N° D'ORDRE 2011-ISAL-0133

Thèse

présentée devant

L'Institut National des Sciences Appliquées de Lyon

pour obtenir le grade de

Docteur

spécialité

Informatique

par

Brice Chardin

SGBD open-source pour historisation de données et impact des mémoires flash

Soutenue publiquement devant le jury :

Luc Bouganim, Directeur de Recherche, INRIA Paris-Rocquencourt .... Rapporteur
Christine Collet, Professeur des Universités, Grenoble INP .... Rapporteur
Bruno Defude, Professeur, Télécom SudParis .... Examinateur
Yann Gripay, Maître de Conférences, INSA de Lyon .... Examinateur
Olivier Pasteur, Ingénieur Chercheur, EDF R&D .... Co-directeur de thèse
Jean-Marc Petit, Professeur des Universités, INSA de Lyon .... Co-directeur de thèse

EDF R&D
Laboratoire d'InfoRmatique en Image et Systèmes d'information
École Doctorale Informatique et Mathématiques

RÉSUMÉ

Les systèmes de production d'électricité d'EDF sont largement instrumentés : au total, environ un million de capteurs échangent de l'ordre du milliard d'enregistrements par jour. L'archivage de ces données industrielles est un problème complexe. En effet, il s'agit de stocker un grand nombre de données – sur la durée de vie des installations, plusieurs décennies – tout en supportant la charge des insertions temps réel et des requêtes d'extraction et d'analyse.

Depuis les années 90, EDF a fait le choix de systèmes dédiées à ce cas d'application : les progiciels d'historisation. Ces produits « de niche » ont continué à évoluer en parallèle des autres systèmes de gestion de données, en se spécialisant pour ce segment du marché. Ces progiciels d'historisation sont des solutions propriétaires avec des coûts de licence de l'ordre de plusieurs dizaines de milliers d'euros, et dont le fonctionnement interne n'est pas dévoilé. Nous avons donc dans un premier temps mis en évidence les spécificités des progiciels d'historisation, tant au niveau des fonctionnalités que des performances.

Néanmoins, pour s'affranchir du coût de licence ou pour intégrer le système d'historisation dans un environnement où les ressources matérielles sont limitées voire où un progiciel d'historisation n'est pas utilisable, un SGBD conventionnel peut être mis en œuvre dans ce contexte applicatif. Un cas concret de ce type de matériel est l'IGCBox.

L'IGCBox est un mini PC industriel utilisant MySQL pour l'archivage court ou moyen terme – de quelques jours à deux ans – des données de production des centrales hydrauliques. Ce matériel présente quelques spécificités, la principale étant son système de mémoire non volatile basé uniquement sur la technologie flash, pour sa fiabilité importante en milieu industriel et sa faible consommation d'énergie. Les SGBD possèdent pour des raisons historiques de nombreuses optimisations spécifiques aux disques durs, et le manque d'optimisation adaptée aux mémoires flash peut dégrader significativement les performances. Le choix de ce type de mémoire a donc eu des répercussions notables sur les performances pour l'insertion, avec une dégradation importante par rapport aux disques durs.

Nous avons donc proposé Chronos, une bibliothèque logicielle intégrable à MySQL pour la gestion des données de l'historisation et l'utilisation de mémoires flash. Pour cela, nous avons en particulier identifié un algorithme d'écriture « quasi-séquentiel » efficace pour accéder à la mémoire. Chronos intègre également des mécanismes de bufferisation et de mise à jour d'index optimisés pour les charges typiques de l'historisation.

Les résultats expérimentaux montrent un gain significatif pour les insertions par rapport à des solutions équivalentes, d'un facteur 20 à 54. En contrepartie, ces optimisations ont un impact sur les performances en extraction, en particulier car celles-ci correspondent à des lectures aléatoires sur le support de stockage. Bien que cette dégradation soit significative avec des disques durs, celle-ci est beaucoup moins marquée avec des mémoires flash, où Chronos propose des performances du même ordre de grandeur par rapport aux autres solutions étudiées – dégradées d'un facteur 1.3 à 2.4. Chronos est donc une solution compétitive lorsque les insertions correspondent à une proportion importante de la charge soumise au SGBD. En particulier, Chronos se distingue en proposant des performances globales 4× à 18× meilleures par rapport aux autres solutions pour les charges typiques des IGCBox.

ABSTRACT

EDF – a leading energy company – operates over five hundred power stations, each of which being extensively instrumented: altogether, about one million sensors transmit around one billion records per day. Archiving this industrial data is a complex issue: a large volume of data – throughout the plant's lifetime, several decades – has to be stored while meeting performance requirements for real-time insertions, along with retrieval and analysis queries.

Since the 1990s, EDF has opted for a category of system dedicated to these applications: data historians. These niche products have evolved concurrently with other data management systems, specializing in this market segment. Data historians are proprietary solutions, with license fees of tens of thousands of dollars, and whose internal mechanisms are not documented. Therefore, we first emphasized data historian specificities, with regards to functionalities as much as performance.

However, to cut license costs or to use devices with limited hardware resources, or even when a data historian is not usable, a conventional DBMS can be adopted for this application. At EDF, this case occurs in a practical case: IGCBoxes.

IGCBoxes are industrial mini PCs using MySQL for short to mid-term – from a few days to two years – hydroelectric power stations data archiving. These equipments expose distinctive features, mainly on their storage system based exclusively on flash memory, for its reliability in an industrial environment and its low energy consumption. For historical reasons, DBMS include many hard disk drive-oriented optimizations, and the lack of adjustment for flash memories can significantly decrease performance. This type of memory thus had notable consequences on insert performance, with a substantial drop compared with hard disk drives.

We therefore designed Chronos, a software library pluggable in MySQL, for historization data management on flash memories. For that purpose, we especially identified an efficient "quasi-sequential" write pattern on flash memories. Chronos also include buffer and index management techniques optimized for historization typical workloads.

Experimental results demonstrate improved performance for insertions over different solutions, by a factor of 20 to 54. On the other hand, these optimizations effect extraction performance negatively, as they involve random reads on the storage system. Although this degradation is important with hard disk drives, it is much less notable with flash memories, with which Chronos performs nearly as well as other solutions – by a factor of 1.3 to 2.4. As a result, Chronos is competitive when insertions make up an extensive part of the workload. For instance, Chronos stands out with the typical workload of IGCBoxes, with global performance 4× to 18× higher that other solutions.

TABLE DES MATIÈRES

Table des matières **1**

1 Introduction et contexte industriel **2**
- 1.1 Contexte industriel 3
- 1.2 Mémoires flash 5
- 1.3 Problématique 5
- 1.4 Contributions 6
- 1.5 Organisation du document 7

2 Les systèmes d'historisation de données **8**
- 2.1 Progiciels d'historisation de données 9
- 2.2 Comparaison fonctionnelle 10
- 2.3 Positionnement parmi les SGBD 20
- 2.4 Benchmark adapté à l'historisation de données 22
- 2.5 Expérimentation du benchmark 29
- 2.6 Synthèse 33

3 Les mémoires flash **34**
- 3.1 Fonctionnement et caractéristiques des mémoires flash 35
- 3.2 Flash Translation Layer 42
- 3.3 Performances 50
- 3.4 Optimisation du système d'exploitation pour les mémoires flash 52
- 3.5 Optimisation des SGBD pour les mémoires flash 55
- 3.6 Amélioration des écritures aléatoires 58
- 3.7 Synthèse 66

4 Chronos : une approche "NoSQL" pour la gestion de données historiques sur flash **68**
- 4.1 Principe de fonctionnement 69
- 4.2 Implémentation 77
- 4.3 Résultats expérimentaux 80
- 4.4 Synthèse 85

5 Conclusion et perspectives **87**

Bibliographie **89**

A Annexes **94**
- A.1 Procédures stockées du benchmark pour MySQL 94
- A.2 Accès aux moteurs de stockage de MySQL en SQL 97
- A.3 Calcul de complexité pour les insertions dans Chronos 99

Chapitre 1

Introduction et contexte industriel

Plan du chapitre

1.1 Contexte industriel 3
1.2 Mémoires flash 5
1.3 Problématique 5
1.4 Contributions 6
1.5 Organisation du document 7

Ce premier chapitre introduit les problématiques de l'historisation de données à partir du contexte industriel d'EDF. Nous présentons également les contributions principales et l'organisation du document.

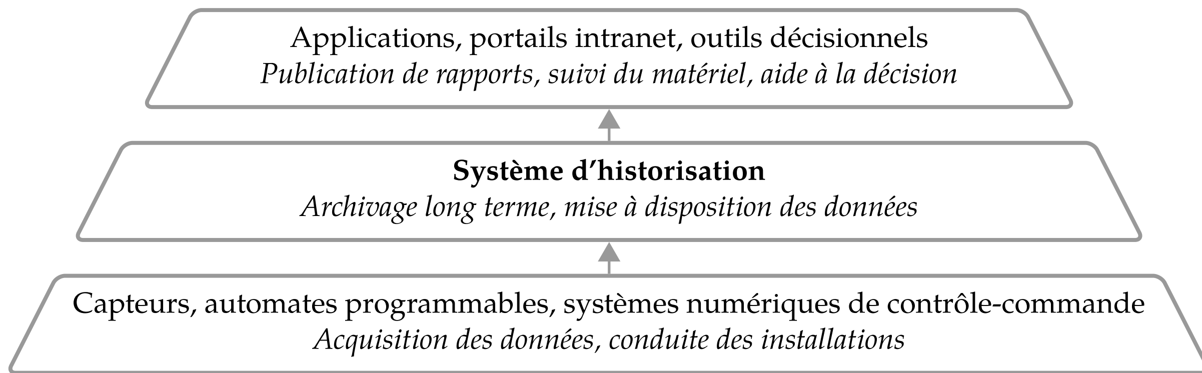


FIGURE 1.1: Position des systèmes d'historisation dans le système d'information de production

1.1 Contexte industriel

EDF est un des principaux producteurs d'électricité dans le monde. En France, le groupe exploite un parc constitué de centrales nucléaires, hydrauliques, thermiques à flammes (gaz, charbon, fioul), éoliennes et solaires.

Ces systèmes de production d'électricité, répartis sur plus de cinq cent sites, sont largement instrumentés : au total, environ un million de capteurs constituent les éléments de base des systèmes d'acquisition de données, et échangent de l'ordre du milliard d'enregistrements par jour. L'exploitation de ce volume important de données nécessite dans un premier temps de disposer de moyens d'accès. Pour cela, des systèmes de gestion de bases de données (SGBD) se chargent de leur archivage et de leur mise à disposition. L'archivage de ces données industrielles est un problème complexe. En effet, il s'agit de stocker un grand nombre de données – sur la durée de vie des installations, soit plusieurs décennies – tout en supportant la charge des insertions temps réel et des requêtes d'extraction et d'analyse soumises au SGBD.

Au sein de ces centrales, le système d'information est constitué dans un premier temps d'un ensemble de dispositifs informatiques utilisés en temps réel pour l'exploitation et la surveillance des installations : des systèmes de contrôle-commande prennent en charge des actions automatiques sur le procédé, tandis que d'autres systèmes acquièrent et traitent les données de production pour permettre leur visualisation et assister les opérateurs dans leurs activités de contrôle du procédé, de suivi des performances ou d'optimisation de la conduite. À ce niveau, les données sont principalement issues des capteurs ou des rapports d'état du contrôle-commande, mais également de saisies manuelles et d'applications métiers spécifiques pour certains traitements.

L'accès direct à ces données est délicat à cause des impératifs de sûreté : toute intervention sur les données doit se faire sans impacter le fonctionnement de l'existant. À EDF, cet accès est fourni par ce qu'il est convenu d'appeler un « système d'historisation », chargé d'archiver les données de production [Pasteur, 2007]. Ces données sont alors mises à disposition de diverses applications sans contraintes temps réel (suivi de matériel, outils décisionnels, etc.) et d'acteurs extérieurs à la centrale (ingénierie, R&D, etc.). La figure 1.1 reprend l'organisation du système d'informations des centrales d'EDF, et présente le positionnement des systèmes d'historisation comme intermédiaires fondamentaux pour l'accès aux données de production.

Les systèmes d'historisation reçoivent principalement des données en provenance du procédé industriel, mais aussi certaines fournies par des entités extérieures, comme les demandes

FIGURE 1.2: Une IGCBOX

de production et les tarifs des matières premières ou de l'électricité, provenant du « Réseau de transport d'électricité » (RTE), ou les prévisions météorologiques. De plus, des capteurs relatifs à l'informatique d'essai peuvent être reliés de manière provisoire au procédé industriel et alimenter le système d'historisation.

À l'échelle d'une centrale, des milliers de séries temporelles sont archivées par le système d'historisation. Chaque série, désignée par un identifiant de « repère » ou « point », est une séquence de mesures horodatées, généralement représentées par un ensemble de paires date-valeur – où la valeur peut posséder plusieurs dimensions, typiquement la mesure et une métadonnée sur sa qualité. Les périodes d'échantillonnage varient selon les repères, allant de 200 ms – pour la vitesse de rotation de la turbine par exemple – à quelques minutes. Certains repères correspondent à des variables de type événement, dont l'acquisition se produit lors d'un changement d'état – l'ouverture ou la fermeture d'une vanne par exemple.

Quelques systèmes dédiés possèdent des fréquences d'acquisition nettement supérieures, par exemple pour les analyses vibratoires où les fréquences sont de l'ordre de 20 kHz ; ces traitements sont souvent effectués indépendamment des systèmes d'historisation, en archivant éventuellement des résultats agrégés.

En dehors de cet archivage long terme, les besoins d'EDF en terme d'archivage existent dans des contextes très différents : les moyens de production isolés (un barrage téléconduit par exemple) ont besoin d'un archivage local à court terme, pouvant être embarqué dans des mini PC industriels ou des automates de télégestion. Un cas concret de matériel en charge de la gestion des données de production à court terme est l'IGCBox (cf. figure 1.2).

L'IGCBox – « IGC » pour Interface Générale de Communication – a été mise en œuvre pour permettre la remontée et l'archivage des données des centrales thermiques et hydrauliques. En particulier, ce matériel est utilisé pour collecter les données des capteurs des centrales hydroélectriques, pour les transmettre ensuite à des serveurs centraux.

Ce contexte d'archivage possède des caractéristiques communes avec l'historisation de données à long terme, comme les contraintes de performance en insertion. L'IGCBox présente néanmoins quelques spécificités, la principale étant son système de mémoire non volatile basé uniquement sur la technologie flash. Le choix de ce type de mémoire, pour sa fiabilité importante en milieu industriel et sa faible consommation d'énergie, a eu des répercussions notables sur les performance pour l'insertion, avec une dégradation importante par rapport aux disques durs [Pasteur et Léger, 2007].

1.2 Mémoires flash

Les mémoires flash sont une solution alternative aux disques durs pour le stockage des données. Bien que la transition d'une technologie à une autre soit facilitée par des interfaces compatibles, un simple changement de matériel ne suffit généralement pas à exploiter au maximum le potentiel des mémoires flash.

En effet, leur fonctionnement diffère de celui des disques durs car les mémoires flash possèdent trois types d'opération bas niveau : la lecture ou l'écriture d'une page (typiquement 528, 2112 ou 4224 octets), et l'effacement d'un bloc (typiquement 32 ou 64 pages). De plus, des mécanismes de correction d'erreurs et de répartition de l'usure sont nécessaires pour améliorer la durée de vie de la mémoire. Pour que cette technologie soit interchangeable avec les disques durs, l'interface d'accès est restreinte aux opérations de lecture et d'écriture. Cette interface est fournie par une sur-couche logicielle fréquemment intégrée à la mémoire flash : la *Flash Translation Layer* (FTL), qui permet également de simplifier la gestion de la mémoire en prenant en charge les codes de correction d'erreurs, la répartition de l'usure et les effacements.

Malgré ces contraintes, les mémoires flash présentent certains avantages sur les disques durs. Concernant les défaillances, la limitation de la durée de vie est généralement prévisible étant donné l'absence de parties mécaniques. Il est également plus facile de concevoir un périphérique résistant à des conditions environnementales extrêmes en terme de chocs, vibrations, variations de températures, présence de particules, humidité, pression et rayonnements. En plus de leur robustesse, les mémoires flash sont plus denses et consomment moins d'énergie que les disques durs. Ces caractéristiques les rendent particulièrement adaptées aux systèmes embarqués. Concernant les performances, les mémoires flash proposent des accès rapides en lecture séquentielle et aléatoire, ainsi qu'en écriture séquentielle. En contrepartie, les performances sont fréquemment dégradées pour les écritures aléatoires.

1.3 Problématique

Dans cette thèse, nous avons attaqué deux problématiques différentes issues de ce contexte applicatif :

▸ **Positionnement des progiciels d'historisation de données par rapport aux SGBD classiques et systèmes NoSQL.**

Les systèmes de gestion de bases de données relationnels (SGBDR) sont, depuis une quarantaine d'années, la solution privilégiée pour manipuler des données structurées. Cependant, ceux-ci sont peu utilisés dans ce contexte industriel très spécifique soumis à des contraintes de performances en insertion. Depuis les années 90, EDF a fait le choix d'une catégorie de SGBD dédiée à ce cas d'application pour gérer ces données : les progiciels d'historisation. Ces systèmes ont continué à évoluer en parallèle des SGBDR, en se spécialisant pour ce segment du marché.

Ces progiciels d'historisation sont des solutions propriétaires avec des coûts de licence de l'ordre de plusieurs dizaines de milliers d'euros, et dont le fonctionnement interne n'est pas dévoilé. D'un autre coté, de nombreuses solutions open-source existent pour manipuler de telles données. Les SGBDR en font partie, mais également certains SGBD « NoSQL » dont l'interface d'accès simplifiée peut être adaptée au contexte. Cependant, leurs différences avec les progiciels d'historisation restent mal connues, tant au niveau des fonctionnalités que des performances : alors que de nombreux benchmarks permettent d'évaluer les SGBD dans divers environnements d'utilisation, il n'en existe à notre connaissance aucun à ce jour pour l'historisation de données industrielles.

▸ **Impact des mémoires flash dans le contexte industriel d'EDF.**

Les progiciels d'historisation sont destinés à des serveurs ayant une capacité de traitement importante, et sont donc peu adaptés à des environnements plus contraints. Cela limite leur utilisation pour l'historisation de données à court terme sur des équipements de type IGC-Box. D'autres solutions de gestion de données peuvent donc être plus pertinentes pour ce cas d'utilisation. Cependant, les SGBD possèdent pour des raisons historiques de nombreuses optimisations spécifiques aux disques durs. Le manque d'optimisation relative aux mémoires flash peut dégrader significativement leurs performances, en particulier lorsqu'ils sont soumis à une charge importante en insertion, comme dans le contexte industriel d'EDF.

La simple substitution d'une technologie par une autre, bien que possible, ne suffit pas à exploiter tout le potentiel des mémoires flash : c'est toute l'architecture des SGBD qui doit être revisitée, puisqu'elle est principalement issue du verrou technologique du temps d'entrées/-sorties entre la mémoire volatile et les disques durs.

1.4 Contributions

Dans ce contexte, la première contribution de ce travail vise à donner des éléments de réponse sur le positionnement des progiciels d'historisation parmi les SGBD. Pour cela, nous proposons d'analyser les différences entre les progiciels d'historisation de données, les SGBDR, les systèmes de gestion de flux de données (*Data Stream Management System* ou DSMS) et les SGBD NoSQL. Cette analyse se base sur les couvertures fonctionnelles de chaque catégorie de système, puis sur la mise en œuvre d'un benchmark adapté aux systèmes d'historisation. Ce benchmark se base sur un scénario inspiré du fonctionnement de l'historisation des données des centrales nucléaires en générant des requêtes d'insertion, de mise à jour, de récupération et d'analyse des données. Des résultats expérimentaux sont présentés pour le progiciel d'historisation InfoPlus.21 [Aspen Technology, 2007], le SGBDR MySQL [Oracle, 2011b], et le SGBD NoSQL Berkeley DB [Olson *et al.*, 1999].

Étant donné le volume important de données à archiver pour l'historisation par les IGCBox, nous proposons ensuite une seconde contribution sur l'optimisation des écritures sur mémoire flash, destinée à une catégorie de périphériques bas ou milieu de gamme comme ceux intégrés à cet équipement. Pour cela, nous identifions une corrélation forte entre les performances des écritures et leur proximité spatiale sur la mémoire flash et, à partir de cette propriété, proposons un algorithme de placement des données permettant d'améliorer les performances en écriture aléatoire, en contrepartie d'une diminution de la capacité utile de la mémoire. L'efficacité de cette technique est validée par une formalisation avec un modèle mathématique et des résultats expérimentaux. Avec cette optimisation, les écritures aléatoires deviennent potentiellement aussi efficaces que les écritures séquentielles, allant jusqu'à améliorer leurs performances de deux ordres de grandeur.

Finalement une convergence entre les deux problématiques est proposée, avec la conception d'un moteur de stockage spécifique aux mémoires flash et dédié aux données de l'historisation. Ce SGBD, appelé « Chronos », intègre l'algorithme d'écriture précédent, ainsi que des mécanisme de bufferisation et de mise à jour d'index optimisés pour les charges typiques des systèmes d'historisation. Il se présente sous la forme d'une bibliothèque, pouvant également être intégrée à MySQL en tant que moteur de stockage. Ses performances sont comparées aux autres solutions à l'aide du benchmark adapté aux systèmes d'historisation. Les résultats expérimentaux mettent en évidence la gestion efficace des insertions par Chronos, en supportant une charge 20× à 54× plus importante. Malgré des performances dégradées en extraction (d'un

facteur 1.3× à 2.4×), Chronos est une solution compétitive lorsque les insertions correspondent à une proportion importante de la charge soumise au SGBD, comme pour l'archivage temporaire par les IGCBox.

1.5 Organisation du document

Ce mémoire s'organise autour des trois principales contributions de la thèse : l'étude du positionnement des progiciels d'historisation parmi les SGBD, l'algorithme d'écriture sur mémoire flash et le SGBD adapté à l'historisation de données sur mémoires flash.

Après cette introduction, le chapitre 2 présente les progiciels d'historisation, et les positionne par rapport aux autres catégories de SGBD. Un benchmark est ensuite défini pour évaluer les performances de ces systèmes ; celui-ci est utilisé avec un progiciel d'historisation, un SGBDR et un SGBD NoSQL. Le chapitre 3 commence par un état de l'art de l'utilisation des mémoires flash dans les SGBD. Suite à cela, nous mettons en évidence l'importance de la localité spatiale pour les écritures, et proposons un algorithme d'écriture pour en tirer parti. Nous en présentons ensuite une approximation par un modèle mathématique, puis des résultats expérimentaux. Le chapitre 4 présente Chronos, notre approche « NoSQL » pour la gestion de données historiques sur mémoires flash. Nous détaillons ses principes de fonctionnement, puis donnons une évaluation de ses performances à l'aide du benchmark défini au chapitre 2, en le comparant avec d'autres SGBD. Le chapitre 5 conclut ce mémoire et présente des perspectives pour les travaux futurs. Trois annexes précisent l'implémentation du benchmark, l'interface d'accès des moteurs de stockage MySQL et les calculs de complexité pour Chronos.

CHAPITRE 2

LES SYSTÈMES D'HISTORISATION DE DONNÉES

Plan du chapitre

2.1 Progiciels d'historisation de données 9
2.2 Comparaison fonctionnelle 10
2.2.1 Intégration 11
2.2.2 Historisation 13
2.2.3 Restitution 17
2.2.4 Configuration 18
2.2.5 Politique tarifaire 19
2.2.6 Synthèse 19
2.3 Positionnement parmi les SGBD 20
2.3.1 Progiciels d'historisation et SGBDR 20
2.3.2 Progiciels d'historisation et systèmes NoSQL 20
2.3.3 Progiciels d'historisation et DSMS 21
2.3.4 Synthèse 21
2.4 Benchmark adapté à l'historisation de données 22
2.4.1 Modèle de données 23
2.4.2 Requêtes 24
2.5 Expérimentation du benchmark 29
2.5.1 Résultats 30
2.6 Synthèse 33

Dans ce chapitre, nous cherchons à répondre à la question suivante : comment se positionnent les progiciels d'historisation de données parmi les systèmes de gestion de données ? Du point de vue d'EDF, la réponse à cette question peut avoir un impact important sur ses systèmes d'information. Pour cela, nous examinons trois autres catégories de systèmes : SGBDR, systèmes de gestion de flux de données et systèmes NoSQL ; et définissons un benchmark dérivé du contexte industriel d'EDF.

2.1 Progiciels d'historisation de données

Les progiciels d'historisation de données – comme InfoPlus.21 [Aspen Technology, 2007], PI [OSIsoft, 2009] ou Wonderware Historian [Invensys Systems, 2007] – sont des progiciels propriétaires conçus pour archiver et interroger les données de séries temporelles issues de l'automatique industrielle. Leurs fonctionnalités de base sont toutefois proches de celles des systèmes de gestion de base de données (SGBD), mais spécialisées dans l'historisation de séries temporelles.

Ces progiciels d'historisation de données – ou *data historians* – ont pour but de collecter en temps réel les données en provenance du procédé industriel et de les restituer à la demande. Pour cela, ils sont capables de gérer les données de plusieurs milliers de capteurs : leur historique, pouvant porter sur des décennies de fonctionnement, est alimenté par un flux constant de données en insertion, tout en effectuant des calculs et en répondant aux besoins des utilisateurs.

Pour centraliser les données fournies par les systèmes d'acquisition, les progiciels d'historisation supportent les interfaces et protocoles de communication industriels – comme OPC (*Object Linking and Embedding for Process Control*) [OPC Foundation, 2003], Modbus [Modbus-IDA, 2006] ou des protocoles spécifiques aux fabriquants des appareils d'acquisition. Ceux-ci peuvent également servir à communiquer avec les logiciels de supervision ou de contrôle de procédé utilisés en centrale.

Les progiciels d'historisation supportent des rythmes d'insertion élevés, et peuvent ainsi traiter plusieurs centaines de milliers d'évènements – soit quelques mégaoctets – à la seconde. Ces performances sont permises par une conception spécifique des buffers d'insertion, qui conservent les valeurs récentes en mémoire volatile pour les écrire ensuite sur disque, triées chronologiquement. Les données acquises dont la date ne correspond pas à la fenêtre du buffer sont écrites dans des zones dédiées avec des performances réduites, ou même supprimées.

Pour stocker un volume important de données avec une utilisation du disque minimale et un taux d'erreur acceptable, les progiciels d'historisation proposent des moteurs de compression efficaces, avec ou sans perte d'information. Chaque repère est alors associé à des règles décrivant les conditions d'archivage des nouvelles valeurs – par exemple avec un intervalle de ré-échantillonnage ou avec des seuils de déviation par rapport à des valeurs interpolées.

Le modèle de données utilisé par les progiciels d'historisation est un *modèle hiérarchique*, permettant de représenter les données suivant la structuration physique des installations et faciliter ainsi la consultation de données similaires groupées par sous-systèmes. Dans le domaine de la production d'électricité par exemple, les repères peuvent être regroupés dans un premier temps par site d'exploitation, puis par unité de production, et enfin par système élémentaire (e.g. le système de production d'eau déminéralisée).

Concernant l'extraction de données, les systèmes d'historisation sont un intermédiaire fondamental dans les systèmes d'information technique, en fournissant les données pour des applications liées au fonctionnement de la centrale – comme la surveillance de matériel ou la maintenance – et d'aide à la décision – comme la publication de statistiques ou le contrôle financier. Ces applications peuvent bénéficier des fonctionnalités des systèmes d'historisation spécifiques aux séries temporelles, particulièrement l'interpolation et le ré-échantillonnage, ou la récupération de valeur précalculées à partir des valeurs brutes. Les mesures indirectes, comme la consommation d'énergie auxiliaire ou le coût du carburant par exemple, les indicateurs de performance, les diagnostiques ou les informations sur la disponibilité peuvent être calculées par des modules associés et archivées par le progiciel d'historisation.

Des fonctionnalités de visualisation des données sont fournies par les clients standards. Ils

facilitent l'exploitation des données archivées en affichant des courbes, tables, statistiques et autres synoptiques. Ces clients peuvent donner une estimation rapide des données en ne récupérant que les points représentatifs de la période considérée.

Les progiciels d'historisation proposent également une vision relationnelle des données archivées, accessible à l'aide d'une interface SQL. Des extensions du langage sont définies, pour faire le lien avec leurs fonctionnalités spécifiques – des vues sont par exemple définies par défaut pour consulter des valeurs ré-échantillonnées.

Les progiciels d'historisation sont des systèmes propriétaires dont le fonctionnement interne est opaque. Il existe par ailleurs des distinctions entre les différents progiciels d'historisation du marché. Dans un premier temps, nous comparons en détail les fonctionnalités proposées par trois des principales solutions, en les mettant en parallèle à deux SGBD open-source : MySQL et Berkeley DB. Nous mettons ensuite en avant les grandes différences entre les progiciels d'historisation et trois catégories de SGBD : SGBDR, DSMS et systèmes NoSQL.

2.2 Comparaison fonctionnelle

Cette section présente les différences de fonctionnalités entre le SGBDR open-source MySQL, le SGBD NoSQL open-source Berkeley DB et trois progiciels d'historisation de données utilisés à EDF : InfoPlus.21, PI et Wonderware Historian.

Cette comparaison porte sur 82 indicateurs, déterminés à partir de discussions avec les utilisateurs, et regroupés en 5 catégories :

- Intégration : les caractéristiques de l'environnement de déploiement (systèmes d'exploitation, support de grappes de serveurs, interfaces d'accès et de communication),
- Historisation : les limites et fonctionnalités liées à l'historisation (types de données, limites du système, capacité de compression et comportement face aux défaillances),
- Restitution : les fonctionnalités liées à la restitution (clients inclus dans l'offre, capacité d'interpolation et de définition d'alarmes),
- Configuration : les possibilités de configuration et de personnalisation du système.
- Politique tarifaire : le type et coût des licences et des offres de support.

Les informations reportées sur ces grilles sont basées sur les documentations fournies par les différents éditeurs. Les mentions *ND* (Non Disponible) et *NA* (Non Applicable) précisent lorsque celles-ci ne sont pas communiquées, ou ne sont pas applicables pour le système considéré. Ces données sont appelées à évoluer en fonction des versions des logiciels ; et ne correspondent pas nécessairement aux distributions les plus récentes. Cette comparaison concerne InfoPlus.21 version 2006.5 [Aspen Technology, 2007], PI version 3.4 [OSIsoft, 2009], Wonderware Historian 9.0 [Invensys Systems, 2007], MySQL 5.5 [Oracle, 2011b] et Berkeley DB version 5.2 [Oracle, 2011a].

Les systèmes dédiés à l'historisation possèdent généralement des modules métiers, notamment au niveau des logiciels clients inclus dans l'offre. Les fonctionnalités offertes par ces modules portent principalement sur des aspects de simulation ou de visualisation spécifiques aux séries temporelles de données de capteurs. Ce comparatif se focalisant sur la fonction d'historisation des données, ces modules métiers ne sont donc pas détaillés.

MySQL et Berkeley DB

Pour l'archivage temporaire des données des centrales hydrauliques, EDF s'intéresse à des solutions open-source afin de s'affranchir du coût de licence et de pouvoir étendre ou optimiser

le SGBD avec des développements spécifiques. Parmi les deux catégories de SGBD étudiées – SGBDR et entrepôt de paires clé-valeur ordonnées –, MySQL et Berkeley DB font partie des solutions open-source les plus répandues, et ont donc été considérés dans ce comparatif.

MySQL est un SGBDR open-source maintenu par Oracle, avec une communauté d'utilisateurs importante. Celui-ci est généralement utilisé pour la gestion de bases de données avec une architecture client-serveur, bien qu'une bibliothèque logicielle – *Embedded MySQL* – soit aussi disponible [Oracle, 2011b].

Berkeley DB [Olson *et al.*, 1999] est un entrepôt de paires clé-valeur open-source maintenu également par Oracle, disponible uniquement en tant que bibliothèque logicielle. De part ce schéma simplifié, Berkeley DB est un système NoSQL dans le sens « non relationnel ». En particulier, ce SGBD ne supporte pas le langage SQL. Cependant, contrairement aux solutions NoSQL conçues pour passer à l'échelle horizontalement (comme Dynamo [DeCandia *et al.*, 2007], Cassandra [Lakshman et Malik, 2009] ou Voldemort [Kreps, 2009]), Berkeley DB n'intègre pas de mécanisme de distribution sur plusieurs serveurs. Toutefois, Berkeley DB correspond à une approche NoSQL pour le stockage local des données – sur une seule machine – et est notamment utilisé par Voldemort pour cet usage.

De part les requêtes usuelles de l'historisation, qui consultent des « séquences » de mesure (*range queries*), les systèmes NoSQL basés sur des tables de hachage – majoritaires dans ce domaine – sont peu adaptés à cette utilisation. Berkeley DB permet quant à lui l'utilisation de B-trees pour stocker les données.

2.2.1 Intégration

Système d'exploitation

Les progiciels d'historisation ont tendance à ne fonctionner qu'avec Windows, pour le serveur comme pour les clients. Jusqu'en 2007, OSIsoft maintenait une version du serveur PI pour HP-UX, Solaris et IBM AIX, mais celle ci devrait, à terme, ne plus être proposée.

À l'inverse, MySQL et Berkeley DB peuvent être compilés pour des systèmes et des architectures divers. En particulier, ceux-ci peuvent être configurés pour s'adapter à des environnements embarqués, contrairement aux progiciels d'historisation destinées à des serveurs ayant une capacité de traitement importante. Berkeley DB est notamment compatible avec plusieurs systèmes d'exploitations mobiles : Android, iOS et Windows Mobile.

Table 2.1: Systèmes d'exploitation supportés par le serveur

	InfoPlus.21	PI	Wonderware	MySQL	Berkeley DB
BSD	non	non	non	oui	oui
HP-UX	non	arrêté	non	oui	oui
IBM AIX	non	arrêté	non	oui	oui
Linux	non	non	non	oui	oui
Mac OS X	non	non	non	oui	oui
SE mobiles	non	non	non	non	oui
Solaris	non	arrêté	non	oui	oui
Windows	oui	oui	oui	oui	oui

De part sa conception, Berkeley DB ne possède pas de clients (cf. sections 2.2 et 2.2.3).

TABLE 2.2: Systèmes d'exploitation supportés par les clients

	InfoPlus.21	PI	Wonderware	MySQL	Berkeley DB
BSD	non	non	non	oui	NA
Linux	non	non	non	oui	NA
Mac OS X	non	non	non	oui	NA
Solaris	non	non	non	oui	NA
Windows	oui	oui	oui	oui	NA

Architecture

Lors de la répartition d'un système d'historisation sur plusieurs serveurs, la réplication des données permet d'améliorer la disponibilité en cas de défaillance matérielle, et éventuellement de répartir la charge pour les opérations d'extraction (récupération des données et analyse). La distribution permet quant à elle de répartir la charge pour tous les types d'opérations – insertions et extractions. Ces deux architectures ne sont pas mutuellement exclusives, si le système le permet, il est possible de composer réplication et distribution.

TABLE 2.3: Réplication

	InfoPlus.21	PI	Wonderware	MySQL	Berkeley DB
Réplication	oui	oui	oui [1]	oui	oui
Basculement automatique en cas de panne	non	partiel [2]	non	non	oui
Équilibrage de charge intégré	non	oui	ND	non	non
Supervision du système	non	oui	ND	non	non
Synchronisation	non	non	non	partiel [3]	oui

TABLE 2.4: Distribution

	InfoPlus.21	PI	Wonderware	MySQL	Berkeley DB
Distribution	non	non	non	oui	non
Basculement automatique en cas de panne	NA	NA	NA	oui	NA
Équilibrage de charge intégré	NA	NA	NA	oui	NA

1. Wonderware Historian permet de répliquer les données à intervalles réguliers, d'une minute à un jour.
2. La gestion de la configuration n'est pas basculée.
3. MySQL supporte les réplications asynchrone et semi-synchrone (i.e. au moins un des serveurs-esclaves est synchronisé).

	InfoPlus.21	PI	Wonderware	MySQL	Berkeley DB
Supervision du système	NA	NA	NA	non	NA

Interfaces d'accès et de communication

Pour le développement de clients ou l'interconnexion avec d'autres éléments du système d'information, les systèmes d'historisation fournissent des interfaces de programmation (API) et peuvent supporter des interfaces standards ou des protocoles de communication industriels.

TABLE 2.5: Interfaces

	InfoPlus.21	PI	Wonderware	MySQL	Berkeley DB
C/C++	oui	oui	partiel [4]	oui	oui
C#	non	oui	oui	oui	oui
Fortran	oui	non	non	non	non
Java	non	non	non	oui	oui
PHP	non	non	non	oui	non
VB	non	oui	partiel [4]	non	non
ODBC	oui	oui	oui	oui	oui
OLE DB	non	oui	oui	non	non
Protocoles de communication industriels	oui	oui	oui	non	non

2.2.2 Historisation

Types de données et contraintes

Les progiciels d'historisation sont généralement conçus pour acquérir des mesures – typiquement des réels ou des entiers – horodatées ; le type de données associé aux dates est donc essentiel dans ce contexte. Pour MySQL, le type standard est peu adapté à l'historisation de données à cause de sa précision à la seconde (à EDF, la précision pour ce type de mesure est de l'ordre de la milliseconde). Il est donc préférable de représenter les dates par un autre type – un entier sur 8 octets par exemple.

Certains progiciels d'historisation imposent des contraintes fortes sur l'horodatage des données. Ces contraintes peuvent porter sur un intervalle de validité, par exemple uniquement des dates passées ou appartenant à une fenêtre temporelle donnée. La séquentialité des données – l'insertion par dates croissantes – peut également être requise. Le non-respect de ces contraintes peut entraîner une diminution notable des performances, voire l'impossibilité d'insérer les données.

