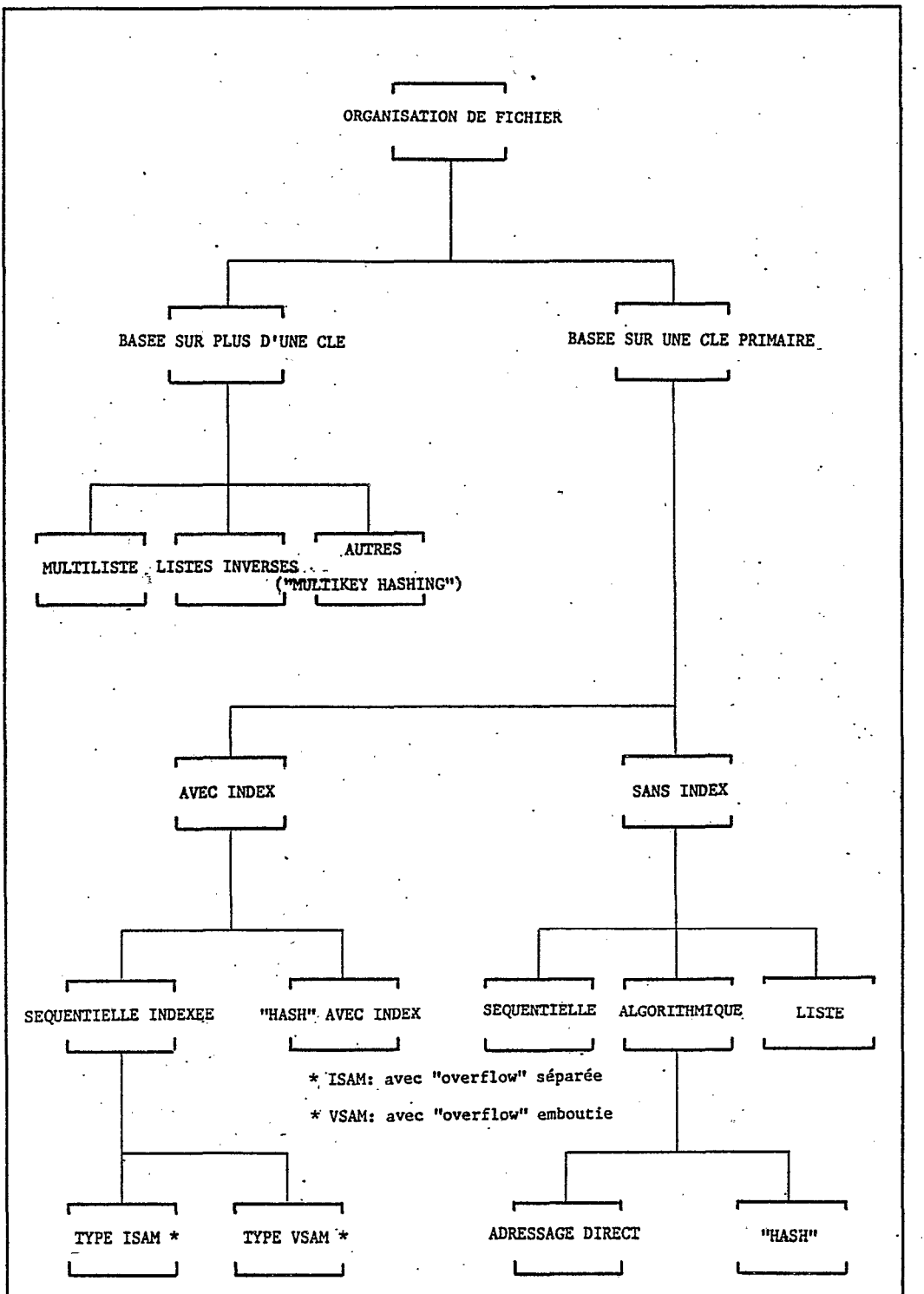


Fig. 5.5 Diverses organisations de fichiers

Fichier primaire

ENREG.	MAT.	NOM	ADRESSE		Cours	Point 1	Point 2
			RUE,...	Ville			
101	1.60	DUPONT		PARIS	MATHS		
102	1.61	DUBOIS		LYON	MATHS		
103	2.57	MARTIN		NANCY	CHIMIE		
104	1.58	ROCHE		NANCY	ANGLAIS		
105	2.59	MAIGNIEN		DIJON	CHIMIE		
106	2.61	MULLON		PARIS	MATHS		
107	2.60	BOURSIER		PARIS	ANGLAIS		
108	2.62	AUBRY		LILLE	ANGLAIS		
109	1.59	KLEIN		LYON	ANGLAIS		
110	2.58	PYENS		NANCY	MATHS		
111	2.63	JACCON		LYON	CHIMIE		
112	1.63	LHOTE		DIJON	CHIMIE		
113	1.55	BRABANT		LILLE	ANGLAIS		
114	1.56	DUMAS		PARIS	MATHS		
115	2.56	GRANDIN		ROUEN	MATHS		
116	2.64	BOREL		PARIS	ANGLAIS		
117	2.55	DELFIEU		ROUEN	MATHS		

