Optimisation des performance des logiciels de traitement de données sur les périphériques de stockage SSD

Arezki Laga

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UBO
Université de Bretagne Occidentale

THESE DE DOCTORAT DE

L'UNIVERSITE
DE BRETAGNE OCCIDENTALE
COMUE UNIVERSITE BRETAGNE LOIRE

ECOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : Informatique

Par **Arezki LAGA**

Optimisation des performances des logiciels de traitement de donnés sur les périphériques de stockage SSD

Thèse présentée et soutenue à l'Université de Bretagne Occidentale, UFR Sciences & techniques,
Salle de conférence : amphithéâtre F, le 20 décembre 2018 à 13h30
Unité de recherche : LabSTICC

Rapporteurs avant soutenance :

Monsieur Kamel BOUKHALFA, Professeur des Universités, USTHB
Monsieur Luc BOUGANIM, Directeur de recherche, INRIA

Composition du Jury :

Président du Jury : Monsieur Olivier BARAIS, Professeur des Universités, Université de Rennes
Madame Sorror SAHRI, Maître de conférences, Université de Paris Descartes
Monsieur Jean-thomas ACQUAVIVA, Chercheur, Data Direct Network (DDN)
Monsieur Kamel BOUKHALFA, Professeur des Universités, USTHB
Monsieur Luc BOUGANIM, Directeur de recherche, INRIA

Dir. de thèse : Monsieur Frank SINGHOFF, Professeur des Universités, Université de Bretagne Occidentale

Co-dir. de thèse : Monsieur Jalil BOUKHOBZA, Maître de conférences, Université de Bretagne Occidentale

REMERCIEMENTS

Je voudrais exprimer mes remerciements les plus sincères et ma haute reconnaissance à mes encadrants de thèse.

Je remercie mon directeur de thèse, **M. Frank Singhoff**, de m'avoir accompagné, soutenu et encouragé pendant tout le long de cette thèse. Je le remercie de m'avoir transmis les secrets d'un bon travail scientifique et de m'avoir tout simplement initié à la science.

Je remercie l'encadrant principal de ma thèse, **M. Jalil Boukhobza**, de m'avoir choisi pour être son étudiant et de m'avoir fait confiance. Je le remercie de m'avoir poussé vers le chemin de l'excellence scientifique, de m'avoir permis d'approfondir au maximum mes travaux afin de pouvoir être fier aujourd'hui du travail réalisé.

Je remercie les membres de jury de ma thèse, **M. Olivier Barais** (président du jury), **M. Luc Bouganim**, **M. Kamel Boukhalfa**, **M. Jean-thomas Acquaviva**, **Mme Sarah Soror**. Je les remercie pour l'évaluation de la qualité de mon mémoire de thèse et celle de ma soutenance. Je les remercie également pour leurs précieux conseils et pour les riches échanges autour de mes travaux de thèse.

Je remercie mes anciens collègues de KoDe software et de Koeus software et spécialement M. Michel Koskas, directeur scientifique et co-fondateur.

Enfin j'adresse mes remerciements les plus chaleureux à ma famille : Mes parents, ma femme, mon frère, ma belle-sœur et tous mes proches, qui m'ont accompagné, aidé, soutenu et encouragé tout le long de ma thèse.

PUBLICATIONS

Revues

- Arezki Laga, Jalil Boukhobza, Frank Singhoff, Michel Koskas, MONTRES : Merge ON-The-Run External Sorting Algorithm For Large Data Volumes On SSD Based Storage Systems, IEEE Transactions on Computers, volume 66, issue 10, 1689-1702, 2017.

Conférences internationales & Workshops

- Mohammed Bey Ahmed Khernache, Arezki Laga, Jalil Boukhobza, MONTRES-NVM : an External Sorting Algorithm for Hybrid Memory, The 7th Non-Volatile Memory Systems and Applications Symposium (NVMSA), Hakodate, 2018.

- Arezki Laga, Jalil Boukhobza, Michel Koskas, Frank Singhoff, Lynx : A Learning Linux Prefetching Mechanism For SSD Performance Model, The 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA), Daegu, 2016.
- Arezki Laga, Michel Koskas, Jalil Boukhobza, Frank Singhoff and Claude Brisson (2016). Revisiting External Sorting Algorithm On Flash Based Storage Systems, In Performance and Scalability of Storage Systems (PER3S), Univ. Versailles, France

Communications

- Arezki Laga. (2016) Revisited external sorting algorithm for SSD performance model, Séminaire de Statistique d'AgroParis- Tech, 20th June 2016

TABLE DES MATIÈRES

1 INTRODUCTION 1
- 1.1 Contexte et problématique . . . 1
- 1.2 Problématiques . . . 3
- 1.3 Contributions . . . 4
  - 1.3.1 Analyse des coûts d'E/S pour les algorithmes de traitement de données . . . 4
  - 1.3.2 Optimisation des algorithmes de traitement de données . . . 4
  - 1.3.3 Optimisation du mécanisme de pré-chargement de données sur les périphériques SSD . . . 4
- 1.4 Plan du mémoire . . . 5

I CONTEXTE ET ÉTAT DE L'ART 7

2 ORGANISATION, ACCÈS ET TRAITEMENT DES DONNÉES 9
- 2.1 Base de données : définitions . . . 9
- 2.2 Tri des données . . . 13
  - 2.2.1 Algorithme de tri par fusion . . . 14
  - 2.2.2 Le tri par sélection et remplacement . . . 16
- 2.3 Gestion des E/S . . . 17
  - 2.3.1 Mécanisme d'accès aux données sur mémoire secondaire . . . 18
  - 2.3.2 Pré-chargement de données dédié à un algorithme spécifique 21
  - 2.3.3 Pré-chargement de données dédié à une séquence d'accès aux données . . . 21
- 2.4 Conclusion . . . 22

3 LES SUPPORTS DE STOCKAGE À BASE DE MÉMOIRE FLASH NAND 23
- 3.1 Généralités sur les mémoires Flash . . . 24
- 3.2 Structure d'une mémoire flash . . . 24
  - 3.2.1 Page . . . 25
  - 3.2.2 Bloc . . . 25
  - 3.2.3 Plan . . . 25
  - 3.2.4 Dé . . . 25
- 3.3 Opérations d'accès à la mémoire flash . . . 25
  - 3.3.1 Opération de lecture . . . 26
  - 3.3.2 Opération d'écriture . . . 27
  - 3.3.3 Opération d'effacement . . . 27
- 3.4 Contraintes liées à l'utilisation des mémoires flash . . . 27
  - 3.4.1 Contrainte C1 : la mise à jour de données . . . 28

3.4.2 Contrainte C2 : l'usure . . . 28
3.4.3 Contrainte C3 : la fiabilité . . . 28
3.5 Gestion des contraintes . . . 28
3.5.1 Gestion de la contrainte C1 : la mise à jour de données . . . 29
3.5.2 Gestion de la contrainte C2 : l'usure . . . 30
3.5.3 Gestion de la contrainte C3 : la fiabilité . . . 30
3.6 Architecture d'un support de stockage à base de mémoire flash . . . 30
3.7 Conclusion . . . 33
4 ETAT DE L'ART CONCERNANT L'OPTIMISATION DES E/S SUR LES PÉRIPHÉRIQUES SSD 35
4.1 Optimisation des algorithmes de tri en mémoire secondaire pour les périphériques SSD . . . 35
4.1.1 FAST(1) . . . 35
4.1.2 FAST(N) . . . 39
4.1.3 Natural Page Run . . . 39
4.1.4 Minsort . . . 41
4.1.5 Fsort . . . 44
4.1.6 TagSort . . . 44
4.1.7 FMSort . . . 46
4.2 Optimisation des mécanismes de gestion des E/S en mémoire secondaire . . . 50
4.2.1 FAST : Fast Application STarter . . . 50
4.2.2 Flashy . . . 51
4.3 Conclusion . . . 53
II CONTRIBUTIONS 55
5 ANALYSE DES COÛTS D'E/S POUR LES ALGORITHMES DE TRI OPTIMISÉS SUR SSD 57
5.1 Hypothèses, définitions et notations du modèle de coût E/S . . . 58
5.1.1 Définitions . . . 58
5.1.2 Hypothèses . . . 58
5.1.3 Notations . . . 59
5.2 Évaluation des coûts d'E/S . . . 60
5.2.1 Le tri par fusion . . . 60
5.2.2 Natural page run . . . 61
5.2.3 MinSort . . . 61
5.2.4 FAST(1) . . . 62
5.2.5 FAST(N) . . . 62
5.3 Analyse des coûts d'E/S . . . 65
5.3.1 Natural Page Run . . . 65
5.3.2 FSort . . . 66
5.3.3 MinSort . . . 66

5.3.4 FAST(N) . . . 67
5.4 Conclusion . . . 67
6 MONTRES : ALGORITHME DE TRI EN MÉMOIRE SECONDAIRE BASÉ SUR LA FUSION À LA VOLÉE 69
6.1 Description de l'algorithme et de ses optimisations . . . 70
6.1.1 Optimisation de la phase de génération des runs . . . 70
6.1.2 Le mécanisme de fusion à la volée . . . 73
6.1.3 Le mécanisme d'expansion des *runs* . . . 75
6.1.4 Préparation de la phase de fusion des runs . . . 77
6.2 Optimisation de la phase de fusion des runs . . . 77
6.2.1 Sélection et fusion des blocs . . . 77
6.2.2 Structure de données utilisées dans l'opération de fusion des *runs* . . . 79
6.3 Evaluation des coûts d'E/S pour MONTRES . . . 79
6.3.1 Coût en E/S pour la génération des *runs* . . . 80
6.3.2 Coût en E/S de la phase de fusion des runs . . . 81
6.4 Analyse et discussion du coût en E/S de MONTRES . . . 82
6.5 Validation Experimentale . . . 83
6.5.1 Méthodologie d'expérimentation . . . 83
6.5.2 Description des expériences . . . 85
6.6 Résultats et discussions . . . 86
6.6.1 Résultats de l'expérience 1 . . . 86
6.6.2 Résultats de l'expérience 2 . . . 87
6.6.3 Analyse des résultats obtenus en désactivant le cache . . . 94
6.6.4 Analyse des optimisations de MONTRES . . . 96
6.7 Conclusion . . . 99
7 LYNX : UN MÉCANISME DE PRÉ-CHARGEMENT DE DONNÉES SUR UN SSD 101
7.1 Introduction . . . 101
7.2 Modèle d'apprentissage et de prédiction . . . 102
7.2.1 Définitions . . . 102
7.2.2 Modéliser un motif d'accès . . . 102
7.3 Mise en œuvre du modèle d'apprentissage et de prédiction dans Lynx . . . 103
7.3.1 Phase d'apprentissage . . . 104
7.3.2 Phase de prédiction . . . 105
7.3.3 Contrôle de la précision pour la prédiction . . . 105
7.3.4 Contrôle du mécanisme par le programme utilisateur . . . 106
7.4 Intégration de Lynx dans le noyau Linux . . . 106
7.4.1 Mécanisme d'apprentissage automatique . . . 106
7.4.2 Mécanisme de prédiction . . . 107

7.5 Expérimentations et discussion des résultats . . . 107
7.5.1 Méthodologie expérimentale . . . 107
7.5.2 Discussion des résultats . . . 108
7.6 Conclusion . . . 110

iii CONCLUSION 113

8 CONCLUSION 115
8.1 Résumé des contributions . . . 115
8.1.1 Étude des coûts d'E/S des algorithmes de tri en mémoire secondaire . . . 115
8.1.2 Optimisation du tri en mémoire secondaire . . . 115
8.1.3 Mécanismes de pré-chargement de données . . . 116
8.2 Perspectives . . . 116
8.2.1 Dans un système réparti . . . 117
8.2.2 Lynx . . . 118
8.2.3 Dans une hiérarchie mémoire hétérogène . . . 118
8.2.4 Lynx . . . 119

TABLE DES FIGURES

FIGURE 1 Schémas de la base de données TPC-H 11
FIGURE 2 Illustration du concept de MapReduce 12
FIGURE 3 Illustration de la génération des *runs* 14
FIGURE 4 Illustration de la fusion des *runs* 15
FIGURE 5 Déroulement du tri avec l'algorithme de remplacement et de sélection 17
FIGURE 6 Illustration d'une opération de lecture déclenchée par une demande de valeur (avec $Valeur_{i,j}$ la j^{eme} valeur dans le i^{eme} bloc). 19
FIGURE 7 Illustration d'une opération d'écriture 20
FIGURE 8 Illustration du mécanisme de read-ahead 22
FIGURE 9 Structure interne hiérarchique d'une mémoire flash 26
FIGURE 10 Structure interne d'un support de stockage SSD 31
FIGURE 11 Déroulement du tri avec l'algorithme FAST 38
FIGURE 12 Déroulement du tri avec l'algorithme "Natural Page Run" 41
FIGURE 13 Déroulement de l'opération de tri d'un ensemble de données avec l'algorithme Minsort 43
FIGURE 14 Déroulement de l'opération de tri avec TagSort 46
FIGURE 15 Déroulement de l'opération de tri avec TagSort 47
FIGURE 16 Déroulement d'une opération de fusion avec FMSort . . . 49
FIGURE 17 Mécanismes inclus dans l'algorithme de génération des *runs* de MONTRES 71
FIGURE 18 Déroulement du mécanisme de sélection des minimums sur un fichier contenant 8 blocs 72
FIGURE 19 Déroulement du mécanisme de fusion à la volée lors de la création du premier run 74
FIGURE 20 Déroulement du mécanisme de fusion à la volée lors de la création du deuxième run 75
FIGURE 21 Déroulement du mécanisme d'expansion des runs 76
FIGURE 22 Structure de de données utilisée par le le mécanisme de fusion dans MONTRES 79
FIGURE 23 Accélération du temps d'exécution du tri par fusion avec MONTRES et FSort, résultats obtenus avec les ensembles de données (A) sur les trois SSD de type DC, PC et server. **RG** : *Run Generation*, **RM** : *Run Merge*. 88

Figure 24 Accélération du temps d'exécution du tri par fusion avec MONTRES et FSort, résultats obtenus avec les ensembles de données **(B)** sur les trois SSD de type DC, PC et server. 91
Figure 25 Accélération du temps d'exécution du tri par fusion avec MONTRES et FSort, résultats obtenus avec les ensembles de données (C) sur les trois SSD de type DC, PC et server. 93
Figure 26 Accélération du temps d'exécution du tri par fusion avec MONTRES et FSort, résultats obtenus avec les ensembles de données (D) sur les trois SSD de type DC, PC et server. 94
Figure 27 Accélération du temps d'exécution du tri par fusion avec MONTRES et FSort, résultats obtenus avec les ensembles de données (E) sur les trois SSD de type DC, PC et server. 95
Figure 28 Accélération du temps d'exécution du tri par fusion avec MONTRES et FSort, résultats obtenus avec les ensembles de données **(A), (B), (C), (D)** et **(E)** sur le SSD de type server et en désactivant le cache. 96
Figure 29 Fichier contenant 8 blocs de 4 valeurs 104
Figure 30 Exemple de chaîne de Markov 104
Figure 31 Exemple d'un tableau de transition 107
Figure 32 Temps d'exécution des requêtes TPC-H 109
Figure 33 Gain en Miss 110
Figure 34 Temps systèmes mesurés sur les requêtes TPC-H 111

LISTE DES TABLEAUX

TABLE 1 Notations 36
TABLE 2 Notations 60
TABLE 3 Coût en E/S des principaux algorithmes de tri en mémoire secondaire 64
TABLE 4 Quantités de valeurs minimales récupérées γ 74
TABLE 5 Performances des SSD utilisés 84
TABLE 6 Résultats obtenus dans l'expérience 1 86
TABLE 7 Mesure des coûts d'E/S pour MONTRES avec l'ensemble de données (A) 90
TABLE 8 Mesure des coûts d'E/S pour MONTRES avec l'ensemble de données **(B)** 92
TABLE 9 Moyenne des coûts d'E/S mesurés pour MONTRES avec les ensembles de données **(C)**, **(D)** et **(E)** 95
TABLE 10 Mesures des accélérations obtenues séparément avec les optimisations de MONTRES sur les jeux de données **(A)**, **(B)** et **(C)** 98
TABLE 11 Notations utilisées dans le modèle d'apprentissage et de prédiction 105

Chapitre 1

INTRODUCTION

Le développement des activités économiques, industrielles et même scientifiques est étroitement lié à l'extraction de la connaissance et de l'information à partir du traitement de données collectées d'un environnement, d'un phénomène ou d'une activité [38]. Par conséquent, la collecte de données est devenue une activité stratégique pour un grand nombre d'organisations [66]. Cette activité s'appuie aujourd'hui sur les technologies du numérique comme les objets connectés, les smartphones et les réseaux sociaux. Avec ces nouvelles technologies, il est devenu possible de récolter des données précises et volumineuses, ce qui amène à une évolution sans précédent des volumes de données.

Cette évolution n'est pas sans impact sur les performances des infrastructures informatiques et des logiciels de traitement de données [5, 38]. Dans ce contexte, plusieurs travaux de recherche sont conduits afin d'améliorer les performances des opérations sur les grands volumes de données.

1.1 CONTEXTE ET PROBLÉMATIQUE

Selon une étude réalisée pour la société Seagate et publiée dans [57], l'humanité a créé 16.1 Zeta octets de données au courant de l'année 2016. On estime, dans cette même étude, que le volume de données qui serait généré en 2025 atteindrait 163 Zeta octets. Ceci présente plusieurs défis pour les infrastructures de stockage et logiciels de traitement de données [61]. D'une part, ces infrastructures doivent permettre le stockage et faciliter l'accès aux données de plus en plus volumineuses et d'autre part, les logiciels de traitement de données doivent assurer un traitement efficace de ces données.

Dans ce contexte, la technologie évolue sur deux axes principaux. Le premier axe consiste à améliorer certaines propriétés des périphériques se trouvant sur le chemin de données (processeurs, mémoires). Le deuxième axe consiste à réduire les temps de réponse des logiciels de traitement de données.

En l'espace d'une vingtaine d'années, la vitesse des processeurs a été multipliée par mille et les capacités des mémoires principales sont passées de quelques

méga-octets à plusieurs giga-octets [58]. Les périphériques de stockage ont également évolué en termes de performance et de capacité. Sur ce même élan, nous avons assisté à l'émergence de nouveaux périphériques de stockage appelés *Solid State Drive* (SSD). Ces périphériques embarquent des mémoires flash et présentent de nouvelles caractéristiques en termes de performance et de consommation énergétique [14].

Plusieurs études ont pour objectif d'accélérer les temps de réponse des logiciels de traitement de données. Ces études suivent de près les évolutions des périphériques se trouvant sur le chemin de données. Chaque fois qu'un périphérique évolue en offrant de nouveaux avantages, les logiciels de traitement de données sont adaptés pour profiter de ces propriétés [5, 58].

Les périphériques de stockage magnétiques appelés *Hard Disk Drive* (HDD) ont longtemps dominé les infrastructures de stockage. Ils sont équipés d'une mémoire magnétique et d'un bras mécanique qui se charge des opérations de lecture et d'écriture. Les opérations d'Entrées / Sorties (E/S) séquentielles sur un périphérique HDD provoquent moins de déplacement du bras mécanique que les opérations de lectures aléatoires. Par conséquent, le débit des opérations d'E/S séquentielles sur un périphérique HDD est plus élevé que celles des opérations aléatoires.

Les périphériques SSD offrent un nouveau modèle de performance différent de celui des HDD [40]. Les opérations de lectures aléatoires et les opérations de lecture séquentielles ont des performances symétriques sur un périphérique SSD, alors que les performances des opérations de lecture sont asymétriques avec celles des opérations d'écriture. En effet, les lectures sont plus performantes que les écritures.

Les périphériques SSD offrent un débit en opération d'E/S plus élevé que les périphérique HDD. Dans une étude présentée dans [49], les auteurs montrent qu'en remplaçant un périphérique HDD avec un périphérique SSD, un logiciel de traitement de données peut être plus performant. En plus des gains en performance, les SSD consomment moins d'énergie et résistent mieux aux chocs et aux températures élevées [14]. Ces avantages ont accéléré leur intégration dans les infrastructures de stockage, bien que leurs coûts soient élevés.

Les périphériques SSD présentent néanmoins quelques inconvénients au niveau des opérations d'effacement de données. En effet, ces opérations ont des latences 10x plus élevées que les opérations de lecture et réduisent la durée de vie des cellules flash du SSD [14]. Nous décrivons plus en détail ces inconvénients dans le chapitre 3.

Cette thèse est réalisée dans le cadre d'une convention CIFRE entre la société KoDe Software et le laboratoire Lab-STICC CNRS UMR 6285 (Laboratoire en Sciences et Techniques de l'Information, de la Communication et de la Connaissance). La société KoDe software est un éditeur de logiciel de base de données,

qui a exploité les brevets décrit dans [42, 43] afin de développer un logiciel de base de données de haute performance. Le logiciel développé par KoDe software est historiquement optimisé pour les performances des périphériques HDD et n'exploite pas le modèle de performance des SSD.

1.2 PROBLÉMATIQUES

Dans le cadre de cette thèse, nous explorons de nouvelles opportunités d'optimisation des coûts d'E/S pour les logiciels de traitement de données, en considérant les caractéristiques des périphériques SSD. Pour ce faire, nous étudions trois problématiques que nous présentons ci-dessous.

Problème 1. *L'évaluation du coût en E/S est une étape à considérer dans tout travail d'optimisation des E/S pour un algorithme de traitement de données.*

Dans la littérature, nous retrouvons quelques modèles d'évaluations des coûts en E/S [6, 29, 41, 63]. Ces modèles sont limités aux E/S sur les périphériques HDD. La question se pose alors quant à leur utilisation avec des périphériques SSD. Si toutefois ces modèles ne sont pas applicables à ces périphériques, lesquels pouvons nous élaborer pour ces périphériques ?

Problème 2. *Les algorithmes de traitement de données en mémoire secondaire sont initialement optimisés pour les périphériques HDD [41], en particulier le tri. Le volume de données augmente plus vite que la capacité mémoire des systèmes informatiques. D'autre part, les périphériques SSD se démocratisent de plus en plus et offrent un modèle de performance spécifique.*

Dans ce contexte, comment faire évoluer les algorithmes de tri afin de satisfaire les nouveaux besoins applicatifs ?

Problème 3. *Une technique connue permettant de réduire les coûts en E/S pour les logiciels de traitement de données consiste à anticiper les opérations de lecture et précharger les données en mémoire principale pour une utilisation ultérieure [27]. Nous retrouvons les techniques de pré-chargement de données dans les logiciels de traitement de données ainsi que dans les systèmes d'exploitation.*

Ces techniques sont initialement conçues pour les accès sur périphériques HDD ; elles considèrent uniquement les lectures séquentielles.

Sur un périphérique SSD, les lectures séquentielles sont aussi performantes que les lectures aléatoires. Dans ce contexte, comment faut il adapter les mécanismes de pré-chargement afin de tirer profit de cette nouvelle propriété ?

1.3 CONTRIBUTIONS

Dans les sections qui vont suivre, nous présentons les contributions que nous avons réalisées pour répondre aux problématiques énoncées précédemment.

1.3.1 *Analyse des coûts d'E/S pour les algorithmes de traitement de données*

Pour répondre à la problématique 1, nous avons conçu un modèle pour calculer le coût en E/S d'un algorithme de traitement de données. Ce modèle permet de calculer, pour un algorithme donné, le nombre d'accès en lecture et en écriture. Ce modèle se base sur trois paramètres principaux : la taille des données, l'espace alloué en mémoire principale et la distribution des données. Le modèle est conçu pour être flexible : il peut intégrer d'autres paramètres liés à la gestion des E/S au niveau du système d'exploitation et du périphérique de stockage. Ce travail a fait l'objet d'une publication en 2017 dans la revue IEEE Transaction On Computers [45].

1.3.2 *Optimisation des algorithmes de traitement de données*

Pour répondre à la problématique 2, nous avons choisi d'étudier et d'optimiser les coûts en E/S sur SSD pour les algorithmes de tri en mémoire secondaire, largement sollicités par les logiciels de traitement de données [41]. Dans ce sens, nous avons proposé un nouvel algorithme de tri en mémoire secondaire appelé MONTRES (*Merge ON-The-Run External Sorting*). Cet algorithme vise à réduire la quantité de données temporaires générée afin de diminuer le nombre d'opérations d'écriture et d'effacement (coûteuses sur SSD). Pour ce faire, l'algorithme utilise des lectures supplémentaires notamment des lectures aléatoires (plus performantes que les écritures) afin de récupérer un maximum de valeurs minimales. Ces valeurs sont écrites directement sur le fichier final trié. Ce travail a également été publié dans la revue IEEE Transaction On Computers [45].

1.3.3 *Optimisation du mécanisme de pré-chargement de données sur les périphériques SSD*

Le noyau Linux met en œuvre une technique de pré-chargement de données en mémoire principale qui permet de réduire l'écart des performances entre la mémoire principale et la mémoire secondaire [26]. Cette technique de pré-chargement tient compte de la localité spatiale des accès aux données pour anticiper les opérations de lectures séquentielles. Or, cette technique ne prend pas

en compte les accès qui ne respectent pas la localité spatiale, elle ne s'applique alors qu'aux lectures séquentielles.

Afin de répondre à la problématique 3, nous avons proposé une nouvelle technique de pré-chargement de données qui permet de prédire les lectures aléatoires en plus des lectures séquentielles. Cette technique utilise une méthode d'apprentissage qui permet de prédire les accès futurs aux données pour ensuite les pré-charger en mémoire principale. Ce travail a fait l'objet d'une publication dans la conférence internationale NVMSA (*Non-Volatile Memory Systems and Applications Symposium*) en Août 2016 [44].

1.4 PLAN DU MÉMOIRE

Ce manuscrit de thèse est composé de huit chapitres que nous présentons dans les sections suivantes : cette introduction, 3 chapitres d'état de l'art, 3 chapitres de contribution et un chapitre de conclusion.

Le **chapitre 2** introduit quelques généralités sur les Systèmes de Gestion de Base de Données et les mécanismes d'accès aux données au niveau d'un système d'exploitation.

Le **chapitre 3** présente des généralités sur les mémoires flash et les périphériques SSD.

Le **chapitre 4** décrit les travaux de l'état de l'art qui considèrent le modèle de performance des SSD pour réduire le temps de réponse des opérations de tri en mémoire secondaire et pour améliorer l'efficacité des mécanismes de pré-chargement de données sur SSD.

Le **chapitre 5** présente une étude des coûts en E/S des algorithmes de tri présentés dans l'état de l'art (chapitre 4) et propose un modèle de calcul des coûts d'E/S.

Le **chapitre 6** présente MONTRES, l'algorithme proposé pour réduire le coût en E/S du tri en mémoire secondaire en tenant compte des propriétés des SSD.

Le **chapitre 7** présente Lynx, notre technique de pré-chargement qui permet d'anticiper des lectures aléatoires sur des périphériques SSD.

Le **chapitre 8** conclut le manuscrit et présente quelques perspectives à ce travail.

Première partie

CONTEXTE ET ÉTAT DE L'ART

Chapitre 2

ORGANISATION, ACCÈS ET TRAITEMENT DES DONNÉES

Aujourd'hui, le succès d'une organisation dépend de ses capacités à transformer les données en informations afin de guider son activité [33]. Par conséquent, l'intérêt pour les données s'est imposé et revêt un caractère stratégique essentiel.

Les données sont généralement récoltées à l'état brut à partir des sources de données, qui peuvent être des sites internet, des objets connectés ou des réseaux sociaux. Ces données sont structurées, puis sauvegardées dans une base de données. Elles subissent alors plusieurs traitements et analyses pour être enfin transformées en information.

Dans ce chapitre, nous introduisons les différents concepts liés aux bases de données, notamment l'organisation, la manipulation et le traitement de données. Ensuite, nous décrivons deux algorithmes de traitement de données et enfin nous présentons la gestion des accès aux données en mémoire secondaire.

2.1 BASE DE DONNÉES : DÉFINITIONS

Dans [32], on définit une base de données comme une collection de données décrivant une activité. Une collection de données est organisée de manière à assurer les propriétés suivantes :

- Optimiser l'accès et le traitement des données.
- Garantir la cohérence et l'homogénéité des données.

L'industrie des bases de données a émergé avec l'apparition des premiers ordinateurs à la fin de la deuxième guerre mondiale. Les premières applications de traitement de données intégraient leur propre programme de gestion de données. Cette approche a permis de répondre aux besoins de l'époque, mais présentait des inconvénients en matière d'évolutivité. En effet, avec cette approche, chaque application se voit faire évoluer son programme de gestion de données pour

répondre à de nouveaux besoins. Par conséquent, il était devenu nécessaire d'externaliser la gestion des données, ce qui a donné naissance aux bases de données et aux Systèmes de Gestion de Base de Données (SGBD).

Définition 1. *Un SGBD est un logiciel qui permet la gestion d'une ou de plusieurs bases de données. Il offre plusieurs fonctionnalités aux administrateurs et utilisateurs de bases de données. Pour les administrateurs, il offre la possibilité de créer et d'optimiser les bases de données, de créer des utilisateurs et de gérer les droits d'accès. Pour les utilisateurs, le SGBD permet d'insérer et de récupérer des données d'une bases de données pour ensuite les traiter [33].*

Afin de rendre homogène le stockage et la gestion des données, il est nécessaire de structurer les données suivant un modèle. Les premiers modèles de données ayant été proposés sont le modèle hiérarchique et le modèle réseau. Ces deux modèles présentent plusieurs inconvénients en termes d'organisation et de requêtes sur les données [33]. Dans [21], l'auteur décrit un nouveau modèle de données dérivé de l'algèbre relationnel. Ce modèle est connu sous le nom de modèle relationnel et a donné naissance aux bases de données relationnelles (voir la définition 2).

Définition 2. *Une base de données relationnelle présente les données sous forme de tables, appelées entités (voir l'exemple 1). Chaque colonne de la table représente un attribut, auquel est associé un domaine de valeurs. Les lignes d'une table sont appelées tuples (en anglais) ou enregistrements en français. Le modèle relationnel définit des relations entre les tables qui associent les enregistrements issus de deux tables ou plus [33].*

Exemple 1. *Dans la Figure 1, nous présentons le schémas d'une base de données relationnelle issue du benchmarck TPC-H [23]. Cette base de données décrit l'activité d'une entreprise de grande distribution. La base de données comporte huit tables (entités), que nous décrivons ci-dessous :*

1. ***Nation** : une ligne de cette table décrit le pays d'un ou de plusieurs fournisseurs et/ou clients.*
2. ***Region** : une ligne de la table **région** décrit un continent.*
3. ***Supplier** : cette table décrit les fournisseurs.*
4. ***Customer** : cette table décrit les clients.*
5. ***Part** : cette table décrit les produits.*
6. ***PartSupp** : cette table décrit les informations communes entre chaque produit et ses différents fournisseurs.*
7. ***Orders** : cette table décrit les informations sur les commandes.*

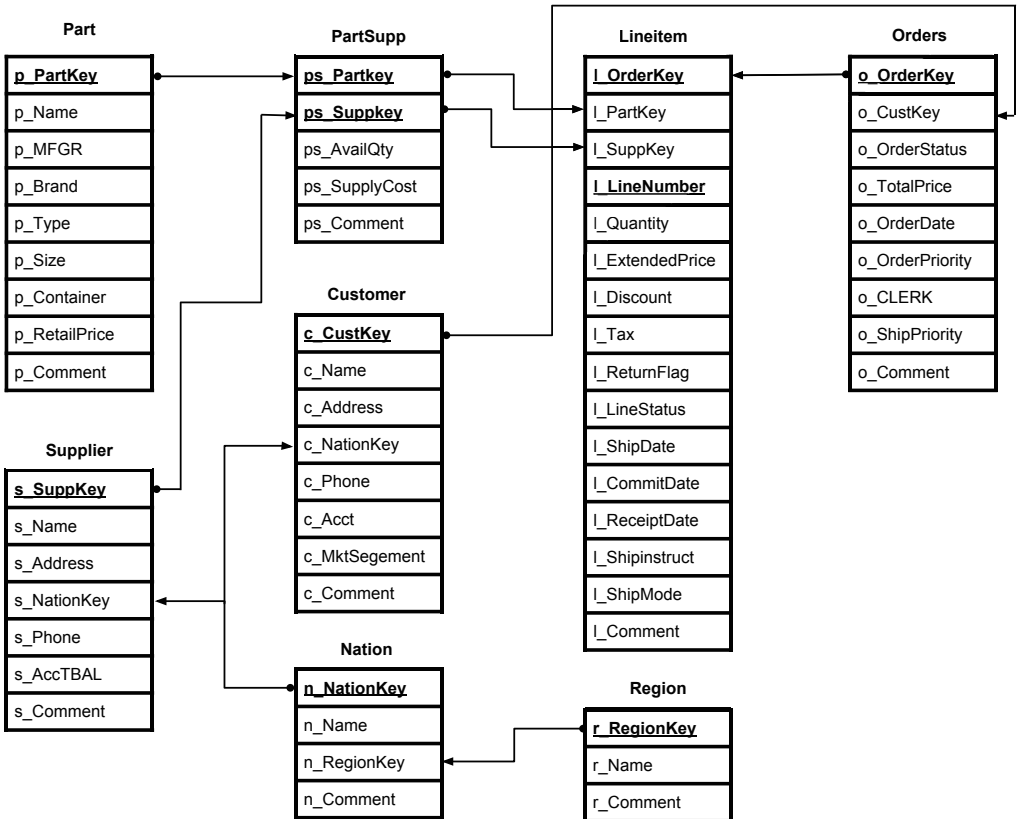

FIGURE 1 : Schémas de la base de données TPC-H

8. ***Lineitem*** *: cette table décrit les informations concernant les ventes.*

Avec l'émergence des grands sites internet dans les années 2000, et l'explosion du volume de données, de nouveaux besoins sont apparus en termes d'organisation, de gestion et de traitement des données [33]. Les bases de données relationnelles n'ayant pas pu évoluer pour répondre à ce besoin, de nouvelles solutions sont alors apparues.

En 2003, Google a révélé son système de fichiers distribué GFS (*Google File System*) [28]. Puis, en 2004, Google a présenté les détails de ses algorithmes de traitement parallèle et distribué *MapReduce* [24] (voir la définition 3). Enfin, en 2006 Google a dévoilé la structure de sa base de données distribuée *BigTable* [17]. Ces nouveaux concepts ont évolué dans la communauté des bases de données pour donner naissance au projet *Hadoop* de Yahoo [33].

Définition 3. *MapReduce est un modèle de programmation qui inclut deux fonctions principales [50] : "map" et "reduce". La première fonction "map" transforme les données en entrées en couples (clé, valeur). La deuxième fonction "reduce" est utilisée pour agréger des valeurs ayant des clés identiques. Ce modèle de programmation peut être mis en œuvre en parallèle, car les deux fonctions "map" et "reduce" peuvent être parallélisées. En effet, la fonction "map" peut être réalisée par plusieurs noeuds d'un système distribué.*

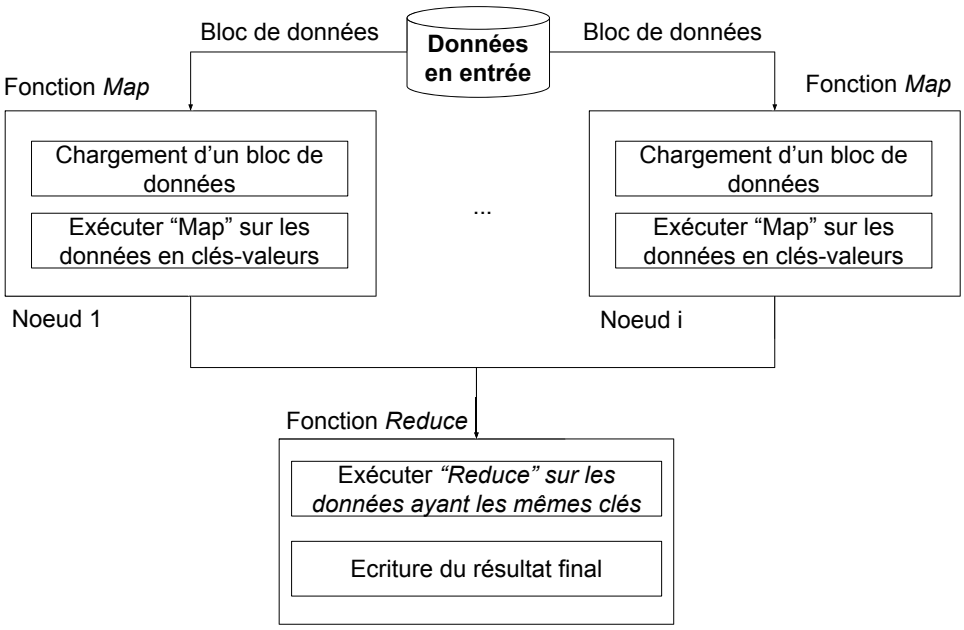

FIGURE 2 : Illustration du concept de MapReduce

Une fois les résultats des fonctions "map" obtenus, la fonction "reduce" est lancée à son tour (voir la Figure 2).

Au niveau des opérations de traitement de données, les logiciels de bases de données offrent la possibilité d'exécuter une myriade d'opérations. Parmi ces opérations, nous retrouvons les opérations algébriques, les opérations arithmétiques ainsi que les opérations de tri. Cette dernière opération est considérée par plusieurs études [30, 41] comme étant la plus sollicitée dans un SGBD relationnel. D'autres études plus récentes [50] démontrent que le tri de données demeure largement sollicité dans les SGBD utilisant le concept *MapReduce* comme *Hadoop*. Nous définissons l'opération de tri en base de données dans la définition 4.

Définition 4. *Trier des données consiste à mettre des valeurs dans un ordre déterminé par une fonction de comparaison [30, 41].*

Le tri est sollicité par MapReduce pour faire des tâches différentes de celles auxquels un SGBD relationnel le sollicite. En effet, dans un SGBD relationnel, les opérations de tri sont sollicitées pour réaliser les tâches suivantes [30] :

- Optimisation des opérations de recherche sur les données, des opérations de création des index, des opérations ensemblistes et des opérations de jointures.
- Détection de l'anatomie des ensembles de données, comme la détection des doublons qui est une opération utilisée pour le contrôle des contraintes de clés primaires.

- Organisation des données avant de les restituer à l'utilisateur.

Dans un SGBD utilisant *MapReduce*, les opérations de tri sont sollicitées pour réaliser des tâches de regroupement de données (*Map*) et de réduction de volume de données (*Reduce*) [50].

Etant donné que l'opération de tri est largement sollicitée par les différents logiciels de base de données, nous nous sommes intéressés dans le cadre de cette thèse à l'étude et l'optimisation des algorithmes de tri utilisés dans les SGBD.

2.2 TRI DES DONNÉES

Mettre des données dans l'ordre nécessite de comparer chaque donnée avec une partie ou la totalité des données d'un ensemble [30, 41]. Les données sont comparées et permutées jusqu'à obtention d'un ordre correct. Alors que les opérations de comparaison se déroulent au niveau du processeur, les opérations de permutation se déroulent au niveau de la mémoire principale, il est donc nécessaire d'avoir en mémoire toutes les données à ordonner.