4. Les API pour Wonderware historian (C/C++ et VB) ne sont utilisables qu'avec le framework .NET.

Table 2.6: Types de données

	InfoPlus.21	PI	Wonderware	MySQL	Berkeley DB
Entiers	oui	oui	oui	oui	non
Taille maximale (octets)	4	4	4	8	NA
Réels	oui	oui	oui	oui	non
Taille maximale (octets)	8	8	8	8	NA
Dates	oui	oui	oui	oui	non
Précision (en secondes)	10^{-3}	1.525×10^{-5}	10^{-7}	1	NA
Date minimum	-68 ans [5]	01/01/1970	01/01/0001	01/01/1000	NA
Date maximum	+68 ans [5]	01/01/2038	31/12/9999	31/12/9999	NA
Insertion de valeurs antérieures à la date courante	oui	oui	oui	oui	NA
Insertion de valeurs ultérieures à la date courante	oui	non	non	oui	NA
Binaire	oui	oui	non	oui	oui
Taille maximale (octets)	600	976	NA	2^{32}	2^{32}
Texte	oui	oui	oui	oui	oui
Taille maximale (octets)	600	976	512	2^{32}	2^{32}

Compression

Les progiciels d'historisation représentent les données de manière compacte dans leurs fichiers d'archive. Cependant, cette compression sans perte native est propriétaire, elle n'est ni configurable, ni documentée.

Des algorithmes de compression avec perte peuvent également être mis en œuvre, pour réduire le nombre de tuples à archiver. Pour les séries temporelles, la compression par bande morte permet d'éliminer les mesures dont les valeurs sont proches de la dernière valeur stockée. La compression par calcul de pente fonctionne de manière similaire, mais en comparant cette fois la mesure à une approximation linéaire basée sur les mesures adjacentes.

5. Par rapport à la date courante.

TABLE 2.7: Compression

	InfoPlus.21	PI	Wonderware	MySQL	Berkeley DB
Sans perte	partiel	partiel	partiel	partiel [6]	oui
Avec perte	oui	oui	oui	non	non
Par calcul de pente	oui	oui	oui	non	non
Par bande morte	oui	oui	oui	non	non

Capacité de stockage

Cette section indique la capacité maximale de chaque système, en ce qui concerne la quantité de données stockées et le nombre de repères.

La limitation sur la quantité de données archivées peut dépendre également du système d'exploitation ou du système de fichiers et, pour MySQL, du moteur de stockage utilisé.

Les licences des progiciels d'historisation comportent une limite sur le nombre de repères par serveur, leur coût étant basé essentiellement sur ce paramètre (cf. section 2.2.5). MySQL et Berkeley DB ne sont pas concernés par cette limitation, qui dépend uniquement du type de donnée utilisé pour représenter l'identifiant.

TABLE 2.8: Passage à l'échelle

	InfoPlus.21	PI	Wonderware	MySQL	Berkeley DB
Taille maximale utilisée pour les données (en téraoctets)	$> 10^3$	$3.69 \cdot 10^{16}$	ND	$3.84 \cdot 10^8$	256
Nombre maximum de repères	10^6	$2.1 \cdot 10^7$	$1.5 \cdot 10^5$ [7]	NA	NA

Intégrité des données

Les données archivées par le système d'historisation doivent être conservées malgré d'éventuelles défaillances. Le tableau suivant indique les risques de pertes de données pour une architecture à un seul serveur – la réplication permettant généralement de s'affranchir des problèmes de perte de données.

Certains moteurs de stockage de MySQL sont transactionnels, et garantissent l'intégrité des données des transactions validées (*commit*).

Les progiciels d'historisation ne supportent pas les transactions, même si les systèmes d'acquisition fournissent généralement plusieurs niveaux de cache distribués sur le réseau pour prévenir les pertes de données en cas de défaillance du serveur ou du réseau. Ce mécanisme

6. Compression commune pour les chaînes de caractères, dépend du moteur de stockage pour les autres types.
7. Les licences au catalogue de Wonderware Historian autorisent jusqu'à 150 000 repères, mais il semble possible d'obtenir des licences pour un nombre plus important.

permet de compenser les pertes de données éventuelles au niveau du serveur pour les acquisitions les plus récentes.

TABLE 2.9: Intégrité des données

	InfoPlus.21	PI	Wonderware	MySQL	Berkeley DB
Gestion de transactions	non	non	ND	partiel [8]	oui
Garantie de conservation des données	non [9]	non [9]	ND	partiel [8]	oui
Récupération de fichiers corrompus	oui	oui	ND	partiel [8]	oui

Pour pallier aux défaillances matérielles du support de stockage, et pour prévenir la corruption des fichiers d'archive, des sauvegardes peuvent être mises en œuvre pour copier le contenu – ou une partie du contenu – de la base à des emplacements différents.

TABLE 2.10: Sauvegardes

	InfoPlus.21	PI	Wonderware	MySQL	Berkeley DB
Sauvegardes cycliques	oui	oui	oui	oui	non
Rétablissement du serveur à partir du fichier de sauvegarde	oui	oui	oui	oui	oui
Modularité de la sauvegarde	fileset [10]	fichier [11]	fichier [12]	table	database
Paramétrage de la taille	oui	non	non	non	non

Conservation de l'historique

Les systèmes d'historisation peuvent être configurés pour forcer la conservation des données, en désactivant les mises à jour ou en conservant un historique des données modifiées.

TABLE 2.11: Sécurité des données

	InfoPlus.21	PI	Wonderware	MySQL	Berkeley DB
Conservation de l'historique	non	non	oui	non	oui
Désactivation des mises à jour	oui	oui	oui	oui	oui

8. Dépend du moteur de stockage utilisé.
9. Perte des données courantes stockées en mémoire volatile, et éventuellement perte des données historisées dans des fichiers corrompus.
10. Un fileset correspond à un ensemble de repères pour une plage de date.
11. Une sauvegarde incrémentale de la totalité de la base peut également être définie.
12. Une sauvegarde des différentiels ou des logs de transaction peut également être définie.

2.2.3 Restitution

Clients

Les systèmes d'historisation peuvent proposer des clients standards pour manipuler les données (clients SQL ou de visualisation par exemple), d'administrer la base de données (pour les modifications du schéma de la base) ou d'administrer le serveur (pour les modifications de la configuration).

Table 2.12: Clients

	InfoPlus.21	PI	Wonderware	MySQL	Berkeley DB
Client SQL	oui	oui	oui	oui	non
Client d'administration à distance de la base de données	oui	oui	oui	oui	non
Client d'administration à distance du serveur	non	oui	oui	oui	NA
Client d'affichage des données sous forme de tableaux, graphiques et synoptiques	oui	oui	oui	non	non

Interpolation des données

Les systèmes d'historisation représentent des phénomènes physiques continus par des séries de mesures (échantillons). Les données manquantes peuvent cependant être approximées par interpolation, à partir des mesures archivées. Pour les progiciels d'historisation, ces algorithmes d'interpolation sont intégrés au système, et sont utilisables en SQL par des extensions – spécifiques à chaque éditeur – enrichissant ce langage.

Table 2.13: Interpolation

	InfoPlus.21	PI	Wonderware	MySQL	Berkeley DB
Interpolation constante par morceaux	oui	oui	oui	non	non
Interpolation linéaire	oui	oui	oui	non	non
Interpolation polynomiale/spline	non	non	non	non	non

Alarmes

Les alarmes constituent un type particulier de « requêtes continues », qui se déclenchent lorsque les nouvelles données vérifie les conditions de l'alarme. Parmi les cinq systèmes étudiés, aucun ne gère cette catégorie de requêtes dans sa globalité. Certains permettent cependant de

définir des critères et des actions à effectuer lorsque les données insérées remplissent ces critères – par exemple l'envoi d'un e-mail lors du dépassement d'un seuil critique.

TABLE 2.14: Alarmes

	InfoPlus.21	PI	Wonderware	MySQL	Berkeley DB
Définition d'alarmes	oui	oui	oui	non [13]	non

2.2.4 Configuration

Partitionnement

Le partitionnement d'une base de données permet de fractionner le stockage à des emplacements physiques différents, pour faciliter la gestion de la base ou améliorer les performances.

TABLE 2.15: Partitionnement

	InfoPlus.21	PI	Wonderware	MySQL	Berkeley DB
Partitionnement par date	oui	oui	oui	oui	oui
Partitionnement par repère	oui	non	non	oui	oui
Partitionnement par quantité de données	oui	oui	ND	non	non

Vues et vues matérialisées

Les vues sont des requêtes nommées. Les vues matérialisées permettent de stocker les données du résultat de l'évaluation de la requête, mais en dupliquant l'information, avec des mises à jour systématiques ou périodiques.

Les progiciels d'historisation permettent de matérialiser les résultats d'opérations complexes, comme des agrégats ou des calculs impliquant plusieurs repères.

TABLE 2.16: Vues

	InfoPlus.21	PI	Wonderware	MySQL	Berkeley DB
Vues	oui	oui	oui	oui	non
Vues matérialisées	oui	oui	oui	non	non

Autres éléments de configuration

Pour MySQL et Berkeley DB, la structure interne de la base de données peut être adaptée au contexte d'utilisation : pour MySQL cette configuration dépend du choix du moteur de stockage, tandis que Berkeley DB permet d'utiliser un B-tree ou une table de hachage pour l'ar-

13. MySQL ne permet pas la définition de telles alarmes ; cependant, il est possible d'obtenir le même résultat par l'intermédiaire de déclencheurs (*triggers*) appelant des fonctions définies par l'utilisateur.

chivage de paires clé-valeur – Berkeley DB propose des structures additionnelles, une pile ou une file par exemple, pour l'archivage de valeurs seules.

TABLE 2.17: Personnalisation

	InfoPlus.21	PI	Wonderware	MySQL	Berkeley DB
Configuration des caches	oui	oui	oui	oui	oui
Personnalisation des index	non	non	non	oui	oui
Personnalisation de la structure interne de la base de données	non	non	non	oui	partiel
Création de plugins	oui	oui	non	oui	NA
Définition de procédures stockées	oui	non	oui	oui	NA
Création de taches périodiques	oui	non	oui	oui	NA

2.2.5 Politique tarifaire

Pour les progiciels d'historisation, il est difficile d'estimer le coût des licences car leur prix est négocié pour chaque client. De plus, ce prix dépend du contenu de l'offre : nombre de repères, nombre d'utilisateurs, interfaces, logiciels clients, support, etc. L'ordre de grandeur de ce coût de licence est de plusieurs dizaine de milliers d'euros par serveur ; avec des tarifs dégressifs en fonction du nombre de repères – allant de 10€ à 0.5€ par repère. Le coût de maintenance annuel correspond à un pourcentage du prix de la licence.

TABLE 2.18: Politique tarifaire

	InfoPlus.21	PI	Wonderware	MySQL	Berkeley DB
Licence	propriétaire	propriétaire	propriétaire	GPL ou propriétaire	Sleepycat (compatible GPL) ou propriétaire
Tarification	contrat d'entreprise	contrat d'entreprise	contrat d'entreprise	0€ à 3999€ [14]	0€
Support et maintenance	oui	oui	oui	oui [15]	non

2.2.6 Synthèse

Dans l'ensemble, les progiciels d'historisation peuvent être caractérisés par :

- un structure de schéma hiérarchique simple, basée sur les repères,

14. Par serveur et par an ; plusieurs offres commerciales sont définies, incluant des outils supplémentaires et différentes options de support technique.
15. Offre commerciale uniquement.

- une architecture centralisée,
- une intégration facilitée par le support de protocoles de communication industriels et une configuration originale adaptée,
- une conception optimisée pour l'archivage long-terme d'un grand volume de données ordonnées chronologiquement, où la date joue un rôle fondamental,
- des algorithmes de compression adaptés,
- une "interface NoSQL" pour les insertions, mais également pour la récupération des données de séries temporelles, éventuellement avec filtrage, ré-échantillonnage ou calcul d'agrégats,
- une interface SQL étendue,
- pas de transactions,
- des applications intégrées spécialisées pour les données industrielles,
- une politique tarifaire basée sur le nombre de repères,

2.3 Positionnement parmi les SGBD

2.3.1 Progiciels d'historisation et SGBDR

Les progiciels d'historisation sont principalement conçus pour acquérir des données de séries temporelles : ils peuvent difficilement être utilisés pour des bases de données relationnelles. D'ailleurs, même si ils peuvent traiter des requêtes SQL, ils peuvent avoir des limitations avec leur optimiseur de requêtes et leur conformité avec la norme SQL dans son ensemble. Pour ces raisons, certains progiciels d'historisation peuvent être associés à un SGBDR. PI par exemple peut être lié à Microsoft SQL Server pour gérer des données suivant le modèle relationnel ; le modèle hiérarchique étant quant à lui plus proche de la structuration physique des installations.

De nombreux SGBDR supportent les transactions, ce qui n'est pas le cas des progiciels d'historisation. Pour les SGBDR, ces transactions permettent en particulier de garantir la durabilité des données en cas de défaillance. Pour les progiciels d'historisation, plusieurs niveaux de cache distribués sur le réseau limitent les pertes de données, mais assurent également, de manière transparente, une certaine continuité de service.

2.3.2 Progiciels d'historisation et systèmes NoSQL

Les progiciels d'historisation fournissent une interface dédiée, différente du SQL, pour les insertions, mises à jour et récupération des données. Les insertions sont fonctionnellement comparable aux requêtes SQL "INSERT", avec des performances améliorées : ces procédures évitent l'analyse syntaxique et les conversions de types. L'interface de récupération des données cependant diffère significativement du SQL. Les extractions peuvent être définies avec des conditions de filtrage (typiquement avec des seuils de valeur ou des vérifications du champ qualité), du ré-échantillonnage ou des calculs d'agrégats sur des intervalles temporels – par exemple pour calculer la moyenne horaire. Bien que les conditions de filtrage et la définition d'intervalles soient traduisibles simplement en SQL, l'interpolation de valeurs (avec divers algorithmes d'interpolation : par pallier, linéaire, etc.) peut être fastidieuse à définir, tant en SQL qu'avec des interfaces NoSQL usuelles, en particulier lorsque plusieurs séries temporelles – possédant leur propre période d'échantillonnage – sont concernées, comme pour le produit de deux séries par exemple.

Néanmoins, les entrepôts de paires clé-valeur ordonnées fournissent des méthodes d'accès NoSQL proches, comme les curseurs de Berkeley DB [Olson *et al.*, 1999]. Ces curseurs peuvent être positionnés sur une valeur de clé, et être incrémentés ou décrémentés suivant l'ordre des clés – pour récupérer les valeurs consécutives d'une série temporelle dans ce contexte. Mal-

gré tout, l'interface des progiciels d'historisation est spécialisée, et donc combine de nombreux algorithmes et traitements usuels adaptés aux besoins industriels, en plus de l'extraction de données brutes.

Les systèmes NoSQL sont généralement conçus pour être distribués sur de nombreux nœuds. Pour les progiciels d'historisation, cette extensibilité horizontale est séparée entre la réplication et la distribution. L'équilibrage de charge est fourni par la réplication, où plusieurs serveurs possèdent les même données, et sont individuellement capables d'exécuter des requêtes d'extraction. Cependant, cette architecture ne diminue pas la charge liée aux insertions : la distribution des données est obtenue déclarativement, en associant un repère à un serveur donné – qui peu à son tour être répliqué. En conséquence, les progiciels d'historisation fournissent seulement un équilibrage de charge et une extensibilité limités par rapport à la majorité des systèmes NoSQL. Cependant, la récupération des données repose sur un interface efficace pour traiter les requêtes sur des intervalles (*range queries*). Typiquement, les entrepôts de paires clé-valeur basés sur les tables de hachage distribuées (DHT) ne sont pas adaptées, ce qui fait de l'extensibilité un problème complexe.

2.3.3 Progiciels d'historisation et DSMS

Les systèmes de gestion de flux de données (Data Stream Management Systems ou DSMS) fournissent des capacités de traitement de requêtes continues en étendant le langage SQL [Arasu *et al.*, 2003] ou par des extensions des SGBDR classiques, comme Oracle. Ces systèmes traitent typiquement les données sur une fenêtre temporelle relativement courte pour exécuter ces requêtes continues.

En ce qui concerne les insertions, les progiciels d'historisation possèdent des mécanismes similaires car ils associent leur buffer d'écriture à une fenêtre temporelle, rejetant ou insérant avec des performances dégradées les données hors de cet intervalle. Cependant, à EDF, les requêtes continues sont gérées par des systèmes de surveillance et de contrôle dédiés, avec des contraintes temps réel du fait de leur caractère critique ; tandis que le progiciel d'historisation se charge de l'archivage long terme.

Pourtant, une nouvelle génération de DSMS permet l'analyse à long terme de données historiques en entreposant les flux de données. Ces entrepôts de flux de données se focalisent toujours sur les requêtes continues, ce qui n'est pas l'objectif des progiciels d'historisation.

2.3.4 Synthèse

Les progiciels d'historisation sont des produits conçus et vendus pour un usage industriel spécifique. Les autres SGBD peuvent avoir des cadres d'application plus variés, pour un coût potentiellement moins important, mais n'incluent généralement pas la plupart des fonctionnalités métier que possèdent les progiciels d'historisation. Ces systèmes ne supportent typiquement pas les protocoles de communication industriels, ni la compression avec perte, l'interpolation ou le ré-échantillonnage.

En quelque sorte, les progiciels d'historisation sont des SGBD non relationnels – donc « NoSQL » – qui ont su s'imposer sur un marché de niche. Pour autant, ils ne correspondent à aucune des catégories de SGBD existantes :

- modèle hierarchique,
- pas de distribution des données,
- pas de transactions,
- uniquement des données de capteur (pas d'image, de blob, etc.).

Ces critères fonctionnels sont importants mais ne permettent pas d'avoir une idée quantitative sur les capacités de traitement de chaque système. Pour cela, nous avons défini un benchmark pour avoir une base objective de comparaison.

2.4 Benchmark adapté à l'historisation de données

Les éditeurs des progiciels d'historisation ne publient pas les capacités de traitement de leurs solutions. L'utilisation d'un benchmark est donc le seul moyen d'évaluer les différences de performances entre ces systèmes. Cependant, cette comparaison ne s'avère pas si facile, les fonctionnalités, les interfaces et le modèle de données sous-jacents étant différents.

Pour cela, nous nous concentrons sur des opérations (requêtes) simples sur un schéma de base de données générique. Nous proposons donc un benchmark et l'utilisons pour évaluer un progiciel d'historisation, un entrepôt de paires clé-valeur ordonnées et un SGBDR, que nous avons optimisés pour ce cas d'utilisation.

Alors que de nombreux benchmarks existent pour les systèmes de gestion de base de données relationnels, comme TPC-C ou TPC-H [Transaction Processing Performance Council, 2007, 2008], il n'en existe, à notre connaissance, pas pour les progiciels d'historisation. L'idée de comparer ces systèmes à l'aide d'un benchmark existant – adapté aux SGBDR – est donc naturelle. Cependant, dans le contexte des applications d'historisation de données d'EDF, il ne nous a pas semblé possible de mettre en oeuvre un benchmark du Transaction Processing Performance Council (TPC) pour les raisons suivantes :

- Les progiciels d'historisation ne respectent pas nécessairement les contraintes ACID et ne permettent généralement pas les transactions.
- L'insertion est une opération primordiale pour les systèmes d'historisation. Cette opération est effectuée de manière continue, ce qui exclut d'utiliser des benchmarks insérant les données par groupements, comme TPC-H.
- Les progiciels d'historisation sont conçus pour traiter des séries temporelles. Il est nécessaire que le benchmark manipule ce type de données pour que les résultats soient significatifs.

Les benchmarks pour les DSMS, comme *Linear Road* [Arasu *et al.*, 2004] auraient aussi pu être envisagés ; mais les progiciels d'historisation ne supportant pas les requêtes continues, leurs mises en œuvre auraient été impossibles. Ces systèmes – et les SGBDR d'ailleurs – fonctionnent différemment en stockant les données pour répondre aux requêtes ultérieures. Dans les benchmarks des DSMS, même les requêtes sur l'historique utilisent un premier niveau d'agrégation sur les données brutes, ce qui n'est pas représentatif de l'utilisation des systèmes d'historisation à EDF.

Pour comparer les performances des progiciels d'historisation avec d'autres SGBD, nous définissons un benchmark basé sur un scénario reprenant le fonctionnement de l'historisation des données des centrales nucléaires d'EDF. Dans ce contexte, les données sont issues de capteurs répartis sur le site d'exploitation et agrégées par un démon servant d'interface avec le système d'historisation. Pour les insertions, ce benchmark simule le fonctionnement de ce démon et génère pseudo-aléatoirement les données à insérer.

Ces données sont alors accessibles par des applications ou utilisateurs distants, qui peuvent envoyer des requêtes pour mettre à jour, récupérer ou analyser ces données. Après la phase d'insertion, ce benchmark génère un ensemble simple mais représentatif de ce type de requêtes.

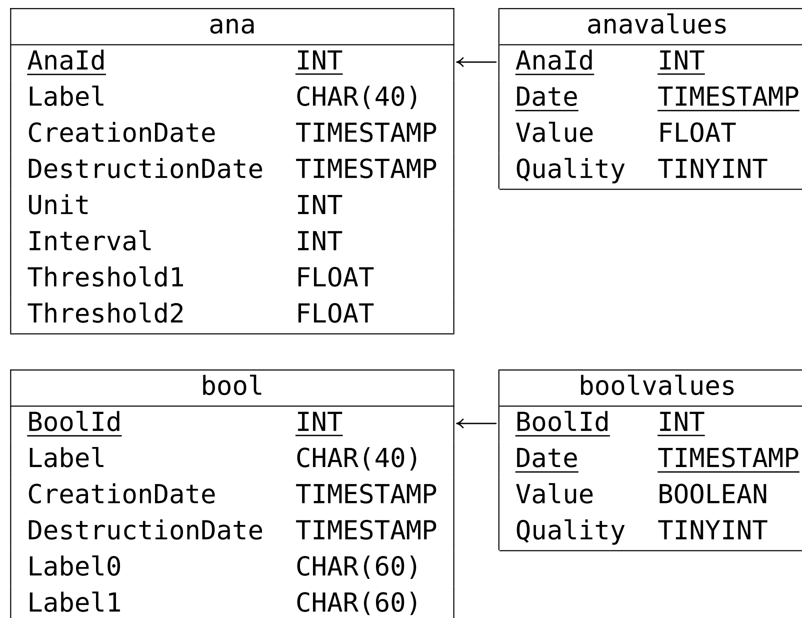


Figure 2.1: Schéma logique de la base de données

2.4.1 Modèle de données

Ce benchmark manipule les données selon un schéma minimal, centré sur les données de séries temporelles. Pour chaque type de variable – analogique ou booléen – une table de description est définie (resp. `ana` et `bool`). Les mesures sont stockées dans des tables différentes (resp. `anavalues` et `boolvalues`). La figure 2.1 présente le schéma logique utilisé pour ce benchmark.

Chaque repère est associé à un identifiant (`AnaId` ou `BoolId`), une courte description – ou nom – (`Label`), une date de création (`CreationDate`) et une date de destruction (`DestructionDate`). Pour les données analogiques, la table de description contient également l'unité de la mesure (`Unit`), qui est normalement décrite dans une table distincte abandonnée pour ce benchmark, une période d'échantillonnage théorique (`Interval`) et deux seuils délimitant les valeurs critiques basses (`Threshold1`) ou hautes (`Threshold2`). Pour les valeurs booléennes, la table de description contient deux courtes descriptions associées à la valeur 0 (`Label0`) et 1 (`Label1`).

Les séries temporelles sont stockées dans les tables `anavalues` et `boolvalues`, qui contiennent l'identifiant (`AnaId` ou `BoolId`), la date de la mesure avec une précision à la milliseconde (`Date`), la valeur (`Value`) et un tableau de huit bits pour les méta données – des informations sur la qualité – (`Quality`). Pour les tables `anavalues` et `boolvalues`, `AnaId` et `BoolId` sont des clés étrangères. On a `anavalues[AnaId]` ⊆ `ana[AnaId]` et `boolvalues[BoolId]` ⊆ `bool[BoolId]`.

Pour que ce benchmark soit compatible avec les modèles de données hiérarchiques utilisés par les progiciels d'historisation, le modèle relationnel défini précédemment ne peut pas être imposé. Dans la figure 2.2, nous proposons un modèle hiérarchique équivalent, permettant de représenter les même données et d'exécuter des requêtes fonctionnellement équivalentes.

La base de données définie précédemment utilise plusieurs types de données :

- `INT` correspond à un entier signé sur 32 bits.
- `FLOAT` correspond à un nombre à virgule flottante sur 32 bits.
- `CHAR(N)` correspond à une chaîne de n caractères.

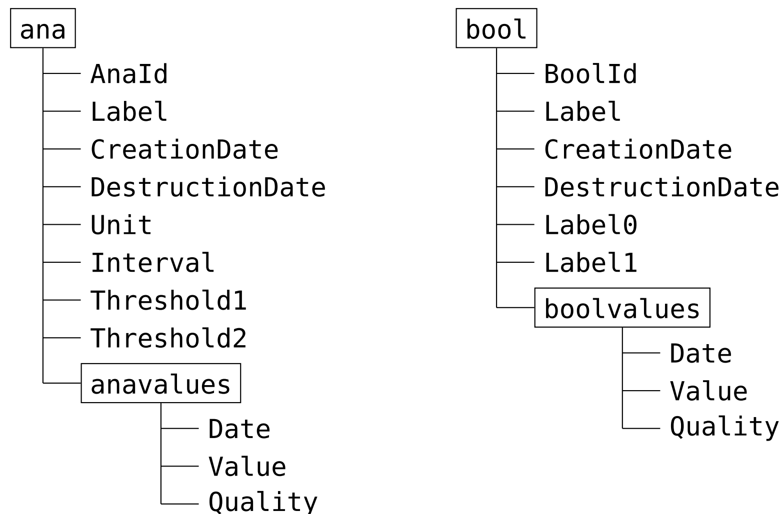


FIGURE 2.2: Schéma logique équivalent pour le modèle hiérarchique des progiciels d'historisation

- TIMESTAMP correspond à un horodatage avec une précision à la milliseconde sur un intervalle d'au moins trente ans.
- TINYINT correspond à un entier signé ou non signé sur 8 bits.
- BOOLEAN correspond à une valeur binaire sur 1 bit.

Les types présentés dans cette section donnent une précision minimale des données manipulées, la substitution d'un type par un autre plus précis est autorisée. Le schéma de la base peut également être adapté pour décomposer un type : un TIMESTAMP peut par exemple être divisé en deux valeurs, une date avec une précision à la seconde et un entier représentant les millisecondes. De plus, les dates ne requièrent pas de format spécifique. Il est par exemple possible d'utiliser des entiers pour stocker ces informations.

2.4.2 Requêtes

Ce benchmark définit douze requêtes représentatives de l'usage d'EDF. Les paramètres de ces requêtes sont notés entre crochets. Ceux-ci doivent être les mêmes entre chaque exécution du benchmark, pour obtenir des données et des requêtes identiques.

Pour conserver une définition simple et faciliter l'analyse des performances, les interactions entre les requêtes ne sont pas prises en comptes : les requêtes sont exécutées une par une dans un ordre établi. En particulier, l'évaluation des performances malgré une charge continue en insertion n'est pas considérée, même si cela correspond à une situation plus réaliste. De même, les traitements spécifiques aux séries temporelles proposés par les progiciels d'historisation ne font pas partie des requêtes exécutées par le benchmark, car leur équivalent en SQL standard peut s'avérer compliqué à définir (par exemple pour calculer une moyenne pondérée par les intervalles temporels – variables – des mesures).

2.4.2.1 Insertion des données

L'insertion de données est une opération primordiale pour les systèmes d'historisation. Pour optimiser ces requêtes, l'interface et le langage ne sont pas imposés (ces requêtes peuvent être traduites du SQL en n'importe quel langage ou appel de fonction de l'API, quel que soit celui qui maximise les performances).

R0.1 Insertion de valeur analogique.
Paramètres ID, DATE, VAL et QUALITY.

```
INSERT INTO anavalues VALUES
  ([ID],[DATE],[VAL],[QUALITY]);
```

R0.2 Insertion de valeur booléenne.
Paramètres ID, Date, VAL et QUALITY.

```
INSERT INTO boolvalues VALUES
  ([ID],[DATE],[VAL],[QUALITY]);
```

2.4.2.2 Modification des données

Au contraire de l'insertion, les mises à jour, récupérations et analyses des données sont généralement effectuées manuellement par les utilisateurs finaux ; les contraintes de performances sont plus flexibles.

Ce benchmark définit dix de ces requêtes pour évaluer les performances de chaque système, et identifier les optimisations spécifiques à certains types de requêtes. Les équivalents NoSQL doivent fournir les mêmes résultats. Nous proposons deux exemples, pour les requêtes R2.1 et R9, en utilisant une interface basée sur les curseurs, qui peuvent être positionnés sur une valeur de clé (position) et incrémentés (readnext).

La mise à jour de données est une opération rare pour les systèmes d'historisation. Le benchmark considère cependant l'impact des mises à jour sur les performances. Les identifiants et les dates doivent correspondre à des données existantes.

R1.1 Mise à jour d'une valeur analogique et de son champ qualité.
Paramètres VAL, ID et DATE.

```
UPDATE anavalues
SET Value = [VAL],
  Quality = (Quality | 128)
WHERE AnaId = [ID]
  AND Date = [DATE];
```

R1.2 Mise à jour d'une valeur booléenne et de son champ qualité.
Paramètres VAL, ID et DATE.

```
UPDATE boolvalues
SET Value = [VAL],
  Quality = (Quality | 128)
WHERE BoolId = [ID]
  AND Date = [DATE];
```

2.4.2.3 Extraction de données brutes

R2.1 Valeurs analogiques brutes.
Paramètres ID, START et END.

```
SELECT *
FROM anavalues
WHERE AnaId = [ID]
  AND Date BETWEEN [START] AND [END]
ORDER BY Date ASC;
```

Algorithme 2.1: Traitement NoSQL de R2.1

```
  input : id, start, end
1 POSITION((id, start))
2 key, value ← READNEXT()
3 while key < (id, end) do
4 │ key, value ← READNEXT()
```

R2.2 Valeurs booléennes brutes.
Paramètres ID, START et END.

```
SELECT *
FROM boolvalues
WHERE BoolId = [ID]
  AND Date BETWEEN [START] AND [END]
ORDER BY Date ASC;
```

2.4.2.4 Calcul d'agrégats

R3.1 Quantité de données analogiques.
Paramètres ID, START et END.

```
SELECT count(*)
FROM anavalues
WHERE AnaId = [ID]
  AND Date BETWEEN [START] AND [END]
```

R3.2 Quantité de données booléennes.
Paramètres ID, START et END.

```
SELECT count(*)
FROM boolvalues
WHERE BoolId = [ID]
  AND Date BETWEEN [START] AND [END]
```

R4 Somme de valeurs analogiques.
Paramètres ID, START et END.

```
SELECT sum(Value)
FROM anavalues
WHERE AnaId = [ID]
  AND Date BETWEEN [START] AND [END]
```

R5 Moyenne de valeurs analogiques.
Paramètres ID, START et END.

```
SELECT avg(Value)
FROM anavalues
WHERE AnaId = [ID]
  AND Date BETWEEN [START] AND [END]
```

R6 Minimum et Maximum de valeurs analogiques.
Paramètres ID, START et END.

```
SELECT min(Value), max(Value)
FROM anavalues
WHERE AnaId = [ID]
  AND Date BETWEEN [START] AND [END]
```

2.4.2.5 Filtrage sur la valeur

R7 Dépassement de seuil critique.
Paramètres ID, START et END.

```
SELECT Date, Value
FROM ana, anavalues
WHERE ana.AnaId = anavalues.AnaId
  AND ana.AnaId = [ID]
  AND Date BETWEEN [START] AND [END]
  AND Value > ana.Threshold2;
```

R8 Dépassement de valeur.
Paramètres ID, START, END et THRESHOLD.

```
SELECT Date, Value
FROM anavalues
WHERE AnaId = [ID]
  AND Date BETWEEN [START] AND [END]
  AND Value > [THRESHOLD];
```

2.4.2.6 Calcul d'agrégats avec filtrage sur plusieurs séries

R9 Repère avec valeurs anormales.
Récupère le repère dont les valeurs ne sont, le plus souvent, pas comprises entre ses deux seuils critiques entre deux dates.
Paramètres START et END.

```
SELECT Label, count(*) as count
FROM ana, anavalues
WHERE ana.AnaId = anavalues.AnaId
  AND Date BETWEEN [START] AND [END]
  AND (Value > Threshold2 OR Value < Threshold1)
GROUP BY ana.AnaId
ORDER BY count DESC
LIMIT 1;
```

Algorithme 2.2: Traitement NoSQL de R9

```
   input : start, end
 1 foreach id in ana.AnaId do
 2    count[id] ← 0
 3    threshold1 ← ana[id].Threshold1
 4    threshold2 ← ana[id].Threshold2
 5    POSITION((id, start))
 6    key, value ← READNEXT()
 7    while key < (id, end) do
 8       if value.Value < threshold1 or value.Value > threshold2 then
 9          count[id]++
10       key, value ← READNEXT()
11 result_id ← i: ∀ id, count[id] ≤ count[i]
12 RETURN(ana[result_id].Label, count[result_id])
```

2.4.2.7 Opérations sur les dates

R10 Vérification de la période d'échantillonnage.
Récupère les repères dont la période d'échantillonnage ne respecte pas la valeur `Interval` donnée dans la table ana.
Paramètres START et END.

```
SELECT values.AnaId, count(*) as count
FROM ana,
(
  SELECT D1.AnaId, D1.Date,
         min(D2.Date-D1.Date) as Interval
  FROM anavalues D1, anavalues D2
  WHERE D2.Date > D1.Date
    AND D1.AnaId = D2.AnaId
    AND D1.Date BETWEEN [START] AND [END]
  GROUP BY D1.AnaId, D1.Date
) as values
WHERE values.AnaId = ana.AnaId
  AND values.Interval > ana.Interval
GROUP BY values.AnaId
ORDER BY count DESC
LIMIT 1;
```

2.4.2.8 Extraction de valeurs courantes

Ces deux requêtes ne possédant pas de paramètre, elles ne sont exécutées qu'une seule fois pour éviter d'utiliser le cache sur les requêtes – stocker leur résultats pour ne pas avoir à les réévaluer. Elles permettent de récupérer les valeurs les plus récentes pour chaque repère de la base.

R11.1 Valeurs analogiques courantes.

```
SELECT anavalues.AnaId, Value
```

```
FROM anavalues,
  (
    SELECT AnaId, max(Date) as latest
    FROM anavalues
    GROUP BY AnaId
  ) as currentdates
WHERE Date = currentdates.latest
  AND anavalues.AnaId = currentdates.AnaId
ORDER BY anavalues.AnaId;
```

R11.2 Valeurs booléennes courantes.

```
SELECT boolvalues.BoolId, Value
FROM boolvalues,
  (
    SELECT BoolId, max(Date) as latest
    FROM boolvalues
    GROUP BY BoolId
  ) as currentdates
WHERE Date = currentdates.latest
  AND boolvalues.BoolId = currentdates.BoolId
ORDER BY boolvalues.BoolId;
```

2.5 Expérimentation du benchmark

Ce benchmark a été exécuté avec le progiciel d'historisation InfoPlus.21, le SGBDR MySQL et le SGBD NoSQL Berkeley DB.

Les progiciels d'historisation sont des solutions propriétaires avec des conceptions distinctes et donc des performances différentes. Nous avons choisi l'un des plus répandus, InfoPlus.21, utilisé à EDF dans le domaine nucléaire.

Nous avons retenu MySQL pour sa facilité d'utilisation et sa pérennité avec une communauté d'utilisateurs importante, essentiels pour un usage industriel. Dans notre contexte, les tuples sont relativement petit (17 octets pour la table `anavalues`), et la majorité des colonnes sont généralement accédées. De plus la sélectivité des requêtes est faible – la plupart des tuples remplissent les critères – dans l'intervalle de temps considéré. Ces propriétés réduisent l'intérêt d'utiliser un SGBD orienté colonnes.

Enfin, nous avons choisi l'entrepôt de paires clé-valeur ordonnées Berkeley DB pour nos expérimentations. Cette catégorie de système NoSQL est particulièrement adaptée aux requêtes usuelles basées sur des intervalles de clés (*range queries*).

Optimisation de MySQL Les résultats suivants ont été obtenus avec le moteur de stockage InnoDB. MyISAM a également été testé, mais les performances se détérioraient avec l'augmentation de la quantité de données. Les résultats avec MyISAM ne sont donc pas détaillés ici.

Par défaut, InnoDB utilise un index sur sa clé primaire – ici (`AnaId`, `Date`) et (`BoolId`, `Date`). Vu les requêtes de ce benchmark, et, dans l'ensemble, les requêtes typiques à EDF sur les données historiques, cet index nous a paru efficace pour la majorité d'entre elles. Nous n'avons donc pas défini d'index supplémentaire pour ne pas ralentir les insertions.

InnoDB est un moteur de stockage transactionnel, ce qui limite ses capacités à bufferiser les insertions. En conséquence, nous avons désactivé cette fonctionnalité avec les options suivantes [16] :

```
innodb_flush_log_at_trx_commit=0
innodb_support_xa=0
innodb_doublewrite=0
```

Pour éviter l'analyse syntaxique des requêtes et les conversions de type des données, ce benchmark utilise les *prepared statements* de l'API C de MySQL pour les insertions. De plus, comme MySQL n'alloue qu'un thread par connexion, nous avons ouvert plusieurs accès parallèles (expérimentalement, la valeur optimale est de 4).

Avec leur définition en SQL, les requêtes R9 et R10 ne sont pas évaluées efficacement par MySQL – par exemple, MySQL ne divise pas l'intervalle de R9 en plusieurs intervalles plus petits (un pour chaque repère). Ce problème est réglé en utilisant des procédures stockées, détaillées en annexe A.1.

Optimisation de Berkeley DB Les capacités transactionnelles de Berkeley DB sont également minimisées pour améliorer les performances. L'option `DB_TXN_NOSYNC` permet de désactiver la synchronisation du log avec les transactions. Les curseurs d'écriture (un par repère) sont configurés pour optimiser les insertions par clé croissantes : les opérations successives tentent de se poursuivre sur la même page de la base de données. La base de données est partitionnée, avec un repère par partition. Sans cela, les insertions sont environ 60% plus lentes.

Pour chaque système, le serveur de test possède un processeur Xeon Quad Core E5405 2.0 GHz, 3 GB de RAM à 667 MHz et trois disques durs de 73 GB 10K avec un contrôleur RAID 5 de 256 MB. Pour ces tests, seul un cœur est activé a cause du manque d'optimisation de notre progiciel d'historisation pour des insertions multi-threadées.