Index "COURS"

COURS	nombre	tête
ANGLAIS	6	104
CHIMIE	4	103
MATHS	7	101

Index "VILLE"

VILLE	nombre	tête
DIJON	2	105
LILLE	2	108
LYON	3	102
NANCY	3	103
PARIS	5	101
ROUEN	2	115

Fig. 5.6 Organisation multiliste

Fichier primaire

ENREG.	MAT.	NOM	ADRESSE		COURS
			RUE,...	VILLE	
101	1.60	DUPONT		PARIS	MATHS
102	1.61	DUBOIS		LYON	MATHS
103	2,57	MARTIN		NANCY	CHIMIE
104	1.58	ROCHE		NANCY	ANGLAIS
105	2.59	MAIGNIEN		DIJON	CHIMIE
106	2.61	MULLON		PARIS	MATHS
107	2.60	BOURSIER		PARIS	ANGLAIS
108	2.62	AUBRY		LILLE	ANGLAIS
109	1.59	KLEIN		LYON	ANGLAIS
110	2.58	PYENS		NANCY	MATHS
111	2.63	JACCON		LYON	CHIMIE
112	1.63	LHOTE		DIJON	CHIMIE
113	1.55	BRABANT		LILLE	ANGLAIS
114	1.56	DUMAS		PARIS	MATHS
115	2.56	GRANDIN		ROUEN	MATHS
116	2.64	BOREL		PARIS	ANGLAIS
117	2.55	DELFIEU		ROUEN	MATHS

Index "COURS"

Cours	nb.	listes d'adresses
ANGLAIS	6	104 107 108 109 113 116
CHIMIE	4	103 105 111 112
MATHS	7	101 102 106 110 114 115

Index "VILLE"

Ville	nb.	listes d'adresses
DIJON	2	105 112
LILLE	2	108 113
LYON	3	102 109 111
NANCY	3	103 104 110
PARIS	5	101 106 107 114 116
ROUEN	2	115 117

Fig. 5.7 Organisation par listes inverses

- "multikey hashing": avec cette technique c'est le fichier principal qui est structuré différemment, par une méthode de "hash", en fonction d'un sous-ensemble d'attributs choisis pour intervenir fréquemment dans les critères d'interrogation. Le fichier est alors partitionné en sous-fichiers tels que tous les éléments qui sont contenus dans l'un d'eux possèdent certaines caractéristiques intéressantes en commun. Pour cette technique encore récente (Rothnie et al., 1974), une grande difficulté réside dans le choix des attributs d'interrogation, dont le nombre est forcément très limité. Pour les autres descripteurs, il est nécessaire de faire appel à l'une ou l'autre des deux méthodes présentées précédemment.

Dans l'exemple choisi, le maintien d'un index par ville de résidence et d'un autre par cours suivi permettra de répondre à la question:

'Quels sont les étudiants qui habitent Paris et qui suivent les cours de Mathématiques?', sans avoir à parcourir tout le fichier des élèves (qui peut être très long).

Compte tenu de l'importance que revêt l'interrogation dans un système d'informations, il apparaît intéressant d'étudier les avantages et inconvénients des deux principales organisations présentées, par rapport au critère de gestion de mémoire.

5.26 Avantages et inconvénients de ces organisations

Tant pour l'organisation multiliste que pour celle par listes inverses, il y a nécessité de maintenir des indexs secondaires: ainsi, si la récupération de l'information est plus rapide, c'est au détriment d'une occupation d'espace complémentaire. On retrouve une fois de plus l'éternel dilemme espace-temps.

Il faut remarquer que l'organisation multiliste oblige à augmenter la taille des enregistrements du fichier principal par adjonction d'une zone de chaînage par descripteur: en compensation, les indexs restent de dimensions très réduites. L'organisation par listes inverses présente quant à elle l'avantage de ne pas augmenter la taille du fichier principal: par contre les indexs secondaires sont plus importants.

Du point de vue de la récupération de l'information, l'organisation par listes inverses est la plus rapide en ce que les adresses des enregistrements sont connus après le seul traitement de l'index, alors que ce résultat n'est atteint, dans l'organisation multiliste, qu'après parcours de toute la chaîne d'enregistrements du fichier principal. Cette affirmation est toutefois discutable dans le cas où l'on dispose d'un adressage indirect.

Pour résoudre une interrogation portant sur l'intersection de deux listes (critère faisant intervenir deux attributs), comme celle qui a été prise en exemple à la fin du paragraphe précédent, au moins l'une d'entre elles devra être parcourue dans le fichier principal (de

préférence la plus petite) pour l'organisation multiliste, alors qu'il suffit de comparer les listes inverses avant d'accéder aux enregistrements désirés dans le second cas.