Les algorithmes de tri classiques, largement étudiés dans la littérature, trient des données en mémoire principale. On les appelle algorithmes de tri en mémoire. Parmi ces algorithmes, nous pouvons citer les plus connus comme le tri rapide et le tri par tas. Comme ces derniers manipulent des données en mémoire principale, ils ne peuvent alors trier des données dont la taille dépasse celle de l'espace alloué en mémoire principale [30, 41].

Le tri des volumes de données qui dépassent l'espace alloué en mémoire principale, se fait avec une autre catégorie d'algorithmes de tri, appelés tri en mémoire secondaire [30, 41]. Ces algorithmes de tri fonctionnent de manière différente des algorithmes de tri en mémoire principale. Ils trient des parties de données (voir la définition 5) en mémoire principale, qui sont ensuite écrites dans des fichiers intermédiaires. Ces parties de données sont ensuite fusionnées pour fournir le résultat final complètement trié en mémoire secondaire.

Définition 5. *Une partie de données est un sous ensemble de données contiguës dans le fichier en entrée dont la taille est inférieure à celle de l'espace mémoire alloué [30, 41].*

Dans le cadre de cette thèse, nous nous intéressons au traitement de grands volumes de données qui ne tiennent pas en mémoire principale. C'est la raison qui nous a conduit à étudier le tri en mémoire secondaire.

Nous trouvons dans la littérature plusieurs algorithmes de tri en mémoire secondaire. Parmi eux, deux sont très utilisés dans les environnements de base de données et sont intégrés dans plusieurs SGBD. Il s'agit du tri par fusion et du tri par sélection et remplacement. Nous allons présenter ces deux algorithmes dans les sections suivantes.

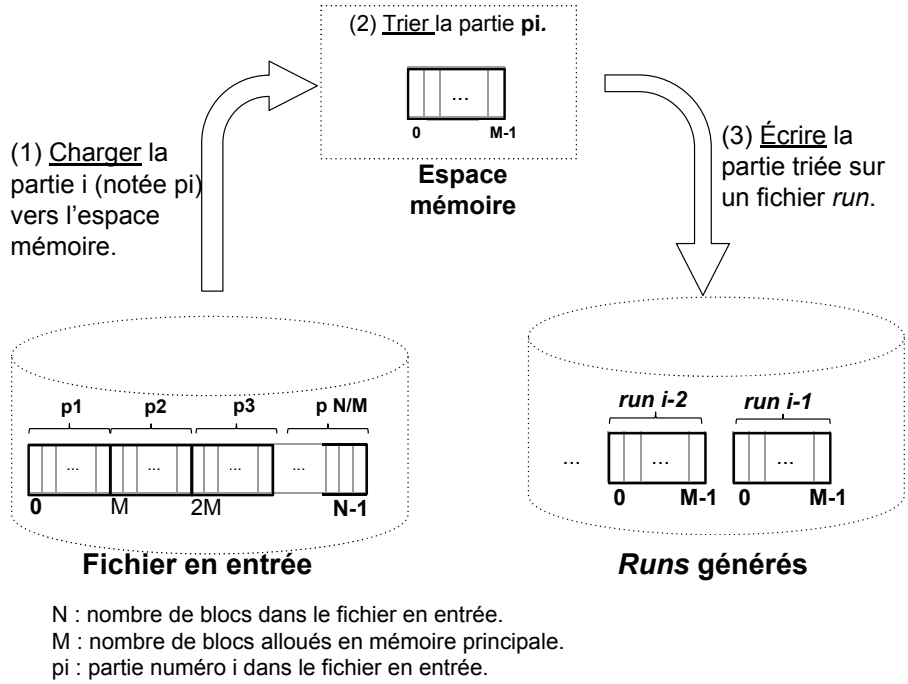

FIGURE 3 : Illustration de la génération des *runs*

2.2.1 *Algorithme de tri par fusion*

C'est l'algorithme le plus répandu pour trier des données en mémoire secondaire [41]. Cet algorithme se déroule en deux phases. Une première phase génère des fichiers triés qui tiennent en mémoire principale nommés *runs* (voir la définition 6). La deuxième phase de fusion des *runs* consiste à regrouper toutes les données issues des *runs* afin de construire le fichier final trié.

Définition 6. *Un run est un fichier intermédiaire généré par un algorithme de tri en mémoire secondaire. Le fichier contient une partie de données triées du fichier en entrée [30, 41]. Il est écrit en mémoire secondaire et sa taille dépend de l'algorithme utilisé.*

2.2.1.1 *Phase de génération des runs*

Le fichier en entrée est divisé en plusieurs parties qui font la taille de la mémoire principale allouée par le SGBD pour trier le fichier en entrée. Les parties sont traitées de manière successive par l'algorithme. Chaque partie de données est chargée en mémoire principale (voir l'étape (1) dans Figure 3), puis triée en utilisant un algorithme de tri en mémoire (voir l'étape (2) dans Figure 3). Une fois qu'une partie de données est triée, le résultat est écrit dans un *run* (voir l'étape (3) dans Figure 3).

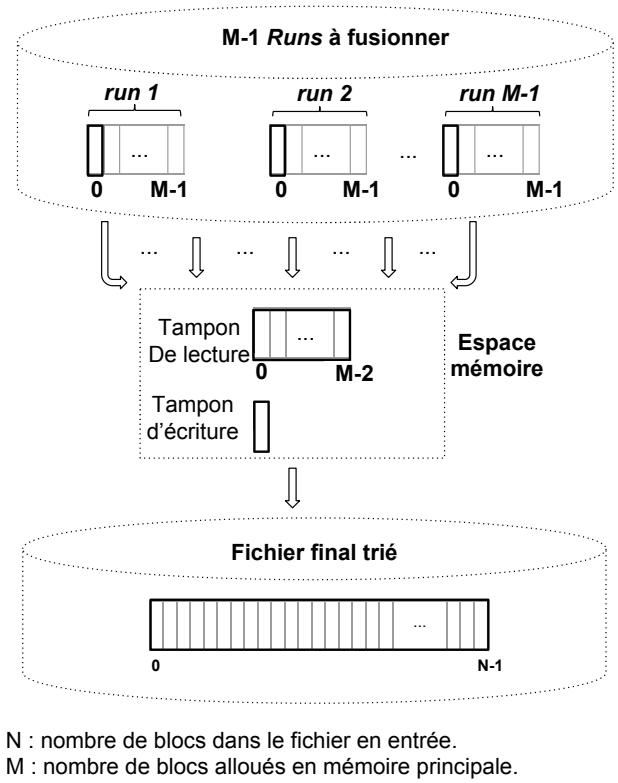

FIGURE 4 : Illustration de la fusion des *runs*

A la fin de la phase de génération des *runs*, toutes les parties de données sont transformées en *runs* contenant des données triées dont la taille est inférieure ou égale à celle de la mémoire principale allouée par le SGBD.

2.2.1.2 *Phase de fusion des runs*

La phase de fusion des *runs* prend en entrée les *runs* générés durant la phase de générations des *runs*. Ces *runs* sont fusionnés progressivement et les données sont insérées dans le fichier final trié. Cette phase est illustrée dans la Figure 4.

Durant la phase de fusion des runs, l'espace mémoire alloué est organisé en deux sections, une section pour les tampons de lecture et une section pour le tampon d'écriture. Les tampons de lecture sont utilisés pour contenir les données des runs. Le tampon d'écriture est utilisé pour contenir les données à écrire sur le fichier final trié.

Afin de fusionner les *runs*, il est nécessaire de charger un bloc de données (voir la définition 7) en mémoire principale à partir de chacun des *runs* générés. Ces blocs de données sont chargés dans des tampons de lecture et ensuite insérés dans une structure de données de type tas, qui permet de récupérer la valeur minimale sur l'ensemble des blocs avec une complexité de $O(1)$. L'algorithme de fusion des *runs* itère sur cette structure de données jusqu'à récupération des valeurs minimales de toutes les données. Lorsque toutes les valeurs d'un bloc

dans la structure de données sont traitées, le bloc est remplacé par son bloc suivant dans le *run* auquel il appartient.

Définition 7. *Un bloc de données est un ensemble de taille fixe contenant des valeurs contigües dans un fichier [30, 41].*

2.2.2 *Le tri par sélection et remplacement*

Cet algorithme passe par les mêmes étapes que l'algorithme de tri par fusion. La différence réside dans le mécanisme de génération des *runs* qui permet de créer des *runs* plus larges qui peuvent faire plusieurs fois la taille de la mémoire principale allouée par le SGBD. En créant des *runs* plus large, le nombre de *runs* générés est réduit, par conséquent la phase de fusion est moins complexe en termes d'opérations CPU [46].

Le mécanisme de génération des *runs* utilise un tas comme structure de données. Une fois le tas alloué en mémoire principale, l'algorithme insère des valeurs lues à partir du fichier en entrée. L'algorithme arrête les insertions de nouvelles valeurs une fois l'espace mémoire complètement occupé par le tas. Une fois les valeurs insérées et organisées dans le tas, l'algorithme retire une Valeur Minimale (VM) pour l'insérer dans un fichier *run* en mémoire secondaire. Cette valeur est remplacée dans le tas avec une Nouvelle Valeur (NV) lue à partir du fichier en entrée. Deux cas se présentent pour la nouvelle valeur insérée :

- **Cas 1 :** si la nouvelle valeur NV est supérieure ou égale à la dernière valeur écrite dans le *run* actuel VM ($NV \geqslant VM$), elle est insérée dans le tas pour intégrer le *run* courant (voir les étapes 3 et 4 dans la Figure 5).
- **Cas 2 :** si la nouvelle valeur NV est inférieure à la dernière valeur écrite dans le *run* VM ($NV < VM$) , cette valeur ne peut plus être insérée dans le fichier *run* courant, elle est donc destinée au prochain *run* (voir l'étape 5 dans la Figure 5).

Ce processus se répète tant que de nouvelles valeurs peuvent être insérées dans le *run* courant.

Ainsi, cet algorithme permet de générer des *runs* plus larges. Avec des données aléatoires, la taille des *runs* peut atteindre deux fois la taille de la mémoire principale allouée par le SGBD [30, 41].

Cet algorithme est avantageux lorsque les données sont partiellement triées ou complètement triées dans le bon ordre [30, 41].

Le mécanisme de fusion des *runs* est exactement le même que celui utilisé dans l'algorithme de tri par fusion. Néanmoins, en réduisant le nombre de *runs* géné-

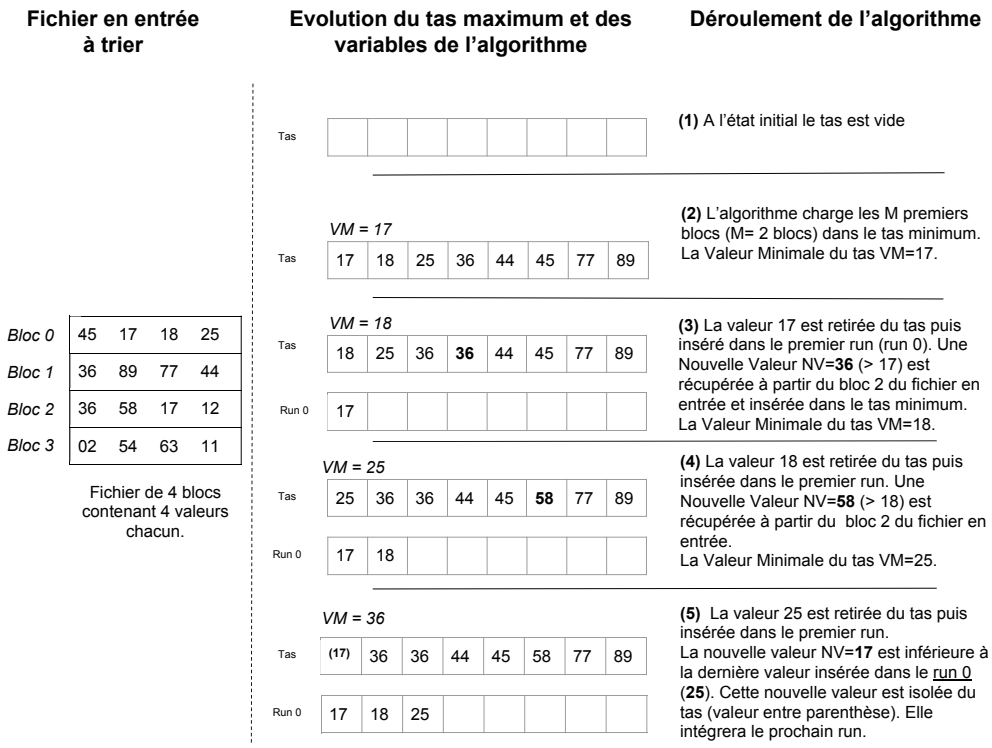

FIGURE 5 : Déroulement du tri avec l'algorithme de remplacement et de sélection

rés, l'algorithme de remplacement et de sélection réduit le coût de l'opération de fusion des *runs*.

Dans la section qui va suivre, nous décrivons un mécanisme de gestion et d'optimisation des E/S qui permet de réduire le goulet d'étranglement des accès aux données lors des opérations de traitement et de manipulation des données.

2.3 GESTION DES E/S

Les performances des algorithmes de traitement et de manipulation de données dépendent des accès aux données en lecture et en écriture. Ces accès aux données se font sur différents niveaux de la hiérarchie mémoire. Un accès sur des mémoires du niveau supérieur de la hiérarchie (registres CPU, cache CPU et mémoire principale) est nettement plus performant que les accès vers les mémoires du niveau inférieur (mémoire secondaire).

Il est commun d'utiliser les mémoires du niveau supérieur de la hiérarchie mémoire pour enregistrer les données fréquemment sollicitées. Pour ce faire, des mécanismes de mise en tampon des données sont mis en œuvre sur différents niveaux de la hiérarchie mémoire. Leur mission consiste à détecter et stocker des données fréquemment utilisées et se trouvant initialement dans les mémoires de niveau inférieur de la hiérarchie mémoire.

En plus de détecter les données fréquemment utilisées, les mécanismes de mise en tampon se basent aussi sur des mécanismes de pré-chargement de données, dont la mission consiste à prédire les prochains accès sur les mémoires de niveau inférieure de la hiérarchie mémoire et anticiper leur lecture sur l'espace tampon (*Cache*).

Dans cette section, nous décrivons les mécanismes d'accès aux données stockées en mémoire secondaire, soit le niveau le plus bas de la hiérarchie mémoire. Nous décrivons ensuite les mécanismes de mise en tampon des données au niveau de la mémoire principale, ainsi que les mécanismes de pré-chargement de données à partir de la mémoire secondaire.

2.3.1 *Mécanisme d'accès aux données sur mémoire secondaire*

Les données stockées en mémoire secondaire, sont organisées dans des fichiers. Les données dans ces fichiers sont organisées par le système d'exploitation en blocs, appelés aussi pages. La taille d'un bloc dans le système Linux est de 4Ko. Nous décrivons dans les sections ci-dessous le déroulement des opérations de lecture et écriture sur un fichier en mémoire secondaire.

2.3.1.1 *Déroulement d'une opération de lecture*

Une opération de lecture sur le fichier F est déclenchée lorsqu'un programme demande une valeur sauvegardée dans le fichier (voir (1) dans Figure 6). L'opération de lecture est lancée par le système de fichiers au niveau du bloc contenant la valeur demandée (voir (2) dans Figure 6). Ce bloc est alors chargé en mémoire principale (voir (3) dans Figure 6) et la valeur est enfin récupérée par le programme (voir (4) dans Figure 6).

Une fois qu'un bloc est chargé en mémoire principale, les demandes de valeurs appartenant au même bloc seront directement satisfaites à partir du bloc chargé en mémoire principale, sans déclencher d'opération de lecture sur la mémoire secondaire.

2.3.1.2 *Déroulement d'une opération d'écriture*

Lorsqu'un programme demande l'écriture d'une valeur dans le fichier F (voir (1) dans Figure 7), celle-ci est d'abord insérée dans un espace tampon en mémoire principale alloué pour le fichier F (voir (2) dans Figure 7). Cet espace tampon est de même taille qu'un bloc dans le fichier, il peut contenir jusqu'à B valeurs. Une fois l'espace tampon plein, il est écrit dans le fichier F par le système d'exploitation (voir (3) dans Figure 7).

L'écriture d'une valeur n'engendre pas forcément une opération d'écriture sur la mémoire secondaire.

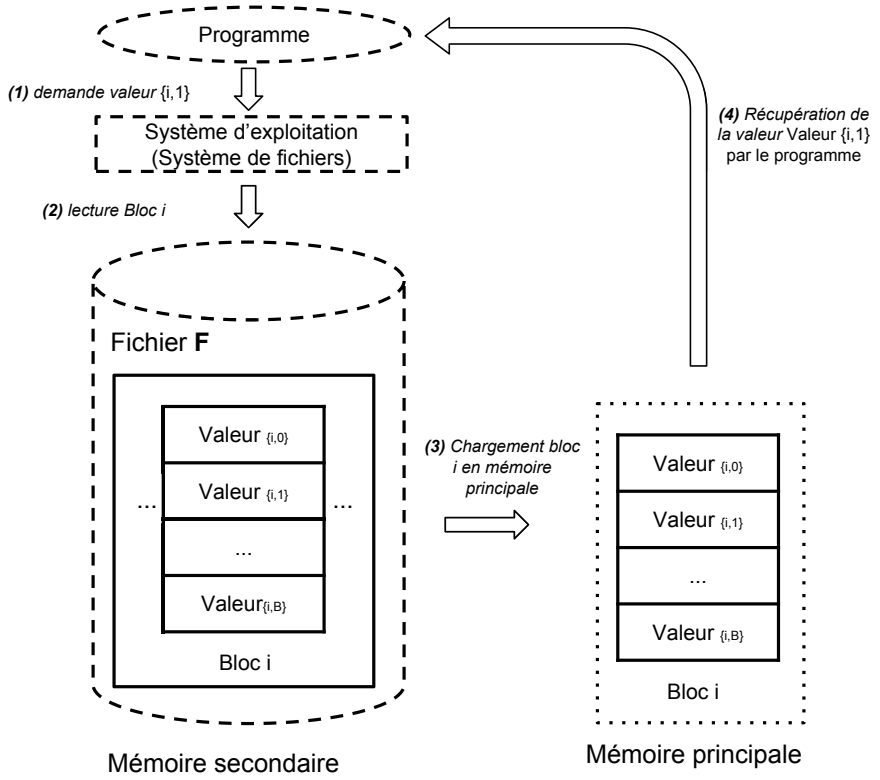

FIGURE 6 : Illustration d'une opération de lecture déclenchée par une demande de valeur (avec $Valeur_{i,j}$ la j^{eme} valeur dans le i^{eme} bloc).

2.3.1.3 *Mécanisme de mise en tampon des données*

Les systèmes d'exploitations et logiciels de traitement de données utilisent des mécanismes de mise en tampon des données pour réduire le nombre d'accès à la mémoire secondaire. Ces mécanismes gardent dans une zone tampon de la mémoire principale les données fréquemment demandées par un programme.

Avant de lancer une opération de lecture en mémoire secondaire, le système d'exploitation ou logiciel de traitement de données vérifient d'abord si cette donnée est présente dans la zone tampon en mémoire principale. Si la donnée est présente, elle est récupérée simplement de la zone tampon évitant ainsi une opération de lecture sur la mémoire secondaire. Cette situation est désignée dans la littérature par le terme Hit. Lorsque la donnée n'est pas présente dans la zone tampon, une opération de lecture est lancée sur la mémoire secondaire. Cette situation est désignée dans la littérature par le terme Miss.

Un mécanisme de mise en tampon des données est mis en œuvre dans le noyau GNU Linux. Il est intégré dans son système de fichiers virtuel VFS (Virtual File Système) et il porte le nom de *Page cache*.

Les logiciels de base de données mettent également en œuvre des mécanismes de mise en tampon spécifiques à leurs besoins [27]. C'est le cas par exemple de PostgreSQL.

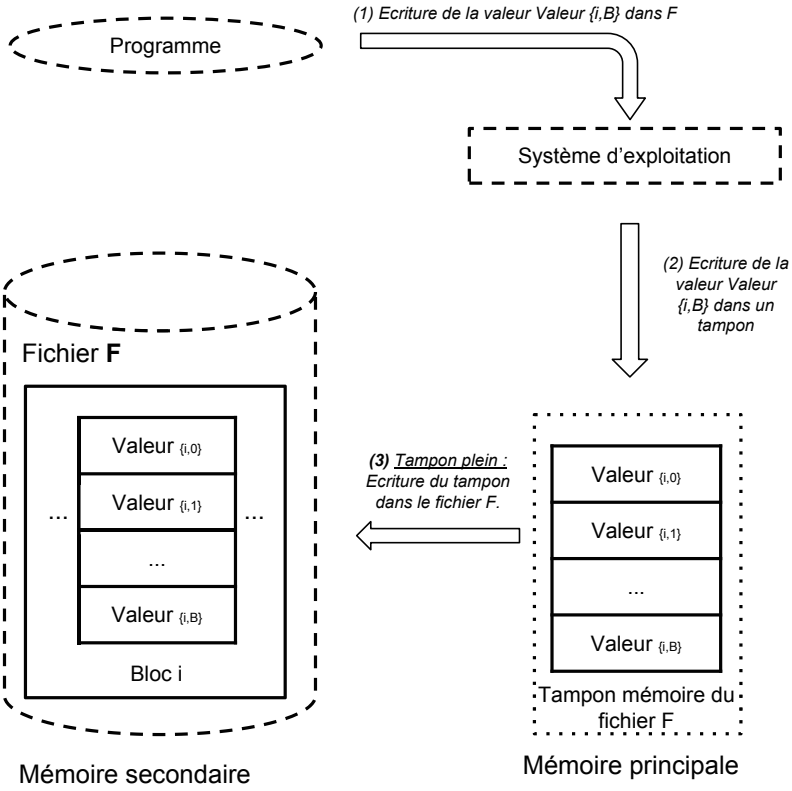

FIGURE 7 : Illustration d'une opération d'écriture

2.3.1.4 *Mécanisme de pré-chargement des données*

Les mécanismes de pré-chargement de données sont utilisées avec les mécanismes de mise en tampon des données, dans le but de réduire le nombre de Miss sur les accès aux données. Pour ce faire, ces mécanismes se basent sur une stratégie de prédiction pour anticiper les prochaines demandes de lecture et lancer des opérations de pré-chargement de données en utilisant des lectures asynchrones (voir la définition 9) [27]. Ces données une fois chargées à partir de la mémoire secondaire, sont gardées dans la zone tampon en mémoire principale. Cette technique est appelée *prefetching* (en Anglais). Dans le cadre de ce document nous parlerons de techniques de pré-chargement de données.

Définition 8. *Une opération de lecture synchrone est une opération d'E/S bloquante [15]. Le résultat de l'opération de lecture est attendu par un programme.*

Définition 9. *Une opération de lecture asynchrone est une opération d'E/S non bloquante [15]. Le résultat de l'opération de lecture n'est pas attendu par un programme. Il reçoit néanmoins une notification dès que la lecture est terminée. Ces opérations sont utilisées pour entrelacer les opérations de lecture avec d'autres opérations (de calcul par exemple).*

Nous trouvons dans la littérature quelques études menées sur les techniques de pré-chargement de données en mémoire secondaire. Parmi ces techniques de

pré-chargement, nous trouvons deux familles de techniques de pré-chargement, celles conçues pour des algorithmes spécifiques (algorithmes de tri en mémoire secondaire, algorithmes de jointures, ...) et celles conçues pour des séquences d'accès spécifique.

2.3.2 *Pré-chargement de données dédié à un algorithme spécifique*

Cette famille de technique de pré-chargement est conçue pour optimiser les E/S d'un algorithme de traitement de données en prédisant ses opérations de lecture en mémoire secondaire. En effet, certains algorithmes peuvent prédire intrinsèquement les prochaines données à lire. C'est le cas de l'algorithme de fusion des *runs* présenté dans la section 2.2.1.2 où les opérations de lecture peuvent être prédites en récupérant les valeurs minimales suivantes dans le tas.

Parmi ces techniques de pré-chargement nous retrouvons celle présentée dans [20]. Cette dernière s'intègre aux algorithmes de jointures et permet de prédire et pré-charger les données suivantes à lire.

2.3.3 *Pré-chargement de données dédié à une séquence d'accès aux données*

Cette famille de techniques de pré-chargement est conçue pour prédire et pré-charger des séquences d'accès régulières. En effet, certains algorithmes de traitement de données accèdent aux données en mémoire secondaire en suivant des séquences régulières. C'est notamment le cas de l'accès séquentiel. L'avantage de l'accès séquentiel aux données en mémoire secondaire, c'est qu'il annule le temps de déplacements mécaniques dans les périphériques de stockage magnétiques, réduisant ainsi les temps d'accès aux données. La prédiction des données à pré-charger dans ce cas se base sur la localité spatiale. Lorsqu'une donnée est demandée, la technique de pré-chargement anticipe la lecture des données suivantes dans le fichier en mémoire secondaire.

Parmi ces techniques de pré-chargement, nous retrouvons celle intégrée dans le noyau Linux (depuis sa version 2.4) appelée *read-ahead*. Cette dernière se base sur la localité spatiale des accès pour prédire et pré-charger les données dans un espace mémoire tampon. Le *read-ahead* utilise une fenêtre de prédiction de 128Ko pour anticiper et pré-charger les données. Cette fenêtre est divisée par deux lorsque la prédiction est jugée peu précise par le noyau Linux.

Le mécanisme de *read-ahead* est configurable à partir d'un programme de l'espace utilisateur à travers les appels systèmes `madvise` et `fadvise`. Le programme peut activer ou désactiver le *read-ahead* et aussi augmenter le nombre de blocs de données à prédire et à pré-charger.

Exemple 2. *Dans la Figure 8, nous avons un fichier avec* N *blocs de données, et un programme qui demande le bloc 2 (voir (1) dans la Figure 8). Le système Linux exécute*

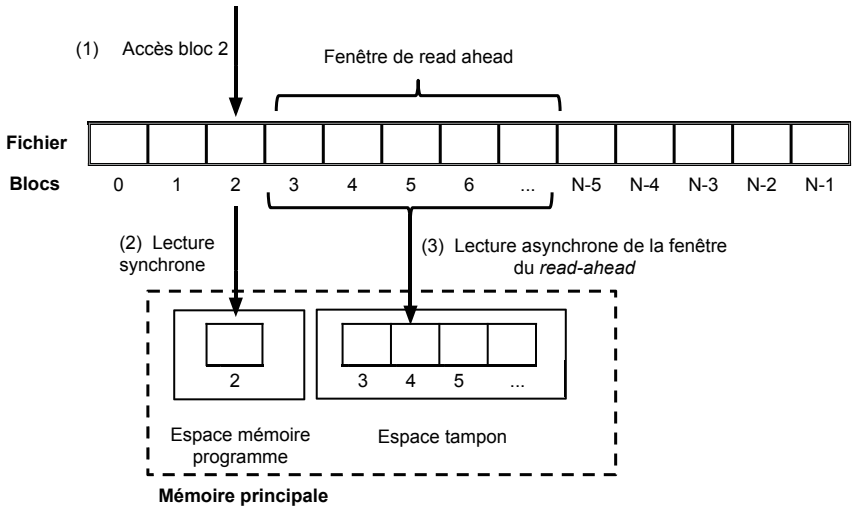

FIGURE 8 : Illustration du mécanisme de read-ahead

une opération de lecture synchrone (voir la définition 8) sur le bloc 2 afin de satisfaire la demande du programme (voir (2) dans la Figure 8). En plus de satisfaire la demande du programme, il anticipe les lectures suivantes sur le fichier. Pour ce faire, il lance une opération de lecture asynchrone sur une fenêtre de blocs adjacents et les charge dans un espace tampon en mémoire principale (voir (3) dans la Figure 8).

2.4 CONCLUSION

Un SGBD embarque une collection d'opérations de traitement de données, qui permettent de résoudre des requêtes sur les données. Il s'avère que les opérations de tri de données sont très fréquemment sollicitées par les SGBD. C'est une raison qui nous a poussé à en faire un sujet d'étude dans ce travail de thèse.

L'accès aux données est un paramètre important à considérer pour l'amélioration des performances des opérations de traitement de données. Pour ce faire, les SGBD se basent sur une politique de mise en tampon des données fréquemment utilisées et réduisent ainsi le nombre d'accès en mémoire secondaire. Avec les caractéristiques matérielles actuelles, de nouvelles politiques de mise en tampon et d'anticipation des lectures peuvent être mises en œuvre. C'est la raison qui nous a conduit à étudier des stratégies de lecture anticipée et de pré-chargement de données.

Cette thèse s'inscrit dans cet élan qui consiste à étudier les performances des SGDB pour les nouvelles caractéristiques matérielles et les nouveaux besoins en volumes de données. Pour cela, nous considérons les caractéristiques des nouvelles technologies de stockage, notamment les mémoires flash qui feront l'objet du chapitre suivant.

Chapitre 3

LES SUPPORTS DE STOCKAGE À BASE DE MÉMOIRE FLASH NAND

Un support de stockage est un périphérique permettant à un système informatique de sauvegarder des données et d'assurer leur persistance dans le temps. Ces périphériques intègrent des mémoires non-volatiles capables de maintenir les données sauvegardées sans source d'alimentation externe.

Les mémoires non-volatiles sont classées en plusieurs catégories caractérisées par leurs propriétés d'accès en écriture et en effacement [13]. Nous listons ci-dessous 5 catégories de mémoires non-volatiles les plus répandues.

- Mémoire ROM (*Read Only Memory*) qui ne sont plus modifiables après écriture.
- Mémoire PROM (*Programmable Read Only Memory*) qui sont programmables une seule fois.
- Mémoire EPROM (*Erasable Programmable Read-Only Memory*) effaçable par rayonnement ultra violet.
- Mémoire EEPROM (*Electrically Erasable Programmable Read-Only Memory*) effaçable par application d'une charge électrique.

Dans ce chapitre, nous décrivons les mémoires flash qui appartiennent à la famille des mémoires EEPROM. Nous présentons ensuite les supports de stockage SSD (*Solid State Drive*) basés sur des mémoires flash. Nous entamons le chapitre en donnant des généralités sur les mémoires flash. Nous présentons ensuite la structure d'une mémoire flash, les opérations d'E/S supportées et leurs contraintes. Enfin, les supports de stockages SSD, leurs architectures et leurs performances sont abordées.

3.1 GÉNÉRALITÉS SUR LES MÉMOIRES FLASH

Nous distinguons deux types de mémoire flash. En fonction de la porte logique utilisée dans ses cellules mémoires [13], les mémoires flash NOR et les mémoires flash NAND.

L'architecture des cellules de mémoire flash de type NOR permet d'adresser les données avec la granularité d'un octet. Ce type de mémoire offre des performances élevées en lecture, soit une latence de 20ns pour la lecture d'une page [56]. Les opérations d'écritures sont moins performantes que celles de lectures, soit une latence de 100ns pour l'écriture d'une page [56]. Les mémoires flash de type NOR sont utilisées dans les mémoires centrales de quelques systèmes embarqués, PC de bureau et serveurs.

Les mémoires flash de type NAND adressent les données avec la granularité d'une page (un groupe de cellules mémoires ayant une taille fixe). Ce type de mémoire flash présente des latences en lecture plus élevées que celles des mémoires flash NOR, soit 25us pour la lecture d'une page. Ceci est principalement dû au fait que la densité des cellules NAND dans une mémoire flash est plus importante que celle des mémoires NOR. Ces mémoires sont principalement utilisées pour le stockage de données. Elles sont très répandues actuellement et sont utilisées comme mémoire secondaire dans des systèmes embarqués, les ordinateurs de bureau (PC) et les serveurs. Elles équipent, par exemple, les périphériques de stockage SSD *(Solid State Drive)*, les clés de stockage USB et les cartes de stockage SD.

Dans cette thèse, nous étudions les algorithmes de traitement de données sur les supports de stockage à base de mémoire flash, nous nous intéressons uniquement aux mémoires flash ayant des cellules NAND.

Les cellules de mémoires flash se répartissent en trois familles : les cellules SLC (*Single Level Cell*), les cellules MLC (*Multi Level Cell*) et les cellules TLC (*Triple Level Cell*). Celles-ci se distinguent en termes de densité de stockage, d'endurance, de performance et de fiabilité.

Dans une cellule SLC, la densité est de 1 bit par cellule, alors que les cellules MLC et TLC stockent 2 et 3 bits respectivement. Si la densité de stockage est plus élevée avec les cellules MLC et TLC, les cellules SLC sont plus avantageuses en termes de performance, d'endurance et de fiabilité.

3.2 STRUCTURE D'UNE MÉMOIRE FLASH

Une mémoire flash est structurée de manière hiérarchique. Au plus haut niveau nous retrouvons un certain nombre de dés. Ces dés contiennent des plans. Les plans contiennent des blocs et ces derniers contiennent des pages. Nous illus-

trons la structure d'une mémoire flash dans la Figure 9 et nous décrivons ci-dessous ses différents composants.

3.2.1 *Page*

Une page représente la granularité avec laquelle les opérations de lecture et écriture sont effectuées. La taille des pages varie en fonction du type de la puce NAND (SLC, MLC, TLC). Les pages ayant des cellules SLC présentent des tailles variables (à partir de 2048 octets) [14]. Pour les pages ayant des cellules MLC, elles peuvent présenter des tailles de 2048, 4096 ou 8192 octets [14].

3.2.2 *Bloc*

Un bloc se compose d'un certain nombre de pages. Ce nombre est une puissance de 2 et dépend du modèle de la puce NAND [14]. Un bloc peut contenir, par exemple, 32 pages, 64 pages ou 128 pages.[14].

3.2.3 *Plan*

Le nombre de plans contenus dans une puce NAND dépend du modèle de cette dernière. Les puces peuvent contenir un ou deux plans [31] et jusqu'à quatre plans [36]. Le nombre de blocs dans un plan dépend aussi du modèle de la puce NAND. [35] présente une puce NAND de 128 Mo contenant un seul plan qui regroupe 8192 blocs.

3.2.4 *Dé*

Une puce de mémoire flash peut contenir un ou plusieurs dés. Chaque dé est constitué d'un ou de plusieurs plans (en général un ou deux plans). Lorsqu'une puce contient plusieurs dés, elle est appelée **packet**. Nous retrouvons alors des **packets** de type *dual die* contenant deux dés et des **packets** de type *quad die* contenant quatre dés.

3.3 OPÉRATIONS D'ACCÈS À LA MÉMOIRE FLASH

Une mémoire flash supporte trois opérations de base : **(1) les opérations de lecture**, **(2) les opérations d'écriture** et **(3) les opérations d'effacement**.

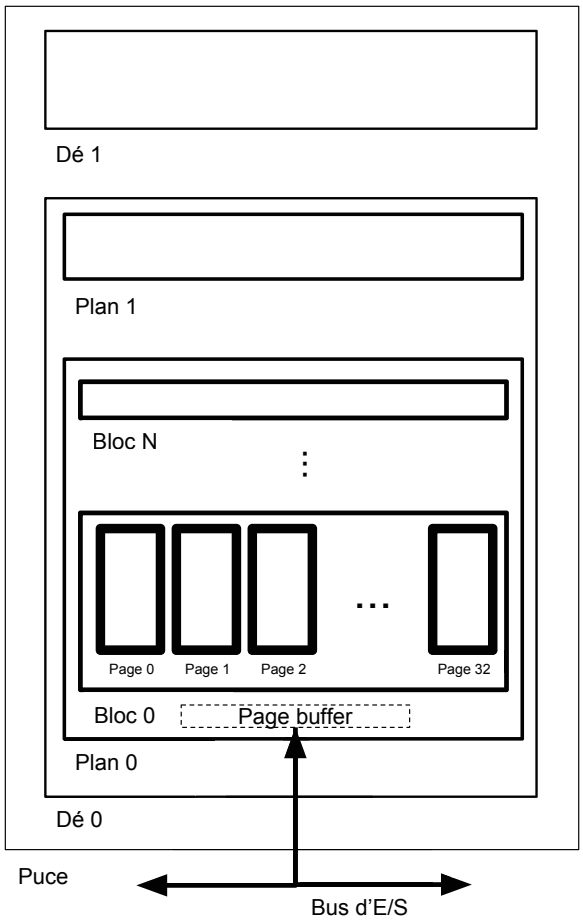

FIGURE 9 : Structure interne hiérarchique d'une mémoire flash

3.3.1 *Opération de lecture*

Une opération de lecture est appliquée sur une page. Elle consiste à charger les données d'une page à partir de la mémoire flash pour les placer dans un espace tampon au niveau du plan qui est appelé *page buffer*. Les données placées dans le tampon (*page buffer*) sont envoyées à travers le bus d'E/S vers le contrôleur, puis transmises à l'hôte.

Le temps d'une opération de lecture est donné par la somme du temps de transfert des données de la matrice NAND vers le tampon (*page buffer*) (voir la Figure 9) et le temps de transfert sur le bus d'E/S. Le temps de lecture d'une page varie en fonction du type de cellules de mémoire flash. Les auteurs de [31] indiquent un temps de lecture de 20µs sur une puce SLC et de 40µs sur une puce MLC.

3.3.2 *Opération d'écriture*

Une opération d'écriture est appliquée sur une page. Elle consiste à écrire des données envoyées par l'hôte dans une page de la mémoire flash. Les données sont transmises à travers le bus d'E/S vers la puce flash. Puis, elles sont placées dans le tampon *page buffer* du plan et ensuite écrites sur une page de la matrice NAND.

Le temps d'une opération d'écriture est donné par la somme du temps de transfert des données sur le bus d'E/S et le temps nécessaire à l'écriture de ces données sur une page de la matrice NAND. De même que le temps d'une opération de lecture, le temps d'écriture d'une page dépend aussi du type de cellules de mémoires flash. Les auteurs de [31] indiquent un temps d'écriture de 300us sur une puce SLC et entre 300us et 1500us sur une puce MLC.

Les opérations d'écriture présentent une forte variation sur les mémoires NAND ayant des puces MLC. Cette variation est due à la microarchitecture des puces MLC, ainsi qu'au processus de programmation des cellules MLC [14].

3.3.3 *Opération d'effacement*

Une opération d'effacement est appliquée sur un bloc. Elle consiste à effacer toutes les pages appartenant à un bloc de données. Une page ne peut être effacée individuellement.

Le temps d'une opération d'effacement se résume au temps nécessaire pour l'effacement d'un bloc de données. Il est supérieur au temps de lecture et d'écriture (jusqu'à 10 fois plus grand pour les temps d'écriture et jusqu'à 100 fois plus grand pour les temps de lecture [31]). Cela est principalement dû au fait que cette opération s'applique sur la granularité d'un bloc. Ainsi le nombre de cellules mémoires traitées est plus important.

