UNIVERSITÉ DE MONTRÉAL

# STRUCTURES DE DONNÉES HAUTEMENT EXTENSIBLES POUR LE STOCKAGE SUR DISQUE DE SÉRIES TEMPORELLES HÉTÉROGÈNES

LOÏC PRIEUR-DREVON
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
MARS 2017

© Loïc Prieur-Drevon, 2017.

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

STRUCTURES DE DONNÉES HAUTEMENT EXTENSIBLES POUR LE STOCKAGE SUR DISQUE DE SÉRIES TEMPORELLES HÉTÉROGÈNES

présenté par : PRIEUR-DREVON Loïc
en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de :

M. PESANT Gilles, Ph. D., président
M. DAGENAIS Michel, Ph. D., membre et directeur de recherche
M. MERLO Ettore, Ph. D., membre

# DÉDICACE

*À mes grands parents,*
*le temps que nous avons partagé était trop bref...*

## REMERCIEMENTS

Le travail présenté dans ce mémoire n'aurait pu être concrétisé sans l'aide et le soutien de plusieurs personnes et organisations. Je tiens donc à prendre le temps de les remercier.

J'aimerais premièrement remercier Coline Briquet, de la Direction des Relations Internationales à l'École Polytechnique, Palaiseau, France, pour avoir fait prospérer l'entente entre les deux Écoles Polytechniques. Sans elle, ni moi ni mes 14 camarades Polytechniciens n'aurions eu la possibilité de venir étudier à Montréal.

Je tiens ensuite à remercier tout particulièrement Michel Dagenais, mon directeur de recherche, pour m'avoir offert l'opportunité de travailler sur ce projet. Son soutien, ses conseils et ses exigences furent clés à l'avancement de ces recherches.

Ensuite merci à Ericsson et Efficios pour le financement de ces travaux de recherche. En prime de financement, ces deux compagnies ont montré un intérêt prononcé pour les avancées de ce travail. Leur utilisation en production des logiciels a permis d'avoir des retours rapides et en situation réelle sur les solutions implémentées.

Mes remerciements s'étendent à deux anciens du laboratoire, pour leur investissement remarquable. Matthew Khouzam, dont la disponibilité et la ténacité permirent de ne laisser aucune pierre non retournée quant aux pistes d'amélioration du projet. Alexandre Montplaisir, par sa contribution dans le projet lors de sa maitrise et sa vision pour son avenir après, me permit de progresser le plus rapidement possible.

De plus, merci à mes collègues du laboratoire DORSAL pour les moments que nous avons vécus ensemble. Surtout merci à Raphaël de m'avoir supporté et aussi Geneviève, pour leur expérience sur le projet, avoir pris le temps d'écouter et redresser mes hypothèses et solutions, même les plus folles !

Finalement, merci au Jury et autres lecteurs pour l'intérêt que vous portez à ce mémoire et à ce projet.

# RÉSUMÉ

Les systèmes informatiques deviennent de plus en plus complexes, et les développeurs ont plus que jamais besoin de comprendre comment interagissent les nombreux composants de leurs systèmes. De nombreux outils existent pour instrumenter, mesurer et analyser les comportements et la performance des logiciels. Le traçage est une technique qui enregistre de nombreux points associés à des événements du système et l'estampille de temps à laquelle ils se sont produits. L'analyse manuelle des traces produites permet de comprendre différents problèmes, mais elle devient fastidieuse lorsque ces historiques contiennent de très grands nombres de points. Il existe des logiciels pour automatiser ces analyses et fournir des visualisations, mais ces derniers peuvent aussi montrer leurs limites pour se mettre à l'échelle des systèmes les plus étendus.

Dans des travaux précédents, Montplaisir et coll. ont présenté une structure de données sur disque, optimisée pour stocker les résultats des analyses de traces sous forme d'intervalles d'états : $< clé, t_{début}, t_{fin}, valeur >$. La structure, nommée State History Tree (SHT), est un arbre pour lequel chaque noeud est associée à un bloc de disque, chaque noeud dispose donc d'une capacité fixe pour stocker des intervalles et est défini par un intervalle de temps tel que cet intervalle est inclus dans celui du noeud parent et que les intervalles de deux enfants ne se superposent pas. Cette structure était plus efficace que d'autres solutions génériques, mais pouvait dégénérer, dans des cas avec un très grand nombre de clés, pour une trace avec de nombreux fils par exemple, la profondeur de l'arbre était alors proportionnelle au nombre de fils, et de très nombreux noeuds "vides" étaient écrits sur disque, gaspillant de l'espace. De plus, les requêtes pour extraire les informations de la structure étaient souvent le goulot d'étranglement pour l'affichage des données.

Dans ce travail, nous analysons les limites de la base de données actuelle qui la conduisent à dégénérer et nous étudierons les cas d'utilisation des requêtes.

Nous proposons des modifications structurelles permettant d'éliminer les cas de dégénérescence lorsque la trace contient de nombreux attributs, tout en réduisant la taille sur disque de la structure pour tous types de traces. Nous ajoutons aussi des métadonnées aux noeuds de l'arbre pour réduire le nombre de noeuds lus pendant les requêtes. Ceci permet de réduire la durée des requêtes de 50% dans la plupart des cas.

Ensuite, nous cherchons à optimiser le processus d'insertion des intervalles dans les noeuds de l'arbre afin de regrouper les intervalles qui seront demandés dans une même requête pour limiter le nombre de blocs de disque à lire pour répondre. Le nombre d'intervalles pris

en compte dans l'optimisation peut augmenter avec le nombre de clés par exemple, ce qui permet de maintenir un équilibre entre le temps supplémentaire requis pour l'optimisation et les gains constatés sur les requêtes qui deviennent plus flagrants lorsque l'analyse produit de nombreuses clés.

Nous introduirons aussi un nouveau type de requête profitant de ces optimisations et permettant de retourner en une requête un ensemble d'intervalles qui précédemment prenait plusieurs requêtes. De plus cette requête assure que chaque noeud est lu au plus une fois, alors que l'utilisation de plusieurs requêtes impliquait que certains noeuds étaient lus plusieurs fois. Nous montrons que l'utilisation de cette requête dans une des vues principales du logiciel de visualisation augmente considérablement sa réactivité.

Nous profiterons ensuite de ces apprentissages pour faciliter la mise à l'échelle d'une seconde structure de données du logiciel d'analyse de trace, qui stocke des objets nommés "segments", sous la forme de $< t_{début}, t_{fin}, valeur >$. Ces objets étaient précédemment stockés en mémoire et donc le nombre que nous pouvions stocker était limité. Nous utilisons une structure en arbre fortement inspirée du SHT. Nous montrons que la structure sur disque est au pire un ordre de grandeur plus lent que les structures en mémoire à la lecture. De plus, cette structure est particulièrement efficace pour un cas d'usage qui demande à retourner des segments triés. En effet, nous utilisons un algorithme réalisant l'évaluation à la demande et un tri partiel entre les noeuds, qui utilise moins de mémoire que le tri de tous les segments.

# ABSTRACT

Computer systems are becoming more and more complex, and developers need more than ever to be able to understand how different components interact. Many tools have been developed for instrumenting, measuring and analysing the behavior and performance of software. One of these techniques is tracing, which records data-points and a timestamp associated to system events. Trace analysis can help solve a number of problems, but manual analysis becomes a daunting task when these traces contain a large number of points. Automated trace analysis software has been developed for this use case, but they too can face difficulties scaling up to the largest systems.

In previous work, Montplaisir et al. presented a disk-based data structure, optimized for storing the results of trace analysis as state intervals: $< key, t_{start}, t_{end}, value >$. This SHT is a tree for which each node is mapped to a block on disk, such that each node has a fixed capacity to store intervals and is defined by a time range that must be included in its parent's range and must not overlap with its siblings' ranges. This structure was demonstrated to be more efficient that other, generic solutions, but could still degenerate for corner cases with many keys, from traces with many threads for example. The tree's depth would then be proportional to the number of threads and many empty nodes would be written to disk, wasting space. Moreover, queries to extract data from the data structure were often the bottleneck for displaying data.

In this work, we analyse the limitations of the current database which cause it to degenerate and study the different use cases for queries.

We suggest structural changes to the data structure, which eliminate the corner case behavior for traces with many attributes, while reducing the disk usage for all types of traces. We also add meta data to the nodes to reduce the number of nodes searched during queries, speeding them up by 50%.

Then, we look into optimizing the nodes into which intervals are inserted, so that those which will be queried together will be grouped. This helps to reduce the number of disk blocks that must be read to answer the query. The number of intervals and nodes taken into account by the optimization process can increase along with the number of attributes, as they are the main cause of query slowdown. This helps to balance the extra time required for the optimized insertion and the gains provided on the queries.

We also introduce a new type of query to benefit from these optimizations and return all

desired intervals in a single query instead of the many queries previously required. This single query reads each node at most once, while the previous version with many queries would read some nodes several times. We show that using this query for one of the main views in the trace visualization software makes it considerably more reactive.

We benefit from all these lessons learned to increase the scalability of another internal backend, the segment store, used for the following type of objects: $< t_{start}, t_{end}, value >$. These were previously stored in memory, which would strongly limit the maximum capacity. We propose a new tree structure similar to the SHT instead. We show that the disk based structure is, in the worst case, only an order of magnitude slower for reads than the in-memory structures. Moreover, this structure is especially efficient for a typical segment store use case, which is a sorted segment query. Indeed by using partial sorts between nodes, memory usage is dramatically reduced compared to sorting all segments in memory.

# TABLE DES MATIÈRES

DÉDICACE . . . . . iii

REMERCIEMENTS . . . . . iv

RÉSUMÉ . . . . . v

ABSTRACT . . . . . vii

TABLE DES MATIÈRES . . . . . ix

LISTE DES TABLEAUX . . . . . xiii

LISTE DES FIGURES . . . . . xiv

LISTE DES SIGLES ET ABRÉVIATIONS . . . . . xvi

CHAPITRE 1 INTRODUCTION . . . . . 1
- 1.1 Définitions et concepts de base . . . . . 1
  - 1.1.1 Traçage . . . . . 1
  - 1.1.2 Analyse d'état . . . . . 2
  - 1.1.3 Enregistrer sur disque les résultats des analyses . . . . . 2
- 1.2 Fonctionnement du système à états . . . . . 2
  - 1.2.1 Arbre à attributs . . . . . 3
  - 1.2.2 États courants . . . . . 3
  - 1.2.3 Arbre à historique d'états . . . . . 4
- 1.3 Éléments de la problématique . . . . . 4
  - 1.3.1 Stocker les intervalles d'état sur disque de manière efficace . . . . . 4
  - 1.3.2 Extraire les informations du disque pour les visualiser de manière efficace 4
- 1.4 Objectifs de recherche . . . . . 5
- 1.5 Plan du mémoire . . . . . 5

CHAPITRE 2 REVUE DE LITTÉRATURE . . . . . 6
- 2.1 Outils d'analyse système dynamiques . . . . . 6
- 2.2 Visualiseurs de traces . . . . . 9
- 2.3 Structures de données . . . . . 12
- 2.4 Arbres R-Tree . . . . . 19

2.4.1 Partitionnement de noeuds . . . . . 19
2.4.2 Variantes du R-Tree . . . . . 20
2.4.3 R-Tree sur disque . . . . . 21
2.4.4 Algorithmes sur les arbres R . . . . . 22
2.5 Bases de données pour séries temporelles . . . . . 22
2.6 Conclusion de la revue de littérature . . . . . 24

CHAPITRE 3 MÉTHODOLOGIE . . . . . 25
3.1 Visualisation . . . . . 25
3.2 Modélisation . . . . . 25
3.3 Analyse des besoins . . . . . 25
3.4 Profilage . . . . . 27
3.5 Jeu de Traces . . . . . 27
3.6 Mesure de la performance . . . . . 28
3.7 Conclusion de la méthodologie . . . . . 28

CHAPITRE 4 ARTICLE 1 : R-SHT : A STATE HISTORY TREE WITH R-TREE PROPERTIES FOR ANALYSIS AND VISUALIZATION OF HIGHLY PARALLEL SYSTEM TRACES . . . . . 30
4.1 Abstract . . . . . 30
4.2 Introduction . . . . . 31
4.3 Related Work . . . . . 32
4.3.1 Trace visualizers . . . . . 32
4.3.2 Stateful Data Structures . . . . . 33
4.3.3 R-Trees . . . . . 35
4.4 Limitations of the State History Tree . . . . . 37
4.4.1 Structure of the State History Tree (SHT) . . . . . 37
4.4.2 Shortcomings on analysis of large systems . . . . . 39
4.5 The Overlapping State History Tree . . . . . 40
4.5.1 Overlapping SHT structure . . . . . 40
4.5.2 Enhanced State History Tree . . . . . 41
4.5.3 Search algorithms . . . . . 41
4.5.4 Comparison of query bounds with SHT . . . . . 43
4.5.5 Query scalability limitations . . . . . 46
4.6 R-SHT model, structure and algorithms . . . . . 47
4.6.1 R-Tree qualities for the SHT . . . . . 47
4.6.2 Build Algorithm . . . . . 49

4.6.3 Clustering Algorithm . . . . . 49
4.6.4 2D Queries . . . . . 50
4.7 Performance . . . . . 51
4.7.1 Single queries . . . . . 51
4.7.2 2D Query . . . . . 53
4.7.3 Full queries . . . . . 53
4.7.4 Impact of the buffer size . . . . . 54
4.8 Results . . . . . 55
4.8.1 Test Environment . . . . . 55
4.8.2 Case Study: Control Flow View . . . . . 56
4.8.3 Storage usage . . . . . 61
4.8.4 Tree Depth . . . . . 61
4.8.5 Single Query . . . . . 62
4.8.6 Full Query . . . . . 64
4.8.7 2D Query . . . . . 64
4.8.8 Build Times . . . . . 65
4.9 Conclusion . . . . . 65

CHAPITRE 5 RÉSULTATS COMPLÉMENTAIRES . . . . . 67
5.1 Optimisation des analyses . . . . . 67
5.2 Déduplication des chaînes de caractères . . . . . 69
5.3 Retirer les intervalles "nuls" . . . . . 70
5.4 L'arbre à segments . . . . . 71
5.4.1 Arbre à Segments par dessus des bases de données existantes . . . . . 72
5.4.2 Arbre à Segments sur le principe de l'arbre à état . . . . . 72
5.4.3 Amélioration de la performance en itération de l'arbre à segments . . 76
5.5 Conclusion résultats complémentaires . . . . . 79

CHAPITRE 6 DISCUSSION GÉNÉRALE . . . . . 81
6.1 Format de retour pour les structures de données sur disque . . . . . 81
6.2 Choix de l'algorithme de regroupement d'intervalles . . . . . 82
6.3 Rétrocompatibilité des fichiers d'historique d'états . . . . . 82

CHAPITRE 7 CONCLUSION . . . . . 83
7.1 Synthèse des travaux . . . . . 83
7.2 Limitations de la solution proposée . . . . . 84
7.3 Améliorations futures . . . . . 84

RÉFÉRENCES . . . . . 85

# LISTE DES TABLEAUX


Tableau 3.1 Caractéristiques des traces de test . . . . . 28

Table 4.1 Scalability trace set specifications . . . . . 56

Table 4.2 Comparison of the History Trees for the `many-threads` trace . . . . 57

Table 4.3 Gains over SHT for displaying data in a Control Flow View. (mean ± standard deviation) on 10 executions. . . . . 59

Tableau 5.1 Gains suite à la déduplication des chaînes de caractères. Trace `ManyThreads` du jeu de traces de test : git.eclipse.org/gitroot/tracecompass/tracecompass-test-traces.git . . . . . 70

Tableau 5.2 Gains suite à la suppression des premiers et derniers intervalles nuls. 71

# LISTE DES FIGURES


Figure 1.1 Composants d'un système à états. Ce schéma illustre le changement d'état du processus 42, qui était `USERMODE` depuis l'estampille de temps $403ns$. L'événement d'ordonnancement change l'état de ce processus, clôt ainsi l'intervalle d'état <83, 403ns, 494ns, `USERMODE`> et l'archive dans le SHT et provoque aussi le changement de l'état courant vers `WAIT_CPU` avec la nouvelle estampille de temps $494ns$. . . . 3
Figure 1.2 Construction de l'arbre à historique d'états [1] . . . . . 4
Figure 2.1 Graphes de flamme montrant les cycles processeur utilisés par MySQL [2] 8
Figure 2.2 Exemple de données recueillies dans une trace noyau LTTng, formaté pour la lecture par babeltrace. . . . . 9
Figure 2.3 Architecture de traçage de Tracealyzer . . . . . 13
Figure 2.4 Arbre de Recherche binaire . . . . . 13
Figure 2.5 Arbre à intervalles augmentés des Segments $\{I_1, I_2, I_3, I_4\}$ et modèle des segments . . . . . 14
Figure 2.6 Arbre des Segments $\{I_1, I_2, I_3, I_4\}$ et modèle des segments . . . . . 15
Figure 2.7 Arbre B de degré 2 . . . . . 15
Figure 2.8 Arbre B MultiVersion . . . . . 17
Figure 2.9 Courbe de Hilbert pour n = 1 à 5 [3] . . . . . 21
Figure 2.10 Processus de fusion de l'arbre LSM [4] . . . . . 24
Figure 3.1 Visualisation de la structure d'un SHT . . . . . 26
Figure 3.2 Utilisation des tests de performance automatisés d'Eclipse pour mesurer un bloc [5]. . . . . 29
Figure 3.3 Résultats produits par les tests de performance automatisés d'Eclipse 29
Figure 4.1 Representation of the `slog2` data structure [6] . . . . . 34
Figure 4.2 Build steps of the State History Tree using an incremental process [7] 38
Figure 4.3 Schematization of a State History Tree with many attributes . . . . 39
Figure 4.4 Comparison of the relations between sibling nodes of SHT (left), oSHT (middle) and eSHT (right) . . . . . 40
(a) Representation of consecutive nodes in SHT . . . . . 40
(b) Representation of overlapping nodes in oSHT . . . . . 40
(c) Representation of enhanced nodes in eSHT . . . . . 40
Figure 4.5 Comparison of tree search between SHT (left) and eSHT (right) . . 42
(a) Representation of a branch search as used by SHT . . . . . 42

(b) Representation of a sub-tree search as used by eSHT . . . . . 42
Figure 4.6 Schematization of the intervals in a Tree . . . . . 43
Figure 4.7 Comparison of SHT and eSHT query times for traces with many attributes . . . . . 46
Figure 4.8 Key range histogram for 10k thread trace using eSHT . . . . . 47
Figure 4.9 Default oSHT Split . . . . . 48
Figure 4.10 R-SHT hybrid structure . . . . . 48
Figure 4.11 Splitting the R-Tree Buffer along the key dimension . . . . . 50
Figure 4.12 Increasing the depth of the R-Tree buffer during construction . . . . 54
Figure 4.13 The Control Flow View, a Time Graph View in Trace Compass . . . 57
Figure 4.14 Comparison of external memory usage for different SHT variants . . 60
Figure 4.15 Comparison of tree depth with different SHT variants . . . . . 61
Figure 4.16 Comparison of single query performance with different SHT variants 62
Figure 4.17 Comparison of full query performance with different SHT variants . 63
Figure 4.18 Comparison of full, single and 2D queries to extract the same data on eSHT and vR-SHT . . . . . 64
Figure 4.19 Comparison of build time with different SHT variants. . . . . 65
Figure 5.1 Comparaison de l'arbre à attributs de l'analyse noyau après et avant optimisation . . . . . 68
(a) L'arbre à attributs de l'analyse noyau avant optimisation . . . . . 68
(b) L'arbre à attributs de l'analyse noyau après optimisation . . . . . 68
Figure 5.2 Comparaison des stocks de segments dans les tests de Trace Compass 73
Figure 5.3 Performance de l'arbre de segments comparé aux implémentations en mémoire . . . . . 74
Figure 5.4 Utilisation mémoire du tri en mémoire comparée au tri de la file de priorité . . . . . 77
Figure 5.5 Temps de l'itération triée en mémoire comparée au tri de la file de priorité . . . . . 78
Figure 5.6 Accélération de l'itération triée par une optimisation unidimensionnelle 80

# LISTE DES SIGLES ET ABRÉVIATIONS

| | |
|---|---|
| ARC | Adaptive Replacement Cache |
| BTF | Best Trace Format |
| CTF | Common Trace Format |
| CUBE | CUBE Uniform Behavioral Encoding |
| eSHT | enhanced State History Tree |
| FIFO | First In First Out |
| GBI | Generalized Bulk-Insertion |
| GDB | GNU Debugger |
| HPC | High-Performance Computing |
| JUL | `java.util.logging` |
| LRU | Least Recently Used |
| LTTng | Linux Trace Toolkit next generation |
| LTTV | Linux Tracing Toolkit Viewer |
| MPE | MPI Parallel Environment |
| MPI | Message Passing Interface |
| OTF | Open Trace Format |
| PID | Process Id |
| PPID | Parent Process Id |
| SHT | State History Tree |
| SIG | Système d'information géographique |
| SST | Synchronized Tree Traversal |
| STLT | Small-Tree-Large-Tree |
| TAU | Tuning and Analysis Utilities |
| TMF | Tracing and Monitoring Framework |
| UCT | Unité Centrale de Traitement |
| ViTE | Visual Trace Explorer |
| ZFS | Zettabyte File System |
| ZIL | ZFS Intent Log |

# CHAPITRE 1 INTRODUCTION

Les systèmes informatiques deviennent de plus en plus complexes, que ce soit un ordinateur avec plusieurs processeurs logiques, voire physiques ou de nombreux serveurs roulant chacun un grand nombre de programmes. Il est essentiel de comprendre leur comportement afin de les mettre en place et de s'assurer que les ressources sont utilisées efficacement.

De nombreuses techniques et logiciels ont été développés pour étudier ces systèmes, en particulier les traceurs permettent d'aider les développeurs à déboguer et optimiser les logiciels. Toutefois, l'interprétation manuelle de ces traces devient une tâche fastidieuse lorsque ces dernières contiennent des millions voire milliards de points décrivant des systèmes très complexes. Pour cela il existe des programmes permettant d'analyser et visualiser ces traces.

Pour ces logiciels aussi, la mise à l'échelle est importante pour pouvoir afficher rapidement à l'écran l'état d'un composant sans avoir à relire toute la trace, surtout lorsque le fichier de trace peut être plusieurs ordres de grandeur plus grands que la mémoire principale d'une machine.

Ce mémoire propose un ensemble de techniques pour stocker efficacement sur disque les données des analyses de traces tout en retournant rapidement le résultat d'une interrogation de la base de données.

## 1.1 Définitions et concepts de base

### 1.1.1 Traçage

Le traçage est une technique d'analyse dynamique de programmes permettant d'enregistrer des informations à certains points du code source. Lorsqu'un tel point est traversé, des informations sont envoyées avec une estampille de temps très précise à un logiciel nommé traceur qui interprète et écrit ces informations sur disque.

Le traçage est souvent comparé aux enregistreurs chronologiques automatiques (*loggers*), toutefois le traçage s'intéresse aux événements plus bas niveau et beaucoup plus fréquents – jusqu'à des dizaines de milliers de fois par seconde.

Lors de l'analyse de programmes complexes, il n'est pas rare de produire des traces contenant des millions voire plus d'événements, ce qui rend leur analyse manuelle fastidieuse. Pour pallier à ces difficultés, plusieurs logiciels permettent des analyses automatiques de traces et la visualisation de ces résultats.

### 1.1.2 Analyse d'état

Dans les logiciels ayans recours à de l'analyse d'états, le système et ses composants sont modélisés par des machines à états. Les attributs sont les plus petites unités du modèle pouvant avoir un état indépendant.

Ces analyses s'intéressent aux états de certains attributs du système, à travers le temps. Certaines prennent pour entrée des traces décrivant déjà les états dans le temps. Pour d'autres, il est nécessaire de convertir une trace d'événements ponctuels en états. Dans le dernier cas, les événements décrivent les transitions entre états du système.

Par exemple, en analysant une trace du noyau Linux avec des événements d'ordonnancement, on pourra déduire l'état des processus au cours du temps.

Les intervalles d'états associent à un attribut une valeur d'état entre deux estampilles de temps :

$$< clé, t_{début}, t_{fin}, valeur > \tag{1.1}$$

où la *clé* est un entier unique associé à l'attribut, $t_{début}$ est le temps de début – inclusif – de l'intervalle, $t_{fin}$ est le temps de fin – exclusif – de l'intervalle et *valeur* est la valeur de l'attribut pendant l'intervalle de temps.

### 1.1.3 Enregistrer sur disque les résultats des analyses

Lors de l'analyse de systèmes particulièrement complexes, il n'est plus envisageable de stocker tous les états de ces systèmes en mémoire. Les analyseurs se mettant le mieux à l'échelle sont capables d'enregistrer les intervalles d'états sur disque pour réduire l'empreinte en mémoire principale et éviter de réaliser à nouveau l'analyse lors de la réouverture du logiciel.

## 1.2 Fonctionnement du système à états

Le logiciel d'analyse de traces Trace Compass intègre un système à état qui permet de convertir un fichier de traces avec des événements ponctuels en une base de données contenant les états au cours du temps pour des attributs. Il est composé de plusieurs modules, l'état courant qui stocke les états non encore terminés avant de les écrire sur disque, l'arbre à attributs qui associe des noms / chemins à des clés numériques et un arbre à historique d'états qui stocke les intervalles d'état sur disque.

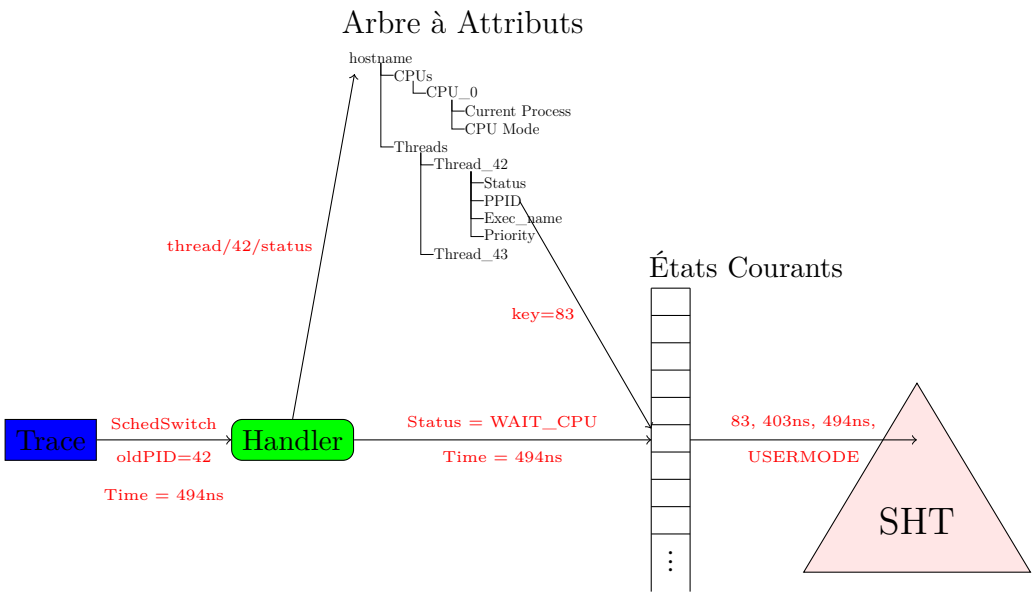



Figure 1.1 Composants d'un système à états. Ce schéma illustre le changement d'état du processus 42, qui était `USERMODE` depuis l'estampille de temps $403ns$. L'événement d'ordonnancement change l'état de ce processus, clôt ainsi l'intervalle d'état <83, 403ns, 494ns, `USERMODE`> et l'archive dans le SHT et provoque aussi le changement de l'état courant vers `WAIT_CPU` avec la nouvelle estampille de temps $494ns$.

### 1.2.1 Arbre à attributs

L'arbre à attributs associe une clé unique au chemin de chaque attribut. Il représente une arborescence en laquelle chaque niveau a un identifiant. Par exemple, le chemin du nom du fil de Process Id (PID) 42 pour une analyse du noyau sera `THREADS/42/EXEC_NAME`. Si nous décomposons le chemin, il y aura une clé pour `THREADS`, une pour `THREADS/42` et une autre pour `THREADS/42/EXEC_NAME`.

L'arbre à attributs permet d'ajouter de nouveaux attributs à la demande au cours de l'analyse de la trace. Les clés d'attributs numériques sont ensuite plus faciles à manipuler pour la recherche d'intervalles et occupent moins de place sur disque que si chaque intervalle stockait le chemin entier. L'arbre à attributs est stocké sur disque pour pouvoir être réutilisé.

### 1.2.2 États courants

Les analyses de traces se font par estampille de temps croissante des événements. Au début de l'analyse, l'état d'un attribut est inconnu, et son temps de début est initialisé au temps de début de la trace. Au cours de l'analyse, l'état courant de chaque attribut est en mémoire, ainsi que l'estampille de temps de l'événement qui correspond au début de l'état. Lorsqu'un

événement modifie d'état d'un attribut, un intervalle avec pour temps de début, le temps de l'état courant, pour temps de fin, le temps de l'événement et pour valeur l'état courant est produit. Ensuite le temps et la valeur de l'événement remplacent ceux de l'état courant

### 1.2.3 Arbre à historique d'états

Les intervalles d'état de la forme $< clé, t_{début}, t_{fin}, état >$ sont stockés dans l'arbre à historique d'état, structure de données sur disque permettant des insertions et requêtes efficaces. Chaque noeud de cet arbre correspond à un bloc de disque, pour des lectures et écritures plus efficaces, ce qui implique une capacité fixe pour chaque noeud.

Chaque noeud est défini par son temps de début et de fin, les noeuds parents peuvent contenir jusqu'à 50 enfants tel que le temps de début de l'enfant $n$ est le temps de fin de l'enfant $n-1$.