Pour ces dernières raisons, la méthode par listes inverses est la plus fréquemment implantée, malgré l'augmentation de volume à laquelle elle conduit pour les indexs secondaires. Il existe cependant quelques variantes de réalisation des deux méthodes (Martin, 1975), qui visent à une adaptation de celles-ci pour une gestion plus économique de l'espace: dans le cas de l'organisation multiliste on cherchera à diminuer la taille des listes, tandis que dans le cas des listes inverses on cherchera à en diminuer l'encombrement.

5.27 Variantes de l'organisation multiliste

Une première variante de cette organisation consiste à fixer une limite à priori pour la longueur de la liste: dans ces conditions, lorsque le nombre des enregistrements du fichier primaire qui appartiennent à une même liste dépasse cette limite, il y a création d'une nouvelle liste, avec une nouvelle entrée dans l'index. La liste correspondante à une valeur déterminée d'un descripteur, auparavant unique, se présente alors comme une succession de listes plus courtes, l'ensemble prenant une forme de 'peigne'.

On peut remarquer qu'avec ce contrôle de la longueur de la liste, on pourra répondre plus efficacement à des questions telles que:

'Quels sont les enregistrements qui possèdent telle valeur de la clé secondaire, dans une gamme préfixée de la clé primaire?'.

La limite du nombre d'enregistrements par liste doit être choisi judicieusement: en effet si l'on augmente indéfiniment cette limite on tend vers l'organisation multiliste pure, et le temps de recherche croît; si on l'a diminue exagérément, on tend vers l'organisation par listes inverses (obtenue pour une limite de la taille de la liste égale à 1), et c'est la taille de l'index qui augmente (voir figure 5.8).

Une seconde variante de cette organisation consiste à prendre les caractéristiques du "hardware" en considération, en adaptant les listes à des 'cellules' qui seront choisies comme étant des divisions importantes de la mémoire en terme d'accès, par exemple cylindres, pistes, pages, etc.... Chaque entrée de l'index coiffe alors la liste d'une cellule, qui peut être lue sans mouvement du mécanisme d'accès: comme l'adresse des cellules fait partie de l'adresse des enregistrements, ceci peut éliminer plusieurs cellules de la recherche (voir figure 5.9).

Lorsque le fichier est très important et occupe plusieurs volumes, les listes cellulaires peuvent être parcourues simultanément, ce qui réduit le temps total de recherche.

Finalement, pour éviter l'inconvénient d'avoir à suivre les chaînes de pointeurs écrites dans le fichier des données, il peut être avantageux de représenter ces listes dans l'index, avec les clés

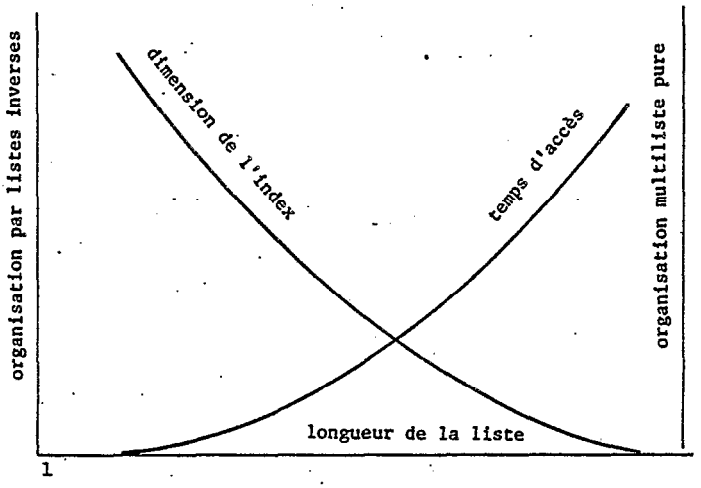
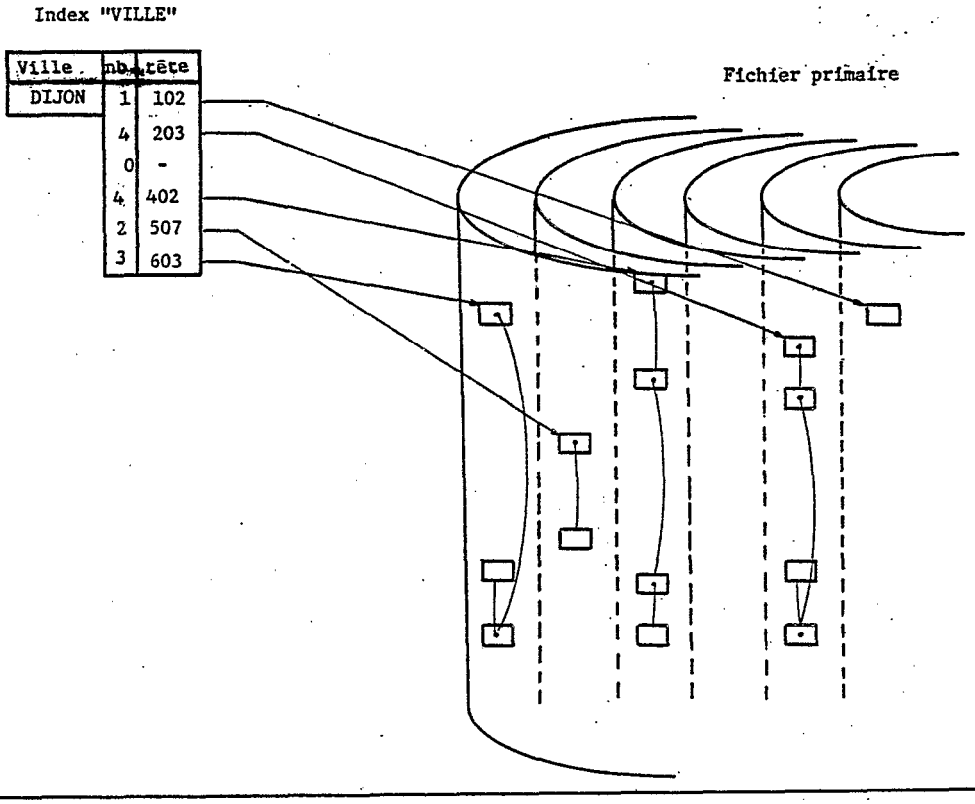