3.4 CONTRAINTES LIÉES À L'UTILISATION DES MÉMOIRES FLASH

La micro-architecture des mémoires flash et le fonctionnement interne des cellules flash imposent quelques contraintes sur les opérations d'E/S. Ces contraintes sont classées en trois catégories : **(C1)** une donnée écrite sur une cellule mémoire flash ne peut être mise à jour ; **(C2)** les cycles d'écriture/effacement réduisent la durée de vie des cellules mémoire flash ; et **(C3)** le manque de fiabilité des opérations de lecture/écriture (risque de *bit flipping*). Nous détaillons ces contraintes dans les sections suivantes.

3.4.1 *Contrainte C1 : la mise à jour de données*

Seules les trois opérations de lecture, écriture et effacement peuvent être appliquées individuellement sur une cellule de mémoire flash. Une opération de mise à jour d'une donnée individuelle déjà écrite sur une cellule est techniquement impossible. Ainsi, la mise à jour d'une page implique des opérations intermédiaires sur la mémoire flash que nous résumons dans les points ci-dessous :

1. Allouer un nouveau bloc, et écrire la donnée mise à jour dans le nouveau bloc.
2. Copier les pages valides de l'ancien bloc (pages non mises à jour) dans le nouveau bloc.
3. Effacer l'ancien bloc.

3.4.2 *Contrainte C2 : l'usure*

Une cellule de mémoire flash ne peut supporter qu'un nombre limité d'opérations d'effacement. Lorsque cette limite est atteinte, la cellule devient inutilisable. La durée de vie d'une cellule flash est généralement donnée par le nombre maximal de cycles d'effacement. Ce dernier dépend du type de la cellule NAND. Il est de l'ordre de 10^4 cycles pour les cellules NAND de type SLC, 10^3 cycles pour les cellules NAND de type MLC à deux niveaux et 5000 pour les cellules de type TLC [14].

3.4.3 *Contrainte C3 : la fiabilité*

Il arrive que l'intégrité des données stockées sur des cellules flash soient remise en cause par une inversion de bit causée par la structure interne de la mémoire flash et les tensions appliquées aux cellules lors des opérations d'écriture et d'effacement [16, 56].

3.5 GESTION DES CONTRAINTES

Les supports de stockage à base de mémoire flash, comme les SSD, intègrent une couche de composants logiciels et matériels qui assure la gestion des contraintes liées aux mémoires flash [14]. Nous utilisons le terme "couche de gestion" pour référencer cette couche de composants (voir définition 10).

Définition 10. *La couche de gestion permet, entre autres, d'abstraire les différentes limites et contraintes liées aux mémoires flash afin de permettre son utilisation de façon transparente [14].*

Dans cette section, nous décrivons pour chacune des contraintes (**C1, C2 et C3**), les différents services mis en œuvre par les couches de gestion afin de rendre les contraintes transparentes pour l'utilisateur.

3.5.1 *Gestion de la contrainte C1 : la mise à jour de données*

Les opérations de mise à jour de données sur mémoire flash impliquent plusieurs opérations de lecture/écriture et une opération d'effacement de bloc. Afin de réduire le coût de ces opérations, les couches de gestion écrivent la nouvelle page mise à jour dans un nouveau bloc et marquent l'ancienne page comme invalide [14]. Ainsi le bloc n'est pas effacé et les autres pages du bloc ne sont pas déplacées vers un nouveau bloc. Des opérations d'effacement périodiques sont lancées sur les blocs constitués uniquement de pages invalides afin de récupérer de l'espace de stockage.

Pour mettre en place cette procédure de mise à jour des données, les couches de gestion exploitent deux mécanismes : **(1)** un mécanisme de traduction d'adresse logique et physique ; et **(2)** un mécanisme de ramasse miettes. Nous décrivons le rôle de ces mécanismes et leur fonctionnement dans les sections qui suivent.

3.5.1.1 *Mécanisme de traduction d'adresses logiques et physiques*

Le rôle du mécanisme de traduction d'adresses est de rendre transparent pour un système d'exploitation les différentes opérations lors de la mise à jour des données. Il s'agit en l'occurrence du déplacement de données entre les pages et les blocs. Pour ce faire, le mécanisme traduit des adresses logiques envoyées (voir définition 11) en adresses physiques (voir définition 12) qui donne la position exacte de la donnée demandée.

Définition 11. *Au niveau du périphérique de stockage, les adresses reçues du système d'exploitation sont appelées* ***adresses logiques*** *[60]*

Définition 12. *Une* ***adresse physique*** *désigne l'adresse d'une donnée au sein d'un périphérique de stockage [60].*

Ce mécanisme met en œuvre une table de traduction d'adresses (*mapping table*) dans la mémoire vive du support de stockage SSD. Chaque entrée de cette table associe une adresse logique avec une adresse physique.

La couche de gestion gère une opération de mise à jour en suivant les étapes ci-dessous :

1. Récupération de l'adresse logique de la page concernée par la requête de mise à jour.
2. Utilisation de la table de traduction d'adresses pour transformer l'adresse logique en adresse physique.

3. Marquage de la page correspondante à cette page physique comme invalide.
4. Ecriture d'une nouvelle page avec les données mises à jour.
5. Mise à jour de l'entrée correspondante à l'adresse logique de la requête dans la table de traduction d'adresse en l'associant avec l'adresse physique de la nouvelle page.

3.5.1.2 *Mécanisme de ramasse miettes*

Le rôle de ce mécanisme consiste à détecter les blocs ayant des pages marquées invalides suite aux mises à jour. Ces blocs sont effacés, puis disponibles pour les prochaines opérations d'écriture. Ce mécanisme est déclenché soit :

- Périodiquement.
- Lorsque le nombre de blocs libres restant est inférieur à un certain seuil.

3.5.2 *Gestion de la contrainte C2 : l'usure*

La couche de gestion implante un mécanisme qui assure une répartition équilibrée de l'usure sur l'ensemble des cellules de la mémoire flash. Pour ce faire, les opérations d'écriture/effacement sont réparties sur toute la mémoire flash afin d'éviter qu'elles se concentrent sur un même ensemble de cellules. Ce mécanisme est connu sous le nom de *wear leveling* [14].

3.5.3 *Gestion de la contrainte C3 : la fiabilité*

Les problèmes de fiabilité des mémoires flash NAND se règlent essentiellement via la mise en place de codes correcteurs d'erreurs (*Error Correcting Codes, ECC*) au niveau de la couche de gestion.

3.6 ARCHITECTURE D'UN SUPPORT DE STOCKAGE À BASE DE MÉMOIRE FLASH

Un support de stockage à base de mémoire flash SSD est constitué de deux parties : une partie contenant les composants de stockage (voir partie **A** dans la Figure 10) et une partie contenant les composants du contrôleur (voir partie **B** dans la Figure 10).

La partie stockage d'un SSD intègre plusieurs puces de mémoire flash, interconnectées avec des bus d'E/S appelés canaux. Ces canaux connectent les puces de mémoires flash avec le contrôleur.

L'intégration de plusieurs puces de mémoires flash dans un SSD apportent deux avantages à ces derniers :

1. L'augmentation de la capacité de stockage
2. La possibilité d'exécuter des opérations en parallèle sur plusieurs puces de mémoire flash.

La deuxième partie d'un support SSD intègre un contrôleur constitué d'une mémoire vive de type DRAM, d'un processeur et d'une interface de connexion logique. Ce contrôleur joue principalement le rôle d'une interface entre les composants du stockage et l'hôte qui peut être la carte mère d'un ordinateur. Il se charge d'exécuter les opérations reçues à partir de l'hôte (lecture, écriture et effacement de données), et restitue ensuite les résultats. Ce contrôleur se charge aussi de l'optimisation et de la gestion de l'espace de stockage et des opérations sur les données.

Dans la Figure 10, nous illustrons un SSD équipé de M canaux connectés à un contrôleur avec un bus d'E/S. Dans chaque canal on retrouve N puces de mémoire flash qui partagent le bus d'E/S du canal.

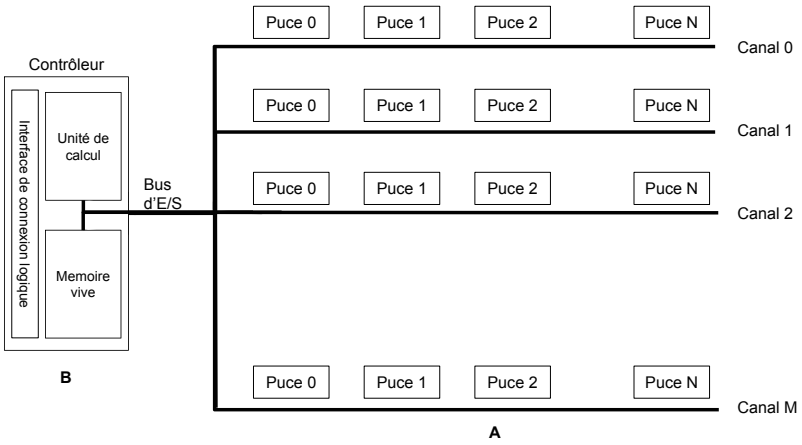

FIGURE 10 : Structure interne d'un support de stockage SSD

Les performances d'un périphérique SSD sont étroitement liées aux éléments intrinsèques et aux choix de conception. C'est la raison pour laquelle, les performances peuvent varier entre les SSD de constructeurs différents et même entre des modèles d'un même constructeur. Nous allons décrire dans les paragraphes qui suivent ces différents éléments intrinsèques qui influencent les performances des supports de stockage.

Parmi ces éléments, nous distinguons des composants matériels comme la mémoire vive, le contrôleur, l'interface d'E/S et le parallélisme et aussi des optimisations logicielles internes aux supports de stockage SSD comme la lecture anticipée des données et les schémas d'allocation.

Les composants matériels d'un SSD qui influencent ses performances sont :

1. **Interface de connexion logique :** le débit de transfert de données peut être différent selon le type d'interface de connexion logique utilisée. Une interface SATA permet de transférer des données avec un débit allant jusqu'à 6Go/s [7, 19, 34], alors qu'une interface Fiber Channel (FC) permet un débit allant jusqu'à 128Go/s [7].

2. **Parallélisme :** un SSD intègre deux niveaux de parallélisme [18] au niveau des canaux (nombre de canaux, voir la Figure 10) et au niveau des puces flash (nombre de puces par canal, voir la Figure 10).

3. **Taille de la mémoire vive :** Un SSD intègre une mémoire vive utilisée comme zone tampon et permet d'accueillir les données les plus utilisées et absorber quelques opérations d'écriture. Àtitre d'exemple, le SSD **850 EVO SATA III 2.5inch SSD 4TB** [3] intègre 4 Go de mémoire vive.

Enfin, les mécanismes logiciels internes au support de stockage SSD qui influencent ses performances sont :

1. **Lecture anticipée des données :** ce mécanisme permet d'accélérer les requêtes d'E/S séquentielles sur les données. Il détecte les séquences de requêtes séquentielles et anticipe les opérations de lecture sur les mémoires flash. Les données chargées avec les lectures anticipées sont placées dans une zone tampon de la mémoire vive interne au SSD [18].

2. **Schéma d'allocation :**

   Un schémas d'allocation a pour objectif de répartir de manière équitable les opérations d'E/S sur les différents canaux, puce flash, dé, plan, bloc et page. Nous distinguons deux stratégies d'allocation : **statique** et **dynamique**. **Le schéma statique** se base sur une équation pour sélectionner le canal, la puce flash, le dé, le plan, le bloc et enfin la page. Le **schéma dynamique** tient en compte l'état (libre/occupé) des canaux, puces et plans pour la sélection de ces derniers. Par conséquent, le schéma d'allocation dynamique est plus adapté pour des charges de requêtes d'E/S parallèle.

3. **Mapping des données :** les périphériques de stockage SSD embarquent des tables de traduction d'adresses logiques en adresses physiques. Nous retrouvons dans les SSD trois différentes stratégies de traduction d'adresses : (1) traduction d'adresses au niveau des pages, (2) traduction d'adresses au niveau des blocs et (3) traduction d'adresse hybride (pages et blocs). La traduction d'adresses au niveau des pages offre plus de flexibilité [12] mais a l'inconvénient d'être gourmande en mémoire. Le schéma de traduction d'adresses au niveau des blocs requièrent moins de mémoire que la première technique mais présente un inconvénient en terme de flexibilité. Les schémas de traduction hybrides sont proposés pour exploiter les avantages des deux premiers schémas.

3.7 CONCLUSION

Dans ce chapitre, nous avons brièvement présenté les mémoires flash et les supports de stockage à base de mémoire flash (SSD). Nous avons commencé par décrire la structure interne des puces mémoires flash de type NAND que nous considérons dans le cadre de ce travail de thèse. Nous avons étudié les différentes opérations supportées par ce type de cellules flash, les contraintes et limites et avons enfin décrit les solutions mises en œuvre pour la gestion de ces contraintes. Nous avons aussi présenté l'architecture interne des supports de stockages SSD et décrit comment certains éléments intrinsèques aux SSD influencent leurs performances.

Nous constatons à partir de l'étude des opérations sur mémoire flash que celles-ci offrent un nouveau modèle de performance qui rassemble les propriétées (**P1**), (**P2**) et (**P3**) présentées ci-dessous :

- **P1** Symétrie des performances des opérations de lecture : les performances des lectures aléatoires et séquentielles sont équivalentes.
- **P2** Asymétrie des performances des opérations d'écriture : les performances des écritures sont asymétriques avec celles des lectures.
- **P3** Asymétrie des performances des opérations d'effacement : les performances des effacement sont asymétriques avec celles des écritures et lectures.

Ce nouveau modèle de performances offre de nouvelles opportunités d'optimisation pour les logiciels de traitement et de gestion de données comme les SGBD. Si la propriété (**P1**) permet d'utiliser les lectures aléatoires au même titre que les lectures séquentielles, les propriétés (**P2**) et (**P3**) incitent à générer moins de données pour réduire les opérations d'écriture et d'effacement de données.

Les mémoires flash présentent trois contraintes : (**C1**), (**C2**) et (**C3**). Celles-ci posent des limites techniques aux requêtes d'E/S, notamment les requêtes de mise à jour de données. Pour résoudre les problèmes issues de ces contraintes, les supports de stockage (SSD) mettent en œuvre des mécanismes dédiés à la gestion de ces contraintes.

Les performances des supports de stockage SSD, en plus d'être associées aux performances des mémoires flash, sont étroitement liées à l'architecture et à la structure interne du support de stockage. Àtitre d'exemple, la mémoire vive intégrée aux supports de stockage SSD peut absorber l'asymétrie des performances entre les lectures et les écritures si les données sont déjà placées dans la zone tampon interne du SSD.

Dans le chapitre suivant, nous décrivons quelques optimisations logicielles qui tiennent compte des propriétés (**P1**), (**P2**) et (**P3**) du modèle de performance des

mémoires flash et des contraintes (**C1**), (**C2**) et (**C3**). Ces optimisations logicielles sont mises en œuvre pour plusieurs buts. Nous citons par exemple :

- L'utilisation des lectures aléatoires au même titre que les lectures séquentielles.
- La suppression de l'asymétrie des opérations d'écriture et d'effacement.
- La réduction des opérations de mise à jour.

Chapitre 4

ETAT DE L'ART CONCERNANT L'OPTIMISATION DES E/S SUR LES PÉRIPHÉRIQUES SSD

Plusieurs études d'état de l'art ont exploré les performances de SSD pour réduire les coûts en E/S des logiciels de traitement de données. Dans le cadre de notre thèse, nous nous focalisons sur les optimisations apportées aux algorithmes de tri en mémoire secondaire ainsi que les mécanismes de pré-chargement de données.

Dans ce chapitre, nous présentons et étudions dans un premier temps les algorithmes de tri en mémoire secondaire. Ensuite nous décrivons deux mécanismes de pré-chargement de données conçues pour SSD.

4.1 OPTIMISATION DES ALGORITHMES DE TRI EN MÉMOIRE SECONDAIRE POUR LES PÉRIPHÉRIQUES SSD

De nouveaux algorithmes de tri en mémoire secondaire, optimisés pour les performances des SSD, ont vu le jour récemment. Nous citons par exemple FAST(1) [55], FAST(N) [55], *Natural page run* [51], MinSort [22], FSort [9], TagSort[64] ou FMSort [48]. Ces algorithmes sont présentés dans les sections qui suivent. Les notations utilisées pour décrire ces algorithmes sont résumées dans le Tableau 1

4.1.1 *FAST(1)*

FAST [55] est un algorithme de tri en mémoire secondaire optimisé pour les mémoires flash et destiné pour les bases de données mises en œuvre dans les appareils mobiles. Par conséquent, l'algorithme considère la faible capacité mémoire et la consommation énergétique dans les appareils mobiles.

Tableau 1 : Notations

Notation	Définition
N	Taille du fichier en entrée en nombre de blocs
M	Taille de l'espace mémoire en nombre de blocs
n	Nombre de valeurs dans le fichier en entrée
m	Nombre de valeurs pouvant être chargées dans l'espace mémoire alloué

4.1.1.1 *Problématique*

Lorsque l'espace mémoire M alloué pour le tri est trop petit (comme cela peut être le cas dans les appareils mobiles), les algorithmes de tri en mémoire secondaire génèrent un nombre de *runs* $\frac{N}{M}$ élevé. SI ce nombre dépasse la quantité de blocs alloués en mémoire principale M, l'opération de fusion s'effectue en $\log_{M-2} \frac{N}{M}$ itérations. Chacune de ces itérations produit N blocs de données temporaires, ce qui se traduit en plusieurs opérations d'écriture et d'effacement sur mémoire flash.

4.1.1.2 *Conception et mise en œuvre*

L'objectif de cet algorithme de tri consiste à réduire le nombre d'itérations et la quantité de données temporaires afin de baisser le coût en opérations d'écriture et d'effacement.

Les auteurs de cet algorithme se basent sur le fait que les opérations d'écriture sur mémoire flash sont moins performantes que celles de lectures. Dans leur étude, les auteurs partent de l'hypothèse que le coût d'une opération d'écriture W est dix fois supérieur au coût d'une opération de lecture R (soit $W = 10 \cdot R$). L'algorithme FAST exploite cette asymétrie en utilisant des lectures supplémentaires pour réduire le coût en opérations d'écriture et effacement.

FAST parcourt plusieurs fois le fichier à trier. Àchaque parcours, il récupère m valeurs minimales, soit M blocs de taille B, pour les mettre directement dans le fichier final trié. Cet algorithme génère alors un fichier final trié au bout de $\frac{N}{M}$ parcours du fichier en entrée, sans qu'aucune données temporaire ne soit écrite en mémoire secondaire.

FAST utilise un tas maximum (voir la définition 13) pour trier les données.

Définition 13. *Un tas maximum est une structure de donnée organisée en arbre de recherche binaire où la recherche de la valeur maximale est réalisée avec une complexité de* $O(1)$.

L'algorithme utilise deux variables appelées K_{max} et O_{max}. La variable K_{max} stocke la valeur maximale écrite dans le fichier trié lors du parcours précédent. La variable O_{max} est utilisée pour stocker les coordonnées des valeurs de K_{max} dans le fichier en entrée.

FAST parcourt le fichier en entrée et vérifie les conditions (1) et (2) pour chacune de ses valeurs v :

1. FAST vérifie que la valeur v n'est pas encore écrite dans le fichier final trié, cela revient à vérifier que v est supérieure à K_{max} ($v > K_{max}$).

2. FAST vérifie que la valeur v n'est pas supérieure à la valeur maximale dans le tas maximum.

Lorsque ces deux conditions sont vraies pour une valeur v, FAST retire la valeur maximale du tas et la remplace avec cette nouvelle valeur v.

Àla fin de chaque parcours, FAST regroupe dans la structure de données en mémoire principale les m plus petites valeurs du fichier en entrée qui sont supérieures à K_{max}. Ces valeurs sont triées en mémoire principale et sont écrites sur le fichier final trié. La valeur maximale écrite dans le fichier final trié, est copiée dans la variable K_{max}

Nous illustrons cet algorithme dans l'exemple 3.

Exemple 3. *Dans la Figure 11, nous présentons le déroulement de FAST sur un fichier ayant* $N = 4$ *blocs de données avec un espace mémoire pouvant contenir* $M = 2$ *blocs de données. Les blocs de données contiennent 4 valeurs.*

- *L'algorithme commence par initialiser à zéro les variables* K_{max} *et* O_{max} *et un tas maximum en mémoire principale.*

- *Une fois les variables initialisées et les structures de données préparées, l'algorithme lit les* $M = 2$ *premiers blocs (*$bloc_0$ *et* $bloc_1$*) à partir du fichier en entrée et insère leurs valeurs dans le tas maximum.*

- *FAST continue la lecture des blocs suivants (*$bloc_2, ...$*). Il compare chaque nouvelle valeur à la valeur maximale dans le tas. Les valeurs* $37, 58, 16, 12, 2$ *et* 11 *de* $bloc_2$ *et* $bloc_3$ *sont inférieures aux valeurs maximales dans le tas, respectivement* $89, 77, 58, 45, 44$ *et* 37 *(voir les étapes (3), (4), (5), (6), (7) et (10) dans la Figure 11). Ces valeurs maximales sont alors retirées du tas pour laisser place aux nouvelles.*

- *Les valeurs* 54 *et* 63 *sont supérieures à la valeur maximale dans le tas (36) (voir les étapes (8) et (9) dans la Figure 11). Par conséquent, ces valeurs sont ignorées par l'algorithme.*

- *Les valeurs du tas maximum sont alors triées par ordre croissant et ensuite écrites dans le fichier final trié (voir l'étape (11) dans la Figure 11). La variable* K_{max} *prend la valeur maximale écrite sur le fichier final trié, soit* $K_{max} = 36$.

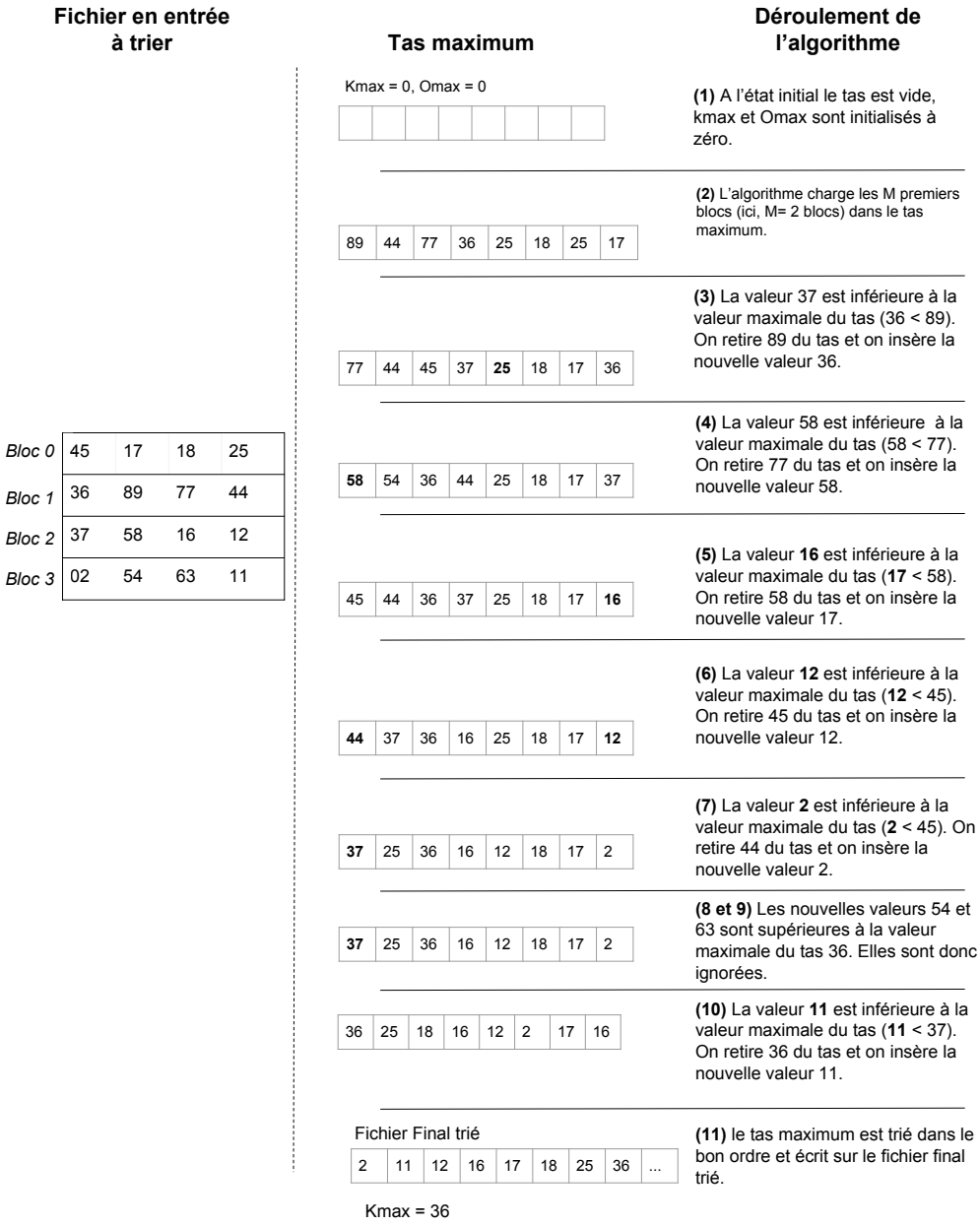

FIGURE 11 : Déroulement du tri avec l'algorithme FAST

4.1.1.3 *Analyse et discussion de l'algorithme*

FAST se base sur l'hypothèse que le coût d'une opération d'écriture W sur mémoire flash est 10 fois plus élevé qu'une opération de lecture R (soit $W = 10 \cdot R$). FAST ne considère pas les caractéristiques intrinsèques des périphériques de stockage SSD permettant de réduire les coût des opérations d'écriture.

4.1.2 *FAST(N)*

FAST(N) est une variante de l'algorithme FAST qui permet le traitement de plus grands volumes de données [55].

FAST(N) divise le fichier à trier en parties de Q blocs, avec $M < Q < N$. L'algorithme FAST est ensuite appliqué sur chacune des parties de Q blocs pour obtenir des fichiers *runs* triés. Les *runs* sont ensuite fusionnés pour obtenir le fichier final trié.

Avec FAST(N), chaque partie de données est lue $\frac{Q}{M}$ fois. Par conséquent, le coût en lecture pour la génération des *runs* est de $N \cdot \frac{Q}{M} \cdot R$. Par ailleurs les données triées sont écrites dans des fichiers *runs*. Il en résulte un coût en écriture de $N \cdot W$.

4.1.3 *Natural Page Run*

Natural Page Run est un algorithme proposé dans [51]. L'algorithme est optimisé pour trier des données partiellement triées.

4.1.3.1 *Problématique*

Les algorithmes de traitement de données traditionnels se déroulent en deux phases ; une phase de génération des runs et une phase de fusion des runs. Àla fin de la phase de génération des runs, ces algorithmes écrivent en mémoire flash N blocs de données temporaires. Ces données sont effacées à la fin de l'opération du tri une fois que le fichier final trié est complètement enregistré. Ces opérations d'écritures et d'effacement de données temporaires pénalisent les performances du tri et réduisent la durée de vie de la mémoire flash.

4.1.3.2 *Conception et mise en œuvre*

Natural page run exploite les propriétés **(P1)** et **(P2)** des mémoires flash avec les propriétés internes de l'ensemble de données à trier. Il utilise alors des lectures supplémentaires, notamment des lectures aléatoires, pour chercher des données partiellement triées dans le fichier en entrée. Ces données sont utilisées pour créer des ***runs* naturels** (voir la définition 14). Ces derniers ne sont pas écrits en mémoire secondaire, réduisant ainsi le coût en écriture et en effacement des données temporaires.

Définition 14. *Un **run naturel** est constitué d'un index en mémoire principale qui référence des blocs de données partiellement triées dans le fichier en entrée [51]. Les entrées d'index sont structurées avec les champs suivants ; Numéro du bloc dans le fichier, valeur minimale dans le bloc, valeur maximale dans le bloc. Les entrées de l'index sont triées en fonction des valeurs minimales.*

Les blocs B_i ($i > 0$) qui constituent les ***runs* naturels** doivent respecter les propriétés énoncées dans les lemmes 1 et 2 :

Lemme 1. *Soient B_{i-1} et B_i deux blocs qui se succèdent dans un run trié. Chaque valeur contenue dans le bloc B_i ($i > 0$) est supérieure à chaque valeur contenue dans le bloc B_{i-1} [51].*

Lemme 2. *Soient B_i et B_j deux blocs dans le fichier en entrée, avec $i < j$. Ces deux blocs appartiennent à un même **run naturel** si et seulement si toutes les valeurs de B_j sont supérieures aux valeurs de B_i [51].*

Former un ***run* naturel** revient à trouver une séquence de blocs qui respecte les lemmes 1 et 2. Pour ce faire, les auteurs font le rapprochement avec un problème connu dans la littérature. Il s'agit du problème de recherche du plus court chemin dans un graphe direct et acyclique :

- Chaque sommet du graphe représente un bloc B_i.
- Un arc est établi entre deux sommet B_i, B_j uniquement lorsque les deux blocs respectent les lemmes 1 et 2.
- Une fois le graphe construit en utilisant tous les blocs du fichier à trier, trouver un ***run* naturel** contenant des blocs qui respectent les lemmes 1 et 2, revient à trouver un chemin de taille M dans le graphe.

Les ***runs* naturels** trouvés dans le graphe sont gardés en mémoire principale sous forme d'index. Les blocs du fichier en entrée qui n'appartiennent à aucun ***run* naturel** , sont utilisés pour former des runs physiques (voir la définition 15).

Définition 15. *Un run physique est un fichier intermédiaire en mémoire secondaire contenant des valeurs triées [51].*

Une fois la génération des runs achevée, l'algorithme entame l'étape de fusion pour former un fichier final trié. L'opération de fusion est réalisée entre des runs physiques contenant des données triées et des ***runs* naturels**.

Exemple 4. *Dans la Figure 12, nous présentons le déroulement de la phase de génération des runs de l'algorithme sur un fichier ayant $N = 8$ blocs de données dans un espace mémoire pouvant contenir $M = 2$ blocs de données contenant 4 valeurs.*

*L'algorithme commence par créer un index sur les blocs du fichier en entrée. Cet index appelé index min-max associe à chaque bloc sa valeur minimale et sa valeur maximale (voir l'étape (1) dans la Figure 12). Il crée ensuite un **run naturel** à partir des entrées de cet index. Dans cet exemple, l'algorithme a réussi à détecter **run naturel run naturel** constitué de trois blocs : bloc 5, bloc 0 et bloc 7 (voir l'étape (2) dans la Figure 12).*

*Une fois les **runs naturels** détectés, l'algorithme crée les runs physiques constitués de $M = 2$ blocs. Ainsi, l'algorithme crée 3 runs de deux blocs à partir des blocs ne faisant pas partie du **run naturel** créé (voir l'étape (3) dans la Figure 12).*

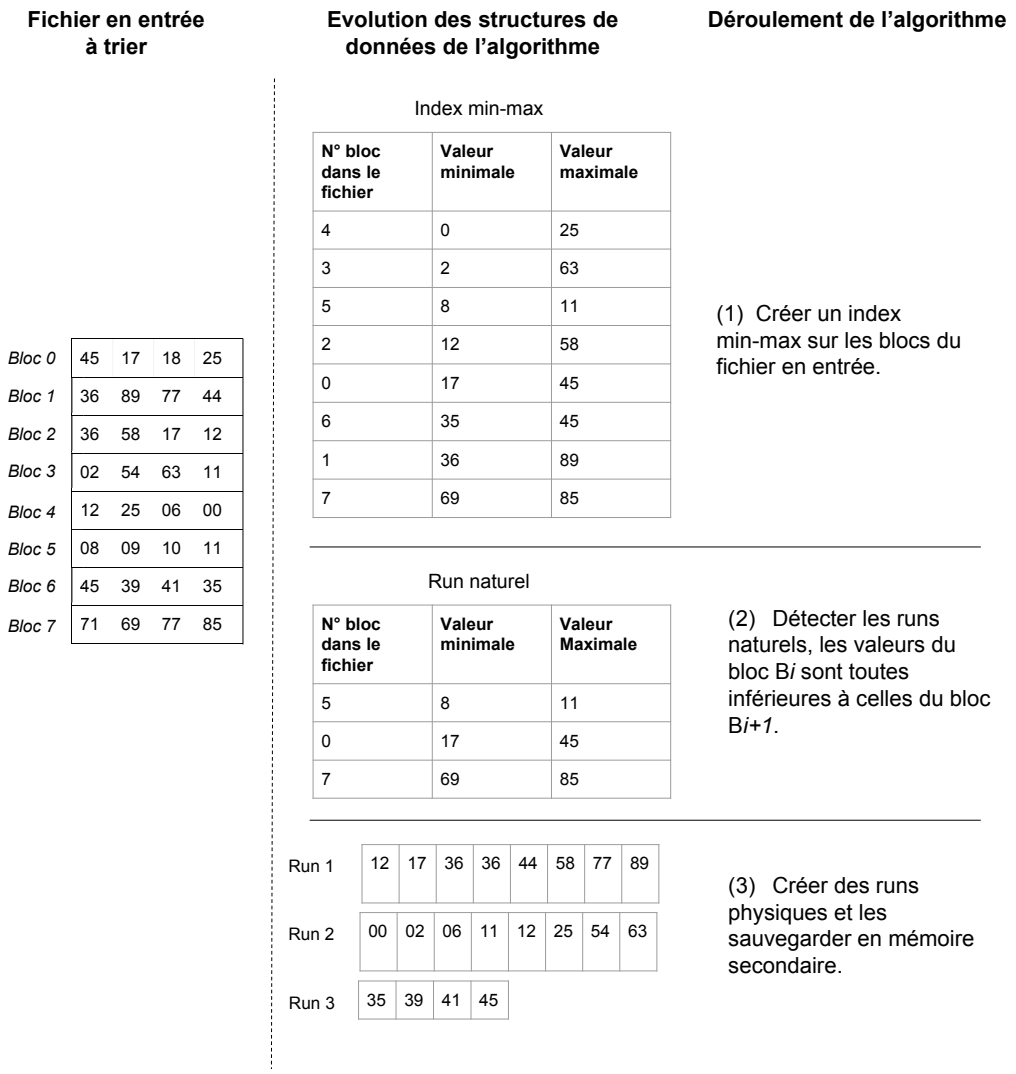

FIGURE 12 : Déroulement du tri avec l'algorithme "Natural Page Run"

4.1.3.3 *Analyse et discussion de l'algorithme*

Natural page run est conçu pour améliorer les performances des opérations de tri de données contenant des ***runs* naturels** . Or, dans un environnement de base de données, les ensembles de données ne contiennent pas forcément des données triées.

4.1.4 *Minsort*

Minsort [22] est un algorithme de tri en mémoire secondaire optimisé pour les mémoires flash. Cet algorithme se focalise sur les bases de données dans un environnement de type système embarqué. Il intègre les contraintes liées aux systèmes embarqués : un espace mémoire principale étroit et des ressources énergétiques limitées.

4.1.4.1 *Problématique*

La problématique traitée par l'algorithme est similaire à celle de FAST(1) (voir la Section 4.1.1.1).

4.1.4.2 *Conception et mise en œuvre*

La conception de l'algorithme Minsort vise deux objectifs. Le premier est d'éliminer les données temporaires que génèrent certains algorithmes de tri en mémoire secondaire (tri par fusion et tri par remplacement et sélection). Pour ce faire, il utilise des lectures supplémentaires, notamment des lectures aléatoires, pour chercher des valeurs minimales dans le fichier à trier. Ces valeurs minimales sont écrites directement dans le fichier final, évitant ainsi la création de données intermédiaires. Le deuxième objectif de cet algorithme est de réduire au maximum l'empreinte mémoire.

Minsort regroupe les données en régions de données. Les régions sont toutes de taille identique. Elles peuvent comprendre un ou plusieurs blocs de données. L'algorithme construit un index sur les régions, appelé index des minimums. Une entrée de l'index doit contenir une référence vers une région accompagnée de la valeur minimum de la région.

Le tri se déroule en plusieurs itérations. À chaque itération une valeur minimale est récupérée avec toutes ses occurrences pour être écrites directement dans le fichier trié final.

Au début de chaque itération, l'algorithme identifie la valeur minimale dans l'index ; cette valeur est ensuite sauvegardée dans une variable appelée `current`. L'étape suivante consiste à identifier les régions ayant la valeur minimale mémorisée dans `current` à partir de l'index. Chacune de ces régions est chargée en mémoire pour récupérer toutes les occurrences de la valeur minimale sauvegardé par `current`. Ces valeurs sont directement écrites dans le fichier final.

En récupérant les valeurs minimales d'une région, l'algorithme garde une trace de la prochaine valeur minimale dans une seconde variable nommé `next`. Une fois que l'algorithme a traité toutes les valeurs de la région, si la variable `next` contient une nouvelle valeur minimale, alors l'entrée correspondante de l'index pour cette région est modifiée avec cette nouvelle valeur minimale.

Une région est supprimée de l'index lorsque toutes ses valeurs sont écrites sur le fichier final.

L'algorithme s'arrête lorsque toutes les valeurs des régions sont écrites dans le fichier final.

Exemple 5. *La Figure 13 décrit le déroulement de Minsort sur un ensemble de données. L'exemple présente uniquement les deux premières itérations. Ici, chaque région est constituée d'un bloc contenant quatre valeurs.*

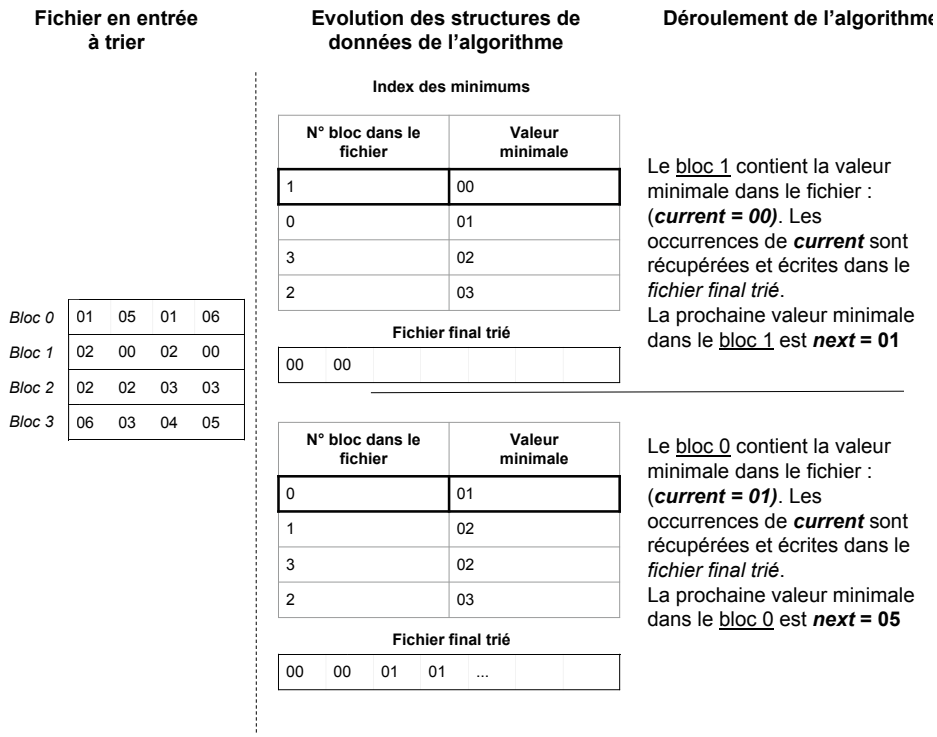

FIGURE 13 : Déroulement de l'opération de tri d'un ensemble de données avec l'algorithme Minsort

Lors de la première itération, la valeur minimale dans l'index des minimums est $\texttt{current} = 0$. *Elle se situe au niveau du* $\texttt{bloc}_1$. *L'algorithme charge le* $\texttt{bloc}_1$ *qui contient la valeur minimale* 0. *Ensuite, il cherche toutes les occurrences de cette valeur minimale qu'il écrit dans le fichier final. La prochaine valeur minimale dans le* $\texttt{bloc}_1$ *est* $\texttt{next} = 01$. *L'entrée de l'index correspondante à cette région est alors mise à jour.*

L'ordre des entrées dans l'index est mis à jour pour tenir compte de la nouvelle valeur minimale du $\texttt{bloc}_1$.