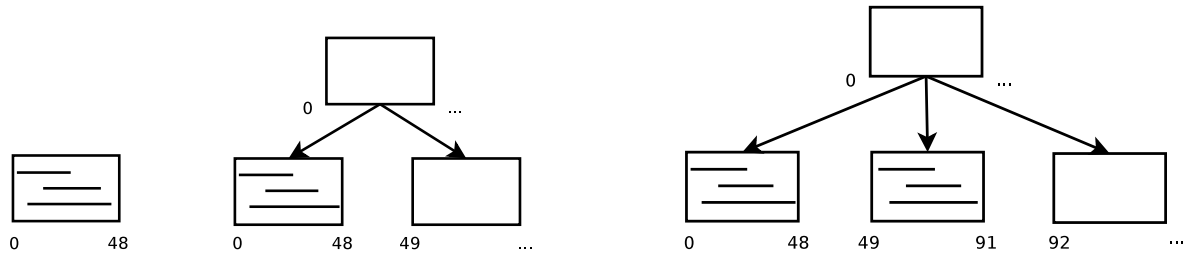



Figure 1.2 Construction de l'arbre à historique d'états [1]

L'arbre est bâti à partir du noeud feuille commençant à la première estampille de la trace. Lorsque ce dernier est plein, on lui ajoute un parent et un frère. Les intervalles sont insérés dans le noeud le plus profond qui débute plus tôt que l'intervalle.

## 1.3 Éléments de la problématique

### 1.3.1 Stocker les intervalles d'état sur disque de manière efficace

Dans des travaux précédents, une structure en arbre permettant de stocker efficacement sur disque les intervalles d'état avait été présentée. Bien que meilleure que d'autres solutions génériques, il est encore possible de l'optimiser et d'éliminer des cas particuliers pour lesquels cette structure dégénérait en taille et en profondeur.

### 1.3.2 Extraire les informations du disque pour les visualiser de manière efficace

Un second enjeu de la visualisation de données en mémoire externe est d'extraire les données suffisamment rapidement pour fournir une visualisation réactive. En effet, l'interface actuelle

de la base de données repose fortement sur les requêtes complètes qui retournent les états de tous les attributs à un instant donné.

## 1.4 Objectifs de recherche

Il est maintenant possible de préciser les objectifs de recherche :

1. Analyser le comportement de la structure de données dans les cas problématiques et proposer des améliorations éliminant le plus d'effets de bord possibles.
2. Analyser comment les vues exploitent les requêtes sur la structure de données et proposer des solutions plus performantes.
3. Proposer d'autres améliorations permettant à TraceCompass d'être plus réactif tout en consommant moins de ressources.

## 1.5 Plan du mémoire

Au chapitre 2 nous ferons l'état de l'art des solutions de traçage, d'analyse et de visualisation de traces ainsi que des structures de données sur disque et structures de données multi-dimensionnelles. Ensuite, le chapitre 3 présentera la méthodologie utilisée pour aboutir aux objectifs de recherche. Le chapitre 4 contient un article de revue présentant les résultats principaux de la recherche. Nous présenterons des résultats complémentaires au chapitre 5 avant d'entamer une discussion générale des résultats au chapitre 6. Finalement, nous conclurons au chapitre 7 en synthétisant les travaux et identifiant de potentielles évolutions futures.

# CHAPITRE 2 REVUE DE LITTÉRATURE

Ce chapitre fait un état de l'art des outils d'analyse dynamiques de système en mettant l'accent sur le traçage. Il adressera ensuite les outils utilisés pour analyser et visualiser les traces logicielles avant de rentrer dans le détail des structures de données pouvant être utilisées pour des recherches rapides. Cet état de l'art couvrira des arbres binaires, jusqu'aux arbres spécialisés dans les données multidimensionnelles sans oublier les méthodes pour les exploiter efficacement en mémoire externe. Les solutions construites dans ce mémoire tireront profit des connaissances présentées dans ce chapitre.

## 2.1 Outils d'analyse système dynamiques

Cette section présente des outils utilisés pour analyser des systèmes et logiciels de manière dynamique, c.-à-d. en l'absence du code source, à partir de données collectées lors de l'exécution.

Le débogueur GNU Debugger (GDB) [8], initialement développé pour GNU, ensuite porté à d'autres systèmes basés sur UNIX, permet de tester et trouver des bogues dans les programmes. Le programme à déboguer est exécuté à travers le débogueur, et est interrompu au franchissement de points d'arrêts, d'exceptions ou de fautes pour pouvoir analyser son état. Le débogueur peut afficher les valeurs des variables et la pile d'exécution et exécuter des programmes pas-à-pas. GDB supporte de nombreuses architectures de processeurs et plusieurs langages de programmation, et peut même déboguer à distance des plateformes embarquées et le noyau. Pour les programmes haut niveau avec code source, le débogueur montre la section de code problématique. Pour les programmes bas niveaux pour lequel le code source n'est pas disponible, GDB peut montrer la ligne désassemblée.

Strace [9] fut développé pour diagnostiquer les interactions entre le noyau et les processus en espace usager sur la plateforme fermée `sunOS` avant d'être porté sur Linux. Son utilisation principale est d'enregistrer les appels systèmes, mais strace peut aussi surveiller les signaux et changements d'état de processus. Le programme à tracer peut être démarré par strace, ou rattaché à une session strace a posteriori. Strace peut révéler les appels systèmes responsables du dysfonctionnement du programme, tels que des tentatives d'ouvrir un fichier inexistant ou une connexion réseau bloquante. Chaque ligne de trace strace contient le numéro de PID appelant, l'appel système, ses arguments entre parenthèses et finalement la valeur de retour de l'appel. Strace dispose aussi d'options pour ajouter des estampilles de temps des appels

en microsecondes ou pour mesurer la durée passée dans l'appel système. Bien que plus facile d'utilisation qu'un débogueur, par exemple, la limitation aux appels systèmes fait de strace un outil spécialisé qui ne peut résoudre tous les problèmes.

Ftrace [10] est un traceur noyau disposant de nombreux modules d'extension pour enregistrer les appels de fonction et autres événements du noyau Linux tels que les événements d'ordonnancement, les interruptions, entrée/sorties et changements de modes de puissance du processeur. Ses outils d'analyse permettent de générer des graphes d'appels, des analyses de pile d'appels, et de mesurer les latences d'interruption et de préemption. Le traçage par ftrace impose d'avoir un noyau compilé avec l'option `CONFIG_FUNCTION_TRACER` activée. Ftrace repose sur les mécanismes de profilage de gcc pour recompiler les fonctions noyau afin qu'elles soient précédées d'appels aux fonctions de traçage. Toutes les interactions avec ftrace se font par le biais de fichiers virtuels, résidant en un système de fichiers nommé `debugfs`, au sein duquel des fichiers stockent les paramètres des modules du traceur ainsi que les traces produites.

Le Linux Trace Toolkit next generation (LTTng) [11] est un traceur libre, développé au laboratoire DORSAL. Il permet de tracer le noyau, les programmes en mode utilisateur, les programmes Python ainsi que les programmes Java avec `java.util.logging` (JUL) ou log4j [1]. Ses avantages sont la modularité et le faible surcoût lorsque le traçage est activé, ce qui permet de minimiser l'impact sur le comportement du système. LTTng permet de définir des points de trace sur mesure, et de ne tracer que certains types de points de trace, pour limiter la taille des traces résultantes, dans le cas des traces noyau par exemple. LTTng est capable d'enregistrer des traces contenant de très nombreux événements grâce à l'utilisation d'un tampon circulaire et un fichier de trace par processeur. Pour les systèmes avec peu d'espace de stockage, LTTng est capable d'enregistrer les événements par le réseau, par TCP vers un daemon qui écrira les traces sur son système de fichier.

Perf [12] fut initialement développé pour accéder aux compteurs de performance sous Linux. En particulier, les compteurs de performance du processeur sont des registres matériels qui comptent les instructions exécutées, les fautes de cache, ou les branches mal prédites. Perf peut extraire les comptes de ces registres à intervalles réguliers – mode profileur – ou à la demande sur annotation du code source ou assembleur. Désormais, perf peut aussi prendre en compte les compteurs de performance logiciels et les points de trace. Les données enregistrées et les analyses fournies par perf permettent de mieux caractériser l'utilisation de ressources en identifiant les points chauds du code, et en fournissant des statistiques et graphes de flammes (flame graph) comme celui montré en exemple à la figure 2.1.

1. Utilitaire d'historique Java d'Apache

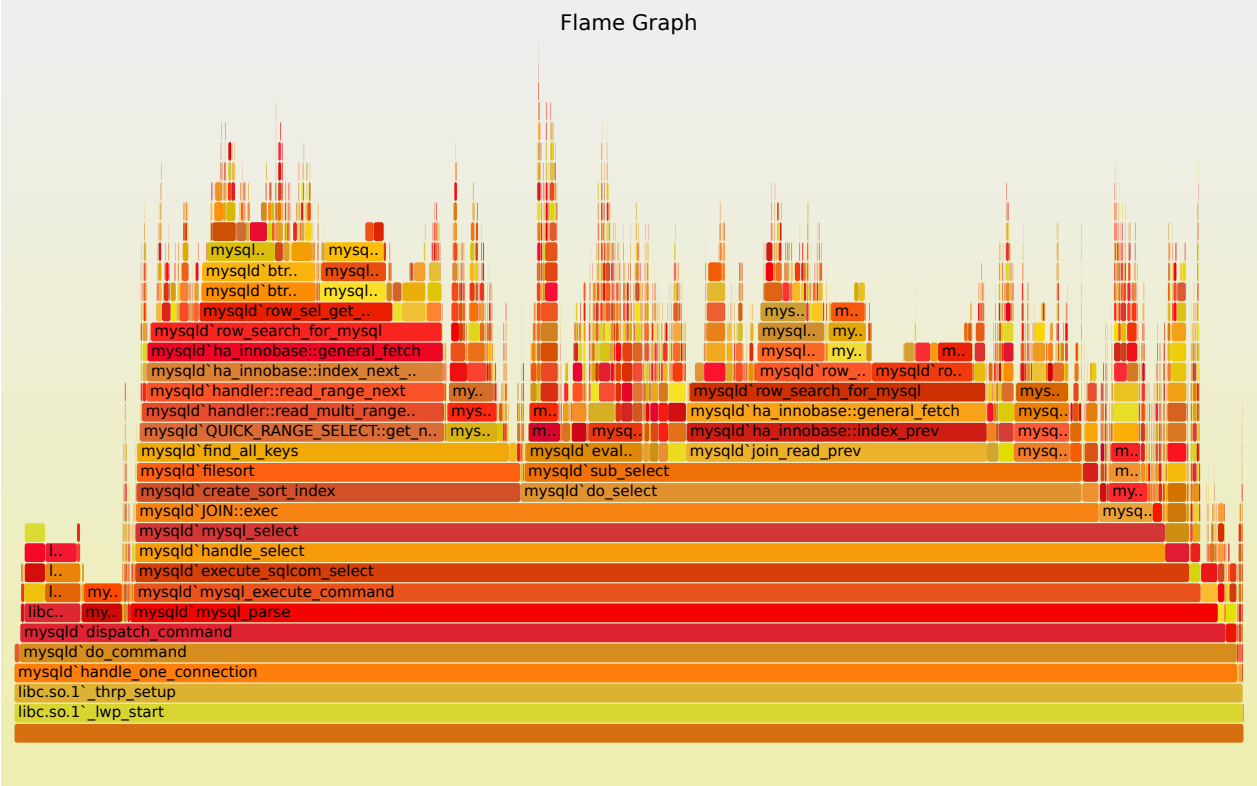



Figure 2.1 Graphes de flamme montrant les cycles processeur utilisés par MySQL [2]

MPI Parallel Environment (MPE) [13] est une suite de logiciels pour Message Passing Interface (MPI) contenant des routines, profileurs, débogueurs et visualiseurs utiles pour instrumenter et déboguer des programmes destinés à rouler sur des systèmes distribués. Bien que développé et fourni avec l'implémentation MPICH – une implémentation gratuite et portable de MPI – MPE est compatible avec toutes les implémentations de MPI. MPE dispose d'une bibliothèque permettant de créer des historiques des exécutions, en enregistrant notamment des informations sur les appels MPI. Ces historiques peuvent être sous forme d'événements, binarisés – format CLOG – ou non – format ALOG, ou sous forme d'états – format SLOG (2.3). MPE inclut aussi les logiciels nécessaires à la visualisation de ces historiques : UpShot, Nupshot et Jumpshot (2.2). MPE dispose aussi de capacités de profilage, mesurant le temps passé dans les routines MPI.

## 2.2 Visualiseurs de traces

Les données de traçage produites sont souvent massives, nous avons travaillé avec des traces de l'ordre du gigaoctet, contenant des dizaines de millions d'événements. Extraire des informations pertinentes avec des outils tels `babeltrace`, qui convertissent les traces binaires en un format lisible par l'homme, demeure une tâche ardue, comme le montre l'exemple de trace à la figure 2.2. Les outils présentés servent à créer des visualisations plus graphiques des traces et d'en extraire automatiquement des statistiques utiles.

Google a intégré des fonctionnalités de traçage au sein du navigateur Chromium [14] pour permettre aux développeurs d'identifier les goulots d'étranglement provenant d'applications JavaScript, C++ ou autres. Le traceur Chromium peut enregistrer la pile d'appels, des événements de profilage, les états des couches HTML et les communications entre les processus, en format JSON, complété par des impressions d'écran du rendu. Le visualiseur affiche sans problèmes l'ensemble des fils d'exécution du navigateur ou de profondes piles d'appels. Toutefois, la trace entière est chargée en mémoire ce qui peut poser problème pour la mise à l'échelle à de très longues traces.

```
[10:06:36.923824697] moya syscall_exit_epoll_wait: { cpu_id = 0 }, { ret = 1, events = 0x7F17A80008C0 }
[10:06:36.923831911] moya syscall_entry_ioctl: { cpu_id = 0 }, { fd = 22, cmd = 62981, arg = 0 }
[10:06:36.923866718] moya syscall_exit_ioctl: { cpu_id = 0 }, { ret = 0, arg = 0 }
[10:06:36.923867568] moya syscall_entry_ioctl: { cpu_id = 0 }, { fd = 22, cmd = 2148070920, arg = 139740057991616 }
[10:06:36.923868149] moya syscall_exit_ioctl: { cpu_id = 0 }, { ret = 0, arg = 139740057991616 }
```

Figure 2.2 Exemple de données recueillies dans une trace noyau LTTng, formaté pour la lecture par babeltrace.

Jumpshot [15] est le visualiseur de traces de la suite MPE. Ce logiciel fournit des visualisations à des niveaux de détails très fins, mais peut aussi produire des analyses et synthèses d'exécutions des programmes. Il peut afficher l'état des noeuds du système distribué au cours du temps, ainsi que les messages échangés entre ces derniers. Les messages sont représentés par des flèches reliant le fil émetteur, à son temps d'émission, au fil émetteur, à son temps de réception. Il est aussi capable de résumer en un histogramme les temps passés dans chaque état, par chaque noeud, ou la durée totale des messages pour chaque paire de noeuds. Jumpshot simplifie largement le calcul du surcoût de MPI dans les programmes distribués. Il dispose aussi de fonctionnalités de recherche pour trouver des événements rares parmi toutes les informations disponibles. Ce visualiseur a recours à une structure de données slog2 (expliquée à la section 2.3) pour maintenir une bonne réactivité même pour des systèmes complexes depuis sa quatrième itération.

Paraprof [16] est le visualiseur/analyseur de la suite Tuning and Analysis Utilities (TAU) dédiée au profilage et au traçage de programmes parallèles. Il utilise les informations de traçage et profilage pour produire un résumé d'informations, permettant de comprendre et se mettre à l'échelle des centres de calcul haute performance. Les histogrammes 3D, largement utilisés dans le programme permettent d'afficher une grande densité d'information sur les états et les profils des fils et des noeuds. Paraprof peut aussi afficher le temps passé dans les appels par chaque fil et des graphes d'appels. Les données sont stockées dans une structure appelée CUBE Uniform Behavioral Encoding (CUBE) arrangeant les informations dans un fichier XML organisé en 3 dimensions. La première dimension correspond aux métriques, qui sont organisées dans un arbre tel que chaque métrique est un noeud, défini par son nom, unité de mesure et noeud parent. La seconde dimension correspond au programme, organisé en un arbre d'appels, avec des numéros de ligne du code. La troisième dimension correspond aux systèmes, organisés en un arbre représentant la structure d'une machine avec des niveaux pour la machine, chacun de ses processeurs SMP, les processus et les fils.

Aftermath [17] est un logiciel libre pour visualiser et analyser les traces de tâches parallèles du cadriciel OpenStream. Ces traces peuvent contenir des états, compteurs ou tâches. Après exécution, Aftermath agrège les données de points de trace, les statistiques de l'exécution OpenStream ainsi que les compteurs de performance. Aftermath dispose d'une vue chronologique affichant les états des fils d'exécution dans le temps avec des flèches pour les communications entre fils, et peut aussi résumer la distribution de durée des tâches et matrices des communications entre les noeuds sur l'intervalle de temps affiché. D'après ses développeurs, ce logiciel supporte des traces de plusieurs gigaoctets tout en restant réactif grâce à l'utilisation d'arbres binaires et d'arbres à intervalles augmentés en arrière-plan pour stocker les adresses et les tâches.

Pajé Visual Trace Explorer (ViTE) [18] est un visualiseur libre pour les traces TAU, Open Trace Format (OTF) ou Pajé. ViTE se spécialise dans les systèmes parallèles utilisant MPI, OpenMP ou PThreads. Il peut afficher des millions d'événements par vue et représenter des grappes de calcul massives grâce à l'utilisation d'arbres binaires pour stocker les informations et d'OpenGL pour le rendu. Une fois importés, les événements et états sont stockés dans une liste et triés par temps de début, avant d'être insérés dans un arbre de tri binaire. L'arbre est ordonné par l'estampille des événements, ou le temps de début pour les intervalles. Pour accélérer le rendu, les intervalles plus courts que la résolution affichable ne sont pas filtrés. Bien que les arbres permettent des accès aux données en temps logarithmiques, ces derniers sont en mémoire et requièrent le tri d'une liste à priori, ce qui limite la capacité de mise à l'échelle de la solution.

Trace Compass [19], (anciennement Linux Tracing Toolkit Viewer (LTTV)) est le visualiseur de Tracing and Monitoring Framework (TMF), développé par le laboratoire DORSAL. Ce visualiseur extensible est capable de réaliser de nombreux types d'analyses sur des traces provenant de LTTng ou d'autres traceurs. Trace Compass est notamment compatible avec les traces Best Trace Format (BTF), Common Trace Format (CTF), GDB et pcap [2]. Il propose de nombreuses analyses, dont l'état des fils, l'utilisation des ressources CPU, mémoire, disque, de pile d'appel, de latence d'appels système et de chemin critique. Ces analyses reposent sur la transformation de traces avec des événements ponctuels en un ensemble d'états – avec temps de début et de fin – des ressources et objets étudiés. De nouvelles analyses et vues sont faciles à rajouter en chargeant un fichier XML les définissant. Il est bâti sur le cadriciel Eclipse et utilise le SHT en arrière-plan pour stocker les résultats d'analyses sous forme d'état et les retourner rapidement pour l'affichage.

Vampir [20] est une suite de logiciels pour mesurer les calculateurs haute performance (High-Performance Computing (HPC)). La composante VampirTrace permet d'instrumenter et tracer des programmes séquentiels, MPI, OpenMP ou hybride MPI-OpenMP. Les traces au format OTF peuvent contenir des données extraites des compteurs de performance matériels, des métriques sur l'allocation mémoire, l'activité en entrée/sortie et sur les appels de fonctions. La composante VampirServer permet aux visualisations et analyses de se mettre à l'échelle en utilisant des principes de relation client/serveur, tels que laisser les données (massives) des traces sur les serveurs qui les ont créés ou utiliser des analyses parallèles pour réduire le temps de traitement, afin de limiter le volume de données transférées et traitées par le client. Le visualiseur Vampir peut afficher des chronologies avec les états des processus, les messages échangés et les valeurs des compteurs au cours du temps. La mise à l'échelle est

2. packet capture : API pour capturer et enregistrer les paquets réseaux

assurée par la répartition des tâches de lecture des traces, d'analyse et de synthèse au sein de la grappe MPI.

TraceAlyzer [21] est un visualiseur et analyseur de traces propriétaire. Il est compatible avec les traces LTTng, mais est spécialisé pour les systèmes d'opération temps réel, tels FreeRTOS, Micrium $\mu$C/OS, SEGGER embOS, Wind River VxWorks, On Time RTOS-32 ou encore Wittenstein SafeRTOS. Il est capable de tracer les systèmes embarqués avec un surcoût minimal en profitant de connexions JTag ou par partage du système de fichiers embarqué avec le système hôte, tel que montré à la figure 2.3. Ses 25 vues peuvent être utilisées pour mieux comprendre les goulots d'étranglement sur des plateformes embarquées. Cela comprend des vues de la charge CPU, des durées des tâches et interruptions, des communications entre tâches, ou encore l'historique des objets noyau. Toutefois, les traces sont stockées en mémoire, ce qui limite la capacité à se mettre à l'échelle du logiciel, son fabriquant conseillant de limiter les traces entre 4 et 40 minutes selon la capacité mémoire de la machine utilisée pour visualiser.

## 2.3 Structures de données

Pour faire face au volume de données des traces, les visualiseurs ne peuvent se contenter de stocker les informations dans des tableaux, en lesquels les recherches se feraient en temps linéaire et dont l'utilisation mémoire serait excessive pour les traces les plus volumineuses. Nous présenterons les structures fréquemment utilisées pour les accès rapides ainsi que celles spécifiques aux logiciels ci-dessus.

Les arbres de recherche binaires [22] sont fréquemment utilisés pour les tables de correspondance associant une clé à une valeur, à cause de leur rapidité en recherche, insertion et suppression. Ce sont des arbres binaires, chaque noeud a au plus 2 enfants et chaque noeud est défini par une clé. Ces arbres sont triés sur les clés tel que pour tout noeud ayant pour clé $k$, toutes les clés du sous-arbre gauche seront strictement inférieures à $k$ et toutes les clés du sous-arbre droit seront strictement supérieures à $k$. Lors de la recherche de la valeur associée à une clé $c$, l'arbre est parcouru à partir du noeud racine, en descendant dans le sous-arbre gauche si $c$ est inférieure à la clé d'un noeud, droit si $c$ est supérieure, ou s'arrêtant au noeud si $c = k$. De la même manière, pour insérer une valeur avec une clé $i$, $i$ est comparée aux valeurs des clés des noeuds, pour descendre dans le bon sous-arbre jusqu'à atteindre un noeud n'ayant pas d'enfant à la place de $i$, un noeud de clé $i$ et portant la valeur est alors déclaré comme l'enfant manquant. Les opérations de suppressions sont triviales, si le noeud à supprimer est une feuille, il suffit de le retirer de l'arbre. Si l'on veut supprimer un noeud ayant un seul enfant, on remplace le noeud par son enfant. Pour supprimer un noeud $P$ ayant

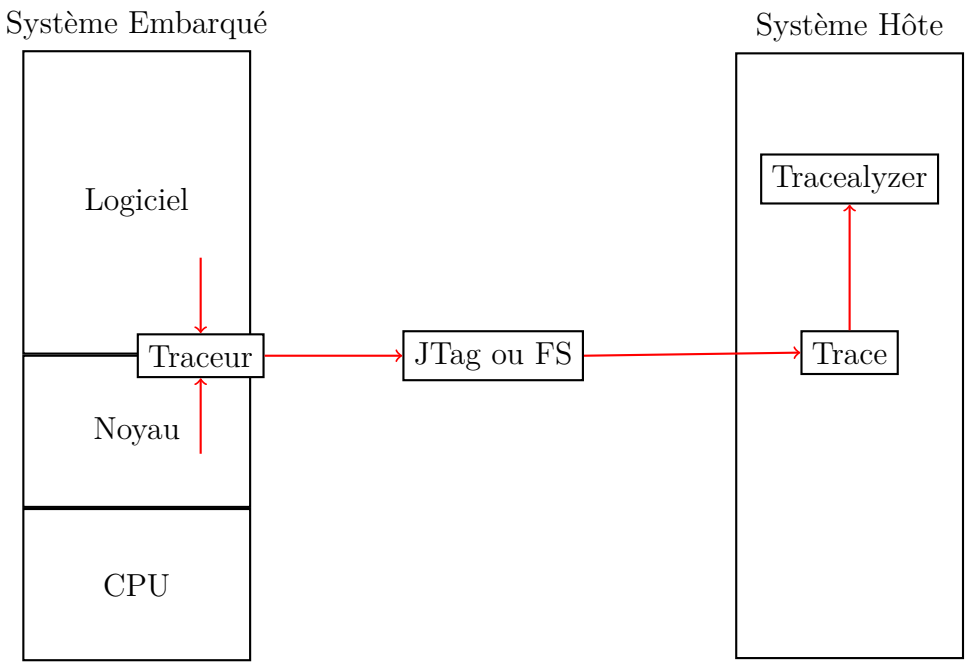



Figure 2.3 Architecture de traçage de Tracealyzer

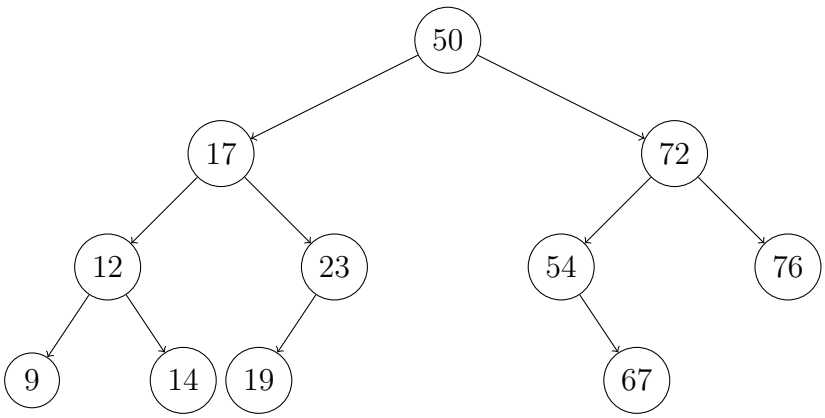



Figure 2.4 Arbre de Recherche binaire

deux enfants, il faut choisir l'un des deux enfants $E$ pour le remplacer, continuer à supprimer récursivement jusqu'aux feuilles. Le choix de l'enfant à supprimer détermine si l'arbre reste équilibré ou non.

Les arbres à intervalles (Interval Trees) [23] sont des structures d'arbres conçues pour être efficaces pour retourner les intervalles superposés à un temps ou un intervalle de temps. Il existe différentes implémentations dans la littérature, le visualiseur `Aftermath`(2.2) utilise des arbres à intervalles augmentés(2.3), tandis que d'autres ont recours à des arbres à segments.

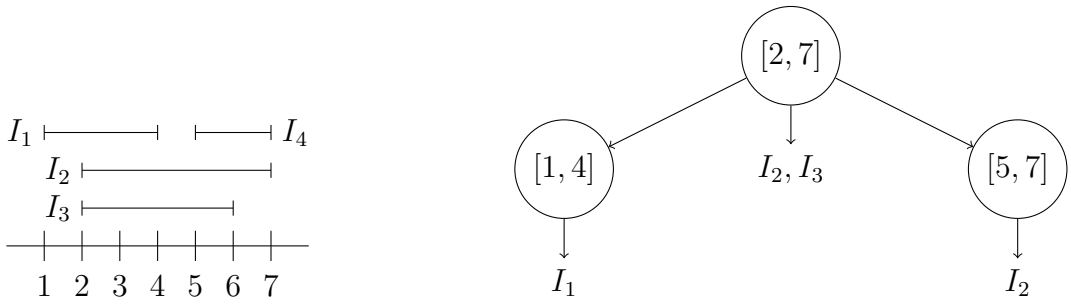



Figure 2.5 Arbre à intervalles augmentés des Segments $\{I_1, I_2, I_3, I_4\}$ et modèle des segments

Les arbres à intervalles augmentés [24] sont basés sur arbres de recherche binaires ou des arbres de recherche équilibrés. Le temps de début des intervalles stockés sert à l'ordre dans la structure. Les noeuds sont "augmentés" par le temps de fin le plus tardif des intervalles contenus dans le sous-arbre. Ce temps de fin permet de réduire le nombre de noeuds que l'algorithme de recherche doit parcourir. Toutefois, il doit être mis à jour sur toute la branche au besoin, lors de l'insertion d'un intervalle qui finirait plus tardivement.

Les arbres à segments [25] sont particulièrement efficaces pour trouver les intervalles superposés à une certaine valeur. Eux aussi sont basés sur des arbres de recherche binaires. Chaque noeud est défini par un intervalle, tel que cet intervalle est contenu dans celui du parent et que les intervalles des deux enfants ne se recouvrent pas, les feuilles ayant l'intervalle entre deux temps le plus petit possible pour pouvoir décrire les segments contenus. Les noeuds non-feuilles contiennent une liste des segments contenant leur intervalle. Pour éviter la duplication, seul le noeud le moins profond sur un chemin donné aura la référence vers le segment. L'arbre est statique, pour le bâtir, une liste des bornes supérieures de tous les segments est établie, puis triée. Des intervalles entre les temps de fin consécutifs sont insérés dans un arbre binaire de recherche.