Fig. 5.8 Contrôle de la longueur de liste

Fig. 5.9 Organisation multiliste cellulaire



primaires: les listes peuvent alors être suivies sans avoir à lire de longs enregistrements de données.

5.28 Variantes de l'organisation par listes inverses

Les listes inverses peuvent être implantées selon les méthodes suivantes:

- sous forme d'une liste d'indicateurs physiques, par exemple l'adresse de chaque enregistrement;

- sous forme d'une liste d'indicateurs logiques, par exemple la clé primaire de chaque enregistrement;

- sous forme d'une chaîne de bits. Dans ce dernier cas, à chaque valeur de la clé secondaire on associe un vecteur de n bits (où n est le nombre total d'enregistrements du fichier primaire), dans lequel le bit i prend la valeur 0 si le ième enregistrement ne satisfait pas cette valeur de l'attribut, et 1 dans la cas contraire.

Bien entendu cette dernière méthode facilite les opérations logiques qui interviennent fréquemment dans l'expression des critères. Par ailleurs, il est clair que cette méthode ne réalise une économie d'espace (par rapport à la méthode classique de liste de pointeurs) qu'à partir d'une certaine dimension de la liste: là encore il faut rechercher le seuil au delà duquel la méthode devient plus rentable (voir figure 5.10).

On a d'autre part pour les listes inverses les mêmes variantes que pour l'organisation multiliste, par cellules (listes inverses cellulaires). Dans ce type d'organisation, un indicateur pointe pour chaque cellule, dans laquelle les enregistrements sont ensuite recherchés. Les indicateurs peuvent en fait être réduits à une simple chaîne de bits, où chaque liste se réfère à une cellule (voir figure 5.11). Lors d'une sélection, il est alors possible d'isoler les cellules dans lesquels une recherche devra être faite à partir de l'opération logique des chaînes de bits, conformément à l'expression logique de la question posée.

On peut également réduire la dimension des indexs en utilisant la méthode des listes inverses enchaînées: on retrouve alors l'organisation multiliste avec contrôle de la longueur de la liste.

Enfin, pour accélerer encore les accès dans les fichiers inverses, il est souvent fait appel aux techniques d'indexs hiérarchisés à plusieurs niveaux. Dans une telle structure on évite de parcourir séquentiellement tout l'index, en créant un index d'ordre supérieur qui contient une entrée pour chaque page du premier. La recherche d'une adresse est alors réalisée à partir d'un balayage de l'index d'ordre supérieur, suivi de la recherche dans la seule page sélectionnée.

Pour s'adapter à la mémoire virtuelle, et réaliser une meilleure gestion de la mémoire dans ce contexte particulier, d'autres