500 000 000 tuples de données sont insérés pour chaque type – analogique et booléen – ce qui correspond à 11.5 GB sans compression (en considérant des date stockées sur 8 octets). Ces tuples sont divisés entre les 200 repères (100 pour chaque type). 1 000 000 mises à jour sont ensuite effectuées ; suivies de 1 000 requêtes R2 à R8, 100 requêtes R9 et R10, et 1 requête R11.1 et R11.2. Les paramètres des requêtes sont générés de manière à accéder, en moyenne, à 100 000 tuples pour les requêtes R2 à R8, et 10 000 000 tuples pour les requêtes R9 et R10.

2.5.1 Résultats

Le tableau 2.19 présente les résultats détaillés pour chaque système. Les capacités de traitement sont données dans la figure 2.3, en indiquant le nombre de tuples traités à la seconde.

Pour l'analyse de ces résultats, il est possible de regrouper les requêtes présentant des performances similaires par catégories. En particulier, les performances sont comparables entre les valeurs analogiques et booléennes. Pour InfoPlus.21 et Berkeley DB, on distingue quatre catégories de requêtes : les insertions (R0.1 et R0.2), les mises à jour (R1.1 et R1.2), les *range queries* (R2.1 à R10) et l'extraction des valeurs courantes (R11.1 et R11.2).

Pour MySQL, les *range queries* présentent quelques disparités. Les requêtes R2.1, R2.2, R7 et R8 retournent les valeurs brutes correspondant au résultat de la requête, ce qui représente un volume important de données. Or ces résultats impliquent une opération de conversion de type sous forme de chaînes de caractères, ce qui dégrade les performances d'environ 36% par

16. Les insertions d'InnoDB sont environ deux fois plus lentes avec le support des transactions.

TABLE 2.19: Temps d'exécution des requêtes

Type de requête (quantité)	Temps d'exécution (en s)		
	InfoPlus.21	MySQL	Berkeley DB
R0.1 (×500 M)	8 003.4	24 671.7	2 849.6
R0.2 (×500 M)	7 085.8	24 086.0	3 115.8
R1.1 (×1 M)	16 762.8	12 239.5	9 031.5
R1.2 (×1 M)	16 071.3	13 088.2	9 348.5
R2.1 (×1 k)	267.6	410.4	693.0
R2.2 (×1 k)	215.1	284.5	655.4
R3.1 (×1 k)	207.5	186.6	531.4
R3.2 (×1 k)	166.8	181.8	533.2
R4 (×1 k)	210.3	192.6	536.8
R5 (×1 k)	189.3	185.7	514.0
R6 (×1 k)	189.1	191.9	513.1
R7 (×1 k)	234.0	234.2	507.7
R8 (×1 k)	231.2	277.7	506.5
R9 (×100)	1 640.6	1 710.0	4 877.7
R10 (×100)	1 688.8	7 660.7	4 977.5
R11.1 (×1)	9.5×10^{-4}	1.15	2.75
R11.2 (×1)	2.8×10^{-4}	1.13	4.81

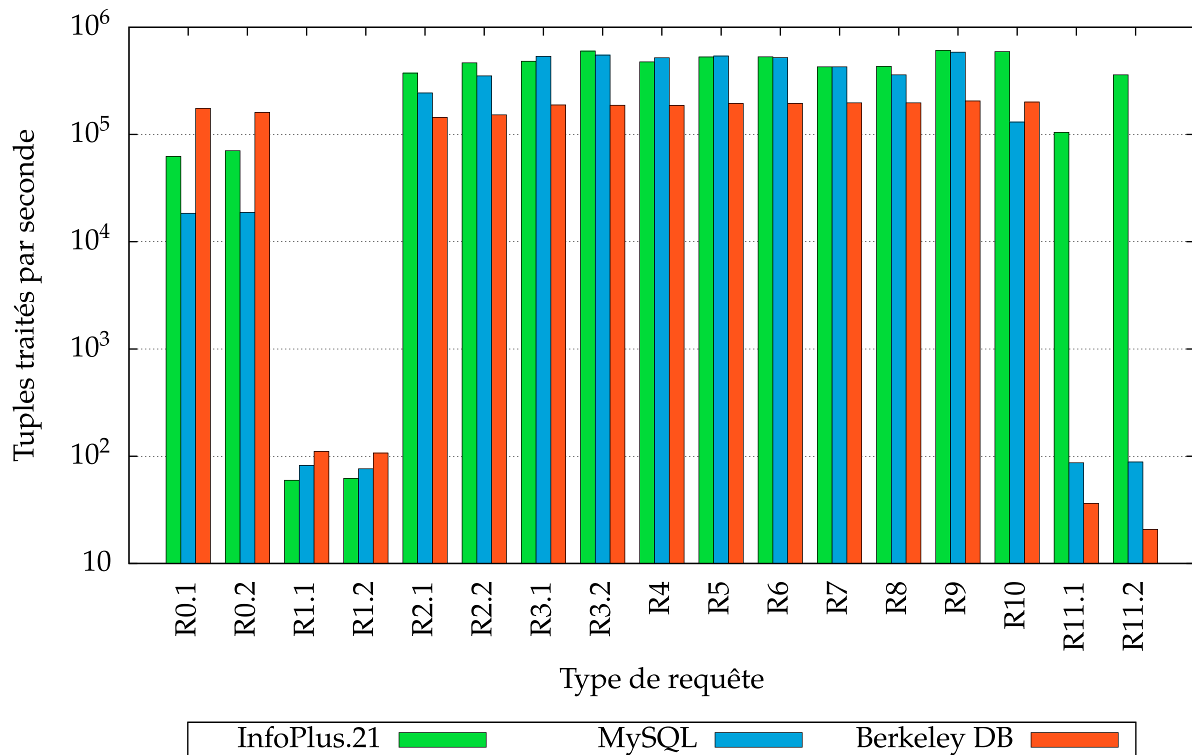


FIGURE 2.3: Capacités de traitement

TABLE 2.20: Synthèse des performances

Catégorie	Requêtes	Meilleurs résultats
Insertion	R0.1 et R0.2	Berkeley DB
Mise à jour	R1.1 et R1.2	Berkeley DB
Valeurs brutes	R2.1* et R2.2	InfoPlus.21
Agrégats	R3.1, R3.2, R4, R5 et R6	MySQL
Filtrage	R7 et R8	InfoPlus.21
Multiples séries	R9 et R10*	InfoPlus.21
Valeurs courantes	R11.1 et R11.2	InfoPlus.21

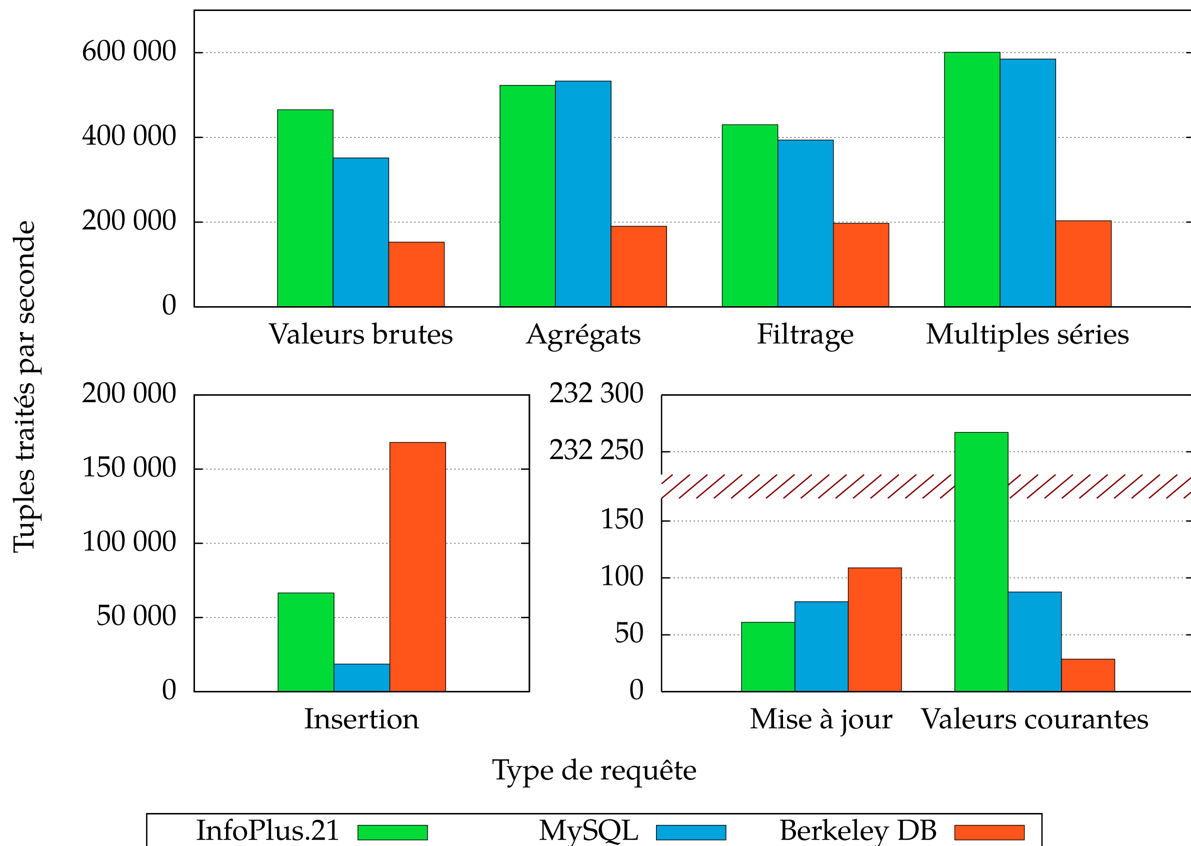


FIGURE 2.4: Capacités de traitement par catégorie

rapport aux calculs d'agrégats. Par ailleurs, la requête R10 fait appel à des procédures stockées permettant d'évaluer la période d'échantillonnage, alors que ce traitement est effectué par le benchmark pour InfoPlus.21 et Berkeley DB. Par rapport à R9, qui interroge sensiblement les mêmes données, les performances sont dégradées d'un facteur 4.5.

Différentes requêtes d'une même catégorie reportant des ratios similaires – R3, R4, R5 et R6 pour les agrégats, R7 et R8 pour le filtrage, et R9 et R10 pour l'interrogation de multiples repères – sont regroupées dans le tableau 2.20 afin de résumer ces résultats. Pour tous les systèmes, la requête R2.1 n'est pas prise en compte à cause des activités en arrière plan dues aux mise à jour, qui dégradent les performances de cette requête (en particulier pour MySQL, et dans une moindre mesure pour InfoPlus.21). La requête R10 est également ignorée parmi les résultats MySQL, à cause de mauvaises performances probablement dues à la définition des procédures

stockées utilisées pour son traitement. La figure 2.4 donne un aperçu des différences de performances en regroupant les requêtes similaires.

Comme on pouvait s'y attendre, les progiciels d'historisation gèrent les insertions efficacement par rapport aux SGBDR : InfoPlus.21 atteint 66 500 insertions par seconde (ips), soit environ 3.2 fois mieux qu'InnoDB et ses 20 500 ips. Toutefois, Berkeley DB atteint 168 000 ips, soit 2.5× mieux qu'InfoPlus.21. Ce résultat est à relativiser car Berkeley DB est utilisé en tant que librairie, sans mécanismes de communication entre processus, ce qui peut avoir un impact important sur les performances par rapport à MySQL ou InfoPlus.21.

L'extraction des valeurs courantes (R11.1 et R11.2) est le deuxième point fort prévisible des progiciels d'historisation, étant donné leur conception particulière où les valeurs les plus récentes sont conservées en mémoire. Cette opération est plus rapide de plusieurs ordres de grandeur par rapport à MySQL (×1 850) ou Berkeley DB (×6 140).

Pour les autres requêtes d'analyse [17], MySQL et InfoPlus.21 présentent des performances très proches, avec au plus de 25% de différence entre ces deux systèmes. Leurs capacités de traitement sont comprises entre 350 000 et 610 000 tuples par seconde selon les requêtes ; tandis que Berkeley DB, de part la simplicité de son interface d'accès, présente des performances stables, mais sensiblement moins bonnes, autour de 190 000 tuples par seconde.

2.6 Synthèse

Dans ce chapitre, nous avons dans un premier temps mis l'accent sur l'historisation de données comme un segment du marché des SGBD avec un besoin industriel significatif. Les progiciels d'historisation, solutions dédiées à ce cas d'utilisation, se distinguent des autres catégories de SGBD par des fonctionnalités métier liées à la nature des données stockées (compression, interpolation, clients spécialisés, etc.) et au contexte d'utilisation (protocoles de communication, configuration "clé en main" adaptée, etc.).

Pour l'évaluation de leurs performances, la comparaison avec d'autres SGBD s'est basée sur un benchmark défini à partir d'un cas d'utilisation significatif au sein d'EDF.

À la vue de ces premiers résultats expérimentaux, les performances globales de chaque système sont du même ordre de grandeur : elles ne proscrivent pas l'utilisation de SGBDR ou de systèmes NoSQL pour l'historisation de données industrielles, en faisant abstraction de certaines fonctionnalités métiers fournies par les progiciels d'historisation. Cependant, avant d'envisager ces solutions en production, des études supplémentaires avec des charges plus réalistes sont nécessaires pour attester de leur utilisabilité.

17. R2.1 et R10 exclues.

CHAPITRE 3

LES MÉMOIRES FLASH

Plan du chapitre

3.1 Fonctionnement et caractéristiques des mémoires flash **35**
3.1.1 Historique 35
3.1.2 Cellules 36
3.1.3 Opérations élémentaires 37
3.1.4 Quantité d'information par cellule 38
3.1.5 Taux d'erreur 39
3.1.6 Autres caractéristiques 41
3.2 Flash Translation Layer **42**
3.2.1 Principes de la FTL 42
3.2.2 Types de FTL 46
3.2.3 Systèmes de fichiers 50
3.3 Performances **50**
3.4 Optimisation du système d'exploitation pour les mémoires flash **52**
3.4.1 Système de fichiers 52
3.4.2 Ordonnanceur d'entrées/sorties 52
3.4.3 Gestion des caches 52
3.5 Optimisation des SGBD pour les mémoires flash **55**
3.5.1 Disposition des données 55
3.5.2 Indexation 56
3.6 Amélioration des écritures aléatoires **58**
3.6.1 Localité spatiale des écritures 58
3.6.2 Regroupement des écritures aléatoires 60
3.6.3 Modèle 62
3.6.4 Résultats 64
3.7 Synthèse **66**

Dans ce chapitre, nous présentons les mémoires flash et un état de l'art des techniques d'optimisation applicables à leur utilisation par des SGBD. Nous proposons ensuite une optimisation visant les types de mémoires flash utilisées pour l'historisation de données à EDF.

Pour archiver à court terme les données des centrales hydrauliques, l'IGCBox stocke ses données sur une mémoire flash de faible capacité (quelques Go). Avec ce type de mémoire économique (bas/milieu de gamme), MySQL présente des performances décevantes. Pourtant, de nombreuses optimisations sont développées dans la littérature. Dans ce chapitre, nous détaillons dans une première partie le fonctionnement des mémoires flash, afin de réaliser un état de l'art des techniques applicables aux SGBD. Nous présentons ensuite notre proposition, permettant d'améliorer les performances des écritures aléatoires sur mémoires flash, type d'accès identifié comme peu performant pour celles intégrées à l'IGCBox.

3.1 Fonctionnement et caractéristiques des mémoires flash

3.1.1 Historique

Les mémoires flash sont des mémoires basées sur les semi-conducteurs. Elles dérivent des *mask-programmed ROM* (Read-Only Memory), dont les données sont encodées physiquement dans le circuit ; celles-ci ne sont donc programmables qu'à leur fabrication. Les PROM (Programmable Read-Only Memory, inventées en 1956) autorisent la programmation lors d'une phase différente de la fabrication. Cette opération est cependant irréversible. L'opération inverse de la programmation, l'effacement, apparaît avec la génération des EPROM (Erasable Programmable Read-Only Memory, inventées en 1971), qui peuvent être remises à zéro par une exposition à un rayonnement UV. Cette opération est simplifié avec l'arrivée des EEPROM (Electrically Erasable Programmable Read-Only Memory, inventées en 1978), qui sont effaçables électriquement.

Les mémoires flash sont également des mémoires de type EEPROM dans le sens où elles sont effaçables électriquement. Cependant, le terme EEPROM est généralement associé aux mémoires dont les trois opérations (lecture, programmation et effacement) sont effectuées par bit ou par octet. Les mémoires flash NOR (commercialisées en 1988) mutualisent les circuits liés à l'effacement par regroupement de cellules, appelés blocs, ce qui permet de réduire la taille des cellules et donc le coût de fabrication. Les mémoires flash NAND (commercialisées en 1995) diminuent à nouveau ce coût en mutualisant cette fois les circuits liés aux opérations de lecture et d'écriture.

Pour les mémoires flash NOR, les accès aléatoires aux octets en lecture et en écriture permettent l'exécution d'un programme sans avoir à le copier préalablement en RAM, on parle de XIP (eXecute In Place). En contrepartie, l'opération d'écriture est particulièrement lente. Typiquement, cette opération est mille fois plus lente que la lecture [Micheloni *et al.*, 2010]. Les mémoires NOR sont habituellement utilisées en remplacement des PROM et EPROM, pour stocker du code exécutable, principalement lu. L'accès aléatoire implique la présence de contact pour chaque cellule, ce qui est un facteur limitant pour la miniaturisation de ce type de mémoire.

Les mémoires flash NAND permettent un accès en lecture et en écriture par regroupements de cellules appelés pages. Cela permet d'obtenir des mémoires plus denses, d'où un meilleur coût par bit et des capacités plus importantes par rapport aux flash NOR. De part leur forte densité et leur faible coût, les flash NAND sont particulièrement adaptés au stockage de données.

Vu notre contexte, seules les mémoires flash de type NAND seront considérées, de part leur prépondérance dans le domaine des systèmes de gestion de bases de données et leurs caractéristiques adaptées à ce cas d'application.

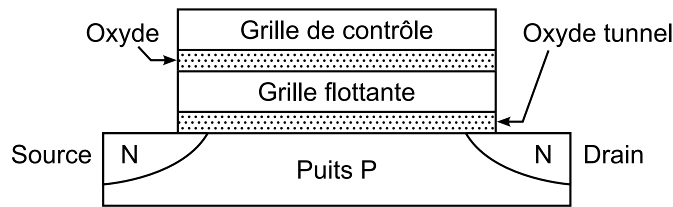


FIGURE 3.1: Transistor à grille flottante (Brewer et Gill, 2008)

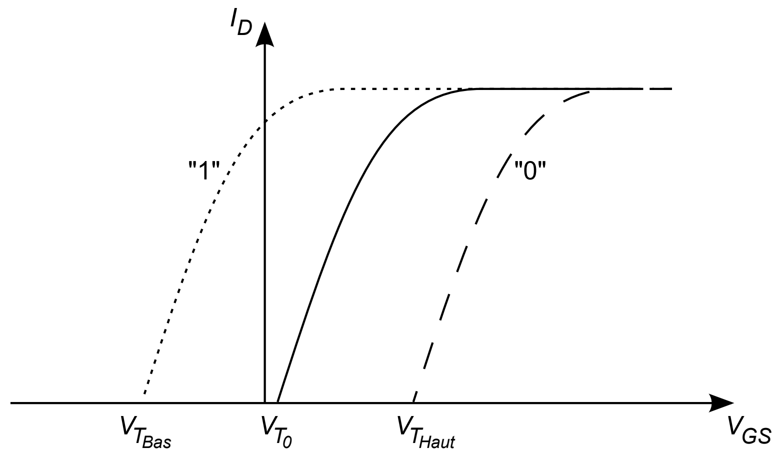


FIGURE 3.2: Seuils de tension d'une cellule flash (Postel-Pellerin, 2008)

3.1.2 Cellules

Les mémoires flash sont des mémoires *solid-state*, c'est à dire constituées uniquement de matériaux solides capables de stocker une charge électrique de manière persistante. Dans le cas des mémoires flash, cette charge électrique est conservée par des transistors à grille flottante, et représente l'information – ou donnée – stockée par les cellules.

La technologie à grille flottante consiste à ajouter une seconde grille en matériau conducteur ou semi-conducteur entre la grille de contrôle et le canal d'un transistor MOS traditionnel, illustré par la figure 3.1. La tension de seuil du transistor est alors dépendante des charges isolées dans cette grille flottante. Une charge positive diminue la tension de seuil du transistor, tandis qu'une charge négative augmente cette tension de seuil, ce qui permet de définir deux états $V_{T_{Bas}}$ et $V_{T_{Haut}}$ de part et d'autre de la tension de seuil naturelle V_{T_0}, représentant respectivement une valeur "1" ou "0". Ces trois états sont présentés sur la figure 3.2, où I_D représente l'intensité du courant circulant entre la source et le drain, et V_{GS} la différence de potentiel entre la grille et la source.

Pour lire l'information stockée par cette cellule, il est possible de distinguer ces deux états en mesurant par exemple l'intensité du courant entre la source et le drain pour une grille de contrôle au potentiel $V_{GS} = 0V$, voir la figure 3.3.

Les mémoires flash de type NAND sont constituées d'un grand nombre de ces cellules, organisées en blocs de $m \times n$ cellules, où m représente le nombre de lignes – ou *pages* – par exemple [1] 64, et n le nombre de colonnes – ou *strings* – par exemple 16 896. La figure 3.4 illustre cette organisation. Les cellules d'une page partagent leur ligne de mot – ou *wordline* (WL) – qui

1. Mémoire flash SLC 2 Gb TC58NVG1S3CTA00

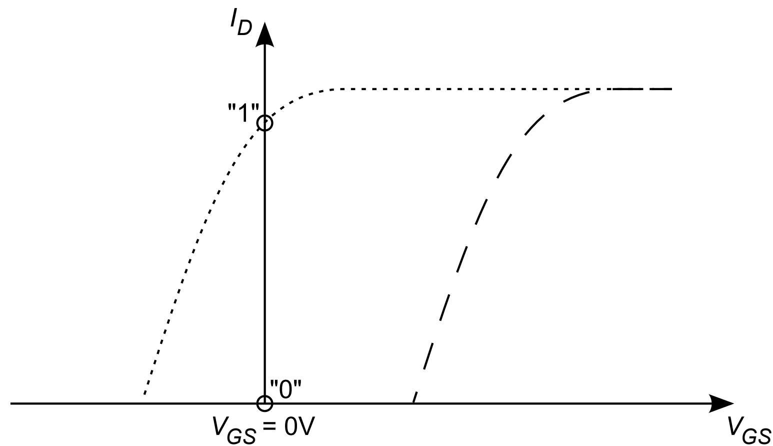


FIGURE 3.3: Lecture d'une cellule flash (Postel-Pellerin, 2008)

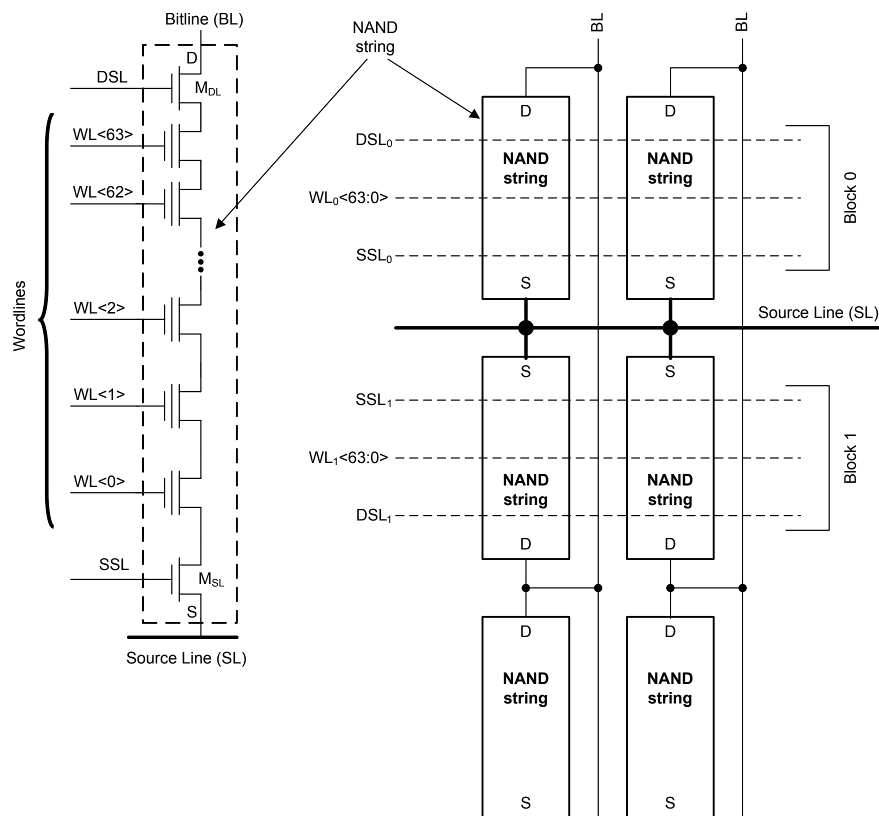


FIGURE 3.4: Organisation des cellules d'une flash NAND (Micheloni *et al.*, 2010)

règle le potentiel de leur grille de contrôle, tandis que les cellules d'un *string* sont connectées en série à la ligne de bit – ou *bitline* (BL) – par leur canal drain-source.

3.1.3 Opérations élémentaires

En fonction des tensions appliquées aux bornes électriques de cette matrice de cellules, les mémoires flash distinguent trois opérations élémentaires : la lecture, la programmation – ou écriture – et l'effacement des données.

Une opération de lecture consiste à mettre la grille de contrôle au potentiel 0V et à appliquer une faible tension $V_{DS} \approx 1V$ entre le drain et la source. Si la grille flottante ne contient pas de charge, le courant passe, indiquant un "1" logique. Dans le cas contraire, le courant ne passe pas car le transistor possède un seuil plus élevé que la tension V_{DS}, indiquant un "0" logique. Pour mesurer l'intensité du courant circulant entre la source et le drain I_D, les autres cellules du bloc doivent former un canal conducteur, pour cela, les grilles de contrôle des pages non sélectionnées (les autres lignes de mot du bloc) sont mises au potentiel $V_{pass} \approx 8V$ pour laisser passer le courant quelle que soit la charge de leur grille flottante.

Pour modifier la charge d'une grille flottante, il faut que les électrons puissent passer la barrière de potentiel entre la grille flottante et le canal drain-source. Pour les mémoires flash de type NAND, l'effet tunnel Fowler-Nordheim permet la programmation (insertion d'électrons dans la grille) et l'effacement (retrait d'électrons de la grille) des cellules [Micheloni *et al.*, 2010].

Lors d'une opération de programmation, un champ électrique important est nécessaire pour que les électrons traversent l'oxyde tunnel. Pour cela, la grille de contrôle est mise à un haut potentiel $V_{prog} \approx 20V$ tandis que le drain est mis au potentiel 0V. La différence de potentiel entraîne un transfert d'électrons du canal vers la grille flottante. Toutes les cellules d'une page partagent la même ligne de mot. Pour ne pas programmer les cellules devant rester à l'état effacé ("1" logique), les lignes de bit correspondantes sont mises au potentiel $V_{CC} \approx 3.3V$ pour inhiber l'effet tunnel. Tout comme la lecture, les autres cellules du bloc doivent former un canal conducteur, leurs lignes de mot sont mises au potentiel $V_{pass} \approx 8V$.

Pour contrôler la quantité de charge insérée dans la grille flottante, l'opération d'écriture est effectuée incrémentalement, avec une phase de vérification (lecture des seuils) après chaque étape. Transférer cette charge par petits incréments améliore également la durée de vie de la mémoire, car l'exposition continue de l'oxyde tunnel à une tension élevée présente un risque de dégradation.

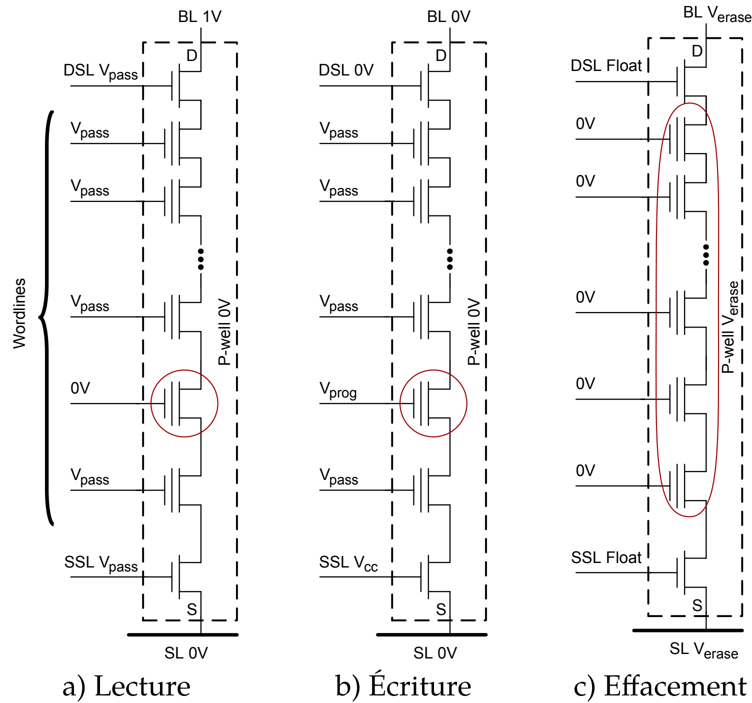
L'effacement est effectué en mettant le puits P (P-well) à un haut potentiel $V_{erase} \approx 20V$, tout en maintenant les lignes de mot du bloc à effacer au potentiel $0V$. Cette différence de potentiel entraîne un transfert d'électrons de la grille flottante vers le puits. Le puits P étant partagé par l'ensemble des cellules du bloc, celles-ci retournent alors toutes à l'état effacé ("1" logique).

Ces trois mécanismes (lecture, écriture et effacement) sont effectués en parallèle sur des ensembles de cellule : les lectures et écritures concernent des pages (typ. 528, 2112 ou 4224 octets), tandis que les effacements s'appliquent aux blocs (typ. 32 ou 64 pages). La figure 3.5 illustre ces granularités et les tensions appliquées aux cellules lors des différentes opérations.

3.1.4 Quantité d'information par cellule

Les mémoires flash distinguent la valeur d'un bit selon la présence ou non de charge dans la grille flottante. En augmentant le nombre de seuils de charge, plus d'un bit peut être stockés par cellule. Deux bits par cellule correspondent à quatre niveaux de seuils représentant les chaînes 11, 10, 01 et 00. On parle alors de flash MLC (Multi Level Cell), en opposition aux flash SLC (Single Level Cell). Plus de deux bits peuvent être stockés par une cellule, mais le nombre de seuils augmente exponentiellement (2^n niveaux pour n bits). La figure 3.6 illustre la définition des différents états pour une flash MLC par rapport à des points de référence (seuils) R1, R2 et R3.

Stocker plusieurs bits par cellule permet d'augmenter la capacité de la mémoire sans augmenter la complexité de fabrication au niveau des cellules. En contrepartie, l'insertion d'une charge dans la grille flottante doit être précise pour que les différents niveaux ne se recouvrent pas. La lecture doit également détecter des variations de tension plus fines. Avec l'augmenta-



a) Lecture b) Écriture c) Effacement

Figure 3.5: Tensions appliquées aux cellules lors des différentes opérations (Micheloni *et al.*, 2010)

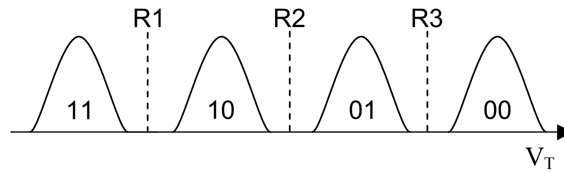


Figure 3.6: Distribution de charge pour une mémoire flash MLC

tion du nombre de niveaux, les durées de lecture et d'écriture augmentent : les lectures des flash MLC sont jusqu'à trois fois plus lentes, et les écritures jusqu'à quatre fois plus lentes que pour les flash SLC [Chang, 2008] ; ces opérations traitent cependant deux fois plus de données. Le choix de la technologie est donc un compromis entre la capacité et les performances de la mémoire. Actuellement, les mémoires flash de type SLC sont généralement destinées à un usage professionnel, tandis que les MLC ciblent un marché plus large, incluant professionnels comme particuliers.

3.1.5 Taux d'erreur

Les mémoires flash ne sont pas intrinsèquement dépourvues d'erreurs, plusieurs mécanismes mènent à des variations des seuils de tension des cellules, et donc à des erreurs sur les données stockées [Cooke, 2007; Grupp *et al.*, 2009; Mielke *et al.*, 2008; Postel-Pellerin, 2008] Ces perturbations peuvent être temporaires ou permanentes, mais entraînent toutes une augmentation progressive du taux d'erreur, éventuellement jusqu'à atteindre des seuils incompatibles avec le contexte applicatif (taux d'erreur cible). À cause de leurs niveaux de charge plus fins, les mémoires de type MLC sont globalement plus sensibles à ces erreurs.

L'effet tunnel Fowler-Nordheim est non uniforme et erratique pour des cellules de faible

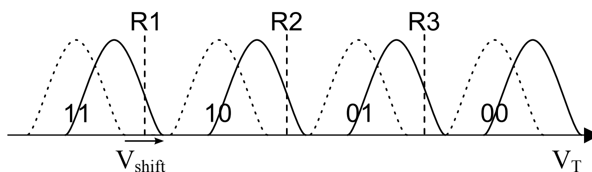


FIGURE 3.7: Décalages des niveaux de charge dus aux lectures et écritures (MLC)

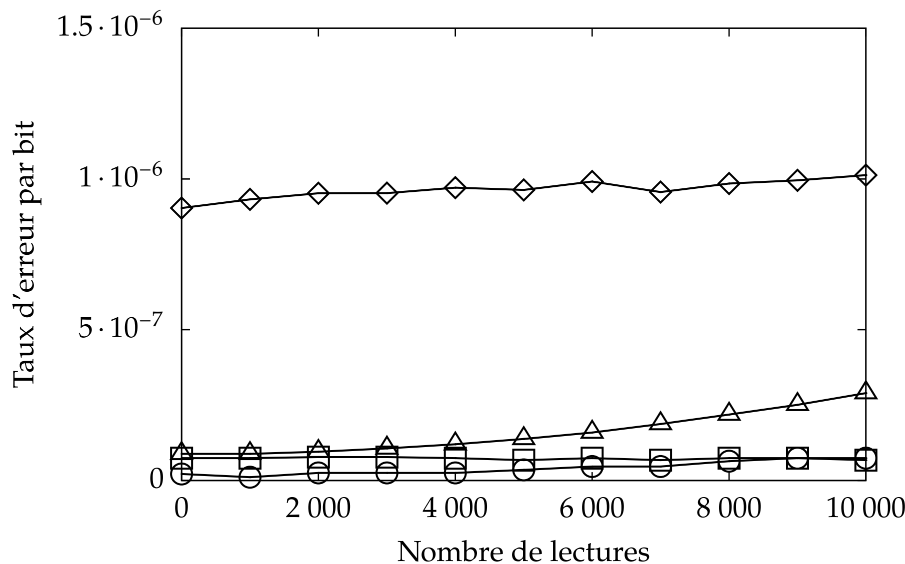


FIGURE 3.8: Perturbations dues aux lectures pour quatre mémoires MLC (Mielke *et al.*, 2008)

dimension en raison de la faible quantité d'électrons impliqués (bruit quantique, bruit télégraphique) et des imperfections du composant. Ces défauts impliquent un taux d'erreur initial non nul, dès l'effacement ou la programmation de la cellule.

Perturbations dues aux lectures et écritures Ce taux d'erreur est accentué par les opérations effectuées sur des cellules voisines. En effet, lors de la lecture ou de l'écriture d'une page, certaines cellules du même bloc sont soumises à des tensions suffisamment élevées pour apparaître faiblement programmées. Ces effets secondaires entraînent un décalage de la tension de seuil d'une grandeur V_{shift} qui peut, dans certains cas, impliquer un changement de la donnée stockée, comme illustré figure 3.7. Ces altérations sont temporaires car la mémoire n'est pas endommagée et un effacement rétablit les cellules à un état effacé normal. Cependant, pour minimiser ces perturbations, l'écriture partielle d'une page est à minimiser, et les pages d'un bloc doivent être écrites séquentiellement (de la WL 0 à la WL 63 dans notre exemple) ; de plus, le nombre de lecture par bloc est limité, si ce seuil est atteint, le bloc doit être effacé et recopié pour rétablir des niveaux de seuils corrects. La figure 3.8 présente l'augmentation du taux d'erreur avec le nombre de lectures.

Endurance Lors d'une opération d'effacement, des charges peuvent rester enfermées dans l'oxyde tunnel de la cellule, ce qui augmente progressivement le taux d'erreur de la mémoire. Cette dégradation est permanente. À cause de ce phénomène, la durée de vie des mémoires flash est limitée en nombre d'effacements de blocs. Typiquement, les mémoires de type MLC

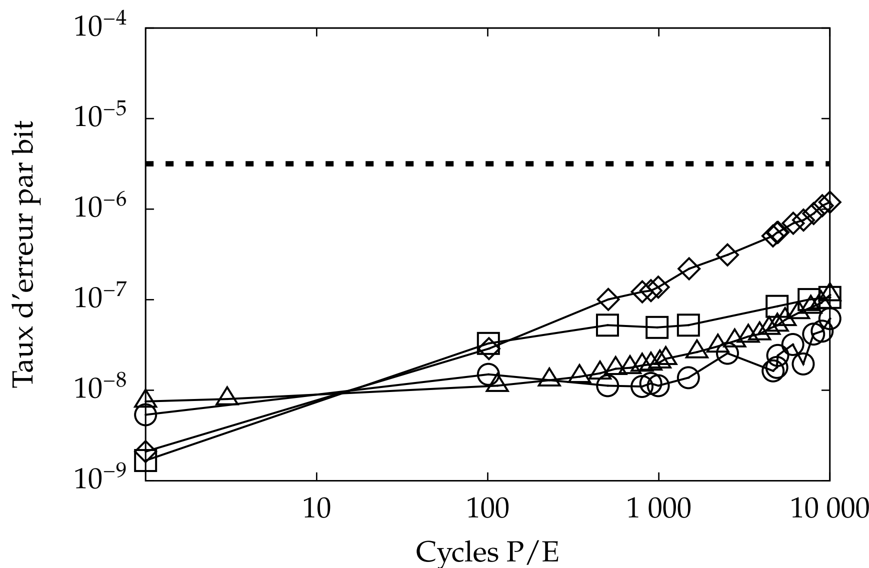


FIGURE 3.9: Dégradation du taux d'erreur en fonction du nombre de cycles Programmation/Effacement pour quatre mémoires MLC (Mielke *et al.*, 2008)

peuvent endurer 10^4 cycles de programmation/effacement, tandis que les flash SLC supportent entre 10^5 et 10^6 cycles. Cette dégradation progressive de la mémoire est présentée figure 3.9.