Les B-Trees [26] figurent parmi les premiers arbres utilisés pour indexer des structures de données en mémoire externe. Les arbres B de degré $d$ étendent les arbres de recherche binaires en donnant à chaque noeud entre $d$ et $2d$ clés et entre $d+1$ et $2d+1$ enfants, excepté le

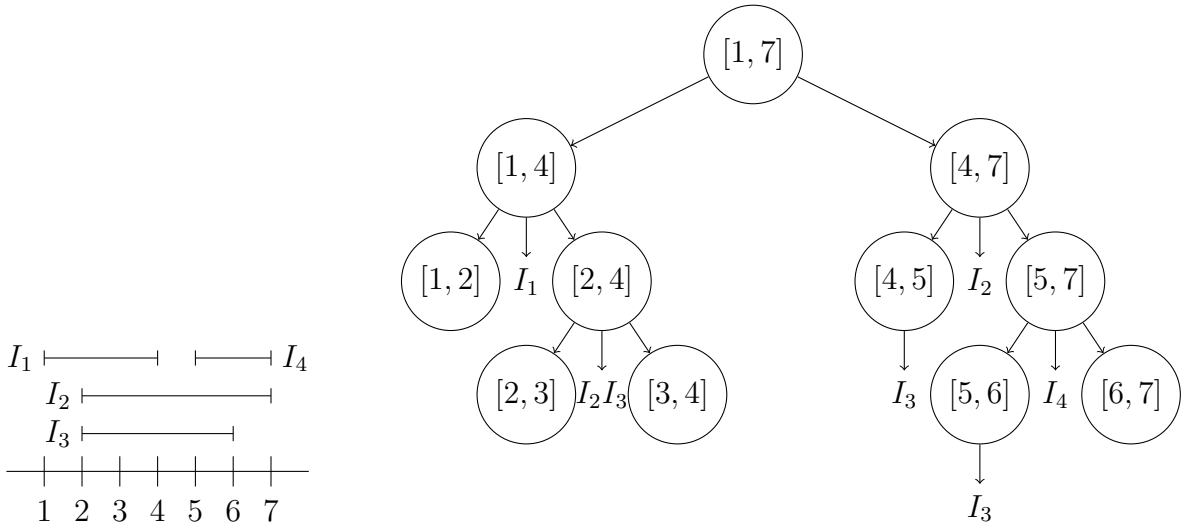



Figure 2.6 Arbre des Segments $\{I_1, I_2, I_3, I_4\}$ et modèle des segments

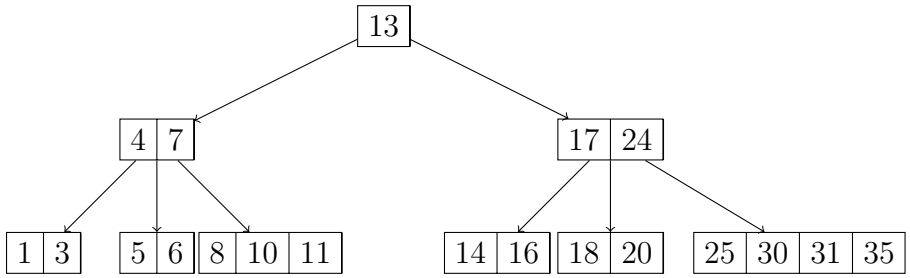



Figure 2.7 Arbre B de degré 2

noeud racine qui peut avoir moins que $d$ enfants. Les clés contenues par le sous-arbre du $(i+1)^{\text{e}}$ pointeur doivent être comprises entre celles de la $i^{\text{e}}$ et $(i+1)^{\text{e}}$ clé – le premier et le dernier sous-arbre ayant respectivement juste une borne supérieure et une borne inférieure. Les recherches dans les arbres B se font également à partir du noeud racine, et progressent dans l'enfant correctement délimité par les bornes, jusqu'à atteindre la clé demandée ou une feuille. Les insertions doivent maintenir l'arbre équilibré : la clé est insérée dans le noeud feuille résultant de la recherche. Si après insertion, la feuille contient $2d+1$ enfants, une procédure de division des noeuds est déclenchée récursivement, de la feuille à la racine : les $2d$ plus petites clés vont en un noeud, les $2d$ plus grandes en un autre, tandis que la médiane est insérée dans le parent pour séparer les deux enfants fraichement créés. Pour supprimer une valeur, on commence par trouver le noeud en question, cela est trivial si la clé est dans une feuille, sinon il faut remplacer la clé par la plus petite clé de la feuille la plus à gauche du sous-arbre à sa droite. Après suppression, on vérifie que les noeuds contiennent toujours plus de $2d$ clés, sinon les clés sont redistribuées avec son voisin de gauche, afin que leur nouveau séparateur soit la médiane de l'ensemble des clés des deux noeuds et leur précédent séparateur. Si les deux noeuds contiennent moins de $2d$ clés, ils sont concaténés avec leur séparateur en plus. À la figure 2.7, nous montrons un exemple d'arbre B de degré 2.

L'arbre à tampon (Buffer Tree) [27] regroupe un ensemble de méthodes pouvant être utilisées pour convertir efficacement des arbres d'intervalles en mémoire principale en arbre en mémoire externe. Les travaux permettent d'accélérer des arbres B de haut degré sur disque, en associant à chaque noeud interne un tampon en mémoire principale, de taille du même ordre de grandeur que le degré de l'arbre. Les opérations sont accumulées dans les tampons jusqu'à ce qu'un seuil de remplissage soit atteint, puis les opérations en tampon sont exécutées récursivement dans les enfants du noeud au tampon plein jusqu'aux feuilles. Les opérations portent une estampille de temps afin d'éviter d'avoir des conflits, notamment lors des supressions et insertions sur les éléments de même clé. L'arbre est maintenu équilibré par des opérations de regroupement et partitionnement de noeuds similaires à celles des arbres B, réalisées lorsque les tampons des feuilles sont vidés.

Les B-Trees Multi Version [28] stockent des intervalles de la forme :$< clé, v_{début}, v_{fin}, valeur >$, où la clé est unique à chaque version et $v_{début}, v_{fin}$ sont les numéros de version de la durée de vie de l'élément. Un exemple d'arbre MVB est disponible à la figure 2.8. Les intervalles qui n'ont pas été supprimés ont pour $v_{fin}$ le métacaractère "*". Chaque opération (insertion ou suppression) crée une nouvelle version. L'arbre comporte une racine de B-Tree pour un intervalle de versions. L'ordre de l'arbre B sous-jacent se fait toujours sur la clé des intervalles. Toutefois, il peut exister plusieurs versions de chaque noeud, avec des versions "passées" et une version "en cours". Dans les noeuds passés, la version $v_{fin} = *$ correspond aux intervalles

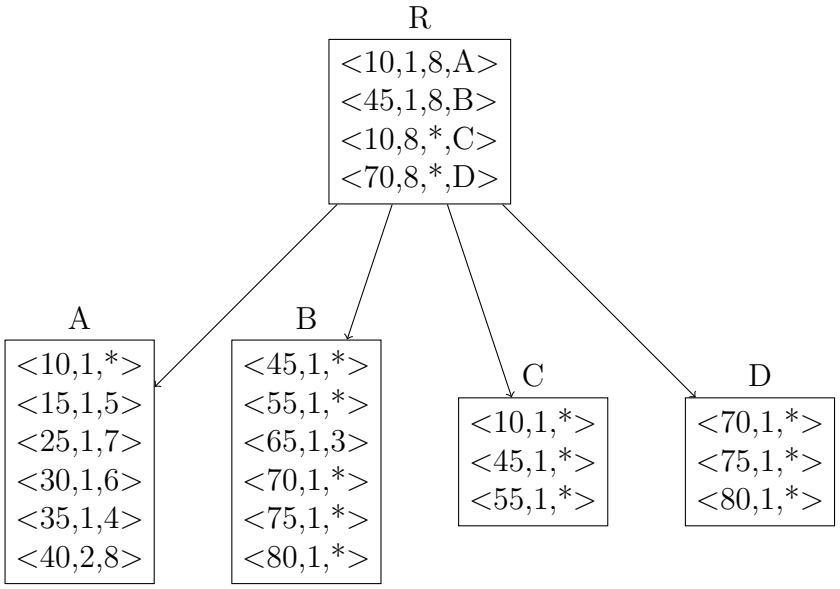



Figure 2.8 Arbre B MultiVersion

qui n'étaient pas supprimés avant la fermeture du noeud. Un nombre d'intervalles minimum $d$ est imposé aux noeuds, sauf aux racines, et un nombre d'intervalles maximum $b$ est imposé à tous les noeuds. Lorsqu'on insère un intervalle dans un noeud plein, ce dernier est fermé, et une nouvelle version, ne contenant que les intervalles ouverts ($v_{fin} = *$), est créée. Les noeuds parents associent aux clés les versions de début et de fin de leurs enfants.

Les Quad-Trees [29] sont très utilisés dans les Système d'information géographique (SIG) pour partitionner des espaces bi-dimensionnels, où leur granularité peut être adaptée à la définition requise pour chaque zone. Les noeuds des Quad-Trees sont définis par des bornes carrées ou rectangulaires de l'espace géographique qu'ils décrivent. Chaque noeud peut être une feuille ou avoir quatre enfants, chacun borné par un quart de l'espace du parent. Les noeuds ont une capacité limitée, lorsqu'un noeud est plein, il faut le partitionner, et répartir ses données parmi ses quatre nouveaux enfants. Comme les bornes ont une forme carrée, les Quad-Trees sont bien adaptés aux données homogènes où les deux dimensions représentent des coordonnées. Parmi les variantes du Quad-Tree, le Quad-Tree "région" impose à ses enfants des longueurs et largeurs égales deux à deux, tandis que le Quad-Tree "point" s'apparente plus à un arbre binaire de recherche, chaque noeud étant décrit par un point dans l'espace bidimensionnel qui partage l'espace en quatre pour ses enfants.

Les arbres kd (kd-trees) [30] sont utilisés pour des problèmes de partitionnement de l'espace. Ce sont des arbres binaires pour lesquels chaque noeud est défini par un point $p$ dans un espace de dimension $k$ et une dimension $d$ qui définissent un hyperplan qui sépare l'espace

en deux, perpendiculairement à l'axe de la dimension. Le sous-arbre gauche contient tous les points inférieurs à $p$ le long de $d$ et le sous-arbre droit contient les points supérieurs à $p$ le long de $d$. L'enjeu de la construction de l'arbre kd consiste en le choix de la dimension selon laquelle partager un noeud, pour assurer l'équilibre de l'arbre. La méthode canonique choisit cycliquement l'axe selon lequel orienter le plan, par exemple dans un espace de dimension $d$, les noeuds de profondeur $p$ seront partagés selon la $(p \mod d)^{\text{e}}$ dimension, tandis que le point sera la médiane des points du sous-arbre, afin qu'il soit équilibré.

La structure de donnée slog2 [6] est une structure d'arbre binaire ordonnée par le temps utilisée par MPE (2.1) pour stocker des intervalles correspondant à des statuts ou messages. Chaque noeud est défini par son temps de début et de fin. Le noeud peut être une feuille ou avoir deux enfants, tel que son fils gauche correspond à l'intervalle du temps de début au milieu, et son fils droit au milieu jusqu'à la fin. L'arbre, et donc le noeud racine, recouvre la durée entière de la trace et les intervalles sont insérés dans le noeud le plus profond possible pouvant les contenir. L'enjeu est de déterminer, avant la construction, la profondeur optimale pour assurer un taux de remplissage élevé. Slog2 permet aux requêtes de visualisation de se mettre efficacement à l'échelle, en ne cherchant que dans les noeuds recouvrant l'intervalle de temps affiché par l'écran du visualiseur. Slog2 est aussi conçu pour être bâti en une seule passe à partir d'une liste d'intervalles désordonnés, ce qui est possible en connaissant à l'avance le nombre d'intervalles et le temps de début et de fin de la trace. Toutefois, les auteurs soulignent que, du à la rigidité des relations entre noeuds parents et enfants, l'arbre peut devenir déséquilibré si les intervalles sont concentrés en un petit espace de temps.

Le SHT [1] a été conçu pour stocker les résultats d'analyse d'états sur disque, sous forme d'intervalles : $< clé, temps_{début}, temps_{fin}, valeur >$. Chaque noeud de l'arbre est défini par ses temps de début et de fin et correspond à un nombre prédéfini de blocs de disque dans lequel les intervalles sont stockés – y compris dans les noeuds qui ne sont pas des feuilles. Ces bornes sont nécessairement incluses dans les bornes du noeud parent et ses enfants recouvrent tout son intervalle de temps, sans se superposer deux à deux. Il n'est pas possible de supprimer des intervalles de cet arbre. La figure 1.2 décrit la construction de l'arbre, qui se fait en une passe et produit un arbre équilibré. La construction du SHT commence par une feuille, dont le temps de début est le temps de début de la trace. Les intervalles sont insérés dans le noeud le plus profond possible de la branche ouverte ayant un temps de début antérieur à celui de l'intervalle. Lorsqu'un noeud est plein, la branche du noeud en question jusqu'à la feuille est "fermée" avec pour temps de fin, le temps de fin maximum du sous-arbre et est écrite sur disque. Ensuite une nouvelle branche de même profondeur est créée à partir du parent du noeud en question jusqu'à la feuille. Si le noeud plein est la racine de l'arbre, on lui ajoute un parent, qui devient la nouvelle racine et le début de la nouvelle branche. Le SHT est efficace

pour sérialiser les intervalles résultants des analyses par états séquentielles sur des traces qui ont la particularité de produire des intervalles par temps de fin croissants, ce qui participe à l'équilibre de la structure. Le SHT est ensuite utilisé pour des requêtes retournant l'intervalle d'une clé $k$ au temps $t$, ou l'ensemble des intervalles superposés au temps $t$ pour lesquelles la structure est particulièrement performante. On retrouve toutefois des cas de dégénérescence lorsque de nombreux intervalles recouvrent un même intervalle de temps.

## 2.4 Arbres R-Tree

Les arbres R-Tree [31] sont utilisés pour indexer des données multidimensionnelles. Chaque noeud est défini par un hyper-rectangle, tel qu'il existe une borne inférieure et supérieure pour chaque dimension, et cet hyper-rectangle est nécessairement inclus dans les bornes de son parent. Les bornes définissent quels points de l'espace peuvent être stockés dans quel sous-arbre, et dans quels noeuds effectuer les recherches. L'arbre est équilibré et les données sont référencées à partir des feuilles. L'enjeu est de minimiser le volume et/ou la superposition des hyper-rectangles afin de réduire au maximum l'espace de recherche tout en maintenant l'arbre équilibré et en assurant que son remplissage est satisfaisant. L'algorithme détermine le noeud optimal dans lequel insérer de nouveaux points : celui qui contient déjà les coordonnées ou requiert le moindre élargissement pour les contenir. Quand le nombre de points contenu dans un noeud dépasse un seuil prédéfini, ce dernier est partitionné, créant deux enfants dans lesquels sont répartis les points. Les arbres R-Tree peuvent être utilisés pour des données spatio-temporelles en assignant le temps à une dimension.

### 2.4.1 Partitionnement de noeuds

Le partitionnement de noeuds fait l'objet de nombreuses recherches pour le rendre optimal. Guttman définit trois algorithmes de bipartition dans son article originel :

- La bipartition linéaire trie les points sur une dimension et partage la liste triée en deux.
- La bipartition quadratique choisit les deux points les plus éloignés (les germes) et associe tous les autres au germe le plus proche, chaque groupe forme un noeud enfant.
- L'algorithme exponentiel / exhaustif recherche toutes les combinaisons possibles et choisit celle qui minimise le volume des hyper-rectangles ou leur recouvrement.

Le Tri-Double [32] est censé offrir la performance de la bipartition quadratique de Guttman au coût de la bipartition linéaire. L'algorithme cherche des points dont les coordonnées peuvent partager le noeud avec une superposition minimale. L'algorithme est plus efficace, car il a recours à des listes triées pour chaque dimension, ce qui permet de trouver un point de

partage en itérant sur chaque liste plutôt que de tester toutes les combinaisons possibles.

L'arbre Packed R-Tree [33] étend la bipartition linéaire de Guttman à des arbres de degré supérieur à 2. Il trie les $N$ points du noeud selon une dimension et crée $\frac{N}{n}$ groupes de $n$ points consécutifs. Le procédé est répété récursivement jusqu'à obtenir le nombre ou la taille de sous-groupes désirés.

Le cR-Tree [34] considère que le problème de partitionnement des noeuds est un problème de clustering. Il a recours à l'algorithme des K-Moyennes [35] pour partitionner les points en $k$ sous-groupes.

Le partitionnement de haut en bas fut introduit avec le VAMSplit R-Tree [36]. Contrairement aux approches qui bâtissent des arbres du bas vers vers le haut, celui-ci considère l'ensemble des points à insérer et réalise une bipartition jusqu'à obtenir des partitions de taille suffisamment petite pour rentrer dans une feuille de l'arbre.

### 2.4.2 Variantes du R-Tree

Le R+-Tree [37] est une variante des R-Tree dont les noeuds voisins ne se superposent pas. En effet, lors de l'insertion d'un rectangle, par exemple, si celui-ci recouvre les hyper-rectangles bornant deux noeuds feuilles, le rectangle sera partitionné et inséré dans les deux noeuds. Les auteurs prouvent que le surcoût en espace de cette insertion multiple est largement compensé par les gains de performance sur les requêtes.

Le R*-Tree [38] est une autre variante des R-Trees en modifiant les algorithmes. En effet l'insertion se fait en cherchant le noeud interne en lequel l'insertion augmente le moins possible le volume, ou le noeud feuille en lequel l'insertion augmente le moins possible le recouvrement. L'algorithme de partitionnement est aussi modifié pour choisir selon quelle dimension partitionner afin d'optimiser le recouvrement ou le volume des noeuds créés. L'arbre R*-Tree a aussi recours à la "réinsertion forcée" qui supprime le noeud en lequel le dernier point a été inséré, s'il ne remplit pas des critères de remplissage, puis réinsère ses points dans l'arbre. Ceci permet de restructurer plus fréquemment l'arbre et ainsi diminuer le recouvrement et l'espace gaspillé.

Le Hilbert R-Tree [39] utilise des courbes de remplissage pour ordonner les données multidimensionnelles. Cette implémentation utilise les courbes de Hilbert, illustrées à la Figure 2.9, qui ont en plus la propriété d'être fractales et permettent d'associer des valeurs de Hilbert proches à des objets géographiquement proches. L'algorithme de construction groupe les points aux valeurs de Hilbert consécutives, ce qui produit des hyper-rectangles de petit volume et de forme cubique.

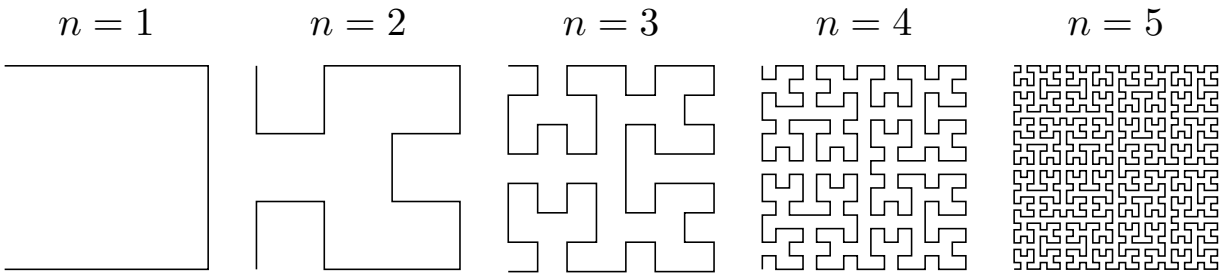



Figure 2.9 Courbe de Hilbert pour n = 1 à 5 [3]

Les arbres R à historique (Historical R-Tree) [40] étendent les arbres R de Guttman, en permettant à l'arbre d'avoir plusieurs états, chacun associé à une estampille de temps. Pour ce faire, chaque noeud est associé à une estampille de temps, et un noeud racine est défini pour chaque état. Les noeuds n'ayant pas été modifiés d'une version à la suivante peuvent être référencés directement par les versions ultérieures pour éviter de dupliquer les données.

L'arbre MV3R (Mutli-Version 3D R-Tree) [41] combine les avantages des arbres B multi version pour les intervalles de courte durée aux avantages des arbres R en lequel les temps de début et de fin sont vus comme des dimensions supplémentaires.

### 2.4.3 R-Tree sur disque

La méthode Small-Tree-Large-Tree (STLT) [42] ou Petit-Arbre-Grand-Arbre sont particulièrement efficaces pour des distributions de points asymétriques. Cette technique cherche à minimiser le temps d'insertion de larges volumes de points tout en maintenant une bonne performance de recherche. Un nouvel arbre R-Tree – le Petit-Arbre – est créé avec les données asymétriques, en dehors de l'arbre principal – le Grand-Arbre –, ce qui permet au Grand-Arbre de continuer à répondre aux requêtes lors de la phase d'insertion. Une fois le Petit-Arbre bâti, ce dernier est inséré à la position optimale du Grand-Arbre.

La méthode Generalized Bulk-Insertion (GBI) [43], ou Méthode d'Insertion Généralisée, étend les résultats précédents aux données sans biais. Cette technique ne se limite pas à un seul petit arbre, mais réalise une opération de *clustering* sur les données à insérer pour produire plusieurs Petits-Arbres qui seront insérés comme dans STLT, ainsi que des points "intrus" qui seront insérés de manière classique dans le Grand-Arbre.

L'arbre R-Tree à tampon ou Buffer R-Tree [44] permet d'utiliser la mémoire système pour rendre plus efficaces les opérations d'insertion. Un nombre prédéfini de points est stocké en mémoire puis, lorsqu'un seuil est atteint, une procédure d'insertion insère efficacement tous

les points. Les auteurs appliquent le même principe aux suppressions et requêtes, montrant des gains d'efficacité pour chaque opération.

### 2.4.4 Algorithmes sur les arbres R

Au-delà de requêtes permettant de retourner les éléments d'un arbre R contenus dans un hyper-rectangle, les propriétés des arbres R peuvent être utilisées efficacement pour de nombreuses autres requêtes.

Utiliser un arbre R pour trouver les k-voisins les plus proches d'un élément est plus efficace que d'itérer à travers tous les points comme le montre l'algorithme de [45]. En partant de la racine, l'algorithme établit une liste triée des enfants du rectangle englobant du plus proche au plus loin. Cet algorithme s'applique récursivement sur chaque noeud. À chaque feuille, la distance du point est comparée aux précédents points les plus proches, qui sont retenus si suffisamment proche. L'algorithme prend fin lorsque les noeuds suivants à explorer se trouvent plus loin que les points les plus proches, impliquant qu'il ne peut y avoir de point proche dans le reste de l'arbre.

Les jointures sur les arbres R permettent d'associer les rectangles d'un arbre R aux rectangles d'un autre arbre R tels qu'ils s'intersectent. Cela est utile en SIG pour appliquer des calques à des régions de l'espace, par exemple. Ceci peut être fait avec des courbes remplissantes pour ordonner des éléments, en itérant avec un parcours en profondeur simultané sur deux arbres (Synchronized Tree Traversal (SST)) ou en itérant sur une liste de rectangles non indexés et cherchant leur association dans l'arbre R. L'algorithme de [46] "balaye" l'espace selon un hyperplan, d'une borne à l'autre sur la direction normale. Cela permet de ne comparer que les rectangles dits "actifs", ceux des deux arbres R, qui intersectent l'hyperplan, réduisant la dimensionnalité du problème.

## 2.5 Bases de données pour séries temporelles

Les bases de données pour séries temporelles sont spécialisées dans le stockage d'informations temporelles, tels les cours boursiers ou les données échantillonnées par des capteurs, ayant une estampille ou un intervalle de temps. Elles répondent à des besoins pour lesquels les bases de données relationnelles montrent leurs limites, notamment car l'ajout de points correspondrait à l'ajout de colonnes en queue de ligne, les points étant souvent ingérés dans l'ordre de leur création qui est chronologique.

Les alternatives pour le stockage de séries temporelles incluent des systèmes profitant des propriétés intrinsèques des systèmes de fichiers modernes. `Daltaminer` [47] exploite les ca-

pacités de caches Adaptive Replacement Cache (ARC), d'écritures asynchrones ZFS Intent Log (ZIL), de sommes de contrôle et de compression du Zettabyte File System (ZFS) ce qui en fait une des bases de données les plus rapides, capables d'écrire 3 millions de métriques par seconde par noeud.

`InfluxDB` a recours à des arbres LSM (Log-Structured Merge-Tree) [4], un arbre sur disque particulièrement efficace pour des charges de travail avec plus d'insertions et de délétions que de requêtes, tels les historiques de transactions. Cette structure repose sur l'utilisation de deux ou plus arbres : l'un en mémoire principale, de taille limitée, et le second sur disque, similaire à un B-Tree, les deux étant ordonnés sur le temps. Les insertions se font dans l'arbre en mémoire, jusqu'à atteindre un seuil de capacité, ce qui déclenche un processus de "fusion" qui extrait des données de l'arbre en mémoire pour les insérer dans l'arbre sur disque. Les noeuds de l'arbre sur disque correspondent à des pages de disque, remplies à 100% pour maximiser l'efficacité des entrées/sorties. Lors du processus de fusion, les données ayant les estampilles les plus anciennes sont extraites de l'arbre en mémoire, noeuds du sous-arbre de l'arbre sur disque correspondant à ces temps, sont lues et stockées dans un tampon. Les données de l'arbre en mémoire sont insérées dedans en respectant la structure ordonnée de l'arbre sur disque. Ces noeuds sont écrits sur disque à de nouvelles pages, tel que montré à la figure 2.10, pour préserver les anciens en cas de besoin pour une récupération. Les lectures commencent par chercher dans l'arbre en mémoire, avant de se propager de manière conventionnelle dans l'arbre sur disque pour retourner toutes les données.

L'arbre TS-Tree [48] est une structure en arbre spécialisée dans le stockage des séries temporelles, avec l'accent mis sur la recherche de séries similaires. Il étend les arbres R, en offrant des noeuds sans superposition et un encodage efficace des métadonnées. En effet, en considérant chaque série temporelle comme un point pour lequel chaque estampille de temps est associée à une dimension, de très hautes dimensions sont atteintes, ce qui rend inefficaces les arbres R. Les arbres TS ont recours à des techniques de réduction de dimensionnalité ce qui permet de limiter la taille et la granularité des séparateurs. Des bornes inférieures et supérieures sur chaque dimension restent stockées dans les métadonnées pour permettre des recherches efficaces. Selon les auteurs, le TS-Tree est plus rapide d'un ordre de grandeur que le R*-Tree.

Finalement de nombreuses autres bases de données pour séries temporelles sont bâties par dessus des solutions de méga données. OpenTSDB [49] utilise Hadoop et HBase, la version Apache du système de stockage distribué pour données structurées de BigTable [50] de Google, ce qui la prédéstine au traitement par lots des données. KairosDB [51] est bâtie sur Cassandra [52], une alternative de HBase, plus performante et accessible pour l'utilisation

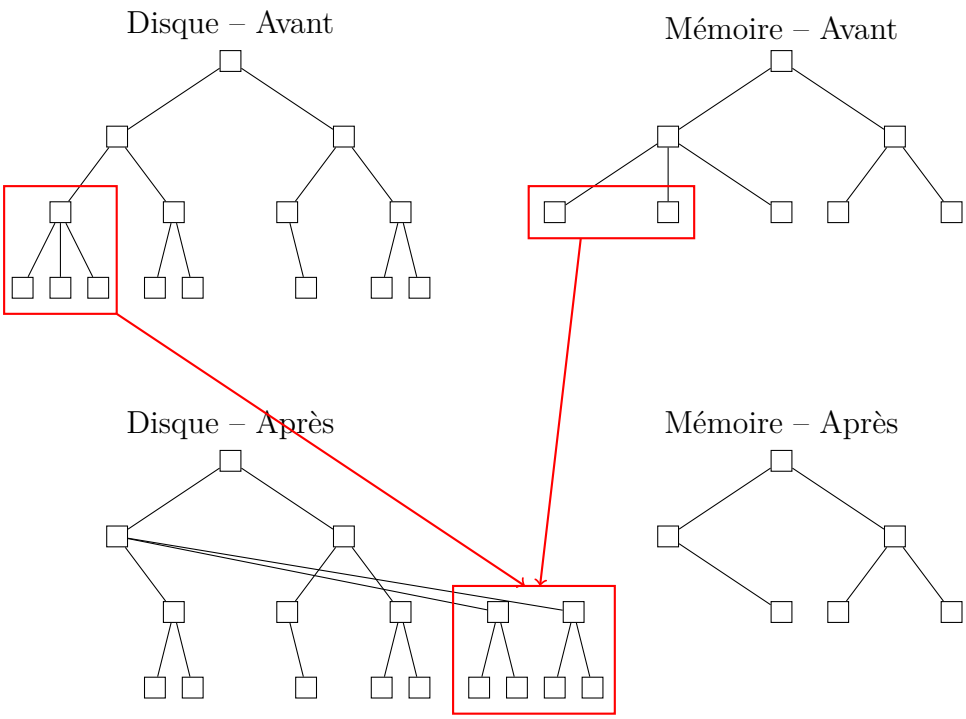



Figure 2.10 Processus de fusion de l'arbre LSM [4]

par des applications ou sites web. Riak TS [53] utilise les principes établis par Amazon avec Dynamo [54], le système de stockage clé/valeur.

## 2.6 Conclusion de la revue de littérature

Les traceurs sont une technique d'analyse dynamique de logiciels capables de générer de gigantesques quantités de données. L'interprétation de ces énormes jeu de données est un défi même pour les logiciels spécialisés, qui doivent les traiter mais aussi fournir des visualisations réactives aux développeurs les utilisant. De nombreuses structures de données en arbre permettent des recherches plus rapide que l'itération à travers une liste de points de trace. Toutefois les solutions pour stocker des intervalles d'états ou de nombreuses séries temporelles hétérogènes localement montrent leurs limites sur les analyses des traces les plus complexes.

# CHAPITRE 3 MÉTHODOLOGIE

Dans cette section, nous décrirons les méthodes utilisées pour comprendre les limites de la structure de données actuelle et chercher les optimisations potentielles.

## 3.1 Visualisation

Afin de mieux nous représenter la structure de l'arbre, nous avons développé un outil permettant de visualiser la structure de l'arbre. Dans la Figure 3.1, nous voyons une impression d'écran d'une vue de l'outil, où le noeud racine est le carré noir, les noeuds au coeur sont des triangles et les feuilles sont des cercles. La couleur correspond au taux de remplissage en intervalles, progressive de rouge – noeud vide – à vert – noeud plein. En cliquant sur les noeuds, il est possible d'afficher son numéro dans l'arbre ainsi que le numéro de son parent ainsi que ses temps de début et de fin, ce qui nous aidera à comprendre les cas de dégénérescence de l'arbre.

## 3.2 Modélisation

Pour comprendre pourquoi le SHT dégénère en taille et profondeur, nous avons modélisé son comportement sur des ensembles d'intervalles simples, pour lesquels les paramètres ayant des conséquences sont le nombre d'attributs et le nombre d'intervalles par attributs. Nous avons principalement utilisé un modèle dans lequel nous avions une trace de durée $T$, et $A$ attributs, avec $I$ intervalles chacun de durée égale, tels qu'il existe un autre attribut dont les intervalles débutent $\frac{T}{AI}$ après les intervalles de celui-ci. Une illustration de ce modèle est disponible à la figure 4.6. Nous considérons aussi que les arbres sont de degré au plus $c$ et que chaque noeud peut contenir un nombre $n$ constant d'intervalles pour simplifier les calculs par rapport au cas réel pour lequel le nombre d'intervalles stockés dépend des données stockées par ces derniers.