La deuxième itération traite la nouvelle valeur minimale dans l'index des minimums ($\texttt{current} = 01$*) qui se situe au niveau du* $\texttt{bloc}_0$.

4.1.4.3 *Analyse et discussion de l'algorithme*

Minsort met en œuvre une optimisation pour réduire les contraintes de durée de vie d'une mémoire flash causée par les opérations d'effacement. Pour ce faire, il utilise des opérations de lectures supplémentaires, notamment des lectures aléatoires pour récupérer successivement des valeurs minimales et les écrire directement dans le fichier final trié. Par conséquent, le nombre de lectures supplémentaires est relatif au nombre de valeurs différentes dans le fichier en entrée à trier.

Aujourd'hui, dans les périphériques de stockage SSD, la couche de gestion met en oeuvre des optimisations comme le *Wear leveling*. Le *Wear leveling* permet

de réduire les contraintes de durée de vie d'une mémoire flash. Par conséquent, l'annulation des opérations d'effacement au niveau algorithmique pourrait s'avérer moins efficace sur les périphériques de stockage SSD actuels. D'autant que cela se fait au détriment du coût en opérations de lectures, qui peuvent impacter le coût global en E/S de l'algorithme sur certaines configurations de données, ce que nous étudions dans le chapitre 5.

4.1.5 *Fsort*

Cet algorithme est proposé dans [9]. Il se base sur l'algorithme de remplacement et de sélection présenté dans la Section 2.2.2. L'algorithme vise à réduire la complexité de la phase de fusion et diminuer ainsi le coût en écriture et en effacement de données.

4.1.6 *TagSort*

Dans [64], les auteurs proposent un algorithme de tri en mémoire secondaire optimisé pour les mémoires flash et conçu pour trier des enregistrements.

4.1.6.1 *Problématique*

Dans un logiciel de traitement de données, l'opérateur de tri peut recevoir en entrée un ensemble de valeurs ou un ensemble d'enregistrements à trier en fonction d'une ou plusieurs clés de comparaison.

Lorsque les données à trier sont organisées sous forme d'un ensemble d'enregistrement, un opérateur de tri utilisant un tri par fusion ou un tri par remplacement et sélection considère tous les attributs de l'enregistrement pour trier les données, alors que seule la clé de comparaison est nécessaire pour définir l'ordre final (voir l'exemple 6). Cela implique des opérations d'E/S supplémentaires et une empreinte mémoire plus importante.

Exemple 6. *Soit la base de données TPC-H présentée dans l'exemple 1 du chapitre 2 (voir la Figure 1). L'opérateur de tri peut recevoir en entré la colonne* `o_orderStatus` *de la table order, comme il peut recevoir la table order, à trier en fonction de la clé* `o_custKey`.

4.1.6.2 *Conception et mise en œuvre*

L'objectif de l'algorithme est de diminuer la quantité de données temporaires et de réduire ainsi le nombre d'opérations d'écriture et d'effacement. Pour ce faire, l'algorithme crée un index sur les données, contenant uniquement les clés de comparaison. Cet index est utilisé pour trier les données et construire le fichier final trié.

La première étape de cet algorithme consiste à créer l'index sur les clés de comparaison. Cet index contient autant d'entrées que d'enregistrements dans le fichier. Chaque entrée de l'index est composée de deux champs :

- Une référence vers un enregistrement e_i
- Une valeur de la clé de comparaison pour l'enregistrement e_i

La taille de l'index dépend du nombre d'enregistrement dans le fichier en entrée et de la taille des clés de comparaison. Lorsque la taille de celui-ci dépasse celle de la mémoire principale, l'index est sauvegardé dans un fichier en mémoire secondaire.

Les étapes suivantes de l'algorithme consistent à **(1) trier l'index** et **(2) reconstruire le fichier final trié**. Nous détaillons ces deux étapes par la suite.

TRIE DE L'INDEX : Pour trier l'index, deux cas se présentent selon la taille de celui-ci :

- **Cas 1 :** la taille de l'index est inférieure à la taille de la mémoire principale allouée. L'index est alors totalement chargé et trié en mémoire principale en utilisant un algorithme de tri en mémoire principale (le tri rapide par exemple).
- **Cas 2 :** la taille de l'index est supérieure à la taille de la mémoire principale allouée. L'index ne peut être chargé en mémoire principale. Dans ce cas, l'index est trié en mémoire secondaire en utilisant un algorithme de tri en mémoire secondaire (le tri par fusion ou le tri par remplacement et sélection).

RECONSTITUTION DU FICHIER FINAL TRIÉ : L'index trié est utilisé pour lire les enregistrements du fichier en entrée. La lecture se fait suivant l'ordre croissant des clés de comparaison dans l'index. Par conséquent les opérations de lectures sont lancées de manière aléatoire sur le fichier en entrée. Les enregistrements sont lus puis écrits directement sur le fichier final trié.

L'algorithme est illustré dans l'exemple 7.

Exemple 7. *Cet exemple montre le déroulement du tri avec l'algorithme TagSort sur un fichier contenant 8 enregistrements (e0, ..., e7), avec une clé de comparaison chacun. Cet exemple est illustré avec deux figures : Figure 14 et Figure 15.*

La Figure 14 montre la construction de l'index sur les enregistrement de données (voir (1) dans la Figure 14). Une fois l'index construit, il est trié en fonction des valeurs dans la clé de comparaison (voir (2) dans la Figure 14).

La Figure 15 montre la construction du fichier final trié à partir de l'index trié selon les valeurs de la clé de comparaison (voir (3) dans la Figure 15).

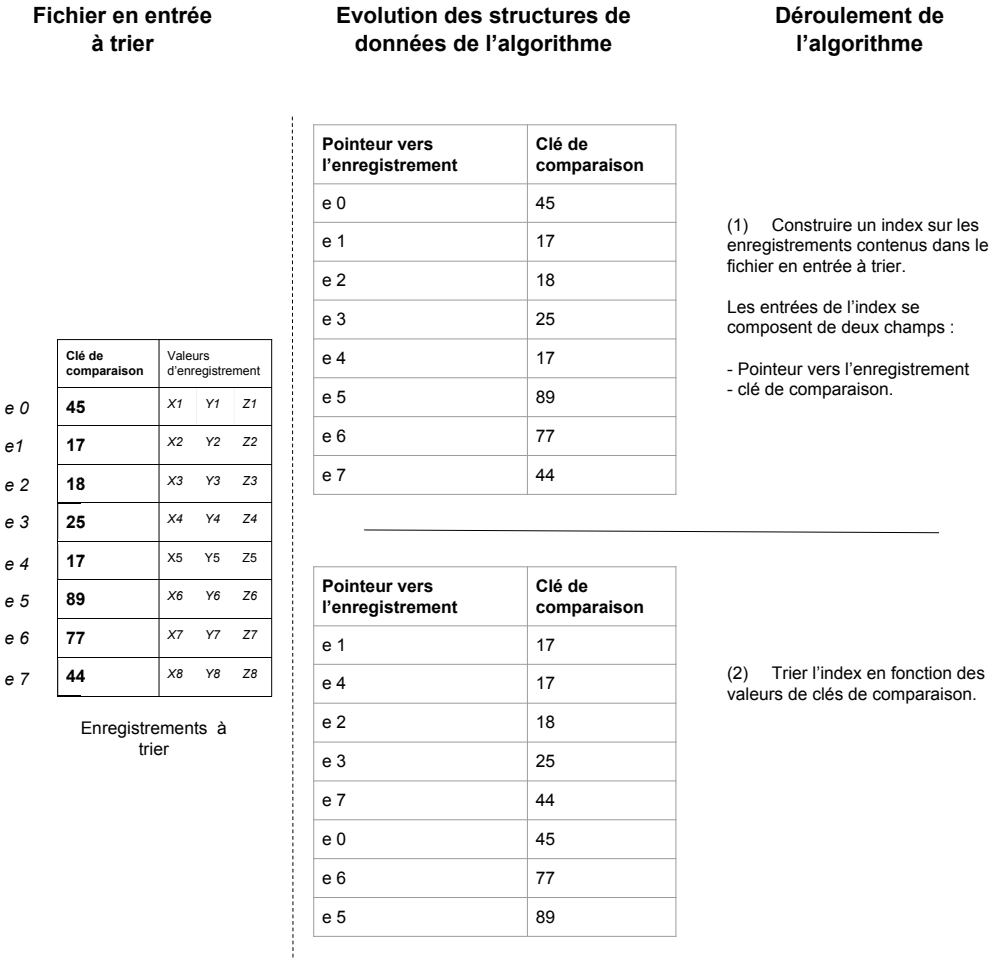

FIGURE 14 : Déroulement de l'opération de tri avec TagSort

4.1.6.3 *Analyse et discussion de l'algorithme*

Cet algorithme est conçu pour trier des enregistrements de données contenant une clé de comparaison. L'algorithme est optimisé pour le modèle de performance des périphériques de stockage SSD. Il réduit la quantité de données à traiter en mémoire principale et à écrire dans les fichiers temporaires.

Cet algorithme réduit les coûts en E/S et en empreinte mémoire lorsque les données sont organisées dans un ensemble d'enregistrements avec une ou plusieurs clés de comparaison. Lorsque les données sont constituées d'un ensemble de valeurs, l'algorithme n'apporte aucun gain en termes de coût d'E/S.

4.1.7 *FMSort*

FMSort (Flash Memory Sort) est un algorithme de tri en mémoire secondaire, proposé dans [48] et optimisé pour les mémoires flash.

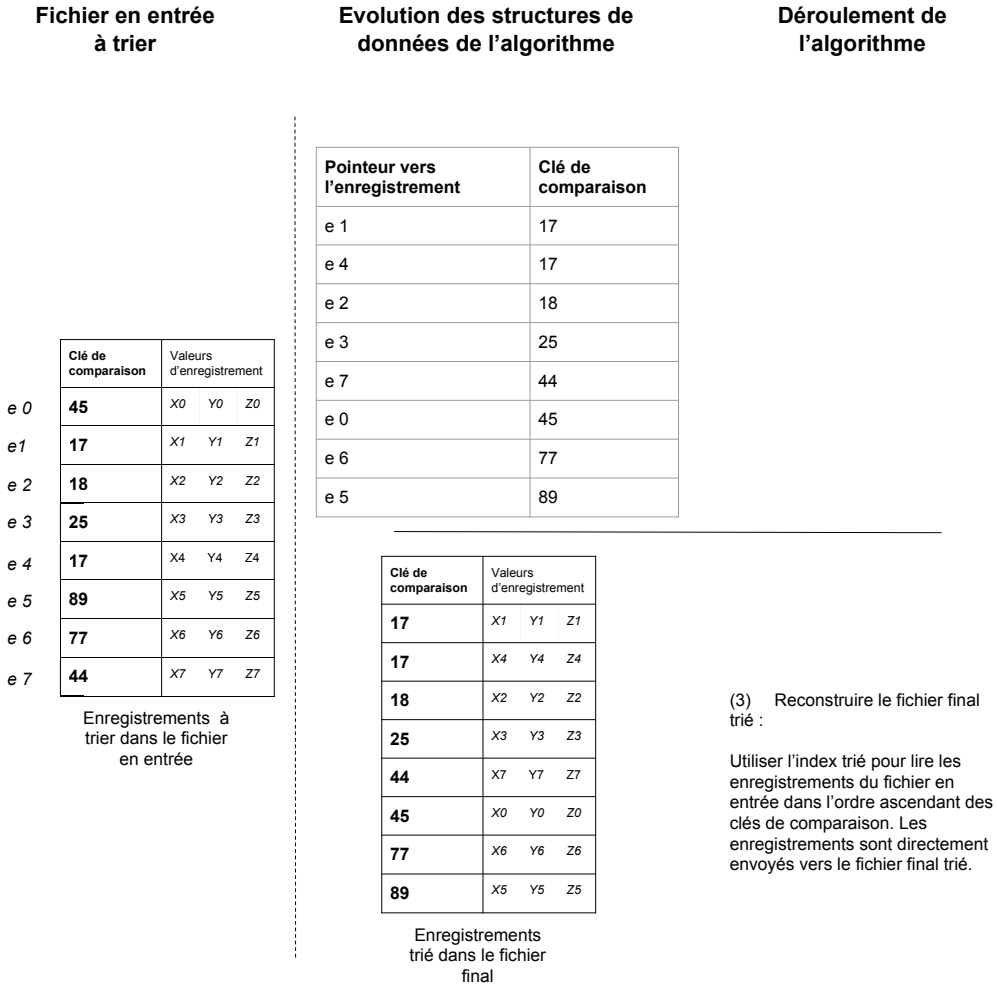

FIGURE 15 : Déroulement de l'opération de tri avec TagSort

4.1.7.1 *Problématique*

Les périphériques de stockage SSD actuels embarquent plusieurs niveaux de parallélisme, permettant l'exécution d'opérations d'E/S intrinsèques en parallèle. Ceci offre aux algorithmes la possibilité de lancer des opérations parallèles sur les périphériques de stockage.

La plupart des travaux ignorent les caractéristiques intrinsèques des périphériques SSD. Par conséquent, ils ne tirent pas profit de tous les avantages d'un périphérique SSD.

4.1.7.2 *Conception et mise en œuvre*

FMSort exploite le parallélisme intrinsèque des périphériques SSD pour réduire le temps d'accès aux données durant l'opération de tri. L'algorithme utilise aussi des opérations de lectures asynchrones, lancées simultanément avec des opérations de traitement de données en mémoire principale.

Comme l'algorithme de tri par fusion en mémoire secondaire, FMSort se déroule en deux phases ; **(1) une phase de génération des runs** et **(2) une phase de fusion des runs**. Ces deux phases sont présentées dans les paragraphes qui suivent.

PHASE DE GÉNÉRATION DES RUNS Durant la phase de génération des runs, FMSort crée des runs de taille M blocs, M étant le nombre de blocs alloués en mémoire principale. FMSort crée également un index sur les blocs de données contenus dans les runs. Chaque entrée de l'index est composée d'un bloc contenu dans un *run* et d'une valeur minimale. L'index est trié en fonction des valeurs minimales dans l'ordre ascendant.

FUSION DES RUNS Pour fusionner M runs, un algorithme de fusion traditionnel lit un bloc de données à partir de chaque run. Les blocs, une fois chargés en mémoire principale, sont insérés dans une structure de données de type tas qui permet de récupérer les valeurs minimales avec une complexité de $O(1)$. Avec cet algorithme de fusion, il n'est pas impossible qu'un bloc contenant des valeurs maximales soit chargé en mémoire principale alors que d'autres blocs contenant des valeurs minimales sont toujours dans les runs.

FMsort utilise l'index des runs pour sélectionner les blocs à charger en mémoire principale en fonction de leur valeur minimale. Il optimise ainsi l'occupation de la mémoire principale.

En plus de la sélection des blocs à charger en mémoire principale, FMSort apporte une seconde optimisation qui réduit le temps d'attente des données lors du chargement des blocs en mémoire principale. Cette optimisation vise à entrelacer les opérations CPU avec la lecture asynchrone des blocs de données en mémoire principale.

L'espace mémoire alloué pour la fusion est divisé en deux parties S (Synchrone) et A (Asynchrone). La première partie S contient les blocs en cours de fusion et la deuxième partie A sert d'espace tampon pour la lecture asynchrone des blocs suivants à fusionner.

Cette technique de fusion est illustrée avec l'exemple 8.

Exemple 8. *Nous illustrons à travers cet exemple l'algorithme de fusion mis en œuvre par FMSort. L'exemple se base sur une configuration minimale, contenant deux fichiers de runs à fusionner. Chacun des runs contient deux blocs (voir les runs à fusionner dans la Figure 16).*

La première étape (voir l'étape (1) dans la Figure 16) consiste à trier l'index pour obtenir un index des runs trié en fonction des valeurs minimales des blocs (voir (1) dans la figure 16). L'étape suivante (voir (2) dans la figure 16) consiste à construire le tas à partir des valeurs minimales issues de chacun des runs. Dans cet exemple, nous aurons deux valeurs minimales : 39 *pour le* $\mathtt{run}_1$ *et* 47 *pour le* $\mathtt{run}_2$.

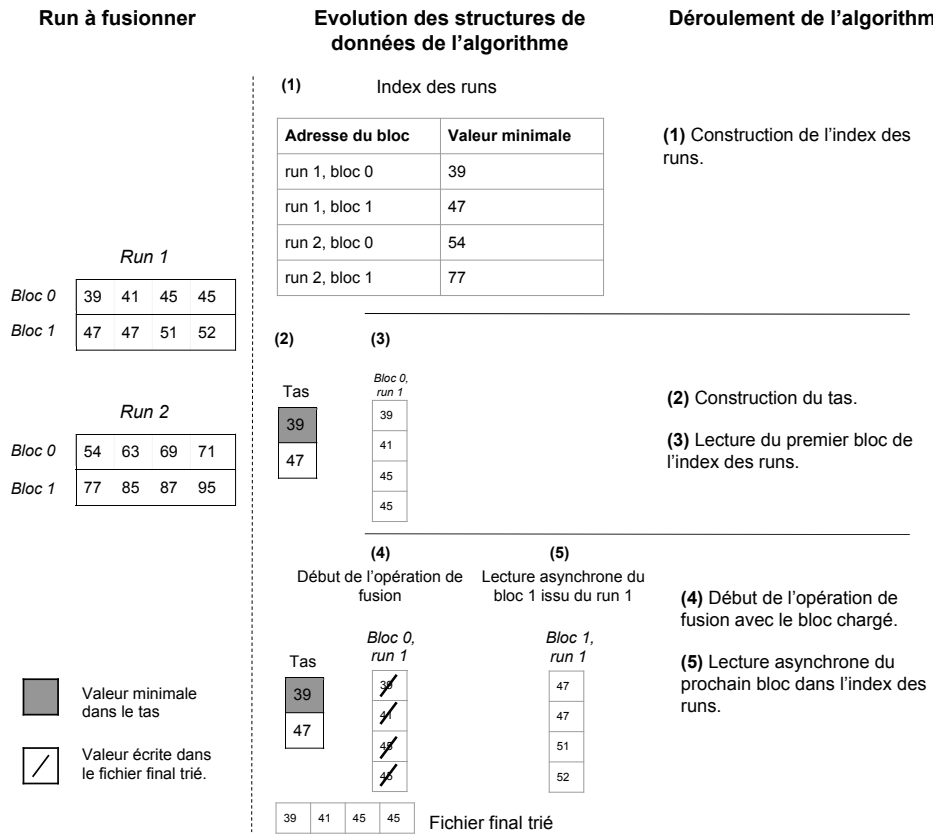

FIGURE 16 : Déroulement d'une opération de fusion avec FMSort

Le premier bloc de l'index des runs ($\texttt{bloc}_0$ *du* $\texttt{run}_1$*), contenant la valeur minimale de l'index est chargé dans la partie S de l'espace mémoire par une lecture synchrone (voir (3) dans la Figure 16).*

Une fois le bloc chargé dans la partie S de la mémoire principale, l'opération de fusion commence (voir (4) dans la figure 16). En parallèle, une opération de lecture asynchrone est lancée sur le bloc suivant de l'index des runs, en l'occurrence le ***bloc 1*** *du* ***run 1****. Celui-ci est chargé dans la partie A de la mémoire principale (voir (5) dans la Figure 16).*

Les étapes (4) et (5) dans la figure 16 sont répétées tant qu'il reste une entrée dans l'index des runs.

4.1.7.3 *Analyse et discussion de l'algorithme*

L'optimisation apportée par FMSort est basée sur l'entrelacement des opérations de lectures avec les opérations de comparaison nécessaires pour réaliser la fusion. Pour ce faire, il est nécessaire de diviser l'espace mémoire alloué en deux parties, une partie pour contenir les blocs en cours de fusion (S), et une autre pour contenir les blocs en cours de lecture asynchrone (A). Cette division de l'espace mémoire alloué augmente la complexité de l'opération de fusion lorsque le nombre de *runs* générés dépasse le nombre de blocs alloués en mémoire principale, impliquant des opérations d'E/S supplémentaires.

4.2 OPTIMISATION DES MÉCANISMES DE GESTION DES E/S EN MÉMOIRE SECONDAIRE

Nous présentons dans cette section deux études sur les mécanismes de pré-chargement de données à partir des périphériques SSD. La première étude **FAST** (Fast Application STarter) [39] concerne le pré-chargement de données pendant le chargement d'une application. La deuxième étude **Flashy** [62] vise à améliorer le pré-chargement des données pour des algorithmes de traitement de données.

4.2.1 *FAST : Fast Application STarter*

FAST est un mécanisme de lecture anticipée conçu pour exploiter les performances des lectures aléatoires d'un périphérique de stockage SSD. Ce mécanisme se base sur l'hypothèse qu'une même séquence d'E/S est lancée à chaque démarrage d'une application. FAST détecte ces séquences d'E/S et les exploite pour anticiper la lecture des données lors des démarrages suivants.

4.2.1.1 *Conception du mécanisme*

La conception de ce mécanisme vise à optimiser le démarrage à froid d'une application, qui représente le pire cas pour le lancement d'une application. Il s'agit du démarrage qui suit directement la mise en route du système d'exploitation.

Le mécanisme détecte une séquence de lecture lors du premier lancement d'une application. Cette séquence est utilisée lors des lancements suivants de l'application pour anticiper les opérations de lecture nécessaires au démarrage de l'application. Ces opérations de lecture sont lancées simultanément avec l'application et concourent pour la lecture du premier bloc de données. Une fois celui-ci chargé en mémoire principale, l'application lance un traitement CPU, entrelacé avec des lectures anticipées pour préparer les données suivantes en mémoire principale.

4.2.1.2 *Mise en œuvre du mécanisme*

FAST se base sur un ensemble de composants logiciels qui permettent de réaliser les opérations suivantes :

- Récolte des informations sur les requêtes d'E/S faites lors du démarrage à froid d'une application.
- Détection des séquences de requêtes d'E/S faites lors du démarrage à froid d'une application cible.
- Définition d'une séquence de requêtes d'E/S pour guider la lecture anticipée d'une application cible.

- Détection du démarrage à froid d'une application et préparation de la lecture anticipée des données.
- Lancement des requêtes de lecture anticipée pour une application cible, en ré-exécutant la séquence de requêtes d'E/S.

FAST est mis en œuvre dans un programme au sein de l'espace utilisateur du système d'exploitation. Il utilise des modules du noyau Linux pour collecter des informations sur un ensemble d'événements liés aux requêtes d'E/S. Ces informations permettent de construire les séquences de requêtes d'E/S lancées lors d'un démarrage à froid d'une application.

Les auteurs de FAST ont présenté une évaluation sur un ensemble d'applications. Ils ont comparé les temps de démarrage sur SSD avec et sans le mécanisme FAST. Les résultats ont montré que FAST réduit le temps de démarrage à froid de 28% en moyenne.

4.2.2 *Flashy*

Flashy est un mécanisme de lecture anticipée [62] également conçu pour les périphériques de stockage SSD. Son objectif est d'améliorer la précision des opérations de lecture anticipée en exploitant :

- Les performances élevés des opérations d'E/S sur un périphérique de stockage SSD.
- La localité temporelle des opérations d'E/S.

4.2.2.1 *Conception du mécanisme*

Ce mécanisme se base sur trois principes : (1) le contrôle de la lecture anticipée en fonction des performances du périphérique de stockage, (2) le contrôle de la lecture anticipée en fonction de son efficacité et enfin (3) la réalisation de lectures anticipées pour différents accès aux fichiers.

Nous décrivons ces principes dans les paragraphes qui suivent.

LECTURE ANTICIPÉE EN FONCTION DES PERFORMANCES DU PÉRIPHÉRIQUE DE STOCKAGE : Le mécanisme de lecture anticipée doit tenir compte de la bande passante du périphérique de stockage SSD pour ajuster le débit des lectures anticipées en fonction de celle-ci. Dans le cas contraire, les lectures anticipées risquent d'occuper toute la bande passante et provoquer un goulet d'étranglement qui bloque les opérations de lectures lancées par les autres programmes.

Pour remédier à ce problème de saturation de la bande passante, **Flashy** intègre un mécanisme qui consiste à évaluer et à ajuster de façon périodique la

quantité de données lue par le mécanisme en fonction des demandes de lecture de l'application et de la bande passante du périphérique de stockage.

LECTURE ANTICIPÉE EN FONCTION DE L'EFFICACITÉ DU MÉCANISME : **Flashy** évalue le bénéfice en performance des lectures anticipées. Dans le cas où le bénéfice est élevé, le mécanisme augmente son agressivité en anticipant plus de lectures. Dans le cas contraire, le mécanisme réduit le nombre de lectures à anticiper.

LECTURES ANTICIPÉES LORS D'ACCÈS MULTIPLES : Lorsque deux applications accèdent en parallèle aux données sur le périphérique de stockage, le mécanisme de lecture anticipée doit les traiter de façon séparée et indépendante. Pour ce faire, chaque lecture sur le périphérique de stockage est accompagnée d'un contexte. Le contexte d'une lecture regroupe plusieurs informations concernant l'environnement d'exécution comme l'identifiant du processus.

4.2.2.2 *Mise en œuvre du mécanisme*

Le mécanisme de lecture anticipée intègre 4 composants logiciels : le collecteur de traces, le détecteur de séquences, le mécanisme de pré-chargement des données et le mécanisme d'évaluation du pré-chargement. Nous décrivons ceux-ci dans les paragraphes qui suivent.

COLLECTEUR DE TRACES Ce composant récolte, avec l'aide du système d'exploitation, des informations sur les demandes d'E/S. Ces informations incluent : l'heure et la date de la demande d'E/S, le nom du processus, l'identifiant du processus, le type de la requête, l'adresse du bloc à lire/écrire et la quantité de données à lire ou à écrire.

DÉTECTEUR DE SÉQUENCES Ce composant se charge de traiter les informations collectées sur les demandes d'accès au périphérique de stockage pour détecter les motifs d'accès. Cette fonctionnalité est mise en œuvre dans un processus nommé `prefetchd`. Celui-ci se déclenche périodiquement. Il consulte les informations récoltées sur les demandes d'accès au stockage et décide du nombre de lectures anticipées et des blocs de données à lire.

Chaque fois que le processus `prefetchd` est déclenché, il parcourt les demandes de lectures récoltées afin d'identifier des séquences d'accès linéaires (voir la définition 16).

Définition 16. *On appelle séquence d'accès linéaire, un ensemble d'opérations de lecture qui accèdent aux blocs d'un fichier dans une même direction (vers la fin du fichier ou vers le début du fichier) et avec des sauts de longueur fixe [62].*

MÉCANISME DE PRÉ-CHARGEMENT DES DONNÉES Une fois que les séquences d'accès sont déterminées, **Flashy** pré-calcule le nombre de lectures anticipées maximum à lancer sur le périphérique de stockage. Ce nombre est déterminé en fonction de deux paramètres ; (1) le ratio entre la demande en lecture de l'application et le nombre de lectures à anticiper et (2) la bande passante du périphérique de stockage.

MÉCANISME D'ÉVALUATION DU PRÉ-CHARGEMENT Ce mécanisme analyse les flux d'opérations de lectures qui finissent avec un défaut de page majeur (lecture en mémoire secondaire). Cette analyse permet de vérifier si les flux traités contiennent des séquences d'accès linéaires. Lorsqu'un flux de lecture est classé comme étant linéaire et que les lectures ne sont pas anticipées par **Flashy**, l'agressivité du mécanisme de pré-chargement de données est augmentée. Inversement, l'agressivité est réduite lorsque **Flashy** anticipe la lecture de données alors que le mécanisme d'évaluation du pré-chargement ne détecte aucune lecture linéaire.

4.2.2.3 *Expérimentation et résultats*

Les auteurs présentent diverses expériences avec des applications centrées sur les données et des benchmarks d'E/S en utilisant des périphériques de stockage SSD, HDD ainsi qu'une configuration de SSD en RAID. Les résultats montrent que **Flashy** apporte une accélération de 20% en moyenne.

4.3 CONCLUSION

Nous avons présenté dans ce chapitre divers travaux destinés à améliorer les performances du tri en mémoire secondaire ainsi que la précision des lectures anticipées sur les périphériques de stockage SSD.

Les travaux présentés suivent différents chemins d'optimisations et profitent des opportunités offertes par les périphériques SSD, que nous résumons par :

- Une large bande passante pour les opérations de lecture, 500Mo/s avec des interfaces SATA et plus de 1000Mo/s avec des interfaces PCIe.
- Des performances des lectures aléatoires similaires avec celles des lectures séquentielles.
- Une architecture interne offrant plusieurs niveaux de parallélisme pour les opérations de lecture/écriture.

Ces travaux exploitent les opportunités présentées ci-dessus afin d'atteindre des objectifs d'optimisations telles que :

- La réduction du nombre d'opérations d'écriture des données temporaires.

- La réduction du nombre d'opérations CPU.
- L'amélioration de la précision des lectures anticipées en considérant la localité temporelle en plus de la localité spatiale.
- L'accélération des opérations de lecture/écriture en exploitant le parallélisme interne des périphériques SSD.

Ces travaux améliorent les performances des algorithmes de tri en mémoire secondaire existant et la précision des mécanismes de lectures anticipées. Ils présentent néanmoins des limites.

En effet, les algorithmes présentés ne tiennent pas compte des volumes de données qui augmentent face à la taille des mémoires principales, comme ils ne tiennent pas compte non plus des différentes distributions de données. Quant aux travaux liés aux optimisations des mécanismes de lectures anticipées, FAST [39] se limite à l'accélération du lancement d'une application et ne s'applique pas au traitement des données et Flashy [62] est limité à des séquences d'accès linéaires.

Dans le chapitre qui suit nous allons évaluer les travaux de tri en mémoire secondaire en quantifiant le nombre d'opération de lecture et d'écriture. Nous allons ensuite analyser leur efficacité en fonction du volume de données, de l'espace mémoire principale alloué et de la distribution des données.

Deuxième partie

CONTRIBUTIONS

Chapitre 5

ANALYSE DES COÛTS D'E/S POUR LES ALGORITHMES DE TRI OPTIMISÉS SUR SSD

Les performances d'un algorithme de tri en mémoire secondaire dépendent de deux paramètres principaux : le temps de calcul et le temps d'accès aux données. Le temps de calcul représente principalement le temps nécessaire pour exécuter les opérations de comparaison. Le temps d'accès aux données représente le temps nécessaire pour charger les données en mémoire principale et écrire les données triées en mémoire secondaire. Lorsque la taille des données dépasse la taille de l'espace mémoire alloué, le temps d'accès aux données devient un goulet d'étranglement pour les performances d'un algorithme de tri en mémoire secondaire [30].

Les performances des accès aux périphériques SSD épousent deux caractéristiques principales : (1) la symétrie entre performance des lectures aléatoires et séquentielles, (2) l'asymétrie entre performance des lectures, des écritures et des effacements. Il existe dans l'état de l'art des évaluations de coût d'E/S [22, 55, 64] qui prennent en compte les caractéristiques précédentes (1) et (2). Toutefois elles ne prennent pas en compte la distribution des valeurs dans l'ensemble des données à trier.

Dans ce chapitre, nous présentons un modèle d'évaluation des coûts d'E/S pour les algorithmes, présentés dans le chapitre 4. Les coûts en E/S de ces algorithmes sont évalués en fonction de la taille des données, de l'espace mémoire alloué et de la distribution des données.

Le chapitre est organisé en 4 sections. La section 5.1 présente les hypothèses, quelques définitions et les notations du modèle d'évaluation des coûts d'E/S. La section 5.2 décrit une évaluation des coûts d'E/S des algorithmes de l'état de l'art. La section 5.3 décrit une analyse des coûts d'E/S en fonction de la distribution des données. Enfin, la section 5.4 résume les résultats de l'analyse et conclut le chapitre.

5.1 HYPOTHÈSES, DÉFINITIONS ET NOTATIONS DU MODÈLE DE COÛT E/S

Nous définissons dans cette section les types de données utilisés dans notre modèle de coût d'E/S. Nous présentons ensuite les hypothèses ayant servies de base pour notre travail. Ces hypothèses concernent l'organisation des données (hypothèse 1) et la distribution des données (hypothèses 2, 3, 4 et 5). Nous présentons enfin les notations utilisées pour modéliser les coûts en E/S.

5.1.1 *Définitions*

Le modèle de calcul des coûts d'E/S se base sur trois distributions de données : aléatoire, triée et redondante. Ces types de données sont présentées dans les définitions 17, 18, 19 et 20 ci-après.

Définition 17. *Un ensemble de données est aléatoire lorsque ses valeurs ne respectent aucune relation d'ordre [30].*

Définition 18. ***Un ensemble de données triées*** *est un ensemble de valeurs qui respecte un ordre ascendant ou descendant, défini par un opérateur ou une fonction de comparaison [30].*

Définition 19. ***Un ensemble de données partiellement triées*** *: un ensemble de données est dit partiellement trié, si et seulement si, il est possible de le transformer en ensemble trié en soustrayant un sous ensemble de valeurs [51].*

Définition 20. *Dans un ensemble contenant des données redondantes, chaque valeur peut être répétée plusieurs fois [30].*

5.1.2 *Hypothèses*

Notre modèle de calcul des coûts d'E/S se base sur cinq hypothèses. La première concerne l'organisation des données (hypothèse 1), la deuxième concerne la distribution des données (hypothèse 2) et les trois hypothèses restantes concernent les types de distribution de données utilisées dans le modèle : aléatoires, triées et redondantes (hypothèses 3, 4 et 5).

Hypothèse 1. ***Organisation des données :*** *nous supposons que les données à trier contiennent* n *valeurs de même type de données et de taille fixe.*

Hypothèse 2. ***Distribution des données :*** *Dans notre modèle de coût d'E/S, nous supposons que la distribution des données est soit* ***aléatoires*** *(voir la définition 17), soit* ***triée*** *(voir les définitions 18 et 19) ou* ***redondantes*** *(voir la définition 20).*

Dans un ensemble aléatoire, la probabilité que deux valeurs qui se suivent soient dans le bon ordre est très faible (proche de zéro). Cette caractéristique découle du désordre entre les valeurs d'un ensemble de données aléatoires. Pour modéliser ce désordre dans le cadre de notre étude, nous établissons l'hypothèse 3.

Hypothèse 3. *Nous supposons que dans un ensemble de données aléatoires, les N valeurs minimales (respectivement maximales) sont distribuées sur les N blocs constituant le fichier en mémoire secondaire [30].*

Un ensemble de données triées ou partiellement triées est obtenu lorsque les valeurs de celui-ci décrivent des actions et événement dépendants, comme l'heure et la date d'une transaction dans un supermarché. Dans le but de modéliser ces deux types d'ensembles de données (triées et partiellement triées), nous établissons l'hypothèse 4

Hypothèse 4. *Nous supposons que dans un ensemble de données triées ou partiellement triées, les N valeurs minimales (respectivement maximales) sont groupées dans un sous-ensemble de blocs de données [30].*

Un ensemble de données redondantes est obtenu lorsque le nombre de valeurs différentes nécessaires pour décrire une propriété se trouve dans un petit intervalle de valeurs. Par exemple, l'état d'une transaction avec un ensemble fini de valeurs : réussie, annulée, échouée, abandonné. C'est le cas par exemple dans la table `orders` de la base de données TPC-H (voir la Figure 1). Pour modéliser ce type d'ensemble de données, nous établissons l'hypothèse 5.

Hypothèse 5. *Nous supposons que dans un ensemble de données redondantes, la distribution des valeurs est uniforme dans un bloc, autrement dit, les valeurs se répètent un même nombre de fois [30].*

5.1.3 *Notations*

Concernant les notations, nous utilisons celles déjà établies dans le chapitre 4, tableau 1. Celles-ci concernent la taille de l'ensemble de données en entrée (N) et l'espace mémoire disponible (M).

Nous définissons également la notation D pour représenter le nombre moyen de valeurs différentes dans les N blocs de données. La notation B représente le nombre de valeurs dans un bloc.

Enfin, nous utilisons R pour représenter le coût d'une opération de lecture et *W* pour le coût d'une opération d'écriture.

Les nouvelles notations définies dans ce chapitre sont résumées dans le tableau 2.

Tableau 2 : Notations

Notation	Définition
D	Nombre moyen de valeurs différentes dans un bloc
B	Nombre de valeurs dans un blocs
R	Coût unitaire d'une opération de lecture
W	Coût unitaire d'une opération d'écriture

5.2 ÉVALUATION DES COÛTS D'E/S

Nous étudions dans les sections suivantes les coûts en E/S des algorithmes présentés dans la section 4.1 du chapitre 4. Cette étude se base sur les hypothèses énoncées dans la section 5.1.

Les résultats de cette étude sont résumés dans le tableau 3.

5.2.1 *Le tri par fusion*

Nous évaluons dans cette section le coût en E/S de l'algorithme de tri par fusion en mémoire secondaire. Cet algorithme est présenté dans la section 2.2.1 du chapitre 2.

La première phase de génération des *runs* nécessite la lecture de l'ensemble du fichier pour le chargement des données en mémoire principale. Les données sont chargées par parties de M blocs de données. Chaque partie de données est triée puis écrite dans un *run* en mémoire secondaire pour laisser la place à la prochaine partie de M blocs de données. Cette phase nécessite N opérations de lecture et N opérations d'écriture sur mémoire secondaire. Par conséquent, le coût en E/S se résume par : $N \cdot (R + W)$.

La phase de génération des *runs* génère des *runs* qui font la taille de la mémoire allouée, qui est de M blocs. Le nombre de *runs* générés est alors de $\frac{N}{M}$.