Rétention Les charges contenues dans la grille flottante ont tendance à se dissiper. Pour cette raison, la durée de rétention des données d'une mémoire flash est limitée. De plus, la rétention des données à tendance à se dégrader avec la détérioration de l'oxyde tunnel due aux cycles programmation/effacement. La plupart des mémoires flash garantissent cependant une durée de rétention d'au moins dix ans ; et une réécriture des données permet de rétablir des niveaux de charge corrects. La figure 3.10 montre l'augmentation du taux d'erreur par bit en fonction de la durée de rétention à température ambiante (25°C). Cette dégradation est accélérée avec l'augmentation de la température, cette propriété est exploitée pour vieillir artificiellement ces mémoires [Postel-Pellerin, 2008].

3.1.6 Autres caractéristiques

Contrairement aux disques durs, dont les défaillances sont événementielles, la limitation de la durée de vie des mémoires flash présente l'avantage d'être généralement prévisible [Mielke *et al.*, 2008] ; en particulier, certains périphériques permettent la récupération du nombre d'effacements effectués. Étant donné l'absence de parties mécaniques dans les mémoires flash, il est également plus facile de concevoir un périphérique résistant à des conditions environnementales extrêmes en terme de chocs, vibrations, variations de températures, présence de particules (poussières), humidité et pression. Les mémoires flash possèdent aussi une bonne résistance aux rayonnements [Irom et Nguyen, 2008]. En plus de leur robustesse, les mémoires flash présentent des avantages par rapport aux disques durs au niveau de la densité (bit/cm^3 et bit/kg) et de la consommation d'énergie. Ces caractéristiques les rendent particulièrement adaptées aux environnements mobiles et embarqués.

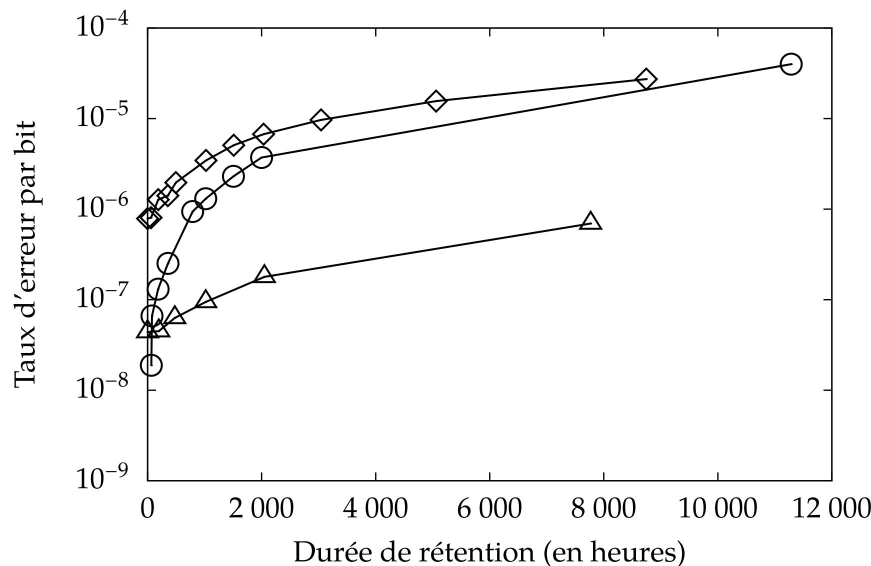


FIGURE 3.10: Dégradation du taux d'erreur avec la durée de rétention pour trois mémoires MLC (Mielke *et al.*, 2008)

3.2 Flash Translation Layer

Les puces flash NAND possèdent donc un fonctionnement défini avec précision, mais complexe. Pour prendre en charge cette complexité, et fournir un interface d'accès simplifiée – compatible avec celle des disques durs –, les mémoires flash intègrent fréquemment une sur-couche logicielle : la Flash Translation Layer (FTL). Cette interface d'accès permet uniquement la lecture et l'écriture de « secteurs » (cette unité est généralement appelée *block* – comme dans *block-device* ; l'appellation secteur permet de la distinguer de l'unité d'effacement des mémoires flash) ; la FTL doit donc gérer les codes de correction d'erreurs, la répartition de l'usure et les effacements.

De part les choix – détaillées dans la suite de cette section – effectués à ce niveau, la FTL possède un impact important sur les performances, et est donc un composant fondamental des mémoires flash.

3.2.1 Principes de la FTL

3.2.1.1 Métadonnées

Pour gérer ces fonctionnalités, des métadonnées doivent être enregistrées sur la mémoire flash. Pour cela, les pages des flash NAND possèdent une section dédiée. Par exemple, une page de 2112 octets est composée de 2048 octets de données, et de 64 octets réservés aux métadonnées, telles que :

- l'adresse logique de la page (voir section 3.2.1.4),
- une indication de l'usure du bloc, comme le nombre d'effacement,
- une indication des blocs défaillants,
- un code de correction d'erreur (voir section 3.2.1.2),
- une indication de version pour distinguer la plus récente de deux pages possédant la même adresse logique.

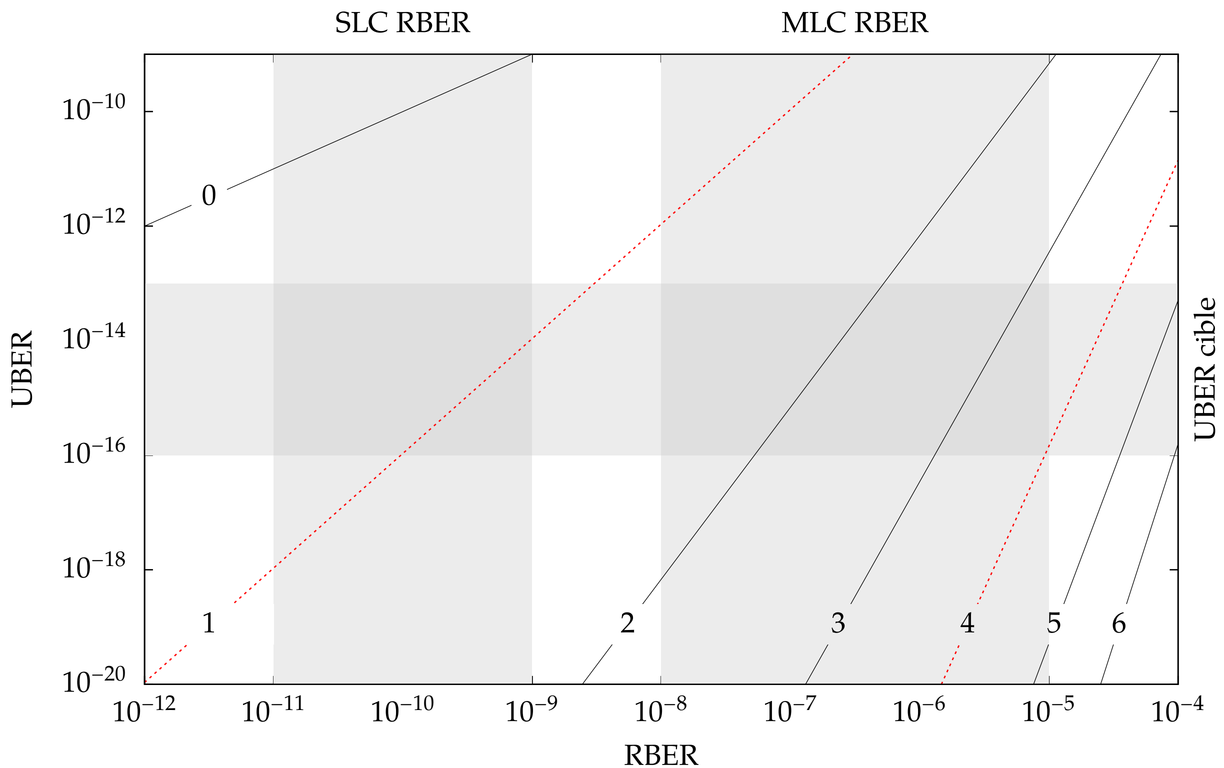


FIGURE 3.11: Impact du code correcteur pour plusieurs seuils de correction (Micheloni *et al.*, 2010)

3.2.1.2 Codes de correction d'erreurs

Le taux d'erreur – ou BER (Bit Error Rate) – des mémoires flash est trop important pour la plupart des applications. En conséquence, un code de correction d'erreurs est ajouté aux données pour en améliorer la fiabilité.

Un code de correction d'erreurs stocke de l'information redondante pour pouvoir corriger un certain nombre d'erreurs (son seuil de correction) pouvant apparaître lors de la récupération des données. On distingue alors le taux d'erreur avant correction – RBER (Raw Bit Error Rate) – et après correction – UBER (Uncorrectable Bit Error Rate). Le choix du code à appliquer passe par un compromis entre le surcoût en espace mémoire de l'information redondante, son coût de calcul pour l'encodage et le décodage, et sa robustesse. De nombreuses techniques applicables aux mémoires flash sont présentées dans [Micheloni *et al.*, 2008].

La figure 3.11 présente l'impact du code de correction d'erreur sur le taux d'erreur pour plusieurs seuils de correction. Les UBER des mémoires flash sont typiquement spécifiés entre 10^{-13} et 10^{-16}, ce qui correspond généralement à un code capable de corriger une erreur pour les mémoires SLC (RBER entre 10^{-9} et 10^{-11}), et quatre erreurs pour les mémoires MLC (RBER entre 10^{-5} et 10^{-8}) [Cooke, 2007; Mielke *et al.*, 2008]. Le stockage de plus de deux bits par cellule nécessite des codes de correction d'erreurs avec des seuils de correction plus importants.

En fonction de l'UBER cible, le dimensionnement du seuil de correction définit un RBER maximum – par exemple, un UBER de 10^{-15} et un code pouvant corriger deux erreurs implique un RBER maximum de 10^{-7}. Ce RBER est lié aux limites présentées dans la section 3.1.5, comme le nombre d'effacements maximum ou la durée de rétention. Le dimensionnement du code correcteur influence donc directement l'endurance, la rétention et la résistance aux perturbations des mémoires flash.

3.2.1.3 Répartition de l'usure

Le taux d'erreur des mémoires flash se dégrade progressivement avec les cycles de programmation/effacement, jusqu'à éventuellement atteindre le seuil de correction du code correcteur. En conséquence, la durée de vie de la mémoire est limitée par le nombre d'effacement. Pour éviter que certains blocs ne se dégradent plus rapidement que d'autres, il est nécessaire de répartir les effacements sur tous les blocs disponibles. Cette technique, appelée *wear-leveling*, maintient un décompte du nombre d'effacements par bloc. Lors de réécritures, les blocs les moins utilisés sont désignés par l'algorithme en remplacement des blocs significativement plus dégradés que la moyenne.

Il existe plusieurs algorithmes de répartition de l'usure ; chacun ajoute de la complexité, mais améliore la durée de vie de la mémoire. On distingue deux principales catégories : la répartition statique (*static wear-leveling*) et la répartition dynamique (*dynamic wear-leveling*).

Répartition statique de l'usure La répartition statique utilise tous les blocs disponibles pour répartir équitablement les écritures. Cette méthode consiste à conserver un compteur d'effacements pour chaque bloc et à distribuer les écritures en sélectionnant le bloc avec la valeur la plus faible. Les données statiques (ie. les blocs dont le compteur n'augmente pas) sont déplacées vers des blocs ayant un nombre élevé d'effacements. Bien que déplacer les données statiques réduise les performances, cette méthode permet de maximiser la durée de vie de la mémoire. [Chang *et al.*, 2007] propose par exemple un algorithme de répartition de l'usure statique permettant, lorsque la mémoire totalise un certain nombre d'effacements, de déplacer les données qui n'ont jamais été effacée pendant cette période.

Répartition dynamique de l'usure La répartition dynamique utilise uniquement les blocs libres pour répartir les écritures. Comme cet algorithme n'induit pas d'effacements supplémentaires, il ne diminue pas significativement les performances. Cependant, il nécessite une réserve de blocs libres pour être efficace. Si la mémoire contient beaucoup de données statiques, la durée de vie est diminuée car les écritures sont réparties sur peu de blocs.

Pour une mémoire flash possédant une interface d'accès similaire aux disques durs, la notion de secteur libre n'existe pas. L'espace mémoire annoncé au système est donc toujours entièrement occupée : la mémoire flash n'a aucun moyen de savoir si les données qu'elle contient sont obsolètes. Ces mémoires flash possèdent donc des blocs de rechange – typiquement de l'ordre de quelques pourcent de la capacité totale, non inclus dans la capacité affichée de la mémoire – pour rediriger les écritures et remplacer les blocs défaillants.

Généralement, les FTL combinent les deux types de répartition de l'usure. Les allocations – sélection du bloc à écrire – font appel à un algorithme dynamique pour maximiser les performances, tandis qu'une répartition statique est déclenchée périodiquement pour déplacer les données statiques [Gal et Toledo, 2005]. La figure 3.12 illustre l'impact de ces algorithmes sur la répartition de l'usure : l'absence de répartition concentre les effacements sur les blocs accédés, un algorithme dynamique permet de répartir ces effacements sur les blocs libres, tandis qu'un algorithme statique efface uniformément tous les blocs de la mémoire.

3.2.1.4 Redirection des écritures

Avant d'écrire une page, celle-ci doit être effacée (cf. section 3.1). Comme l'effacement n'est possible que sur un bloc complet, il est nécessaire de copier intégralement le contenu du bloc. La modification d'une seule page peut donc se révéler aussi coûteuse que la modification de tout le bloc – soit typiquement 32 ou 64 pages. A cause de cela, l'effacement est généralement

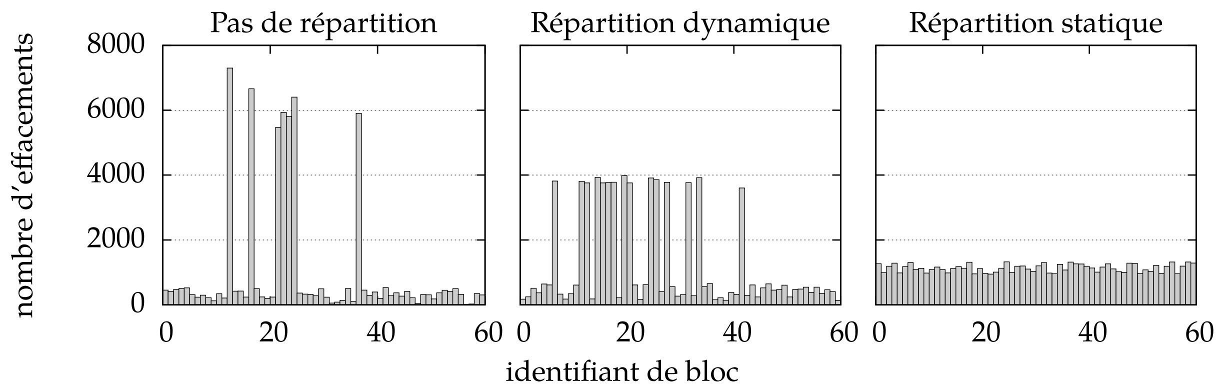


FIGURE 3.12: Techniques de wear-leveling (75% de données statiques)

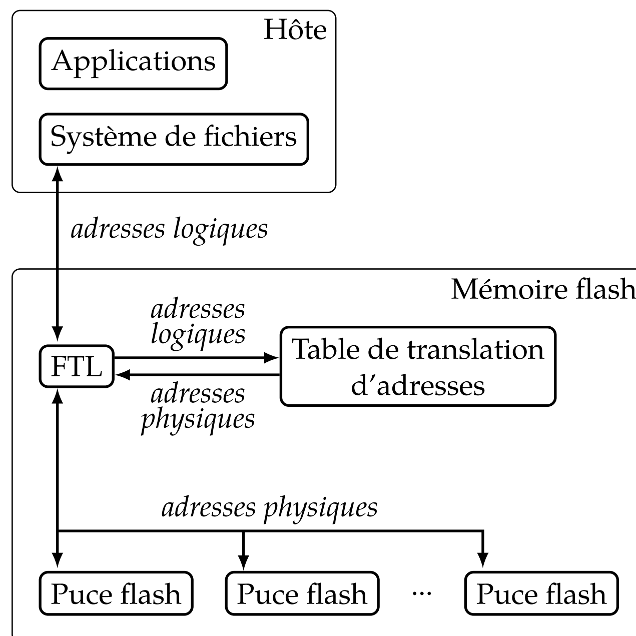


FIGURE 3.13: Translation d'adresses au niveau de la FTL

différé en écrivant les données dans une page libre (préalablement effacée), tandis que la version antérieure de cette page est marquée comme invalide. Une page invalide est une page dont les données ne sont plus à jour, et en attente d'être effacée. La réclamation de l'espace occupé par les pages invalidées fait appel à un mécanisme de ramasse-miettes.

Les FTL maintiennent à jour une table de translation d'adresses pour conserver la correspondance entre les adresses logiques, échangées avec l'hôte, et les adresses physiques, désignant la position réelle des données dans la ou les puces flash. La figure 3.13 indique les différents types d'adresses utilisés dans les communications entre l'hôte, la FTL et les puces flash.

La granularité – finesse de la conversion, ou *mapping* – de cette table de translation d'adresse conditionne les choix des emplacements physiques des données par la FTL et influe fortement sur les performances de la mémoire flash.

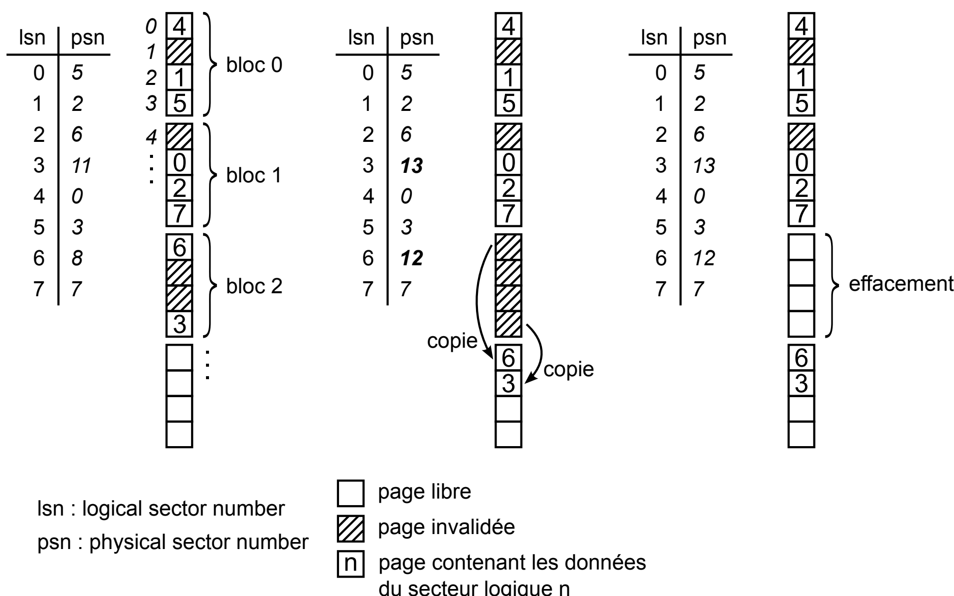


FIGURE 3.14: Réclamation d'un bloc pour les page-mapping FTL

3.2.2 Types de FTL

3.2.2.1 Translation d'adresse par page

Les FTL avec une conversion d'adresse par page [Birrell *et al.*, 2007] – *page-mapping FTL* – possèdent une table de translation d'adresse capable d'associer chaque page logique à une page physique. Lors d'une écriture, la FTL alloue n'importe quelle page libre pour stocker les données, invalide la version antérieure et met à jour sa table de translation d'adresse. Lorsque le nombre de pages libres devient insuffisant, la FTL fait appel à un mécanisme de ramasse-miettes pour réclamer l'espace occupé par les données obsolètes.

Pour cette réclamation, il existe idéalement dans la mémoire un ou plusieurs blocs contenant uniquement des pages invalidées. Ces blocs sont alors effacés pour que leurs pages soient à nouveau utilisables pour rediriger les prochaines écritures. Cependant, il est possible que tous les blocs de la mémoire possèdent au moins une page valide. Dans ce cas, l'algorithme de ramasse-miettes désigne le bloc contenant le plus de données obsolètes. Les pages valides de ce bloc doivent être copiées dans des emplacements libres avant de pouvoir l'effacer. Ce mécanisme est illustré figure 3.14, pour la réclamation du bloc 2, i.e. le bloc avec le plus de données invalides. Dans cet exemple, la réclamation modifie les adresses physiques des secteurs logiques 3 et 6 ; la table de translation d'adresses est donc mise à jour pour référencer ces nouveaux emplacements (le secteur logique 6 est déplacé de l'adresse physique 8 à 12 et le secteur logique 3 de 11 à 13).

La finesse de cette granularité permet une grande flexibilité dans la redirection des écritures, cependant le coût en RAM de la table de translation d'adresse est généralement trop important pour une implémentation embarquée dans la mémoire flash. Par exemple, pour une mémoire flash de 4 Go [2], cette table occupe 2.5 Mo de RAM. Ce coût augmente en $O\big(n \cdot \log(n)\big)$ avec n le nombre de pages dans la mémoire.

2. Mémoire flash NAND TH58DVG5S0ETA20, avec 64 pages de 4096 octets par bloc

3.2.2.2 Translation d'adresse par bloc

De manière similaire, les FTL avec une conversion d'adresse par bloc – *block-mapping FTL* – possèdent une table de translation d'adresses capable d'associer des blocs logiques à des blocs physiques. Comme pour la translation d'adresse par page, le coût de cette table reste en $O\big(n \cdot \log(n)\big)$, mais avec n le nombre de blocs dans la mémoire. Cela représente généralement deux ordres de grandeur de moins que pour les page-mapping FTL. Pour l'exemple précédant, ce coût descend à 28 Ko – pour référencer les 16 384 blocs de la mémoire, au lieu des 1 048 576 pages.

Le principe de ces FTL est que la position de la page au sein d'un bloc – *offset* – reste fixe. Avec cette contrainte, l'écriture d'une page entraîne nécessairement la copie des autres pages du bloc, ce qui dégrade fortement les performances. Plusieurs propositions de FTL hybrides permettent d'atténuer cet effet.

3.2.2.3 FTL hybrides

Les FTL hybrides sont basées sur une translation d'adresses par bloc. Pour cela, l'adresse logique du secteur est généralement scindée en deux parties : le préfixe constitue l'adresse logique associée au bloc, tandis que le suffixe représente la position de la page au sein du bloc. Par exemple, pour des blocs de 64 pages, le secteur 140 fait partie du bloc logique 2 ($\lfloor 140/64 \rfloor = 2$) et la position de la page dans ce bloc est 12 ($140 \equiv 12 \mod 64$). Comme pour une translation d'adresse par bloc, l'adresse logique du bloc est associé à une adresse physique. Cependant, une seconde translation d'adresse peut être effectuée au niveau de la page pour rediriger les écritures dans des blocs réservés à cet effet, appelés blocs journalisés – *log blocks* [Chung *et al.*, 2009]. La proportion de l'espace réservé aux blocs journalisés est généralement faible, pour limiter l'utilisation de la RAM (translation par page) et le surcoût au niveau de la mémoire flash (ces blocs ne sont pas adressables).

Block Associative Sector Translation (BAST) BAST [Kim *et al.*, 2002] est une FTL hybride basée sur une translation par bloc. Lors d'une écriture, un bloc journalisé est alloué depuis une réserve de blocs libres pour stocker temporairement (mais de manière non volatile) les données. BAST maintient donc deux tables de translation : une table de translation par bloc pour l'ensemble de la mémoire adressable, et une table de translation par page pour chaque bloc de données possédant un bloc journalisé associé.

Un mécanisme de ramasse-miettes permet de réclamer les pages invalidées. Dans le cas général, les versions les plus récentes des pages contenues dans le bloc de données et dans son bloc journalisé sont copiées dans un nouveau bloc (*full-merge*, voir figure 3.15.a). Deux cas particuliers autorisent une réclamation plus performante. Si les pages d'un bloc sont écrites séquentiellement, le bloc journalisé devient un bloc de données et l'ancien bloc de données est effacé ; cette permutation intervient au niveau des tables de translation d'adresses (*switch-merge*, voir figure 3.15.b). Si le bloc journalisé contient les n premières pages, il est complété avec les pages suivantes avant d'effectuer une permutation (*partial-merge*, voir figure 3.15.c).

Lorsque les accès en écriture sont répartis sur plus de bloc logiques qu'il n'y a de bloc libres (écriture aléatoires), les blocs journalisés doivent être réclamés alors qu'ils sont peu remplis, ce qui augmente la fréquence des opérations de fusion (coûteuses) et dégrade fortement les performances. En contrepartie, les écritures séquentielles correspondent au cas le plus favorable de l'algorithme de ramasse-miettes (*switch-merge*).

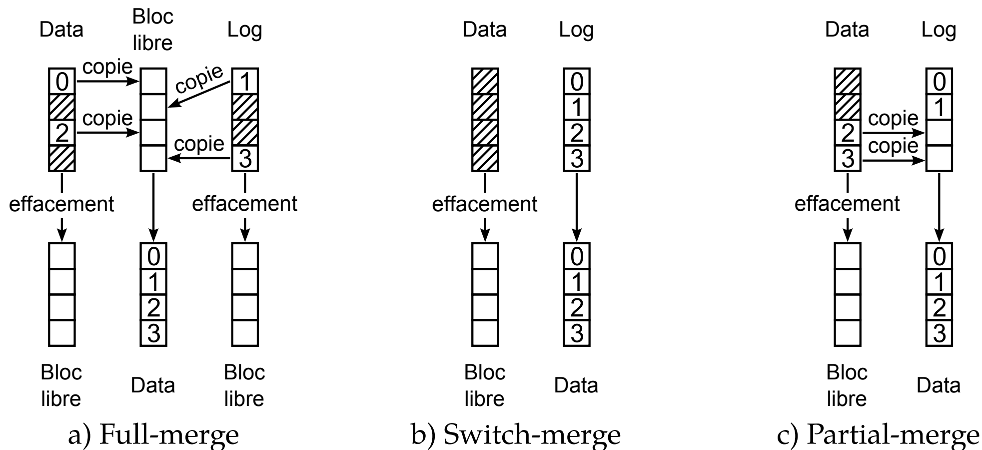


FIGURE 3.15: Réclamation des blocs pour les FTL hybrides

Fully Associative Sector Translation (FAST) FAST [Lee *et al.*, 2007] est une FTL hybride similaire à BAST. Sa particularité est de mutualiser les blocs journalisés – chacun de ces blocs peut contenir les pages de plusieurs blocs logiques distincts – pour améliorer leur taux d'utilisation. Ces blocs servent donc de buffer non volatile d'écriture. Lorsque ceux-ci sont pleins, l'espace est réclamé en fusionnant les blocs journalisés avec les blocs de données. Par rapport à BAST, l'opération de fusion est particulièrement coûteuse avec FAST, car typiquement plusieurs blocs de données doivent être réécris. En contrepartie, cette opération est moins fréquente lors d'écritures aléatoires, car un bloc journalisé est nécessairement plein avant sa réclamation. Un autre désavantage de la mutualisation des blocs journalisés est la diminution de la fréquence des opérations de *switch-merge*. Pour conserver de bonnes performances en écritures séquentielles, FAST dédie un bloc journalisé, associé à un seul bloc de données, à ce type d'accès – de manière similaire à BAST.

Locality-Aware Sector Translation (LAST) LAST [Lee *et al.*, 2008a] possède un fonctionnement similaire, mais réserve plusieurs blocs journalisés aux écritures séquentielles. En fonction du type d'accès (séquentiel ou aléatoire), les écritures sont redirigées vers des blocs dédiés aux écritures séquentielles (comme pour BAST), ou aux écritures aléatoires (comme pour FAST). De plus, LAST divise les écritures aléatoires en deux catégories en fonction de leur fréquence afin de réduire les coûts de réclamation. Cette FTL possède de bonnes performances pour les accès séquentiels parallélisés, éventuellement mixés avec des écritures aléatoires.

Superblock-based FTL Pour cette Superblock-based FTL [Kang *et al.*, 2006], une translation d'adresse par bloc est maintenu en RAM. Ces blocs sont regroupés en super-blocs, au sein desquels une translation d'adresses par page est effectuée grâce à un index hiérarchique stocké dans l'espace réservé aux métadonnées des pages du super-bloc. La dimension de cet espace réservé limite la taille de l'index, et donc celle des super-blocs. Un super-bloc regroupe typiquement au maximum huit blocs.

Par super-bloc, cet index est constitué d'un nœud racine et d'un niveau intermédiaire, les feuilles étant les pages de données. Lorsqu'une page est écrite, la hiérarchie de l'index la concernant doit être recopiée pour mettre à jour les pointeurs correspondants. En conséquence, l'espace réservé aux métadonnées stocke à la fois la racine et le niveau intermédiaire parent à la page. La table de translation d'adresse en RAM contient, en plus de l'adresse physique du super-bloc, un pointeur vers la racine de l'index, dont la version la plus récente est toujours

rattachée à la dernière page écrite. Comme l'accès à une page nécessite de lire cet index, cette FTL utilise des caches sur ces métadonnées pour améliorer les performances.

La finesse de ce mapping au sein des super-blocs permet à l'algorithme de ramasse-miettes de regrouper les pages accédées fréquemment afin de diminuer les coûts de réclamation.

3.2.2.4 Flash resident page-mapping FTL

Demand-based Page-mapped FTL (DFTL) DFTL [Gupta *et al.*, 2009] est une FTL basée sur un translation par page, dont la table de translation d'adresse est stockée sur la mémoire flash. Lors de la lecture ou de l'écriture d'une page, la partie de la table concernée par cette opération est chargée en RAM, à la demande. Pour libérer la RAM, ce chargement peut nécessiter l'éviction d'une autre partie de la table de translation d'adresse, qui est alors écrite sur des pages spécialisés appelées pages de translation – *translation pages*. La gestion des pages de translation en RAM est basée sur un algorithme LRU (Least Recently Used).

Contrairement aux FTL hybrides, DFTL possède un surcoût commun aux lectures et écritures, mais propose la même flexibilité que les FTL avec une translation d'adresse par page pour les mécanismes de ramasse-miettes, avec un coût en RAM moins important. De plus, DFTL possède les mêmes performances que ces FTL lorsque les accès sont restreints à un nombre limité de pages (localité temporelle des accès), car les parties correspondantes de la table de translation d'adresse sont conservées en RAM, sans chargement ni éviction.

3.2.2.5 Parallélisme

Les mémoires flash, en particulier les SSD, incluent généralement plusieurs puces flash. Il est donc nécessaire de paralléliser les accès à ces puces pour obtenir des performances optimales. Les FTL présentées précédemment concernent généralement une seule puce. La gestion de volumes plus importants doit prendre en compte la répartition et l'ordonnancement des opération [Agrawal *et al.*, 2008].

3.2.2.6 NoFTL [3]

A cause de la complexité des FTL, les performances des mémoires flash sont difficiles à estimer. Plusieurs travaux proposent donc des approches duales, pour permettre au système d'obtenir des performances optimales, à condition qu'il gère lui-même, au moins en partie, la complexité de la FTL.

FTL Bimodale Une FTL bimodale [Bonnet et Bouganim, 2011] garantit des performances optimales sous certaines restrictions du mode d'accès aux données, proche des accès bas niveaux des puces flash : les écritures doivent être effectuées avec la granularité d'une page, séquentiellement au sein d'un bloc. De plus, le contenu d'un bloc doit être invalidé en intégralité par une commande TRIM [Stevens, 2009] avant que celui-ci soit réécrit. En cas de non respect de ces contraintes, une FTL classique est utilisée pour autoriser les autres types d'accès et conserver une interface compatible avec celle des disques durs. Cette FTL ne doit cependant pas interférer avec les accès aux blocs optimaux. Les auteurs identifient également une FTL minimale, commune aux deux types d'accès, pour gérer la répartition de l'usure : cette FTL permet d'empêcher une détérioration trop rapide de la mémoire, tout en ayant un impact minimal sur les performances.

3. Not-only FTL, par analogie au concept NoSQL

Nameless Writes Pour réduire les coûts liés à l'indirection, le système peut prendre en charge la gestion de l'emplacement physique des données par l'intermédiaire de *nameless writes* [Arpaci-Dusseau *et al.*, 2010]. Ces écritures permettent à la mémoire flash de choisir (et retourner) l'adresse où sont stockées les données, de manière similaire à une allocation dynamique. Une mise à jour des données peut alors être effectuée à leur emplacement d'origine (*named write*), ou en déplaçant les données – en libérant l'espace alloué précédemment avec une commande TRIM, suivie d'une nouvelle *nameless write*. Ce type d'écriture convient aux systèmes de fichiers basés sur des mécanismes de *shadow paging*, comme ZFS, WAFL ou LFS, car les nouvelles versions des données sont copiées à des emplacements différents.

3.2.3 Systèmes de fichiers

De manière alternative à la FTL, la gestion des contraintes liées aux mémoires flash, telles que l'effacement ou la répartition de l'usure, peut être déléguée à un système de fichier adapté. Cette solution peut être plus efficace que d'empiler un système de fichier conçu pour les disques durs au dessus d'une FTL [Gal et Toledo, 2005]. La suppression d'un fichier peut par exemple permettre de réclamer l'espace précédemment alloué au lieu d'utiliser l'approche classique marquant simplement le fichier comme effacé. YAFFS, JFFS, et NANDFS [Zuck *et al.*, 2009] sont des exemples de systèmes de fichiers s'interfaçant uniquement avec des puces flash. Ceux-ci sont particulièrement utilisés dans les environnements embarqués, où les mémoires flash n'intègrent généralement pas de FTL.

3.3 Performances

Les FTL sont des logiciels propriétaires intégrés aux mémoires flash par les constructeurs. Elles peuvent être souvent mises à jour, et ne sont que très rarement documentées. Bien que les performances des puces flash soient connues avec une bonne précision, les périphériques implémentant une FTL peuvent se comporter différemment de part les compromis choisis. A cause de cela, certains travaux s'intéressent à l'évaluation des performances des mémoires flash.

uFLIP [Bouganim *et al.*, 2009] est une collection de neuf micro-benchmarks, définis pour caractériser les performances des mémoires flash en fonction de différentes propriétés des accès :

- la granularité : la taille des secteurs à lire ou à écrire,
- l'alignement : le décalage entre l'adresse du secteur et les pages,
- la localité : le regroupement des accès aléatoires sur une zone limitée de la mémoire,
- le partitionnement : l'utilisation de plusieurs zones de la mémoire pour des accès séquentiels parallèles,
- l'ordre : le sens lors d'accès séquentiels (décroissant, en place, croissant ou avec un écart entre deux accès successifs),
- le parallélisme : l'utilisation simultanée de plusieurs accès identiques,
- la combinaison : l'impact du ratio lecture/écriture et accès séquentiel/aléatoire,
- la pause : l'insertion d'un temps de pause entre chaque opération,
- les bursts : l'insertion d'un temps de pause entre chaque série d'opérations.

Ce benchmark est exécuté sur onze périphériques allant de la clé USB au SSD. Des caractéristiques communes sont mises en évidence – comme les bonnes performances en lectures séquentielles et aléatoires ou l'impact minime de la combinaison des accès – tandis que d'autres sont spécifiques à certains périphériques.

Type d'accès Les mémoires flash possèdent de bonnes performances en lecture séquentielle et aléatoire, ainsi qu'en écriture séquentielle. En contrepartie, les performances sont dégradées pour les écritures aléatoires qui, de plus, possèdent une latence très variable [Chen *et al.*, 2009].

[Birrell *et al.*, 2007] met en avant la possibilité que les faibles performances des écritures aléatoires soient dues aux compromis faits pour réduire la table de translation d'adresses. Une des conclusions est que les performances des accès aléatoires ne peuvent être améliorées qu'en augmentant la mémoire volatile utilisée pour stocker cette table. En effet, les SSD haut de gamme possèdent généralement de bonnes performances pour les écritures aléatoires [Lee *et al.*, 2009]. Cependant, lorsque ce type d'accès est maintenu sur une durée importante, un phénomène de fragmentation apparaît, ce qui dégrade également les performances des écritures séquentielles [Chen *et al.*, 2009; Master *et al.*, 2010].

Parallélisme Pour certains SSD haut de gamme, les lectures aléatoires possèdent des performances inférieures aux lectures séquentielles lorsque la requête concerne un secteur de petite taille [Chen *et al.*, 2009]. En effet, les performances d'un SSD dépendent de l'utilisation de ses unités accessibles en parallèle, les secteurs accédés en lecture doivent donc être suffisamment larges pour pouvoir être scindés en multiple requêtes exécutables en parallèle. Cependant, il est également possible de communiquer au SSD suffisamment de lectures asynchrone pour que celui-ci puisse les distribuer parmi ses composants [Baumann *et al.*, 2010]. Le *Native Command Queuing* (NCQ) est un mécanisme permettant au SSD de recevoir plusieurs requêtes de lecture ou d'écriture et de le laisser déterminer l'ordre d'exécution optimal de ces requêtes. Avec suffisamment de requêtes en attente d'exécution dans cette queue, les lectures aléatoires deviennent aussi performantes que les lectures séquentielles [Baumann *et al.*, 2010; Lee *et al.*, 2009].

Concernant les écritures, le parallélisme a peu d'impact sur les performances, car la finesse du mapping des SSD haut de gamme leur permet généralement de rediriger l'écriture vers une puce disponible [Master *et al.*, 2010].

Activité en arrière plan [Bouganim *et al.*, 2009; Chen *et al.*, 2009] mettent en évidence une forte activité en arrière plan pour les SSD haut de gamme, ce qui peut impacter les performances des accès concurrents. Cette activité est généralement liée aux mécanismes de ramasse-miettes.

Taux de remplissage Lorsque le système permet la réclamation de la mémoires par un mécanisme similaire à la commande TRIM, le taux de remplissage de la mémoire influence les performances des écritures aléatoires sur le long terme. Ce phénomène est lié au nombre de déplacement de pages valides lors de la réclamation, qui augmente avec le taux de remplissage [Master *et al.*, 2010].