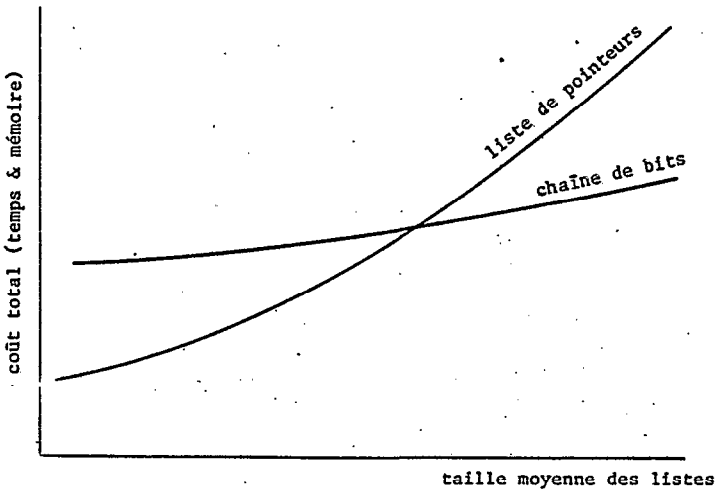
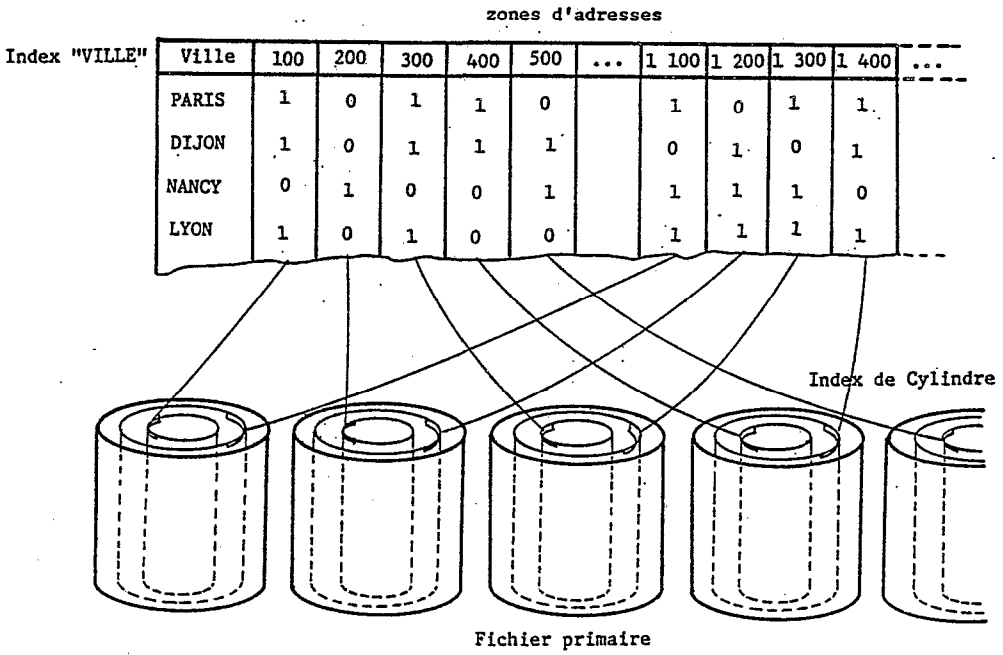


Fig. 5.10 Utilisation de chaîne de bits

Fig. 5.11 Listes inverses cellulaires parallèles



ressources peuvent être utilisées conformément aux règles de programmation qui doivent être respectées dans cet environnement.

5.29 Autres moyens d'adaptation à la mémoire virtuelle

D'autres techniques sont utilisées pour diminuer le "working-set" des "jobs" qui interviennent dans un système de gestion de base de données:

- en tout premier lieu, une structure modulaire est utilisée pour le programme lui-même. Ainsi, l'architecture d'un SGBD est généralement basée sur l'emploi d'un module principal, appelé noyau, qui assure toutes les fonctions du système et gère la base de données à laquelle lui seul a accès. Autour de ce module central, on trouvera des programmes utilitaires qui assurent les principales fonctions nécessaires à la création, l'administration, l'entretien, l'intégrité et la sécurité de la base de données. De la même façon, le moniteur de multitraitement, l'éditeur d'états et l'interprétateur de langage conversationnel sont également traités comme des modules indépendants qui accèdent à la base à travers du noyau (voir figure 5.12). Dans le cas particulier du moniteur de télétraitement, celui-ci pourra être exécuté dans une partition de mémoire différente de celle où est exécuté le noyau, pour traiter les accès sollicités par les différents utilisateurs et les passer au noyau, à l'aide d'une zone commune;

- une structure modulaire est également utilisée pour les fichiers. Ainsi les informations sont généralement stockées indépendamment de leurs relations. Il est également fait appel à la structuration des fichiers en plusieurs niveaux d'index, qui économise les échanges de pages en diminuant le nombre d'accès;

- la compression des données est également une façon simple et économique de diminuer les encombrements de mémoire. Cette technique présente en plus certains avantages secondaires non négligeables, notamment dans les économies qu'elle procure au niveau du télétraitement, ainsi que dans la sécurité naturelle des informations (puisqu'elle représente une certaine forme d'encryptage). Il existe essentiellement deux types de compression, avec de nombreuses variantes dans l'implantation: la compression par différenciation, qui utilise un enregistrement standard en ne représentant que les différences observées par rapport à celui-ci, et la compression par codification statistique, qui convertit chaque élément par un code constitué d'une chaîne de bits dont la longueur est inversement proportionnelle à sa fréquence d'utilisation. Avec des techniques de compression bien adaptées, on obtient souvent une réduction de moitié de l'encombrement (et, partant, une réduction sensible du temps d'accès) des informations, moyennant un surcroît de temps de traitement dû aux opérations de compression et décompression;