Le deuxième phase de cet algorithme, consiste à fusionner $\frac{N}{M}$ *runs* de taille M blocs pour obtenir un fichier final trié. L'algorithme de tri par fusion utilise deux blocs en mémoire comme tampons d'E/S et les $M-2$ blocs restants sont utilisés pour l'opération de fusion.

Le nombre d'itérations nécessaires pour réaliser l'opération de fusion est donné par : $\log_{M-2} \frac{N}{M}$ [41]. Chaque itération nécessite la lecture des données triées sauvegardées dans les fichiers *runs*. Ces données sont ensuite fusionnées, puis écrites dans de nouveaux fichiers *runs*. Les données sont écrites dans le fichier

final trié dans le cas de l'itération finale. Par conséquent, le coût en E/S de cette phase est égale à : $N \cdot \lceil \log_{M-2} \frac{N}{M} \rceil \cdot (R + W)$.

Le coût global est donné par l'équation 1. Il regroupe le coût en E/S des deux phases, génération des *runs* et fusion des runs.

$$\begin{aligned} \text{Coût\_ES} &= N \cdot (R + W) + N \cdot \left\lceil \log_{M-2} \frac{N}{M} \right\rceil \cdot (R + W) \\ &= N \cdot (1 + \left\lceil \log_{M-2} \frac{N}{M} \right\rceil) \cdot (R + W) \end{aligned} \tag{1}$$

5.2.2 *Natural page run*

Pour identifier des *natural runs*, *Natural Page Run* réalise un premier parcours du fichier afin de récupérer les valeurs minimales et maximales de chaque bloc. Cette première étape de l'algorithme engendre N opérations de lecture, soit un coût équivalent à $N \cdot R$.

Lors de la génération des runs, Cet algorithme lance N opérations de lecture pour charger des parties de données de taille M en mémoire principale, ce qui donne un coût en E/S équivalent à $N \cdot R$.

Avec la génération des *natural runs* qui ne sont pas écrits en mémoire secondaire, cet algorithme réalise une économie de P écritures de blocs durant la phase de génération des *runs*. Par conséquent le coût en écriture est de $(N - P) \cdot W$.

La phase de fusion des *runs* est réalisée avec le même coût que l'algorithme de fusion de base, soit $N \cdot \lceil \log_{M-2} \frac{N}{M} \rceil \cdot (R + W)$

Le coût total de l'algorithme est alors donné par l'équation 2 ci-dessous.

$$\text{Coût\_ES} = 2 \cdot N \cdot R + ((N - P) \cdot W) + \left(N \cdot \left\lceil \log_{M-2} \frac{N}{M} \right\rceil \cdot (R + W) \right) \tag{2}$$

5.2.3 *MinSort*

Dans une première étape, cet algorithme divise le fichier en régions constituées d'un ou de plusieurs blocs. L'algorithme réalise un premier parcours du fichier afin de récupérer les valeurs minimales de chaque région. Avec ces valeurs minimales, l'algorithme crée un index des minimums. Lors de cette première étape, l'algorithme engendre N opérations de lecture, soit un coût de $N \cdot R$.

Ensuite, chaque région i du fichier en entrée est lue d_i fois, d_i étant le nombre de valeurs différentes dans la région. Soit D la valeur moyenne des d_i pour un fichier en entrée donné. Le nombre d'opérations de lecture est alors de $N \cdot (1 + D)$, ce qui donne un coût en lecture de $N \cdot (1 + D) \cdot R$.

Les valeurs minimales sont écrites dans le fichier final trié au fur à mesure que celles-ci sont récupérées à partir de l'index des minimums. Par conséquent, l'algorithme réalise N opérations d'écriture au total, ce qui donne un coût total de $N \cdot W$.

Le coût global en E/S pour l'algorithme MinSort est donc donné par l'équation 3 :

$$\begin{aligned} \text{Coût\_ES} &= (N \cdot (1 + D) \cdot R) + (N \cdot W) \\ &= N \cdot ((1 + D) \cdot R + W) \end{aligned} \tag{3}$$

5.2.4 *FAST(1)*

FAST(1) parcourt les N blocs du fichier en entrée et récupère M blocs de données triées. Pour obtenir le fichier final trié, FAST(1) réalise $\frac{N}{M}$ parcours du fichier en entrée.

Le parcours du fichier en entrée, implique un coût en lecture égale à $N \cdot R$. Par conséquent, les $\frac{N}{M}$ parcours du fichier en entrée nécessaires pour obtenir le fichier final trié, impliquent un coût en lecture de $N \cdot \frac{N}{M} \cdot R$.

Le fichier final trié est construit progressivement. À chaque parcours du fichier, l'algorithme récupère M blocs de données triées qu'il écrit directement sur le fichier final trié, ce qui donne un coût en écriture de $M \cdot W$. Après $\frac{N}{M}$ parcours du fichier en entrée, le coût total en écriture atteint $M \cdot \frac{N}{M} \cdot W = N \cdot W$.

FAST(1) obtient directement un fichier final trié sans passer par une phase de fusion des *runs*. Par conséquent le coût en E/S de la phase de fusion des *runs* est nul.

Le coût total en E/S est donné par l'équation 4 :

$$\begin{aligned} \text{Coût\_ES} &= \left(N \cdot (\frac{N}{M} \cdot R)\right) + (N \cdot W) \\ &= N \cdot \left((\frac{N}{M} \cdot R) + W\right) \end{aligned} \tag{4}$$

5.2.5 *FAST(N)*

Cet algorithme divise le fichier en entrée en partie de Q blocs, ce qui donne $\frac{N}{Q}$ parties. Chaque partie est traitée avec l'algorithme FAST(1), ce qui donne un coût en E/S égale à $Q \cdot \left((\frac{Q}{M} \cdot R) + W\right)$. Par conséquent, le coût total pour les $\frac{N}{Q}$ parties, est de $\frac{N}{Q} \cdot Q \cdot \left((\frac{Q}{M} \cdot R) + W\right)$, après simplification, nous obtenons l'équation 5.

$$\text{Coût\_ES} = N \cdot \left((\frac{Q}{M} \cdot R) + W\right) \tag{5}$$

Les parties de données triées en utilisant FAST(1) sont ensuite fusionnées en utilisant l'algorithme de fusion des runs traditionnel. Le coût en E/S de cette opération de fusion est évalué avec l'équation 6.

$$\text{Coût\_ES} = N \cdot \lceil \log_{M-2} \frac{N}{Q} \rceil \cdot (R + W) \tag{6}$$

Tableau 3 : Coût en E/S des principaux algorithmes de tri en mémoire secondaire

Algorithmes	**RGRC**	**RGWC**	**RMRC**	**RMWC**	**Coût en E/S**
Tri par fusion [41]	$N \cdot R$	$N \cdot W$	$N \cdot \lceil \log_{M-2} \frac{N}{M} \rceil \cdot R$	$N \cdot \lceil \log_{M-2} \frac{N}{M} \rceil \cdot W$	Coût en E/S logarithmique
Natural page run[51]	$2 \cdot N \cdot R$	$(N - P) \cdot W$	$N \cdot \lceil \log_{M-2} \frac{N}{M} \rceil \cdot R$	$N \cdot \lceil \log_{M-2} \frac{N}{M} \rceil \cdot W$	Coût en E/S logarithmique
FSort [9]	$N \cdot R$	$N \cdot W$	$N \cdot \lceil \log_{M-2} \frac{N}{2M} \rceil \cdot R$	$N \lceil \log_{M-2} \frac{N}{2M} \rceil \cdot W$	Coût en E/S logarithmique
MinSort [22]	$N \cdot (1 + D) \cdot R$	$N \cdot W$	0	0	$1 + D$ opérations de lecture par bloc
FAST [55]	$N \cdot \frac{N}{M} \cdot R$	$N \cdot W$	0	0	Coût en lecture quadratique
FAST(N) [55]	$N \cdot \frac{Q}{M} \cdot R$	$N \cdot W$	$N \cdot \lceil \log_{M-2} \frac{N}{Q} \rceil \cdot R$	$N \cdot \lceil \log_{M-2} \frac{N}{Q} \rceil \cdot W$	Coût en lecture quadratique

5.3 ANALYSE DES COÛTS D'E/S

Dans cette section, nous analysons les coûts des E/S pour les algorithmes de l'état de l'art en fonction des paramètres annoncés dans la section 5.1, en l'occurrence la taille de l'ensemble de données, la taille mémoire et le type d'ensemble de données. Le tableau 3 résume les coûts des E/S pour chacun des algorithmes de tri étudiés. Nous utilisons dans ce tableau les abréviations suivantes : **RGRC** (*Run Generation Read Cost*) pour le coût en lecture de la génération des runs, **RGWC** (*Run Generation Write Cost*) pour le coût en écriture de la génération des runs, **RMRC** (*Run Merge Read Cost*) pour le coût en lecture de la fusion des *runs* et enfin **RMWR** (*Run Merge Write Cost*) pour le coût en écriture de la fusion des runs.

On constate à partir du tableau 3 que les coûts des E/S pour les algorithmes de tri en mémoire secondaire sont fortement corrélés par la taille des données ainsi que l'espace mémoire alloué.

5.3.1 *Natural Page Run*

Cet algorithme effectue un parcours supplémentaire du fichier en entrée avec un coût de $N \cdot R$ afin d'économiser P écritures de blocs à la fin de la phase de génération des *runs*, soit un coût de $P \cdot W$. Le coût de la phase de fusion des *runs* reste inchangé par rapport au coût de l'algorithme de tri par fusion traditionnel.

Cet algorithme est efficace, si et seulement si, la réduction du coût en écriture $P \cdot W$ est plus importante que le coût supplémentaire en lecture $N \cdot R$ (voir l'inégalité 7).

$$P \cdot W > N \cdot R \tag{7}$$

Certaines études ont montré que le coût des opérations d'écriture est au minimum 10 fois plus important que celui des lectures [18]. Or, les périphériques SSD actuels embarquent une mémoire tampon permettant de réduire l'écart entre les performances des écritures et celles les lectures. Ainsi, sur les périphériques SSD que nous utilisons, le ratio entre les coûts des écritures et celles des lectures $\frac{W}{R}$ a été mesuré dans des conditions normales et se situe entre 1 et 2. En considérant le rapport $W = 2 \cdot R$ dans l'inéquation 7, on obtient l'inégalité 8. Par conséquent, *Natural page run* est efficace lorsque le nombre de bloc intégrant des *Natural run* P dépasse les $\frac{N}{2}$ blocs (soit la moitié des blocs du fichier en entrée).

$$P > \frac{N}{2} \tag{8}$$

Lorsque l'ensemble des données en entrée est trié ou partiellement trié, cet algorithme peut s'avérer efficace et le nombre de blocs intégrant les *Natural run* P pourrait dépasser $\frac{N}{2}$ surtout lorsque les données sont initialement triées.

Lorsque les données sont aléatoires, selon l'hypothèse 3, les N valeurs minimales (respectivement maximales) sont distribuées sur les N blocs du fichier. Il est donc impossible de trouver des blocs dans le fichier en entrée qui respectent le lemme 2 concernant un *natural run*. L'algorithme ne détecte aucun *natural run*, le nombre d'écritures économisés P est égale à 0. Cette valeur de P ne respecte pas l'inégalité 8, par conséquent l'algorithme *Natural page run* donne un coût en E/S supérieur à celui du tri par fusion traditionnel.

5.3.2 *FSort*

Le coût en E/S de cet algorithme dépend de la taille de l'ensemble de données N et de la mémoire principale M. Cet algorithme réduit le coût en E/S en diminuant le nombre d'itérations nécessaires pour fusionner les *runs* générés à $\lceil \log_{M-2} \frac{N}{2M} \rceil$. Par conséquent, le coût en E/S est au pire des cas égal à celui du tri par fusion.

5.3.3 *MinSort*

Le coût des E/S pour MinSort dépend de la taille du fichier en entrée N et de la distribution des données D. Bien que le tri s'effectue en une seule phase, l'algorithme réalise jusqu'à $N \cdot (1 + D)$ opérations de lecture sur le fichier en entrée.

MinSort réalise des lectures supplémentaires pour réduire le coût en écriture. En considérant que $W = 2 \cdot R$. On obtient l'équation 9 suivante :

$$\text{Coût\_ES} = N \cdot (3 + D) \cdot R \tag{9}$$

Minsort est efficace en termes de coût d'E/S comparé à l'algorithme de tri par fusion, si et seulement si l'inégalité ci-dessous est satisfaire.

$$N \cdot (3 + D) \cdot R < N \cdot (3 \cdot R) + N \cdot \left\lceil \log_{M-2} \frac{N}{M} \right\rceil \cdot (3 \cdot R) \tag{10}$$

On obtient à partir de l'inégalité 10 : $D < 3 \cdot \lceil \log_{M-2} \frac{N}{M} \rceil$. Pour un ratio $\frac{N}{M}$ égale à 8192, La valeur de D doit être inférieure à 9 pour que MinSort soit plus performant que le tri par fusion traditionnel. Cela veut dire qu'un bloc pouvant contenir jusqu'à 1024 entiers doit contenir uniquement 9 valeurs différentes. Par conséquent, cet algorithme peut être efficace uniquement avec des ensembles de données aléatoires ou partiellement triées ayant un intervalle de valeur restreint.

5.3.4 *FAST(N)*

Le coût en E/S de cet algorithme dépend de la taille du fichier en entrée N. La phase de génération des *runs*, génère des *runs* de taille Q, avec $M \leqslant Q \leqslant N$, en effectuant $\frac{N \cdot Q}{M}$ opérations de lecture supplémentaires sur le fichier en entrée, soit un coût en lecture de $\frac{N \cdot Q}{M} \cdot R$.

Nous obtenons alors l'inégalité suivante :

$$N \cdot R < RGRC < \frac{N^2}{M} \cdot R \tag{11}$$

L'inégalité 11 montre que le coût en lecture pour $Q = M$ est de $N \cdot M$, soit le même coût en lecture que la phase de génération des *runs* du tri par fusion (voir le tableau 3).

Le coût en E/S de la phase de génération des *runs* dépend de la taille des *runs* générés $M \leqslant Q \leqslant N$. Lorsque $Q = M$, le nombre de *runs* générés est $\frac{N}{M}$ et le coût en E/S de la phase de fusion de *runs* est équivalent à celui du tri par fusion, soit $\lceil \log_{M-2} \frac{N}{M} \rceil$. Lorsque $Q = N$, l'algorithme génère directement le fichier final trié en effectuant $\frac{N^2}{M}$ opérations de lecture sur le fichier en entrée. L'algorithme atteint alors un coût quadratique en opérations de lecture.

Dans le meilleur des cas, FAST(N) affiche le même coût que l'algorithme de tri par fusion traditionnel. Au pire des cas, il atteint un coût en opérations de lecture quadratique par rapport à la taille du fichier en entrée N et dépasse le coût en E/S de l'algorithme de tri par fusion traditionnel.

On en conclut que FAST(N) est moins efficace que le tri par fusion lorsqu'il s'agit de trier de grands volumes de données.

5.4 CONCLUSION

Dans ce chapitre, nous avons présenté une évaluation des coûts des E/S pour les différents algorithmes de tri en mémoire secondaire. Cette évaluation montre que les algorithmes comme Natural Page Run [51] et MinSort [22] dépendent largement de la distribution des données. Ces algorithmes ciblent des configurations d'ensemble de données partiellement triées pour le premier et des données redondantes pour le deuxième. Par ailleurs, les coûts en E/S sont supérieurs à ceux du tri par fusion traditionnel. L'évaluation montre aussi que les algorithmes comme FAST et FAST(N) dépendent de la taille du fichier N. Le coût en E/S de ces algorithmes est quadratique. Il dépasse alors le coût du tri par fusion qui est linéaire avec un facteur logarithmique et qui dépend de la taille des données N (voir le tableau 3).

A partir des résultats de l'étude réalisée dans ce chapitre, nous établissons quatre objectifs qu'un algorithme de tri en mémoire secondaire doit tenter d'at-

teindre pour assurer un coût en E/S réduit dans un environnement de SGBD. Nous énonçons ci-dessous ces objectifs :

- **Objectif 1** : garder un coût en E/S linéaire avec l'évolution du volume des données.
- **Objectif 2** : minimiser le coût en E/S quelque soit l'espace mémoire alloué.
- **Objectif 3** : minimiser le coût en E/S en exploitant la distribution des valeurs dans un ensemble de données pour réduire les coûts en E/S.
- **Objectif 4** : minimiser le coût en E/S avec des données complètement aléatoires.

Les principes annoncés ci-dessus nous ont servi de base pour la conception d'un nouvel algorithme de tri en mémoire secondaire. Celui-ci fera l'objet du prochain chapitre.

Chapitre 6

MONTRES : ALGORITHME DE TRI EN MÉMOIRE SECONDAIRE BASÉ SUR LA FUSION À LA VOLÉE

Dans la littérature, plusieurs algorithmes de tri en mémoire secondaire conçus pour les périphériques de stockage SSD ont été proposés [9, 22, 51, 55, 64]. Ces algorithmes sont présentés dans le chapitre 4 et leurs coûts en E/S sont modélisés dans le chapitre 5. Ce chapitre a soulevé plusieurs problèmes que nous résumons ci-dessous :

- Lorsque le volume de données augmente face à la taille de la mémoire principale allouée, le coût en E/S de FAST [55] et Natural Page Run [51] dépasse celui de l'algorithme de tri par fusion traditionnel.
- Lorsque l'ensemble de données est aléatoire, Natural Page Run [51] et Min-sort [22] atteignent des coûts en E/S qui dépassent celui du tri par fusion traditionnel.

Le chapitre 5 nous a conduit à définir 4 objectifs qu'un algorithme de tri en mémoire secondaire doit viser afin d'être efficace dans le contexte d'une base de données. Ces objectifs ont motivé la conception et la mise en œuvre de MONTRES. Pour ce faire MONTRES exploite les optimisations ci-dessous :

- **Optimisation 1 :** Réduction de la quantité de données intermédiaires.
- **Optimisation 2 :** Expansion des *runs*.
- **Optimisation 3 :** Fusion des *runs* en une seule itération.

L'optimisation 1 permet à MONTRES de réduire la quantité de données à fusionner ainsi que l'espace mémoire nécessaire pour l'opération de fusion. Cette optimisation permet d'atteindre les **objectifs 1 et 2**. Elle exploite la distribution des données, atteignant ainsi les **objectifs 3 et 4**. De plus, **l'optimisation 2** réduit

l'espace mémoire nécessaire à la fusion, respectant ainsi **l'objectif 2**. Enfin, **l'optimisation 3** porte sur les **objectifs 3 et 4** en s'appuyant sur la distribution des données.

Une validation expérimentale a été réalisée avec plusieurs jeux de données ayant différentes distributions de valeurs minimales. Les résultats montrent que MONTRES réduit le temps d'exécution du tri par 30% en moyenne.

Dans les parties qui suivent, nous détaillons les différentes optimisations exploitées par MONTRES. Nous analysons ensuite les coûts en E/S et la complexité en temps de l'algorithme. Enfin, nous présentons les expérimentations et analysons les résultats.

6.1 DESCRIPTION DE L'ALGORITHME ET DE SES OPTIMISATIONS

Tout comme l'algorithme de tri par fusion, MONTRES se compose de deux phases principales : une phase de **génération des runs** et une phase de **fusion des runs**. L'algorithme de génération des *runs* utilisé dans MONTRES exploite les optimisations **1** et **2**. Pour ce faire, il inclut un **(1) mécanisme de détection des minimums** et un **(2) mécanisme de fusion à la volée** pour mettre en œuvre **l'optimisation (1)**. Il comporte en plus un **(3) mécanisme d'expansion de la taille des runs** pour mettre en œuvre **l'optimisation (2)**.

L'algorithme de fusion des *runs* utilisé dans MONTRES exploite **l'optimisation 3**. Pour ce faire, MONTRES indexe les blocs appartenant aux *runs* avec leurs valeurs minimales. Cet index permet de réaliser la fusion au niveau des blocs et non au niveau des *runs* (comme c'est le cas dans les algorithmes de l'état de l'art). Il permet ainsi de récupérer, au fur à mesure, les valeurs minimales des blocs et de les envoyer vers le fichier final trié.

Dans les sections qui suivent, les optimisations incluses dans la **phase de génération des runs** et la **phase de fusion de runs** sont présentées.

6.1.1 *Optimisation de la phase de génération des runs*

Dans cette section, nous décrivons les trois mécanismes appliqués au niveau de la phase de génération des *runs* (voir Figure 17) : *(1) le mécanisme de sélection des minimums*, *(2) le mécanisme de fusion à la volée* et *(3) le mécanisme d'expansion de la taille des runs*.

6.1.1.1 *Le mécanisme de sélection des minimums*

L'objectif de ce mécanisme est de détecter les valeurs minimales locales (voir la définition 21) réparties dans le fichier en entrée. Pour ce faire, le mécanisme par-

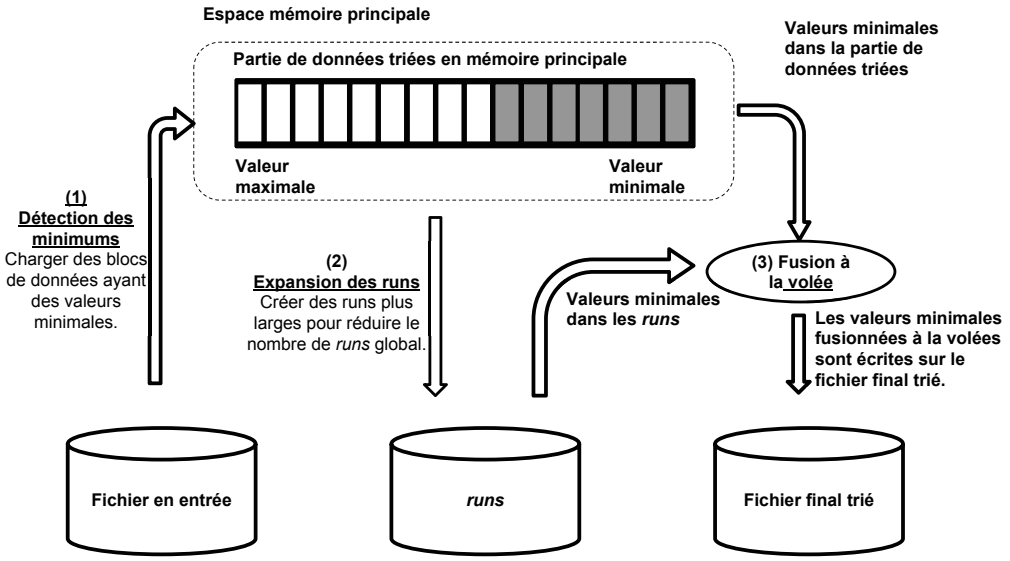

FIGURE 17 : Mécanismes inclus dans l'algorithme de génération des *runs* de MONTRES

court le fichier séquentiellement pour détecter les plus petites valeurs de chaque bloc du fichier. Les valeurs minimales détectées sont insérées dans un index appelé *min-index*. Cet index associe chaque valeur minimale avec le bloc auquel elle appartient dans le fichier.

Ce mécanisme engendre un coût supplémentaire en lecture et en opération CPU également. Le parcours séquentiel du fichier engendre $N \cdot R$ lectures supplémentaires. La détection des minimums locaux nécessite n opérations de comparaison et la construction du *min-index* nécessite $N \cdot \log_2 N$ opérations de comparaison supplémentaires. Par conséquent, la complexité en opérations CPU est évaluée avec cette formule : $O(n + N \cdot \log_2 N)$.

Définition 21. *Une valeur minimale est dite locale lorsque cette valeur est inférieure à toutes les autres valeurs dans un bloc du fichier en entrée.*

Une fois *min-index* créé, les données sont chargées en mémoire principale à partir du fichier en entrée. M blocs de données contenant les M plus petites valeurs minimales dans *min-index* sont chargés à la fois. Dès lors qu'un bloc est chargé en mémoire principale, son entrée est supprimée dans *min-index*.

Les valeurs minimales chargées en mémoire principale sont inférieures à toutes les autres valeurs minimales locales restantes dans *min-index*. Le nombre de valeurs minimales regroupées en mémoire principale est estimé à M dans le pire cas (une valeur minimale locale par bloc chargé en mémoire principale).

Une fois les données chargées, elles sont triées en utilisant un algorithme de tri en mémoire principale. Le tri rapide (*QuickSort*) est utilisé. Cet algorithme de tri présente plusieurs propriétés intéressantes. La première de ces propriétés est qu'il trie des données sur place, c'est-à-dire que l'algorithme n'alloue pas

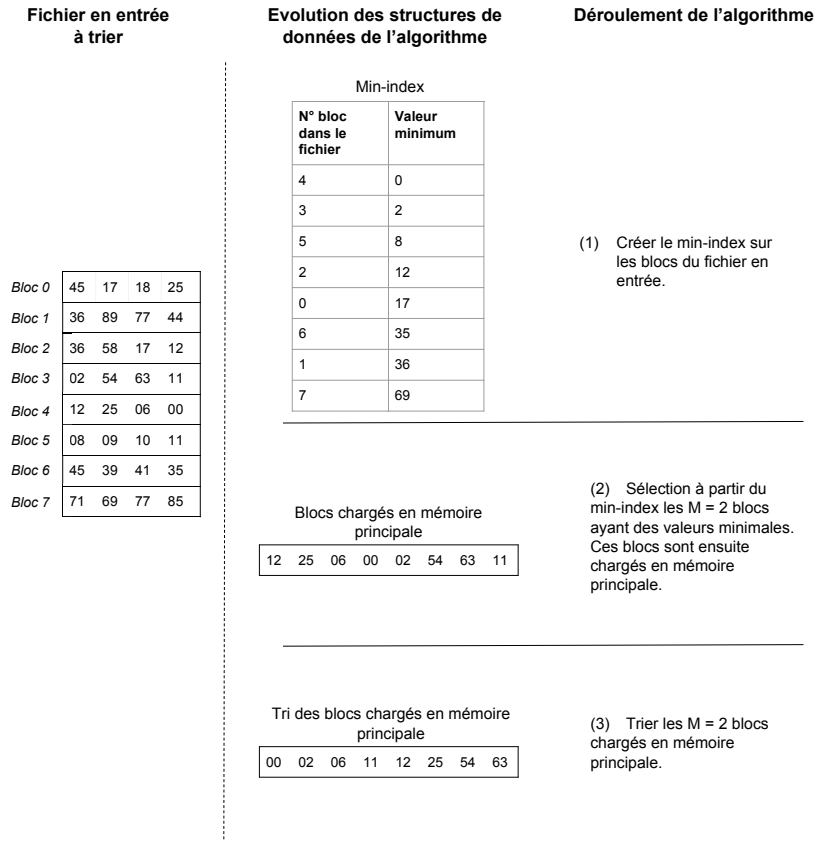

FIGURE 18 : Déroulement du mécanisme de sélection des minimums sur un fichier contenant 8 blocs

d'espace mémoire supplémentaire. Deuxièmement, il est très efficace pour trier des données aléatoires avec une complexité asymptotique de $O(n \cdot \log_2 n)$.

Les données sont alors triées en mémoire principale, contrairement à l'algorithme de fusion traditionnel, nous n'écrivons pas toutes les données triées dans le fichier de *run* intermédiaire. MONTRES écrit seulement une partie des données supérieures à la valeur minimale suivante dans *min-index*. Les valeurs restantes sont gardées en mémoire et seront écrites dans le fichier final trié par le mécanisme de fusion à la volée.

L'exemple 9 présente le déroulement du mécanisme de sélection des minimums sur un fichier contenant $N = 8$ blocs.

Exemple 9. *Dans la Figure 18, le mécanisme de sélection des minimums est appliqué sur un fichier contenant* 8 *blocs de* 4 *valeurs. Le fichier en entrée est parcouru bloc par bloc. Pour chacun des blocs, l'algorithme cherche la valeur minimale et ajoute une entrée dans min-index (Numéro bloc, valeur minimale). Une fois le fichier parcouru et tous les blocs traités, l'index est trié selon les valeurs minimales des blocs.*

6.1.2 *Le mécanisme de fusion à la volée*

Le principal intérêt du mécanisme de fusion à la volée est de raccourcir le chemin vers le fichier final trié pour un ensemble de valeurs minimales. Pour ce faire, ce mécanisme récupère les valeurs minimales locales inférieures à l'actuelle valeur minimale dans *min-index*. Ces valeurs minimales sont récupérées à partir des données triées en mémoire principale mais aussi à partir des *runs* déjà créés. Elles sont écrites directement dans le fichier final trié.

Lors de la génération d'un run, ce mécanisme considère toutes les valeurs en mémoire inférieures à la plus petite valeur dans *min-index*. Ces valeurs en mémoire principale sont fusionnées avec les valeurs des *runs* générés précédemment et inférieures à la plus petite valeur minimale dans *min-index*. Le résultat de la fusion est écrit directement dans le fichier final trié.

Les valeurs minimales transférées de la mémoire principale vers le fichier final trié ne sont pas écrites dans un fichier *run* intermédiaire.

La quantité de valeurs envoyées directement dans le fichier final trié, notée γ, dépend des caractéristiques intrinsèques de l'ensemble de données. Nous distinguons dans le cadre de cette thèse trois types d'ensemble de données : **(1)** données aléatoires, **(2)** données triées ou partiellement triées et **(3)** données redondantes. Ces ensembles de données sont décris dans le chapitre 5.

Dans le cas de données aléatoires, les valeurs minimales sont réparties sur les N blocs du fichier en entrée (voir l'hypothèse 3), par conséquent MONTRES récupère $\gamma = M$ valeurs minimales à chaque fois qu'un *run* est généré. Cette quantité est négligeable comparée au nombre de valeurs n dans le fichier.

Dans le cas de données triées, les valeurs minimales sont regroupées dans un sous ensemble de blocs du fichier en entrée. Par conséquent, un bloc sélectionné par le mécanisme de sélection des minimums réunit d'autres valeurs minimales inférieures à la valeur minimale suivante dans le *min-index*. Ainsi, MONTRES peut récupérer jusqu'à $\gamma = m$ valeurs minimales à chaque génération d'un *run*.

Lorsque les données sont partiellement triées, des blocs contenant des valeurs minimales peuvent contenir aussi une quantité de valeurs maximales, notée λ. Cette quantité λ est écrite dans les fichiers intermédiaires *runs*. Ainsi, la quantité de valeurs envoyées directement dans le fichier final trié lors de la génération d'un *run* est donnée par $\gamma = m - \lambda$. Lorsque les données sont totalement triées, la valeur de λ s'approche de 0. Sinon lorsque les données sont partiellement triées, λ est compris entre M et m, $M < \lambda < m$.

Lorsque les données sont redondantes, une valeur minimale se répète $\frac{B}{D}$ fois dans un même bloc (voir l'hypothèse 5). Par conséquent, les M blocs sélectionnés par le mécanisme de sélection des minimums nous fournissent $\gamma = M \cdot \frac{B}{D}$ (soit $\frac{B}{D}$ valeurs minimales par bloc).

Tableau 4 : Quantités de valeurs minimales récupérées γ

Données aléatoires	Données partiellement triées	Données redondantes
$\gamma = N$	$\gamma = \frac{N \cdot m}{M}$	$\gamma = \frac{N \cdot B}{D}$

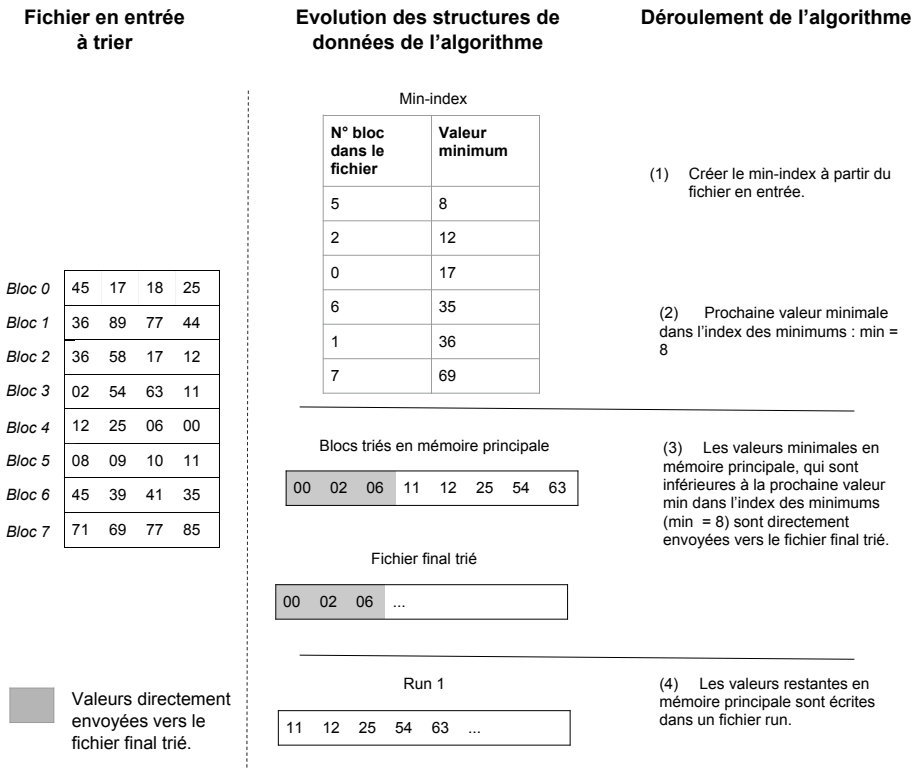

FIGURE 19 : Déroulement du mécanisme de fusion à la volée lors de la création du premier run

Le tableau 4 résume les différentes quantités de valeurs minimales γ récupérées et envoyées directement vers le fichier final trié en fonction des caractéristiques d'un ensemble de données.

L'exemple 10 présente le déroulement du mécanisme de fusion à la volée sur un fichier contenant $N = 8$ blocs.

Exemple 10. *Dans les Figures 19 et 20, on montre le déroulement de la fusion à la volée respectivement lors de la création du premier et du deuxième run. Lors de la création du premier run, les valeurs minimales en mémoire sont directement envoyées vers le fichier final trié, alors qu'avec la création des autres runs, les valeurs minimales en mémoire sont fusionnées avec les valeurs minimales des runs créés précédemment.*

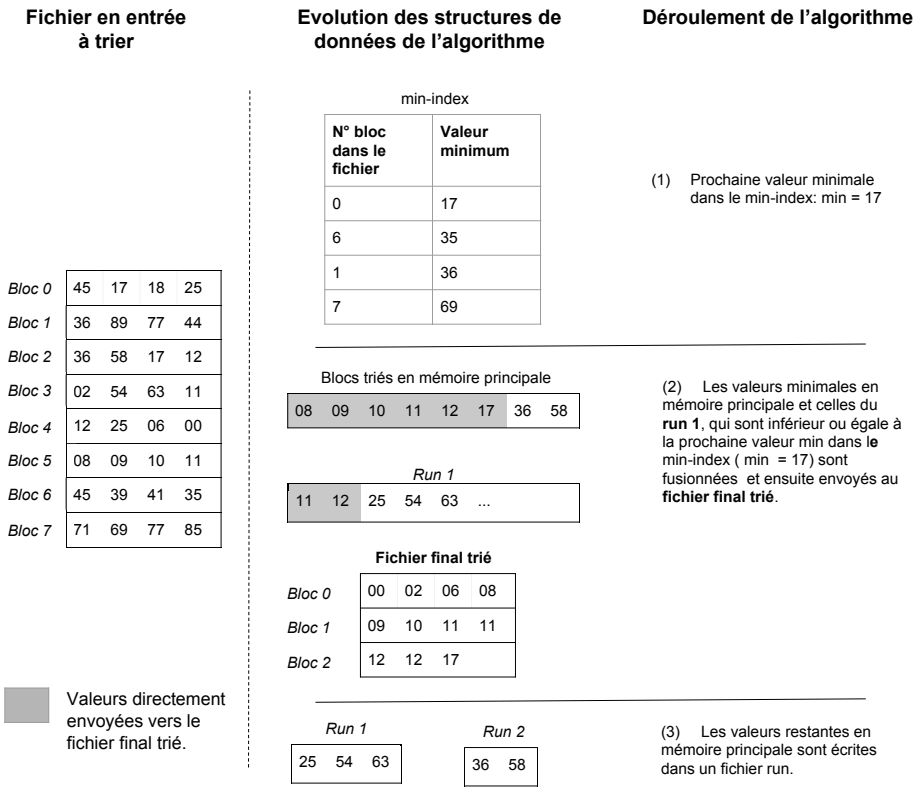

FIGURE 20 : Déroulement du mécanisme de fusion à la volée lors de la création du deuxième run

6.1.3 *Le mécanisme d'expansion des runs*

Le rôle de ce mécanisme consiste à étendre la taille des *runs*, réduisant ainsi le nombre de *runs* générés et au même temps le coût en E/S de la phase de fusion.

Lorsqu'un *run* de M blocs est généré, ce mécanisme essaie d'étendre sa taille au maximum en rajoutant des blocs dont les valeurs sont supérieures à la valeur maximale du *run*, que l'on note V_{max}. Pour ce faire, le mécanisme cherche dans *min-index* les blocs ayant des valeurs minimales supérieures à V_{max}. Ces blocs sont alors chargés en mémoire principale, ensuite triés avec un algorithme de tri en mémoire principal et enfin le résultat est écrit à la fin du *run* généré pour étendre la taille de ce dernier.

Tout comme le mécanisme de fusion à la volée, l'expansion des *runs* dépend des caractéristiques intrinsèques des données. Nous distinguons alors les mêmes cas que lors de l'évaluation du mécanisme de fusion à la volée : **(1)** données aléatoires, **(2)** données triées ou partiellement triées et **(3)** données redondantes.

Lorsque l'ensemble contient des données aléatoires, les N valeurs minimales et maximales sont réparties sur les N blocs du fichier en entrée (voir l'hypothèse 3). La génération d'un *run* réunit M valeurs minimales et maximales. Il est

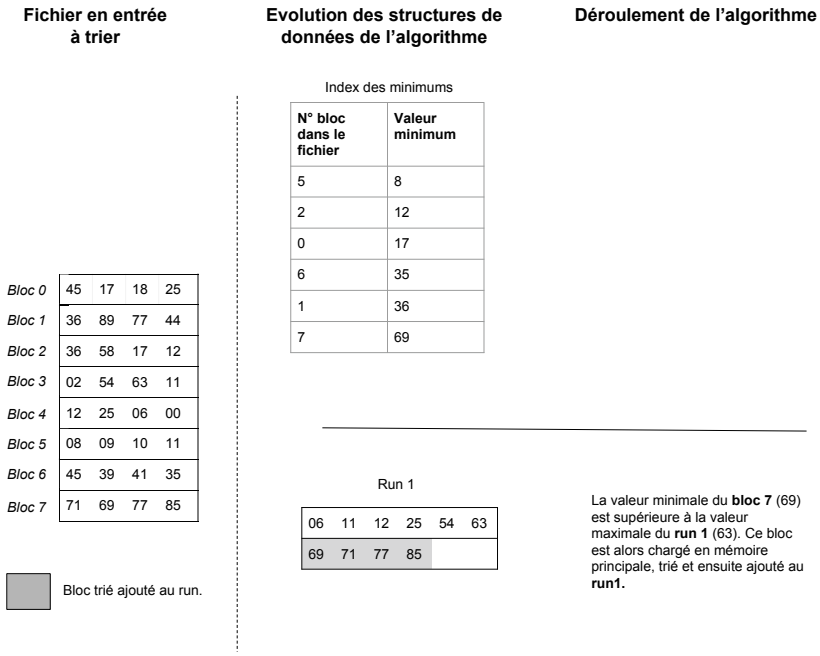

FIGURE 21 : Déroulement du mécanisme d'expansion des runs

donc impossible de trouver dans *min-index*, un bloc ayant une valeur minimale supérieure à la valeur maximale dans le *run* généré.