Les performances des mémoires flash sont donc variables d'un périphérique à un autre, mais également notablement différentes de celles des disques durs. [Rajimwale *et al.*, 2009] met en évidence de nombreuses hypothèses faites par le système d'exploitation pour les disques durs et pas toujours applicables aux mémoires flash :

- les accès séquentiels sont beaucoup plus performants que les accès aléatoires,
- la proximité des adresses logiques traduisent une proximité au niveau des emplacements physiques correspondants,
- les performances sont homogènes sur l'ensemble de l'espace mémoire adressable (ce qui n'est pas le cas lorsque la mémoire flash combine des technologies SLC et MLC),
- la durée d'une opération est a peu près proportionelle à sa taille,
- il n'y a pas de désallocation de la mémoire,

- il n'y a pas ou peu d'activité en arrière-plan.

Cependant, en l'état, les SSD peuvent être – et sont – employés en remplacement des disques durs, car, à coût similaire, les mémoires flash peuvent posséder de meilleures performances globales, malgré un espace de stockage réduit par rapport aux disques durs [Gray et Fitzgerald, 2008]. Certains cas d'application favorisent aussi notablement les mémoires flash, en particulier les écritures séquentielles associées à des lectures aléatoires. Ce type d'accès se retrouve notamment dans l'utilisation de tables temporaires, en particulier par des algorithmes de tri et de jointure, et dans l'écriture de logs. [Lee *et al.*, 2008b] par exemple montre un gain en performances significatif lors de l'utilisation d'un SSD pour des requêtes de tri externe, de jointure par hachage et de jointure tri fusion.

3.4 Optimisation du système d'exploitation pour les mémoires flash

3.4.1 Système de fichiers

Bien qu'il existe des systèmes de fichiers conçus pour s'interfacer avec les puces flash (voir section 3.2.3), [Wang *et al.*, 2009] mettent également en évidence une catégorie de systèmes de fichiers particulièrement adaptée aux mémoires flash intégrant une FTL. Les *log-structured file systems* traitent le support de stockage comme un log circulaire (i.e. de capacité virtuellement infinie) en écrivant séquentiellement à la fin du log. Pour la plupart des FTL-hybrides, les accès séquentiels en écriture restent privilégiés, d'où l'intérêt de ce type de système de fichiers : Wang *et al.* [2009] montrent un gain en performances significatif – d'un facteur 3 à 6.6 – en utilisant NILFS2 par rapport à EXT2.

3.4.2 Ordonnanceur d'entrées/sorties

[Kim *et al.*, 2009] proposent un ordonnanceur d'E/S adapté aux caractéristiques des mémoires flash. Celui-ci distingue les lectures des écritures – ce qui n'est pas le cas des ordonnanceurs adaptés aux disques durs – car la réorganisation des lectures est inutile alors que la réorganisation des écritures améliore les performances. Les écritures sont regroupées et réagencées au sein d'un bloc, tandis que les lectures conservent une politique *first in, first out* (FIFO). De plus, une seconde amélioration des performances est observée lorsque l'ordonnanceur favorise les opérations de lecture par rapport aux écritures.

3.4.3 Gestion des caches

3.4.3.1 Caches d'écriture pour les puces flash

Ces caches d'écriture ont pour objectif de réduire les coûts liés à la réclamation. Ces mécanismes ne sont applicables que lorsque ils disposent d'un accès bas niveau aux puces flash, c'est à dire généralement en les intégrant au sein de la FTL.

Flash-Aware Buffer (FAB) FAB [Jo *et al.*, 2006] est un algorithme de gestion de cache regroupant les pages appartenant à un même bloc afin de ne l'effacer qu'une seule fois pour mettre à jour toutes les pages qu'il contient. Le bloc à réclamer est celui contenant les plus de pages modifiées ; ou celui accédé le moins récemment en cas d'égalité (LRU). En conséquence, moins de copies de pages valides sont nécessaires vu que de nombreuses pages d'un même bloc sont écrites en même temps.

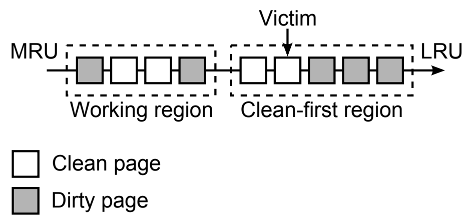


FIGURE 3.16: Gestion du cache par CFLRU

Block Padding LRU (BPLRU) BPLRU [Kim et Ahn, 2008] présente un fonctionnement similaire, en utilisant un algorithme LRU pour stocker les blocs accédés récemment, et différer les écritures sur ceux-ci. BPLRU se distingue de FAB en favorisant les blocs accédés récemment, indépendamment du nombre de pages modifiées qu'il contient.

L'intérêt de ces deux approches est de tenir compte du mécanisme d'effacement par bloc des mémoires flash pour regrouper les mises à jour des pages contenues dans un même bloc. Ces deux mécanismes présentent une amélioration significative par rapport à l'absence de bufferisation, ou une bufferisation ne prenant pas en compte la granularité particulière des mémoires flash.

Multi-Chip based replacement Algorithm (MCA) MCA [Seol *et al.*, 2009] est une optimisation de l'algorithme LRU pour les caches d'écriture des SSD. Lorsque le système écrit des données dont la taille est inférieure à celle d'une page, le SSD doit récupérer les parties manquantes avant de pouvoir écrire la page entière à un autre emplacement. Ce mécanisme de lecture-modification-écriture est problématique pour les SSD avec un translation d'adresse par page, car si il est possible de rediriger l'écriture sur une puce flash disponible, la lecture par contre fait appel à une puce particulière, qui peut être occupée par une autre opération. Afin de minimiser le temps d'inactivité des puces, l'éviction par MCA d'une page du cache à écrire en lecture-modification-écriture ne se fait que si la puce contenant les données correspondantes est disponible. Dans le cas contraire, l'algorithme réclame la page précédente de la LRU.

3.4.3.2 Gestion des caches des mémoires flash

À la différence des caches d'écriture, ces mécanismes visent des mémoires flash incluant une FTL. Ils permettent de maximiser l'efficacité du cache en prenant en compte les caractéristiques particulières des mémoires flash.

Clean-First LRU (CFLRU) CFLRU [Park *et al.*, 2006] prend en compte l'asymétrie lecture-écriture des mémoires flash en conservant les pages modifiées plus longtemps que les pages non modifiées. Pour cela, CFLRU divise la liste LRU du cache en une zone de travail – *working region* – et une zone d'effacement – *clean-first region*. Les évictions concernent en priorité les pages non modifiées de la zone d'effacement. Les pages modifiées ne sont retirées du cache que lorsque cette zone ne contient plus aucune page non modifiée. Ce mécanisme est illustré sur la figure 3.16. La taille de la zone d'effacement étant fixé, CFLRU s'adapte mal aux changements de ratio entre les lectures et les écritures. En effet, un espace important dédié à cette zone favorise les écritures au dépend des lectures.

Clean-First Dirty-Clustered (CFDC) CFDC [Ou *et al.*, 2009] reprend les principes de CFLRU, mais divise la zone d'effacement – appelée *priority region* – en deux listes LRU, une dédiée aux pages non modifiées, et une pour les pages modifiées. Dans cette seconde liste, les pages sont

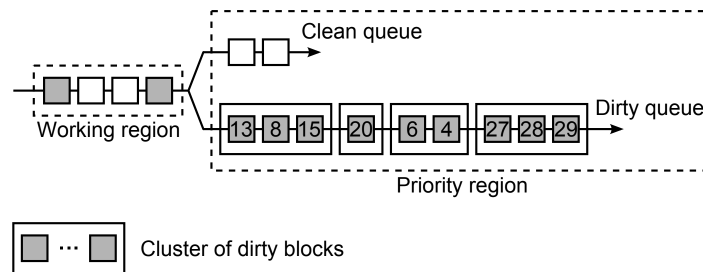


FIGURE 3.17: Gestion du cache par CFDC

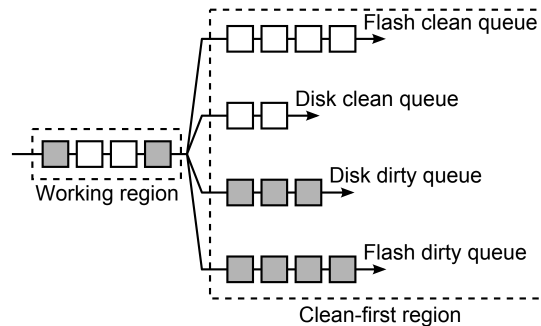


FIGURE 3.18: Gestion hybride du cache sur SSD et disque dur

regroupées par bloc, de manière similaire à BPLRU, afin d'améliorer la localité spatiale des écritures. La figure 3.17 illustre ces structures.

Gestion du cache hybride pour mémoires flash et disques durs [Koltsidas et Viglas, 2008] présente un algorithme pour placer une page sur un SSD ou un disque dur en fonction de la charge qui lui est associée. Lorsque cette page est souvent accédée en lecture, elle est stockée sur le SSD ; dans le cas contraire, elle est placée sur le disque dur. L'algorithme de placement est dynamique, et s'adapte aux variations de charge de chaque page afin de les placer de manière optimale. Le cache est géré de manière similaire à CFLRU et CFDC , en distinguant quatre listes LRU pour les différents types d'accès (lecture ou écriture sur mémoire flash et lecture ou écriture sur disque dur) ; ces structures sont illustrées sur la figure 3.18. L'éviction des pages se base sur le coût des opérations correspondantes avec, par ordre de priorité, les lectures sur mémoire flash, les lectures sur disque dur, les écritures sur disque dur puis les écritures sur mémoire flash.

Cache non volatile (StableBuffer) Pour certaines mémoires flash, les écritures aléatoires localisées (typiquement sur quelques mégaoctets [Bouganim *et al.*, 2009]) sont quasiment aussi efficaces que les écritures séquentielles. Cette propriété est exploitée par StableBuffer [Li *et al.*, 2010], qui alloue un espace mémoire de cette taille pour rediriger temporairement les écritures. Lorsque ce cache (non volatile) est plein, un mécanisme de réclamation identifie des ensembles de pages dont la disposition (séquentiel ou groupé) permet une copie efficace des données vers leur emplacement d'origine.

3.5 Optimisation des SGBD pour les mémoires flash

La plupart des SGBD incluent des optimisations spécifiques aux disques durs, qui sont peu pertinentes pour les mémoires flash. Avec l'utilisation de ce type de support de stockage par les SGBD, de nouvelles approches ont été définies au niveau de la disposition des données et de l'indexation.

3.5.1 Disposition des données

In-Page Logging (IPL) Dans les mémoires flash, lorsqu'une page doit être mise à jour celle-ci est marquée comme invalide et recopiée à un endroit différent, même lorsque la modification est mineure. L'approche IPL [Lee et Moon, 2007] propose de n'écrire (loguer) que les modifications afin de réduire le nombre de réécritures. Pour cela, quelques pages sont réservées dans chaque bloc pour stocker ces modifications. Lorsque tout l'espace réservé pour le log est occupé, les pages du bloc sont mises à jour avec les modifications qu'il contient. Ce mécanisme permet d'améliorer les performances en écriture, mais ralentit la lecture car un accès à une page nécessite la lecture du log qui lui est associé. Ce compromis dépend du nombre de pages utilisées pour le log. Ce mécanisme peut également servir à remplacer le système de log pour les SGBD.

Page-Differential logging (PDL) PDL [Kim *et al.*, 2010] propose une approche similaire à IPL, en loguant le différentiel par rapport à la page originale au lieu de chaque modification. Lors d'une écriture, ce différentiel est recalculé et stocké dans les pages réservées au log, ce qui invalide les différentiels précédents. Ainsi, une lecture ne nécessite au maximum que l'accès à deux pages : la version originale et son dernier différentiel.

Append and Pack L'approche Append and Pack [Stoica *et al.*, 2009] est similaire aux systèmes de fichiers *log-structured*, en écrivant les données exclusivement séquentiellement (*append*). La réclamation (*pack*) s'effectue sur les données les moins récemment mises à jour, en les recopiant en tête du log. Pour éviter de recopier systématiquement les données statiques, deux logs *hot* et *cold* sont différenciés en fonction des fréquences d'accès : la réclamation (*pack*) copie les données dans l'ensemble *cold*, tandis que l'écriture (*append*) s'effectue en tête du log *hot*. Comme pour les *nameless write* ou les systèmes de fichiers log-structured, cette technique présente l'inconvénient de changer l'adresse des données en cas de modification.

Partition Attribute Across (PAX) PAX [Ailamaki *et al.*, 2002] est une disposition orientée colonne au niveau des pages de données, illustrée par la figure 3.19. Avec cette disposition, l'accès à un nombre limité de colonnes permet de diminuer l'utilisation de la bande passante, voire également du nombre de lectures lorsque les pages de données du SGBD sont plus grande que les pages de la mémoire flash. FlashScan [Tsirogiannis *et al.*, 2009] est un algorithme de sélection-projection conçu pour retarder au maximum la lecture des données (matérialisation tardive) : les colonnes projetées ne sont accédées qu'après la sélection. De manière similaire, FlashJoin est un algorithme de jointure basé sur une stratégie de matérialisation tardive. Ces deux techniques présentent des gains de performances par rapport aux algorithmes usuels.

Bucket File (B-File) B-File [Nath et Gibbons, 2008] est une structure de stockage pour des données possédant une date d'expiration. En fonction de cette date, les données sont écrites séquentiellement dans l'unité d'effacement appropriée pour éviter d'avoir à copier des données valides lors de leur effacement. Cette technique est utilisée pour stocker des échantillons aléatoires dans un réseau de capteur, où la date d'expiration de l'échantillon est déterminé au moment de la génération de la mesure.

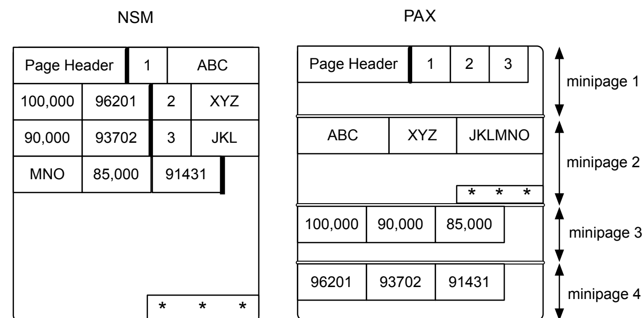


FIGURE 3.19: Disposition des données avec N-ary Storage Model (NSM) et Partition Attribute Across (PAX) (Tsirogiannis *et al.*, 2009)

FlashLogging FlashLogging [Chen, 2009] utilise des mémoires flash pour l'écriture des journaux de transaction, car pour les écritures synchrones (bloquantes) de petite taille, ce type de mémoire est plus adapté que les disques durs, limités par le délai rotationnel. Une analyse des performances des clés USB utilisées montre un écart important entre les performances des écritures : la majorité des écritures sont rapides, mais quelques opérations peuvent avoir une latence plus importante de plusieurs ordres de grandeur (car celles-ci déclenchent une réclamation coûteuse). Pour pallier à ce problème, lorsqu'une écriture dépasse de deux fois sa durée nominale, les données sont copiée sur une autre clé USB disponible, car cette nouvelle requête possède une probabilité importante de finir avant la première.

3.5.2 Indexation

3.5.2.1 B-tree

μ-tree A cause de l'impossibilité de réécrire une page sur une puce flash, celles-ci sont généralement copiées dans un autre emplacement. Lors de la modification d'un nœud d'un arbre de type B-tree, l'adresse du nœud change, nécessitant de mettre à jour le nœud parent, car celui ci possède des pointeurs vers ses nœuds fils, et cela successivement jusqu'à la racine de l'arbre. μ-tree [Kang *et al.*, 2007] est une adaptation des arbres B-tree pour les mémoires flash où tous les nœuds mis à jour sont stockés dans la même page, de la feuille à la racine. Lorsqu'un nœud est modifié, tous ses parents le sont également, mais cela ne nécessite l'écriture que d'une page, améliorant ainsi les performances. En contrepartie, les nœuds contiennent moins de données et l'espace occupé par μ-tree est estimé à deux fois celui d'un arbre B-tree.

BFTL Avec BFTL [Wu *et al.*, 2007], les mises à jour des nœuds de l'arbre sont dans un premier temps bufferisées en RAM, puis écrites dans des pages de log en les regroupant. Une table de translation de nœuds permet de reconstruire les nœuds logiques en indiquant les secteurs physiques où sont stockées les données. Cette approche de type log-structured spécifique aux B-tree permet de réduire le coût de mise à jour de l'arbre, au détriment de la lecture qui doit accéder à toutes les pages de log associées pour reconstruire le nœud.

FlashDB FlashDB [Nath et Kansal, 2007] définit deux types de nœuds : un nœud classique, mis à jour intégralement lorsque cela est nécessaire, de la même manière qu'une implémentation pour les disques durs , et un nœud structuré sous forme de log, de manière similaire à BFTL. FlashDB adapte dynamiquement pour chaque nœud le type à utiliser en fonction du ratio de

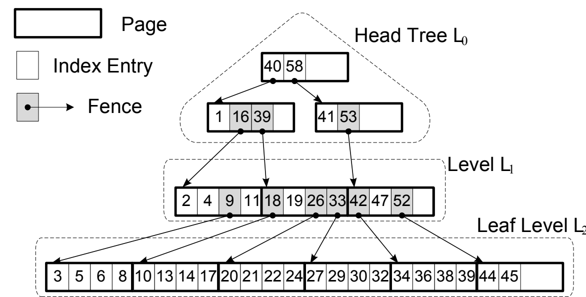


FIGURE 3.20: Un exemple de FD-tree (Li *et al.*, 2009)

lecture/écriture qui lui est associé. Cette technique présente un gain de performance par rapport à une approche uniquement basée sur les logs, ou une mettant constamment à jour les nœuds.

3.5.2.2 Autres arbres équilibrés

Lazy-adaptive tree (LA-tree) Contrairement aux index traditionnels, LA-tree [Agrawal *et al.*, 2009] ne propage pas immédiatement les mises à jour de l'index vers les nœuds fils. La mémoire flash stocke des buffers de mises à jour associés aux nœuds, et contenant les opérations à appliquer à ce nœud ou à ses descendants. La réclamation d'un buffer consiste à faire descendre les informations qu'il contient, jusqu'à éventuellement atteindre une feuille de l'arbre. Comme le coût de lecture d'un nœud augmente avec la taille du buffer, LA-tree se base sur la charge courante pour déterminer si la réclamation du buffer est avantageuse.

FD-tree FD-tree [Li *et al.*, 2009] est un ensemble de séries de données triées. Chaque série constitue un niveau, en commençant par le niveau L0 stocké en RAM par un B-tree ; les niveaux suivants (L1, L2, etc.) sont *n* fois plus grand que le précédent. La figure 3.20 illustre cette structure. Lorsqu'un niveau est plein, celui-ci est fusionné avec le niveau suivant, ce qui donne des écritures séquentielles. Cependant, la fusion de deux niveaux en bas de l'arbre peut être particulièrement coûteuse et bloquer les autres accès jusqu'à sa complétion.

Indexation multidimensionnelle L'accès à des index multidimensionnelles de type R*-tree implique principalement des lectures aléatoires. En conséquence, au delà de quelques pourcent de taux de sélection des données, en lire l'intégralité (accès séquentiel) est plus performant que de passer par un index spatial (accès aléatoire) avec un disque dur. [Emrich *et al.*, 2010] étudient les changements apportés par les SSD pour l'indexation multidimensionnelle. De part leurs performances en lecture aléatoire, le seuil du taux de sélection passe à quelques dizaines de pourcent. Pour les accès en lecture, les mémoires flash sont donc particulièrement adaptées à ce type d'index.

3.5.2.3 Filtres de Bloom

Les filtres de Bloom [Bloom, 1970] sont des structures d'index conçues pour représenter de manière compacte un ensemble d'éléments. En utilisant ce type d'index, il est possible de déterminer l'appartenance d'un élément à un ensemble avec peu de faux positifs et aucun faux négatif. PBFilter [Yin *et al.*, 2009] propose d'accélérer les tests d'appartenance en utilisant un index basé sur les filtres de Bloom. Pour accélérer le parcours de l'index, celui ci est partitionné, avec chaque partition stockant une partie du vecteur de bits. Ainsi, pour retrouver une valeur, seules les partitions contenant les bits dont les positions sont déterminées par des fonctions de

hachage sont à examiner. Cet index présente l'avantage d'être compact, et de nécessiter peu de RAM, caractéristique fondamentale pour les systèmes embarqués.

3.6 Amélioration des écritures aléatoires

L'intégration de mémoires flash bas/milieu de gamme aux IGCBox a mis en évidence le manque d'optimisation des accès pour ce type de mémoire. Dans cette section, nous proposons un algorithme d'écriture adapté qui, en généralisant, permet d'améliorer les écritures aléatoires sur mémoires flash [Chardin *et al.*, 2011]. Cette optimisation se base sur une corrélation forte entre les performances des écriture et leur localité spatiale – propriété mise en évidence par des travaux précédents (cf. section 3.6.1). Nous définissons une distance pour vérifier et quantifier cet effet, puis validons l'efficacité de notre proposition à l'aide d'un modèle mathématique et par des résultats expérimentaux. Avec cette optimisation, les écriture aléatoires deviennent potentiellement aussi efficace que les écritures séquentielles, améliorant ainsi les performances de deux ordres de grandeur.

Par rapport aux approches précédentes pour journaliser les écritures (IPL [Lee et Moon, 2007], PDL [Kim *et al.*, 2010] et *Append and Pack* [Stoica *et al.*, 2009]), notre proposition ne fait pas appel à des mécanismes de ramasse-miettes pour réclamer des zones d'écriture contiguës. De plus, IPL et PDL dégradent significativement les performances en lecture, alors que notre algorithme ne possède qu'un impact négligeable pour cette opération. En contrepartie, une partie de la mémoire flash est réservée pour le fonctionnement interne de notre algorithme, ce qui en diminue la capacité « utile ». La consommation en RAM de notre approche est également significative, ce qui proscrit sa mise en œuvre dans des équipements où cette ressource est fortement limitée.

Les SSD haut de gamme proposent généralement de bonnes performances en écriture aléatoire, en échange de RAM, de capacité de traitement et de blocs de rechange (non accessibles par l'hôte) intégrés au périphérique [Agrawal *et al.*, 2008; Lee *et al.*, 2009]. Cependant, ces conceptions fournissent des performances homogènes sur l'ensemble de la mémoire flash. Comme la plupart des applications dans les bases de données mélangent les accès aléatoires et séquentiels, ce gain en performance n'est pas requis sur l'ensemble de la mémoire. En ajoutant une surcouche logicielle, notre optimisation permet l'utilisation de mémoires flash plus économiques, tout en proposant de bonnes performances en écriture aléatoire.

3.6.1 Localité spatiale des écritures

La localité spatiale des écritures possède un impact sur les performances : [Birrell *et al.*, 2007] identifient une corrélation forte entre la latence moyenne d'une opération d'écriture et l'espace entre les écritures successives, tant que cet espace est inférieur à la taille de deux blocs. Ils en concluent que les performances en écriture dépendent de la probabilité que plusieurs écritures concernent le même bloc, ce qui est symptomatique d'une translation par bloc ou hybride. En conséquence, une translation plus fine est nécessaire pour les mémoires flash à haute performance, mais nous considérons qu'une telle finesse peut être fournie par une sur-couche distincte de la FTL. En effet, [Wang *et al.*, 2009] étudient l'efficacité des systèmes de fichiers journalisés (*log-structured file systems*) pour les SGBD, car ce type de système de fichiers à tendance à écrire de larges blocs de données de manière contiguë. Leurs expérimentations confirment des bénéfices potentiels car ils obtiennent une amélioration des performances s'élevant jusqu'à x6.6.

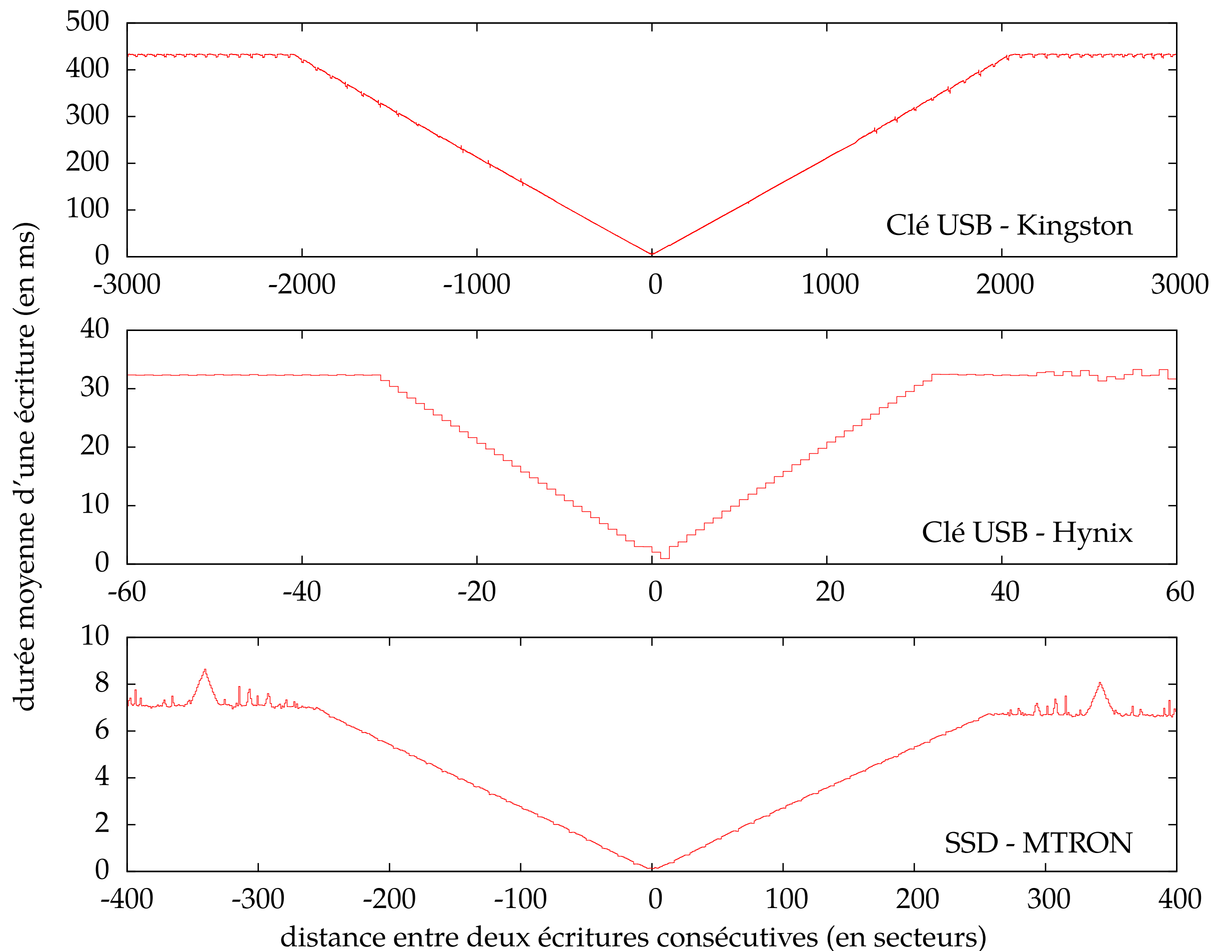


FIGURE 3.21: Impact de la distance sur la latence moyenne

Certains micro-benchmarks de uFLIP [Bouganim *et al.*, 2009] s'intéressent en particulier à la localité spatiale et aux incréments entre des écritures successives. Leurs résultats expérimentaux confirment que regrouper les écritures aléatoires améliore significativement leur efficacité, et des incréments importants peuvent mener à des écritures peu performantes.

Nous proposons une approche similaire pour quantifier l'impact de la localité spatiale sur les mémoires flash, en introduisant une notion de distance entre des écritures successives. Dans nos essais, la latence moyenne d'écriture pour chaque distance d est évaluée en ignorant $|d| - 1$ secteurs entre deux écritures successives. Cette métrique peut être négative pour distinguer les adresses croissantes et décroissantes. À partir des résultats des travaux précédents, notre hypothèse est que, jusqu'à une distance d_{max}, le coût moyen d'une opération d'écriture $cost(d)$ est approximativement proportionnel à d.

Pour valider cette hypothèse, nous mesurons l'impact de cette distance sur des mémoires flash diverses. Bien que les latences soient erratiques, leur moyenne convergent lorsque ce type d'accès est prolongé. La fig. 3.21 montre que notre hypothèse est vérifiée pour un SSD [4] et pour deux clés USB [5] [6].

4. Mtron MSD SATA3035-032
5. Puce flash HYNIX HY27UG088G5B et un contrôleur ALCOR AU6983HL
6. Kingston DataTraveler R500 64 Go

Lorsque les écritures sont éloignées les unes des autres ($d \geq d_{max}$), leurs latences sont typiquement 20 à 100 fois plus élevées que celles des écritures séquentielles pour les mémoires flash avec une translation par bloc [Bouganim *et al.*, 2009]. En conséquence, et à cause de la proportionnalité des performances, réduire la distance moyenne entre les écritures successives peut améliorer significativement les performances, même sans atteindre un accès strictement séquentiel ($d = 1$). L'optimisation décrite dans la section suivante se focalise sur ce type d'accès, en essayant de minimiser cette distance.

3.6.2 Regroupement des écritures aléatoires

Notre contribution permet de convertir les écritures aléatoires en écritures séquentielles, en ignorant les secteurs contenant des données valides ; ce type d'accès étant performant pour les mémoires flash. Avec cette optimisation, les secteurs contenant des données valides et les secteurs libres sont mélangés sur la mémoire. Une couche de redirection est utilisée pour rediriger les écritures logiques vers des secteurs libres en minimisant la distance entre les écritures successives.

Pour permettre la récupération des données, l'association entre un secteur logique et sa position physique – sur le périphérique – est stockée dans une table de translation d'adresse, avec chaque secteur logique associé à un secteur physique. Les secteurs libres – ceux n'étant associés à aucun secteur logique – ne contiennent aucune donnée utiles, et constituent une réserve de secteurs disponibles pour les écritures.

Pour réécrire un secteur logique, les données sont écrites sur un secteur libre proche de l'écriture précédente. L'association secteur logique - secteur physique est ensuite mise à jour pour refléter ce changement. L'ancien secteur associé est donc ajouté à la réserve de secteur libre. L'exemple 3.1 illustre le fonctionnement de l'algorithme, en présentant comment les écritures logiques sont assignées à des secteurs physiques.

Cette méthode ne nécessite pas d'algorithme de ramasse-miettes, car la taille de la réserve reste constante : les secteurs physiques contenant des données obsolètes sont immédiatement ajoutés à cette réserve, et peuvent être réécris. Cependant, la FTL peut posséder son propre mécanisme de ramasse-miettes interne pour gérer les effacements, indépendamment de l'optimisation.

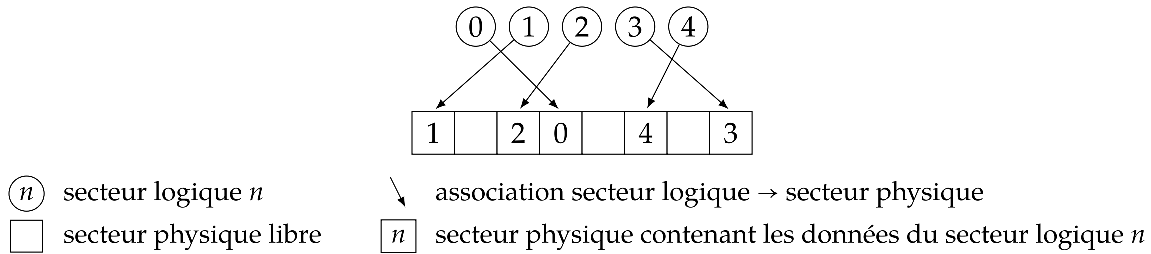
Quel que soit le type d'accès en écriture au niveau logique, celui-ci sera traduit en un accès « quasi-séquentiel » au niveau physique. En conséquence, la distance moyenne (et donc l'efficacité des écritures) est déterminé exclusivement par la proportion de secteurs libres sur le périphériques. Comme l'augmentation de cette réserve nécessite la réservation d'espace sur la mémoire flash, cet usage peut être ajusté pour obtenir une efficacité souhaitée. En contrepartie, les lectures séquentiels sont converties en lectures aléatoires. Mais cet aspect n'est pas problématique pour les mémoires flash, pour lesquelles les deux types d'accès sont aussi efficaces [Bouganim *et al.*, 2009]. Les lectures requièrent uniquement la consultation de la table de translation d'adresse, qui constitue un surcoût négligeable.

Pour éviter de revisiter des régions de la mémoire accédées récemment, où la réserve de secteurs libres devrait être épuisée, seules les distances positives sont considérées par cette optimisation : les adresses d'écriture sont croissantes. De plus, l'espace mémoire est supposé circulaire, pour éviter des traitements spécifiques pour les bords.

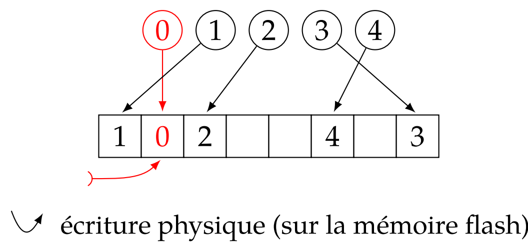
Pour la description de cet algorithme, le secteur accédé le plus récemment en écriture est noté $f_{\curvearrowleft}$. La première structure de données utilisée par cet algorithme d'écriture est la table de translation d'adresses. Cette table – appelée T – lie chaque secteur logique à un secteur phy-

EXEMPLE 3.1: Illustration du fonctionnement de l'algorithme

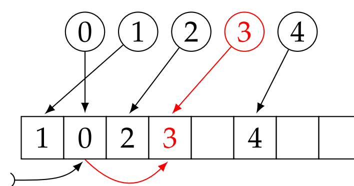
Une mémoire flash possède 8 secteurs physiques, dont 3 sont réservés pour rediriger les écritures. 5 (8 – 3) secteurs logiques sont alors accessibles par les applications – au dessus de l'optimisation – et sont identifiés par les entiers 0 à 4. Pour cet exemple, chaque secteur logique est initialement associé à un secteur physique comme suit :



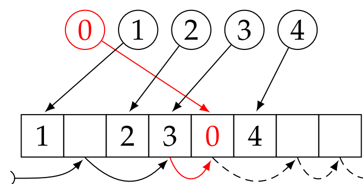
Une application demande alors à écrire le secteur logique 0. Au lieu d'écraser les données du secteur physique associé, l'écriture est redirigée vers le secteur libre le plus proche de l'écriture précédente – pour la première itération, l'algorithme choisit le premier secteur libre. L'association secteur logique → secteur physique pour l'identifiant 0 est mise à jour, afin de refléter le nouvel emplacement des données.



L'application écrit ensuite le secteur logique 3 ; l'algorithme redirige également l'écriture vers le secteur libre le plus proche de l'écriture précédente. Dans le cas présent, le secteur physique écrasé contient une ancienne version des données du secteur logique 0, dont l'emplacement a été libéré lors de l'écriture précédente.



Le secteur 0 est ensuite réécrit par l'application, ce qui correspond à nouveau à une redirection et à la modification de l'association secteur logique → secteur physique par l'algorithme. Cet algorithme permet d'accéder en écriture à la mémoire flash de manière « quasi-séquentielle », quel que soit le type d'accès au niveau logique.



ALGORITHME 3.1: écriture d'un secteur logique

```
  input : data, logical_sector
1 physical_sector ← SEARCH_CLOSEST(f↶)
2 WRITE(data, physical_sector)
3 table[logical_sector] ← physical_sector   { mise à jour de la table de translation
    d'adresses }
4 f↶ ← physical_sector   { pour conserver la référence du secteur f↶ }
```

ALGORITHME 3.2: écriture d'un secteur logique à l'aide d'une liste de secteurs libres

```
  input : data, logical_sector
1 old_sector ← table[logical_sector]
2 new_sector ← pool.POP()
3 WRITE(data, new_sector)
4 table[logical_sector] ← new_sector   { mise à jour de la table de translation
    d'adresses }
5 pool.ADD(old_sector)   { mise à jour de la liste des secteurs libres }
```

sique sur la mémoire flash. L'algorithme 3.1 est une première version simplifiée de l'opération d'écriture.

L'opération (1) – la recherche du secteur libre le plus proche de l'écriture précédente – doit être implémentée avec une structure de données adaptée. Dans notre implémentation, pour accélérer cette recherche, nous conservons les références de chaque secteur libre dans une liste triée par distances croissantes avec $f_{\curvearrowleft}$, notée P – pour *Pool*. Le premier élément de cette liste, $P(0)$, est donc le secteur libre le plus proche de $f_{\curvearrowleft}$, suivi de $P(1)$, etc..

L'ajout de cette liste permet de récupérer rapidement le secteur libre le plus proche. Néanmoins, cette liste doit être maintenue à jour pour chaque nouveau secteur libéré, à chaque fois que la table de translation d'adresse est modifiée. L'algorithme 3.2 intègre cette structure de données. La complexité des différentes opérations devient alors :

(1) *chercher le secteur le plus proche de* $f_{\curvearrowleft}$: $O(1)$,

(4) *mettre à jour la table de translation d'adresses* : $O(1)$,

(5) *mettre à jour la liste des secteurs libres* : $O(\log |\mathbb{P}|)$ avec une structure de données adaptée, comme les skip-lists [Pugh, 1990].

L'opération (5) est la seule opération avec un surcoût en CPU significatif. Cependant, celle-ci peut être effectuée de manière asynchrone (c'est à dire pendant l'écriture suivant dans un environnement soumis à des écritures intensives) sans ajouter de problème de consistance, car la liste des secteurs libres peut être reconstruite à partir de la table de translation d'adresses. En conséquence, cette liste peut ne pas être à jour lors de chaque nouvelle demande d'écriture, ce qui peut entraîner une légère augmentation de la distance moyenne si le secteur libre le plus proche de $f_{\curvearrowleft}$ n'est pas encore référencé par cette liste. Ce cas est toutefois peu fréquent lorsque la réserve de secteurs libres est importante, et peut être négligé.

3.6.3 Modèle

Pour estimer l'amélioration de la vitesse d'écriture apportée par cet algorithme, nous proposons de modéliser son comportement en calculant la latence moyenne d'une écriture. Ce modèle est basée sur les hypothèses simplificatrices suivantes : les secteurs libres sont distribués unifor-

mément parmi les secteurs physiques – cas défavorable par rapport à la distribution réelle [7] – et cet état reste stable quelles que soient les écritures. De plus, la latence d'une écriture est censée être déterminée exclusivement par sa distance avec l'écriture précédente.