- les données à l'intérieur d'un fichier peuvent également être structurées d'une façon mieux adaptée aux sélections attendues. En effet, lorsque le fichier des données n'est pas structuré, le nombre de demandes de pages est sensiblement proportionnel au nombre de sélections réalisées (du moins tant que celui-ci reste inférieur au

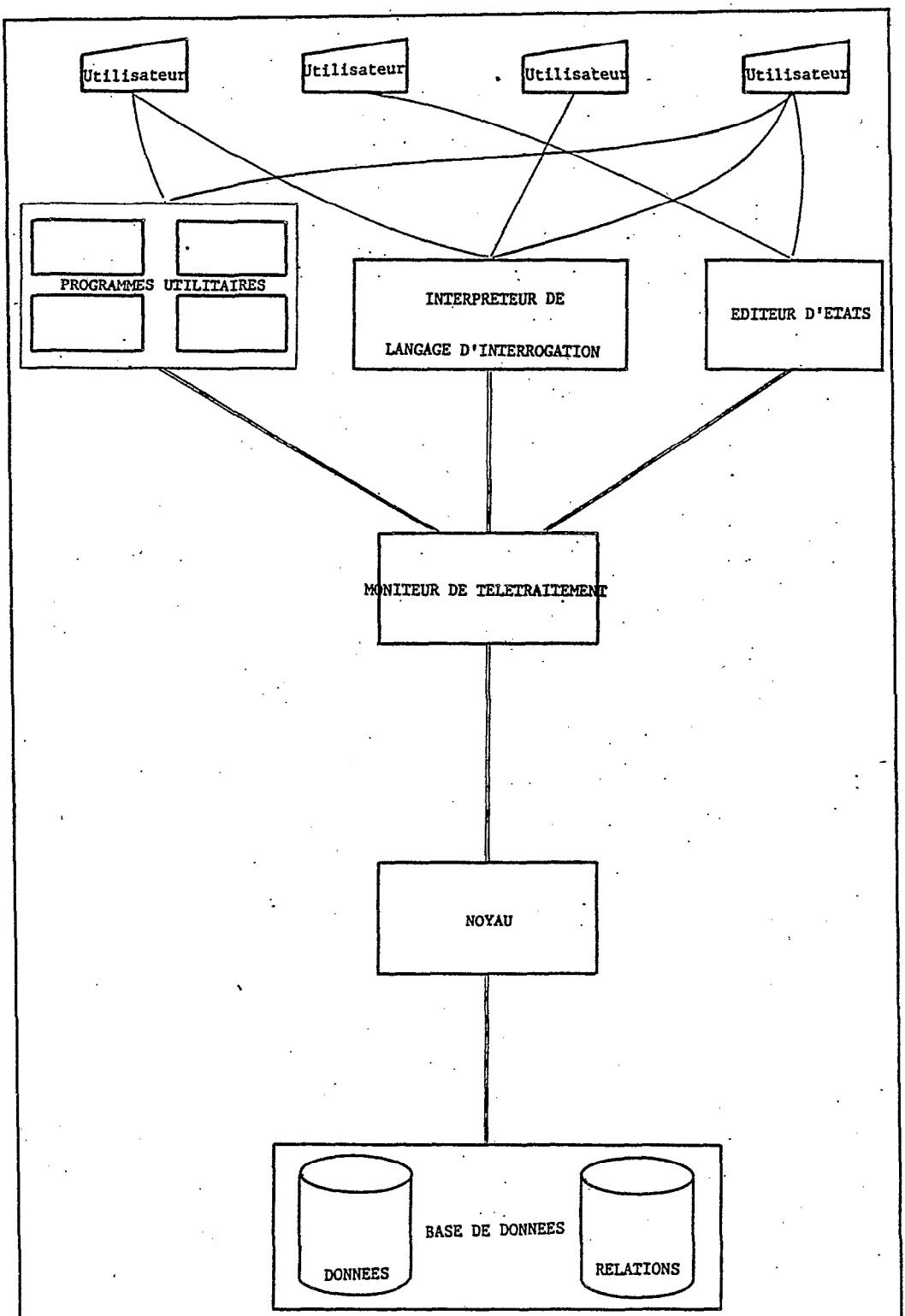


Fig. 5.12 Modularité d'un Système de gestion de bases de données

nombre total de pages occupées par le fichier). On peut espérer diminuer ce nombre d'accès par une réorganisation du fichier en fonction d'un type particulier de sélection, si l'on connaît à l'avance les attributs sur lesquels la sélection peut porter, ce qui fige le domaine d'utilisation de la base.

En conclusion, on a pu observer que les structures de données complexes entraînaient généralement un surcoût élevé, ainsi qu'une diminution de l'efficacité au profit d'une vision intégrée des informations, avec un accès plus large à celles-ci. Le respect des quelques règles générales de programmation dans un environnement de mémoire paginée, doit toutefois permettre d'assurer des performances satisfaisantes, lesquelles ne sont généralement pas les préoccupations majeures d'un analyste lors d'un projet de base de données.