## 3.3 Analyse des besoins

Pour répondre aux besoins auxquels répond le SHT, nous avons analysé en détail chaque requête faite à l'arbre à historique d'états pour évaluer la pertinence des requêtes existantes – la requête unique retournant l'intervalle d'un attribut $q$ à l'instant $t$ ou la requête pleine retournant les intervalles de tous les attributs à l'instant $t$. Chaque occurrence nous a permis

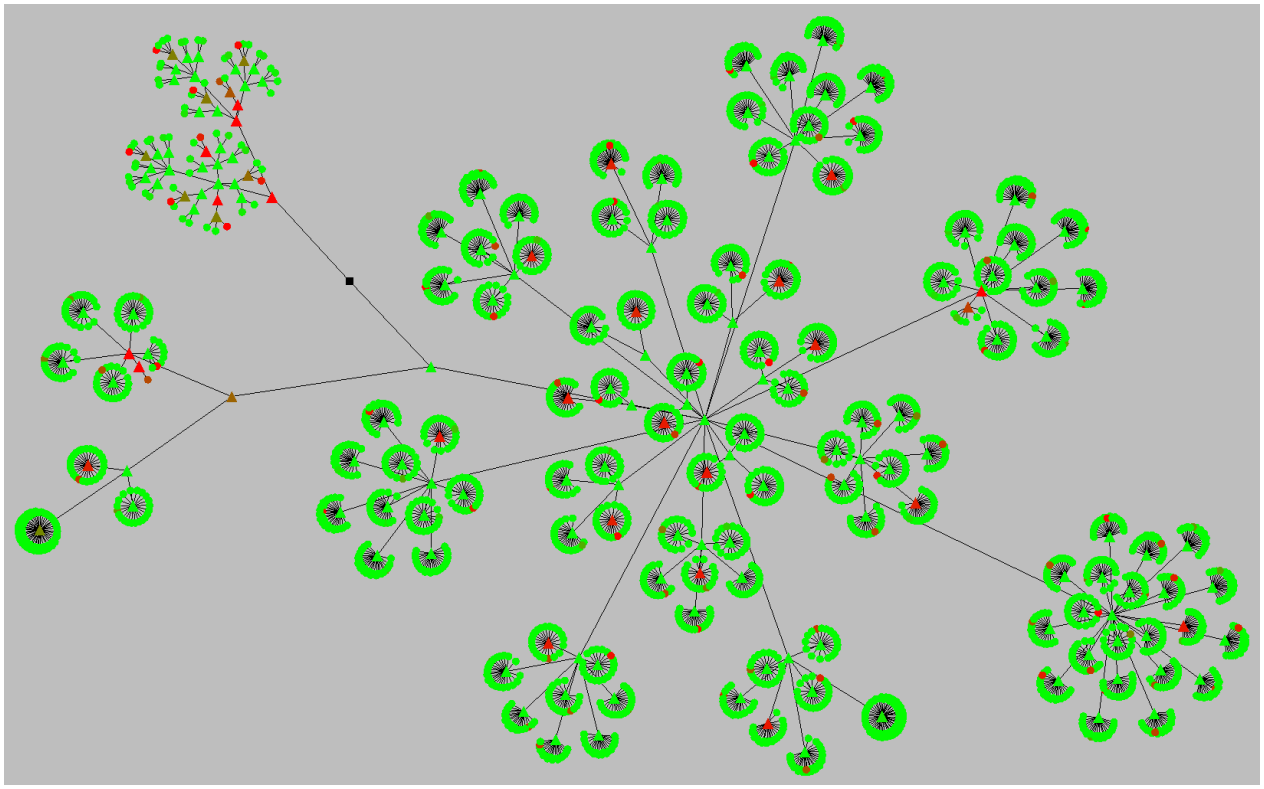


Figure 3.1 Visualisation de la structure d'un SHT

d'analyser si ces deux requêtes sont effectivement les plus efficaces, ou s'il peut exister des alternatives plus efficaces pour parcourir l'arbre et retourner les informations désirées.

## 3.4 Profilage

Pour finir d'optimiser le comportement du SHT, nous avons eu recours au profilage, une forme d'analyse dynamique de programmes, mesurant l'utilisation mémoire et CPU du programme. En particulier, le temps passé dans les appels de fonctions et le nombre d'objets créés nous seront particulièrement utiles pour améliorer la réactivité et diminuer l'empreinte mémoire. Nous avons choisi le profileur Yourkit [55] [1] pour sa richesse en fonctionnalités [56]. Ce dernier est hautement configurable et s'intègre à l'environnement Eclipse, nécessaire au développement de Trace Compass. Il procède par l'ajout de code pour l'instrumentation dans le code machine, dont le surcoût en termes de performances est estimé entre 1 et 5%. Sans cette instrumentation, YourKit peut cependant prendre des clichés de la mémoire et échantillonner l'utilisation CPU, mais elle est nécessaire pour enregistrer les allocations mémoire ou tracer le CPU. La fonctionnalité permettant de comparer les clichés de performance ou de mémoire est particulièrement intéressante pour comparer l'impact entre deux implémentations différentes. Le profilage mémoire, quant à lui est utile pour identifier les fuites de mémoire et analyser quels objets occupent de l'espace mémoire.

## 3.5 Jeu de Traces

Pour évaluer la mise à l'échelle du SHT au nombre d'attributs, nous avons recours à l'analyse noyau, par ailleurs très utilisée. Pour maximiser le nombre d'attributs de cette analyse, il faut générer de nombreux PID car ces derniers font partie du chemin d'accès des attributs. Pour obtenir de très grands nombres d'attributs, nous augmentons la valeur maximale de PID du noyau Linux de sa valeur par défaut, $2^{15}$, à $2^{22}$ et traçons un programme qui crée de nombreux fils.

Toutes les traces ont été effectuées sur Debian avec le noyau Linux `4.6.0-1-amd64`, et LTTng `2.9.0` sur une machine avec un CPU i7-3770 CPU @ 3.40GHz, 16 Go de RAM et un SSD Samsung 850 Pro de 512 Gib. Les principales caractéristiques des traces sont résumées dans le tableau 3.1.

1. `https://www.yourkit.com/java/profiler/features/`

Tableau 3.1 Caractéristiques des traces de test

| nombre de fils | nombre d'événements | taille |
|---:|---:|---:|
| 160 | 7266 | 600K |
| 336 | 13133 | 772K |
| 736 | 19979 | 1004K |
| 1600 | 27722 | 1.3M |
| 3440 | 57721 | 2.3M |
| 7424 | 122552 | 4.4M |
| 16000 | 273666 | 8.9M |
| 34464 | 585565 | 19M |
| 74256 | 1263257 | 40M |
| 160000 | 2721160 | 85M |
| 344704 | 5917942 | 184M |
| 742640 | 12619069 | 390M |
| 1600000 | 27159383 | 839M |
| 3447088 | 58640566 | 1.8G |

## 3.6 Mesure de la performance

Pour mesurer les gains de performance entre diverses implémentations des structures de données, nous avons implémenté des tests JUnit, des modules d'extension spécifiques aux classes désirées, pour limiter le surcoût par rapport à lancer le logiciel Trace Compass entier, avec son interface graphique. Nous avons eu recours aux tests de performance automatisés d'Eclipse [5] qui permettent de mesurer les temps d'exécution et l'utilisation de la mémoire sur différents scénarios de tests, et de calculer les statistiques et évolutions de ces dernières, sous le format de la figure 3.3.

## 3.7 Conclusion de la méthodologie

Les outils et techniques que nous avons décrit ci-dessus nous permettrons de comprendre les difficultés que rencontre le SHT pour se mettre à l'échelle des traces les plus volumineuses. Il nous seront également utiles pour pouvoir facilement reproduire des expériences et les quantifier correctement.

```java
public void testMyOperation() {
        Performance perf = Performance.getDefault();
        PerformanceMeter performanceMeter =
                perf.createPerformanceMeter("nomDuScenario");
        try {
                for (int i = 0; i < nbRepetitions; i++) {
                        performanceMeter.start();
                        /* Block à mesurer. */
                        toMeasure();
                        performanceMeter.stop();
                }
                performanceMeter.commit();
                perf.assertPerformance(performanceMeter);
        } finally {
                performanceMeter.dispose();
        }
}
```

Figure 3.2 Utilisation des tests de performance automatisés d'Eclipse pour mesurer un bloc [5].

```
Scenario 'Random Iterate sorted by length: MapDB store' (average over 10 samples):
  System Time:            1.42s         (95% in [1.36s, 1.48s])
   Measurable effect: 100ms (1.3 SDs) (required sample size for an effect of 5% of mean: 20)
  Used Java Heap:          272M         (95% in [271.75M, 272.26M])
   Measurable effect: 459.39K (1.3 SDs)
  Working Set:               0          (95% in [0, 0])
  Elapsed Process:        1.42s         (95% in [1.36s, 1.48s])
   Measurable effect: 100ms (1.3 SDs) (required sample size for an effect of 5% of mean: 20)
  Kernel time:               0ms        (95% in [0ms, 0ms])
  CPU Time:               1.42s         (95% in [1.36s, 1.47s])
   Measurable effect: 98ms (1.3 SDs) (required sample size for an effect of 5% of mean: 20)
  Hard Page Faults:          0          (95% in [0, 0])
  Soft Page Faults:          0          (95% in [0, 0])
   Measurable effect: 0 (1.3 SDs) (required sample size for an effect of 5% of stdev: 6400)
  Text Size:                 0          (95% in [0, 0])
  Data Size:                 0          (95% in [0, 0])
  Library Size:              0          (95% in [0, 0])
```

Figure 3.3 Résultats produits par les tests de performance automatisés d'Eclipse

# CHAPITRE 4 ARTICLE 1 : R-SHT : A STATE HISTORY TREE WITH R-TREE PROPERTIES FOR ANALYSIS AND VISUALIZATION OF HIGHLY PARALLEL SYSTEM TRACES

**Authors**

Loïc Prieur-Drevon
Polytechnique Montréal
loic.prieur-drevon@polymtl.ca

Raphaël Beamonte
Polytechnique Montréal
raphael.beamonte@polymtl.ca

Michel R. Dagenais
Polytechnique Montréal
michel.dagenais@polymtl.ca

**Soumis à** Elsevier Journal of Systems and Software, Special Issue on Program Debugging, le 31 janvier 2017

## 4.1 Abstract

Understanding the behaviour of distributed computer systems with many threads and resources is a challenging task. Dynamic analysis tools such as tracers have been developed to assist programmers in debugging and optimizing the performance of such systems. However, complex systems can generate huge traces, with billions of events, which are hard to analyze manually. Trace visualization and analysis programs aim to solve this problem. Several programs have resorted to stateful analysis to rearrange data into more query friendly structures.

In previous work, we suggested modifications to the State History Tree (SHT) data structure to correct its disk and memory usage. While the improved structure, eSHT, made near optimal disk usage and had reduced memory usage, we found that query performance, while twice as fast, exhibited scaling limitations.

In this paper, we use R-Tree techniques to improve query performance. We explain the hybrid scheme and algorithms used to optimize the structure to model the expected behaviour. Finally, we benchmark the data structure on highly parallel traces and on a demanding trace visualization use case.

Our results show that the R-SHT retains the eSHT's optimal disk usage properties while providing several orders of magnitude speed up to queries on highly parallel traces.

**Keywords** Data Structures, Tree, Stateful Analysis

## 4.2 Introduction

Understanding the runtime behavior of complex computer systems is a daunting task. Tracing is one of many runtime analysis methods used to instrument and collect data on systems and applications. Compared to logging, tracers have much lower overhead and can produce hundreds of thousands of events per second, at nanosecond precision, providing extremely detailed information on kernel, process and hardware states.

Tracers produce trace files, a series of chronological events, which are optimized for low overhead and data storage but challenging for human operators to understand. A number of software solutions, called trace visualizers, have been developed to facilitate the understanding of these files by providing graphical visualizations, statistics and detailed analysis of certain use cases. These programs perform stateful analysis, that transform event-based data structures into state-based structures, and reorganize data from lists to trees, for faster access.

Indeed, when traces reach gigabyte or terabyte size, efficient data structures are important for maintaining sustainable performance levels for analysis, and low latency for interactive visualizations. Said data structures must be able to scale horizontally – for tracing programs over a large duration – as well as vertically – for tracing systems with many processors, threads and resources.

Among the existing data structures, some are optimized for disk storage, build time or perhaps query performance. When working on trace visualization, the latter is fairly important. R-Trees are a family of data structures used to index multi-dimensional data sets and offer excellent query performance.

In previous work [57], we presented a self-defined tree structure, optimized for external memory storage and with satisfactory query performance. However, we found that query performance scaled linearly to the number of components in the system, which led to slowdowns for the analysis of systems with many threads for example.

In this paper, we propose an enhanced, configurable build algorithm that reorganizes data in the sub-trees so that they reflect properties of an efficient R-Tree.

This paper is organized as follows. First we cover related research on trace visualizers and underlying data structures in section 4.3. Then we present the architecture of the current data structure in section 4.4 as well as that of the evolutions we suggest in sections 4.5 and 4.6. In section 4.7, we model the behavior of the query algorithms before benchmarking them on real-life traces in section 4.8. Finally, we conclude and suggest future work.

## 4.3 Related Work

### 4.3.1 Trace visualizers

In this section, we compare open source trace visualizers that deal with stateful analysis and have a documented data structure to store this information.

`Jumpshot` [6] is the visualization component for the MPI Parallel Environment software package. It displays the nodes' states evolutions over time and the messages that they have exchanged. Jumpshot uses the `slog2` format to reduce the cost of accessing trace data. When using the MPE tracing framework for MPI, users have the option for a state based logging format, in which the tracer directly produces state intervals, as opposed to event based tracing, which produces a list of timestamped events. However, Jumpshot is focused on MPI visualization and doesn't provide detailed analysis capabilities.

`Paraprof` [16] uses tracing and profiling techniques to summarize information, allowing it to scale well to HPC applications. It stores data in a `CUBE` [16] data structure, which is based around a Cube data model, with one dimension for metrics, another for programs and a third dimension for the system. When in memory, Paraprof stores its data as a double level map of vectors, keyed by the metric, then the call path and finally the process number. Despite all its capabilities, Paraprof does not provide stateful information on the systems' performance, rather focusing on metrics.

`Aftermath` [17] provides visualization and analysis for traces from task-parallel work-flows. As part of the OpenStream project, it relies heavily on aggregation of trace points from the application as well as the runtime, and performance counters. Its creators state that the software can scale up to traces of several gigabytes in size while remaining fast thanks to the use of augmented interval trees as a backend. However, Aftermath stores the entire trace in memory, thus limiting its scalability.

Google has built tracing into `Chromium` [14] to help developers identify slowdowns originating from either JavaScript, C++, or other bottlenecks. The visualizer easily scales to the number of threads used by chrome and the flame-graphs of some deep call stacks. Withal, Chromium Tracing is obviously restricted to analyzing Chrome's performance, yet shows the appeal of tracing and analysis for diversified applications.

`Pajé ViTE`[18] is developed for Pajé or OTF traces from parallel or distributed applications. It can scale to display millions of events per view and large computing clusters by storing trace events in a balanced binary tree, which is however limited by the size of the main memory.

`Trace Compass` [19] is the extensible trace visualizer and analyzer for traces generated by the `LTTng` [11] tracer and other tracing tools. It is built using the Eclipse framework and uses State History Trees (SHT) to store state data in a query-efficient structure. It supports a number of different trace formats and offers comprehensive analysis modules. Because of its flexibility, it is equally effective for analysing real-time programs running on a single system, as it is with multi-threading, DSP and GPU architectures, and distributed or virtualized systems.

Distributed systems, which rely on the MPI standard also have a number of dedicated tools to analyse their specificities.

`HPCTraceviewer` is the visualization component in the HPCToolkit. It is used for performance measurement and analysis on large supercomputers. By relying on a client/server architecture, it avoids moving gigabytes of trace files and benefits from the computing power and memory of MPI nodes to process raw data.

The `VampirTrace` [20] visualizer relies on a client/server architecture with parallel servers to scale up for reading large distributed traces. The nodes interact via standard MPI primitives and precompute the required information before sending the results over to the client.

`ScalaTrace` [58] relies on local and global compression to reduce the sizes of MPI traces dramaticaly and preprocess trace comparison. This results in constant size or sublinear growth sizes compared to the number of nodes.

However, when working on huge traces, the aforestated software cannot afford to query directly the trace itself, as the query length could grow linearly with the trace size. This is why such programs transform traces into other data structures that are more efficient for querying. Most programs choose to store "stateful" data, i.e., one object per state [59]. For example, the state of the Attribute "thread/42/Status" could be "Sleep" between two specific time-stamps.

### 4.3.2 Stateful Data Structures

In this section, we compare the data structures used by aforementioned trace visualizers and generic data structures used for multidimensional data. The following structures focus on query performance.

`B-Trees` [26] were one of the first index structures developed to accelerate accesses to external memory data structures. B-Trees extend binary search trees by giving each node between $d$ and $2d$ keys as well as $d+1$ to $2d+1$ pointers to children nodes, in which case the tree is of `order d`. All the values in the sub-tree referenced by the $i^{th}$ pointer are larger than the $i^{th}$

key and smaller than the $(i+1)^{th}$.

`Multi-version B-Trees` [28] store data items of the type $< key, t_{start}, t_{end}, pointer >$ where $key$ is unique for every version and $t_{start}, t_{end}$ are the version numbers for the item's lifespan. It has a number of B-Tree root nodes that each stand for an interval of versions. Each operation (insertion or deletion) creates a new version. Versioning uses live blocks which duplicate the open intervals of the old block and have free space to store future values.

`Interval Trees` [23] are tree structures designed to efficiently find time intervals that overlap a certain timestamp. Different implementations of interval trees exist in the literature. `Aftermath` for example, uses `Augmented Interval Trees` [24] which are based on ordered tree structures. These are typically binary trees or self-balancing binary search trees, where the interval start time is used for ordering. Each node is "augmented" with the latest end time of the associated sub-tree. Knowing the end times of the sub-tree tells the algorithms which nodes they can skip when searching for intervals. The `Centered Interval Tree` implementation is similar to a binary search tree, with each node using a time $t$ as a key such that all the intervals in the left node end before $t$, all the intervals in the right node start after $t$, and the node contains all intervals overlapping $t$. The tree is balanced when the left and right sub-trees contain a similar number of intervals. `Segment Trees` [25] are especially efficient for retrieving segments that overlap a certain value. Segment Trees are based on binary search trees, with nodes defined by the range they span, called "interval". Each segment may have several pointers in the tree, in the shallowest possible nodes, such that these nodes' intervals span the segment but that the parent's interval does not span the segment.

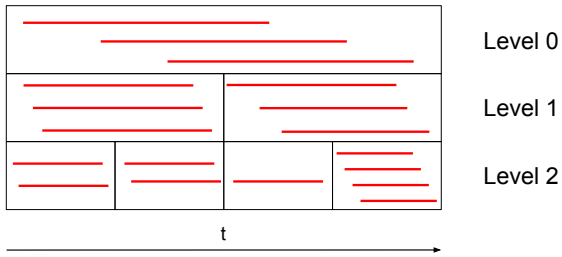



Figure 4.1 Representation of the `slog2` data structure [6]

The `slog2` data structure [6] uses a balanced binary tree structure keyed by time. Each node is defined by a start and end time, such that children nodes' durations are half that of their parent's. Moreover, sibling nodes' times cannot overlap and the root node's duration is that of the trace. Intervals fit in the shortest node that can contain both their end and start time. As nodes and state intervals cannot overlap, the challenge is finding the right depth (leaf node length) to obtain a high fill ratio. Figure 4.1 shows a representation of that data

structure.

The `State History Tree` (SHT) [1] structure was designed with event-based trace analysis and visualization storage in mind. Stateful analysis results, in the form of state intervals: $< key, time_{start}, time_{end}, value >$, are stored in the tree in a single pass through the trace. The SHT is also designed to perform well on rotating media, so each node is mapped to a block on external memory. SHTs support the creation of new keys assigned to a state machine and the update of said states. SHT nodes are defined by their start and end times., All the intervals stored in a node must be included in these bounds. Moreover a child's bounds must be included in its parent's and cannot overlap between siblings. The SHT's construction begins with a single leaf node, siblings and parents are added as nodes are filled. The SHT's structure and limitations are further discussed in section 4.4.

### 4.3.3 R-Trees

R-Trees [31] are used to store multidimensional data. The points stored in each node are mandatorily included in the node's Minimum Bounding Rectangle (MBR), a hyper-rectangle which bounds the points on each dimension of the tree. Children nodes' MBRs are included in their parent's MBR. The challenge is to minimize the volume or overlap of these MBRs in order to limit the number of nodes that need to be searched during queries. Structuring the tree is usually done during the insertion phase. When inserting a new point, the algorithms select the best node into which insert new points, usually the ones with MBR that overlap the point, or require the least enlargement to contain it. When nodes overflow, i.e. the number of points they contain exceeds a pre-defined threshold, the node is "split", creating two child nodes into which points are assigned. R-Trees can be used for spatio-temporal data by assigning the time to one dimension.

#### R-Tree Node Splitting

Guttman originally proposed 3 bi-partition algorithms for the R-Tree:

1. The linear / sort-based algorithm sorts points along a dimension and then splits the resulting list in half.
2. The quadratic / seed-based algorithm finds the two most distant points (seeds) in a node then associates the other points to the seed which is the closest.
3. The exponential / exhaustive split algorithm explores all the possible combinations and chooses the one with the lowest coverage or overlap.

Node splitting is a vast subject of research and a number of algorithms have been proposed to extend this approach.

For instance, `Double-Sorting` [32] offers the query performance of Guttman's quadratic split at the cost of the linear split. It searches for points whose coordinates can divide the node with minimal overlap. This algorithm based on two sortings allegedly offers better splitting on complicated datasets.

The `Packed R-Tree` [33] extends Guttman's linear sorting to trees with more than 2 children, sorting $N$ points along one dimension, then packing them into $\lfloor \frac{N}{n} \rfloor$ consecutive groups of $n$ points. This process is carried out recursively until the groups have the desired number of points or desired number of subgroups.

The `cR-tree` [34] considers that node-splitting is a clustering problem, which can extend further than to 2 children. The authors use MacQueen's popular k-means [35] algorithm in order to split nodes into multiple children.

### R-Tree Variants

`R`$^+$`-Trees` [37] are variants of the R-Tree in which sibling nodes have no overlap. Indeed, the node splitting algorithm allows rectangles to be split, then inserted into several nodes.

`R`$^*$`-Trees` [38] attempt to minimize overlap and coverage by reinserting points from overflowing nodes into the tree, thus reducing the effect of the initial order of the points on the tree's structure. Node split is deferred, in order to ensure that the resulting nodes' fill is higher. Finally, R*-Trees use a topological split that minimizes overlap.

The `Hilbert R-Tree` [39] is an R-Tree variant in which points are ordered and grouped according to their value along the Hilbert curve or other space-filling curves. The fractal properties of these curves tend to group close points together, thus minimizing the overlap and area of nodes.

`Historical R-Trees` (HR-Trees) [40] are a modification of R-Trees to support versioning, by building an R-Tree for each timestamp and sharing common nodes with links between trees. This is efficient when there are few modifications between a small number of versions.

The `MV3R-Tree` [41] combines a Historical R-Tree – for the infrequent state changes and a 3D R-Tree – for the shorter lived states, to benefit from the properties of each tree on the type of data they are better suited for.

**External Memory R-Trees**

The `Small-Tree-Large-Tree` (STLT) [42] method makes it possible to bulk load points into a tree while reducing the time during which it is unavailable for queries. It proceeds by creating a new R-Tree on the side – the Small-Tree – into which the points are loaded. Then, this tree is inserted into the optimal position of the main tree – the Large-Tree. This approach works particularly well for skewed data.

The `Generalized R-Tree Bulk-Insertion Strategy` (GBI) [43] generalizes the STLT approach for less skewed data. The data to bulk load is split into clusters and outliers with a variation of the K-means algorithm. Each cluster is bulk loaded separately with the STLT method, while the outliers are inserted as single points into the tree.

The `Buffer R-Tree` [60] takes avantage of the system's main memory to reduce external memory I/O, by delaying insertions or deletions to the external memory structure until a certain number of operations can be bulk executed in a more efficient fashion.

In this paper, we propose a scalable data structure, which has performance gains compared to previous implementations, is well suited to parallel systems, and offers much improved query times.

## 4.4 Limitations of the State History Tree

In this section, we briefly present the implementation of the State History Tree (SHT) [7], a data structure designed for state storage on external memory, and detail the issues that it encounters when dealing with highly parallel traces.

### 4.4.1 Structure of the State History Tree (SHT)

The State History Tree (SHT) [1] is suited for storage of stateful information that is computed while reading through the trace. It is used for tracking the states of various state machines over the duration of a trace. For example, it is possible to know the status of a process at any time in the trace, when analysing a Linux kernel trace. Trace events produce the transitions in the state machine, and stateful analysis computes the states between transitions, before storing them in the SHT. During the trace analysis, events are processed in chronological order, therefore we track unclosed states with their start times. Every time an event changes a state, we write the ending state interval to external memory and update the current state value and start time.

The stored data takes the form of intervals, which consist of an attribute key, a start time,

an end time and a value: $< key, time_{start}, time_{end}, value >$. The start and end times are specified with a nanosecond granularity. The attribute key is a unique identifier for the object whose state we are tracking. The value is the payload of the interval which can be a null, a boolean, an integer, a long or a character string. For each attribute, there are contiguous intervals from the beginning until the end of the trace.

The SHT is composed of nodes created as the tree is built. A node is defined by a unique sequence number, a start time and an end time. They have a header which contains the sequence numbers of their parent as well as the sequence numbers and start times of their children, for searching purposes. Each node is mapped to blocks on storage media, and can therefore store a limited number of intervals. As the serialized size of intervals can vary based on the type of value they carry, this number is not fixed. The maximum number of children in a SHT node is limited – usually to 50 – and its capacity is limited to $64kB$. The unique sequence numbers of the nodes represent the position of their block in the history tree file.

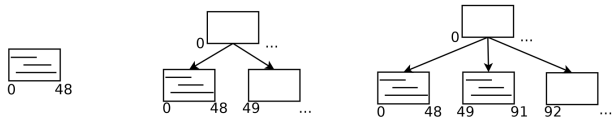



Figure 4.2 Build steps of the State History Tree using an incremental process [7]

The tree is built in a single pass upon performing the state analysis of a trace. As events in the trace are in chronological order, the resulting intervals are generated and inserted with increasing end times. The tree's construction starts from a single node, that has the same start time as the trace. The rightmost branch of the tree is kept in main memory so that intervals can be efficiently inserted into them while the rest of the nodes are kept serialized in external memory until it is necessary to read from them. As a node's time range must be included in its parent, start times of the nodes in the rightmost branch increase from the leaf to the root node, as shown in Figure 4.2. Therefore intervals are inserted into the deepest node with a start time smaller than that of the interval. Once a node $node_{full}$ has reached the maximum number of children or interval capacity, all the rightmost children from itself to the leaf have their end times set to that of the last interval which was inserted $time_{last}$ and are written to external memory. If the node $node_{full}$ has a parent, the rightmost branch is rebuilt with nodes starting at $time_{last} + 1$ to ensure that the tree is balanced. If $node_{full}$ was the root node, a new node, starting at the start time of the trace, becomes the new root node.

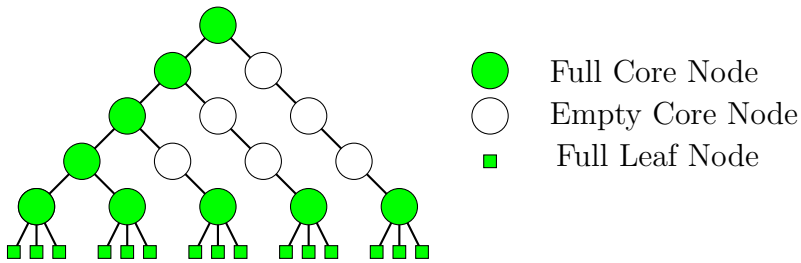



Figure 4.3 Schematization of a State History Tree with many attributes

### 4.4.2 Shortcomings on analysis of large systems

We encountered traces for systems with a large number (30000) of threads, which had a reasonable size (less than $100Mb$), yet created a large SHT file with inefficient storage usage (around 9% of the storage space reserved by the many nodes is actually used to store interval data). Moreover, these traces brought our visualization software to a halt, leading to further investigation.

Since we store intervals from the tree's start time for every attribute (such as threads), the SHT has many intervals that begin at the SHT's start time. The SHT's design imposes that intervals that begin at the SHT's start time fit in the left most node. Therefore, a trace for a many-threaded program leads to a very deep tree compared to a well balanced, high fan-out tree. Indeed, the depth of the tree increases when we try to insert an interval beginning at the tree's start time and it cannot fit into the deeper non filled nodes, as its start time is earlier than theirs. Therefore, it can only fit in the root node and, when the root node is full, another depth level is added. In the case of our trace with many threads, the tree was 206 nodes deep, whereas an efficiently loaded tree for the same data should have been 5 nodes deep.

We have a very deep tree but most intervals fit in leaf nodes, therefore the branches are mostly empty from the leftmost node to the leaves. As the SHT nodes are mapped to external memory blocks, many empty nodes means a very low storage usage rate. As queries on an SHT search down a branch for which the nodes cover the queried time, and most intervals are stored in the leaf nodes, the average query takes a long time.

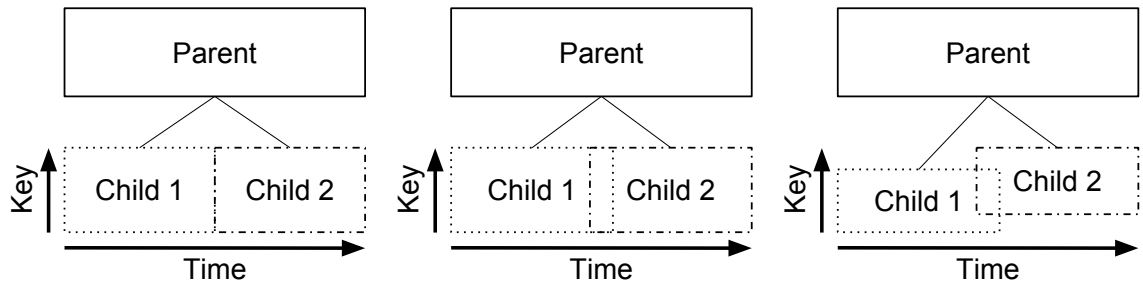
Therefore, we can compute the depth $d$ of a SHT with many attributes, as $d = \frac{A}{n}$, with $A$ the number of attributes and $n$ the average number of attributes that fit per node. This differs from the depth of a packed, balanced tree $d = \lceil log_c(\frac{N}{n}) \rceil$, of degree $c$ with $\frac{N}{n}$ nodes, $N$ being the number of intervals in the tree.