Avec ces approximations, l'efficacité de l'algorithme peut être évaluée à partir de la probabilité d'obtenir chaque distance, et les latences associées. Pour ce modèle, les notations suivantes sont utilisées :

Définitions

Soit $\mathbb{F}$ l'ensemble des secteurs physiques (*Flash*).

Soit $\mathcal{D}(a, b)$ la distance entre deux secteurs physiques a et $b \in \mathbb{F}$.

Soit $\mathbb{L}$ l'ensemble des secteurs logiques.

Soit $\mathbb{P}$ l'ensemble des secteurs physiques libres (*Pool*) ; $\mathbb{P} \subset \mathbb{F}$ et $|\mathbb{P}| = |\mathbb{F}| - |\mathbb{L}|$.

Soit $p(d)$ la probabilité que le secteur $f_i \in \mathbb{P}$ qui minimise $\mathcal{D}(f_{\curvearrowleft}, f_i)$ vérifie aussi $\mathcal{D}(f_{\curvearrowleft}, f_i) = d$, c'est à dire obtenir la distance d entre deux écritures successives.

Avec l'hypothèse d'une distribution uniforme, la probabilité $p(d)$ d'obtenir la distance d entre deux écritures successives peut être évaluée comme le ratio des distributions favorables sur les distributions possibles :

$$p(d) = \frac{\binom{|\mathbb{F}| - d - 1}{|\mathbb{P}| - 1}}{\binom{|\mathbb{F}| - 1}{|\mathbb{P}|}} \tag{3.1}$$

Le coût d'une écriture en fonction de la distance $cost(d)$, peut être approximé comme étant proportionnel à d, mais pour nos évaluation, nous avons utilisé les mesures faites sur le périphérique, comme celles de la figure 3.21, plus précises.

À partir de ces deux paramètres $p(d)$ et $cost(d)$, le coût moyen $cost_{avg}$ est de :

$$cost_{\text{avg}} = \sum_{d=1}^{|\mathbb{L}|} p(d) \times cost(d) \tag{3.2}$$

Les estimations de ce modèle sont présentées dans la section 3.6.4, avec les résultats expérimentaux. En plus des gains de performances, l'utilisation des ressources peut être quantifiée, comme cette optimisation échange de la mémoire (RAM) de l'hôte, ainsi que de l'espace sur la mémoire flash contre un gain de performance en écriture.

Le surcoût en RAM est causé par la table de translation d'adresses et la liste de secteurs libres. Ces surcoûts s'élèvent à $O(|\mathbb{L}| \times \log |\mathbb{F}|)$ pour la table de translation d'adresses, et $O(|\mathbb{P}| \times \log |\mathbb{F}|)$ pour la liste. Le coût total en RAM est donc de $O(|\mathbb{F}| \times \log |\mathbb{F}|)$.

Comme les secteurs libres font partie des secteurs physiques, et ne contiennent aucune donnée utile, le surcoût en espace disponible sur la mémoire flash est de $|\mathbb{P}|$ secteurs.

Le dernier compromis significatif en termes de performances concerne les écritures séquentielles. Avec cet algorithme, les performances ne dépendent plus du type d'accès au niveau logique : les écritures séquentielles possèdent les même performances que les écritures aléatoires.

7. La mémoire étant accédée dans l'ordre des adresses croissantes, les secteurs libres ont tendance à être plus fréquents dans les régions à visiter que dans les régions récemment visitées.

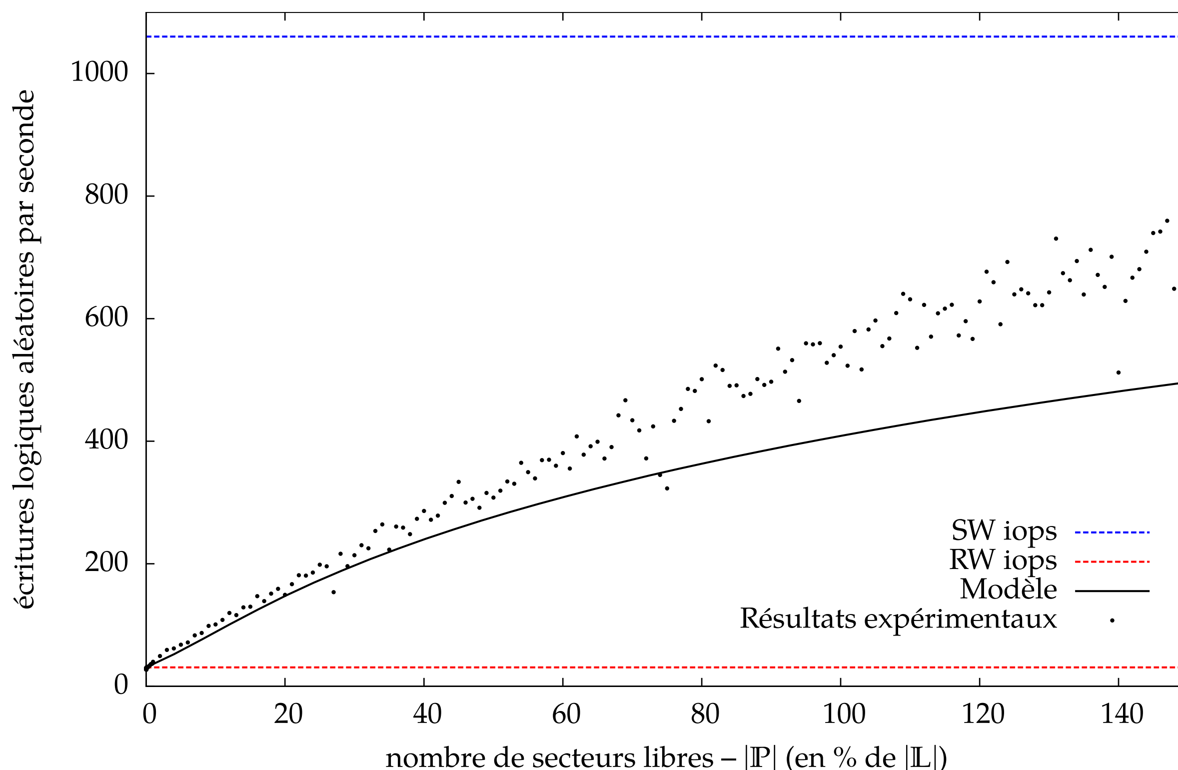


FIGURE 3.22: Performance des écritures aléatoires pour 100 000 secteurs logiques sur la clé USB

Les techniques existantes pour séquentialiser les accès n'apportent donc aucune amélioration et peuvent être supprimées.

Pour les lectures, trouver les correspondances entre les secteurs logiques et les secteurs physiques dans la tables de translation d'adresses est effectué en temps constant, et est négligeable comparé à la durée d'une lecture sur mémoire flash.

3.6.4 Résultats

Pour valider cette optimisation ainsi que le modèle détaillé dans la section précédente, l'algorithme d'écriture est testé avec deux des périphériques présentés dans la section 3.6.1. Ces tests consistent à évaluer le coût moyen d'écritures aléatoires en faisant varier la taille de la réserve de secteurs libres.

La figure 3.22 montre les résultats expérimentaux et les estimations du modèle pour une clé USB [8]. Pour comparer les performances avec les types d'accès conventionnels, le nombre d'opérations par seconde pour les écritures aléatoires (*RW iops*, 30) et séquentielles (*SW iops*, 1060) sont également reportées sur la figure.

Pour obtenir des performances équivalentes aux écritures séquentielles, des sacrifices conséquents doivent être acceptés en terme d'espace sur la mémoire flash. Dans nos expérimentations, 95% de l'efficactité des écritures séquentielles est atteint lorsque la reserve est à peu près trois fois plus importante que l'espace logique adressable. Cependant, des améliorations significatives par rapport aux écritures aléatoires sont atteintes avec un compromis acceptable : une amélioration d'un facteur dix pour un surcoût en espace mémoire flash de 50%.

8. Puce flash HYNIX HY27UG088G5B et un contrôleur ALCOR AU6983HL

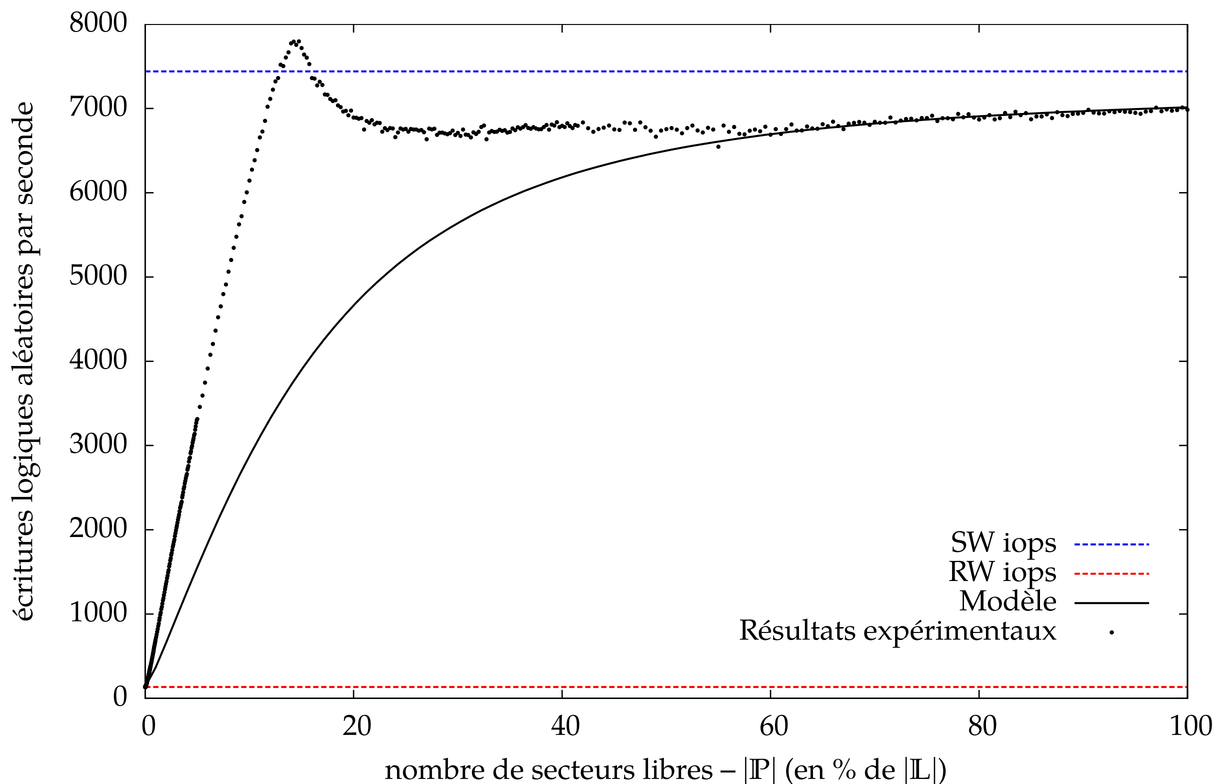


FIGURE 3.23: Performance des écritures aléatoires pour 200 000 secteurs logiques sur le SSD

Le SSD [9] montre des améliorations notables des performances avec des sacrifices moins importants. En effet, contrairement à la clé USB, la vitesse d'écriture du SSD est améliorée pour des distances inférieures à d_{max} = 256, au lieu de d_{max} = 32 pour la clé USB. Les résultats expérimentaux sont présentés sur la figure 3.23.

Un autre différence notable était suggérée par les mesures effectuée dans la section 3.6.1 : la fig. 3.24 se focalise sur les distances de faible valeur. Curieusement, un accès quasi-séquentiel avec une distance de 4 présente des performances relativement bonnes. Dans nos expérimentations, les meilleures performances sont obtenues pour une réserve d'à peu près 29000 secteurs libres, qui entraîne une distance moyenne – mais toujours aléatoire – de 4. Cette propriété permet d'atteindre les performances optimales à un coût moins important. En effet, nous obtenons 7796 iops pour des écritures aléatoires pour seulement 687 Ko de RAM et 14.5% de surcoût en espace mémoire flash, pour 800 MB d'espace utilisable. Par rapport aux 134 iops des écritures (physiques) aléatoires, les performances sont améliorées d'un facteur 58.

Déterminer la taille optimale de la réserve n'est pas simple, et dépend de la taille des secteurs. Avec des secteurs de 16 Kio, les résultats expérimentaux donnent une distance optimale de 2, et de 1.6 pour des secteurs de 32 Kio. Une explication possible à ce comportement est que l'*interleaving* favorise des distances supérieures à zéro pour accéder aux multiples puces flash internes aux périphérique en parallèle [Zhou et Meng, 2009]. Cette particularité n'est qu'un bénéfice supplémentaire facultatif car elle ne faisait pas partie de nos hypothèses de départ. Notre modèle devrait être plus proche des résultats d'une SSD ordinaire – sans ce « pic » de performances –, et présente tout de même des gains considérables, comme une amélioration d'un facteur 40 par rapport aux écritures aléatoires pour un surcoût de 25% de la mémoire.

9. Mtron MSD SATA3035-032

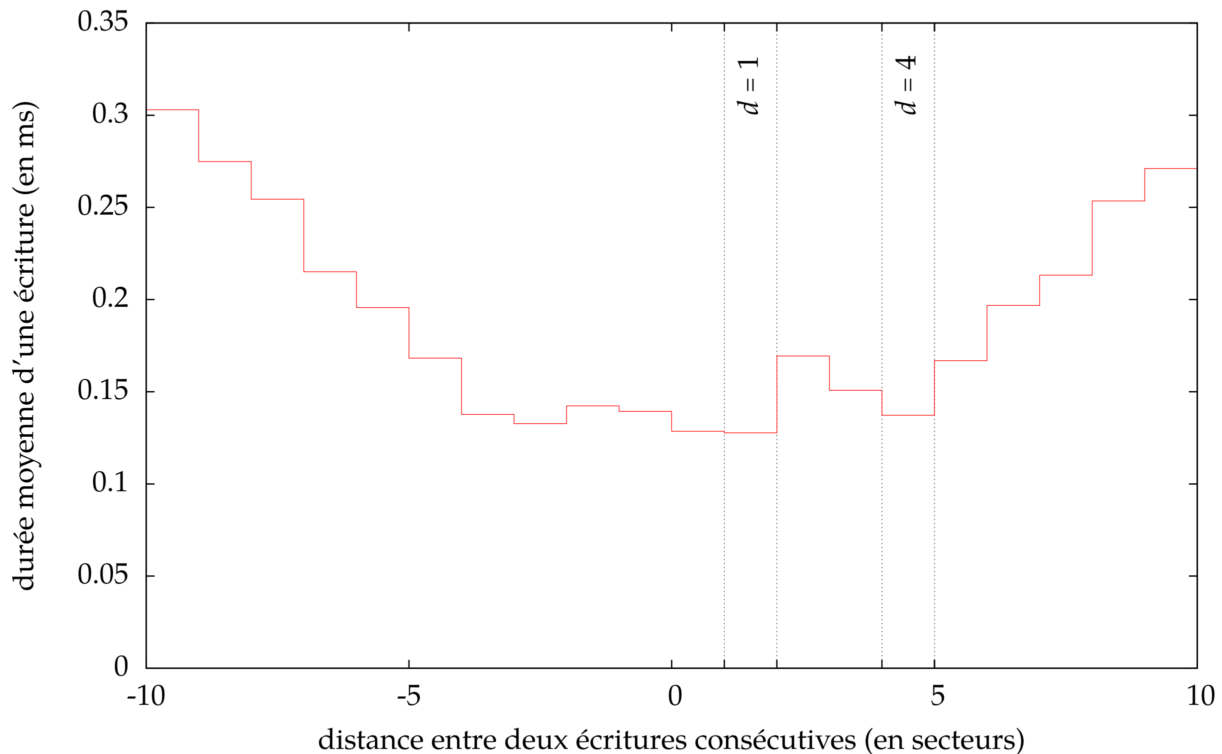


FIGURE 3.24: Petites variations de la distance pour les écritures quasi-séquentielles du SSD

3.7 Synthèse

Dans ce chapitre, nous avons tout d'abord rappelé les concepts technologiques des mémoires flash puis proposé un algorithme de placement des données améliorant significativement les performances des écritures aléatoires. Notre contribution met en avant l'importance de la localité spatiale pour les mémoires flash, en soulignant l'efficacité des accès « quasi-séquentiels ».

Par rapport aux écritures natives de la FTL, notre optimisation profite des ressources de l'hôte (RAM et capacité de traitement) pour améliorer l'efficacité des écritures aléatoires. Cette méthode permet d'ajuster localement les performances, contrairement aux mémoires flash qui présentent des comportements homogènes sur tout l'espace adressable.

Pour le SSD utilisé lors de nos expérimentations, nous avons obtenu un gain de performances d'un facteur 58 pour un surcoût en mémoire de 14.5%. Cette optimisation est également applicable aux mémoires flash de plus faible capacité, avec des résultats pour une clé USB montrant une amélioration d'un facteur 10 pour un surcoût de 50% de la capacité de la mémoire flash. De plus, nous avons proposé un modèle formel pour prédire les gains en performance.

Dans cette implémentation, les données sont néanmoins volatiles. En effet, la table de translation d'adresses, stockée en RAM, est nécessaire pour reconstruire l'association entre adresses logiques et adresses physiques. En l'état, cette optimisation est déjà applicable pour l'indexation ou les tables temporaires, où la volatilité est acceptable ; mais ce problème peut également être résolu par des solutions présentées dans l'état de l'art, comme FlashLogging [Chen, 2009] ou en intégrant l'adresse logique avec les données, de manière similaire à la FTL.

Dans le contexte des SGBD, cette adresse logique peut être remplacée par une métadonnée beaucoup plus simple car les tuples d'une table ne possèdent pas d'ordre particulier : en sup-

posant qu'aucun tuple ne soit scindé entre plusieurs pages, la récupération de l'identifiant de la table suffit pour reconstruire la base en cas de défaillance. Concrètement, une solution simple consiste à réserver quelques octets au début de chaque page pour y stocker cet identifiant. Cette solution a été intégrée à « Chronos », un moteur de stockage de SGBD pour MySQL, présenté dans le chapitre suivant.

Chapitre 4

Chronos : une approche “NoSQL” pour la gestion de données historiques sur flash

Plan du chapitre

4.1 Principe de fonctionnement **69**
4.1.1 Accès à la mémoire flash 69
4.1.2 Gestion des insertions 70
4.1.3 Durabilité 77
4.2 Implémentation **77**
4.2.1 Librairie 77
4.2.2 Intégration à MySQL 78
4.3 Résultats expérimentaux **80**
4.3.1 Sur disque dur 80
4.3.2 Sur mémoire flash 84
4.4 Synthèse **85**

Dans ce chapitre, nous décrivons Chronos, un moteur de stockage MySQL optimisé pour la gestion de données historiques sur mémoire flash. Ses performances sont ensuite analysées à l’aide du benchmark présenté au chapitre 2.

4.1 Principe de fonctionnement

Chronos est un entrepôt ordonné de paires clé-valeur (*ordered key-value store*) dédié à l'historisation de données sur mémoires flash. Il est utilisable en tant que librairie autonome, ou intégré à MySQL comme plugin (moteur de stockage).

Chronos inclut des optimisations portant sur les interactions avec la mémoire flash et sur la gestion des accès typiques du contexte industriel d'EDF, en mettant l'accent sur les insertions. Comme ces accès correspondent généralement à des *range queries*, Chronos utilise un B-tree pour stocker et indexer les données ; cette structure étant alors particulièrement adaptée, et déjà mise en œuvre dans de nombreux SGBD.

Paires clé-valeur

Tout comme Berkeley DB, Chronos est conçu pour manipuler des paires clé-valeur. Cette généralisation permet de faire abstraction de la nature des clés et des valeurs, et ainsi simplifier le fonctionnement du SGBD. Cette interface correspond également à l'utilisation des moteurs de stockage MySQL (cf. section 4.2.2.1), et facilite ainsi l'intégration de Chronos.

Dans le contexte d'EDF, la clé correspond à une concaténation de l'identifiant du repère et de la date (timestamp) de la mesure, tandis que la valeur intègre la mesure et sa qualité.

4.1.1 Accès à la mémoire flash

Chronos est dédié à l'utilisation de mémoires flash possédant de bonnes performances en écriture « quasi-séquentielles » et en lecture aléatoire. Chronos intègre donc l'algorithme d'écriture présenté dans le chapitre précédent. Pour cela, une abstraction simple des accès à la mémoire flash est celle des *nameless writes* [Arpaci-Dusseau *et al.*, 2010]. Cette interface permet d'écrire un secteur sans connaître à l'avance son adresse physique, celle-ci étant retournée par la méthode d'accès :

```
address ← nameless_write(data)
```

Les écritures « en place » restent possibles, en spécifiant une adresse de secteur à écrire, tout comme la lecture :

```
write(address, data)
data ← read(address)
```

La réutilisation des secteurs nécessite de pouvoir les libérer. Si la mémoire flash le permet, cette opération peut – de manière facultative – indiquer au périphérique que les données stockées à cette adresse sont obsolètes.

```
trim(address)
```

Pour utiliser les écritures « quasi-séquentielles » au niveau de cette interface, Chronos conserve la liste des secteurs libres dans une structure de données – un *set*, basé sur un arbre binaire de recherche – possédant trois opérations :

- `pop` pour récupérer l'adresse physique du prochain secteur à écrire, avec une complexité en $O(1)$
- `insert` pour ajouter un secteur à cette liste, avec une complexité en $O(\log n)$
- `erase` pour supprimer un secteur de cette liste, avec une complexité en $O(\log n)$

Les algorithmes 4.1, 4.2 et 4.4 présentent les interactions entre Chronos et cette structure de données.

ALGORITHME 4.1: nameless_write

```
   input : address, data
1  address ← list.POP()
2  flash.WRITE(address, data)
3  RETURN(address)
```

ALGORITHME 4.2: write

```
   input : address, data
1  list.ERASE(address) /* supprime l'adresse de la liste des secteurs libres, ne rien faire
   si elle n'en fait pas partie */
2  flash.WRITE(address, data)
```

ALGORITHME 4.3: read

```
   input : address
1  data ← flash.READ(address)
2  RETURN(data)
```

ALGORITHME 4.4: trim

```
   input : address
1  list.INSERT(address)
2  flash.TRIM(address) /* si le périphérique le supporte */
```

Les nameless writes sont utilisées pour l'ajout de nouveaux secteurs de données et pour leur mise à jour. L'utilisation d'un B-tree pour accéder aux données permet à Chronos de s'affranchir de la table de translation d'adresses, les adresses physiques des secteurs étant conservées dans l'index.

Cette approche journalisée est cependant incompatible avec les B+trees dont les feuilles de données sont chaînées par des références avec la feuille suivante – optimisation pour les *range queries* – car, à chaque modification ou insertion d'une feuille, le changement d'adresse entraîne la modification de tout l'arbre par récurrence.

Cette interface – qui entre dans le cadre plus général d'une approche journalisée – permet de dissocier le choix de l'emplacement des données du reste des traitements du SGBD. Il est alors envisageable de remplacer l'algorithme de sélection du secteur à écrire par une autre solution – par exemple des écritures purement séquentielles avec un mécanisme de ramasse-miettes –, voire de laisser ce choix au périphérique si celui-ci en est capable.

4.1.2 Gestion des insertions

4.1.2.1 Caractérisation des insertions

Dans un cas typique d'historisation de données, les clés sont de la forme (identifiant, date) avec les insertions effectuées par date croissante. Dans ce contexte, la relation d'ordre entre deux clés est donnée par :

$$(id_x, date_x) \leq (id_y, date_y) \Leftrightarrow (id_x \leq id_y) \vee (id_x = id_y \wedge date_x \leq date_y)$$

Si l'on considère l'exemple suivant, avec trois repères possédants respectivement les identifiants 1, 2 et 3 :

identifiant	date	valeur
1	09:00:10	5.43
1	09:00:12	11.07
1	09:00:18	13.94
2	09:00:11	-58.46
3	09:00:00	17.62
3	09:00:20	-8.32

Les insertions suivantes devraient être supérieures à 09:00:18 pour l'identifiant 1, 09:00:11 pour l'identifiant 2 et 09:00:20 pour l'identifiant 3. Dans cet exemple, les intervalles des insertions futures sont donc :

- $\left[(1, 09{:}00{:}18), (2, 09{:}00{:}11)\right]$ pour le repère 1
- $\left[(2, 09{:}00{:}11), (3, 09{:}00{:}00)\right]$ pour le repère 2
- $\left[(3, 09{:}00{:}20), (+\infty)\right]$ pour le repère 3

De manière générale, les insertions concernent les intervalles « vides » de la table – la table ne contient aucune donnée dans cet intervalle – avec en particulier les intervalles $\left[(\text{identifiant}, \text{date courante}), (\text{identifiant suivant}, \text{première date})\right]$ pour les valeurs courantes.

En généralisant, on peut considérer l'insertion d'un ensemble de n paires clé-valeur $(k_i, v_i)_{1 \leq i \leq n}$ avec des clés strictement croissantes ($k_i < k_{i+1}$) tel que la table ne possède au préalable aucune clé dans l'intervalle $[k_1, k_n]$.

Ce type d'accès correspond à l'utilisation de curseurs, autorisant le parcours séquentiel du contenu d'une table. De manière similaire aux curseurs de lecture, les curseurs d'écriture permettent d'insérer des données dont les clés se succèdent.

4.1.2.2 Curseurs d'écriture

La notion de curseur d'écriture pour Chronos est liée à celle des *fence keys* [Graefe, 2011], qui sont une copie des clés séparatrices des nœuds parents. En effet, un curseur d'écriture pour Chronos correspond directement à une feuille de l'arbre, et permet d'insérer des valeurs comprises dans l'intervalle de validité de cette feuille. Cet intervalle est délimité par les clés adjacentes aux clés contenues dans la feuille, dont les valeurs sont généralement stockées par les nœuds parents, à l'exception des cas particuliers $-\infty$ et $+\infty$.

Pour vérifier la cohérence des insertions, et éviter de parcourir l'arbre afin de vérifier l'intervalle de validité, ces bornes sont conservées avec la représentation en mémoire volatile d'une feuille. Bien que l'intervalle réel puisse être modifié par d'autres accès à l'arbre (typiquement la suppression d'un tuple associé à ces bornes), il s'agira dans tous les cas d'un minorant de l'intervalle de validité. Il est suffisant pour vérifier la validité d'une insertion sans faux positifs. Si cette condition n'est pas vérifiée, le curseur met à jour son intervalle de validité avant de refuser l'insertion.

L'algorithme 4.5 présente la gestion de l'intervalle de validité lors de l'ouverture d'un curseur, qui permet alors l'insertion de paires clé-valeur dans cet intervalle, sans avoir à parcourir l'arbre.

ALGORITHME 4.5: ouverture d'un curseur

```
   input : key
 1 node ← root
 2 lower_bound ← −∞
 3 upper_bound ← +∞
 4 while node is not a leaf do
 5 │ lower_bound ← MAX({keys k ∈ node : k < key} ∪ {lower_bound})
 6 │ upper_bound ← MIN({keys k ∈ node : k > key} ∪ {upper_bound})
 7 └ node ← node.SON(key) /* charger le nœud en mémoire si nécessaire */
 8 node.lower_bound ← lower_bound
 9 node.upper_bound ← upper_bound
10 node.nb_cursors++
11 cursor.leaf ← node
12 RETURN(cursor)
```

4.1.2.3 Algorithme d'insertion dans un B-tree

Avec ces curseurs, les insertions successives tentent d'écrire dans la même feuille de l'arbre que l'écriture précédente. Berkeley DB propose déjà ce type d'optimisation ; cependant, la division d'un nœud en deux requiert toujours la traversée de l'arbre, pour ajouter la clé médiane au nœud parent. De plus, cette opération laisse les deux nœuds à moitié pleins, alors que, pour l'historisation de données, aucune nouvelle écriture ne devrait concerner le nœud possédant les clés inférieures, correspondant aux dates passées.

Le deuxième point est relativement facile à corriger : lors d'une telle opération, au lieu de diviser un nœud en deux nœuds de tailles égales, Chronos alloue simplement un nouveau nœud ne contenant que la clé la plus grande. Cette optimisation n'est applicable que si les nouvelles insertions correspondent aux valeurs maximales de clés du nœud. Dans le cas contraire, celui-ci est divisé au niveau de la dernière clé ajoutée, pour que cette condition soit remplie pour les prochaines insertions.

Ce mécanisme est illustré par la figure 4.1 pour les feuilles de l'arbre, et par la figure 4.2 pour les nœuds internes. À la différence des feuilles, les nœuds internes font remonter la clés séparatrice sans en conserver une copie, mais l'algorithme reste similaire. L'insertion de la clé 27 correspond au cas défavorable – la clé est différente du maximum – tandis que l'insertion de la clé 29 correspond au cas favorable.

Pour éviter d'avoir un B-tree trop vide dans les niveaux les plus hauts de l'arbre, ce mécanisme n'est appliqué que pour les *n* niveaux les plus bas – en fait uniquement pour les feuilles –, l'algorithme usuel étant utilisé pour les niveaux supérieurs.

Pour éviter la traversée de l'arbre lors de la division d'un nœud, la représentation en mémoire volatile de chaque nœud possède un pointeur vers son nœud parent. Ces pointeurs ne sont donc jamais stockés sur la mémoire flash. Pour simplifier la gestion de l'arbre, lorsqu'un nœud est conservé en mémoire, son parent l'est également, récursivement jusqu'à la racine de l'arbre. Au niveau des feuilles de l'arbre, celles-ci ne sont représentées en mémoire volatile que si un curseur, de lecture ou d'écriture, leur est attaché.

Ce mécanisme permet également de compenser l'absence de référence entre les feuilles de l'arbre successives pour conserver une exécution efficace des *range queries*, sans avoir à parcourir l'arbre à partir de la racine pour accéder à la feuille suivante.

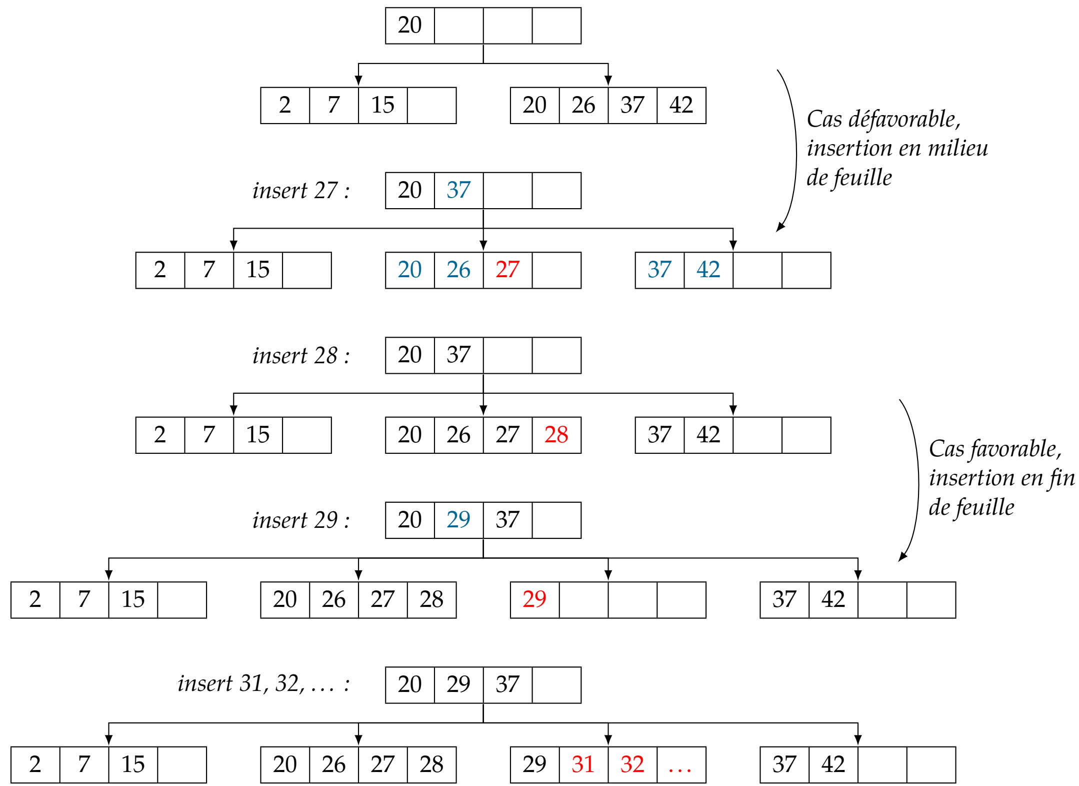


FIGURE 4.1: Division d'une feuille de l'arbre

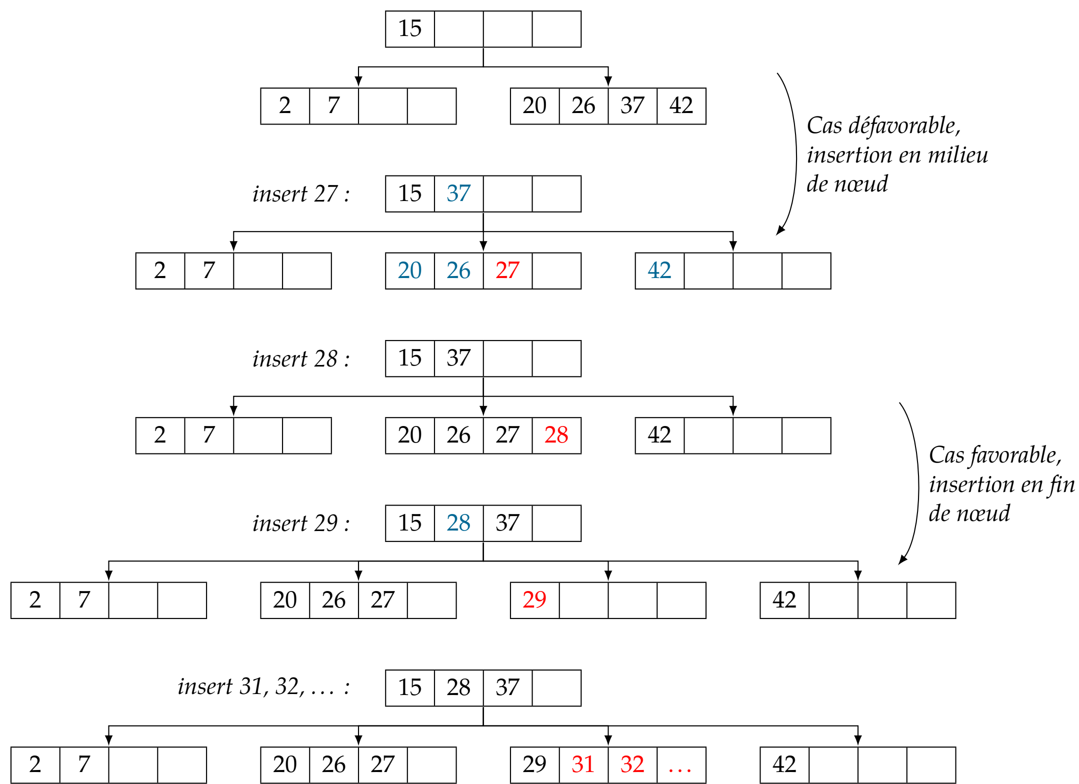


FIGURE 4.2: Division d'un nœud interne de l'arbre

ALGORITHME 4.6: insertion d'une paire clé-valeur dans une feuille

```
   input : key, value
 1 if key ≤ leaf.lower_bound or key ≥ leaf.upper_bound then
 2   RETURN(out of range)
 3 leaf.ADD(key, value)
 4 if leaf is overfull then
 5   if key = MAX({keys k ∈ leaf}) then /* cas favorable */
 6     split_key ← key
 7     new_leaf ← {tuples <k,v> ∈ leaf : k < split_key}
 8     leaf ← {tuples <k,v> ∈ leaf : k ≥ split_key} /* mise à jour de la
       feuille, qui ne contient alors que la dernière paire (clé, valeur) insérée */
 9     leaf.lower_bound ← split_key
10     address ← NAMELESS_WRITE(new_leaf)
11     leaf.parent.INSERT_LEFT(split_key, address) /* diviser récursivement
       les nœuds internes si nécessaire */
12   else/* cas défavorable */
13     split_key ← MIN({keys k ∈ leaf : k > key})
14     new_leaf ← {tuples <k,v> ∈ leaf : k ≥ split_key}
15     leaf ← {tuples <k,v> ∈ leaf : k < split_key}
16     leaf.upper_bound ← split_key
17     address ← NAMELESS_WRITE(new_leaf)
18     leaf.parent.INSERT_RIGHT(split_key, address) /* diviser récursivement
       les nœuds internes si nécessaire */
19 RETURN(ok)
```

L'algorithme 4.6 présente l'insertion d'une paire clé-valeur dans une feuille. Dans le cas favorable de la division d'une feuille, les tuples associés aux clés inférieures sont transférés vers une nouvelle feuille écrite sur mémoire flash, avec un taux de remplissage de 1. L'ancienne feuille ne contient alors plus qu'un seul tuple : le dernier inséré. Dans le cas défavorable, les clés supérieures sont transférées, et le taux de remplissage est indéfini.

4.1.2.4 Complexité

Pour les charges en insertion caractéristiques de l'historisation de données, Chronos s'efforce de minimiser les accès à la mémoire flash. Ces accès peuvent être quantifiés, et s'avèrent être proches de solutions optimales – sans considérer la compression des données. Bien que la mémoire flash soit le principal facteur limitant les performances, l'utilisation du processeur est également évaluée, afin d'identifier une éventuelle limitation pour le passage à l'échelle.

Soit un arbre avec des nœuds de taille moyenne F – chaque nœud contient F clés – et des feuilles de taille moyenne L – chaque feuille contient L paires clé-valeur. Cet arbre contient N paires clé-valeur au total. On considère le cas où l'on insère k paires clé-valeur $(key_i, value_i)_{0\leq i<k}$ successivement dans la table – par indice croissant.

Hypothèse 4.1. Les insertions sont effectuées par clés strictement croissantes.

$$key_0 < key_1 < \cdots < key_{k-1}$$

Hypothèse 4.2. La table ne contient initialement aucune clé dans l'intervalle $[key_0, key_{k-1}]$.