En particulier les techniques de chaînage (fichier multiliste, par exemple) semblent peu favorisées par la mémoire virtuelle, sauf à contrôler les listes pour que leurs dimensions n'excède pas la taille d'une page (organisation cellulaire). Par contre, l'utilisation d'indexs à plusieurs niveaux est recommandable pour les listes inverses, pour les économies qu'elle permet de réaliser sur le nombre d'accès, et donc d'échanges de pages. D'autres améliorations peuvent encore être envisagées, comme la lecture anticipée pour prévenir les demandes de pages.

MODE DE GESTION DE MEMOIRE		CHOIX DE L'ALGORITHME
allocation séquentielle simple		indépendant (minimiser le "Working-set")
allocation de mémoire partitionnée		indépendant (minimiser le "Working-set")
allocation de mémoire avec réallocation		indépendant (minimiser le "Working-set")
allocation de mémoire par pages		indépendant (minimiser le "Working-set")
allocation de mémoire par pages avec demande de pages	substitution au hasard	indépendant (fixer les pages du programme)
	algorithme FIFO	favorise la référenciation linéaire
	algorithme LRU	favorise la référenciation linéaire
	algorithme LFU	favorise la référenciation linéaire
	algorithme WS	favorise la référenciation linéaire
	algorithme PFF	favorise la référenciation linéaire
	algorithme CPFR	favorise la référenciation linéaire
allocation de mémoire avec segmentation		indépendant (favorise la modularité du programme)
allocation de mémoire par "swapping"		indépendant (minimiser le "Working-set")

Tableau 5.1: CHOIX DE L'ALGORITHME EN FONCTION DU MODE DE GESTION DE MEMOIRE

6.CONCLUSION

L'analyse de la méthode de tri présentée dans ce travail a été essentiellement axée sur l'étude de l'adéquation du procédé envisagé à un système de mémoire virtuelle. On a mis ainsi en évidence les précautions qui doivent être prises lors de la codification des programmes utilitaires dans ce contexte, où les concepts traditionnels de tri interne et externe sont profondément altérés. En particulier, il est important de rappeler que la référenciation des données selon un mode linéaire serait la règle fondamentale de programmation dans ce contexte.

Par ailleurs, une schématisation originale du phénomène de pagination, alliée à l'adoption d'un ensemble restrictif, mais réaliste, d'hypothèses simplificatrices, permet d'aboutir à une évaluation raisonnable des temps de traitement, sévèrement grèvés par les opérations d'entrée-sorties qui interviennent dans les échanges de pages.

Le système développé conformément aux conditions d'optimisation qui ont été mises en évidence, s'est montré viable, comme on a pu le constater à la lumière des comparaisons expérimentales réalisées. L'acquisition et l'utilisation de modules projetés pour fournir des statistiques sur le phénomène de pagination (et qui sont disponibles chez le constructeur), seraient très intéressantes pour contrôler et reconnaître dans le futur l'influence de l'utilisation de la mémoire virtuelle.

La structure modulaire du programme a été projetée pour autoriser le développement postérieur de diverses phases d'entrée et de sortie. En particulier l'introduction d'un module de lecture pour un fichier séquentiel indexé traditionnel (ISAM, ou encore VSAM), relativement facile à codifier dans le contexte de ce programme, serait sans aucun doute très utilisée par les programmeurs. D'autre part le programme développé peut être utilisé dans le cas d'indexs secondaires sur disques, particulièrement courants dans l'optique 'Base de Données'.

La généralisation qui a été proposée dans le dernier chapitre, étend les résultats de cette étude à des situations plus diverses. Ainsi, pour toute mémoire de type paginée et, plus généralement, pour toute hiérarchie de stockage à deux ou plusieurs niveaux, c'est la récupération calquée sur la séquence physique des informations qui est la plus économique. Il est donc important que les algorithmes soient conçus pour s'accommoder au mieux de cette condition.

ABSTRACT

This work is concerned with a sort method chiefly destinated to ordering, in relation to secondary key, large unconventional files. The search of program efficiency in paging environment leads to a deep look at internal and external sort algorithms behavior, in this particular context.

The selection of well-adapted sort patterns is realized from the general survey of those algorithms known as being the most efficient. A comparative study of the chosen method is based on the evaluation of each job step duration, particularly paging duration. A threshold concerning the input file size has been evidenced by analytical results precedently obtained. Beyond this limit the sort made through distributing before merging philosophy becomes faster than the method based on internal sort philosophy.

The software designed in order to offer a maximum of facilities to the user (input and output device-types and access methods, blocking features, record format, etc...), was implemented on a IBM/370-135 equipment.

To demonstrate the project efficiency, experimental comparisons between the chosen model and the other sort possibilities given to the unconventional file user have been achieved, in the particular case of a B-tree organized file.

Finally, a rapid survey of the behaviour of the sort method towards the different methods of memory management, as well as a description of the types of adaptations of other algorithms to the particular virtual memory, allow to show the relative stable conclusions of this work under various conditions.