Likewise the number of nodes of a *comb* tree, such as the one in Figure 4.3 will be $\left(\frac{A}{n} - 1\right)\left(\frac{\frac{A}{n}-1}{2} + c\right)$

instead of $\frac{N}{n}$.

Because the tree is very deep and the State System stores its "in progress" branch in memory, the SHT construction may even crash with a JVM OutOfMemoryError on the deepest trees.

## 4.5 The Overlapping State History Tree




(a) Representation of consecutive nodes in SHT

(b) Representation of overlapping nodes in oSHT

(c) Representation of enhanced nodes in eSHT

Figure 4.4 Comparison of the relations between sibling nodes of SHT (left), oSHT (middle) and eSHT (right)

### 4.5.1 Overlapping SHT structure

The overlapping SHT structure has been introduced as the enhanced State History Tree (eSHT) [57]. This structure fixes the original SHT's tendency to degenerate into combs in cases with many attributes.

As explained above, imposing that sibling nodes' time ranges be consecutive causes the tree to degenerate into combs. The consecutive (non overlap) constraint is shown in Figure 4.4(a). We do away with this constraint and use the first inserted interval's start time as the new node's start time. The removal of this constraint ensures that intervals inserted in the future fit into the leaf nodes, which in turn prevents the tree's depth from degenerating. However, the overlap shown in Figure 4.4(b) requires modifications to the data structure and algorithms.

Because sibling nodes overlap, we modify the query algorithm – described in Algorithm 4.1 – to query on sub-trees instead of branches. The different search algorithms are represented respectively in Figure 4.5(b) and 4.5(a). In order to query the correct sub-tree, the children's end times are added to their parent node's header, alongside their start times. A node's time range must still be included in its parents time range.

There are a number of benefits to allowing the nodes to overlap. We can now fit a large number of attributes that cover the same time ranges, which is typical for highly parallel trace analysis, without degenerating into a list. Therefore, the tree should be shallower, so queries may be shorter and the build would use less memory. There should also be far less empty nodes, so better use of storage space would be made, reducing the number of writes when building the tree (and consequently, build times).

### 4.5.2 Enhanced State History Tree

We now consider the data we are handling as multidimensional (time and key), so we add key bounds to the header. These describe the minimum and maximum keys for the intervals stored in the node, as shown in Figure 4.4(c). The core nodes' headers also store their children's bounds, to help narrow down the number of nodes searched during a query. As was the case with the time bounds, a node's key bounds must also be included in its parents bounds. For example, if a core node's header says that one of it's sub-trees has bounds $[t_{min}, t_{max}]$ for the time and bounds $[k_{min}, k_{max}]$ for the keys, there is no point in searching it for a key $k_s$ such that $k_s < t_{min}$ or $k_s > t_{max}$. We call *enhanced State History Tree* (eSHT) the overlapping State History Tree with added key bounds.

### 4.5.3 Search algorithms

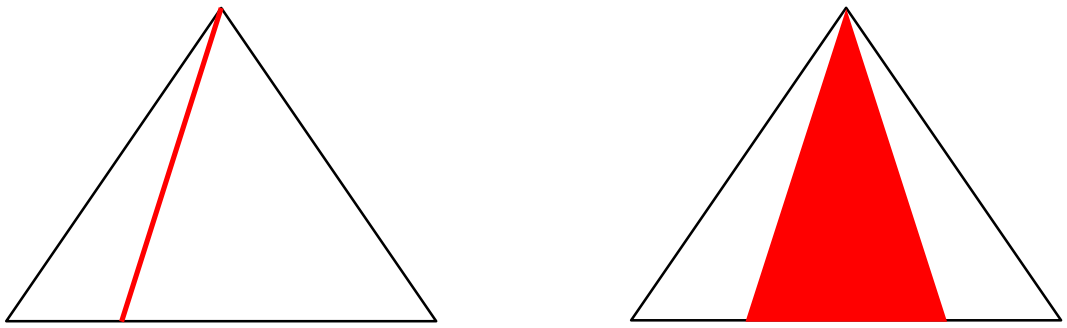
The SHT can be queried for the state of one or all attributes at a time $t$.

— A single query for key $k$ at time $t$ returns the interval for key $k$ that overlaps $t$. The single queries search down the branch of the tree that overlaps $t$ until they find the interval with the correct key and time range.

— A full query at time $t$ returns all the intervals (one per key) that overlap $t$. Full queries search the entire branch that overlaps $t$, and add all the intervals that overlap $t$ – one per attribute – to a list

Concurrent accesses are handled by using a shared-exclusive lock for the current states and one shared-exclusive lock per node.

Unlike SHT and slog2, queries on eSHTs cover a sub-tree, and not just a branch in the tree, since there are potentially several children nodes that contain the relevant time. Therefore, each core node contains an index on the start and end times of each of its children to determine which sub-tree to explore. The method **node.getChildren(k, t)** produces a list of **node**'s children which contain key $k$ and time $t$ in Algorithm 4.1. The **node.getInterval(k, t)** method retrieves the interval intersecting $t$ with key $k$ when it exists in **node**.



(a) Representation of a branch search as used by SHT

(b) Representation of a sub-tree search as used by eSHT

Figure 4.5 Comparison of tree search between SHT (left) and eSHT (right)

**Algorithm 4.1** Single State Query

```
function singleQuery(key, time)
    interval ← null
    /* rootNode is the tree's root node.                    */
    queue ← List(rootNode)
    while interval = null do
        node ← queue.pop()
        if node.type() = CoreNode then
            queue.addAll(node.getChildren(key, time))
        interval ← node.getInterval(key, time)
    return interval
```

### 4.5.4 Comparison of query bounds with SHT

The State History Tree aims for query performance, so we consider that the number of nodes searched per query is a good measure of the data structure's query efficiency. We will theoretically compare the number of nodes searched for queries on our enhanced State History Tree to the original one.

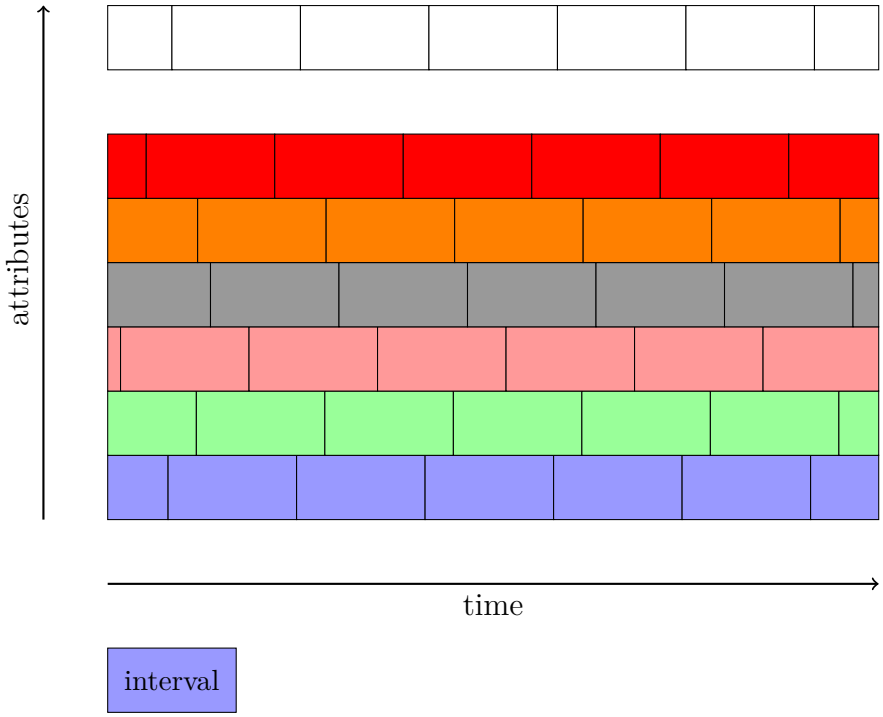



Figure 4.6 Schematization of the intervals in a Tree

The theoretical intervals used for comparison are represented in Figure 4.6. That tree is of duration $T$ and contains $A$ different attributes. Each attribute is split into $I$ equal intervals through the trace. Intervals from different attributes are offset by $\frac{T}{AI}$. The order of the Attributes is shuffled. Each node can store up to $n$ attributes and have up to $c$ children.

On the SHT, queries search down a branch, as explained in 4.5.3, therefore the upper bound is the tree's depth. With our comparison trace set, with its large number $A$ of attributes, we will reach the comb situation. The query bound for SHT $Q_{SHT}$ can thus be formulated in this way:

$$Q_{SHT} = \left\lceil \frac{A}{n} \right\rceil$$

As for the eSHT, we need to determine the number of nodes which overlap the queried time.

The query bound $Q_{eSHT}$ for eSHT can then be expressed as the following:

$$Q_{eSHT} \leq h + \left(\frac{A+n}{n+1}\right) \times \left(\frac{1-c^{-h}}{1-c^{-1}}\right)$$

With $h$ being the depth of the eSHT and $c$ the maximum number of children per node. Considering that, in our example, all the data is in the leaf nodes, we can use the standard formula [28] to compute the tree's height $h$:

$$h = log_c\left(\frac{AI}{n}\right)$$

We approximate $Q_{eSHT}$ by computing the required number of nodes to fit $A$ intervals. Due to the algorithm used to build eSHT and our worst case theoretical trace, all the intervals reside in leaf nodes. As intervals are inserted by increasing end times, the node duration $D$ is:

$$D = \frac{T}{I} + n \times \frac{T}{AI}$$

Where $\frac{T}{I}$ is the interval duration, for the average interval, i.e. neglecting border effects for the first and last intervals of each attribute. We call $\Theta$ the number of nodes in the tree which overlap a time $t$. It can be computed as the ratio of the node duration $D$ over node offset $\Delta t$:

$$\Theta = \frac{D}{\Delta t}$$

Which is then:

$$\Theta = \frac{\frac{T}{I} + n \times \frac{T}{AI}}{(n+1) \times \frac{T}{AI}}$$

And can be reduced to:

$$\Theta = \frac{n+A}{n+1}$$

However, we also have to consider the core nodes which have to be searched to reach the leaves from the root node. Knowing that each core node can reference up to $c$ children, we deduce the following relation:

$$Q_{eSHT} = \sum_{i=0}^{h} \left\lceil \frac{\Theta}{c^i} \right\rceil$$

And then develop the upper bound for $Q_{eSHT}$:

$$Q_{eSHT} \leq \sum_{i=0}^{h} \left( \frac{\Theta}{c^i} + 1 \right)$$

That can then be reduced to the following:

$$Q_{eSHT} \leq \Theta \times \frac{1 - c^{-h}}{1 - c^{-1}} + h$$

While this is slightly larger than the query on a SHT, the average query size on an eSHT is half that of the upper bound, as the intervals are uniformly spread over the possible DFS or BFS search path.

Meanwhile, on the SHT, most branches are empty, as seen in Figure 4.3. We can compute the average depth of intervals in the tree, knowing that intervals are either in the left most nodes, in the deepest core nodes, or in the leaf nodes. If we consider:

— $H$ as the height of the tree
— $N_{\text{leaf}} = c(H-1)$ as the number of **leaf** nodes, of depth $H$
— $N_{\text{core}} = (H-1)$ as the number of **core** nodes, of depth $H-1$
— $N_{\text{left}} = (H-2)$ as the number of **left** nodes, with increasing depths from 0 to $H-2$.

We can express the average depth of nodes containing intervals as the following:

$$d_{\text{avg}} = \frac{\sum d}{\sum N}$$

Which can be developed as:

$$d_{\text{avg}} = \frac{\sum_{i \in \text{leaf}} d + \sum_{i \in \text{core}} d + \sum_{i \in \text{left}} d}{N_{\text{leaf}} + N_{\text{core}} + N_{\text{left}}}$$

And thus:

$$d_{\text{avg}} = \frac{\sum_{i=0}^{H-2} i + (H-1)^2 + c(H-1)H}{H - 2 + H - 1 + c(H-1)}$$

As we know that $H \gg 1$, the equation can finally be reduced to:

$$d_{\text{avg}} \simeq H$$

Therefore, the average eSHT query on traces with many attributes is close to twice as fast as the average SHT query.

### 4.5.5 Query scalability limitations

In early results, we modeled the behavior of the SHT and the eSHT and found the number of nodes that needed to be searched for queries. We found an upper bound of $\frac{A}{n}$ nodes, with $A$ being the number of attributes and $n$ the average number of intervals per node. This upper bound was similar for both trees and the average single query was only twice as fast in the case of the eSHT, as can be seen in Figure 4.7.

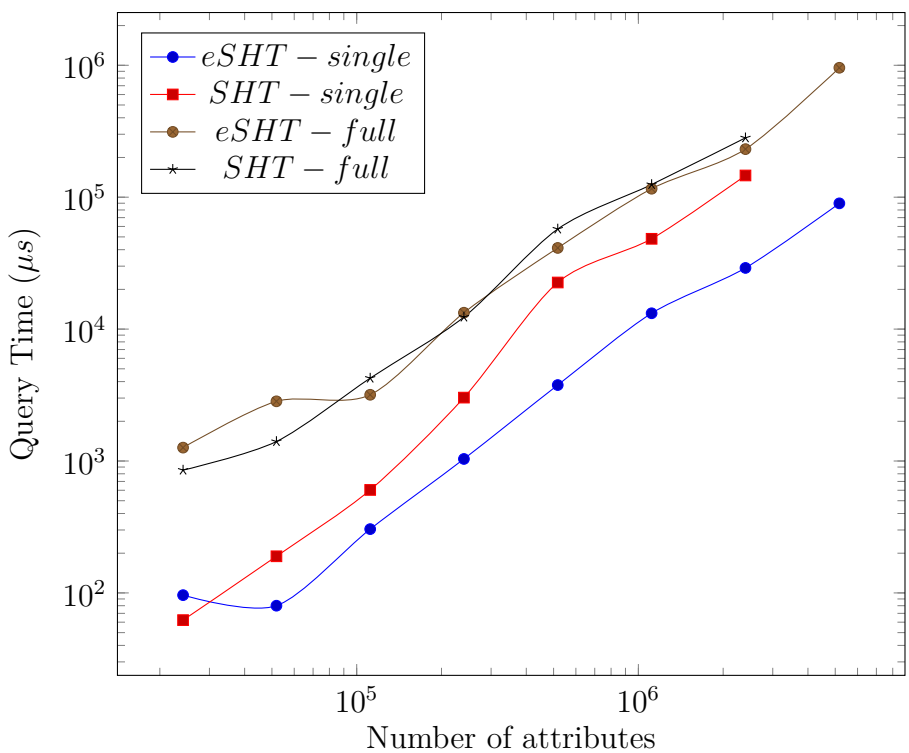



Figure 4.7 Comparison of SHT and eSHT query times for traces with many attributes

To try and understand why the eSHT does not speed up single queries more than twofold, we look at the key range covered by the key bounds discussed in Section 4.5.2. We define the key range by the difference between each node's minimum and maximum key.

The key range histogram in Figure 4.8 shows that the key range distribution is sub-optimal. Indeed, if the intervals had been arranged efficiently in the tree, the range would have been

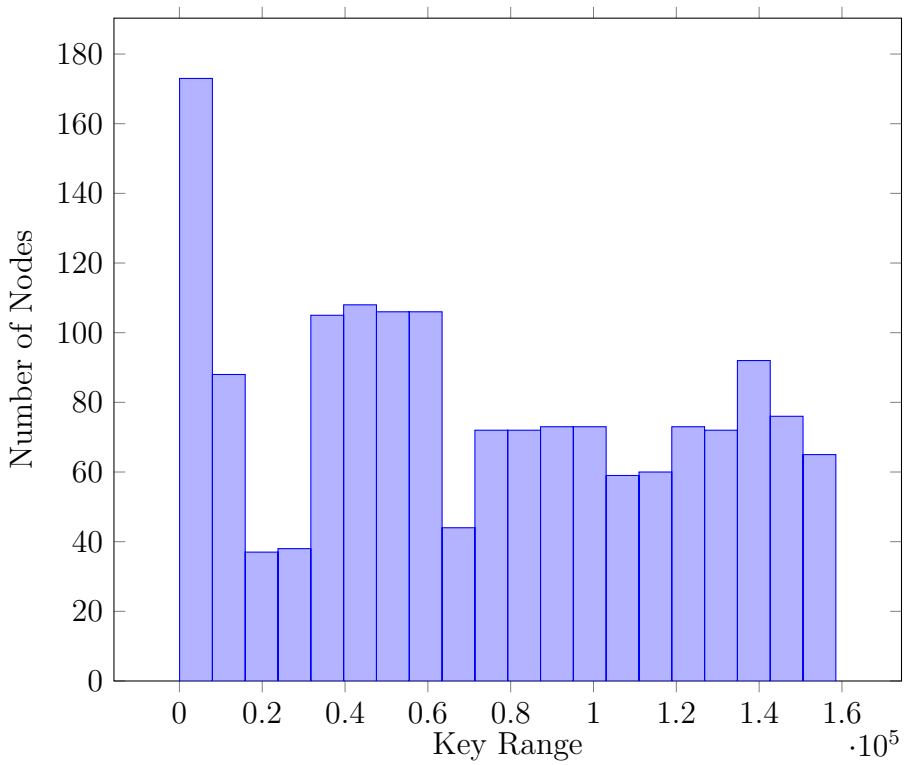



Figure 4.8 Key range histogram for 10k thread trace using eSHT

minimized for deeper nodes and only shallower nodes would have contained the entire range.

## 4.6 R-SHT model, structure and algorithms

In this section, we propose a modification inspired by R-Trees. To prove its relevance and performance gains, we develop a model to describe the data structure's behavior and quantify the expected performance gains.

### 4.6.1 R-Tree qualities for the SHT

In early results, we suggested considering the SHT as a two dimensional data structure, with the time and attribute dimensions. However, indexing the key values provided only minor gains on search performance, compared to more optimized R-Trees in particular. Queries on R+-Tree structures [37] search through as many nodes as the tree is deep. This was due to a sub-optimal organization of intervals in the nodes : the average node covered a wide range of attributes. As the MBR overlap remained high (Figure 4.9), single queries were only slightly narrowed down.

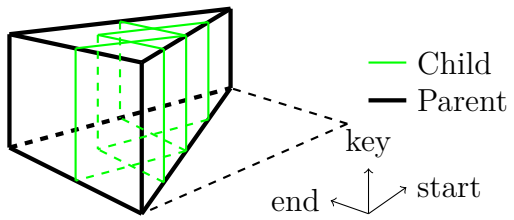



Figure 4.9 Default oSHT Split

To improve the query performance we want to provide the properties of an optimal R-Tree to the SHT. However, [1] has shown that Guttman R-Trees had worse insertion performance than the SHT, due to chronic re-balancing and was ill suited to large traces. Indeed, frequent re-balancing required in memory processing for performance reasons, while large traces would not fit in main memory.

Therefore we propose a R/SHT hybrid using Small-Tree-Large-Tree and Bulk Loading techniques. We consider using the eSHT structure for the upper nodes and buffering shorter intervals before inserting them into R-Trees in the lower level nodes.

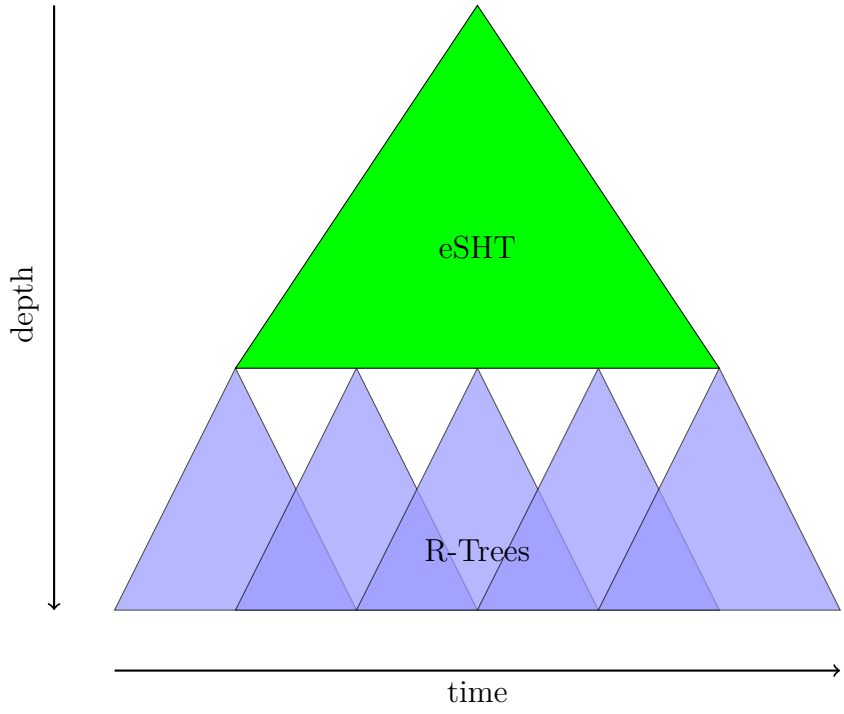



Figure 4.10 R-SHT hybrid structure

### 4.6.2 Build Algorithm

The build algorithm works similarly to the eSHT, with intervals with earlier start times inserted into shallower nodes. However, intervals starting later than the deepest SHT node go into a temporary buffer. This buffer has the capacity of an eSHT of the same depth, so that the intervals can be mapped to eSHT nodes once built.

**Algorithm 4.2** Top Down Buffer to R-SHT copy algorithm

```
1 function rBuild(intervals, node, height)
2     if node = LEAF then
3         node.insert(intervals)
4     else
5         node.insert(intervals.outliers())
6         for cluster ∈ intervals.cluster() do
7             rBuild(cluster, new child(), height − 1)
```

Algorithm 4.2 presents the algorithm that we use to move intervals from the R-Buffer into the eSHT when the buffer is full. This algorithm works top down, from the R-Buffer's root node level, to the leaf node level, and allows for additional flexibility by choosing the most relevant **outliers** and **cluster** methods.

### 4.6.3 Clustering Algorithm

As explained in [43], the choice of the **outliers** and **clustering** functions in Algorithm 4.2 plays an important role in determining in which node the intervals will be inserted.

We consider the intervals from Figure 4.6. There are at most $\frac{n}{\delta q}$ intervals per attribute in a node, with $\delta q$ the node's attribute range and $n$ the number of intervals per node. The time span $\delta t$ of node's MBR is the sum of complete intervals' durations and offsets between attributes:

$$\delta t = \frac{n}{\delta q}\frac{T}{I} + \delta q \times \frac{T}{AI}$$

With $T$ the total trace duration, $A$ the total number of attributes and $I$ the number of intervals per attribute.

We want to minimize the nodes' area:

$$\min(\delta q \times \delta t) = \min\left(\left(n + \frac{\delta q^2}{A}\right) \times \frac{T}{I}\right)$$

Which is equivalent to minimizing $\delta q$.

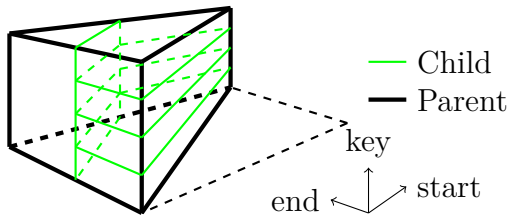



Figure 4.11 Splitting the R-Tree Buffer along the key dimension

Therefore, the splitting which minimizes the nodes' area minimizes the range along the query dimension. We move longest intervals into the parent node as outliers, since they are not accounted for by the model and naturally go into the shallower nodes in SHTs and following implementations. The remaining intervals are sorted by their key, and consecutive sub-ranges are mapped to sub-trees.

Figure 4.11 shows the split performed by the key sort algorithm, with the green plane separating the longest intervals, which go into the parent node, from the shorter intervals, which go into the children. Finally, the green planes show the split along the key dimension.

### 4.6.4 2D Queries

When we need to extract data from the tree for multiple timestamps and multiple attributes at those timestamps, the current solution was to use single or full queries. Such approaches respectively mean multiple requests or a request that returns unneeded intervals. We thus introduce 2D queries to make extracting data from the tree faster. These queries take as arguments a list of keys and either a time range or a list of timestamps, depending on the requirement. They return a map, where each queried key's associated value is the ordered lists of intervals matching the query:

```
Map(key -> intervalList) 2Dquery(keyList, timeList)
Map(key -> intervalList) 2Dquery(keyList, start, end)
```

The query is executed recursively from the root node. The time and key lists are narrowed down from the parent's to the child's bounds by using binary searches. Sub-trees which bounds do not contain any element from the time and key lists are not searched. The outcome of these queries is to execute a single search of the tree that returns all of the required intervals and only the required intervals.

## 4.7 Performance

In this section we look at how much faster we can make the single queries, full queries and 2D queries with the R-SHT compared to a typical eSHT or SHT. We use the intervals model from Figure 4.6, stored in an R-SHT with a R-Tree section of depth $r$, up to $c$ children per node and $n$ intervals per node.

For all the models on R-SHT, determining the number of nodes searched will consist in computing how many nodes are searched in the upper / eSHT levels and how many nodes are searched in each of the R-Trees of the structure:

$$S_{R-SHT} = S_e + n_R \times S_R$$

With $S_e$, the number of nodes searched in the upper / eSHT levels of the tree, $n_R$ the number of R-Trees searched and finally $S_R$, the number of nodes searched per R-Tree.

### 4.7.1 Single queries

The number of R-Trees that are queried during a single query is the number of R-Trees that overlap the queried time. The R overlap for any time is the R-Tree's duration $D_R$ over the R offset $\delta t_R$:

$$n_R = \frac{D_R}{\delta t_R}$$

With the R-Tree's duration being the sum of all the intervals' two-by-two offsets and a full interval duration:

$$D_R = c^{r-1} n \frac{T}{AI} + \frac{T}{I}$$

And the R offset being the sum of all the intervals' two-by-two offsets:

$$\delta t_R = c^{r-1} n \frac{T}{AI}$$

Therefore $n_R$ is:

$$n_R = \frac{c^{r-1} n + A}{c^{r-1} n}$$

The number of nodes searched in the eSHT levels $S_e$ is also the same, whichever R-Tree clustering algorithm is chosen. The eSHT levels are $log_c(\frac{AI}{n}) - r$ nodes deep. We compute the overlap for a specific time for every level considering each node's duration $D_d$ to be that of it's sub-tree.

$$D_d = \min\left(c^{h-d}n\frac{T}{AI} + \frac{T}{I}, T\right)$$

And the level's offset $\delta t_d$ is the sum of the sub-tree's intervals' offsets:

$$\delta t_d = c^{h-d}n\frac{T}{AI}$$

Therefore, the number of nodes searched in the eSHT levels is the total number of nodes that overlap $t$ in the eSHT levels, or the sum of overlaps per level:

$$S_e = \sum_{d=0}^{h-r} \frac{D_d}{\delta t_d}$$

Which can be developed as:

$$S_e = 1 + \sum_{d=1}^{h-r} \frac{c^{h-d}n + A}{c^{h-d}n}$$

That can be computed as:

$$S_e = h - r + \frac{A}{n}\frac{c(c^{-r} - c^{-h})}{c-1}$$

And is approximately equal to:

$$S_e \simeq h - r + c^{-r}\frac{A}{n}$$

For single queries on an R-SHT built with the key-sort algorithm, the intervals have been sorted by their key, so we know into which R-Tree branch to search, as there are fewer intervals per attribute in the R-Tree than can fit into a node.

$$S_R = \left\lfloor r + \frac{c^{r-1}}{A} \right\rfloor$$

Therefore, the total number of nodes that must be searched in the full tree is:

$$S_{R-SHT} = h - r + c^{-r-1}\frac{A}{n} + \frac{c^{r-1}n + A}{c^{r-1}n}\left\lfloor r + \frac{c^{r-1}}{A}\right\rfloor$$

Which approximates to:

$$S_{R-SHT} \simeq h + \frac{A(1+r)}{c^{r-1}n}$$

This is faster than on the SHT for which the number of nodes to be searched is $\frac{A}{n}$, as the R-SHT's height $h$ is orders of magnitude smaller than $\frac{A}{n}$ and $c >> r$

### 4.7.2 2D Query

There are too many argument combinations to compute the average complexity of the 2D queries introduced in section 4.6.4. We can however try to determine the number of nodes searched in the worst case scenario.

The main advantage of the 2D query is that even for long lists of queried keys and times, it will never search a node more than once. Therefore, its upper bound is the number of nodes in the tree:

$$bound = \frac{AI}{n}$$

When the key and time lists are very sparse, e.g. with respectively $k$ and $t$ items, the worst case is similar to returning each interval with a single query $SQ$:

$$sparse_bound = k \times t \times SQ$$

We modeled the 2D queries' performance to be faster than that of the equivalent single or full queries to extract the equivalent information.

### 4.7.3 Full queries

Full queries return the states of all the intervals intersecting a specified timestamp $t$, therefore the $S_{eSHT}$ and $n_R$ values for these queries are the same as for single queries.

However, on the key-sort R-SHT, the $S_R$ value becomes equal to the number of nodes in the R-Tree levels. Indeed, by sorting intervals by keys, we end up spreading intervals overlapping $t$ over the entire sub-tree.

$$S_R = c^r$$

Therefore, we see that full queries on an R-SHT will be noticeably longer than on a SHT:

$$S_{R-SHT} = h - r + c^{-r}\frac{A}{n} + \frac{c^{r-1}n + A}{c^{r-1}n} \times c^{r-1}$$

Which approximates to:

$$S_{R-SHT} \simeq h - r + c^{r-1} + \frac{A}{n}$$

R-SHT does not aim to speed up the full queries, as they are an inefficient implementation and can easily be replaced by single or 2D queries. The previous model shows that they are indeed slower than on the SHT, which searches $\frac{A}{n}$ nodes.

### 4.7.4 Impact of the buffer size

We use the height of the R-Buffer as the degree to which the organization of the data structure will be optimized. The higher said tree, the faster the queries, at the cost of longer optimization and tree build process.

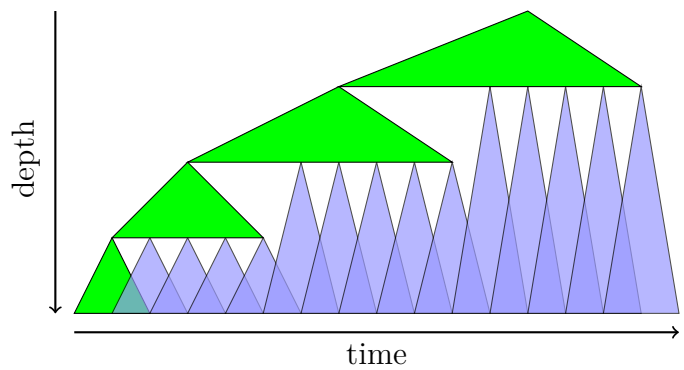



Figure 4.12 Increasing the depth of the R-Tree buffer during construction

However the required depth of the R-Buffer is linked to the key range, which is rarely known in advance. We deal with this situation by increasing the depth of the buffer during the tree construction, when the maximum key exceeds a certain threshold. As can be seen in Figure 4.12, the depth can be increased once the current R-Tree's parent is full, and a new parent needs to be initialized. This new parent will be the root node of the new, deeper R-Tree.