Les hypothèses 4.1 et 4.2 correspondent au cas favorable de l'algorithme d'écriture. En conséquence, le taux de remplissage (*fill factor*) des feuilles reste de 1. Pour les nœuds, on suppose que le nombre de clés qu'ils contiennent reste constant.

Hypothèse 4.3. Les tailles moyennes des nœuds et des feuilles F et L restent constantes.

Accès à la mémoire flash

Propriété 4.1. Après k insertions, $\frac{k}{L(F-1)}$ nœuds et $\frac{k}{L}$ feuilles sont ajoutés à l'arbre.

Démonstration. Initialement, l'arbre possède une hauteur h, A feuilles, B nœuds tels que :

$$h = \log_F\left(\frac{N}{L}\right) \qquad A = \frac{N}{L} \qquad B = \sum_{i=1}^{h} \frac{N}{LF^i}$$

Après k insertions, le nombre de paires contenues dans l'arbres est $N' = N + k$, qui possède alors une hauteur h', A' feuilles et B' nœuds tels que :

$$h' = \log_F\left(\frac{N'}{L}\right) \qquad A' = \frac{N'}{L} \qquad B' = \sum_{i=1}^{h'} \frac{N'}{LF^i}$$

On obtient alors (cf. annexe A.3) :

$$h' - h = \log_F\left(\frac{N'}{N}\right) \qquad A' - A = \frac{k}{L} \qquad B' - B = \frac{k}{L(F-1)}$$

Après k insertions, l'arbre contient donc bien $\frac{k}{L(F-1)}$ nœuds et $\frac{k}{L}$ feuilles supplémentaires.
□

Propriété 4.2. Pour k insertions, $\frac{kF}{L(F-1)} + \epsilon$ secteurs de la mémoire flash sont accédés en écriture, avec $\epsilon \leq \log_F\left(\frac{N}{L}\right)$.

Démonstration. Lors d'insertions avec un curseur ouvert – nos hypothèses impliquent que le curseur ouvert initialement reste toujours valide –, une écriture sur mémoire flash n'est effectuée que lors de la division d'une feuille ou d'un nœud (cf. algorithme 4.6). Le nombre d'écritures correspond donc au nombre de nouveaux éléments dans l'arbre.

D'après la propriété 4.1, pour k insertions, $\frac{k}{L(F-1)}$ nœuds et $\frac{k}{L}$ feuilles sont ajoutés à l'arbre.

$$\frac{k}{L(F-1)} + \frac{k}{L} = \frac{kF}{L(F-1)}$$

En ignorant les phases d'ouverture et de fermeture du curseur, le nombre d'écritures est donc de $\frac{kF}{L(F-1)}$.

Les seuls nœuds de l'arbre initial ayant pu être modifiés sont ceux accédés lors de l'ouverture du curseur, soit h nœuds – certains d'entre eux peuvent cependant ne pas avoir été modifiés.

Le nombre total de secteurs de la mémoire flash est donc bien majoré par $\frac{kF}{L(F-1)} + \log_F\left(\frac{N}{L}\right)$. □

En pratique, $F \gg 1$, donc $F/F-1 \simeq 1$ – ce qui est équivalent à dire que l'arbre est principalement constitué de feuilles [1]. Chronos écrit donc en moyenne un secteur toutes les $L(F-1)/F \simeq L$ insertions.

Utilisation du processeur

Propriété 4.3. La complexité amortie d'une insertion en temps processeur est constante, en $O\left(1+\frac{F}{L}\right)$.

Démonstration. L'ouverture et fermeture du curseur correspondent à des parcours d'arbre « classiques », possèdant une complexité en $O(log(N))$ et $O(log(N')) = o(k)$ [Graefe, 2011], qui disparaissent donc dans le calcul de la complexité amortie.

En ignorant les phases d'initialisation et de déinitialisation (ouverture et fermeture de curseur), la complexité CPU provient d'un ensemble d'opérations élémentaires :

- l'insertion d'une paire clé-valeur dans une feuille, effectuée en $O(1)$,
- la division d'une feuille, effectuée en $O(F+L)$ – sans considérer les divisons successives dues à la récursion,
- la division d'un nœud, effectuée en $O(F)$ – sans considérer les divisons successives dues à la récursion.

Pour la division d'une feuille, L'algorithme 4.6 permet de suivre le calcul de complexité. Seules les opérations suivantes possèdent une complexité significative en temps processeur, nos hypothèses correspondant au cas favorable :

- opération 7 en $O(L)$: copie (d'une partie) du contenu de la feuille,
- opération 11 en $O(F)$: insertion d'une clé dans le nœud parent.

La complexité est similaire pour la division de nœuds, en $O(2F) = O(F)$.

D'après les résultats précédents, $\frac{k}{L(F-1)}$ nœuds et $\frac{k}{L}$ feuilles sont divisées au total – récursions incluses – pour k insertions. La complexité CPU totale C est donc de :

$$
\begin{aligned}
C &= k \times O(1) + \frac{k}{L(F-1)} \times O(F) + \frac{k}{L} \times O(F+L) \\
&= O\left(k\left(1+\frac{F^2}{L(F-1)}\right)\right)
\end{aligned}
$$

or, $F > 2 \Rightarrow 1 > \frac{F}{F-1} > 2$, donc :

$$
C = O\left(k\left(1+\frac{F}{L}\right)\right), \text{ et } \frac{C}{k} = O\left(1+\frac{F}{L}\right)
$$

□

1. Par exemple, dans le cas du benchmark, avec des tuples de 17 octets (dont 12 pour la clé) et des secteurs de 4096 octets, $L = 240$ et $F = 256$, donc 99.6% des éléments de l'arbre sont des feuilles.

Pour le calcul de cette complexité, le seul paramètre dépendant de N correspond à l'ouverture et à la fermeture du curseur, qui devient négligeable lorsque le nombre d'insertion k devient suffisamment grand.

4.1.2.5 Désordre

La latence du réseau peut introduire un certain désordre pour les insertions, c'est à dire que la position de chaque tuple du flux entrant peut être différente de sa position dans un flux idéal. Nous considérons que ce désordre est limité par une borne supérieure ϵ.

Soient les dates des tuples insérés $date_i$ avec i représentant leur ordre d'arrivée :

$$\forall(i,j), j < i - \epsilon \Rightarrow date_j < date_i$$

En conséquence, il suffit d'avoir une fenêtre de taille ϵ pour pouvoir réordonner les tuples. Pour cela, les curseurs d'écriture possèdent un buffer dont la taille est un compromis entre les performances (les tuples arrivant trop « tard » nécessitent l'ouverture d'un nouveau curseur) et la quantité de données perdues en cas de défaillance (les données du buffer ne sont pas conservées).

4.1.3 Durabilité

Dans le contexte de l'historisation de données, le déni de service en cas de défaillance est généralement une cause suffisante de perte de données, les capteurs ayant une capacité de stockage limitée. Cependant, Chronos possède également quelques limites concernant la durabilité des données.

La hiérarchie de chaque curseur ouvert est maintenue en mémoire volatile. Pour les nœuds internes, comme ceux-ci ne sont pas nécessaires pour reconstruire la table, cela n'impacte pas la durabilité des données. Cependant les données des feuilles de l'arbre stockées en RAM sont perdues. Une partie de ces données peut être retrouvée à partir d'anciennes versions de cette feuille pouvant être conservées sur la mémoire flash.

Les données de la fenêtre d'insertion pour les tuples du flux entrant sont en particulier perdues en cas de défaillance. De manière générale, ces pertes de données concernent les insertions récentes et sont limitées, par curseur, à la taille d'une feuille plus la taille de le fenêtre de tri : $L + \epsilon$.

La reconstruction de la table à partir des données stockées sur la mémoire flash nécessite d'en parcourir l'intégralité, ce qui peut être particulièrement coûteux lorsqu'une grande quantité de données sont archivées. De plus, des données effacées peuvent être récupérées lors de cette reconstruction. Il est cependant possible de définir des points de sauvegarde (*checkpoints*) où une version de l'index est figée. Il suffit alors de parcourir les secteurs écrits par la suite pour reconstruire la table.

4.2 Implémentation

4.2.1 Librairie

En mode librairie, Chronos gère une base de donnée unique associée à un fichier, qui peut être un périphérique d'entrée-sortie, comme une mémoire flash.

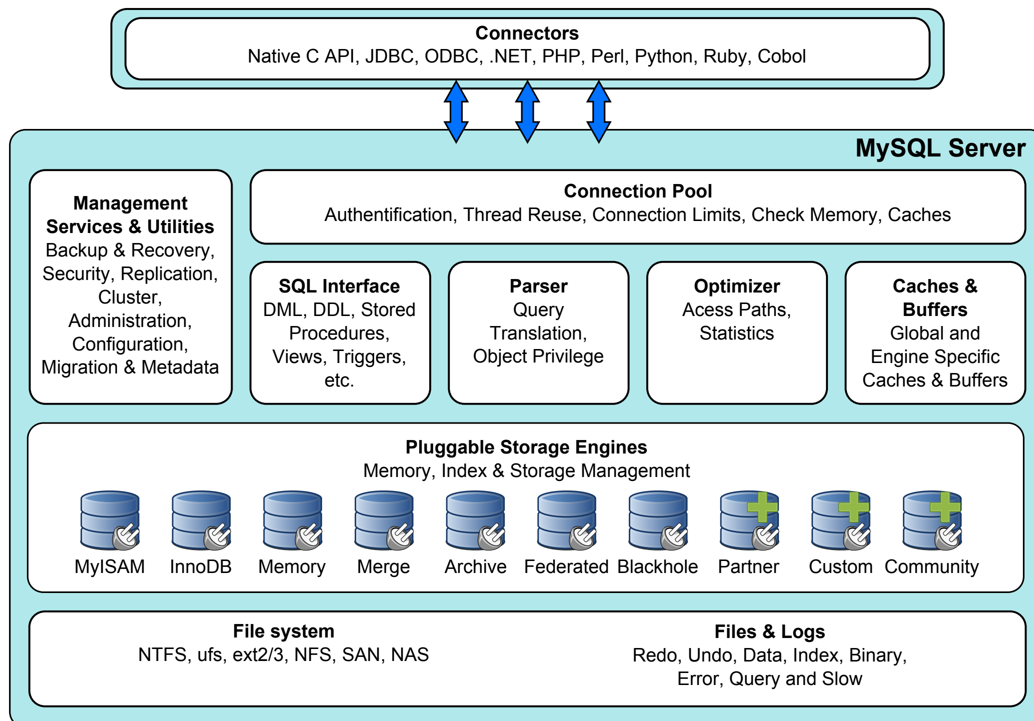


FIGURE 4.3: Architecture globale de MySQL (Oracle, 2011b)

Le premier niveau de gestion de la base concerne les tables. Les opérations possibles sont la création, la suppression, l'ouverture et la fermeture. Chaque table est associée à une chaîne de caractère servant d'identifiant unique.

Après son ouverture, une table permet l'ouverture et la fermeture de curseurs. Ces curseurs peuvent ensuite être utilisés pour insérer, lire ou supprimer des paires clé-valeur, ou mettre à jour la valeur associée à une clé. La lecture permet de parcourir la table par valeurs de clés croissantes ou décroissantes.

4.2.2 Intégration à MySQL

4.2.2.1 Description des moteurs de stockage MySQL

MySQL est un système de gestion de base de données complexe. Bien que ce projet soit open-source, la lecture et l'analyse des centaines de milliers de lignes de codes qui le compose peuvent s'avérer délicates.

Les moteurs de stockage servent principalement d'intermédiaires entre l'optimiseur de requêtes de MySQL et le système de fichier (ou éventuellement un périphérique de stockage sans système de fichier). Chaque table de la base de données est associée à un moteur de stockage, instancié à sa création ou à son ouverture. MySQL effectue ensuite des opérations sur la table par l'intermédiaire de l'interface abstraite des moteurs de stockage. La figure 4.3 donne une vue d'ensemble de l'architecture de MySQL.

L'interface des moteurs de stockage définit en particulier des accès pour :

- la gestion des tables : création, suppression, ouverture et fermeture ;
- la gestion unitaire des tuples, pour :
  - `write_row` : insérer un tuple dans la table ;

  - `delete_row` : supprimer un tuple de la table ;
  - `update_row` : mettre à jour un tuple de la table ;
- le parcours de la table (*full table scan* : initialisation et lecture du tuple suivant ;
- la gestion des index, pour :
  - `index_read` : récupérer un tuple grâce à une valeur de clé dans l'index ;
  - `index_read_last` : récupérer le dernier tuple dans l'ordre spécifié par l'index, correspondant à une valeur de clé donnée (en particulier des préfixes d'index sur des colonnes multiples) ;
  - `index_first` : récupérer le premier élément de l'index ;
  - `index_last` : récupérer le dernier élément de l'index ;
  - `index_next` : récupérer le tuple suivant dans l'ordre spécifié par l'index ;
  - `index_next_same` : récupérer le tuple suivant possédant la même valeur de clé ;
  - `index_prev` : récupérer le tuple précédent dans l'ordre spécifié par l'index ;
- la gestion des transactions.

Le contexte d'historisation de données d'EDF n'imposant pas d'utiliser un moteur de stockage transactionnel, les méthodes associées à cette dernière fonctionnalité ne sont pas détaillées.

La commande SQL `HANDLER` permet de récupérer les données avec un accès direct aux interfaces des moteurs de stockage pour le parcours de la table et la lecture de l'index. Cette commande est décrite en annexe A.2 pour détailler les principes liés à ces accès.

Un moteur de stockage doit informer MySQL de ses fonctionnalités, et reporter des statistiques sur les données contenues dans ses tables pour que l'optimiseur puisse établir les plan d'exécution des requêtes le plus efficacement possible. Ces statistiques sont des valeurs approximatives pour :

- le nombre de tuples contenus dans la table
- le nombre de tuples contenus dans un intervalle (de clé) donné
- le nombre de blocs de données à lire pour une lecture complète de la table
- le nombre de blocs de données à lire pour lire n tuples à partir de m intervalles d'un index

Cette architecture permet de mutualiser la gestion des communications avec les clients, l'analyse syntaxique des requêtes (parsing), la génération du plan d'exécution, la réplication ainsi que la plupart des commandes d'administration (à l'exception de quelques opérations sur les tables : optimize, analyze, etc.). Les moteurs de stockage se chargent eux de la persistance des données, de la gestion de la concurrence, des transactions, des index et des buffers pour les opérations d'entrée-sortie avec le support de stockage. Cependant, la majorité de ces fonctionnalités sont optionnelles ; au niveau de l'implémentation, on peut obtenir un moteur de stockage minimaliste en surchargeant peu de fonction. Les moteurs de stockage *example* et *blackhole* sont des exemples de tels moteurs de stockage.

Blackhole est un moteur de stockage qui accepte des données en entrée sans les conserver. Les sélections retournent systématiquement un ensemble vide. Cette fonctionnalité peut être utile pour les bases de données distribuées où les données sont répliquées automatiquement (ce qui nécessite la définition d'une table) sans être stockées localement.

Example est un modèle de moteur de stockage ne faisant rien. Il permet la création de tables, mais aucune donnée ne peut être insérée ou extraite, le moteur refuse ces opérations. Celui-ci sert d'exemple (stub) pour le développement d'un nouveau moteur de stockage.

4.2.2.2 Interface avec MySQL

L'interface des moteurs de stockage est simplement une sur-couche à Chronos, permettant de traduire les appels MySQL. Cette interface est relativement proche des méthodes d'accès de Chronos. La seule différence notable concerne les insertions, pour lesquelles MySQL ne permet pas l'association de curseurs – ces opération sont effectuées de manière unitaire uniquement.

Le moteur de stockage doit donc gérer un ensemble de curseurs d'écriture, et identifier le curseur à utiliser pour chaque insertion. Dans le cas général, il est possible de retrouver le curseur à utiliser en les conservant triés par intervalle de validité. Cette opération possède alors une complexité en $O(\log k)$, avec k le nombre de curseurs ouverts.

Dans le cas de l'historisation de données, une structure de données plus adaptée permet de retrouver le curseur associé avec une complexité en $O(1)$. En effet, pour les insertions de valeurs horodatées avec la date courante, il suffit de maintenir N curseurs, avec N correspondant au nombre de repères actifs, et de trouver le curseur correspondant à l'aide de l'identifiant du repère – ou, dans un cas plus général, un préfixe de clé. Retrouver le curseur à partir d'un identifiant est faisable en temps constant avec une table de hachage par exemple. Dans un cadre plus général, il n'est pas toujours possible d'extraire un préfixe – associé à un curseur – à partir de la clé à insérer.

4.3 Résultats expérimentaux

Bien que Chronos ait été conçu pour fonctionner avec des mémoires flash, des expérimentations avec des disques durs ont également été effectuées, avec le même cas d'utilisation que lors du comparatif présenté dans le chapitre 2. Le même benchmark est ensuite exécuté, en remplaçant cette fois le système de stockage original (trois disques durs de 73 GB 10K avec un contrôleur RAID 5 de 256 MB) par une clé USB (Kingston DataTraveler R500 64 Go).

Les résultats des expérimentations sont reportés dans les tableaux 4.1 et 4.2, respectivement sur disques durs et sur mémoire flash. Les figures 4.4 et 4.5 donnent ensuite un aperçu de capacités de traitement de chaque système. Ces résultats sont alors regroupés par catégories et reportés sur les figures 4.6 et 4.7.

4.3.1 Sur disque dur

Utilisé comme librairie, Chronos dépasse les 2 millions d'insertions par seconde, soit 12× mieux que Berkeley DB, 34× mieux qu'InfoPlus.21 et 115× mieux qu'InnoDB. Cependant, tout comme Berkeley DB, l'utilisation d'une librairie permet de s'affranchir de la communication entre processus, ce qui peut améliorer significativement les performances .

Intégré à MySQL, les performances de Chronos sont d'environ 28 500 insertions par seconde. Cependant, les performances des insertions avec MySQL sont limitées – au niveau du processeur – par les traitements effectués indépendamment du moteur de stockage. En effet, le moteur de stockage Blackhole n'effectue aucune action lors des différentes opérations et présente des performances similaires, plafonnées à environ 30 000 insertions par seconde. Les performances pour les autres types de requêtes sont très proches de celles en mode librairie [2].

Les accès unitaires (mises à jour – R1.1 et R1.2 – et extraction de valeurs courantes – R11.1 et R11.2) correspondent à des accès aléatoires pour tous les systèmes, qui présentent alors des

2. En dehors des insertions (R0.1 et R0.2) sur disque dur, les accès au périphérique de stockage sont toujours le facteur limitant.

TABLE 4.1: Temps d'exécution des requêtes sur disque dur

Requête (quantité)	Temps d'exécution (en s)			
	InfoPlus.21	MySQL	Berkeley DB	Chronos
R0.1 (×500 M)	8 003.4	24 671.7	2 849.6	235.7
R0.2 (×500 M)	7 085.8	24 086.0	3 115.8	222.3
R1.1 (×1 M)	16 762.8	12 239.5	9 031.5	8 402.1
R1.2 (×1 M)	16 071.3	13 088.2	9 348.5	8 522.5
R2.1 (×1 k)	267.6	410.4	693.0	1 263.2
R2.2 (×1 k)	215.1	284.5	655.4	1 018.4
R3.1 (×1 k)	207.5	186.6	531.4	1 214.3
R3.2 (×1 k)	166.8	181.8	533.2	994.9
R4 (×1 k)	210.3	192.6	536.8	1 215.8
R5 (×1 k)	189.3	185.7	514.0	1 115.2
R6 (×1 k)	189.1	191.9	513.1	1 090.9
R7 (×1 k)	234.0	234.2	507.7	1 059.3
R8 (×1 k)	231.2	277.7	506.5	1 054.5
R9 (×100)	1 640.6	1 710.0	4 877.7	4 878.4
R10 (×100)	1 688.8	7 660.7	4 977.5	4 898.7
R11.1 (×1)	9.5×10^{-4}	1.15	2.75	1.06
R11.2 (×1)	2.8×10^{-4}	1.13	4.81	0.90

TABLE 4.2: Temps d'exécution des requêtes sur mémoire flash

Requête (quantité)	Temps d'exécution (en s)			
	InfoPlus.21	MySQL	Berkeley DB	Chronos
R0.1 (×500 M)	34 412.2	26 468.4	10 887.5	637.8
R0.2 (×500 M)	26 848.9	25 443.8	11 137.2	492.6
R1.1 (×1 M)	820 011	229 096	53 690.3	109 450
R1.2 (×1 M)	731 862	251 751	56 397.2	94 505.5
R2.1 (×1 k)	341.0	1 066.2	2 839.4	419.8
R2.2 (×1 k)	201.3	271.9	1 744.5	231.1
R3.1 (×1 k)	102.6	168.4	104.1	260.1
R3.2 (×1 k)	92.8	159.8	101.6	228.2
R4 (×1 k)	104.1	171.0	102.2	261.0
R5 (×1 k)	96.9	169.9	101.0	253.8
R6 (×1 k)	96.3	170.6	101.0	254.3
R7 (×1 k)	204.8	226.9	101.0	257.9
R8 (×1 k)	201.9	288.0	100.6	252.1
R9 (×100)	1 992.9	1 722.1	999.5	2 554.1
R10 (×100)	1 989.1	7 497.2	1 001.5	2 484.7
R11.1 (×1)	9.9×10^{-4}	0.31	0.39	0.37
R11.2 (×1)	2.7×10^{-4}	0.38	0.69	0.39

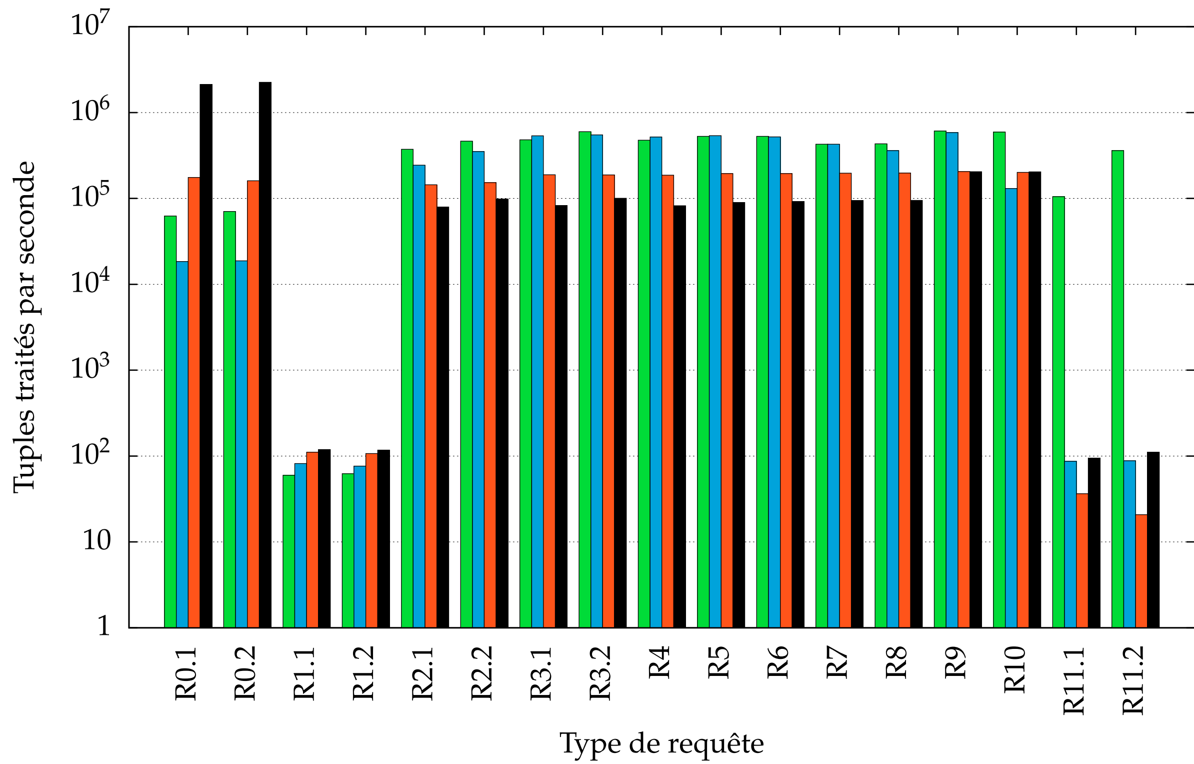


FIGURE 4.4: Capacités de traitement sur disque dur

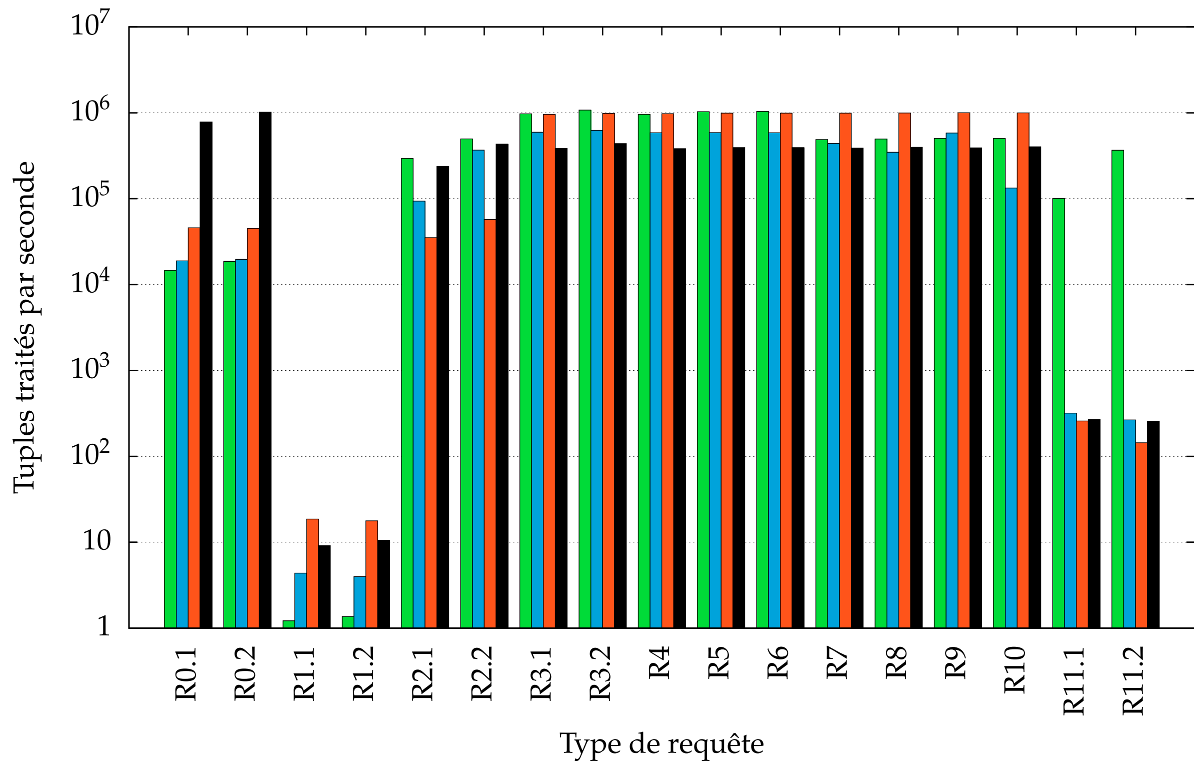
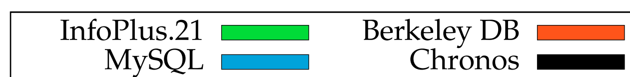


FIGURE 4.5: Capacités de traitement sur mémoire flash



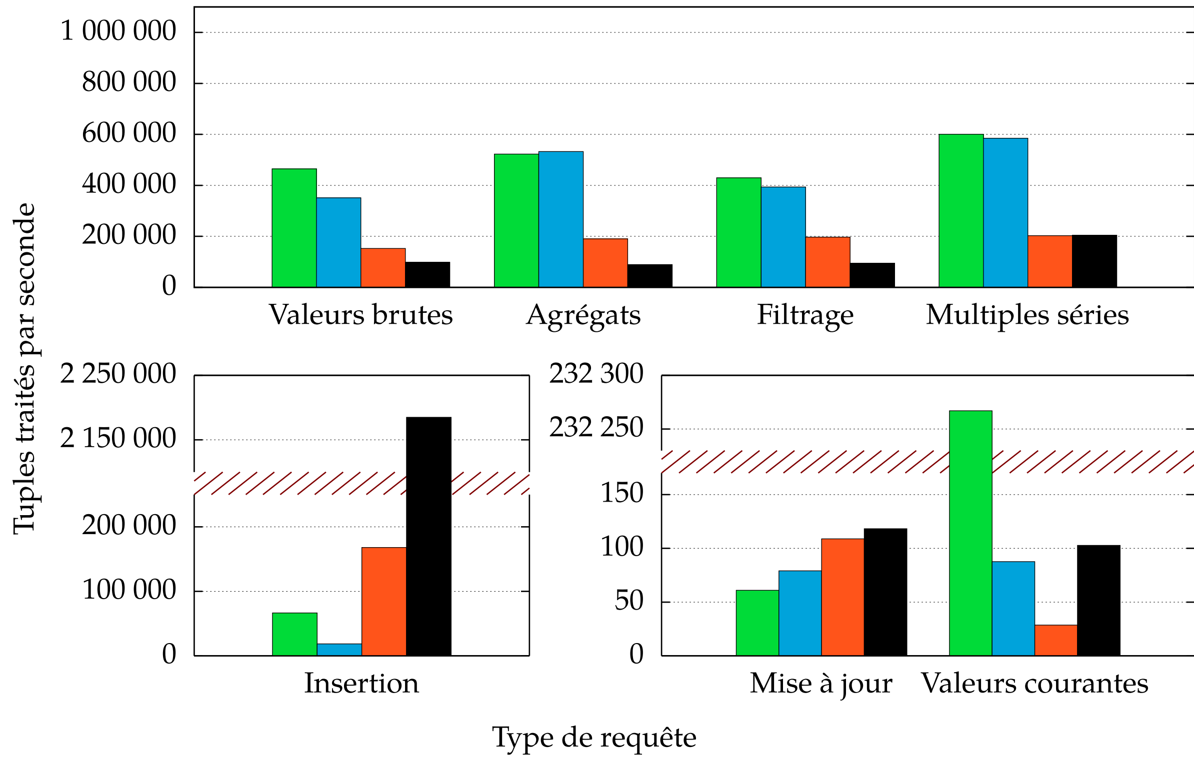


FIGURE 4.6: Capacités de traitement par catégorie sur disque dur

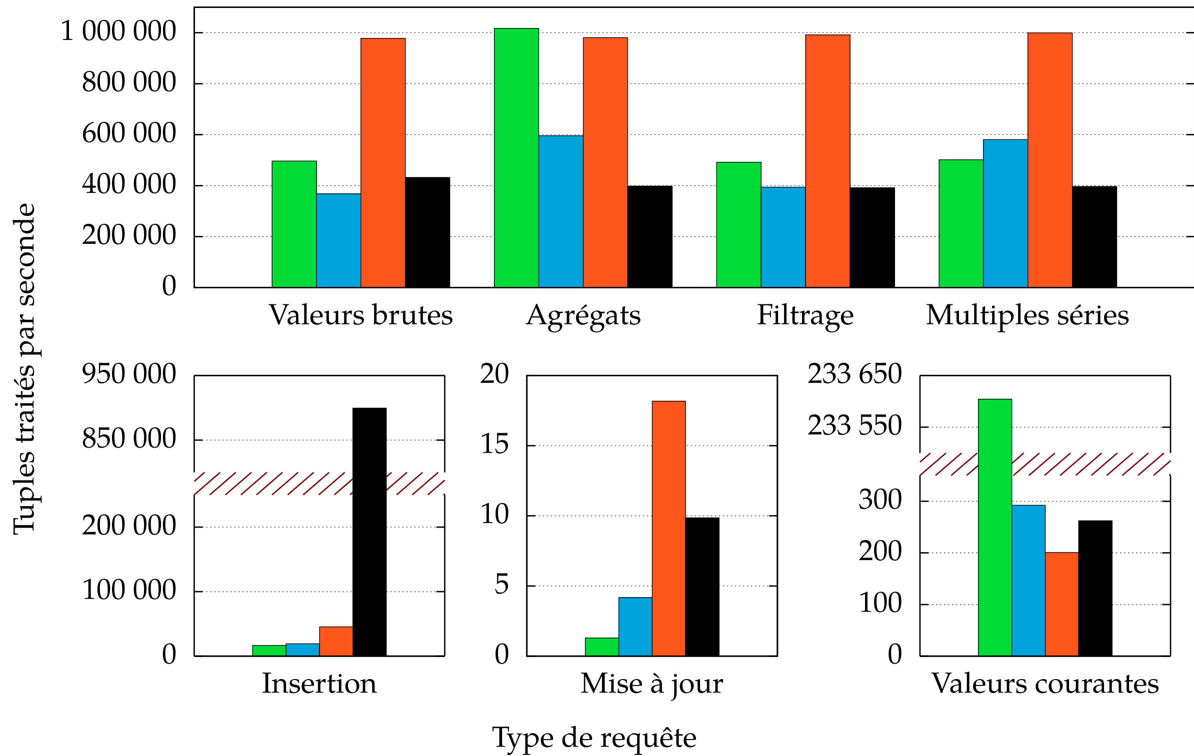


FIGURE 4.7: Capacités de traitement par catégorie sur mémoire flash

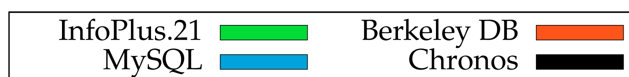


TABLE 4.3: Synthèse des performances sur mémoire flash

Catégorie	Requêtes	Meilleurs résultats
Insertion	R0.1 et R0.2	Chronos
Mise à jour	R1.1 et R1.2	Berkeley DB
Valeurs brutes	R2.1* et R2.2	Berkeley DB
Agrégats	R3.1, R3.2, R4, R5 et R6	InfoPlus.21
Filtrage	R7 et R8	Berkeley DB
Multiples séries	R9 et R10*	Berkeley DB
Valeurs courantes	R11.1 et R11.2	InfoPlus.21

performances similaires – à l'exception notable d'InfoPlus.21 pour l'extraction de valeurs courantes, cf. section 2.5.1.

Pour les autres requêtes, Chronos fait appel à des lectures aléatoires, alors que les autres systèmes se basent sur des lectures séquentielles. L'interrogation de tous les repères – requêtes R9 et R10 – diminue légèrement cet impact car, pour Chronos, les données de différents repères pour une même date possèdent une forte localité spatiale sur le disque. Ces deux requêtes sont donc traitées environ deux fois plus rapidement que les autres requêtes d'analyse, où Chronos est environ 2× plus lent que Berkeley DB, soit environ 5× moins bien qu'InfoPlus.21 ou MySQL.

Les accès en lecture étant basés sur des lectures aléatoires, les performances de ces requêtes sont naturellement bien meilleures avec des mémoires flash.

4.3.2 Sur mémoire flash

Le tableau 4.3 reprend les différentes catégories de requêtes et indique le système présentant les meilleurs résultats. Quelques ajustements sont effectués par rapport aux résultats expérimentaux : comme pour les disques durs, seule la requête R2.2 est prise en compte pour l'extraction de valeurs brutes, et la requête R10 n'est pas intégrée aux résultats de MySQL. De plus, Berkeley DB possède une interface d'accès simplifiée, utilisée de manière identique pour les requêtes R2 à R10 ; les performances de celles-ci devraient donc être similaires. Or, R2.1 et R2.2 sont particulièrement lentes, probablement à cause des mises à jour effectuées préalablement. Une ré-exécution de ces deux types de requêtes montre que leurs performances sont effectivement semblables aux autres *range queries* (R3-R8). Pour la suite de cette analyse, ces seconds résultats sont considérés à la place des premières expérimentations. Avec ces ajustements, la figure 4.7 donne un aperçu des différences de performances en regroupant les requêtes similaires.

Utilisé comme librairie, Chronos atteint 900 000 insertions par seconde, soit 20× mieux que Berkeley DB, 47× mieux qu'InnoDB et 54× mieux qu'InfoPlus.21.

Les mises à jour (R1.1 et R1.2) sont particulièrement lentes pour tous les systèmes : les capacités de traitement vont de 1 à 19 tuples par seconde. Sur la clé USB utilisée, les écritures aléatoires sont peu performantes, ce qui explique ces résultats – de plus, dans l'implémentation actuelle, Chronos n'utilise pas la redirection d'écriture pour les mises à jour, qui sont effectués *sur place*.

Avec l'utilisation de mémoires flash, les mises à jour dégradent significativement les requêtes suivantes pour tous les systèmes. En particulier, le temps d'exécution de la requête R2.1 par Chronos est supérieur de 61% par rapport aux autres requêtes similaires. Comme Chronos n'effectue aucun traitement différé, cette dégradatation est probablement due à l'activité en ar-

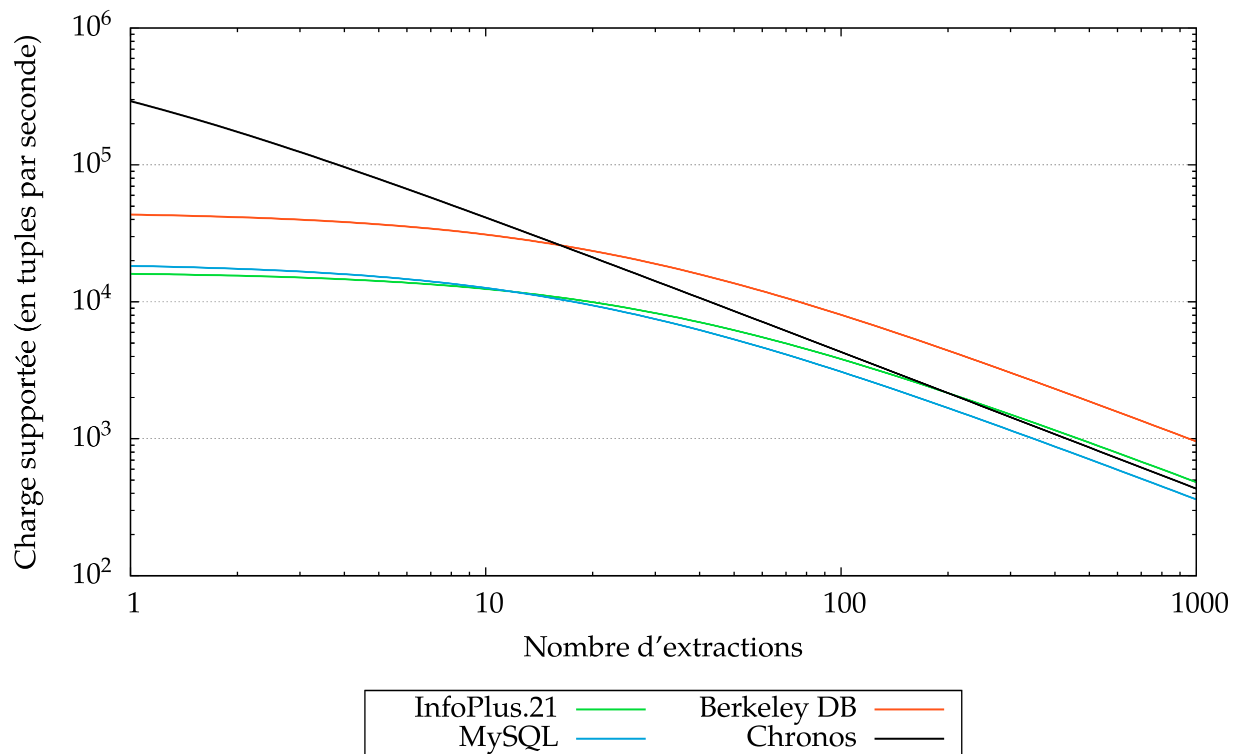


FIGURE 4.8: Performances en fonction du ratio extraction/insertion

rière plan de la mémoire flash – des mécanismes de ramasse-miettes internes au périphérique – causées par les écritures lors des requêtes R1.1 et R1.2.