RESUMO

Apresenta-se neste trabalho um método de ordenação, voltado essencialmente para a classificação, baseada numa chave secundária, de um arquivo grande, não convencional. A procura da eficiência do programa, no contexto de memória virtual, conduziu a enfatizar o comportamento dos algoritmos de ordenação, tanto interno quanto externo, neste ambiente particular.

A seleção de alguns modos de ordenação, a priori mais convenientes, apoia-se num exame geral dos algoritmos conhecidos por serem os mais eficientes. Um estudo comparativo dos métodos selecionados é realizado através da avaliação das quantidades de tempo gasto em cada etapa do processamento, em particular pelo fenómeno de paginação. Os resultados analíticos obtidos evidenciam um valor limite do tamanho do arquivo de entrada acima do qual a ordenação, seguindo uma filosofia de formação e fusão de corridas, torna-se mais econômica que o método baseado numa simples filosofia de ordenação interna.

O sistema, projetado para oferecer ao usuário o máximo de recursos (suporte, blocagem, formatos, etc...), foi implementado numa linguagem de baixo nível, em um computador IBM/370-135. Para demostrar a eficácia do sistema desenvolvido, no caso particular de um arquivo organizado segundo uma estratégia de árvore-B, foram feitas comparações experimentais entre o modo escolhido e os outros recursos de ordenação disponíveis ao usuário desse tipo de arquivo.

Enfim, um breve exame do comportamento dos métodos de ordenação em relação aos outros métodos existentes de gerenciamento de memória, bem como uma apresentação dos métodos de adaptação de outros algoritmos no ambiente particular de memória virtual, evidenciam a relativa estabilidade das conclusões deste estudo, nas situações mais diversificadas.

REFERENCES BIBLIOGRAPHIQUES

BAYER; R., Mc CREIGHT, Organization and Maintenance of Large Ordered Indexes, Berlin, Springer Verlag, Acta Informatica (1) pp.173-189

BRAWN; B.S., GUSTAVSON; F.G., MANKIN; E.S., Sorting in a Paging Environment, Comm. ACM 13,8 pp.483-494 (1970)

COSTA; S.O., Utilização da árvore-B na administração de arquivos com independência de dados, Recife, UFPe (1980)

CRUZ; P.N., Desenvolvimento e implementação de um sistema em disco para acesso aleatório, Rio de Janeiro, UFRJ (1975)

DATE; C.J., An Introduction to Database Systems, Reading - Massachusetts, Addison-Wesley (1975)

de SOUZA; F.P., KOELSCH, Introdução a programação Assembler para os sistemas /360 e /370, Rio de Janeiro (1979)

DIGITAL DECSystem 10 SORT/MERGE User's Guide (1977)

FURTADO; A.L., dos SANTOS; C.S., Organização de Bancos de Dados, Rio de Janeiro (1979)

HOARE; C.A.R., Algorithm 64: Quicksort, Comm. ACM 4,7 p.322 (1961)

IBM (1) System /360 DOS Tape and Disk Sort/Merge Program C28-6676 (1969)

IBM (2) DOS/VS Data Management Guide GC33-5372 (1973)

IBM (3) Introduction to DOS/VS GC33-5370 (1973)

IBM (4) System /370 Principles of Operation GA22-7000 (1974)

IBM (5) OS/VS - DOS/VS - VM/370 Assembler Language GC33-4010 (1975)

IBM (6) DOS/VS Supervisor and I/O Macros GC33-5373 (1977)

IBM (7) DOS/VS System Management Guide GC33-5371 (1977)

KNUTH; D.E., The Art of Computer Programming, Reading - Massachusetts, Addison Wesley (3) pp. 1-388 (1973)

LAVELLE; P.J., Sistemas Operacionais, Recife, SUDENE-CIN (1975)

LOESER; R., Some performance tests of "Quicksort" and descendants, Comm. ACM 17,3 pp.143-152 (1974)

MADNICK; S.E., DONOVAN; J.J., Operating Systems, New York, Mc Graw-Hill (1974)

MARTIN; J., Computer Database Organization, Englewood Cliffs, New Jersey, Prentice Hall (1975)

MEYER; B., BAUDOIN, Méthodes de programmation, Paris, Eyrolles (1978)

MONGIOVI; A., Estudos de técnicas de paginaçao, Campina Grande, UFPb (1976)

QUEIROZ; M.G., LA: Linguagem para escrever Algoritmos, Recife UFPe (1978)

ROTHNIE; J.B., LOZANO; T., Attribute Based File Organization in a Paged Memory Environment, Comm. ACM 17,2 pp.63-69 (1974)

SCOWEN; R.S., Algorithm 271: Quickersort, Comm. ACM 8,11 pp.669-670 (1965)

SEDGEWICK; R., Implementing Quicksort Programs, Comm. ACM 21;10 pp.847-856 (1978)

TREMBLAY; J.P., SORENSON, An Introduction to Data Structures, pp. 465-486 (1976)

WILLIAMS; J.W.J., Algorithm 232: Heapsort, Comm. ACM (1964)