We can define the threshold such that consistent query performance is maintained, whichever the attribute range, once we know the theoretical query performance relative to the attribute range $A$ and R-Tree buffer's depth $r$.

Knowing that the single queries complexity is $S_{eSHT} \simeq h - r + \lceil c^{-r}\frac{A}{n}\rceil$, we must find $r$ such that we search the same number of nodes for all values of $A$. We want that:

$$S_{R-SHT}(A) \leq S_{SHT}(A_{arb})$$

With $A_{arb}$ an arbitrary number of attributes chosen to reflect satisfactory SHT performance. That can be developed:

$$h + c^{-r-1}\frac{A}{n} \leq \frac{A_{arb}}{n} + r$$

From which we compute:

$$r = \left\lceil log_c\left(\frac{A}{n}\right)\right\rceil$$

The benefits of increasing the depth of the buffer only when needed are that we no longer need to determine the depth in advance. Furthermore, trees with fewer attributes, which require less optimization, can be built faster than if the depth was hard-coded or computed otherwise.

## 4.8 Results

In this section we look into the performance results of the SHT, eSHT and R-SHT on trace analysis workloads.

### 4.8.1 Test Environment

All experiments were conducted on an Intel Core i7 3770 @ 3.4GHz with 16 GB RAM, a Samsung 850 PRO-Series 512GB Solid State Drive, using Eclipse version 4.5.1 and OpenJDK version 1.8.0_66. The trace files were generated using LTTng version 2.7.0 and the Debian Linux kernel version 4.5.0-2-amd64. We set the kernel's maximum PID value to $2^{22}$ up from its default value of $2^{15}$, so that each process will have a unique PID. The trace files contain the detailed execution trace at kernel level, including all the system calls, scheduling events and interrupts. For the following benchmarks, we traced a burn program, which creates many parallel threads, leading to many attributes in the State System.

We use `Trace Compass`' Kernel Analysis to perform our benchmarks and generate the History Trees. This analysis reads all the events from the trace and tracks various attributes by storing

them in the SHT. In particular, there are several attributes stored for each thread and CPU of the analyzed system. This means that the bigger the trace is, and the more activity there was on the system during the trace, the bigger the generated SHT will be. This analysis will thus allow us to perform a thorough comparison between the original SHT (labelled `SHT` in the following experiments), the enhanced SHT (labelled `eSHT`) and R-SHT metrics. We will compare R-SHTs of heights from 2 to 4 (labelled `R2`, `R3` and `R4`), as well as the variable height R-SHT (labelled `vR`), with maximum height of 4.

For our scalability benchmarks, we generate the traces in table 4.1, to demonstrate the scalability of our solution to traces with many threads for example.

Table 4.1 Scalability trace set specifications

| # threads | # events | size |
|---:|---:|---:|
| 160 | 7266 | 600K |
| 336 | 13133 | 772K |
| 736 | 19979 | 1004K |
| 1600 | 27722 | 1.3M |
| 3440 | 57721 | 2.3M |
| 7424 | 122552 | 4.4M |
| 16000 | 273666 | 8.9M |
| 34464 | 585565 | 19M |
| 74256 | 1263257 | 40M |
| 160000 | 2721160 | 85M |
| 344704 | 5917942 | 184M |
| 742640 | 12619069 | 390M |
| 1600000 | 27159383 | 839M |
| 3447088 | 58640566 | 1.8G |

### 4.8.2 Case Study: Control Flow View

We look into how the R-SHT can be exploited to make Trace Compass faster. In particular, Time Graph views, such as the one in Figure 4.13, which show the states of a list of attributes through time, require either many queries to obtain the required intervals or more complex ones that cover a larger time range.

We consider the `many-threads` trace from the Trace Compass test package. It is a 7.73 MiB trace from LTTng 2.8 with 240644 events. The resulting Kernel Analysis generates 741492 intervals, for 50598 attributes with a raw size of 18.62 MiB. We compare the build times, depth and file size for each implementation of the History Tree in Table 4.2. As the R-SHTs

Figure 4.13 The Control Flow View, a Time Graph View in Trace Compass

Table 4.2 Comparison of the History Trees for the `many-threads` trace

| | Build (ms) | | | Depth | Size (MiB) |
|---|---|---|---|---|---|
| **SHT** | 1 925 | ± | 353.1 | 45 | 86.31 |
| **oSHT** | 1 812 | ± | 332.9 | 3 | 21.87 |
| **eSHT** | 1 762 | ± | 345.4 | 3 | 21.44 |
| **R2** | 6 604 | ± | 727.9 | 3 | 20.87 |
| **R3** | 8 926 | ± | 1 485 | 3 | 20.81 |
| **vR** | 6 546 | ± | 387.4 | 3 | 21.00 |

are 3 nodes deep, we do not consider R4 as it is indistinguishable from R3. The original SHT stands out by its excessive depth and file size.

We look into two metrics related to displaying the Control Flow View: `PsTree` is the time to build the process tree on the left hand column of the Control Flow View, which requires each thread's *PPID* and *Exec_name*. `Zoom` is the total time for an automated sequence of scrolling and zooming the View, including the query to the State History Tree and the post processing of the returned states. The sequence begins by filling in the states for a completely zoomed out view (i.e. the trace timeline fills the entire width of the view), then geometrically zooms in ten times, and horizontally scrolls from the beginning to the end of the trace.

For each of those metrics, we compare three querying strategies: **Full** is the currently implemented strategy. It does a full query for every horizontal pixel. Between each full query, the returned values are used to build the `PsTree` and render the threads' statuses. **Single** does single queries for each thread's *PPID* and *Exec_name* for the `PsTree` and does single queries for the visible processes when zooming. **2D** does a query which returns all the *PPID* and *Exec_name* for the threads in one pass and another query per zoom to return a downsampled set of the visible processes' statuses.

The current implementation, which we will use as a baseline, is the SHT with full queries. The 2D query is able to extract all the required information to populate either the PsTree or the States in a single pass of the tree. Because the key dimension is defined from the eSHT version onwards, we cannot provide such results for SHT and oSHT.

As expected, the full query is the most efficient with the SHT, as that is the implementation for which said query has to search through the least number of nodes.

However, the PsTree phase requires information from every thread. These keys are spread out over the attribute range, and concern relatively few intervals. Therefore, it is better suited to the 2D queries, as Table 4.3 shows. Indeed the 2D query is bounded by the number of nodes in the tree. The full query implementation would search entire SHT branches or the full R3 tree for every horizontal pixel.

As for the zooming phase, the full query implementation is always the slowest while the single queries are slower than the 2D queries. Moreover the zooms are the fastest on R2 and vR trees. While this trace triggers R3 when built with the vR algorithm, this happens rather late and therefore does not influence the tree structure too much.

Table 4.3 Gains over SHT for displaying data in a Control Flow View. (mean ± standard deviation) on 10 executions.

| Query Type | PsTree (ms) | | | Zoom (ms) | | |
|---|---|---|---|---|---|---|
| **SHT** | | | | | | |
| *full* | 5 360 | ± | 645.9 | 29 880 | ± | 547.6 |
| *single* | 21 210 | ± | 860.4 | 15 300 | ± | 314.1 |
| **oSHT** | | | | | | |
| *full* | 6 536 | ± | 849.4 | 26 860 | ± | 715.8 |
| *single* | 19 500 | ± | 588.1 | 15 000 | ± | 143.3 |
| **eSHT** | | | | | | |
| *full* | 6 168 | ± | 396.0 | 30 310 | ± | 1 432 |
| *single* | 940.6 | ± | 80.41 | 6 894 | ± | 122.6 |
| *2D* | 226.8 | ± | 25.61 | 4 408 | ± | 174.9 |
| **R2** | | | | | | |
| *full* | 19 220 | ± | 4 143 | 42 310 | ± | 487.3 |
| *single* | 1 882 | ± | 43.35 | 7 861 | ± | 123.1 |
| *2D* | 136.3 | ± | 28.57 | 3 950 | ± | 183.1 |
| **R3** | | | | | | |
| *full* | 65 590 | ± | 11 730 | 267 000 | ± | 9 780 |
| *single* | 914.9 | ± | 48.34 | 6 995 | ± | 168.9 |
| *2D* | 118.4 | ± | 53.09 | 4 422 | ± | 870.4 |
| **vR** | | | | | | |
| *full* | 17 100 | ± | 7 261 | 40 970 | ± | 2 878 |
| *single* | 1 969 | ± | 55.73 | 8 183 | ± | 261.0 |
| *2D* | 135.7 | ± | 58.38 | 4 041 | ± | 170.7 |

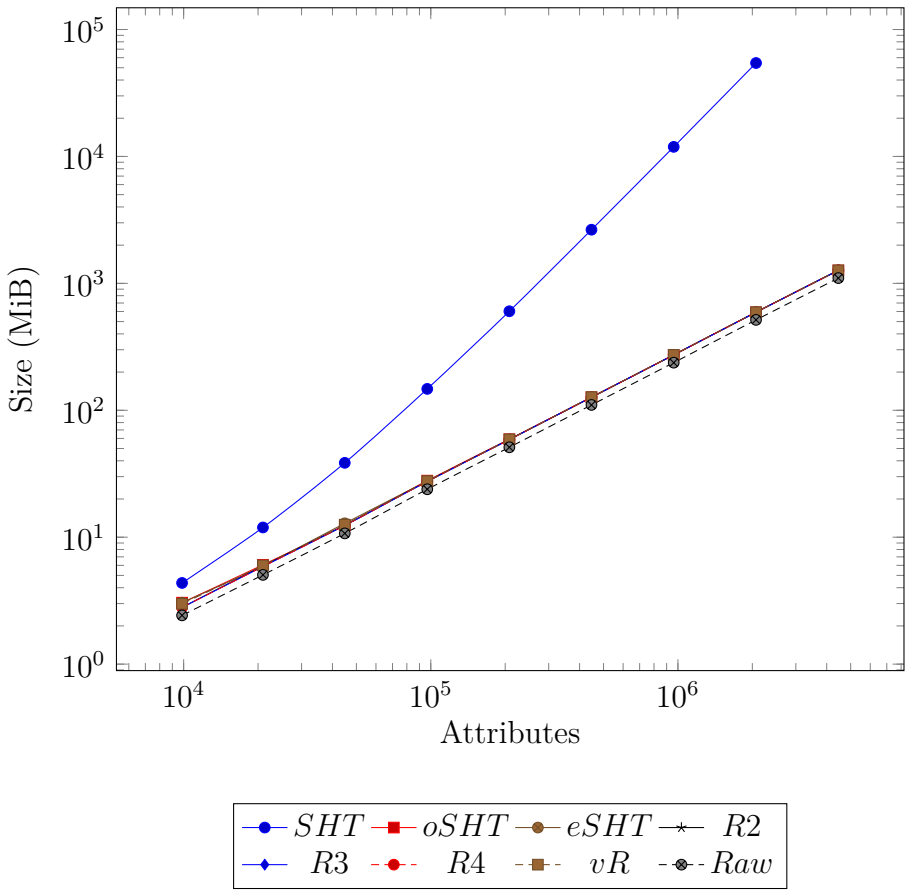



Figure 4.14 Comparison of external memory usage for different SHT variants

### 4.8.3 Storage usage

We see in Figure 4.14 that the size for the classic SHT would be proportional to the square of the number of attributes. Sizes for the eSHT and R-SHTs are aligned with the raw size of the intervals that are being stored, with a slight overhead for the header information. Therefore, the data structure no longer wastes space when stored in external memory. Because of the SHT's inefficient serialization, we could not fit it in external memory for traces producing more than $2.1 \times 10^6$ attributes, whereas the other implementations executed properly until $4.5 \times 10^6$.

### 4.8.4 Tree Depth

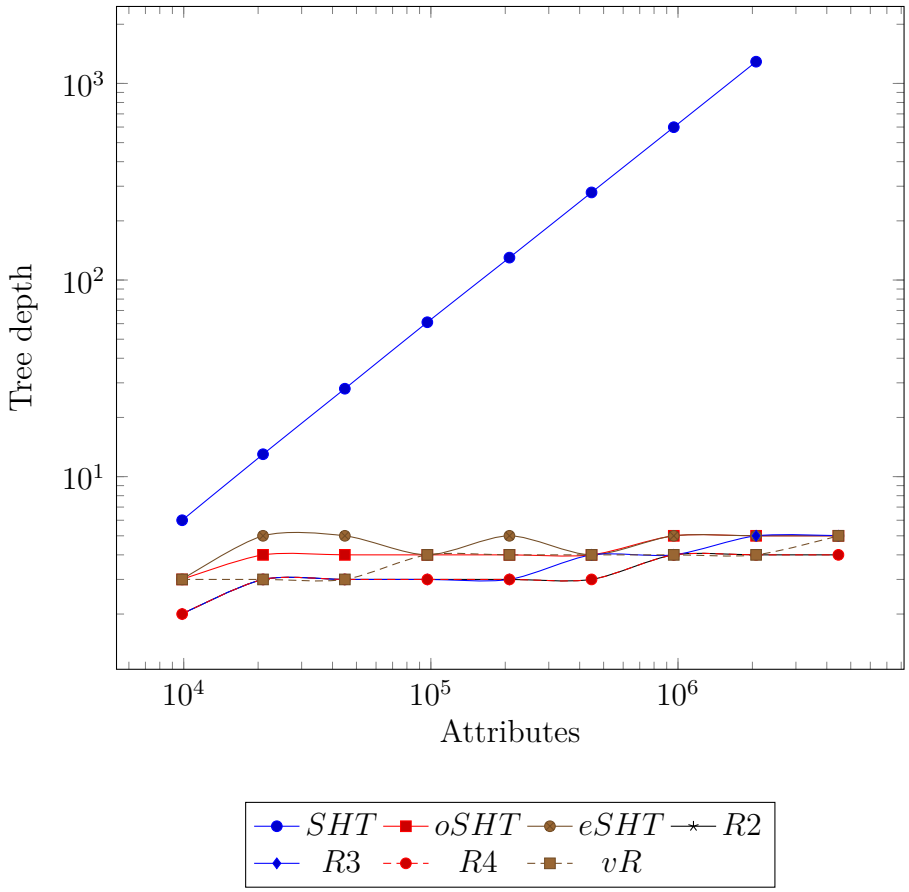



Figure 4.15 Comparison of tree depth with different SHT variants

We see in Figure 4.15 that the depth of the classic SHT would be proportional to the number of attributes. Depths for the eSHT and R-SHTs are closer to those of packed and balanced trees. As the latest branch is kept in memory before being serialized, this results in substantial

memory savings during the build process. Moreover, we no longer have long empty branches as we had in Figure 4.3.

### 4.8.5 Single Query

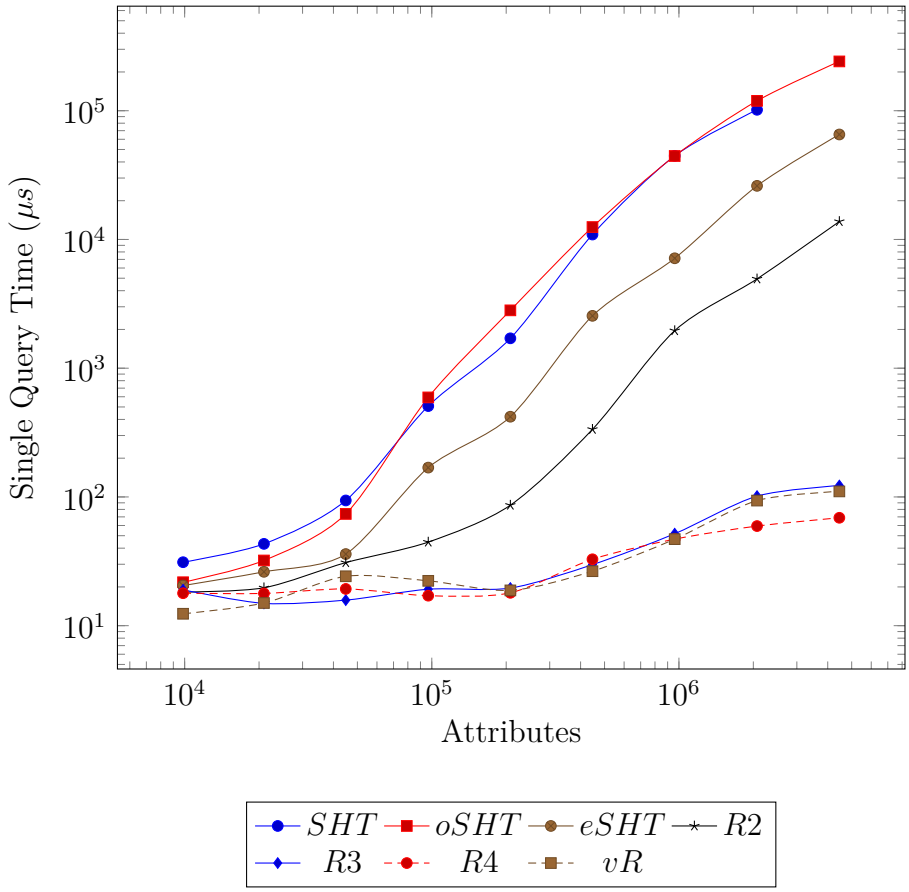



Figure 4.16 Comparison of single query performance with different SHT variants

While the eSHT does not make queries significantly faster, Figure 4.16 shows how the reduction of node overlap along with increasing the R-Tree buffer depth reduces the single query search time.

Moreover, the depth increase of the R-buffer with the variable depth implementation is clearly visible, with query times similar to $R = 2$ until $Attributes \approx 2 \times 10^6$ then close to $R = 3$ for $Attributes = 4 \times 10^6$ and on.

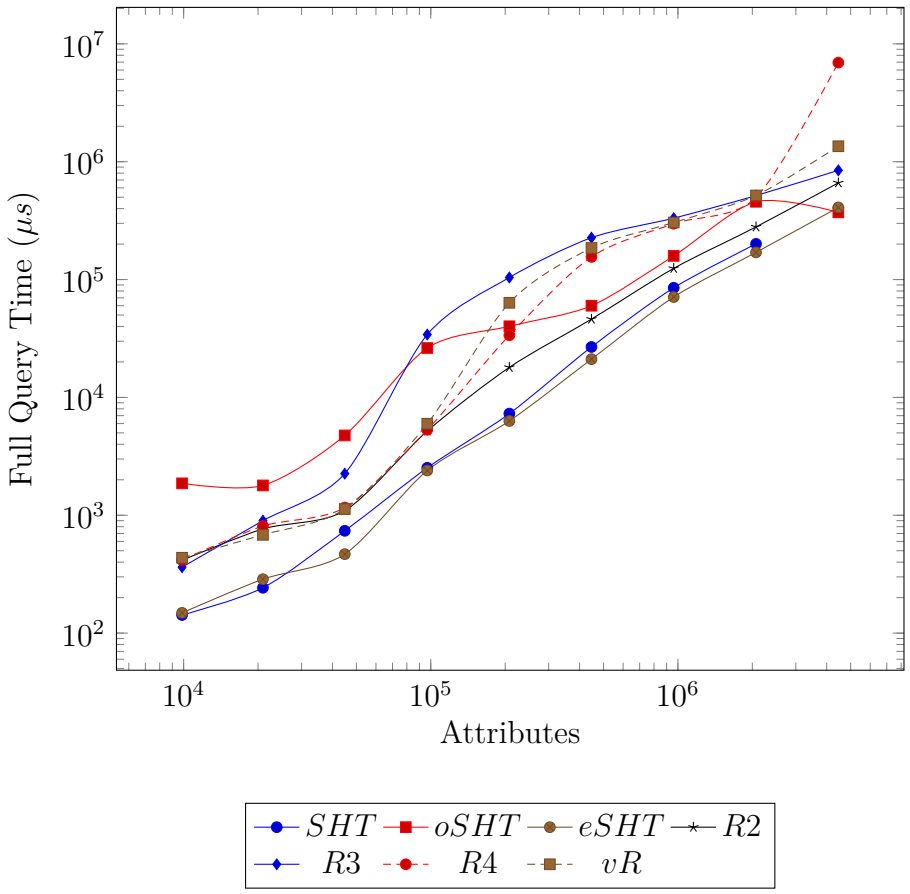



Figure 4.17 Comparison of full query performance with different SHT variants

### 4.8.6 Full Query

In Figure 4.17 , the move from SHT to eSHT does not impact the full query performance, as the same number of nodes remains to be searched. Meanwhile, the effect of the R-SHT, grouping intervals with the same attributes, at the cost of similar time range groupings, significantly reduces the full query performance.

### 4.8.7 2D Query

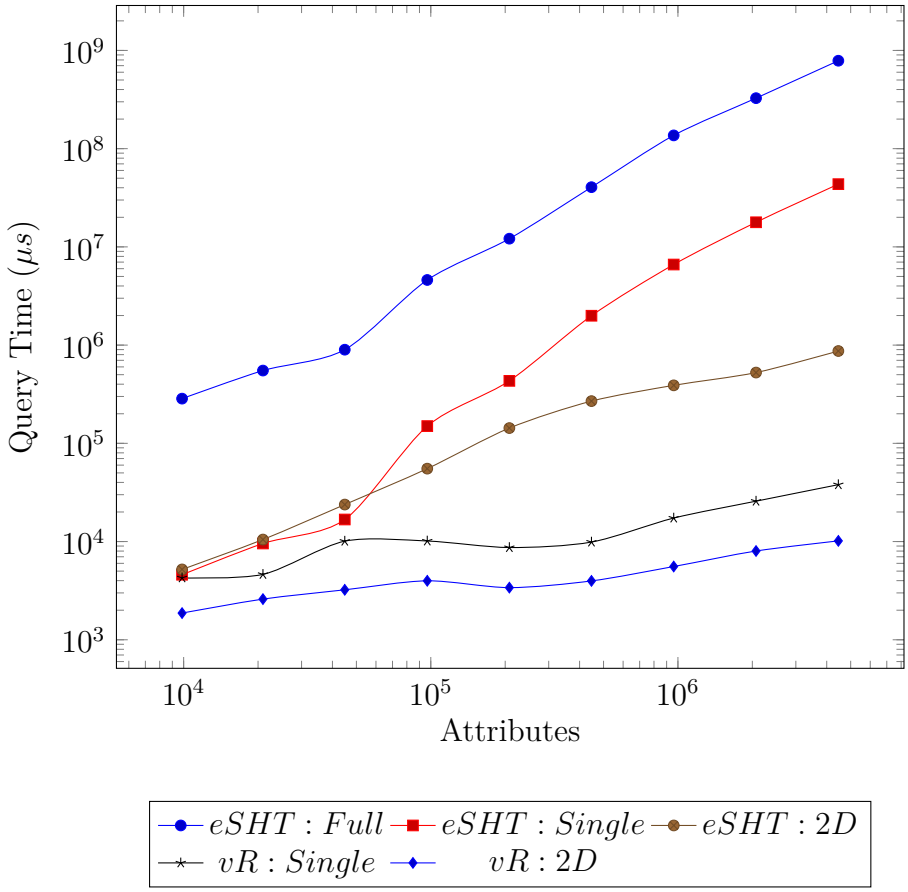



Figure 4.18 Comparison of full, single and 2D queries to extract the same data on eSHT and vR-SHT

We benchmarked the 2D query on its ability to extract a sparse set of 100 keys on a set of 2000 timestamps. We compare how much time it would have taken to query the same data with single or full queries. We do not represent the full queries on the vR tree, as we saw in the previous section that they are slower than on the eSHT.

We see in Figure 4.18 that the 2D query approach is faster than repeated single queries on both eSHT and vR trees. Moreover, both types of queries are faster on the vR tree.

### 4.8.8 Build Times

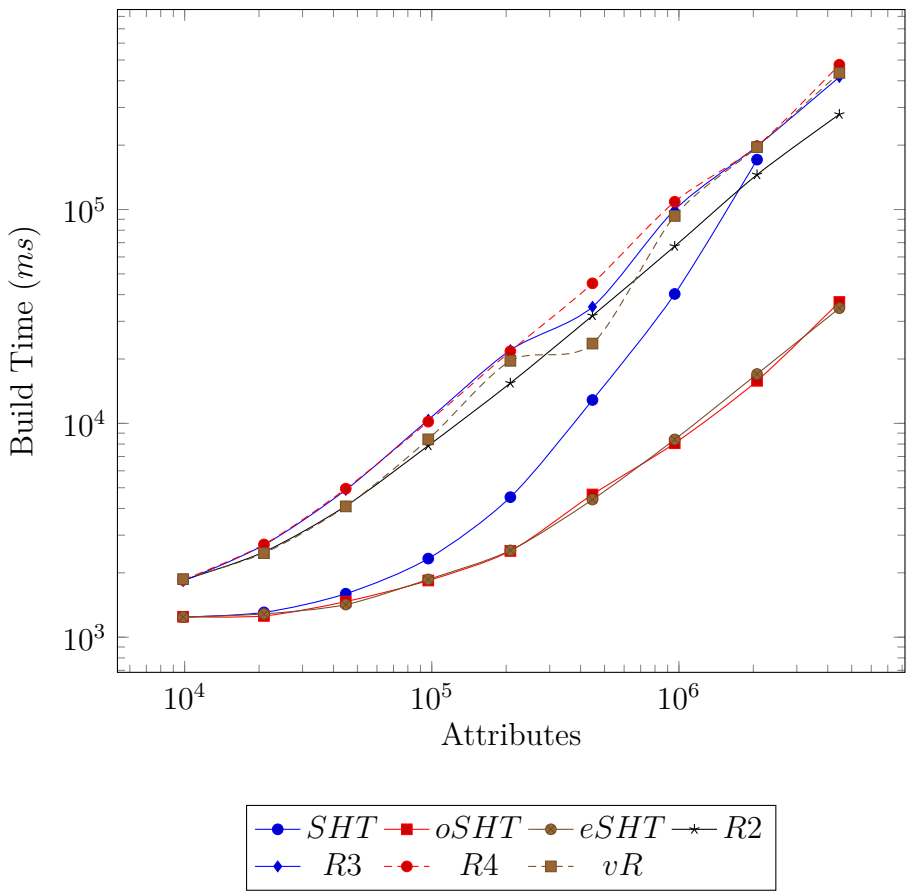



Figure 4.19 Comparison of build time with different SHT variants.

As the move from SHT to eSHT strongly reduces the number of nodes that need to be created and written to external media, the eSHT build is much faster than the R-SHT's, as can be seen in Figure 4.19. However, the R-SHT optimizations resort to sorting a list of intervals as long as the R-Tree buffer's capacity, which increase the build time.

## 4.9 Conclusion

In this paper, we have presented two new evolutions of the State History Tree (SHT) data structure.

The first variant, the enhanced State History Tree (eSHT), addresses the issue of scalability when systems with many threads are analyzed. It focuses on near-optimal storage usage and tree depth, both of which impact the build time of the tree.

The second variant, R-SHT, focuses on query performance, which the shift from SHT to eSHT

did not significantly improve. R-SHT uses R-Tree techniques to more efficiently organize the data in the tree structure, so that it is faster to retrieve. Moreover, we suggest a data-driven method to adjust the level of optimization so that query times remain below a threshold, regardless of the number of attributes.

We modeled the data structures' behaviors, showing the gains on query performance with different levels of optimization. We put the new implementations to trial by analysing highly parallel traces to show the benefits on query performance. Finally, we studied the implementation of the data structure in the Trace Visualization and Analysis software Trace Compass, finding significant gains (at least 7×) in the time it took to retrieve and render complex views.

We find that the R-SHT is up to three orders of magnitude faster for single queries and 2D queries than the SHT and eSHT for traces with more than one million attributes. However, it is less efficient on full queries, but we showed that using full queries to populate views is less efficient than the others.

We believe that these new structures, especially the R-SHT, will enable the analysis and visualization of traces to scale to highly parallel systems with millions of threads or cores. Moreover, vR solves the problem of determining the optimization level, by changing it on the fly according to the data properties.

Future work could use Mip-Mapping techniques to store a summary of the more detailed information stored in the structure, as to reduce the computation required at less zoomed-in levels.

**Acknowledgement**

The support of the Natural Sciences and Engineering Research Council of Canada (NSERC), Prompt, Ericsson and EfficiOS is gratefully acknowledged.

# CHAPITRE 5 RÉSULTATS COMPLÉMENTAIRES

Dans la section précédente, nous avons présenté une approche permettant d'améliorer le choix de noeud pour insérer des intervalles dans le SHT. Cependant, nous avons aussi conduit d'autres expériences dont les résultats n'étaient pas aussi probants ou reliés au contenu de l'article précédent. Dans cette section, nous traitons des résultats secondaires que nous avons découvert au cours de nos travaux. Nous verrons comment les analyses peuvent être optimisées pour réduire leur impact sur les ressources. Ensuite nous résumerons les résultats d'approches initialement considérées pour réduire le volume de données à stocker dans l'arbre à historique d'états. Nous terminerons en montrant comment nous avons pu exploiter les résultats sur l'arbre à historique d'états pour une autre base de données temporelle de notre logiciel.

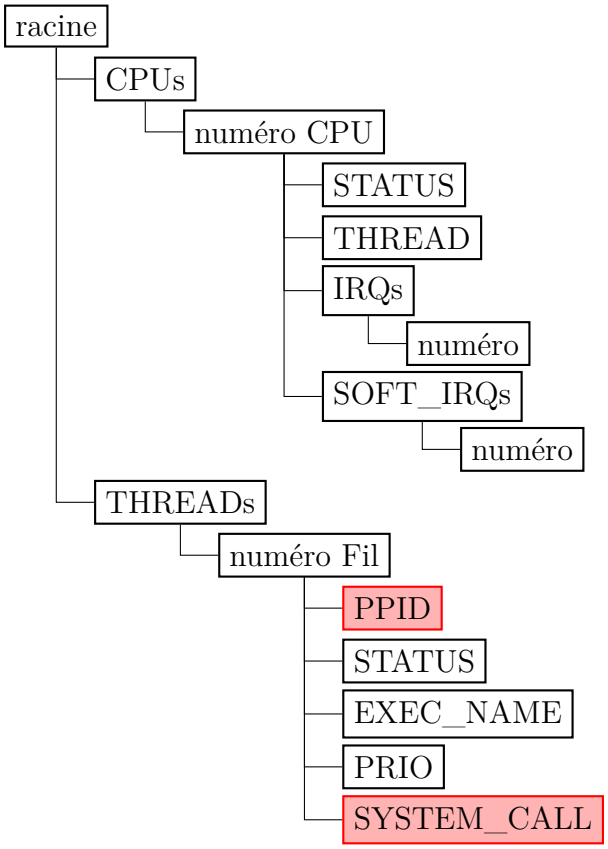
## 5.1 Optimisation des analyses

Certaines analyses, particulièrement l'analyse du noyau qui est exécutée quasi systématiquement, produisent des arbres de taille démesurée. L'analyse du noyau produisait l'arbre représenté à la figure 5.1(a).