Chronos sur mémoire flash voit ses performances globales améliorées d'un facteur 3.9× pour les requêtes R2 à R10 par rapport à l'utilisation de disques durs. De même, Berkeley DB profite des mémoires flash en traitant ces requêtes 5.3× plus rapidement. En comparaison, MySQL ne gagne que 6%, et InfoPlus.21 44%.

Pour les *range queries*, Chronos traite en moyenne 400 000 tuples par seconde, soit 1.3× moins que MySQL (500 000 tuples par seconde), 1.8× moins qu'InfoPlus.21 (730 000 tuples par seconde), et 2.4× moins que Berkeley DB (980 000 tuples par seconde)

Pour l'archivage temporaire par les IGCBox, les données sont insérées et extraites au moins une fois, mais peuvent également faire l'objet de requêtes en provenance d'un système de supervision local à la centrale. Le ratio extraction/insertion reste cependant faible – typiquement compris entre 1 et 2. Face à cette charge, Chronos se distingue en proposant des performances 4× à 18× meilleures par rapport aux autres solutions. Par exemple, si chaque donnée est extraite en moyenne 1.5 fois, Chronos peut supporter une charge de 220 000 tuples par seconde en insertion, alors que Berkeley DB n'en supporte que 42 000, 18 000 pour MySQL et 16 000 pour InfoPlus.21. Les charges maximales supportées en fonction de ce ratio extraction/insertion sont reportées sur la figure 4.8.

4.4 Synthèse

Chronos est un SGBD simple, adapté au contexte de l'historisation de données sur mémoires flash, et optimisé pour les insertions. De part sa conception, celui-ci possède également quelques

limites pouvant compromettre son utilisation dans d'autres contextes : la durabilité des données n'est garantie que dans une certaine mesure, et les accès – en insertion comme en extraction – sont basés sur les *range queries*.

Les résultats expérimentaux obtenus avec Chronos permettent cependant de valider les optimisations mises en œuvre : les écritures « quasi-séquentielles » sur mémoires flash, l'adaptation des algorithmes de division des nœuds d'un B-tree et l'utilisation de curseurs pour les insertions. En effet ce SGBD possède les performances les plus élevées en insertion parmi les différentes solutions testées. En contrepartie, ces optimisations ont un impact sur les performances en extraction, en particulier car celles-ci correspondent à des lectures aléatoires sur le support de stockage. Cette dégradation est significative avec l'utilisation de disques durs, mais est limitée sur mémoire flash.

Dans ce contexte, Chronos est une solution compétitive, en particulier lorsque les insertions correspondent à une proportion importante de la charge soumise au SGBD.

CHAPITRE 5

CONCLUSION ET PERSPECTIVES

L'historisation de données industrielles est un contexte applicatif pour lequel les SGBDR sont peu utilisés, au profit de produits "de niche" spécialisés pour les traitements spécifiques des applications sous-jacentes. Les raisons de cette segmentation du marché sont historiques : elles se basent sur les contraintes de performances auxquelles les systèmes d'historisation sont soumis. Pourtant, même si ces progiciels d'historisation ont été conçus pour supporter une charge particulièrement importante en insertion de données, les résultats du benchmark que nous avons proposé mettent en évidence des performances du même ordre de grandeur pour les SGBDR et systèmes NoSQL : avec une configuration adaptée, MySQL et Berkeley DB pourraient supporter les charges identifiées à EDF et donc pourraient être compétitifs du point de vue "gestion de données".

Cependant, la comparaison fonctionnelle met en avant la force des progiciels d'historisation : ces fonctionnalités « métier » tendent à prendre le pas sur les performances. Les clients métiers et la facilité d'intégration (communication avec les autres systèmes, configuration originale adaptée au contexte, etc.) constituent une différence importante avec les autres catégories de SGBD. Néanmoins, pour s'affranchir du coût de licence ou pour intégrer le système d'historisation dans un environnement où les ressources matérielles sont limitées voire où un progiciel d'historisation n'est pas utilisable (système d'exploitation non supporté, etc.), un SGBD conventionnel peut être mis en œuvre dans ce contexte applicatif.

Les IGCBox – mini PC industriels utilisés pour la remontée et l'archivage des données des centrales hydrauliques – en sont un exemple, avec MySQL pour archiver les données. Pour autant, MySQL n'a été que peu optimisé pour ce cas d'application – archivage court terme avec une charge importante en insertion – et dispose donc d'une marge de progression substantielle, ce qui laisse penser que ces approches sont compétitives. Par ailleurs, les mémoires flash, comme celles inclues dans les IGCBox, diffèrent significativement des disques durs et amènent à reconsidérer l'architecture même des SGBD.

Avec Chronos, un moteur de stockage MySQL dédié à l'historisation de données et à l'utilisation de mémoires flash, nous avons mis l'accent sur les performances en insertion. Pour cela, nous avons en particulier identifié un algorithme d'écriture « quasi-séquentiel » efficace afin de compenser la lenteur des écritures aléatoires de certaines catégories de mémoires flash. Nous avons aussi exploité les spécificités de l'historisation de données, où les accès sont généralement basés sur des séquences de clés contiguës.

Les résultats expérimentaux montrent un gain significatif pour les insertions par rapport à des solutions équivalentes (Berkeley DB, InfoPlus.21 et MySQL), d'un facteur 20 à 54, tandis que les performances en extraction restent de même ordre de grandeur, avec un différence d'un facteur 1.3 à 2.4.

Perspectives

Approximation des traitements Une des caractéristiques de Chronos est d'ignorer la nature des données stockées. De nombreuses optimisations sont donc envisageables en considérant plus finement les caractéristiques des données de l'historisation. En effet, un partie importante de ces données constitue une représentation approximative d'une réalité physique, approximations dues à l'échantillonnage et aux imprécisions de mesure. Les données peuvent ainsi, dans une certaine mesure, perdre en précision sans perdre leur signification, ce qui permet d'une part de les stocker de manière compacte à l'aide d'une compression avec perte d'information, mais également de fournir des réponses approximatives aux requêtes.

La consultation d'un historique sur plusieurs décennies pour visualiser les données par exemple peut se baser sur une représentation imprécise si cela permet d'améliorer les temps de réponse. Ce type de requête peut adopter un modèle multirésolution, où le premier résultat approximatif voit sa précision améliorée incrémentalement par des réponses successives.

Matérialisation d'agrégats La matérialisation d'agrégats est une mise en œuvre simple d'une approche multirésolution. Ces agrégats sont précalculés pour différents intervalles de clés – des intervalles temporels pour l'historisation – comme par exemple les multiples niveaux des nœuds de l'index, où chaque nœud conserverait la valeur de l'agrégat pour son sous-arbre.

La matérialisation d'agrégats simples (count, min, max, etc.) permet de traiter plus rapidement certaines requêtes : calculs d'agrégats, génération d'aperçus, filtrage des valeurs – où la connaissance du minimum et du maximum suffit pour déterminer si il est nécessaire de considérer des résolutions plus fines – etc. Il est cependant envisageable de matérialiser des agrégats plus complexes, comme par exemple la somme pondérée par le temps, utile pour représenter l'énergie produite par exemple (somme pondérée de la puissance).

Distribution L'utilisation de grappes de serveurs pour les systèmes d'historisation est une perspective à laquelle EDF s'intéresse particulièrement : du fait de l'importance croissante de ces systèmes dans le SI industriel, leur disponibilité devient une propriété fondamentale ; d'autant plus qu'une indisponibilité prolongée peut entraîner une perte de données. Pour l'instant, l'amélioration des performances permise par ce type d'architecture n'est pas requise pour supporter les charges identifiées, mais les performances d'un seul serveur peuvent devenir un facteur limitant si l'on considère de nouveaux cas d'application potentiels comme la centralisation des données issues des compteurs intelligents.

Chronos n'inclut pas de mécanisme de distribution sur plusieurs serveurs. Néanmoins, un SGBD distribué pourrait intégrer Chronos pour la gestion locale – sur une seule machine – des données, comme le fait Voldemort avec Berkeley DB [Burd, 2011].

Par ailleurs, les mémoires flash sont disponibles dans de nombreux formats, ciblant aussi bien les systèmes embarqués que les serveurs, avec une utilisation de plus en plus fréquente dans de nombreux domaines. L'approche générique de Chronos peut ainsi être adaptée à d'autres contextes présentant des problématiques similaires, c'est à dire une charge importante en insertion et des accès localisés sur des valeurs de clés contiguës, aussi bien en insertion qu'en extraction.

BIBLIOGRAPHIE

AGRAWAL, D., GANESAN, D., SITARAMAN, R., DIAO, Y. et SINGH, S. (2009). Lazy-Adaptive Tree : An Optimized Index Structure for Flash Devices. *Proceedings of the VLDB Endowment*, 2(1):361–372. [57]

AGRAWAL, N., PRABHAKARAN, V., WOBBER, T., DAVIS, J. D., MANASSE, M. et PANIGRAHY, R. (2008). Design Tradeoffs for SSD Performance. *In USENIX'08 : 2008 USENIX Annual Technical Conference*, pages 57–70. [49, 58]

AILAMAKI, A., DEWITT, D. J. et HILL, M. D. (2002). Data Page Layouts for Relational Databases on Deep Memory Hierarchies. *VLDB Journal*, 11(3):198–215. [55]

ARASU, A., BABCOCK, B., BABU, S., DATAR, M., ITO, K., MOTWANI, R., NISHIZAWA, I., SRIVASTAVA, U., THOMAS, D., VARMA, R. et WIDOM, J. (2003). STREAM : The Stanford Stream Data Manager. *Data Engineering Bulletin*, 26(1):19–26. [21]

ARASU, A., CHERNIACK, M., GALVEZ, E., MAIER, D., MASKEY, A. S., RYVKINA, E., STONEBRAKER, M. et TIBBETTS, R. (2004). Linear Road : A Stream Data Management Benchmark. *In VLDB'04 : 30th International Conference on Very Large Data Bases*, pages 480–491. [22]

ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H. et PRABHAKARAN, V. (2010). Removing The Costs Of Indirection in Flash-based SSDs with NamelessWrites. *In HotStorage'10 : 2nd Workshop on HotTopics in Storage and File Systems*, pages 1–5. [50, 69]

ASPEN TECHNOLOGY (2007). *Database Developer's Manual*. Version 2006.5. [6, 9, 10]

BAUMANN, S., de GIEL NIJS, STROBEL, M. et SATTLER, K.-U. (2010). Flashing Databases : Expectations and Limitations. *In DaMoN'10 : 6th International Workshop on Data Management on New Hardware*, pages 9–18. [51]

BIRRELL, A., ISARD, M., THACKER, C. et WOBBER, T. (2007). A Design for High-Performance Flash Disks. *Operating Systems Review*, 41(2):88–93. [46, 51, 58]

BLOOM, B. H. (1970). Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426. [57]

BONNET, P. et BOUGANIM, L. (2011). Flash Device Support for Database Management. *In CIDR'11 : 5th Biennial Conference on Innovative Data Systems Research*, pages 1–8. [49]

BOUGANIM, L., JÓNSSON, B. T. et BONNET, P. (2009). uFLIP : Understanding Flash IO Patterns. *In CIDR'09 : 4th Biennial Conference on Innovative Data Systems Research*. [50, 51, 54, 59, 60]

BREWER, J. E. et GILL, M., éditeurs (2008). *Nonvolatile Memory Technologies with Emphasis on Flash : A Comprehensive Guide to Understanding and Using Flash Memory Devices*. Wiley. [36]

BURD, G. (2011). Is Berkeley DB a NoSQL solution ? `http://blogs.oracle.com/berkeleydb/entry/is_berkeley_db_a_nosql_solutio`. [88]

CHANG, L.-P. (2008). Hybrid Solid-State Disks : Combining Heterogeneous NAND Flash in Large SSDs. *In ASP-DAC'08 : 13th Asia and South Pacific Design Automation Conference*, pages 428–433. [39]

CHANG, Y.-H., HSIEH, J.-W. et KUO, T.-W. (2007). Endurance Enhancement of Flash-Memory Storage Systems : An Efficient Static Wear Leveling Design. *In DAC'07 : 44th Design Automation Conference*, pages 212–217. [44]

CHARDIN, B., PASTEUR, O. et PETIT, J.-M. (2011). An ftl-agnostic layer to improve random write on flash memory. *In FlashDB'11 : 1st International Workshop on Flash-based Database Systems*, pages 214–225. [58]

CHEN, F., KOUFATY, D. A. et ZHANG, X. (2009). Understanding Intrinsic Characteristics and System Implications of Flash Memory based Solid State Drives. *In SIGMETRICS'09 : 11th International Joint Conference on Measurement and Modeling of Computer Systems*, pages 181–192. [51]

CHEN, S. (2009). FlashLogging : Exploiting Flash Devices for Synchronous Logging Performance. *In SIGMOD'09 : 35th International Conference on Management of Data*, pages 73–86. [56, 66]

CHUNG, T.-S., PARK, D.-J., PARK, S., LEE, D.-H., LEE, S.-W. et SONG, H.-J. (2009). A survey of Flash Translation Layer. *Journal of Systems Architecture*, 55(5-6):332–343. [47]

COOKE, J. (2007). The Inconvenient Truths of NAND Flash Memory. Flash Memory Summit. [39, 43]

DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P. et VOGELS, W. (2007). Dynamo : Amazon's Highly Available Key-value Store. *SIGOPS Operating Systems Review*, 41(6):205–220. [11]

EMRICH, T., GRAF, F., KRIEGEL, H.-P., SCHUBERT, M. et THOMA, M. (2010). On the Impact of Flash SSDs on Spatial Indexing. *In DaMoN'10 : 6th International Workshop on Data Management on New Hardware*, pages 3–8. [57]

GAL, E. et TOLEDO, S. (2005). Algorithms and Data Structures for Flash Memories. *ACM Computing Surveys*, 37(2):138–163. [44, 50]

GRAEFE, G. (2011). Modern B-Tree Techniques. *Foundations and Trends in Databases*, 3(4):203–402. [71, 76]

GRAY, J. et FITZGERALD, B. (2008). Flash Disk Opportunity for Server Applications. *ACM Queue*, 6(4):18–23. [52]

GRUPP, L. M., CAULFIELD, A. M., COBURN, J., SWANSON, S., YAAKOBI, E., SIEGEL, P. H. et WOLF, J. K. (2009). Characterizing Flash Memory : Anomalies, Observations, and Applications. *In MICRO'09 : 42nd International Symposium on Microarchitecture*, pages 24–33. [39]

GUPTA, A., KIM, Y. et URGAONKAR, B. (2009). DFTL : A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings. *In ASPLOS'09 : 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 229–240. [49]

INVENSYS SYSTEMS (2007). *Wonderware Historian 9.0 High-Performance Historian Database and Information Server*. [9, 10]

IROM, F. et NGUYEN, D. N. (2008). Radiation Tests of Highly Scaled High Density Commercial Nonvolatile Flash Memories. Rapport technique, National Aeronautics and Space Administration. [41]

JO, H., KANG, J.-U., PARK, S.-Y., KIM, J.-S. et LEE, J. (2006). FAB : Flash-Aware Buffer Management Policy for Portable Media Players. *IEEE Transactions on Consumer Electronics*, 52(2):485–493. [52]

KANG, D., JUNG, D., KANG, J.-U. et KIM, J.-S. (2007). μ-Tree : An Ordered Index Structure for NAND Flash Memory. *In EMSOFT'07 : 7th International Conference on Embedded Software*, pages 144–153. [56]

KANG, J.-U., JO, H., KIM, J.-S. et LEE, J. (2006). A Superblock-based Flash Translation Layer for NAND Flash Memory. *In EMSOFT'06 : 6th International Conference on Embedded software*, pages 161–170. [48]

KIM, H. et AHN, S. (2008). BPLRU : A Buffer Management Scheme for Improving Random Writes in Flash Storage. *In FAST'08 : 6th USENIX Conference on File and Storage Technologies*, pages 1–14. [53]

KIM, J., KIM, J. M., NOH, S. H., MIN, S. L. et CHO, Y. (2002). A Space-Efficient Flash Translation Layer for Compact Flash Systems. *IEEE Transactions on Consumer Electronics*, 48(2):366–375. [47]

KIM, J., OH, Y., KIM, E., CHOI, J., LEE, D. et NOH, S. H. (2009). Disk Schedulers for Solid State Drives. *In EMSOFT'09 : 7th International Conference on Embedded Software*, pages 295–304. [52]

KIM, Y.-R., WHANG, K.-Y. et SONG, I.-Y. (2010). Page-Differential Logging : An Efficient and DBMS-independent Approach for Storing Data into Flash Memory. *In SIGMOD'10 : 36th International Conference on Management of Data*, pages 363–374. [55, 58]

KOLTSIDAS, I. et VIGLAS, S. D. (2008). Flashing Up the Storage Layer. *Proceedings of the VLDB Endowment*, 1(1):514–525. [54]

KREPS, J. (2009). Project Voldemort : Scaling Simple Storage at LinkedIn. `http://blog.linkedin.com/2009/03/20/project-voldemort-scaling-simple-storage-at-linkedin`. [11]

LAKSHMAN, A. et MALIK, P. (2009). Cassandra - A Decentralized Structured Storage System. White paper. [11]

LEE, S., SHIN, D., KIM, Y.-J. et KIM, J. (2008a). LAST : Locality-Aware Sector Translation for NAND Flash Memory-Based Storage Systems. *Operating Systems Review*, 42(6):36–42. [48]

LEE, S.-W. et MOON, B. (2007). Design of Flash-Based DBMS : An In-Page Logging Approach. *In SIGMOD'07 : 33rd International Conference on Management of Data*, pages 55–66. [55, 58]

LEE, S.-W., MOON, B. et PARK, C. (2009). Advances in Flash Memory SSD Technology for Enterprise Database Applications. *In SIGMOD'09 : 35th International Conference on Management of Data*, pages 863–870. [51, 58]

LEE, S.-W., MOON, B., PARK, C., KIM, J.-M. et KIM, S.-W. (2008b). A Case for Flash Memory SSD in Enterprise Database Applications. *In SIGMOD'08 : 34th International Conference on Management of Data*, pages 1075–1086. [52]

LEE, S.-W., PARK, D.-J., CHUNG, T.-S., LEE, D.-H., PARK, S. et SONG, H.-J. (2007). A Log Buffer-Based Flash Translation Layer Using Fully-Associative Sector Translation. *ACM Transactions on Embedded Computing Systems*, 6(3). [48]

LI, Y., HEY, B., LUO, Q. et YI, K. (2009). Tree Indexing on Flash Disks. *In ICDE'09 : 25th International Conference on Data Engineering*, pages 1303–1306. [57]

LI, Y., XU, J., CHOI, B. et HU, H. (2010). StableBuffer : Optimizing Write Performance for DBMS Applications on Flash Devices. *In CIKM'10 : 19th International Conference on Information and Knowledge Management*, pages 339–348. [54]

MASTER, N. M., ANDREWS, M., HICK, J., CANON, S. et WRIGHT, N. J. (2010). Performance Analysis of Commodity and Enterprise Class Flash Devices. *In PDSW'10 : 5th Petascale Data Storage Workshop*. [51]

MICHELONI, R., CRIPPA, L. et MARELLI, A., éditeurs (2010). *Inside NAND Flash Memories*. Springer. [35, 37, 38, 39, 43]

MICHELONI, R., MARELLI, A. et RAVASIO, R. (2008). *Error Correction Codes for Non-Volatile Memories*. Springer. [43]

MIELKE, N., MARQUART, T., WU, N., KESSENICH, J., BELGAL, H., SCHARES, E., TRIVEDI, F., GOODNESS, E. et NEVILL, L. (2008). Bit Error Rate in NAND Flash Memories. *In IRPS'08 : 46th International Reliability Physics Symposium*, pages 9–19. [39, 40, 41, 42, 43]

MODBUS-IDA (2006). *MODBUS Application Protocol Specification*. [9]

NATH, S. et GIBBONS, P. B. (2008). Online Maintenance of Very Large Random Samples on Flash Storage. *Proceedings of the VLDB Endowment*, 1(1):970–983. [55]

NATH, S. et KANSAL, A. (2007). FlashDB : Dynamic Self-tuning Database for NAND Flash. *In IPSN'07 : 6th International Conference on Information Processing in Sensor Networks*, pages 410–419. [56]

OLSON, M. A., BOSTIC, K. et SELTZER, M. I. (1999). Berkeley DB. *In FREENIX'99 : 1999 USENIX Annual Technical Conference, FREENIX Track*, pages 183–191. [6, 11, 20]

OPC FOUNDATION (2003). *Data Access Custom Interface Standard*. [9]

ORACLE (2011a). *Berkeley DB Programmer's Reference Guide*. Version 11.2.5.2. [10]

ORACLE (2011b). *MySQL 5.5 Reference Manual*. [6, 10, 11, 78]

OSISOFT (2009). *PI Server System Management Guide*. Version 3.4.380. [9, 10]

OU, Y., HÄRDER, T. et JIN, P. (2009). CFDC : A Flash-aware Replacement Policy for Database Buffer Management. *In DaMoN'09 : 5th International Workshop on Data Management on New Hardware*, pages 15–20. [53]

PARK, S.-Y., JUNG, D., KANG, J.-U., KIM, J.-S. et LEE, J. (2006). CFLRU : A Replacement Algorithm for Flash Memory. *In CASES'06 : 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 234–241. [53]

PASTEUR, O. (2007). Etat de l'art des approches actuelles sur les données historisées d'exploitation des centrales. Publication interne EDF, H-P1D-2007-01076-FR. [3]

PASTEUR, O. et LÉGER, S. (2007). Résultats de l'étude sur les capacités d'historisation du SGBD MySQL. Publication interne EDF, H-P1D-2007-02670-FR. [4]

POSTEL-PELLERIN, J. (2008). *Fiabilité des Mémoires Non-Volatiles de type Flash en architectures NOR et NAND*. Thèse de doctorat, Université de Provence Aix-Marseille 1. [36, 37, 39, 41]

PUGH, W. (1990). Skip Lists : A Probabilistic Alternative to Balanced Trees. *Communications of the ACM*, 33(6):668–676. [62]

RAJIMWALE, A., PRABHAKARAN, V. et DAVIS, J. D. (2009). Block Management in Solid-State Devices. *In USENIX'09 : 2009 USENIX Annual Technical Conference*. [51]

SEOL, J., SHIM, H., KIM, J. et MAENG, S. (2009). A Buffer Replacement Algorithm Exploiting Multi-Chip Parallelism in Solid State Disks. *In CASES'09 : 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 137–146. [53]

STEVENS, C. E. (2009). *Working Draft ATA/ATAPI Command Set - 2 (ACS-2)*. American National Standards Institute, Inc. Revision 2. [49]

STOICA, R., ATHANASSOULIS, M., JOHNSON, R. et AILAMAKI, A. (2009). Evaluating and Repairing Write Performance on Flash Devices. *In DaMoN'09 : 5th International Workshop on Data Management on New Hardware*, pages 9–14. [55, 58]

TRANSACTION PROCESSING PERFORMANCE COUNCIL (2007). *TPC Benchmark C Standard Specification*. [22]

TRANSACTION PROCESSING PERFORMANCE COUNCIL (2008). *TPC Benchmark H Standard Specification*. [22]

TSIROGIANNIS, D., HARIZOPOULOS, S., SHAH, M. A., WIENER, J. L. et GRAEFE, G. (2009). Query Processing Techniques for Solid State Drives. *In SIGMOD'09 : 35th International Conference on Management of Data*, pages 59–72. [55, 56]

WANG, Y., GODA, K. et KITSUREGAWA, M. (2009). Evaluating Non-In-Place Update Techniques for Flash-Based Transaction Processing Systems. *In DEXA'09 : 20th International Conference on Database and Expert Systems Applications*, pages 777–791. [52, 58]

WU, C.-H., KUO, T.-W. et CHANG, L. P. (2007). An Efficient B-Tree Layer Implementation for Flash-Memory Storage Systems. *ACM Transactions on Embedded Computing Systems*, 6(3). [56]

YIN, S., PUCHERAL, P. et MENG, X. (2009). A Sequential Indexing Scheme for Flash-Based Embedded Systems. *In EDBT'09 : 12th International Conference on Extending Database Technology*, pages 588–599. [57]

ZHOU, D. et MENG, X. (2009). RS-Wrapper : Random Write Optimization for Solid State Drive. *In CIKM'09 : 18th International Conference on Information and Knowledge Management*, pages 1457–1460. [65]

ZUCK, A., BARZILAY, O. et TOLEDO, S. (2009). NANDFS : A Flexible Flash File System for RAM-Constrained Systems. *In EMSOFT'09 : 7th International Conference on Embedded Software*, pages 285–294. [50]

ANNEXES

A.1 Procédures stockées du benchmark pour MySQL

```
CREATE PROCEDURE R9 (
  IN start BIGINT UNSIGNED,
  IN end BIGINT UNSIGNED
)
READS SQL DATA
BEGIN
  DECLARE labelmax VARCHAR(40) DEFAULT NULL;
  DECLARE maxcount BIGINT UNSIGNED DEFAULT 0;
  DECLARE count BIGINT UNSIGNED;
  DECLARE lab VARCHAR(40);
  DECLARE id INT UNSIGNED;
  DECLARE th1 FLOAT;
  DECLARE th2 FLOAT;
  DECLARE no_more_rows BOOLEAN;
  DECLARE ana_cursor CURSOR FOR
    SELECT Label, AnaId, Threshold1, Threshold2 FROM ana;
  DECLARE CONTINUE HANDLER FOR NOT FOUND
    SET no_more_rows = TRUE;
  OPEN ana_cursor;
  the_loop: LOOP
    FETCH ana_cursor INTO lab, id, th1, th2;
    IF no_more_rows THEN
      CLOSE ana_cursor;
      LEAVE the_loop;
    END IF;
    SELECT count(*) INTO count FROM anavalues
      WHERE AnaId = id AND (Value > th2 OR Value < th1)
      AND Date BETWEEN start AND end;
    IF count > maxcount THEN
      SET maxcount = count;
      SET labelmax = lab;
    END IF;
  END LOOP the_loop;
  SELECT labelmax, maxcount;
END
```

LISTING A.1: Procédure stockée MySQL pour R9

Cette procédure stockée ne correspond pas exactement à la définition SQL de la requête R10 : l'analyse s'arrète un tuple plus tôt. Étant donné la quantité de données traitée par les requêtes du benchmark, cette différence a un impact négligeable sur le temps de traitement.

```
CREATE PROCEDURE R10subquery (
  IN id INT UNSIGNED,
  IN start BIGINT UNSIGNED,
  IN end BIGINT UNSIGNED,
  IN interval INT UNSIGNED,
  OUT count BIGINT UNSIGNED
)
READS SQL DATA
BEGIN
  DECLARE newdate BIGINT UNSIGNED;
  DECLARE olddate BIGINT UNSIGNED;
  DECLARE no_more_rows BOOLEAN;
  DECLARE anavalues_cursor CURSOR FOR
    SELECT Date FROM anavalues WHERE AnaId = id
    AND Date BETWEEN start AND end ORDER BY Date ASC;
  DECLARE CONTINUE HANDLER FOR NOT FOUND
    SET no_more_rows = TRUE;
  OPEN anavalues_cursor;
  SET count = 0;
  FETCH anavalues_cursor INTO olddate;
  the_loop: LOOP
    FETCH anavalues_cursor INTO newdate;
    IF no_more_rows THEN
      CLOSE anavalues_cursor;
      LEAVE the_loop;
    END IF;
    IF (newdate - olddate) > interval THEN
      SET count = count + 1;
    END IF;
    SET olddate = newdate;
  END LOOP the_loop;
END
```

LISTING A.2: Procédure stockée MySQL pour R10 (1)

```
CREATE PROCEDURE R10 (
  IN start BIGINT UNSIGNED,
  IN end BIGINT UNSIGNED
)
READS SQL DATA
BEGIN
  DECLARE idmax INT UNSIGNED;
  DECLARE maxcount BIGINT UNSIGNED DEFAULT 0;
  DECLARE interval INT UNSIGNED;
  DECLARE count BIGINT UNSIGNED;
  DECLARE id INT UNSIGNED;
  DECLARE no_more_rows BOOLEAN;
  DECLARE ana_cursor CURSOR FOR SELECT AnaId, Period FROM ana;
  DECLARE CONTINUE HANDLER FOR NOT FOUND
    SET no_more_rows = TRUE;
  OPEN ana_cursor;
  the_loop: LOOP
    FETCH ana_cursor INTO id, interval;
    IF no_more_rows THEN
      CLOSE ana_cursor;
      LEAVE the_loop;
    END IF;
    SET interval = interval * 1000;
    CALL R10subquery(id, start, end, interval, count);
    IF count > maxcount THEN
      SET maxcount = count;
      SET idmax = id;
    END IF;
  END LOOP the_loop;
  SELECT idmax, maxcount;
END$$
```

LISTING A.3: Procédure stockée MySQL pour R10 (2)

A.2 Accès aux moteurs de stockage de MySQL en SQL

La commande SQL `HANDLER` permet de récupérer les données avec un accès direct aux interfaces des moteurs de stockage. Cette commande sert ici à introduire les principes liés à ces accès. Deux modes d'accès principaux sont identifiés, le premier utilise un index, alors que le second correspond à un parcours de la table.

A titre d'exemple pour cette section, on considérera la table `Mesures (INT identifiant, DATETIME date, FLOAT valeur)` avec un index sur sa clé primaire `(identifiant, date)` désigné par le mot-clé `PRIMARY`. Cette table contient deux tuples :

identifiant	date	valeur
3	'2010-06-20 09:00:00'	1762.6
3	'2010-06-20 09:00:30'	-8.327

La lecture du contenu d'une table à partir d'un index consiste, dans un premier temps, à positionner un curseur sur cet index, pour ensuite incrémenter ou décrémenter ce curseur afin d'accéder aux tuples avec des valeurs de clé adjacentes. Ce positionnement initial peut correspondre à la première ou la dernière clé de l'index grâce aux mots clés `FIRST` et `LAST`, ou bien à une valeur de clé donnée.

```
HANDLER Mesures READ `PRIMARY` LAST;

HANDLER Mesures READ `PRIMARY` = (3,'2010-06-20 09:00:00');
```

Après avoir été positionné, le curseur peut être déplacé en suivant l'ordre de l'index par l'utilisation des mots clés `NEXT` et `PREV`.

```
HANDLER Mesures READ `PRIMARY` NEXT;
```

Pour le positionnement initial du curseur, la valeur de clé peut ne correspondre qu'à un sous-ensemble des colonnes couvertes par cet index. Ce sous-ensemble doit contenir la partie "gauche" de l'index. Dans notre exemple, le positionnement avec une valeur d'`identifiant` est valide, mais une valeur de `date` ne l'est pas.

```
HANDLER Mesures READ `PRIMARY` = (3); -- valide

HANDLER Mesures READ `PRIMARY` = ('2010-06-20 09:00:00'); -- invalide
```

Si la clé ne correspond à aucune valeur de l'index, le pointeur est positionné "entre deux clés", et prendra la valeur correspondante au prochain appel de `NEXT` ou `PREV`. La première séquence d'instructions renvoie la valeur associée à la date suivante (`'2010-06-20 09:00:30'`), tandis que la seconde séquence d'instructions renvoie la valeur associée à la date précédente (`'2010-06-20 09:00:00'`).

```
HANDLER Mesures READ `PRIMARY` = (3, '2010-06-20 09:00:10');
HANDLER Mesures READ `PRIMARY` NEXT;

HANDLER Mesures READ `PRIMARY` = (3, '2010-06-20 09:00:10');
HANDLER Mesures READ `PRIMARY` PREV;
```

On peut obtenir le même résultat en remplaçant l'égalité par une inégalité pour le positionnement initial.

```
HANDLER Mesures READ `PRIMARY` >= (3, '2010-06-20 09:00:10');
```

La lecture des données d'une table sans passer par un index présente un fonctionnement similaire, mais limitée aux mots clés `FIRST` et `NEXT`. Ce mode de fonctionnement correspond à un "full table scan", qui consiste à lire l'intégralité du contenu d'une table pour évaluer une requête.

```
HANDLER Mesures READ FIRST;

HANDLER Mesures READ NEXT;
```

À l'initialisation du curseur et pendant le parcours de l'index ou de la table, il est possible d'indiquer des clauses `WHERE` pour restreindre les données récupérées. Les interactions entre les clauses `WHERE` et les moteurs de stockage sont limitées.

```
HANDLER Mesures READ `PRIMARY` >= (3, '2010-06-20 09:00:00')
  WHERE valeur < 1000;
```

Le tableau suivant récapitule les correspondances entre les commandes SQL `HANDLER` et les fonctions de l'API des moteurs de stockage.

Commande SQL	Fonction de l'API
`HANDLER [table] READ [index] >= [value]`	`index_read`
`HANDLER [table] READ [index] <= [value]`	`index_read_last`
`HANDLER [table] READ [index] FIRST`	`index_first`
`HANDLER [table] READ [index] LAST`	`index_last`
`HANDLER [table] READ [index] NEXT`	`index_next`
`HANDLER [table] READ [index] PREV`	`index_prev`
`HANDLER [table] READ FIRST`	`rnd_init`
`HANDLER [table] READ NEXT`	`rnd_next`

A.3 Calcul de complexité pour les insertions dans Chronos

Cette annexe détaille les calculs pour la propriété 4.1.

Rappel des notations :
- N : nombre de paires clé-valeur initialement contenues dans l'arbre,
- F : taille moyenne des nœuds,
- L : taille moyenne des feuilles.

Propriété A.1 (4.1)**.** Après k insertions, $\frac{k}{L(F-1)}$ nœuds et $\frac{k}{L}$ feuilles sont ajoutés à l'arbre.

Démonstration. Initialement, l'arbre possède une hauteur h, A feuilles, B nœuds tels que :

$$h = \log_F\left(\frac{N}{L}\right) \qquad A = \frac{N}{L} \qquad B = \sum_{i=1}^{h} \frac{N}{LF^i}$$

Après k insertions, le nombre de paires contenues dans l'arbres est $N' = N + k$, qui possède alors une hauteur h', A' feuilles et B' nœuds tels que :

$$h' = \log_F\left(\frac{N'}{L}\right) \qquad A' = \frac{N'}{L} \qquad B' = \sum_{i=1}^{h'} \frac{N'}{LF^i}$$

$$h' - h = \log_F\left(\frac{N'}{L}\right) - \log_F\left(\frac{N}{L}\right) = \log_F\left(\frac{N'}{N}\right)$$

$$A' - A = \frac{N'}{L} - \frac{N}{L} = \frac{k}{L}$$

$$\begin{aligned} B' - B &= \sum_{i=1}^{h'} \frac{N'}{LF^i} - \sum_{i=1}^{h} \frac{N}{LF^i} \\ &= \sum_{i=h+1}^{h'} \frac{N'}{LF^i} + \sum_{i=1}^{h} \frac{N'}{LF^i} - \sum_{i=1}^{h} \frac{N}{LF^i} \\ &= \sum_{i=h+1}^{h'} \frac{N'}{LF^i} + \frac{k}{L} \sum_{i=1}^{h} \frac{1}{F^i} \end{aligned}$$

$$\text{or, } \sum_{i=h+1}^{h'} \frac{N'}{LF^i} = \frac{N'}{LF^{h+1}} \left(\frac{1 - \frac{1}{F^{h'-h}}}{1 - \frac{1}{F}} \right)$$

$$= \frac{N'}{LF^{h+1}} \left(\frac{1 - \frac{1}{F^{\log_F\left(\frac{N'}{N}\right)}}}{1 - \frac{1}{F}} \right)$$

$$= \frac{N'}{LF^{h+1}} \left(\frac{1 - \frac{N}{N'}}{1 - \frac{1}{F}} \right)$$

$$= \frac{N'}{LF^{h+1}} \left(\frac{\frac{k}{N'}}{1 - \frac{1}{F}} \right)$$

$$= \frac{k}{LF^{h+1}} \left(\frac{1}{1 - \frac{1}{F}} \right)$$

$$\text{donc, } B' - B = \frac{k}{LF^{h+1}} \left(\frac{1}{1 - \frac{1}{F}} \right) + \frac{k}{LF} \left(\frac{1 - \frac{1}{F^h}}{1 - \frac{1}{F}} \right)$$

$$= \frac{k}{LF} \left(\frac{\frac{1}{F^h}}{1 - \frac{1}{F}} + \frac{1 - \frac{1}{F^h}}{1 - \frac{1}{F}} \right)$$

$$= \frac{k}{LF} \left(\frac{1}{1 - \frac{1}{F}} \right)$$

$$= \frac{k}{L(F-1)}$$

En résumé,

$$h' - h = \log_F \left(\frac{N'}{N} \right) \qquad A' - A = \frac{k}{L} \qquad B' - B = \frac{k}{L(F-1)}$$

Après k insertions, l'arbre contient donc bien $\frac{k}{L(F-1)}$ nœuds et $\frac{k}{L}$ feuilles supplémentaires.

□