Lorsque les données sont initialement triées **(2)**, dans l'ordre ascendant ou descendant, le mécanisme d'expansion des *runs* est capable de créer directement le fichier final trié. En effet, l'index contient une série de blocs b_i. $\forall i$, $0 < i < N$, toutes les valeurs du bloc b_i sont supérieures aux valeurs du bloc b_{i-1}. Par conséquent, le premier *run* généré est étendu avec des blocs sélectionnés de *min-index* jusqu'au dernier bloc. Le *run* généré est considéré comme un fichier final trié.

Lorsque l'ensemble contient des données redondantes, une valeur minimale/-maximale est répétée sur l'ensemble des blocs. Par conséquent, les blocs ont tous la même valeur minimale/maximale. Il est donc peu probable de trouver un bloc pouvant étendre un fichier *run*.

Nous illustrons ce mécanisme avec l'exemple 11.

Exemple 11. *Dans la Figure 21, on montre le déroulement de l'expansion des runs sur le premier run créé :* $\mathtt{run}_0$. *Etant donné que la valeur maximale du run est égale à* $V_{max} = 63$ *et que la valeur minimale du* $\mathtt{bloc}_7$, *égale à* 69, *est supérieure à* V_{max}, *le* $\mathtt{bloc}_7$ *est alors chargé en mémoire, trié et inséré à la fin du* $\mathtt{run}_0$.

6.1.4 *Préparation de la phase de fusion des runs*

Les *runs* créés sont écrits physiquement en mémoire secondaire dans des fichiers. Afin de réaliser la fusion des *runs*, le mécanisme de fusion intégré dans MONTRES a besoin d'informations concernant la distribution des données dans les blocs contenant les *runs*. Par conséquent, un second index de minimums est créé au fur à mesure que les *runs* sont élaborés : celui-ci est appelé *run-index*. Cet index contient une entrée pour chaque bloc appartenant à un des *runs* générés. Chaque entrée est constituée de l'adresse du bloc accompagnée de la valeur minimale de ce bloc.

6.2 OPTIMISATION DE LA PHASE DE FUSION DES RUNS

MONTRES intègre une technique de fusion des *runs* permettant de réaliser la fusion en une seule itération, même avec un nombre de *runs* qui dépasse le nombre de blocs en mémoire principale M. Par conséquent, l'algorithme réduit la quantité de données intermédiaires, réduisant ainsi le coût en E/S.

Contrairement aux techniques de fusion de l'état de l'art qui récupèrent les valeurs minimales à partir des *runs*, celle utilisée dans MONTRES récupère les valeurs minimales à partir des blocs sélectionnés dans le *run-index*. Ces blocs sélectionnés contiennent donc les plus petites valeurs minimales dans le *run-index*.

Nous décrivons dans les paragraphes ci-après l'opération de fusion des *runs* et la structure de données.

6.2.1 *Sélection et fusion des blocs*

La sélection des blocs dépend essentiellement du nombre de *runs* créés (au maximum $\frac{N}{M}$) et du nombre de blocs en mémoire principale ($M-1$ blocs utilisés comme tampons de lecture, le bloc restant est utilisé comme tampon d'écriture).

On distingue deux cas différents pour la fusion des *runs*, selon les valeurs de $\frac{N}{M}$ et $M-1$. Le premier cas se produit lorsque le nombre de runs générés $\frac{N}{M}$ est inférieur ou égale au nombre de tampons de lecture alloués en mémoire principale $M-1$. Le deuxième cas se produit lorsque $\frac{N}{M}$ est supérieur à $M-1$.

Dans le premier cas ($\frac{N}{M} \leqslant M-1$), l'algorithme sélectionne $M-1$ blocs à partir du *run-index*. Ces blocs sont chargés dans les tampons de lecture à partir des fichiers de *runs* intermédiaires.

Dans le deuxième cas ($\frac{N}{M} > M-1$), la sélection des blocs doit obéir à une stratégie délicate qui permet de réunir toutes les valeurs minimales des $\frac{N}{M}$ *runs* générés. Cette stratégie est détaillée dans les paragraphes qui suivent.

MONTRES commence par sélectionner $M-1$ blocs à partir du *run-index*. Ces blocs contiennent les $M-1$ premières valeurs minimales dans l'index.

Les $M-1$ blocs sélectionnés sont chargés dans les tampons de lecture alloués en mémoire principale. Les entrées correspondantes à ces blocs dans le *run-index* sont supprimées.

MONTRES commence alors l'opération de fusion sur les $M-1$ blocs chargés dans les tampons de lecture. L'algorithme de fusion récupère toutes les valeurs minimales inférieures à l'actuelle valeur minimale dans *run-index*, notée $next-min$ (voir la définition 22). Ces valeurs sont directement écrites dans le fichier final trié. Lorsque toutes les valeurs d'un tampon de lecture sont inférieures à celle de $next-min$, elles sont alors toutes fusionnées et écrites dans le fichier final trié. Le tampon de lecture est donc vidé.

Définition 22. *Nous appelons* $next-min$*, la plus petite valeur minimale dans* $run-index$*.*

L'opération de fusion est suspendue dès que la valeur minimale dans les tampons de lecture est supérieure à $next-min$. Dans ce cas, l'algorithme de fusion distingue deux cas :

- **cas 1** : au moins un tampon de lecture est vide.
- **cas 2** : les $M-1$ tampons de lecture contiennent encore des valeurs à fusionner.

Dans le premier cas, de nouveaux blocs sont sélectionnés à partir du *run-index*. L'algorithme charge ces blocs dans les tampons vides et continue l'opération de fusion.

Dans le deuxième cas, l'algorithme engage deux techniques pour libérer des tampons de lecture : (1) la fusion des tampons de lecture ou (2) l'éviction des maximums.

FUSION DES TAMPONS DE LECTURE : Cette première technique consiste à détecter des tampons à moitiés vides et à les fusionner deux à deux pour libérer quelques tampons de lecture.

EVICTION DES MAXIMUMS : MONTRES fait appel à cette deuxième technique lorsque la première ne peut être appliquée. L'éviction des maximums consiste à libérer les tampons de lecture contenant des valeurs maximales. Les données contenues dans les tampons de lecture libérés sont référencées dans le *run-index* afin d'être ré-utilisés dans une itération ultérieure.

Une fois l'une des deux techniques appliquée, l'algorithme charge les blocs sélectionnés dans les tampons de lecture libérés et démarre l'opération de fusion.

La fusion des blocs est faite à l'aide d'une structure de données en mémoire principale. Cette structure est décrite dans la section suivante.

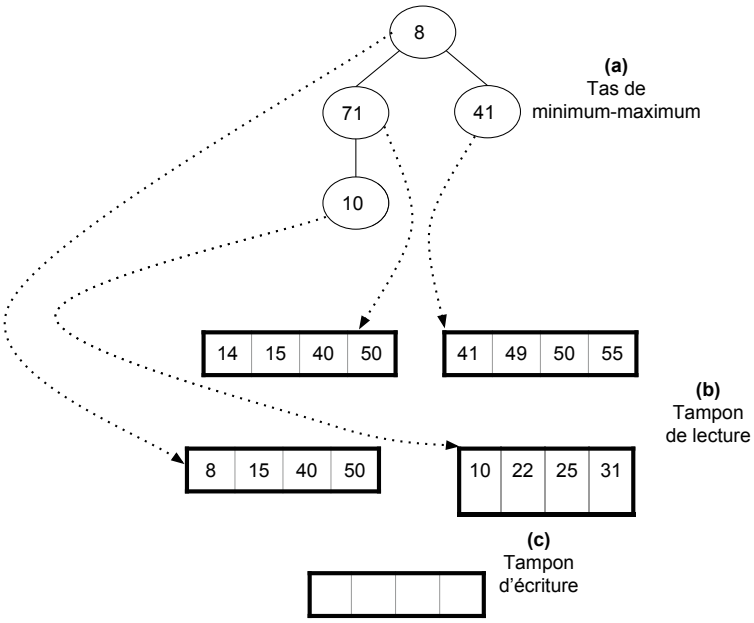

FIGURE 22 : Structure de de données utilisée par le le mécanisme de fusion dans MONTRES

6.2.2 *Structure de données utilisées dans l'opération de fusion des runs*

L'espace alloué en mémoire principale pour l'opération de fusion est réparti en trois parties. La première **(a)** contient la structure de données de l'opération de fusion des *runs* (voir **(a)** dans Figure 22), la deuxième partie **(b)** contient les tampons de lecture ($M-1$ blocs) et la troisième partie **(c)** est le tampon d'écriture de la taille d'un bloc.

MONTRES utilise un tas de minimum-maximum [10] comme structure de données. Son principal avantage est qu'elle permet l'accès aux valeurs minimales et maximales avec une complexité de $O(1)$. L'insertion d'un nouvel élément est majoré avec une complexité logarithmique sur le nombre d'éléments de la structure de données.

6.3 EVALUATION DES COÛTS D'E/S POUR MONTRES

Les optimisations intégrées dans MONTRES se basent en grande partie sur la distribution des données. Ces optimisations supposent que les valeurs minimales sont corrélées entre elles à l'intérieur des blocs. C'est une hypothèse valable pour les données partiellement triées et redondantes, mais qui n'est plus valable lorsque les données sont aléatoires. Il est donc nécessaire d'étudier les perfor-

mances de MONTRES pour les trois types de distribution de données étudiés dans le chapitre 5, section 5.1.

Dans l'évaluation du coût d'E/S de MONTRES, nous distinguons les coûts des deux phases : les coûts pour la génération des *runs* et ceux pour la fusion des *runs*. Nous étudions ces coûts dans les sections 6.3.1 et 6.3.2

6.3.1 *Coût en E/S pour la génération des runs*

La phase de génération des *runs* comporte trois mécanismes : **(1)** la sélection des blocs, **(2)** la fusion à la volée et **(3)** l'expansion des *runs*. Nous évaluons les coûts en E/S de ces mécanismes dans les paragraphes suivants.

6.3.1.1 *Coût en E/S du mécanisme de sélection des blocs*

Ce mécanisme commence par créer *min-index* sur l'ensemble du fichier en entrée. Pour ce faire, le mécanisme parcourt tout le fichier en entrée avec N opérations de lecture. Par conséquent, le coût en E/S est donné par l'équation 12 ci-après.

$$\text{Coût\_ES} = N \cdot R \tag{12}$$

La génération des *runs* nécessite N opérations de lecture pour charger les blocs en mémoire principale et N opérations d'écriture pour écrire les blocs en mémoire secondaire. Ces données sont soit écrites dans le fichier final trié, soit écrites dans un fichier *run* intermédiaire. Cela engendre un coût en E/S donné par l'équation 13.

$$\text{Coût\_ES} = N \cdot (R + W) \tag{13}$$

La génération des *runs* dans MONTRES est accompagnée du mécanisme de fusion à la volée, dont le rôle est d'envoyer les valeurs minimales directement vers le fichier final trié. Nous étudions le coût de ce mécanisme dans la section suivante (6.3.1.2).

6.3.1.2 *Coût en E/S du mécanisme de fusion à la volée*

Le mécanisme de fusion à la volée réalise P_R lectures supplémentaires pour lire, à partir des *runs*, les blocs contenant des valeurs minimums. Le mécanisme réalise aussi $P_W + \Gamma$ écritures supplémentaires pour écrire l'ensemble de données fusionnées à la volée dans le fichier final trié. Nous obtenons alors le coût en E/S donné par l'équation 14.

$$\text{Coût\_ES} = (P_R \cdot R) + (P_W \cdot W) \tag{14}$$

Le mécanisme de fusion à la volée, envoie directement dans le fichier final trié γ valeurs minimales, soit Γ blocs de données.

6.3.1.3 *Coût en E/S du mécanisme d'expansion des runs*

Le mécanisme d'expansion des *runs* réalise des opérations de lecture et d'écriture de blocs pour étendre la taille des runs. Ces opérations sont comptabilisées dans le coût en E/S de la génération des *runs* (équation 13). Àla fin de la génération des runs, il reste $N - P_W - \Gamma$ blocs de données à fusionner. Ces blocs sont regroupés et organisés dans le *run-index*.

6.3.1.4 *Coût en E/S total de la génération des runs*

Pour obtenir le coût en E/S de la phase de génération des *runs* (RGRC + RGWC), il suffit de faire la somme des coûts donnés par les formules 12, 13 et 14. Nous obtenons alors le coût total en E/S dans l'équation 15.

$$\begin{aligned} RGRC + RGWC &= [N \cdot R] + [N \cdot (R + W)] + [P_R \cdot R + P_W \cdot W] \\ &= (2 \cdot N + P_R) \cdot R + (N + P_W) \cdot W \end{aligned} \tag{15}$$

6.3.2 *Coût en E/S de la phase de fusion des runs*

MONTRES se base sur *run-index* pour sélectionner les blocs à fusionner. Il sélectionne en premier les blocs contenant des valeurs minimales. L'opération de fusion retire les valeurs minimales des blocs sélectionnés. Les valeurs retirées sont envoyées dans le fichier final trié et le restant des valeurs est inséré à nouveau dans *run-index* pour une itération ultérieure.

Lorsque les données sont aléatoires, un bloc de données peut contenir des valeurs minimales et maximales à la fois. Par conséquent, le bloc peut être lu plusieurs fois sur la mémoire secondaire. Nous notons par α le nombre de fois où un bloc intermédiaire est chargé en mémoire principale durant l'opération de fusion. Le coût en E/S de la fusion des *runs* est alors donné par l'équation 16.

$$RMRC + RMWC = (N - P_W - \Gamma + \alpha) \cdot (R + W) \tag{16}$$

Lorsque le nombre de *runs* générés $\frac{N}{M}$ est inférieur à $M - 1$, un bloc intermédiaire est lu qu'une seule fois, on obtient alors $\alpha = 0$. Néanmoins, lorsque le nombre de *runs* générés $\frac{N}{M}$ dépasse $M - 1$, un bloc peut être lu plusieurs fois ($\alpha > 1$).

Le pire cas se produit lorsque les données sont aléatoires. Dans ce cas, chaque *run* contient M valeurs minimales et M valeurs maximales issues des M blocs sélectionnés dans *min-index*. Par conséquent, chaque fois qu'un bloc est chargé

dans un tampon de lecture, au moins M valeurs sont fusionnées pour récupérer les valeurs minimales. Par conséquent, un bloc est chargé $\lceil \frac{B}{M} \rceil$ fois dans un tampon de lecture. Ainsi α est majoré par $N \cdot \lceil \frac{B}{M} \rceil$.

Lorsque les données sont partiellement triées ou redondantes, les valeurs minimales dans les *runs* sont regroupées dans un sous ensemble de blocs et les valeurs maximales dans un second sous ensemble. Par conséquent, un bloc est chargé une seule fois en mémoire principale ($\alpha = 0$).

6.4 ANALYSE ET DISCUSSION DU COÛT EN E/S DE MONTRES

L'équation 15 montre que la phase de génération des *runs* réalise $N + P_R$ opérations de lectures et P_W opérations d'écritures supplémentaires. Ces opérations d'écritures permettent à MONTRES de réduire la quantité de données à fusionner à $N - P_W$.

Le coût total en E/S de l'algorithme MONTRES est donné par l'équation 17.

$$\begin{aligned} \text{Coût\_ES} &= [(2 \cdot N + P_R) \cdot R + (N + P_W) \cdot W] + [(N - P_W - \Gamma + \alpha) \cdot (R + W)] \\ &= (3 \cdot N + P_R + \alpha) \cdot R + (2 \cdot N - \Gamma) \cdot W. \end{aligned} \tag{17}$$

Le coût total en E/S de MONTRES dépend de la distribution des données en entrée : aléatoires, redondantes ou partiellement triées. Dans cette analyse, nous considérons le cas le plus défavorable (données aléatoires) et le cas le plus favorable (données partiellement triées).

Lorsque les données sont aléatoires, nous obtenons $\gamma = N$ valeurs (soit $\Gamma = \frac{N}{B}$ blocs), $P_R = \frac{N}{M}$ et $\alpha = N \cdot \lceil \frac{B}{M} \rceil$. Nous obtenons alors un coût total donné par la formule 18.

$$\text{Coût\_ES} = (3 \cdot N + \frac{N}{M} \cdot (1 + B)) \cdot R + (2 \cdot N - \frac{N}{B}) \cdot W \tag{18}$$

Lorsque les données sont partiellement triées, la valeur de Γ est proche de N et $P_R = 0$. Nous obtenons alors un coût total donné par la formule 19.

$$\text{Coût\_ES} = 2 \cdot N \cdot R + N \cdot W = N \cdot (2 \cdot R + W) \tag{19}$$

En considérant les deux formules 18 et 19, nous obtenons l'inégalité 20 qui nous donne le coût minimum et le coût maximum en E/S.

$$2 \cdot N \cdot R + N \cdot W \leqslant \text{Coût\_ES} \leqslant (3 \cdot N + \frac{N}{M} \cdot (1 + B)) \cdot R + (2 \cdot N - \frac{N}{B}) \cdot W \tag{20}$$

Àpartir de la formule 20, nous constatons que le coût en E/S le plus favorable pour MONTRES est inférieur au coût favorable du tri par fusion ($2 \cdot N \cdot (R + W)$). En effet, MONTRES apporte un gain de N opérations d'écriture ($N \cdot R$).

Concernant le cas défavorable, le coût en E/S du tri par fusion est $N \cdot (1 + \lceil \log_{M-2} \frac{N}{M} \rceil)(R + W)$. Lorsque $\frac{N}{M} > M - 2$, le tri par fusion nécessitent plusieurs itérations $\lceil \log_{M-2} \frac{N}{M} \rceil$ pour produire le fichier final trié. Nous supposons que le nombre d'itérations est égale à 2, nous obtenons alors : $3 \cdot N \cdot R + 3 \cdot N \cdot W$. Dans ce cas, MONTRES réalise $\frac{N}{M} \cdot (1 + B)$ opérations de lectures supplémentaires et produit un gain de $N + \frac{N}{B}$ opérations d'écritures. Ce gain en écriture est supérieur au coût supplémentaire en lecture $(N + \frac{N}{B}) \cdot W > (\frac{N}{M} \cdot (1 + B)) \cdot R$. Par conséquent, MONTRES surpasse le tri par fusion au niveau du coût en E/S.

6.5 VALIDATION EXPERIMENTALE

Dans cette section, nous présentons les expérimentations menées dans le but de valider les optimisations intégrées dans l'algorithme MONTRES.

6.5.1 *Méthodologie d'expérimentation*

Dans un environnement de production, les performances d'un algorithme de tri en mémoire secondaire peuvent dépendre de plusieurs facteurs matériels et logiciels. Parmi les facteurs matériels, nous citons les performances du périphérique de stockage et la quantité de mémoire principale disponible. Quant aux facteurs logiciels, nous citons la configuration du système d'exploitation ainsi que le type de données en entrée.

En considérant les facteurs ci-dessus, nous avons sélectionné les 4 paramètres ci-après, à faire varier dans le cadre de nos expérimentations :

- Le modèle du périphérique de stockage SSD.
- L'espace mémoire alloué pour trier les données.
- Le type de données en entrée pour l'algorithme de tri.
- La configuration du cache des E/S au niveau du système d'exploitation.

Nous présentons ces 4 paramètres dans les sections qui suivent.

6.5.1.1 *Modèles de périphériques de stockage SSD*

Nous avons expérimenté avec trois modèles de SSD. Le premier est conçu pour équiper les ordinateurs personnels (PC SSD). Il porte la référence **SSD 850 PRO series**. Le deuxième SSD est conçu pour équiper les stations de travail (Server

Tableau 5 : Performances des SSD utilisés

SSD	PC SSD	server SSD	DC SSD
Lectures aléatoires (IOPS)	100K	92K	85K
Lectures Séquentielles (MB/s)	550	530	550
Ecritures aléatoires (IOPS)	36K	50K	43K
Ecritures séquentielles (MB/s)	520	460	300

SSD). Cet SSD porte la référence **SSD 845 DC PRO MZ-7WD400EW**. Le dernier SSD est conçu pour équiper les centres de données (Data Center DC SSD). Il porte la référence **intel DC serie S3710**.

Le tableau 5 présente les performances en E/S pour chacun des SSD (PC SSD, Server SSD et DC SSD), ces chiffres sont issues des fiches techniques [1, 2, 4] présentées par les constructeurs de ces SSD. Nous constatons que ces SSD offrent des performances élevées pour les lectures ainsi que les écritures séquentielles, mais les performances sont à la baisse pour les écritures aléatoires, notamment pour le PC SSD.

6.5.1.2 *Espace mémoire alloué pour le tri des données*

Afin de valider la capacité de MONTRES à trier des grands volumes de données dans des espaces mémoires limités, nous avons réalisé des expérimentations avec différents ratio $R_i = \frac{N}{M}$. La taille du fichier sur disque est fixée à 8Go, avec N égale à 2097152 blocs, où chaque bloc contient 1024 entiers, soit 4096 octets. Nous avons fait varier l'espace mémoire alloué M, de manière à avoir plusieurs valeurs de $R_i = \frac{N}{M}$ différentes : $R_1 = 8, R_2 = 32, R_3 = 64, R_4 = 128, R_5 = 512, R_6 = 2048, R_7 = 8192$. Plus la valeur de R_i est grande, plus l'espace mémoire est limité.

6.5.1.3 *Ensemble de données*

Nous avons utilisé des ensembles de données issues du benchmark TPC-H [23]. Ces ensembles de données ont différentes caractéristiques : données aléatoires, données partiellement triées et données redondantes (voir le chapitre 5, section 5.1). Nous décrivons ci-dessous les ensembles de données utilisés dans l'expérimentation.

ENSEMBLE DE DONNÉES ALÉATOIRES : Nous avons utilisé un ensemble de données aléatoires noté **(A)**. Cet ensemble de données est issue de la colonne `quantity` de la table `lineitem` dans la base de données utilisée pour le bench-

mark TPC-H. Cette colonne décrit la quantité de produits achetés par transaction.

ENSEMBLE DE DONNÉES REDONDANTES : Nous avons utilisé un ensemble de données redondantes noté **(B)**. Cet ensemble de données est issu du champs année de la colonne `date` de la table `orders`. Cette colonne renseigne la date d'une commande.

ENSEMBLE DE DONNÉES PARTIELLEMENT TRIÉES : Nous avons utilisé trois ensembles de données partiellement triées : **(C)**, **(D)** et **(E)**. Ces ensembles de données sont générés à partir de l'ensemble **(A)**. Ce dernier est trié puis mis à jour, en variant à chaque fois le pourcentage de valeurs mise à jour dans le fichier : 20% pour **(C)**, 40% pour **(D)** et 60% **(E)**.

6.5.1.4 *Configuration du cache des E/S au niveau du système d'exploitation*

Les systèmes d'exploitation modernes mettent en œuvre un *cache* de données en mémoire principale. Le *cache* est principalement utilisé pour maintenir les données fréquemment utilisées en mémoire principale et réduire ainsi les opérations de lectures et écritures sur la mémoire secondaire.

Nous avons utilisé un système d'exploitation GNU Linux installé sur un serveur DELL *precision* T7910 ayant 32 Go de mémoire principale.

Dans nos expériences, nous avons mesuré les performances de MONTRES avec le **page cache** activé et désactivé.

6.5.2 *Description des expériences*

Dans le cadre de la validation expérimentale de MONTRES, nous avons réalisé quatre expériences que nous décrivons ci-après.

Expérience 1. *L'objectif de cette expérience est de comparer les temps d'exécution des algorithmes de l'état de l'art Minsort et Natural Page Run avec ceux de MONTRES. Cette expérience est réalisée avec le SSD destiné aux serveurs en utilisant les ensembles de données suivants :* ***(A)***, ***(B)*** *et* ***(C)****.*

Expérience 2. *L'objectif de cette expérience est de comparer les temps d'exécution de MONTRES avec ceux de l'algorithme de l'état de l'art FSort [55] optimisé pour les mémoires flash et l'algorithme de tri par fusion traditionnel [41]. Cette expérience est réalisée avec tous les ensembles de données* ***(A)***, ***(B)***, ***(C)***, ***(D)*** *et* ***(E)****. L'expérience utilise tous les SSD présentés dans le tableau 5.*

Expérience 3. *L'objectif de cette expérience est d'étudier l'impact du page cache sur le temps d'exécution de MONTRES. L'expérience mesure les temps d'exécution de MONTRES une première fois en activant le page cache et une deuxième fois en le désactivant.*

Tableau 6 : Résultats obtenus dans l'expérience 1

Ensemble de données	MONTRES	MinSort	Natural Page *run*
(A)	2400 secondes	> 6000 secondes	> 6000 secondes
(B)	1990 secondes	105 secondes	> 6000 secondes
(C)	1670 secondes	> 6000 secondes	3100 secondes

*L'expérience est réalisée avec les ensembles de données aléatoires **(A)**, redondantes **(B)** et partiellement triées : **(C)**, **(D)** et **(E)**. L'expérience utilise le SSD destiné aux serveurs présenté dans le tableau 5.*

Expérience 4. *L'objectif de cette expérience est de mesurer l'impact de chaque mécanisme intégré dans MONTRES sur son temps d'exécution global. L'expérience est réalisée avec les ensembles de données aléatoires **(A)**, redondantes **(B)** et partiellement triées **(C)**. L'expérience utilise le SSD destiné aux serveurs présentés dans le tableau 5.*

6.6 RÉSULTATS ET DISCUSSIONS

Dans cette section, nous présentons et discutons les résultats des expériences menées dans le cadre de la validation de MONTRES.

6.6.1 *Résultats de l'expérience 1*

Nous présentons les résultats obtenus avec l'expérience 1 dans le tableau 6. Ce tableau présente les temps d'exécution mesurés sur les algorithmes MONTRES, MinSort et Natural Page *run* sur les ensembles de données **(A)**, **(B)** et **(C)**.

Résultat 1. *MONTRES réalise le tri de données sur l'ensemble **(A)** au bout de 2400 secondes alors que MinSort et Natural Page run ne produisent pas de résultat trié au bout de 6000 secondes d'exécution.*

Résultat 2. *MinSort réalise le tri de données sur l'ensemble **(B)** au bout de 105 secondes alors que MONTRES produit un résultat au bout de 1990 secondes.*

Résultat 3. *Natural Page run réalise le tri de données sur l'ensemble **(C)** au bout de 3100 secondes, soit 1430 secondes de plus que MONTRES.*

Nous analysons dans ce qui suit les performances de MONTRES avec les deux algorithmes *Minsort* et *Natural Page Run*.

MINSORT Les performances de Minsort dépendent de la distribution des valeurs dans l'ensemble de données à trier. Avec l'ensemble de données **(B)**, Minsort affiche de meilleures performances que MONTRES. Ceci est due au fait que ce jeu de données contient beaucoup de valeurs redondantes, la distribution moyenne des valeurs est égale à $D = 7$.

Néanmoins, cet algorithme atteint ses limites avec un ensemble de données aléatoires comme **(A)**. La distribution moyenne des valeurs dans les blocs de cet ensemble est égale à $D = 993$. L'entropie de cet ensemble de données est égale à $18,9$. Les résultats nous montrent que Minsort est peu performant sur ce type d'ensemble de données. Alors que MONTRES trie 8Go de données en 2400 secondes, Minsort trie seulement 8Mo sur la même durée.

NATURAL PAGE RUN Les performances de cet algorithme dépendent également de la nature du jeu de données. Lorsque celui-ci est trié ou partiellement trié, l'algorithme se trouve alors dans un cas favorable. L'algorithme utilise une heuristique pour maximiser le nombre de *runs* naturels crées. Cette heuristique est très gourmande en calcul et alourdit donc son temps d'exécution sur de grands volumes de données. Alors que MONTRES trie 8Go de données en moins d'une heure, cet algorithme trie la même quantité au bout de deux heures.

6.6.2 *Résultats de l'expérience 2*

Nous présentons et discutons dans cette section les résultats de l'expérience 2. Ces résultats montrent l'accélération apportée par MONTRES et FSort sur les opérations de tri en mémoire secondaire utilisant le tri par fusion traditionnel.

Nous présentons et discutons les résultats séparément dans les paragraphes suivants, en fonction des caractéristiques des données (aléatoires, triées et redondantes).

6.6.2.1 *Données aléatoires*

Ici nous présentons et discutons les résultats obtenus avec l'ensemble de données aléatoires **(A)**.

Présentation des résultats

La Figure. 23 montre l'accélération obtenue par MONTRES et FSort sur des données aléatoires **(A)**. Ces accélérations sont calculées sur les temps d'exécution des phases de génération des *runs* et de fusion des *run*. Les temps d'exécution sont mesurés pour chaque configuration $R_i = \frac{N}{M}$ présentées dans la section 6.5.1.2 en utilisant les trois modèles de SSD : DC, PC et Server présentés dans le tableau 5.

Le tableau 7 présente les mesures des coûts d'E/S réalisées pour MONTRES. La ligne $P_W + \Gamma$ donne la quantité de données écrites directement dans le fichier final trié par le mécanisme de fusion à la volée. La ligne P_R donne le nombre de

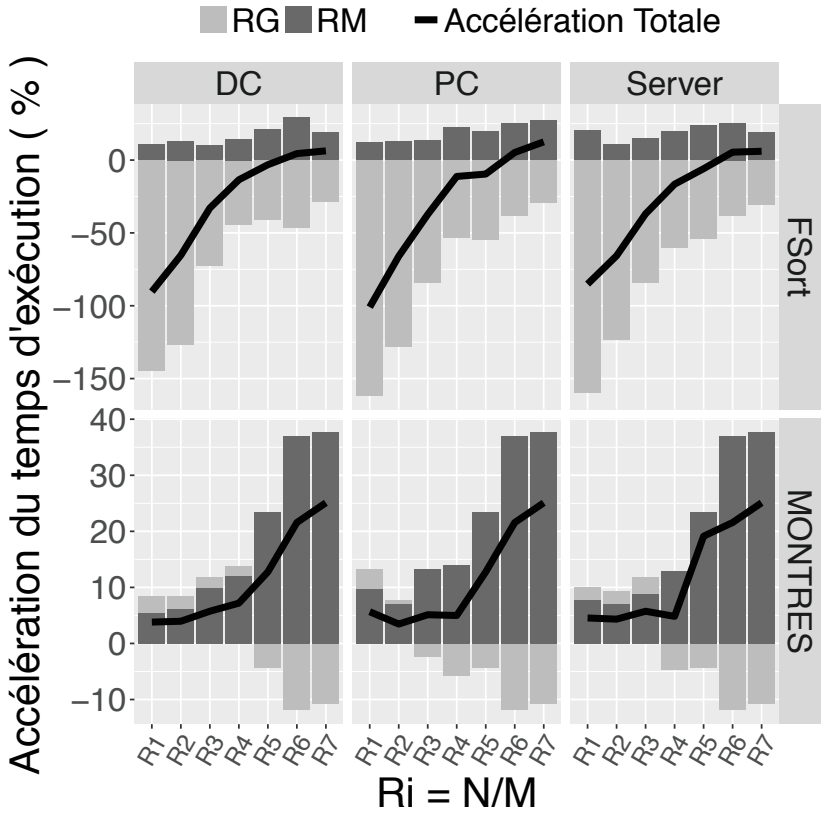

FIGURE 23 : Accélération du temps d'exécution du tri par fusion avec MONTRES et FSort, résultats obtenus avec les ensembles de données (A) sur les trois SSD de type DC, PC et server. **RG** : *Run Generation*, **RM** : *Run Merge*.

lectures supplémentaires nécessaires pour réaliser le mécanisme de fusion à la volée. Nous détaillons ci-après les résultats obtenus à partir du tableau 7 et de la Figure 23.

Résultat 4. *MONTRES et FSort obtiennent leurs accélérations maximales lorsque le ratio* $\frac{N}{M}$ *est égale à* 8192 (R_7)*. Dans ce cas, MONTRES atteint une accélération maximale de* 25% *par rapport au tri par fusion traditionnel et FSort atteint une accélération maximale de* 8% *par rapport au tri par fusion.*

Le résultat 4 s'explique par le fait que MONTRES et FSort embarquent des optimisations pour réduire le coût des E/S lorsque le ratio $\frac{N}{M}$ est élevé. En effet MONTRES réduit le coût en E/S lors de la phase de fusion des *runs* en diminuant la quantité de données à fusionner (mécanisme de fusion à la volée) et en réalisant la fusion au niveau des blocs (mécanisme de fusion des *runs*).

Résultat 5. *Le pire cas pour les deux algorithmes MONTRES et FSort se produit lorsque le ratio* $\frac{N}{M}$ *est égale à* 8 (R_1)*. Dans ce cas, MONTRES affiche une accélération de* 5% *avec l'ensemble de données (A) et FSort alourdit les temps d'exécution de* 141% *en moyenne sur les trois modèles de SSD (DC, PC et server).*

Le résultat 5, concernant l'accélération minimale s'explique par le fait que l'algorithme de tri par fusion est dans un cas favorable avec un nombre de *runs* générés $\frac{N}{M} = 8$ inférieur à $M-2$ ($M-2 = 262144$ blocs). L'opération de fusion des *runs* est alors réalisée en une seule passe : $\lceil \log_{M_2} \frac{N}{M} \rceil = 1$.

Résultat 6. *MONTRES accélère la phase de fusion des runs de 5% à 25%, alors que FSort ne l'accélère pas plus de 6%.*

Le résultat 6 révèle que l'accélération apportée par MONTRES est jusqu'à 8 fois plus importante que celle apportée par FSort. Ce résultat s'explique par le fait que MONTRES, avec son mécanisme de fusion à la volée, réduit la quantité des données à fusionner (de 6.6% à 7.3% de N, la taille du fichier en entrée, voir la ligne $P_W + \Gamma$ dans le tableau 7). Par ailleurs, MONTRES réalise la fusion avec une seule passe sans création de données intermédiaires, alors que FSort réalise la fusion avec $\lceil \log_{M-2} \frac{N}{2M} \rceil$ passes.

Résultat 7. *Avec un ratio $\frac{N}{M}$ de 8, 32 et 64 (respectivement R_1, R_2 et R_3), MONTRES réduit en moyenne le temps d'exécution de la phase de génération des runs de 2% et celle de la phase de fusion des runs de 7%.*

D'après le résultat 7, MONTRES accélère la phase de génération des *runs* lorsque le ratio $\frac{N}{M}$ est égale à $8, 32$ et 64.

Ce résultat s'explique par deux principales raisons que nous détaillons ci-dessous :

- **(1)** Le mécanisme de fusion à la volée sollicite moins de données en mémoire secondaire, parce que le nombre de *runs* générés $\frac{N}{M}$ est bas.
- **(2)** Les parties de données sont formées en sélectionnant des blocs contenant des valeurs minimums ascendantes, ce qui permet de réduire le nombre d'opérations de comparaisons faites par l'algorithme de tri en mémoire, le tri rapide dans le cas de MONTRES.

Résultat 8. *Avec le ratio $\frac{N}{M}$ dépassant la valeur de 64 (R_3), MONTRES ralentit en moyenne la phase de génération des runs de 4% et augmente en moyenne l'accélération de la phase de fusion des runs de 20% .*

Résultat 9. *FSort ralentit le temps d'exécution de la phase de génération des runs de 71% en moyenne et réduit le temps d'exécution de la phase de fusion des runs de 7%.*

D'après les résultats 8 et 9, MONTRES et FSort augmentent le temps d'exécution de la génération des runs. Le résultat 8 s'explique par le fait que MONTRES réalise des opérations de lectures et d'écriture supplémentaires afin de réduire la quantité de données à fusionner. Quant au résultat 9, il s'explique par les opérations CPU supplémentaires permettant de créer des *runs* plus larges.

Tableau 7 : Mesure des coûts d'E/S pour MONTRES avec l'ensemble de données (A)

R_i	R_1	R_2	R_3	R_4	R_5	R_6	R_7
$P_W + \Gamma(\%N)$	7.3	7.1	7.0	6.9	6.7	6.7	6.6
$P_R(\%N)$	1.8	1.9	1.9	2.0	2.0	2.0	2.3

Résultat 10. *Plus l'espace mémoire M est petit, plus le temps d'exécution de la phase de génération des runs est réduit et le temps d'exécution de la phase de fusion des runs est élevé.*

Le résultat 10 s'explique par deux raisons que nous détaillons ci-dessous :

1. Le nombre d'opérations de comparaisons réalisées durant la phase de génération des *runs* dépend de la taille mémoire allouée M. Plus M est élevé plus le nombre d'opérations de comparaisons est élevé et inversement.

2. Le coût en E/S de la phase de fusion dépend du nombre de *runs*, ce dernier dépend de la taille de l'espace mémoire alloué M. Plus l'espace mémoire alloué M est petit, plus le nombre de *runs* générés est élevé et plus le coût en E/S est élevé.

Résultat 11. *La quantité de données $P_W + \Gamma$ écrites directement dans le fichier final trié par le mécanisme de fusion à la volée reste stable autour de 6.6% et 7.3% de N, la taille du fichier en entrée. Il en va de même pour le nombre de lectures supplémentaires P_R qui reste stable autour de 1.8% et 2.3% de N. Malgré ces valeurs stables de $P_W + \Gamma$ et P_R, l'accélération de la phase de fusion est de 6% lorsque $\frac{N}{M}$ est égal à 8 et cette accélération atteint 37% lorsque $\frac{N}{M}$ atteint la valeur de 8192.*

Le résultat 11 s'explique par le fait que MONTRES réalise l'opération de fusion en une seule passe alors que les autres algorithmes le font en $\lceil \log_{M-2} \frac{N}{M} \rceil$ passes.

6.6.2.2 *Données redondantes*

Dans cette section, nous présentons et discutons les résultats obtenus avec l'ensemble de données redondantes : **(B)**.

Présentation des résultats

La Figure. 24 montre l'accélération obtenue par MONTRES et FSort sur l'ensemble de données **(B)**).