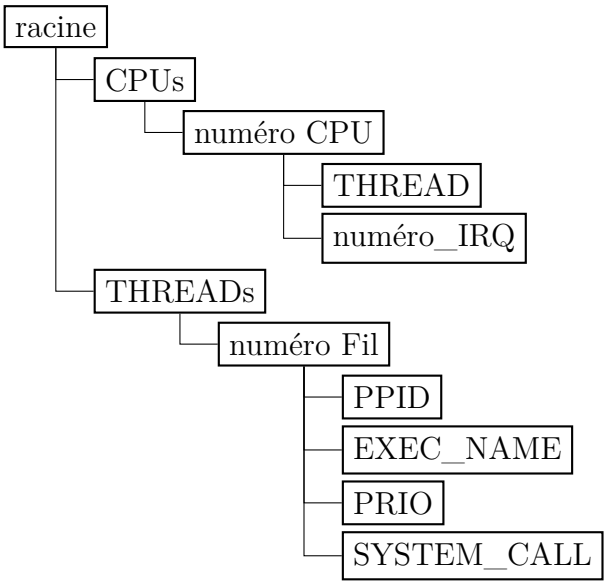
Nous nous sommes rendu compte que l'analyse produisait de nombreux intervalles porteurs de valeur nulle et de la longueur entière de la trace. Cela est problématique, car nous avions une entrée inutilisée – valeur nulle – qui faisait dégénérer l'arbre en profondeur. Après investigation, nous avons décelé 2 problèmes : le premier était que chaque noeud parent (pour un processeur ou un processus léger par exemple) créait un attribut, mais ne stockait pas de valeur. Pour résoudre ce problème, nous pouvions soit réécrire l'arbre entier pour annoter certains attributs comme étant des "dossiers", n'ayant pas d'intervalles, mais seulement des sous-attributs, cela impliquant de réécrire toutes les autres analyses. L'autre option était d'utiliser le noeud racine pour stocker un des sous attributs. Cette option était de loin la plus simple, nous avons utilisé la racine pour stocker l'attribut **STATUS** des processeurs et processus légers.

Cette modification permettait de stocker un attribut de moins par fil ou Unité Centrale de Traitement (UCT) et donc de réduire d'autant la profondeur du SHT classique. Sur les SHT avec superposition, les gains sont moindres, car on ne retire au final qu'un intervalle pour chaque attribut gagné.

Le second problème était que les attributs **Parent Process Id (PPID)** et **SYSTEM_CALL**




(a) L'arbre à attributs de l'analyse noyau avant optimisation




(b) L'arbre à attributs de l'analyse noyau après optimisation

Figure 5.1 Comparaison de l'arbre à attributs de l'analyse noyau après et avant optimisation

étaient créés systématiquement pour chaque processus léger, que l'on ait l'information ou non. En l'absence d'information pour l'attribut, c'est encore un intervalle porteur de la valeur nulle et de la longueur entière de la trace qui est inséré dans l'arbre. Ici la solution fut de créer l'attribut seulement lorsque la valeur est connue, et de traiter correctement son absence dans les consommateurs de données.

Le troisième problème était que pour chaque UCT, nous avions un sous-attribut pour les interruptions matérielles (**IRQs**) et un pour les interruptions logicielles (**SOFT_IRQs**), chacun avec pour enfant les attributs créés à la demande pour les interruptions. Cela engendre donc 2 attributs inutiles par UCT, nous optons pour supprimer les attributs **IRQs** et **SOFT_IRQs** afin que chaque interruption soit un enfant direct d'une UCT, et encodons les interruptions logicielles en changeant leur signe pour éviter des conflits avec des interruptions matérielles qui auraient potentiellement le même numéro.

Ces trois opérations permirent de réduire de 25% à 40% le nombre moyen d'attributs par

arbre. L'arbre amélioré est visible à la figure 5.1(b).

## 5.2 Déduplication des chaînes de caractères

Au cours de nos diverses expériences, nous avons constaté qu'une grande partie de l'espace disque du SHT était consommé par les intervalles portant des chaînes de caractères, ce qui est naturel, car chaque chaîne peut occuper plus de place qu'un entier, même sur 64 bits. Toutefois, nous avons aussi constaté que dans de nombreux cas, un grand nombre de chaînes étaient dupliquées, par exemple pour des fils ayant le même nom, ou encore pour les appels systèmes, dont les noms entiers étaient stockés à chaque occurrence.

Pour essayer de réduire la duplication dans l'arbre à historique d'états, nous nous sommes inspirés de l'internalisation des chaînes en Java [61]. La pile Java contient une zone de mémoire en laquelle sont stockées les chaînes de caractères. Si on demande la création d'une chaîne de caractères, mais que celle-ci existe déjà, la copie précédente sera référencée au lieu d'allouer de l'espace pour une nouvelle chaîne. Cela est possible, car les chaînes sont immuables.

Nous reprenons ce principe lors de la construction de l'arbre à historique d'états. Nous enregistrerons sur disque seulement la première occurrence d'une chaîne de caractères, les suivantes référenceront cette occurrence par le numéro du noeud et l'indice dans le noeud de l'intervalle ayant cette chaîne.

Pour ce faire, lors de la construction de l'arbre, nous utilisons une table associant une chaîne de caractère à la référence du noeud la contenant. Pour chaque intervalle porteur d'une chaîne de caractères inséré dans l'arbre, nous vérifions si la chaîne est dans la table. Si c'est le cas, nous remplaçons la chaîne par la référence. Le cas échéant, nous ajoutons l'intervalle à l'arbre et ajoutons la référence de l'insertion à la table d'associativité.

Lors des requêtes, si l'intervalle retourné contient une référence et non la chaîne, il suffit d'aller lire le noeud référencé et d'aller chercher la chaîne à l'intervalle indexé pour remplacer la référence par la chaîne correcte. Nous utilisons un cache par association directe pour éviter de relire fréquemment les chaînes associées sur disque.

Nous constatons effectivement un gain de place dû à la déduplication des chaînes de caractère à la figure 5.1. Comme la taille de l'arbre est réduite, les requêtes sont aussi accélérées, tant que nous n'évaluons pas la valeur de la référence. Au contraire lorsque nous évaluons la valeur de la référence, les requêtes simples sont du même ordre de grandeur qu'avant déduplication tandis que l'évaluation des valeurs de tous les états de la requête pleine ralentit cette dernière de manière non négligeable.

De plus les requêtes étaient systématiquement deux fois plus lentes, rendant moins inté-

Tableau 5.1 Gains suite à la déduplication des chaînes de caractères. Trace `ManyThreads` du jeu de traces de test : git.eclipse.org/gitroot/tracecompass/tracecompass-test-traces.git .

| | Avant | | | Après | | |
|---|---|---|---|---|---|---|
| Taille (Mo) | 81.24 | | | 74.49 | | |
| Requête unique avec évaluation (ms) | 3.00 | ± | 0.17 | 2.67 | ± | 0.07 |
| Requête unique sans évaluation (ms) | 3.00 | ± | 0.17 | 2.74 | ± | 0.10 |
| Requête pleine avec évaluation (ms) | 7.01 | ± | 0.17 | 11.50 | ± | 0.15 |
| Requête pleine sans évaluation (ms) | 7.01 | ± | 0.17 | 6.70 | ± | 0.07 |

ressante la déduplication, quelques mégaoctets étant un faible coût à payer pour accélérer grandement les recherches.

## 5.3 Retirer les intervalles "nuls"

Avec l'implémentation actuelle du SHT, chaque attribut doit avoir un état, du temps de début de la trace $t_{début}$, jusqu'à sa fin. Cela veut dire qu'il aura des intervalles contigus sur toute la durée. Or, au début de l'analyse, avant le premier changement d'état d'un attribut au temps $t$, son état est inconnu, un intervalle portant une valeur "nulle" de $t_{début}$ à $t$ est donc stocké dans l'arbre à historique d'états. Et si l'on pouvait supprimer ces intervalles pour gagner de la place sur disque et réduire la dégénérescence de l'arbre par l'insertion d'intervalles de longue durée ? En l'absence de ces intervalles, trouver le premier état non nul d'un attribut exigerait de chercher tous les noeuds de l'arbre par temps croissant, alors que précédemment une requête à $t_{début}$ retournait directement le temps auquel il fallait chercher, l'information reste pertinente, existe-t-il un autre moyen de la stocker ?

Nous expérimentons avec ajouter le temps de début $t_{d,a}$ et le temps de fin $t_{f,a}$ aux attributs, de manière à remplacer l'information précédemment stockée par les intervalles. Nous comparons ensuite la performance de l'arbre à historique d'état après et avant la suppression de ces intervalles. Les résultats de ces expériences sont visibles dans le tableau 5.2. Nous constatons des gains considérables avec le SHT classique, pour lequel retirer de longs intervalles a un gros impact. Toutefois, sur les arbres avec superposition, qui sont plus résilients, les gains sont équivalents à retirer un petit nombre d'intervalles.

Tableau 5.2 Gains suite à la suppression des premiers et derniers intervalles nuls.

| Type | SHT | | | eSHT | | |
|---|---|---|---|---|---|---|
| Métrique | Avant | Après | Gain | Avant | Après | Gain |
| Construction (ms) | 17128.48 | 8244.06 | 2.07 | 6314.64 | 6154.9 | 1.02 |
| Taille (MiB) | 1257.94 | 579.32 | 2.17 | 170.44 | 168.76 | 1.01 |
| Noeuds | 20055 | 9115 | 2.20 | 2655 | 2546 | 1.04 |
| Profondeur | 182 | 116 | 1.56 | 4 | 4 | 1 |
| Requête Simple (us) | 3192 | 117.8 | 27.09 | 2959 | 1836.4 | 1.61 |
| Requête Entière (us) | 14378 | 16479.7 | 0.87 | 12154 | 15408.1 | 0.78 |
| Requête Attribut (us) | 13371 | 139.2 | 96.05 | 3455 | 471 | 7.32 |

## 5.4 L'arbre à segments

Certaines analyses utilisent une structure de données similaire à celle de l'arbre à historique d'état appelé Stock de segments (Segment Store en anglais). La principale différence est que cette dernière stocke des segments au lieu d'intervalles et que l'API est différent. Les segments peuvent être assimilés à des intervalles dépourvus de clés, c.-à-d. un temps de début, un temps de fin et une valeur : $< t_{début}, t_{fin}, valeur >$. L'API du Stock de Segments est une Collection Java à laquelle ont été rajoutés des itérateurs permettant de parcourir les intervalles recouvrant un intervalle de temps, potentiellement triés selon une dimension.

Cette dernière structure était historiquement implémentée en mémoire, mais avec l'augmentation de la taille des traces à analyser et la complexité des analyses, conserver ces résultats en mémoire principale n'était plus possible dans une empreinte mémoire raisonnable. Les implémentations en mémoire actuelles sont basées sur :

`TreeMapSegmentStore` : Avec deux arbres binaires équilibrés Java, l'un ordonné sur les temps de début, l'autre sur le temps de fin de l'intervalle référencé.

`ArrayListSegmentStore` est basée sur un tableau redimensionnable, en lequel les segments sont insérés de manière à ce que le tableau soit trié par temps de début croissant des intervalles.

`LazyArrayListSegmentStore` comme ArrayListSegmentStore, sauf que les segments sont insérés à la fin du tableau qui n'est trié que lorsqu'on en a besoin pour une requête.

Pour tester la performance des stocks de segments, nous avons des tests de performance qui mesurent le temps mis pour insérer un jeu de segments dans la structure, et le temps mis pour itérer au travers l'ensemble des segments. Nous utilisons deux distributions :

`bruitée` engendre des segments avec un temps de début qui croit de manière linéaire avec un bruit uniformément distribué et une durée constante.

`aléatoire` engendre des segments avec un temps de début pseudo-aléatoire tiré d'un ensemble uniformément distribué entre 0 et $2^{63} - 1$ et une durée constante.

### 5.4.1 Arbre à Segments par dessus des bases de données existantes

La première étape pour stocker les segments fut de faire l'état des solutions de bases de données libres pouvant s'intégrer à notre environnement. En particulier, nous avons regardé la performance de MapDB [62] une bibliothèque Java se décrivant comme à la frontière des collections et des bases de données. Nous avons implémenté l'interface Stock de Segments par-dessus celle de MapDB, afin de pouvoir comparer les performances des deux et estimer lequel était le plus pertinent.

Nous avons choisi deux implémentations de MapDB qui semblaient propices à notre usage :

BTreeMap est basé sur l'arbre B* (B*-Tree)[63], une extension de l'arbre B (B-Tree) qui permet des opérations concurrentes plus efficaces en ne verrouillant que les noeuds modifiés.

IndexTreeList est basé sur un tableau redimensionnable, associé à un arbre pour maintenir l'ordre des clés.

Dans le diagramme à bandes 5.2, nous voyons que les implémentations de MapDB en mémoire sont toujours plus lentes que les précédentes.

### 5.4.2 Arbre à Segments sur le principe de l'arbre à état

Nous avons donc cherché comment réexploiter les résultats de l'amélioration de l'arbre à historique d'états sur le stock de segments, les segments n'étant au final que des intervalles dépourvus de clés !

L'implémentation naïve consistait à reprendre l'arbre à intervalles tel quel, en le modifiant pour accepter les segments et répondre à l'API.

Le véritable défi résidait en permettre l'itération triée de manière efficace en profitant des propriétés d'indexation de l'arbre.

Voici l'algorithme d'itération non trié naïf permettant cependant de ne lire les noeuds sur disque que lorsque cela est nécessaire :

Dans la Figure 5.3, nous comparons les performances des implémentations sur des tests synthétiques. Le test comprend des insertions de segments suivant une distribution reproductible, suivi d'itérations sur les segments triés par temps de début croissants, par temps de fin croissants ou par durée croissante. Dans la distribution bruitée, les temps de début des segments

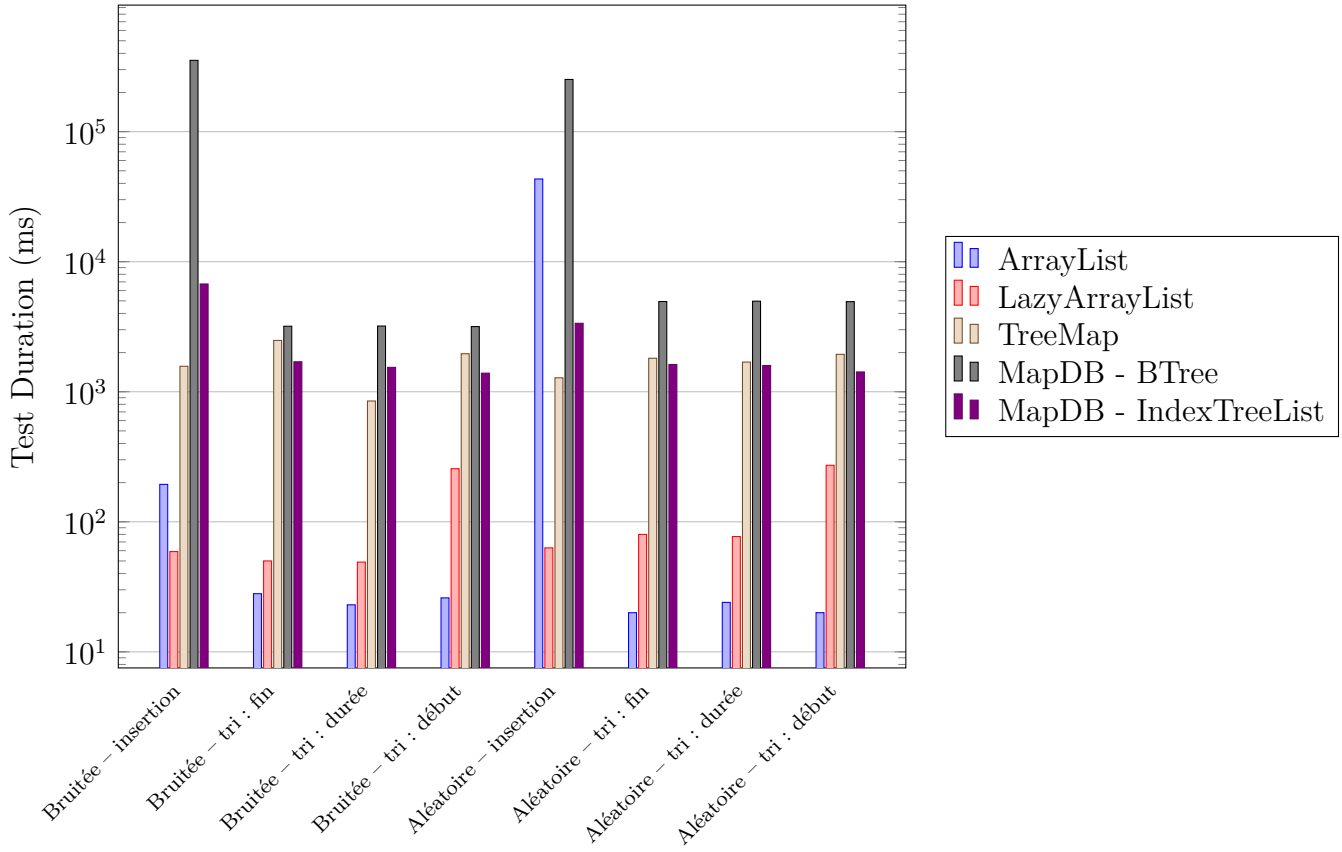



Figure 5.2 Comparaison des stocks de segments dans les tests de Trace Compass

**Algorithm 5.1** Algorithme naïf d'itération sur les segments intersectant une estampille

```
1  class iterator(time)
2      nodes ← [rootNode] ;
3      segments ← [] ;
4      function hasNext()
5          while segments = ∅ ∧ nodes ≠ ∅ do
6              node ← tree.readNode(nodes.remove()) ;
7              if node.type() = CORE then
8                  nodes.add(node.intersectingChildren(time)) ;
9              segments.add(node.intersectingSegments(time)) ;
10         return segments ≠ ∅ ;
11     function next()
12         return segments.remove() ;
```

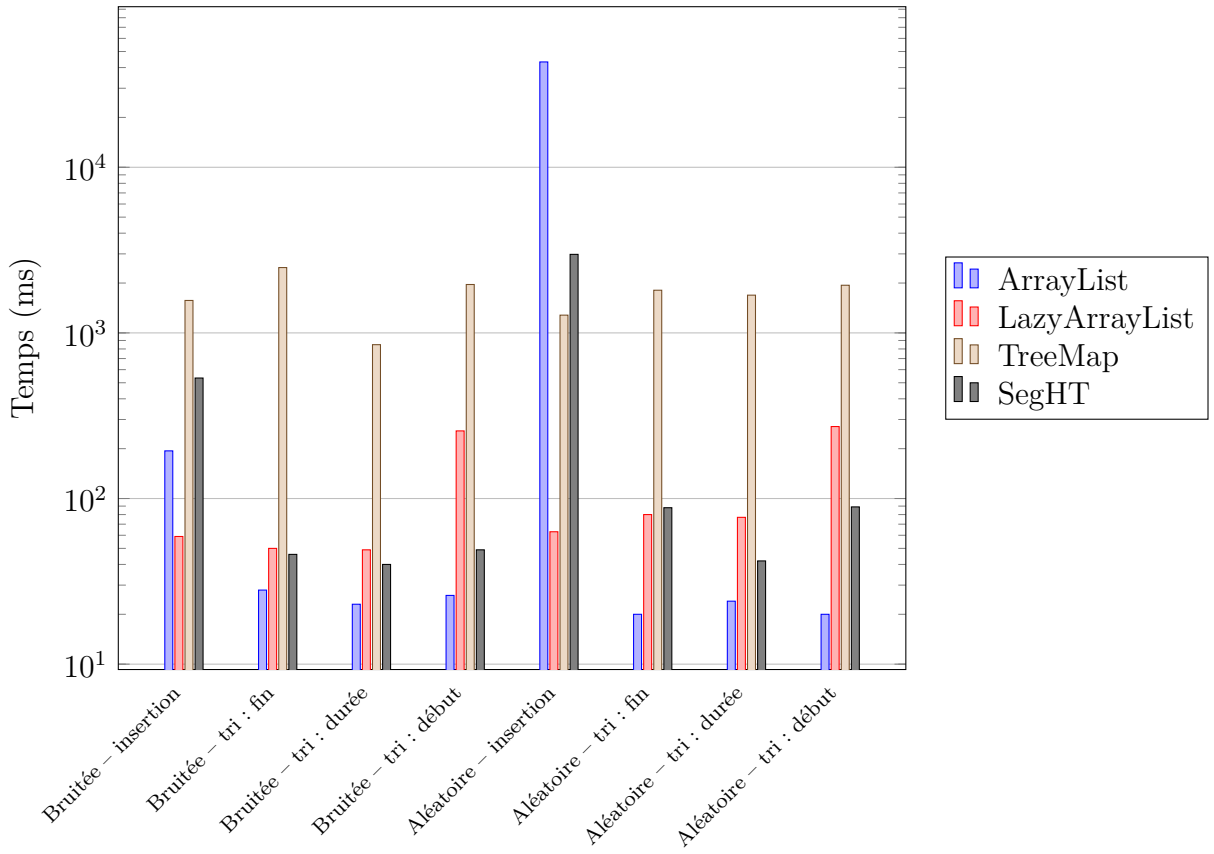



Figure 5.3 Performance de l'arbre de segments comparé aux implémentations en mémoire

résultent de bruit uniforme autour d'une droite affine. La distribution aléatoire engendre des temps de début uniformément distribués entre 0 et $2^{63} - 1$.

Dans la figure 5.3, nous voyons que les performances en requête de l'arbre à segments sont du même ordre de grandeur que celles des implémentations basées sur des ArrayList. Naturellement, les performances en insertion sont plus lentes, car les données doivent être écrites sur disque. Les performances du stock de segments basées sur TreeMap sont régulièrement parmi les plus lentes, mais cette implémentation naïve est déconseillée.

Toutefois, un des défis avec le stock de segments est qu'il autorise l'itération ordonnée selon n'importe quel ordre – défini par un Comparateur Java. Certains ordres sont communs à tous les stocks de segments : ordres sur les temps de début, temps de fin et durée des segments, comme ce sont des propriétés communes à tous les segments. D'autres ordres sont spécifiques à une classe de segments – segments ayant une valeur – et peuvent ne pas être connus à l'avance. Sur les structures en mémoire, retourner l'ensemble des segments, puis les trier en mémoire peut être une solution acceptable. Toutefois lorsque cet ensemble de segments est plus grand que l'espace mémoire, la structure de données doit pouvoir proposer une manière efficace de retourner les segments dans l'ordre désiré.

**Algorithm 5.2** Algorithme d'itération ordonnée généralisé

```
1  class iterator(order, time)
     /* Files de noeuds et segments triées par l'ordre en paramètre */
2    nodes ← PriorityQueue(rootNode);
3    segments ← PriorityQueue;
4    function hasNext()
       /* Vérifier que le premier segment de la file de segments précéde ceux
          du reste de l'arbre selon l'ordre. */
5      while (segments = ∅ ∨
        order.compare(segments.head(), nodes.head().key(order)) > 0) ∧ nodes ≠ ∅ do
6        node ← tree.readNode(nodes.remove());
7        if node.type() = CORE then
8          nodes.add(node.intersectingChildren(time));
9        segments.add(node.intersectingSegments(time));
10     return segments ≠ ∅;
11   function next()
12     return segments.remove();
```

Nous proposons l'algorithme 5.2 qui généralise l'itération triée sur les segments en utilisant le paradigme de "plane-sweeping" [46]. Nous voyons qu'il faut compléter les métadonnées contenues dans les noeuds afin d'avoir accès à une borne inférieure et supérieure pour chaque

noeud et chaque ordre. De plus la connaissance de cet algorithme nous permettra d'évaluer comment optimiser le placement des segments dans l'arbre pour itérer plus vite.

Les métadonnées supplémentaires peuvent être ajoutées aux noeuds parents – c'est déjà le cas pour les temps de début et de fin des intervalles enfants – ou dans une structure supplémentaire qui associe à chaque indice de noeud la borne pour l'itérateur demandé. Ces métadonnées peuvent être ajoutées lors de la construction pour les ordres qui sont connus d'avance ou à la demande pour les ordres qui seront déclarés après la construction de l'arbre.

La solution retenue sera hybride, avec les bornes des ordres communs à tous les segments (temps de début, fin et durée) stockées dans les noeuds parents, et les autres ordres dans une structure secondaire.

Dans la Figure 5.4, nous comparons l'utilisation mémoire des itérateurs triés, lorsque le tri est effectué en mémoire, au tri utilisant les files de priorités. Comme nous nous y attendons, profiter des propriétés d'indexation de la structure de données réduit considérablement l'empreinte mémoire par rapport à trier en mémoire principale un tableau de 100 millions de segments.

Nous vérifions ensuite que les performances augmentent réellement dans la Figure 5.5 et constatons que les temps pour itérer sur l'ensemble des segments sont du même ordre de grandeur que le tri en mémoire. Toutefois, l'itération peut commencer plus rapidement comme le montre le temps pour obtenir les 100 premiers segments, car il n'est pas nécessaire de trier la totalité des segments avant de pouvoir itérer.

### 5.4.3 Amélioration de la performance en itération de l'arbre à segments

Bien que l'organisation naturelle des segments à l'intérieur de l'arbre permette une itération relativement efficace, notre expérience avec l'arbre R-SHT nous a montré qu'il était possible de réorganiser ces derniers pour optimiser les requêtes.

Comment organiser les segments à l'intérieur de l'arbre pour maximiser la performance en lecture ? Si nous voulons optimiser l'insertion pour un ordre de tri, par exemple les temps de début, nous insérerons les segments par temps de début croissant, d'un noeud à son voisin. L'optimisation devient plus complexe lorsque l'on veut obtenir un compromis entre le temps pour bâtir l'arbre ainsi que les différents ordres de tris.

Nous adaptons l'algorithme permettant de bâtir récursivement des sous-arbres proposés dans la section 4.6 aux arbres à segments. Il nous faut ensuite choisir un algorithme de groupement des segments qui non seulement respecte les prérequis – bâtir des groupes dont la taille demeure inférieure à la capacité de stockage du sous-arbre – mais qui est aussi rapide et

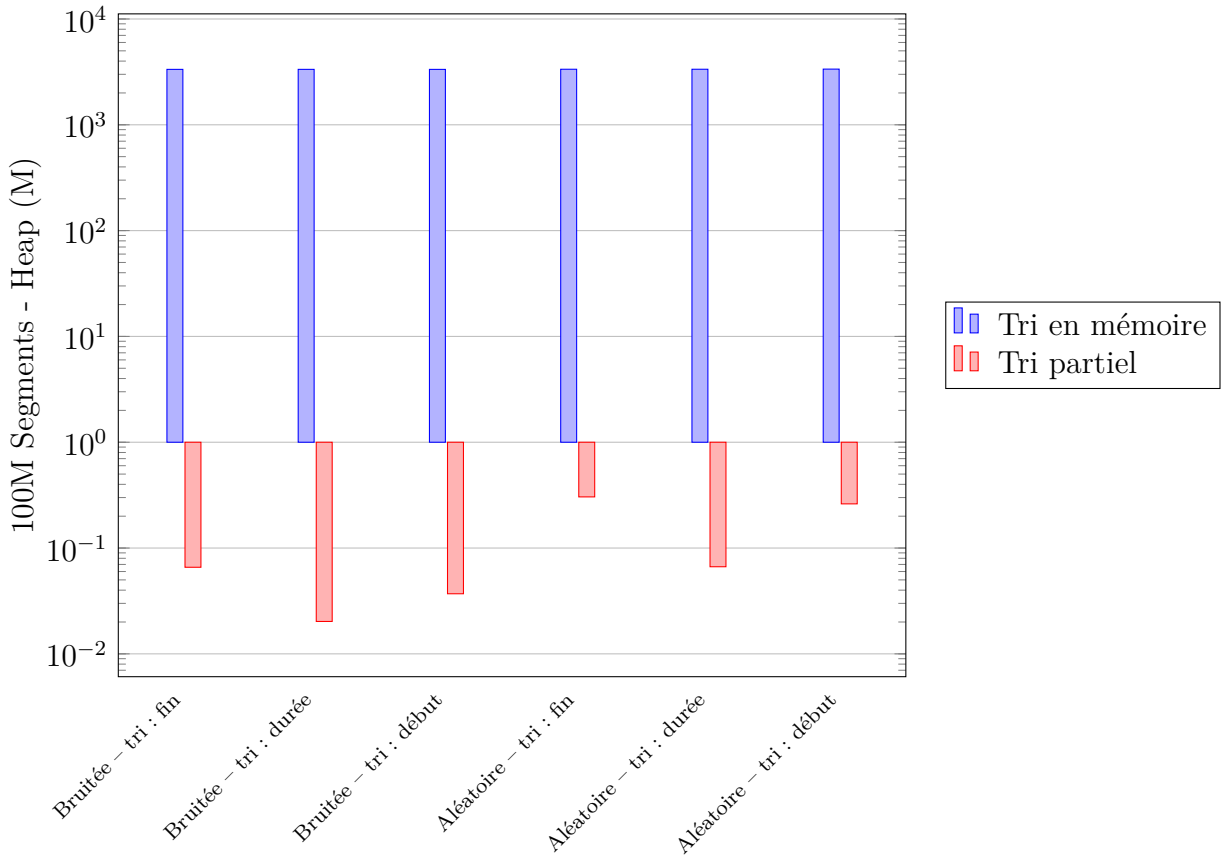



Figure 5.4 Utilisation mémoire du tri en mémoire comparée au tri de la file de priorité

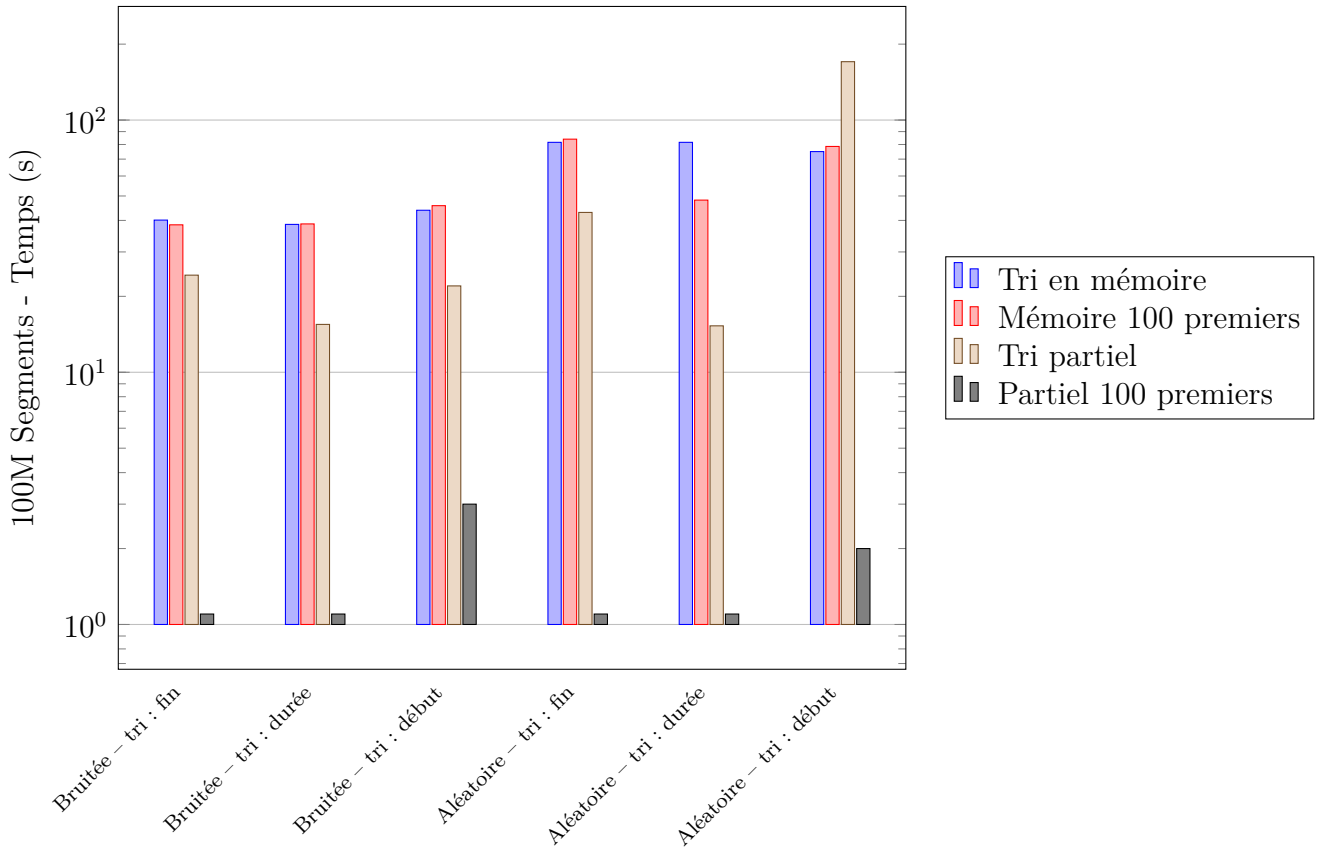



Figure 5.5 Temps de l'itération triée en mémoire comparée au tri de la file de priorité

accélère sensiblement la performance en itération triée de l'arbre.

Nous commençons par l'algorithme naïf du paragraphe précédent pour confirmer la validité de notre approche et groupons les segments dans l'ordre du comparateur désiré. Nous constatons aussi que le groupement selon un des comparateurs dégrade la performance selon les autres.

Dans la Figure 5.6, nous bâtissons un arbre R, optimisé selon une dimension, et mesurons l'accélération de l'itération triée selon cette dimension. Nous constatons des gains de performance sur la dimension du tri, au détriment de la performance de l'itération selon les autres dimensions. Toutefois, par rapport aux gains de performance constatés sur l'arbre à historique à états, les gains ici sont bien moins intéressants pour l'augmentation du temps de construction associée. Nous avons donc décidé qu'appliquer les techniques similaires à R-SHT sur les arbres à segments n'était pas justifié.

## 5.5 Conclusion résultats complémentaires

Dans cette section, avons vu que retirer intervalles porteurs de la valeur nulle ou la déduplication des chaînes de caractères permettrait d'économiser de l'espace de stockage, au dépens de la performance en lecture du SHT. Nous avons aussi apporté de petites modifications au modèle d'une des analyses principales permettant de réduire l'espace nécessaire pour la stocker sans avoir à revoir toute la logique du système à états. Finalement nous utilisons les techniques du SHT pour permettre au stock de segments d'utiliser la mémoire externe pour stocker des entrées sous la forme $< temps_{début}, temps_{fin}, valeur >$ efficacement. Nous proposons aussi des algorithmes pour effectuer des opérations de tri sur le stock de segments avec une utilisation fortement réduite de la mémoire principale.

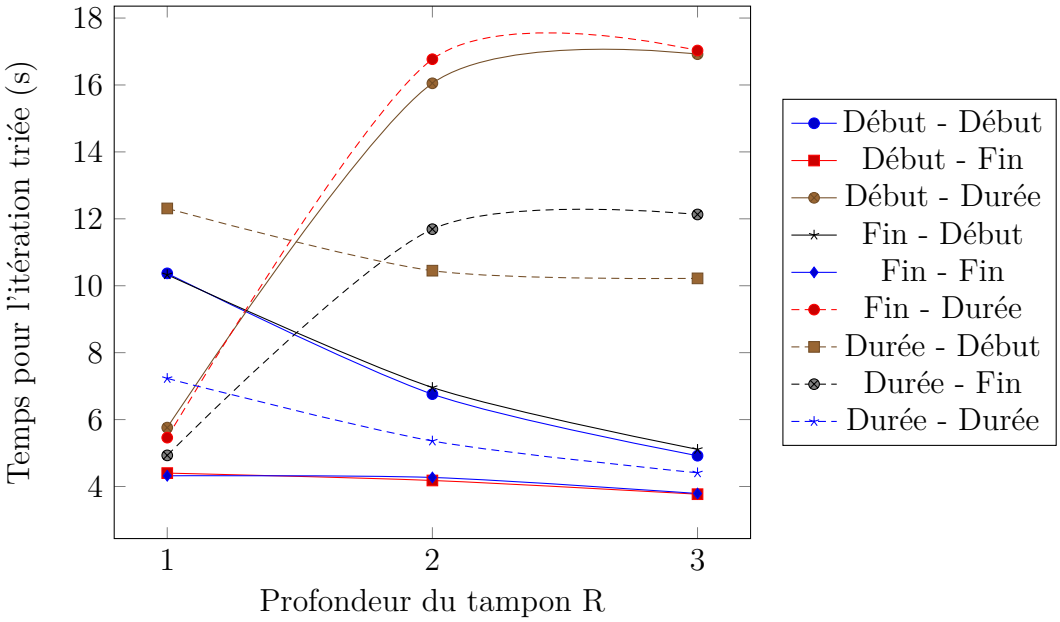



Figure 5.6 Accélération de l'itération triée par une optimisation unidimensionnelle

# CHAPITRE 6 DISCUSSION GÉNÉRALE

## 6.1 Format de retour pour les structures de données sur disque

Les structures de données dont nous parlons dans ce mémoire sont faites pour stocker des informations en mémoire externe, car la quantité d'information stockée dépasse souvent la taille de la mémoire principale disponible. Ces structures seraient inutiles si nous ne pouvions extraire les informations du disque. Au cours de nos travaux, nous avons été confrontés à plusieurs cas dans lesquels il était nécessaire d'extraire une grande quantité de données. Par exemple lorsque nous cherchons tous les noms et processus parents des processus dans la vue principale de Trace Compass pour établir l'index de ces derniers. Choisir une structure pour retourner ces informations a été au coeur de nombreuses réflexions.

Pour les requêtes 2D présentées dans l'article à la section 4.6.4, nous avions initialement choisi de retourner un dictionnaire associant les clés des attributs à une liste triée des intervalles demandés pour cette clé [1]. Ceci posait de nombreux problèmes :

- Etait-il vraiment nécessaire de passer autant de temps à trier les intervalles, alors que ça ne serait pas toujours nécessaire ?
- Il n'était pas possible d'interrompre proprement une requête en cours lors qu'elle n'était plus nécessaire – lors de la fermeture ou du déplacement par l'utilisateur d'une vue par exemple.
- L'objet retourné était monolithique et pouvait rapidement occuper beaucoup de mémoire à lui tout seul.

Dans des versions ultérieures, nous avons opté pour retourner un itérateur non trié, utilisant l'évaluation paresseuse, ce qui comportait plusieurs avantages :

- Faible surcout en mémoire par l'utilisation d'un itérateur et d'un nombre réduit d'objets en arrière-plan pour l'alimenter.
- La requête en cours peut désormais être interrompue.
- Il est toujours possible de bâtir une dictionnaire comme celui retourné précédemment lorsque cela est nécessaire.
- L'arbre à attributs permet de retrouver facilement et rapidement l'objet associé à l'intervalle retourné.

1. Type Java : HashMap<Integer, ArrayList<ITmfStateInterval> >

## 6.2 Choix de l'algorithme de regroupement d'intervalles

Dans la section 4.6.3, nous choisissions de rassembler les intervalles dans un même noeud de l'arbre si leurs clés étaient proches. Ce choix a le mérite de produire de bons résultats pour les requêtes pour lesquelles il est utilisé, mais apparait aussi comme une approche naïve par rapport aux choix qui sont faits dans certaines implémentations de R-Trees – notamment [32, 34, 38, 39].

Nous avons aussi expérimenté avec des algorithmes de classification tel K-Means [35], mais celui-ci augmentait drastiquement le temps de construction de l'arbre, sans accélérer les requêtes.

## 6.3 Rétrocompatibilité des fichiers d'historique d'états

Chaque changement apporté à la manière d'enregistrer des fichiers d'historique d'états peut rompre la compatibilité avec les versions précédentes. Pour éviter les erreurs à la lecture, l'entête du fichier contient un numéro de version pour confirmer que son encodage est compatible avec le décodage du logiciel utilisé pour le lire. Lorsque les versions diffèrent, l'historique n'est pas ouvert, il y a alors deux options. Si la trace est toujours disponible, elle sera analysée de nouveau et les résultats de l'analyse prendront la place de l'ancien historique d'états – à la manière d'un cache de résultats. En pratique la plupart des échanges d'informations entre développeurs se font par échange des traces donc le seul coût est celui de reproduire l'analyse. Toutefois, lorsque la trace n'est plus disponible, le fichier d'historique est le seul à contenir les informations nécessaires, et il faut charger la version du logiciel adaptée pour accéder aux analyses.

Les changements proposés dans ce présent mémoire ont poussé à repenser la gestion des différentes versions, d'autant plus que ces dernières impliquent aussi des changements profonds de la structure du fichier. En effet, d'autres logiciels dépendant du format original d'historique d'état, il aurait été déraisonnable de retirer tout support pour ce dernier. La solution choisie fut donc de supporter plusieurs types d'historiques simultanément (SHT, oSHT et eSHT, les R-SHT ayant la même sérialisation que l'eSHT), plutôt que d'abandonner les plus anciennes. Bien que ce choix requiert plus de support que l'abandon, il permet de faciliter la transition vers les nouveaux historiques.

# CHAPITRE 7 CONCLUSION

Pour conclure ce mémoire, nous dressons une synthèse des contributions apportées au domaine des structures de données pour séries temporelles et le stockage des résultats d'analyse de traces. Nous rappellerons aussi certaines limitations de la solution et émettons quelques pistes pour des évolutions futures.

## 7.1 Synthèse des travaux

Dans ces travaux, nous avons proposé un ensemble de modifications à la base de données pour intervalles d'états et pour le logiciel de visualisation Trace Compass visant à permettre à ces derniers de se mettre à l'échelle de tous types de traces qui pourraient être analysées.

Premièrement nous avons modifié la structure afin d'éviter que l'arbre ne dégénère en profondeur lorsque les analyses produisent de nombreux attributs. Ce faisant, nous évitons aussi le gaspillage d'espace disque, pour les traces problématiques, comme pour les traces typiques.

Nous avons ensuite ajouté la dimension des clés des attributs aux métadonnées des noeuds, afin de limiter quels noeuds sont parcourus lors de la recherche. Le surcout en espace utilisé est faible, mais permet aux requêtes ponctuelles d'être deux fois plus rapides que la structure originale.

Remarquant que les intervalles sont insérés d'une manière non optimale dans les noeuds, et pouvant encore mieux exploiter l'indexation sur deux dimensions, nous avons proposé un algorithme de construction alternatif, pour restreindre le plus possible l'écart entre les bornes sur le temps comme sur les clés. À travers des tests sur des traces réelles et une modélisation du comportement, nous avons montré que le surcout à la construction permettait d'accélérer considérablement les requêtes. Nous modifions à nouveau l'algorithme de construction pour qu'il adapte automatiquement le degré d'optimisation aux données en entrée, plus spécifiquement au nombre d'attributs.

Nous avons ensuite ajouté un nouveau type de requête au système d'états pour permettre au logiciel d'extraire plus rapidement les informations de la base de données, tout en limitant le nombre d'objets intermédiaires stockés en mémoire et réduisant le nombre de noeuds lus.

Finalement, nous avons appliqué ces améliorations à une autre structure utilisée pour stocker des informations temporelles, tout en proposant des algorithmes efficaces pour les requêtes spécifiques à cette base de données.

## 7.2 Limitations de la solution proposée

Bien que nous ayons éliminé les cas de dégénérescence les plus connus, il peut encore exister des jeux de données pouvant faire dégénérer la structure ou pouvant être des cas inefficaces. Malgré le nombre de traces et d'analyses que nous avons essayé, nous n'avons pas encore rencontré de cas aussi problématique que les traces avec de nombreux fils.

Désormais la limite en taille pour les résultats des analyses de traces est la capacité de la mémoire externe, la solution actuelle ne propose aucun moyen de dépasser ces limites, en distribuant les intervalles sur plusieurs machines par exemple. Une solution dotée de ces fonctionnalités pourrait se mettre à l'échelle pour l'analyse de systèmes bien plus complexes et aussi bénéficier des capacités de calcul distribuées pour accélérer l'analyse des traces.

## 7.3 Améliorations futures

Parmi les améliorations futures, nous cherchons un algorithme de regroupement des intervalles qui considérerait le regroupement sur les deux dimensions – temps et clés – au lieu de juste sur le temps, tout en groupant rapidement les intervalles.

Comme il existe désormais de nombreuses similarités entre la base de données des intervalles et la base de données des segments, il serait certainement possible de faire converger les deux afin de réduire le volume de code à maintenir et simplifier le travail des développeurs.

Il est aussi possible d'améliorer les paramètres de degré maximum – actuellement 50 – et de taille des noeuds – actuellement $64ko$ – par des valeurs plus adéquates que ces défauts. Ces nouvelles valeurs pourraient même être calculées automatiquement selon les propriétés des traces – leur taille et le nombre d'attributs par exemple – et du système de fichier – disque mécanique ou non.

Une solution pour économiser de la place sur disque tout en maintenant des bonnes performances en lecture serait d'encoder les temps des intervalles relativement au temps de début du noeud les contenant, permettant ainsi d'utiliser moins d'octets pour les encoder avec une technique comme les `varints` des `protobuf` de Google [64].

# RÉFÉRENCES


[1] A. Montplaisir-Goncalves, N. Ezzati-Jivan, F. Wininger, et M. Dagenais, "State history tree : An incremental disk-based data structure for very large interval data," in *Social Computing (SocialCom), 2013 International Conference on*, Sept 2013, pp. 716–724.

[2] B. Gregg, "Flame graphs," http://www.brendangregg.com/flamegraphs.html, accessed : 2016-11-22.

[3] M. van Dongen, "Hilbert curve," http://www.texample.net/tikz/examples/hilbert-curve/, accessed : 2016-11-22.

[4] P. O'Neil, E. Cheng, D. Gawlick, et E. O'Neil, "The log-structured merge-tree (lsm-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[5] D. Williams, M. Mailänder, et J. Arthorne, "Performance/automated tests - eclipsepedia," https://wiki.eclipse.org/Performance/Automated_Tests, accessed : 2016-11-17.

[6] A. Chan, W. Gropp, et E. Lusk, "An efficient format for nearly constant-time access to arbitrary time intervals in large trace files," *Scientific Programming*, vol. 16, no. 2-3, pp. 155–165, 2008.

[7] A. Montplaisir, N. Ezzati-Jivan, F. Wininger, et M. Dagenais, "Efficient model to query and visualize the system states extracted from trace data," in *International Conference on Runtime Verification*. Springer, 2013, pp. 219–234.

[8] R. Stallman, R. Pesch, S. Shebs *et al.*, "Debugging with gdb," 2002.

[9] M. Desnoyers et M. Dagenais, "Os tracing for hardware, driver and binary reverse engineering in linux," *CodeBreakers Journal*, vol. 1, no. 2, 2006.

[10] T. Bird, "Measuring function duration with ftrace," in *Proceedings of the Linux Symposium*. Citeseer, 2009, pp. 47–54.

[11] M. Desnoyers et M. R. Dagenais, "The lttng tracer : A low impact performance and behavior monitor for gnu/linux," in *OLS (Ottawa Linux Symposium)*, vol. 2006. Citeseer, 2006, pp. 209–224.

[12] V. M. Weaver, "Linux perf_event features and overhead," in *The 2nd International Workshop on Performance Analysis of Workload Optimized Systems, FastPath*, 2013, p. 80.

[13] C. E. Wu, A. Bolmarcich, M. Snir, D. Wootton, F. Parpia, A. Chan, E. Lusk, et W. Gropp, "From trace generation to visualization : A performance framework for

distributed parallel systems," in *Proc. of SC2000 : High Performance Networking and Computing*, November 2000.

[14] I. Google. (2013) Web tracing framework. [en ligne]. Disponible sur : http://google.github.io/tracing-framework/

[15] O. Zaki, E. Lusk, W. Gropp, et D. Swider, "Toward scalable performance visualization with Jumpshot," *High Performance Computing Applications*, vol. 13, no. 2, pp. 277–288, Fall 1999.

[16] M. Geimer, B. Kuhlmann, F. Pulatova, F. Wolf, et B. Wylie, *Scalable Collation and Presentation of Call-Path Profile Data with CUBE*, ser. NIC series. Jülich : John von Neumann Institute for Computing, 2007, vol. 38, pp. 645–652, record converted from VDB : 12.11.2012. [en ligne]. Disponible sur : http://juser.fz-juelich.de/record/58173

[17] A. Pop et A. Cohen, "Openstream : Expressiveness and data-flow compilation of openmp streaming programs," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 53 :1–53 :25, janv. 2013. [en ligne]. Disponible sur : http://doi.acm.org/10.1145/2400682.2400712

[18] K. Coulomb, A. Degomme, M. Faverge, et F. Trahay, "An open-source tool-chain for performance analysis," in *Tools for High Performance Computing 2011*. Springer, 2012, pp. 37–48.

[19] M. Côté et M. R. Dagenais, "Problem Detection in Real-Time Systems by Trace Analysis," *Advances in Computer Engineering*, vol. 2016, 2016, article ID 9467181.

[20] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, et W. E. Nagel, *The Vampir Performance Analysis Tool-Set*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2008, pp. 139–155. [en ligne]. Disponible sur : http://dx.doi.org/10.1007/978-3-540-68564-7_9

[21] M. I. Mughal, R. Javed, et J. Kraft, "Recording of scheduling and communication events on complex telecom systems," Mémoire de maîtrise, School of Innovation, Design and Engineering, Mälardalen University, Västerås and Eskilstuna, Sweden, 2008.

[22] H. Samet, *The design and analysis of spatial data structures*. Addison-Wesley Reading, MA, 1990, vol. 199.

[23] T. H. Cormen, *Introduction to algorithms*, 3rd ed. MIT Press, 2009, ch. 14.

[24] S. Har-Peled, *Geometric approximation algorithms*. American mathematical society Providence, 2011, vol. 173.

[25] M. d. Berg, O. Cheong, M. v. Kreveld, et M. Overmars, *Computational Geometry : Algorithms and Applications*, 3rd ed. Santa Clara, CA, USA : Springer-Verlag TELOS, 2008, ch. 10.

[26] D. Comer, “Ubiquitous B-Tree,” *ACM Comput. Surv.*, vol. 11, no. 2, pp. 121–137, juin. 1979. [en ligne]. Disponible sur : http://doi.acm.org/10.1145/356770.356776

[27] L. Arge, “The buffer tree : A new technique for optimal i/o-algorithms,” in *Workshop on Algorithms and Data structures*. Springer, 1995, pp. 334–345.

[28] B. Becker, S. Gschwind, T. Ohler, B. Seeger, et P. Widmayer, “An asymptotically optimal multiversion b-tree,” *The VLDB Journal*, vol. 5, no. 4, pp. 264–275, déc. 1996. [en ligne]. Disponible sur : http://dx.doi.org/10.1007/s007780050028

[29] R. A. Finkel et J. L. Bentley, “Quad trees a data structure for retrieval on composite keys,” *Acta informatica*, vol. 4, no. 1, pp. 1–9, 1974.

[30] J. L. Bentley, “Multidimensional binary search trees used for associative searching,” *Commun. ACM*, vol. 18, no. 9, pp. 509–517, sept. 1975. [en ligne]. Disponible sur : http://doi.acm.org/10.1145/361002.361007

[31] A. Guttman, “R-trees : A dynamic index structure for spatial searching,” in *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’84, vol. 14, no. 2. New York, NY, USA : ACM, juin. 1984, pp. 47–57. [en ligne]. Disponible sur : http://doi.acm.org/10.1145/602259.602266

[32] A. Korotkov, “A new double sorting-based node splitting algorithm for R-tree,” *Programming and Computer Software*, vol. 38, no. 3, pp. 109–118, mai 2012. [en ligne]. Disponible sur : http://link.springer.com/article/10.1134/S0361768812030024

[33] N. Roussopoulos et D. Leifker, “Direct spatial search on pictorial databases using packed r-trees,” *SIGMOD Rec.*, vol. 14, no. 4, pp. 17–31, mai 1985. [en ligne]. Disponible sur : http://doi.acm.org/10.1145/971699.318900

[34] S. Brakatsoulas, D. Pfoser, et Y. Theodoridis, “Revisiting r-tree construction principles,” in *Proceedings of the 6th East European Conference on Advances in Databases and Information Systems*, ser. ADBIS ’02. London, UK, UK : Springer-Verlag, 2002, pp. 149–162. [en ligne]. Disponible sur : http://dl.acm.org/citation.cfm?id=646046.676614

[35] J. MacQueen *et al.*, “Some methods for classification and analysis of multivariate observations,” in *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, vol. 1, no. 14. Oakland, CA, USA., 1967, pp. 281–297.

[36] D. A. White et R. C. Jain, “Similarity indexing : Algorithms and performance,” in *Electronic Imaging : Science & Technology*. International Society for Optics and Photonics, 1996, pp. 62–73.

[37] T. K. Sellis, N. Roussopoulos, et C. Faloutsos, “The r+-tree : A dynamic index for multi-dimensional objects,” in *Proceedings of the 13th International*

*Conference on Very Large Data Bases*, ser. VLDB '87. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1987, pp. 507–518. [en ligne]. Disponible sur : http://dl.acm.org/citation.cfm?id=645914.671636

[38] N. Beckmann, H.-P. Kriegel, R. Schneider, et B. Seeger, "The r*-tree : An efficient and robust access method for points and rectangles," *SIGMOD Rec.*, vol. 19, no. 2, pp. 322–331, mai 1990. [en ligne]. Disponible sur : http://doi.acm.org/10.1145/93605.98741

[39] I. Kamel et C. Faloutsos, "Hilbert r-tree : An improved r-tree using fractals," in *Proceedings of the 20th International Conference on Very Large Data Bases*, ser. VLDB '94. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1994, pp. 500–509. [en ligne]. Disponible sur : http://dl.acm.org/citation.cfm?id=645920.673001

[40] M. A. Nascimento et J. R. O. Silva, "Towards historical r-trees," in *Proceedings of the 1998 ACM Symposium on Applied Computing*, ser. SAC '98. New York, NY, USA : ACM, 1998, pp. 235–240. [en ligne]. Disponible sur : http://doi.acm.org/10.1145/330560.330692

[41] Y. Tao et D. Papadias, "The mv3r-tree : A spatio-temporal access method for timestamp and interval queries," in *Proceedings of Very Large Data Bases Conference (VLDB), 11-14 September, Rome*, 2001.

[42] L. Chen, R. Choubey, et E. A. Rundensteiner, "Bulk-insertions into r-trees using the small-tree-large-tree approach," in *Proceedings of the 6th ACM International Symposium on Advances in Geographic Information Systems*, ser. GIS '98. New York, NY, USA : ACM, 1998, pp. 161–162. [en ligne]. Disponible sur : http://doi.acm.org/10.1145/288692.288722

[43] R. Choubey, L. Chen, et E. A. Rundensteiner, "Gbi : A generalized r-tree bulk-insertion strategy," in *Proceedings of the 6th International Symposium on Advances in Spatial Databases*, ser. SSD '99. London, UK, UK : Springer-Verlag, 1999, pp. 91–108. [en ligne]. Disponible sur : http://dl.acm.org/citation.cfm?id=647226.719080

[44] L. Arge, K. Hinrichs, J. Vahrenhold, et J. Vitter, "Efficient bulk operations on dynamic r-trees1," *Algorithmica*, vol. 33, pp. 104–128, 2002.

[45] N. Roussopoulos, S. Kelley, et F. Vincent, "Nearest neighbor queries," in *ACM sigmod record*, vol. 24, no. 2. ACM, 1995, pp. 71–79.

[46] C. Gurret et P. Rigaux, "The sort/sweep algorithm : A new method for r-tree based spatial joins," in *Proceedings of the 12th International Conference on Scientific and Statistical Database Management*. IEEE Computer Society, 2000, p. 153.

[47] H. N. Gies, "Dalmatinerdb - design," https://dalmatiner.readme.io/docs/ddb-design, accessed : 2016-11-18.

[48] I. Assent, R. Krieger, F. Afschari, et T. Seidl, “The ts-tree : efficient time series search and retrieval,” in *Proceedings of the 11th international conference on Extending database technology : Advances in database technology*. ACM, 2008, pp. 252–263.

[49] “Opentsdb,” http://opentsdb.net/, accessed : 2016-11-18.

[50] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, et R. E. Gruber, “Bigtable : A distributed storage system for structured data,” *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[51] “Kairosdb,” https://kairosdb.github.io/, accessed : 2016-11-18.

[52] A. Lakshman et P. Malik, “Cassandra : A decentralized structured storage system,” *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, avr. 2010. [en ligne]. Disponible sur : http://doi.acm.org/10.1145/1773912.1773922

[53] “Time series database | nosql time series database | riak ts | basho,” http://basho.com/products/riak-ts/, accessed : 2016-11-18.

[54] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, et W. Vogels, “Dynamo : Amazon’s highly available key-value store,” *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 205–220, oct. 2007. [en ligne]. Disponible sur : http://doi.acm.org/10.1145/1323293.1294281

[55] “Yourkit : Yourkit java profiler,” https://yourkit.com/java/profiler/features/, accessed : 2016-10-18.

[56] A. Flaig, D. Hertl, et F. Krüger, “Evaluation von java-profiler-werkzeugen,” 2014.

[57] L. Prieur-Drevon, R. Beamonte, N. Ezzati, et M. Dagenais, “Enhanced State History Tree (eSHT) : a Stateful Data Structure for Analysis of Highly Parallel System Traces,” in *2016 IEEE International Congress on Big Data*. IEEE, 2016.

[58] M. Noeth, P. Ratn, F. Mueller, M. Schulz, et B. R. de Supinski, “Scalatrace : Scalable compression and replay of communication traces for high-performance computing,” *Journal of Parallel and Distributed Computing*, vol. 69, no. 8, pp. 696 – 710, 2009, best Paper Awards : 21st International Parallel and Distributed Processing Symposium (IPDPS 2007). [en ligne]. Disponible sur : //www.sciencedirect.com/science/article/pii/S074373150800169X

[59] N. Ezzati-Jivan et M. R. Dagenais, “A stateful approach to generate synthetic events from kernel traces,” *Advances in Software Engineering*, vol. 2012, p. 6, 2012.

[60] L. Biveinis, S. Šaltenis, et C. S. Jensen, “Main-memory operation buffering for efficient r-tree update,” in *Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment, 2007, pp. 591–602.

[61] K. Kawachiya, K. Ogata, et T. Onodera, “Analysis and reduction of memory inefficiencies in java strings,” *SIGPLAN Not.*, vol. 43, no. 10, pp. 385–402, oct. 2008. [en ligne]. Disponible sur : http://doi.acm.org/10.1145/1449955.1449795

[62] J. Kotek, “Mapdb,” http://www.mapdb.org/, accessed : 2016-11-1.

[63] P. L. Lehman et s. B. Yao, “Efficient locking for concurrent operations on b-trees,” *ACM Trans. Database Syst.*, vol. 6, no. 4, pp. 650–670, déc. 1981. [en ligne]. Disponible sur : http://doi.acm.org/10.1145/319628.319663

[64] J. Feng et J. Li, “Google protocol buffers research and application in online game,” in *IEEE Conference Anthology*, Jan 2013, pp. 1–4.

[65] S. Har-Peled, *Geometric approximation algorithms.* American mathematical society Providence, 2011, vol. 173.

[66] E. Achtert, H.-P. Kriegel, et A. Zimek, “Elki : A software system for evaluation of subspace clustering algorithms,” in *Proceedings of the 20th International Conference on Scientific and Statistical Database Management*, ser. SSDBM ’08. Berlin, Heidelberg : Springer-Verlag, 2008, pp. 580–585. [en ligne]. Disponible sur : http://dx.doi.org/10.1007/978-3-540-69497-7_41

[67] L. Williams, “Pyramidal parametrics,” in *ACM Siggraph Computer Graphics*, vol. 17, no. 3. ACM, 1983, pp. 1–11.