Le tableau 8 présente les mesures des coûts d'E/S réalisées sur MONTRES. La ligne $P_W + \Gamma$ donne la quantité de données écrites directement dans le fichier

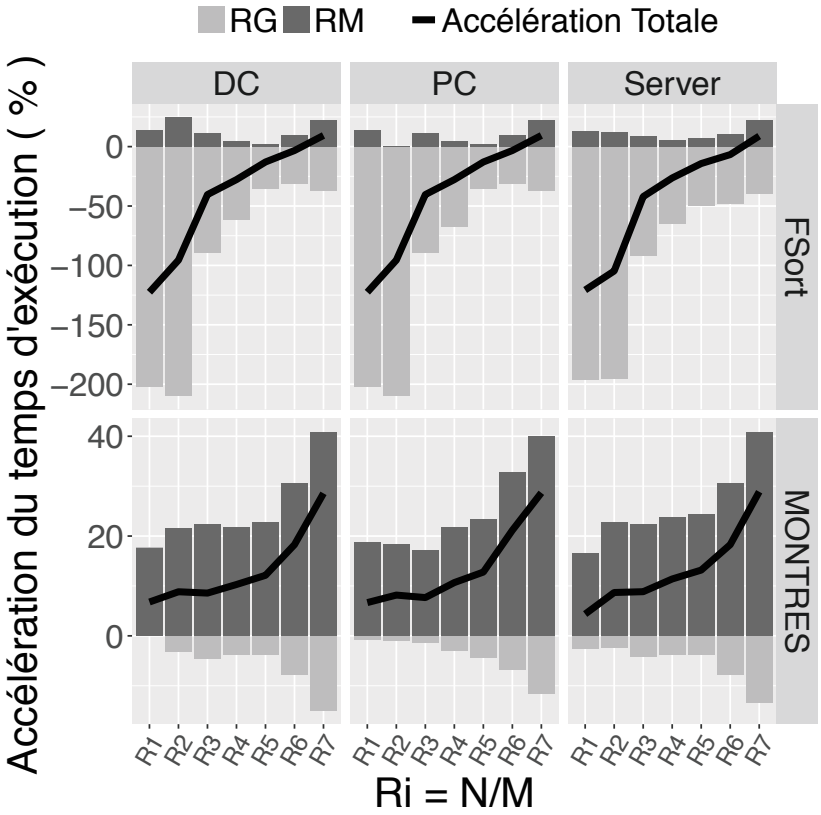

FIGURE 24 : Accélération du temps d'exécution du tri par fusion avec MONTRES et FSort, résultats obtenus avec les ensembles de données **(B)** sur les trois SSD de type DC, PC et server.

final trié par le mécanisme de fusion à la volée. La ligne P_R donne le nombre de lectures supplémentaires nécessaires pour réaliser la fusion à la volée.

Nous détaillons ci-après les résultats obtenus à partir du tableau 8 et de la Figure 24.

Résultat 12. *MONTRES et FSort obtiennent l'accélération maximale lorsque le ratio* $\frac{N}{M}$ *est de* 8192 *(*R_7*). Dans ce cas, MONTRES atteint son accélération maximale de* 28% *avec l'ensemble de données* ***(B)*** *et FSort atteint une accélération maximale de* 8%*.*

Résultat 13. *MONTRES réduit de* 20% *le temps d'exécution de la phase de fusion des runs pour un ratio de* $\frac{N}{M}$ *égale à* 8, 32, 64, 128 *et* 512*. L'algorithme réduit respectivement de* 32% *et* 40% *le temps d'exécution de la phase de fusion, respectivement pour* $\frac{N}{M} = 2048$ *et* $\frac{N}{M} = 8192$*.*

Résultat 14. *L'accélération obtenue avec MONTRES est en moyenne plus élevée de* 10% *sur l'ensemble de données* ***(B)*** *par rapport à celles obtenues avec l'ensemble de données* ***(A)****.*

Résultat 15. *La quantité* $P_W + \Gamma$ *envoyée directement vers le fichier final trié reste stable autour de* 24% *de la taille du fichier en entrée* N*. Cette quantité est* 3 *fois plus élevée que celle mesurée sur l'ensemble de données* ***(A)****.*

Tableau 8 : Mesure des coûts d'E/S pour MONTRES avec l'ensemble de données **(B)**

R_i	R_1	R_2	R_3	R_4	R_5	R_6	R_7
$P_W + \Gamma(\%N)$	24.2	24.2	24.1	24.3	24.3	24.3	24.1
$P_R(\%N)$	8.2	8.3	8.0	8.0	8.1	8.1	8.1

Discussion

D'après le résultat 12, l'accélération obtenue par MONTRES / FSort atteint un pic lorsque $\frac{N}{M} = 8192$. Ce résultat s'explique par le fait que MONTRES et FSort exploitent des optimisations pour réduire le coût des E/S lorsque le ratio $\frac{N}{M}$ est élevé (8192 dans le cadre de nos expériences). En effet, MONTRES réduit le coût en E/S lors de la phase de fusion des *runs* en réduisant la quantité de données à fusionner (mécanisme de fusion à la volée) et en réalisant la fusion au niveau des blocs (mécanisme de fusion des *runs*).

Lorsque le ratio $\frac{N}{M}$ est inférieur à $M-1$, c'est le cas avec $\frac{N}{M} = 8, 32, 64, 128, 512$, MONTRES se base sur les optimisations (1) et (2) pour réduire le temps d'exécution. Il obtient alors une accélération de 20% comparée au tri par fusion traditionnel (voir le résultat 13).

Lorsque $\frac{N}{M}$ est supérieur à $M-1$, c'est le cas avec $\frac{N}{M} = 2048, 8192$. MONTRES se base sur les optimisations (1), (2) et (3) pour réduire le temps d'exécution, il obtient alors une accélération de 32% pour $\frac{N}{M} = 2048$ et 40% pour $\frac{N}{M} = 8192$ (voir le résultat 13).

Le résultat 15 s'explique par le fait que l'ensemble de données **(B)** contient des valeurs redondantes ; le nombre moyen de valeurs différentes dans un bloc est égale à $D = 7$. Par conséquent, le mécanisme de fusion à la volée permet de regrouper une plus grande quantité de valeurs minimales redondantes. Ces dernières sont envoyées directement au fichier final trié court-circuitant ainsi la phase de fusion des runs.

6.6.2.3 *Données partiellement triés*

Dans cette section, nous présentons et discutons les résultats obtenus avec les ensembles de données **(C)**, **(D)** et **(E)**.

Présentation des résultats

Les Figures 25, 26 et 27 montrent l'accélération obtenues par MONTRES et FSort sur des données partiellement triées **(C)**, **(D)** et **(E)**.

Le tableau 9 présente les mesures des coûts d'E/S réalisées sur MONTRES. La ligne $P_W + \Gamma$ donne la quantité de données écrites directement dans le fichier final trié par le mécanisme de fusion à la volée. La ligne P_R donne le nombre

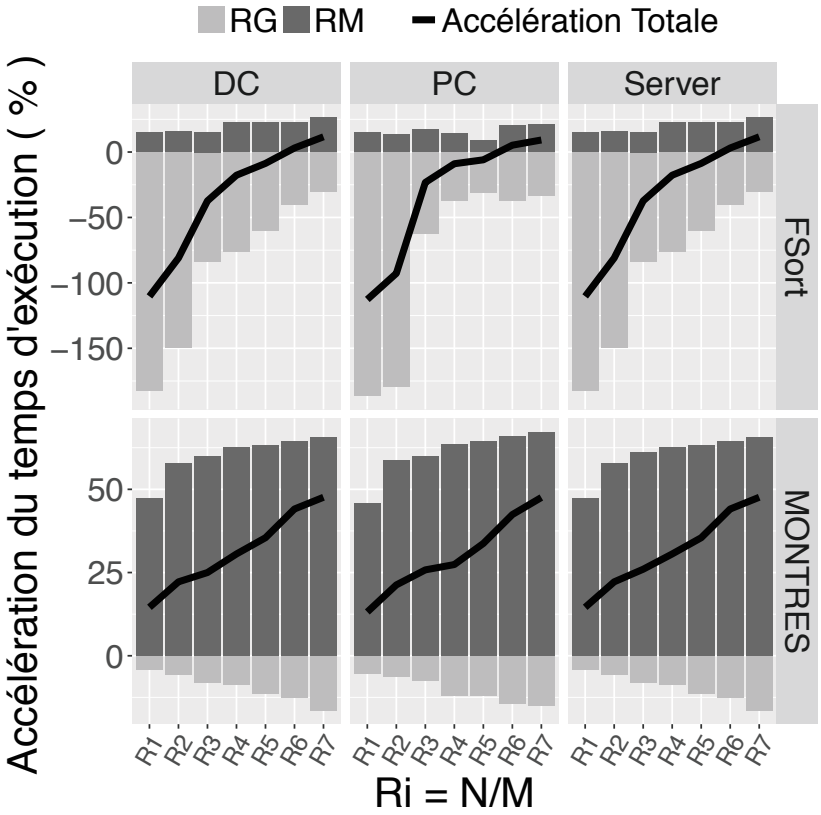

FIGURE 25 : Accélération du temps d'exécution du tri par fusion avec MONTRES et FSort, résultats obtenus avec les ensembles de données (C) sur les trois SSD de type DC, PC et server.

de lecture supplémentaires nécessaires pour réaliser le mécanisme de fusion à la volée.

Nous détaillons ci-après les résultats obtenus à partir du tableau 9 et des Figures 25, et 26 et 27.

Résultat 16. *MONTRES réduit jusqu'à* 54% *le temps d'exécution du tri par fusion traditionnel lorsque* $\frac{N}{M} = 8192$.

Résultat 17. *Nous observons que les performances obtenues avec l'ensemble de données* ***(C)*** *dans la Figure 25 sont plus elvées que celles obtenues avec les autres ensembles de données* ***(A)***, ***(B)***, ***(D)*** *et* ***(E)*** *présentées respectivement dans les Figures 23, 24, 26 et 27.*

Résultat 18. *MONTRES ralentit respectivement de* 9.5%, 8.5% *et* 6.7% *la phase de génération des runs sur les ensembles de données* ***(C)***, ***(D)*** *et* ***(E)*** *lorsque* $\frac{N}{M} = 8162$. *En contrepartie, MONTRES accélére la phase de fusion des runs de* 60%, 59% *et* 43% *respectivement sur les ensembles de données* ***(C)***, ***(D)*** *et* ***(E)***.

Résultat 19. *Le tableau 9 montre que la quantité des valeurs minimales écrites directement dans le fichier final trié* $P_W + \Gamma$ *atteint entre* 43% *et* 46% *du fichier en entrée.*

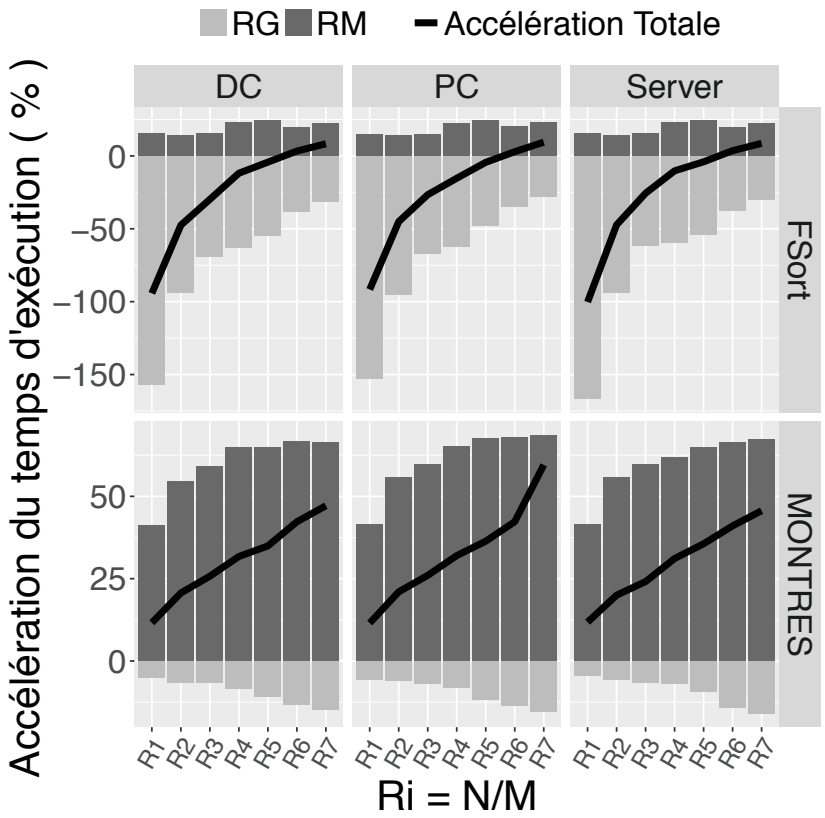

FIGURE 26 : Accélération du temps d'exécution du tri par fusion avec MONTRES et FSort, résultats obtenus avec les ensembles de données (D) sur les trois SSD de type DC, PC et server.

Discussion

Les performances atteintes par MONTRES sur les données partiellement triées sont supérieures à celles atteintes avec des données aléatoires et redondantes (voir les résultats 4, 12, 16, 17 et 19). Ces performances sont dues au fait que les valeurs minimales, respectivement maximales, dans les ensembles de données partiellement triées sont regroupées dans les mêmes blocs. Ainsi, le mécanisme de fusion récupère des quantités de valeurs minimales qui atteignent 60% sur l'ensemble de données **(C)**. Par conséquent, le temps de fusion des *runs* est réduit jusqu'à 54% (voir le résultat 16).

Le mécanisme de fusion à la volée produit jusqu'à 60% du fichier final trié. Ce mécanisme alourdit le temps d'exécution de cette phase (voir le résultat 19).

6.6.3 *Analyse des résultats obtenus en désactivant le cache*

La Figure 28 présente l'accélération obtenue par MONTRES en désactivant le *page cache*. Nous remarquons que l'accélération de MONTRES atteint son mi-

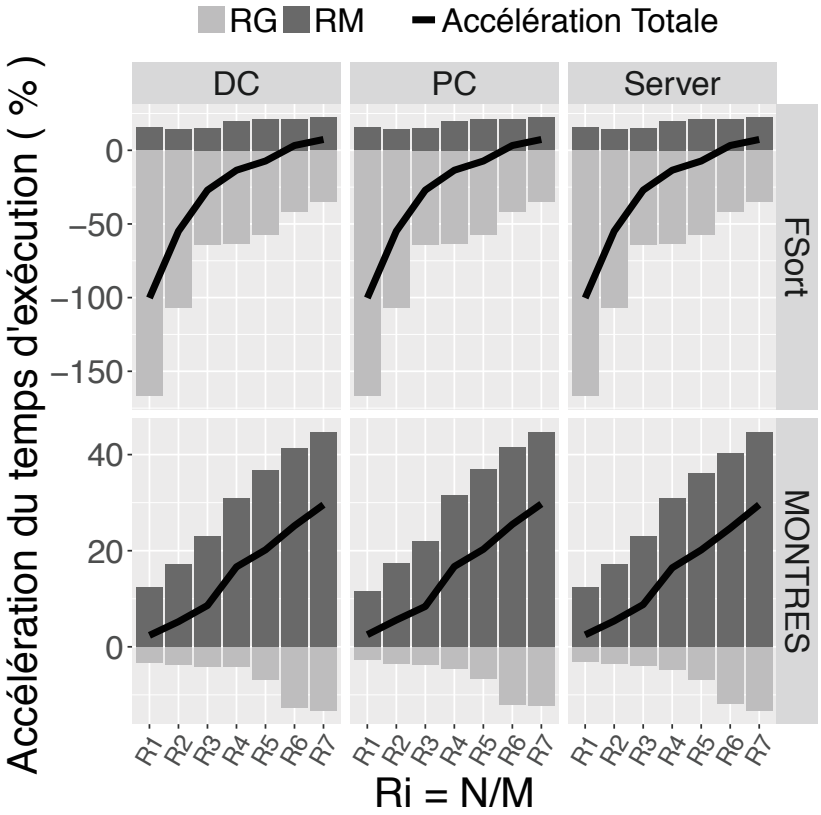

FIGURE 27 : Accélération du temps d'exécution du tri par fusion avec MONTRES et FSort, résultats obtenus avec les ensembles de données (E) sur les trois SSD de type DC, PC et server.

Tableau 9 : Moyenne des coûts d'E/S mesurés pour MONTRES avec les ensembles de données **(C)**, **(D)** et **(E)**

R_i	R_1	R_2	R_3	R_4	R_5	R_6	R_7
$P_W + \Gamma(\%N)$	45.7	45.6	45.4	45.3	44.2	44	43.8
$P_R(\%N)$	11.0	11.1	11.2	11.2	11.8	12.1	12.3

nimum sur l'ensemble de donnés **(A)** avec $\frac{N}{M} = 8$. Le temps d'exécution de MONTRES est 1.13% plus élevé que celui de l'algorithme de tri par fusion traditionnel.

Nous observons que les accélérations réalisées par MONTRES sur les ensembles de données **(A)**, **(B)**, **(C)**, **(D)** et **(E)** avec $\frac{N}{M} = 8192$ ne dépassent pas 21%, alors qu'avec le **page cache** activé l'accélération atteint 50% en moyenne pour les ensembles de données partiéllement triés.

MONTRES utilise le *page cache* pour réduire le nombre d'accès en mémoire secondaire durant la fusion à la volée et aussi durant la fusion des *runs*. En

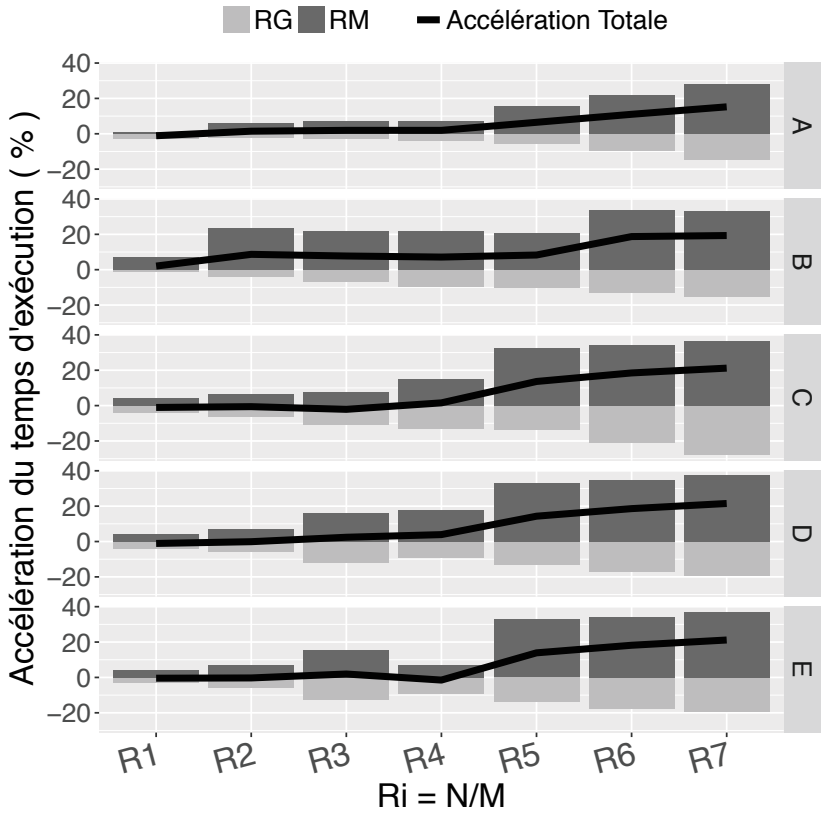

FIGURE 28 : Accélération du temps d'exécution du tri par fusion avec MONTRES et FSort, résultats obtenus avec les ensembles de données **(A), (B), (C), (D)** et **(E)** sur le SSD de type server et en désactivant le cache.

désactivant le *page cache*, le coût en opération d'E/S augmente réduisant ainsi l'accélération apportée par l'algorithme MONTRES.

6.6.4 *Analyse des optimisations de MONTRES*

Le tableau 10 montre les accélérations apportées par les trois mécanismes d'optimisation de MONTRES (mécanisme d'expansion des *runs*, mécanisme de fusion à la volée et mécanisme de fusion des runs). Àpartir des mesures présentées dans le tableau 10, nous constatons les résultats suivants :

Résultat 20. *Le mécanisme d'expansion des runs contribue à hauteur de* 3.22% *dans l'accélération de l'algorithme de tri par fusion sur l'ensemble de données* ***(C)****. Sa contribution sur les ensembles de données* ***(A)*** *et* ***(B)*** *est négligeable.*

Résultat 21. *Le mécanisme de fusion des runs contribue à l'accélération de l'algorithme de fusion à la volée lorsque* $\frac{N}{M} > M-1$*, soit* $\frac{N}{M} = 2048,\ 8192$*. Le mécanisme contribue à hauteur de* 63.12%, 49.16%, 59.20% *sur les ensembles* ***(A)***, ***(B)*** *et* ***(C)****, avec* $\frac{N}{M} = 8192$*.*

Résultat 22. *Le mécanisme de fusion à la volée est à l'origine des accélérations obtenues sur les ensembles* ***(A)***, ***(B)*** *et* ***(C)*** *lorsque* $\frac{N}{M} < M-1$*, soit* $\frac{N}{M} = 8, 32, 64, 128, 512$*.*

D'après l'hypothèse 4 sur les données partiellement triées, les valeurs minimales sont groupées dans un sous-ensemble de blocs et les valeurs maximales dans un autre sous ensemble. Cette propriété permet au mécanisme d'expansion des *runs* de détecter les blocs contenant des valeurs maximales et d'augmenter la taille des *runs*. Ceci explique l'accélération obtenue par le mécanisme d'expansion des *runs* avec l'ensemble de données **(C)** (voir le résultat 20).

Néanmoins, lorsque $\frac{N}{M} > M-1$, le nombre de blocs disponibles en mémoire principales $M-1$ n'est pas suffisant pour allouer $\frac{N}{M}$ tampons de lecture. Dans ce cas, le mécanisme de fusion réalise l'opération de fusion en une seule passe, alors que le tri par fusion traditionnel nécessite plusieurs passes ($\lceil \log_{M-2} \frac{N}{M} \rceil > 1$). Ceci réduit le coût en E/S par rapport au tri par fusion traditionnel. Ceci explique l'accélération apportée par le mécanisme lorsque $\frac{N}{M} = 2048, 8192$ (voir le résultat 21).

Lorsque $\frac{N}{M} < M-1$, MONTRES dispose en mémoire principale d'espace pour allouer $\frac{N}{M}$ tampons de lecture. La fusion des *runs* est réalisée en une seule passe ($\lceil \log_{M-2} \frac{N}{M} \rceil = 1$) par l'algorithme de tri par fusion. Le coût en E/S de l'opération de fusion réalisée par MONTRES est alors similaire au coût en E/S obtenu avec le tri par fusion traditionnel. Ceci explique pourquoi le mécanisme de fusion des *runs* inclut dans MONTRES apporte une accélération négligeable lorsque $\frac{N}{M} = 8, 32, 64, 512$.

Le mécanisme d'expansion des *runs* est efficace seulement sur les données triées et partiellement triées. Ainsi, le mécanisme de fusion à la volée est le seul à apporter une accélération (voir le résultat 22) lorsque $\frac{N}{M} < M-1$, soit $\frac{N}{M} = 8, 32, 64, 128, 512$.

Tableau 10 : Mesures des accélérations obtenues séparément avec les optimisations de MONTRES sur les jeux de données **(A)**, **(B)** et **(C)**

Ensemble de données	Optimisations	R_1	R_2	R_3	R_4	R_5	R_6	R_7
(A)	Expansion des *runs*	0%	0%	0%	0%	0.8%	1.6%	1.45%
	Fusion à la volée	98.9%	98.5%	100%	99.2%	98.8%	37%	32%
	Fusion des *runs*	0%	0%	0%	0%	0%	62.12%	63.12%
(B)	Expansion des *runs*	0%	0%	0%	0%	0%	0.01%	0.02%
	fusion à la volée	100%	100%	100%	100%	100%	49.63%	54.65%
	Fusion des *runs*	0%	0%	0%	0%	0%	45.81%	49.16%
(C)	Expansion des runs	1.36%	1.78%	2.44%	2%	3.22%	2.8%	3.13%
	Fusion à la volée	97.9%	96.1%	98.1%	98.3%	98.9%	49.94%	38.07%
	Fusion des *runs*	1.20%	1.41%	1.74%	2.17%	2.30%	36.87%	59.20%

6.7 CONCLUSION

Dans le chapitre 5, nous avons réalisé une évaluation des coûts d'E/S pour les algorithmes d'état de l'art optimisé pour SSD. Cette évaluation a motivé la conception et la mise en œuvre de MONTRES. Cet algorithme est conçu pour trier différentes échelles de volumes de données et différents types de données (aléatoires, partiellement triées et redondantes) sur des périphériques SSD.

Cet algorithme a été validé avec des analyses de coût des E/S dans la section 6.3 et avec une série d'expérimentations dans la section 6.5. Les résultats des expérimentations prouvent que l'algorithme MONTRES réduit le temps d'exécution du tri en mémoire de 30% en moyenne.

Chapitre 7

LYNX : UN MÉCANISME DE PRÉ-CHARGEMENT DE DONNÉES SUR UN SSD

7.1 INTRODUCTION

Les mécanismes de pré-chargement de données sont conçus pour réduire les coûts des lectures séquentielles sur les périphériques de stockage HDD. Aujourd'hui, avec les nouveaux périphériques de stockage SSD, les opérations de lectures aléatoires sont aussi performantes que celles des lectures séquentielles. Il est donc nécessaire d'étudier de nouveaux mécanismes de pré-chargement de données.

Le rôle d'un mécanisme de pré-chargement de données est de prédire les prochains accès aux données et de les anticiper. Néanmoins, prédire des accès complètement aléatoires est techniquement impossible, nous établissons alors l'Hypothèse 6 sur les motifs d'accès aux données.

Hypothèse 6. ***Motifs d'accès*** *: nous supposons que les accès aux données suivent des motifs qui se répètent régulièrement. Ces motifs d'accès sont connus dans la littérature sous le nom de localité algorithmique [37].*

Contrairement aux accès aléatoires, prédire des accès respectant une localité algorithmique est possible. Une solution envisageable consiste à utiliser un mécanisme d'apprentissage automatique qui permet d'évaluer les motifs d'accès précédents pour prédire les accès à venir.

Dans Lynx, nous proposons la mise œuvre d'un mécanisme d'apprentissage automatique basé sur une chaine de Markov.

Lynx a été intégré au sein du noyau Linux pour compléter l'activité du mécanisme de pré-chargement *read-ahead* limité aux lectures séquentielles. Cette solution a été validée avec une série d'expérimentations réalisées sur des algorithmes de traitement de données utilisant le benchmark TPC-H. Les résultats montrent que Lynx divise par deux en moyenne les temps de traitement des données.

Dans les sections qui suivent, nous décrivons la conception de Lynx, sa mise en œuvre, les expérimentations réalisées dans le cadre de sa validation. Enfin nous discutons des résultats et concluons le chapitre.

7.2 MODÈLE D'APPRENTISSAGE ET DE PRÉDICTION

Dans cette section, nous décrivons comment nous appliquons l'apprentissage automatique basé sur les chaînes de Markov pour résoudre le problème de prédiction des accès algorithmiques aux données. Nous entamons la section avec quelques définitions. Nous décrivons ensuite le modèle proposé.

7.2.1 *Définitions*

Lynx utilise l'apprentissage automatique (voir la définition 23) afin de résoudre le problème de prédiction des accès algorithmiques.

Définition 23. *L'apprentissage automatique [52] concerne la conception, l'analyse, le développement et l'implémentation de méthodes permettant à un programme informatique d'évoluer en fonction de l'analyse des résultats*

L'apprentissage automatique utilisé se base sur une chaîne de Markov (voir la définition 24). Cette dernière nous permet de prédire avec une probabilité l'accès à l'instant $t+1$ en fonction de l'instant t.

Définition 24. *Une chaîne de Markov [11] est une suite de variables aléatoires* $(X_t, t \in \mathbb{N})$ *qui permet de modéliser l'évolution dynamique d'un système aléatoire :* X_t *représente l'état du système à l'instant* t*. La propriété fondamentale des chaînes de Markov, dite propriété de Markov, est que son évolution future* X_{t+1} *ne dépend du passé qu'au travers de sa valeur actuelle* X_t*. Les applications des chaînes de Markov sont très nombreuses (réseaux, génétique des populations, mathématiques financières, . . .).*

Dans notre modèle, nous représentons la chaîne de Markov sous forme d'un automate appelé automate de Markov (voir la définition 25).

Définition 25. *Une chaîne de Markov est représentée avec un automate d'état fini [11], ou chaque état correspond à un des états* X_t *de la chaine de Markov.*

7.2.2 *Modéliser un motif d'accès*

Soit un fichier de N blocs en mémoire secondaire notés $(B_1, ..., B_N)$. Nous modélisons dans cette section les motifs d'accès sur ce fichier en utilisant une chaine de Markov.

Les états X_t, X_{t+1} de la chaine de Markov représentent deux opérations de lecture successives sur des blocs B_i, B_j aux instants t et $t+1$.

Une transition de l'état X_t (lecture sur le bloc B_i) vers un état X_{t+1} (lecture sur le bloc B_j) est noté $T_{i,j}$. La probabilité d'une transition de l'état X_t (lecture sur le bloc B_i) vers l'état X_{t+1} (lecture sur le bloc B_j) est notée $P_{i,j}$. Dans une chaine de Markov, la probabilité $P_{i,j}$ est égale au rapport du nombre de transitions $T_{i,j}$ avec le nombre totale de transitions qui démarrent de l'état X_t correspondant à une lecture sur le bloc B_i.

Chaque transition $T_{i,j}$ est représentée dans l'automate à l'aide d'un arc allant de l'état X_t (lecture sur le bloc B_i) vers l'état X_{t+1} (lecture sur le bloc B_j). Cet arc est pondéré avec la probabilité de transition $P_{i,j}$.

Pour prédire le prochain accès à partir du bloc B_i, nous sélectionnons à partir de l'automate de Markov l'arc sortant de l'état i ayant la probabilité la plus élevée.

Exemple 12. *L'exemple traite des requêtes de lecture lancées par une succession de 3 opérations de recherche dicothomique sur un fichier contenant $N = 8$ blocs et chaque bloc contient 4 valeurs (voir le fichier dans la Figure 29).*

La première opération de recherche dichotomique, est lancée pour rechercher la valeur 10 dans le fichier F. Celle-ci, opère des lectures sur les blocs B_4, B_2 et B_1 pour enfin trouver la valeur 10. La deuxième opération de recherche est menée sur la valeur 11. Elle effectue des lectures sur les blocs B_4, B_2 et B_1 pour enfin trouver la valeur 11.

Dans ce exemple, les opérations de lecture réalisées par l'algorithme font passer la chaîne de Markov par les 3 états suivants :

- *X_t correspond à la lecture du bloc B_4*
- *X_{t+1} correspond à la lecture du bloc B_2*
- *X_{t+2} correspond à la lecture du bloc B_1*

Il résulte alors deux transitions $T_{4,2}$ et $T_{2,1}$ ayant les probabilités $P_{4,2} = 1$ et $P_{2,1} = 1$. Avec ces transitions et ces probabilités, nous obtenons l'automate dans la Figure 30.

Nous résumons les notations utilisées par ce modèle dans le tableau 11.

7.3 MISE EN ŒUVRE DU MODÈLE D'APPRENTISSAGE ET DE PRÉDICTION DANS LYNX

Nous décrivons dans cette section les phases d'apprentissage et de prédiction mises en œuvre dans Lynx. Deux mécanismes de contrôle intégrés dans Lynx sont également présentés.

B0	00	02	06	08
B1	09	10	11	11
B2	12	12	17	17
B3	18	25	25	25
B4	26	26	39	41
B5	44	45	45	54
B6	58	63	69	71
B7	77	77	85	89

FIGURE 29 : Fichier contenant 8 blocs de 4 valeurs

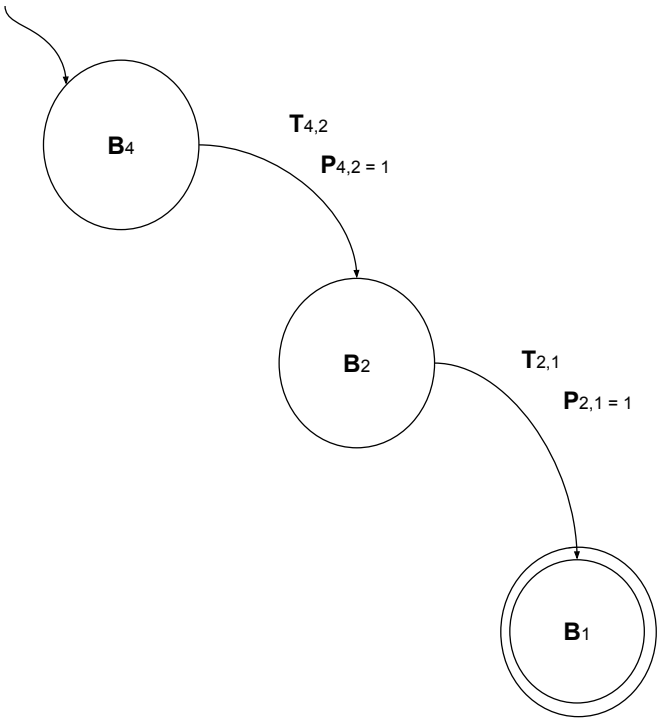

FIGURE 30 : Exemple de chaîne de Markov

7.3.1 *Phase d'apprentissage*

L'objectif de la phase d'apprentissage est d'identifier des séquences de lecture qui se produisent fréquemment sur un fichier en mémoire secondaire. Pour ce faire, Lynx réalise les deux tâches suivantes :

1. Récolter des lectures sur les blocs B_i du fichier et les modéliser sous forme de transitions $T_{i,j}$.

Tableau 11 : Notations utilisées dans le modèle d'apprentissage et de prédiction

Notation	Définition
X_t	Etat de la chaine de Markov à l'instant t. Il correspond à une lecture sur le bloc B_i
$T_{i,j}$	Transition du bloc B_i vers le bloc B_j
$P_{i,j}$	Probabilité d'une transition du bloc B_i vers le bloc B_j

2. Pondérer chaque transition $T_{i,j}$ avec une probabilité $P_{i,j}$.

Les transitions récoltées sont utilisées pour construire et maintenir à jour l'automate de Markov.

7.3.2 *Phase de prédiction*

La prédiction se base sur l'automate de Markov pour identifier le prochain bloc à pré-charger. Pour ce faire, Lynx détecte dans l'automate de Markov la transition $T_{i,j}$ ayant la probabilité la plus élevée. Le bloc B_j est considéré comme le prochain bloc à pré-charger.

Le mécanisme de prédiction est capable de prédire et de pré-charger une séquence de S blocs à partir d'un bloc B_i. Pour ce faire, le mécanisme transite sur S états de l'automate à partir de l'état X_t correspondant aux blocs B_i en détectant à chaque fois les transitions ayant la probabilité la plus élevée.

Exemple 13. *Après deux opérations de recherche dichotomique, nous obtenons l'automate de Markov illustré par la Figure 30.*

Une nouvelle opération de recherche dichotomique est lancée pour trouver la valeur 12 *dans le fichier. Cette opération engendre une lecture sur le bloc* B_4 *du fichier. En se basant sur l'automate de Markov illustré par la Figure 30, le mécanisme de prédiction identifie la transition* $T_{4,2}$ *à partir de l'état* B_4 *ainsi que la transaction* $T_{2,1}$ *à partir de l'état* B_2*. Ces deux transitions sont uniques à partir des états* B_4 *et* B_2*. Le mécanisme de prédiction identifie par défaut ces transactions. Une fois* $T_{4,2}$ *et* $T_{2,1}$ *identifiées, Lynx lance une opération de lecture asynchrone sur les blocs* B_2 *et* B_1 *afin de les pré-charger et de les préparer en mémoire pour une utilisation ultérieure par l'opération de recherche dichotomique.*

7.3.3 *Contrôle de la précision pour la prédiction*

La prédiction est jugée correcte lorsque les blocs prédis par le mécanisme de prédiction, puis pré-chargés par Lynx sont effectivement demandés par le pro-

gramme utilisateur. Autrement dit, lorsque Lynx prédit et pré-charge des blocs de données qui ne sont pas demandés par le programme utilisateur, la prédiction est jugée incorrecte.

Les conséquences d'une prédiction incorrecte peuvent être néfastes pour les performances des E/S. Nous citons ci-dessous quelques-unes de ces conséquences :

- Pollution de la bande passante d'E/S avec des lectures de données inutiles pour le programme.
- Pollution de la mémoire principale avec des blocs de données non utilisées.

Pour réduire l'impact des prédictions incorrectes, nous évaluons en continue le nombre de Hit et de Miss sur le fichier. Lorsque le nombre de Hit dépasse un seuil configurable par l'utilisateur, Lynx considère la prédiction comme étant incorrecte et réinitialise l'automate de Markov.

7.3.4 *Contrôle du mécanisme par le programme utilisateur*

Le concept d'apprentissage et de prédiction proposé dans Lynx ne peut être appliqué à toute charge de travail d'E/S. Lorsque les opérations de lecture sont complètement aléatoires, Lynx peut obtenir un nombre de prédictions incorrects élevés. Pour remédier à ce problème, Lynx peut être activé ou désactivé par le développeur en fonction de la charge d'E/S.

7.4 INTÉGRATION DE LYNX DANS LE NOYAU LINUX

Nous décrivons ci-dessous la mise en œuvre des différents mécanismes intégrés dans Lynx.

7.4.1 *Mécanisme d'apprentissage automatique*

Le mécanisme d'apprentissage est mis en œuvre au niveau fichier. A chaque fichier est alors associé une structure de données contenant les différents états de l'automate, des transitions entre les différents états ainsi que le compteur d'occurrences pour chacune des transitions.

Pour un fichier F contenant N blocs de données, l'automate représente alors N état et N^2 transitions possibles. Afin de limiter l'empreinte mémoire, nous limitons le nombre de transitions à 20 dans notre étude. Nous créons alors une structure de données portant le nom `transition_table`, contenant un tableau de taille N. Chaque case du tableau représente un état de l'automate (voir la Figure 31) et pointe vers une liste de transitions possibles.

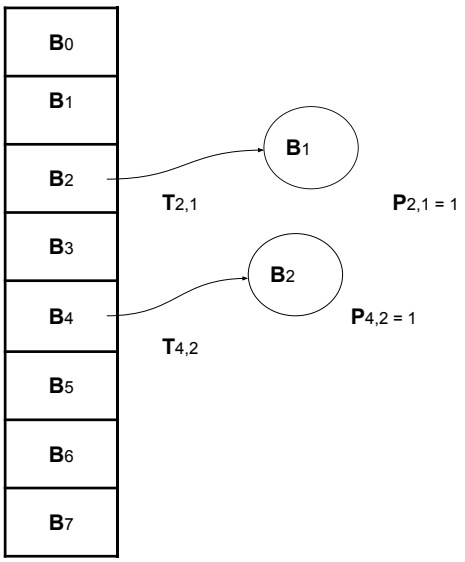

FIGURE 31 : Exemple d'un tableau de transition

7.4.2 *Mécanisme de prédiction*

Lorsqu'une demande de lecture du bloc B_i est interceptée par Lynx, le mécanisme de prédiction recherche la transition $T_{i,j}$ ayant la probabilité la plus élevée dans la liste de transitions associée à l'état X_t (lecture sur le bloc B_i) de la chaine de Markov. La transition $T_{i,j}$ est alors élue comme candidate à l'opération de pré-chargement. Le bloc B_j est alors pré-chargé par Lynx avec une opération de lecture asynchrone.

7.5 EXPÉRIMENTATIONS ET DISCUSSION DES RÉSULTATS

Nous décrivons dans cette section les expérimentations réalisées dans le but de valider Lynx. Nous présentons les expérimentations réalisées, puis les résultats obtenus et enfin nous discutons et analysons ces résultats.

7.5.1 *Méthodologie expérimentale*

Nous évaluons Lynx face au mécanisme de pré-chargement $\texttt{Read}-\texttt{ahead}$, implanté dans le noyau Linux. Nous présentons ici les métriques mesurées, les données utilisées et enfin l'environnement matériel exploité.

7.5.1.1 *Métriques mesurées*

Dans le but d'évaluer les performances de Lynx par rapport à *read-ahead* nous avons mesuré (1) le temps d'exécution, (2) le nombre de Miss et de Hit, et enfin (3) le temps système définit ci-dessous.

1. **Temps système** : cette métrique permet de mesurer le temps CPU nécéssaire à l'exécution des opérations systèmes. Nous évaluons avec cette métrique le gain en temps d'exécution au niveau du noyau.
2. **Temps d'exécution** : Nous évaluons avec cette métrique le gain en temps d'exécution au niveau du programme utilisateur de Lynx.
3. **Nombre de** Miss **/** Hit : cette métrique permet de mesurer le gain en opérations de lecture.

7.5.1.2 *Charge de travail évalué*

Nous avons utilisé dans cette expérimentation le benchmark TPC-H [23]. Celui-ci offre une base de données décisionnelle accompagnée de 22 requêtes. Nous avons créé une base de données TPC-H de 10Go en utilisant un SGBD industriel développé par la société Kode Software.

L'expérimentation s'est déroulée sur deux phases : une phase d'initialisation et une phase de mesure.

Durant la première phase d'initialisation, nous avons lancé les 22 requêtes TPC-H pour que Lynx puisse construire les automates : c'est la phase d'apprentissage. Aucune métrique n'est mesurée durant cette phase.

Durant la deuxième phase, on relance les 22 requêtes TPC-H en activant la mesures des métriques.

7.5.1.3 *Environnement de l'expérimentation*

Nous avons réalisé l'expérimentation sur une machine virtuelle Linux qui utilise la version 3.2 du noyau. La machine virtuelle est équipée de 4Go de mémoire principale et 64Go de mémoire secondaire. La machine virtuelle est sauvegardée sur un périphérique de stockage SSD de type Toshiba THNSNF ayant une capacité de 256Go. La bande passante du SSD est de 100K IOPS pour les lectures aléatoires et 500 Mo/sec pour les lectures séquentielles. Afin d'annuler les effets de cache dans la machine hôte, nous avons désactivé le *cache* du noyau.

7.5.2 *Discussion des résultats*

Dans cette section, nous présentons les résultats obtenus.

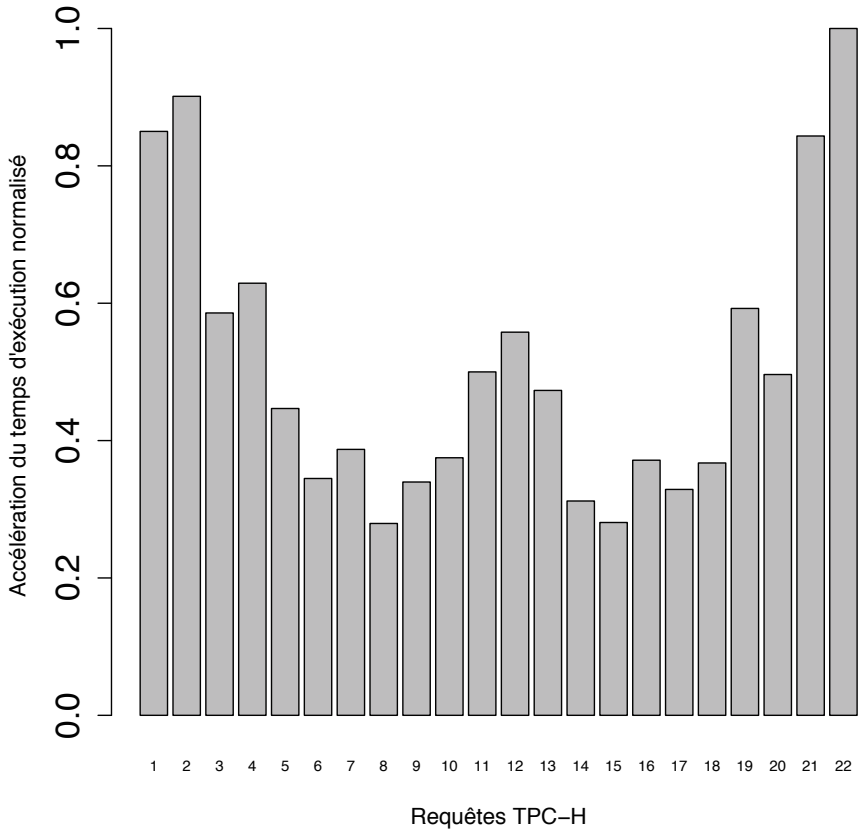

FIGURE 32 : Temps d'exécution des requêtes TPC-H

7.5.2.1 *Temps d'exécution des requêtes TPC-H*

La Figure 32 présente les temps d'exécution obtenus par les requêtes TPC-H avec les deux mécanismes : *read-ahead* et Lynx. Les temps d'exécution de Lynx sont normalisés par rapport à ceux obtenus avec le *read-ahead*.

Les résultats présentés montrent que Lynx réduit les temps d'exécution de toutes les requêtes TPC-H mis à part la requête 22. En effet, cette dernière utilise des fichiers précédemment utilisés par les autres requêtes. Par conséquent l'automate de Markov créé sur ces fichiers ne correspond pas aux séquences de lectures lancées par la requête 22.

Lynx réduit le temps d'exécution des requêtes par 50% en moyenne et jusqu'à 70% pour la requête 8.

7.5.2.2 *Miss*

La Figure 33 présente le nombre de Miss pour chacune des requêtes TPC-H obtenus avec *read-ahead* et Lynx. Les nombres de Miss obtenus avec Lynx sont normalisés par rapport à ceux du *read-ahead*.

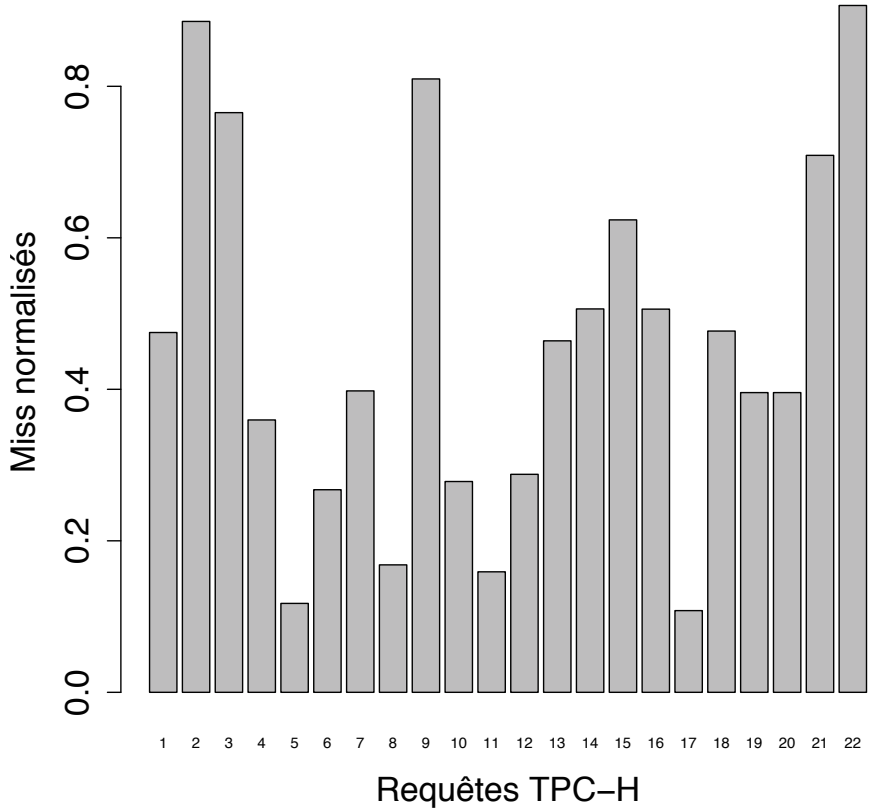

FIGURE 33 : Gain en Miss

Les résultats présentés dans cette figure montrent que Lynx réduit de 54% en moyenne le nombre de Miss (de 80203 Miss jusqu'à 29094). Ce gain montre la correction des prédictions et des pré-chargement faites par Lynx.

7.5.2.3 *Temps système des requêtes TPC-H*

La Figure 34 présente les temps systèmes pour chacune des requêtes TPC-H obtenus avec *read-ahead* et Lynx. Les temps mesurés avec Lynx sont normalisés avec ceux du *read-ahead* Nous constatons à partir de la figure, que le temps système est corrélé à la réduction du nombre de Miss. Ce qui explique la réduction, dans la même proportion, du temps système des requêtes TPC-H.

7.6 CONCLUSION

Lynx est un mécanisme de pré-chargement de données qui utilise qui exploite la localité algorithmique des accès aux données.

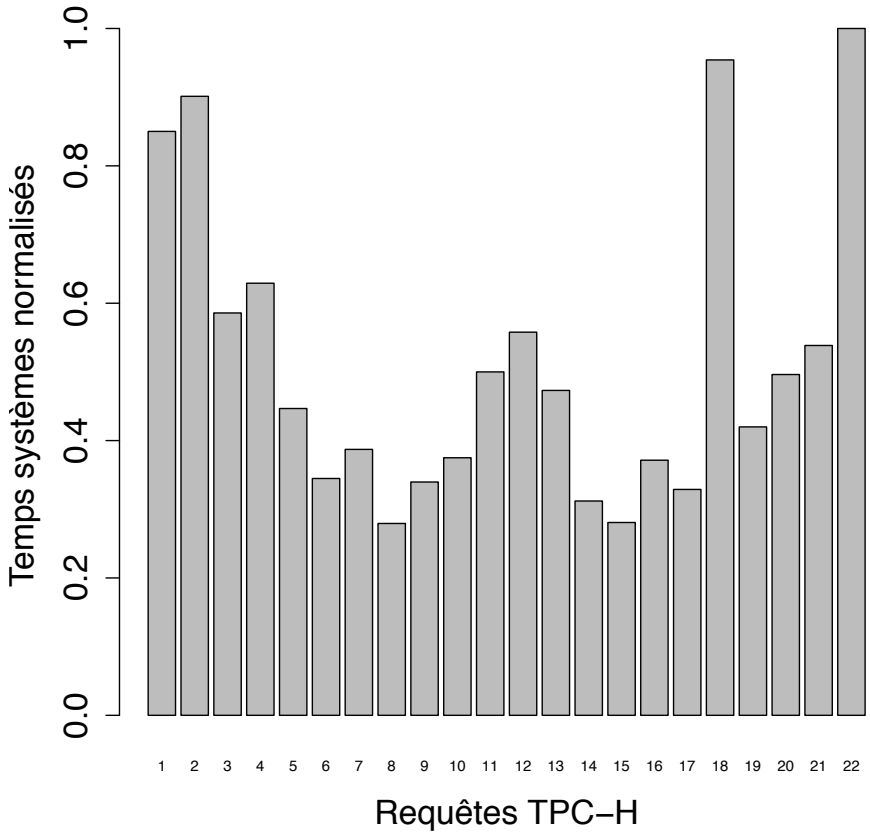

Figure 34 : Temps systèmes mesurés sur les requêtes TPC-H

La conception de Lynx se base sur le fait qu'un algorithme de traitement de données peut réaliser les mêmes séquences de lecture sur un fichier. Ces séquences sont identifiées par Lynx en utilisant une chaine de Markov.

Lynx a fait l'objet d'une validation expérimentale en utilisant le Benchmarck TPC-H. Les résultats des expérimentations ont montrés que Lynx accélère de 50% en moyenne les performances des requêtes TPC-H par rapport au mécanisme de *read-ahead*.

Troisième partie

CONCLUSION

Chapitre 8

CONCLUSION

Cette thèse concerne l'optimisation des performances de SGBD pour les périphériques de stockage SSD. Dans ce chapitre, nous rappelons les différentes contributions apportées dans cette thèse. Nous étudions ensuite quelques perspectives de ces contributions dans de nouveaux contextes logiciels et matérielles.

8.1 RÉSUMÉ DES CONTRIBUTIONS

Nous avons apporté trois contributions dans le cadre de cette thèse. Les deux premières contributions sont consacrées à l'étude et à l'optimisation des algorithmes de tri en mémoire secondaire. La troisième contribution est consacrée au mécanisme de pré-chargement de données.

8.1.1 *Étude des coûts d'E/S des algorithmes de tri en mémoire secondaire*

Le tri en mémoire secondaire est une opération fréquemment sollicitée par les logiciels de traitement de données. Elle implique d'intense opérations de comparaison, mais aussi des opérations d'accès aux données en mémoire principale et en mémoire secondaire.

Dans le cadre de cette étude, nous avons proposé un modèle de coûts en opérations de lecture et d'écriture pour un ensemble d'algorithmes de tri en mémoire secondaire conçus pour les périphériques de stockage SSD. Cette étude prend en compte la taille des données en entrée, la taille de l'espace mémoire alloué et la distribution des données.

8.1.2 *Optimisation du tri en mémoire secondaire*

Le modèle de coûts en E/S des algorithmes de tri en mémoire secondaire pour SSD nous a permis de définir des objectifs d'optimisation pour ce type d'algo-

rithme. Ces objectifs nous ont mené à la conception de MONTRES, un nouvel algorithme de tri en mémoire secondaire.

MONTRES considère la distribution des données dans le fichier en entrée et exploite les lectures aléatoires des périphériques SSD pour diminuer le nombre d'itérations nécessaires pour obtenir le fichier final trié. Il résulte alors un gain en termes de coûts en lectures, écritures et comparaisons également.

MONTRES a fait l'objet d'une validation expérimentale. Pour cela, nous avons utilisé trois modèles différents de SSD et différents jeux de données issus du benchmark TPC-H. Nous avons obtenu un gain en temps d'exécution de 30% en moyenne. Ce gain peut atteindre 60% avec des données partiellement triées.

8.1.3 *Mécanismes de pré-chargement de données*

Les techniques de pré-chargement de données sont conçues pour exploiter les performances des opérations séquentielles sur les périphériques de stockage HDD. Elles exécutent des opérations de lectures asynchrones sur le périphérique de stockage afin de préparer les données en mémoire principale. Le gain de ces techniques de pré-chargement est limité aux lectures séquentielles.

Dans le cadre de notre thèse, nous avons conçu une nouvelle technique de pré-chargement, nommée Lynx, qui exploite les performances des lectures aléatoires sur SSD pour anticiper des lectures ayant une localité algorithmique. Pour ce faire, Lynx se base sur deux mécanismes : un mécanisme d'apprentissage automatique des lectures utilisant le principe d'une chaine de Markov et un second mécanisme qui se charge de prédire les lectures futures à partir des résultats de l'apprentissage.

Nous avons réalisé une validation expérimentale de Lynx sur des opérations de traitement de données. Nous obtenons en moyenne un gain en performance de 50%.

8.2 PERSPECTIVES

Les contributions de cette thèse supposent un SGBD relationnel qui s'exécute sur un seul serveur avec une capacité de mémoire principale limitée. Nous avons principalement exploité la distribution des données en entrée (MONTRES). Nous avons également exploité les performances des périphériques de stockage SSD (MONTRES et Lynx).

Les contributions abordées dans cette thèse peuvent s'appliquer dans d'autres contextes et/ou domaines applicatifs. Dans les sections qui suivent, nous discutons comment ces contributions peuvent s'appliquer dans un contexte de systèmes répartis puis dans un contexte de hiérarchies mémoires hétérogènes (les

noœuds du système réparti différent en terme de capacité et performance mémoire).

8.2.1 *Dans un système réparti*

Les travaux de cette thèse traitent des problématiques de volumes de données qui dépassent de plusieurs ordres de grandeur la capacité de mémoire principale. Cette problématique ne se limite pas uniquement aux SGBD relationnels. Elle existe également dans d'autres domaines applicatifs. C'est notamment le cas du BigData [38] où le volume de données peut impliquer la répartition des données et/ou du traitement [65]. Enfin, les systèmes répartis, par leur taille, sont fréquemment hétérogènes (hétérogénéité des nœuds d'un système réparti en termes de capacité et de performance mémoire) [59].

Dans les sections suivantes, nous discutons comment les contributions apportées dans cette thèse peuvent ouvrir de nouvelles pistes de recherche dans ce contexte.

8.2.1.1 *Modèle de coût des E/S*

Lorsque les données sont distribuées sur le réseau, le coût en opérations de communications peut alourdir les performances d'un algorithme de traitement de données. De plus, les coûts des opérations de lectures et d'écritures en mémoire secondaire peuvent varier en fonction des nœuds du système réparti.

Dans ce cas, les questions qui peuvent se poser sont les suivantes : comment évaluer les coûts en opérations de communications d'un algorithme de traitement de données ? Comment considérer les coûts des accès en lectures et écritures hétérogènes entre les différents nœuds du système réparti ?

8.2.1.2 *MONTRES*

Dans un système réparti, les nœuds sont équipés de périphérique de stockage hétérogènes pouvant varier en termes de bande passante d'E/S. Il arrive alors que les opérations d'E/S sur un nœud soient plus lentes que sur les autres nœuds. Par conséquent, selon la répartition des données, paralléliser MONTRES sur plusieurs nœuds du système réparti implique des performances de lectures de données différentes. Dans ce cas, il est nécessaire d'étudier comment paralléliser MONTRES de façon à réduire le coût global en opérations de lectures.

De plus paralléliser MONTRES sur plusieurs nœuds implique également la mise en œuvre de leur synchronisation par des communications. Il est donc également important de paralléliser MONTRES de façon à réduire ces coûts en communication.

En général, la parallélisation d'un algorithme est étudiée en connaissant les ressources disponibles (nombre de processeur, puissance de calcul, capacité mémoire). Les applications ciblées dans ce travail, peuvent être déployées dans des architectures de type *Cloud*. Certaines architectures cloud autorisent des allocations de ressources éphémères à des coûts moindres mais sans garantie forte sur leurs disponibilités [8]. Cette incertitude sur la disponibilité des ressources complique d'autant plus la parallélisation de MONTRES.

Enfin, dans les systèmes répartis de grande taille, la probabilité d'une panne d'une machine ne peut être écartée. La version actuelle de MONTRES ne suppose aucune panne lors des calculs et des lectures de données. L'étude de la tolérance aux pannes de MONTRES est donc nécessaire dans un contexte réparti.

8.2.2 *Lynx*

Lorsque les données sont distribuées sur plusieurs nœuds, l'accès aux données se fait sur des nœuds distants du système réparti. Ceci impliquent aussi des opérations de communications entre les nœuds du système.

Dans ce cas, comment améliorer le modèle d'apprentissage mis en œuvre dans Lynx pour prédire les lectures sur des nœuds distants ?

Il est également important d'étudier le placement des données pré-chargées. Faut-il considérer un placement local au niveau du cache du nœud contenant ces données ? Ou plutôt un placement distant en diffusant en mode asynchrone les données pré-chargées sur les nœuds susceptibles de demander ces données ? Cette deuxième solution est plus agressive en opérations de communications, elle peut s'avérer efficace lorsque les nœuds susceptibles d'utiliser les données sont identifiables.

8.2.3 *Dans une hiérarchie mémoire hétérogène*

Les travaux de cette thèse supposent des performances de lectures aléatoires symétriques à celles des lectures séquentielles et des performances d'écritures asymétriques à celles des lectures. Ces travaux supposent par ailleurs une granularité au niveau du bloc pour les opérations de lecture et d'écriture.

Nous assistons aujourd'hui à l'émergence de nouvelles mémoires NVM (*Non-Volatile Memory*) [54]. Celles-ci offrent de nouveaux modèles de performance et une granularité fine pour les opérations d'E/S. Par exemple, les mémoires de types PCM (*Phase Change Memory*) offrent une granularité au niveau de l'octet et des performances proches de celles des DRAM.

Ces nouvelles mémoires peuvent s'intégrer horizontalement pour étendre les capacités de mémoires principales ou verticalement pour servir de cache à la

mémoire secondaire [53]. L'émergence de ces nouvelles mémoires ouvre d'autres perspectives pour nos travaux, perspectives que nous décrivons ci-après.

8.2.3.1 *Modèle de coût des E/S*

Lorsque la hiérarchie mémoire est hétérogène, le modèle de performance peut varier entre les différentes mémoires. Dans ce cas, il est nécessaire de modéliser les coûts des E/S en considérant les différents modèles de performances.

De plus, les opérations d'E/S peuvent avoir différentes granularités. Il est également important d'étudier comment considérer cette différence de granularité dans l'évaluation des coûts en E/S. Notre modèle de coût devra donc être adapté à ce nouveau contexte matériel.

8.2.3.2 *MONTRES*

Pour appliquer MONTRES dans ce contexte, nous nous posons la question du placement des données à trier entre les différentes mémoires afin d'utiliser au mieux les caractéristiques de celles-ci.

Plus généralement, une hiérarchie mémoire hétérogène peut comporter plus de deux niveaux de mémoire principale. Par exemple, les mémoires PCM peuvent étendre significativement les capacités de mémoire principale [25, 47]. Ces nouvelles hiérarchies mémoires peuvent alors invalider les hypothèses que nous avons émises pour MONTRES concernant le rapport entre volume de données N et la mémoire principale allouée M. Une perspective intéressante consiste à étudier l'adéquation de MONTRES à ces nouvelles hiérarchies mémoires.

8.2.4 *Lynx*

Dans une hiérarchie mémoire hétérogène, nous pouvons avoir plusieurs niveaux de cache avec différents modèles de performance. Il est donc nécessaire d'étudier le placement des données pré-chargées sur les différents niveaux de cache. Ce placement doit considérer principalement le modèle de performance des différents niveaux de cache ainsi que l'utilisation des données pré-chargées.

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UBO
Université de Bretagne Occidentale

Titre : Optimisation des performances des logiciels de traitement de donnés sur les périphériques de stockage SSD

Mots clés : base de données, tri, pré-chargement de données, hiérarchie mémoire, mémoire flash, SSD...

Résumé : Nous assistons aujourd'hui à une croissance vertigineuse des volumes de données. Cela exerce une pression sur les infrastructures de stockage et les logiciels de traitement de données comme les Systèmes de Gestion de Base de Données (SGBD).

De nouvelles technologies ont vu le jour et permettent de réduire la pression exercée par les grandes masses de données.

Nous nous intéressons particulièrement aux nouvelles technologies de mémoires secondaires comme les supports de stockage SSD (Solid State Drive) à base de mémoire Flash.

Les supports de stockage SSD offrent des performances jusqu'à 10 fois plus élevées que les supports de stockage magnétiques. Cependant, ces nouveaux supports de stockage offrent un nouveau modèle de performance. Cela implique l'optimisation des coûts d'E/S pour les algorithmes de traitement et de gestion des données.

Dans cette thèse, nous proposons un modèle des coûts d'E/S sur SSD pour les algorithmes de traitement de données. Ce modèle considère proncipalement le volume des données, l'espace mémoire alloué et la distribution des données.

Nous proposons également un nouvel algorithme de tri en mémoire secondaire : MONTRES. Ce dernier est optimisé pour réduire le coût des E/S lorsque le volume de données à trier fait plusieurs fois la taille de la mémoire principale. Nous proposons enfin un mécanisme de pré-chargement de données : Lynx. Ce dernier utilise un mécanisme d'apprentissage pour prédire et anticiper les prochaines lectures en mémoire secondaire.

Title : Performance optimization for Data processing software on SSD storage devices

Keywords : Database, sorting, prefetching, memory hierarchy, flash memory, SSD ...

Abstract : The growing volume of data poses a real challenge to data processing software like DBMS (DataBase Management Systems) and data storage infrastructure. New technologies have emerged in order to face the data volume challenges. We considered in this thesis the emerging new external memories like flash memory-based storage devices named SSD (Solid State Drive).

SSD storage devices offer a performance gain compared to the traditional magnetic devices. However, SSD devices offer a new performance model that involves IO cost optimization for data processing and management algorithms.

We proposed in this thesis an IO cost model to evaluate the data processing algorithms. This model considers mainly the SSD IO performance and the data distribution.

We also proposed a new external sorting algorithm: MONTRES. This algorithm includes optimizations to reduce the IO cost when the volume of data is greater than the allocated memory space by an order of magnitude. We proposed finally a data prefetching mechanism: Lynx. This one makes use of a machine learning technique to predict and to anticipate future access to the external memory.

[1] Intel dc s3710. http://www.intel.fr/content/www/fr/fr/solid-state-drives/solid-state-drives-dc-s3710-series.html. Accessed : 2017-03-07.

[2] Ssd 850 pro. http://www.samsung.com/fr/business/business-products/ssd/850-pro-series/MZ-7KE256BW. Accessed : 2017-03-07.

[3] 850 evo sata iii 2.5inch ssd 4tb. https://www.samsung.com/uk/memory-storage/850-evo-sata-3-2-5-inch-ssd/MZ-75E4T0BEU/.

[4] Ssd 845 dc pro - mz-7wd800. http://www.samsung.com/fr/business/business-products/ssd/845-dc-pro/MZ-7WD800EW. Accessed : 2017-03-07.

[5] Daniel Abadi, Rakesh Agrawal, Anastasia Ailamaki, Magdalena Balazinska, Philip A Bernstein, Michael J Carey, Surajit Chaudhuri, Jeffrey Dean, AnHai Doan, Michael J Franklin, et al. The beckman report on database research. *Communications of the ACM*, 59(2) :92–99, 2016.

[6] Alok Aggarwal, Jeffrey Vitter, et al. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9) :1116–1127, 1988.

[7] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D Davis, Mark S Manasse, and Rina Panigrahy. Design tradeoffs for ssd performance. In *USENIX Annual Technical Conference*, pages 57–70, 2008.

[8] Maryam Amiri and Leyli Mohammad-Khanli. Survey on prediction models of applications for resources provisioning in cloud. *Journal of Network and Computer Applications*, 82 :93–113, 2017.

[9] Panayiotis Andreou, Orestis Spanos, Demetrios Zeinalipour-Yazti, George Samaras, and Panos K. Chrysanthis. Fsort : External sorting on flash-based sensor devices. In *Proceedings of the Sixth International Workshop on Data Management for Sensor Networks*, DMSN '09, pages 10 :1–10 :6, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-777-6. doi : 10.1145/1594187.1594201.

[10] Michael D Atkinson, J-R Sack, Nicola Santoro, and Thomas Strothotte. Min-max heaps and generalized priority queues. *Communications of the ACM*, 29 (10) :996–1000, 1986.

[11] Roy Billinton and Ronald N Allan. Discrete markov chains. In *Reliability Evaluation of Engineering Systems*, pages 206–224. Springer, 1983.

[12] Jalil Boukhobza. Flashing in the Cloud : Shedding some Light on NAND Flash Memory Storage Systems. In *Data Intensive Storage Services for Cloud Environments*, pages 241–266. IGI Global, April 2013.

[13] Jalil Boukhobza and Pierre Olivier. *Flash Memory Integration : Performance and Energy Issues*. Elsevier, 2017.

[14] Jalil Boukhobza and Pierre Olivier. *Flash Memory Integration : Performance and Energy Issues*. Elsevier, 2017.

[15] Daniel P Bovet and Marco Cesati. *Understanding the Linux Kernel : from I/O ports to process management*. " O'Reilly Media, Inc.", 2005.

[16] Paolo Cappelletti and Alberto Modelli. Flash memory reliability. In *Flash Memories*, pages 399–441. Springer, 1999.

[17] David F Carr. How google works. *Baseline Magazine*, 6(6), 2006.

[18] Feng Chen, David A Koufaty, and Xiaodong Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *ACM SIGMETRICS Performance Evaluation Review*, volume 37, pages 181–192. ACM, 2009.

[19] Feng Chen, Rubao Lee, and Xiaodong Zhang. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 266–277, Feb 2011. doi : 10.1109/HPCA.2011.5749735.

[20] Shimin Chen, Anastassia Ailamaki, Phillip B Gibbons, and Todd C Mowry. Improving hash join performance through prefetching. *ACM Transactions on Database Systems (TODS)*, 32(3) :17, 2007.

[21] Edgar F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6) :377–387, 1970.

[22] Tyler Cossentine and Ramon Lawrence. Efficient external sorting on flash memory embedded devices. *International Journal of Database Management Systems*, 5(1), 2013.

[23] Transaction Processing Performance Council. Tpc-h benchmark specification. *Published at http ://www.tpc.org/tpch/*, 2008.

[24] Jeffrey Dean and Sanjay Ghemawat ËĹ MapReduce. Simplified data processing on large clusters google. *2004*, 2004.

[25] Y. Fang, H. Li, and X. Li. Lifetime enhancement techniques for pcm-based image buffer in multimedia applications. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(6) :1450–1455, June 2014. ISSN 1063-8210. doi : 10.1109/TVLSI.2013.2266668.

[26] WU Fengguang, XI Hongsheng, and XU Chenfeng. On the design of a new linux readahead framework. *ACM SIGOPS Operating Systems Review*, 42(5) : 75–84, 2008.

[27] Hector Garcia-Molina, Jeffrey D Ullman, and Jennifer Widom. *Database system implementation*, volume 654. Prentice Hall Upper Saddle River, NJ :, 2000.

[28] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS operating systems review*, volume 37, pages 29–43. ACM, 2003.

[29] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys (CSUR)*, 25(2) :73–169, 1993.

[30] Goetz Graefe. Implementing sorting in database systems. *ACM Comput. Surv.*, 38(3), September 2006. ISSN 0360-0300. doi : 10.1145/1132960.1132964.

[31] Laura M Grupp, Adrian M Caulfield, Joel Coburn, Steven Swanson, Eitan Yaakobi, Paul H Siegel, and Jack K Wolf. Characterizing flash memory : anomalies, observations, and applications. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 24–33. IEEE, 2009.

[32] Jan L Harrington. *Relational database design and implementation : clearly explained*. Morgan Kaufmann, 2009.

[33] Guy Harrison. *Next Generation Databases : NoSQLand Big Data*. Apress, 2015.

[34] Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Hao Luo, and Shuping Zhang. Performance impact and interplay of ssd parallelism through advanced commands, allocation strategy and data granularity. In *Proceedings of the international conference on Supercomputing*, pages 96–107. ACM, 2011.

[35] MICRON INC. *Small-Block vs. Large-Block NAND Flash Devices*. Rapport TN-29-07, 2005.

[36] Intel. Intel and micron develop the world's fastest nand flash memory with 5x faster performance. *PubliÃ' dans https ://www.intel.com/pressroom/archive/releases/2008/20080201corp.htm, visitÃ' le 01-02-2018*, 2008.

[37] Bruce Jacob, Spencer Ng, and David Wang. *Memory Systems : Cache, DRAM, Disk*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007. ISBN 0123797519, 9780123797513.

[38] Xiaolong Jin, Benjamin W Wah, Xueqi Cheng, and Yuanzhuo Wang. Significance and challenges of big data research. *Big Data Research*, 2(2) :59–64, 2015.

[39] Yongsoo Joo, Junhee Ryu, Sangsoo Park, and Kang G Shin. Fast : Quick application launch on solid-state drives. In *FAST*, pages 259–272, 2011.

[40] Vamsee Kasavajhala. Solid state drive vs. hard disk drive price and performance study. *Proc. Dell Tech. White Paper*, pages 8–9, 2011.

[41] Donald Ervin Knuth. *The art of computer programming : sorting and searching*, volume 3. Pearson Education, 1998.

[42] Michel Koskas. Method of computation of a short path in valued graphs, July 29 2008. US Patent 7,406,047.

[43] Michel Koskas. Method for organizing a data base, May 12 2009. US Patent 7,533,078.

[44] Arezki Laga, Jalil Boukhobza, Michel Koskas, and Frank Singhoff. Lynx : a learning linux prefetching mechanism for ssd performance model. In *Non-Volatile Memory Systems and Applications Symposium (NVMSA), 2016 5th*, pages 1–6. IEEE, 2016.

[45] Arezki Laga, Jalil Boukhobza, Frank Singhoff, and Michel Koskas. Montres : Merge on-the-run external sorting algorithm for large data volumes on ssd based storage systems. *IEEE Transactions on Computers*, 2017.

[46] Per-Ake Larson. External sorting : Run formation revisited. *IEEE Transactions on Knowledge and Data Engineering*, 15(4) :961–972, 2003.

[47] Benjamin C Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 2–13. ACM, 2009.

[48] J. Lee, H. Roh, and S. Park. External mergesort for flash-based solid state drives. *IEEE Transactions on Computers*, 65(5) :1518–1527, May 2016. ISSN 0018-9340. doi : 10.1109/TC.2015.2451631.

[49] Sang-Won Lee, Bongki Moon, Chanik Park, Jae-Myung Kim, and Sang-Woo Kim. A case for flash memory ssd in enterprise database applications. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1075–1086. ACM, 2008.

[50] Boduo Li, Edward Mazur, Yanlei Diao, Andrew McGregor, and Prashant Shenoy. A platform for scalable one-pass analytics using mapreduce. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 985–996. ACM, 2011.

[51] Yang Liu, Zhen He, Yi-Ping Phoebe Chen, and Thi Nguyen. External sorting on flash memory via natural page run generation. *The Computer Journal*, 54 (11) :1882–1990, 2011.

[52] Stephen Marsland. *Machine learning : an algorithmic perspective*. CRC press, 2015.

[53] Jeffrey C Mogul, Eduardo Argollo, Mehul Shah, and Paolo Faraboschi. Operating system support for nvm hybrid main memory. 2009.

[54] Onur Mutlu. *More than Moore Technologies for Next Generation Computer Design*, chapter Main Memory Scaling : Challenges and Solution Directions, pages 127–153. Springer New York, New York, NY, 2015. ISBN 978-1-4939-2163-8. doi : 10.1007/978-1-4939-2163-8_6.

[55] Hyoungmin Park and Kyuseok Shim. Fast : Flash-aware external sorting for mobile database systems. *Journal of Systems and Software*, 82(8) :1298–1312, 2009.

[56] Detlev Richter. Fundamentals of reliability for flash memories. In *Flash Memories*, pages 149–166. Springer, 2014.

[57] David Reinsel John Gantz John Rydning. *Data Age 2025 : The Evolution of Data to Life-Critical*. IDC white paper, 2017.

[58] Michael Stonebraker, Samuel Madden, Daniel J Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era :(it's time for a complete rewrite). In *Proceedings of the 33rd international conference on Very large data bases*, pages 1150–1160. VLDB Endowment, 2007.

[59] Andrew S Tanenbaum and Maarten Van Steen. *Distributed systems : principles and paradigms*. Prentice-Hall, 2007.

[60] Andrew S Tanenbaum and Albert S Woodhull. *Operating systems : design and implementation*, volume 2. Prentice-Hall Englewood Cliffs, NJ, 1987.

[61] Ashish Thusoo, Zheng Shao, Suresh Anthony, Dhruba Borthakur, Namit Jain, Joydeep Sen Sarma, Raghotham Murthy, and Hao Liu. Data warehousing and analytics infrastructure at facebook. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 1013–1020. ACM, 2010.

[62] Ahsen J Uppal, Ron C Chiang, and H Howie Huang. Flashy prefetching for high-performance flash drives. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1–12. IEEE, 2012.

[63] A Inkeri Verkamo. Performance comparison of distributive and mergesort as external sorting algorithms. *Journal of Systems and Software*, 10(3) :187–200, 1989.

[64] Chin-Hsien Wu and Kuo-Yi Huang. Data sorting in flash memory. *ACM Transactions on Storage (TOS)*, 11(2) :7, 2015.

[65] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets : A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[66] J Leon Zhao, Shaokun Fan, and Daning Hu. Business challenges and research directions of management analytics in the big data era. *Journal of Management Analytics*, 1(3) :169–174, 2014.

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UBO
Université de Bretagne Occidentale

Titre : Optimisation des performances des logiciels de traitement de donnés sur les périphériques de stockage SSD

Mots clés : base de données, tri, pré-chargement de données, hiérarchie mémoire, mémoire flash, SSD...

Résumé : Nous assistons aujourd'hui à une croissance vertigineuse des volumes de données. Cela exerce une pression sur les infrastructures de stockage et les logiciels de traitement de données comme les Systèmes de Gestion de Base de Données (SGBD).

De nouvelles technologies ont vu le jour et permettent de réduire la pression exercée par les grandes masses de données.

Nous nous intéressons particulièrement aux nouvelles technologies de mémoires secondaires comme les supports de stockage SSD (Solid State Drive) à base de mémoire Flash.

Les supports de stockage SSD offrent des performances jusqu'à 10 fois plus élevées que les supports de stockage magnétiques. Cependant, ces nouveaux supports de stockage offrent un nouveau modèle de performance. Cela implique l'optimisation des coûts d'E/S pour les algorithmes de traitement et de gestion des données.

Dans cette thèse, nous proposons un modèle des coûts d'E/S sur SSD pour les algorithmes de traitement de données. Ce modèle considère proncipalement le volume des données, l'espace mémoire alloué et la distribution des données.

Nous proposons également un nouvel algorithme de tri en mémoire secondaire : MONTRES. Ce dernier est optimisé pour réduire le coût des E/S lorsque le volume de données à trier fait plusieurs fois la taille de la mémoire principale. Nous proposons enfin un mécanisme de pré-chargement de données : Lynx. Ce dernier utilise un mécanisme d'apprentissage pour prédire et anticiper les prochaines lectures en mémoire secondaire.

Title : Performance optimization for Data processing software on SSD storage devices

Keywords : Database, sorting, prefetching, memory hierarchy, flash memory, SSD ...

Abstract : The growing volume of data poses a real challenge to data processing software like DBMS (DataBase Management Systems) and data storage infrastructure. New technologies have emerged in order to face the data volume challenges. We considered in this thesis the emerging new external memories like flash memory-based storage devices named SSD (Solid State Drive).

SSD storage devices offer a performance gain compared to the traditional magnetic devices. However, SSD devices offer a new performance model that involves IO cost optimization for data processing and management algorithms.

We proposed in this thesis an IO cost model to evaluate the data processing algorithms. This model considers mainly the SSD IO performance and the data distribution.

We also proposed a new external sorting algorithm: MONTRES. This algorithm includes optimizations to reduce the IO cost when the volume of data is greater than the allocated memory space by an order of magnitude. We proposed finally a data prefetching mechanism: Lynx. This one makes use of a machine learning technique to predict and to anticipate future access to the external memory.