REYKJAVÍK UNIVERSITY
HÁSKÓLINN Í REYKJAVÍK

Dynamic Behavior of Balanced NV-trees

Arnar Ólafsson

Master of Science
June 2008

Reykjavík University - School of Computer Science

M.Sc. Thesis

REYKJAVÍK UNIVERSITY
HÁSKÓLINN Í REYKJAVÍK

Dynamic Behavior of Balanced NV-trees

by

Arnar Ólafsson

Thesis submitted to the School of Computer Science
at Reykjavík University in partial fulfillment
of the requirements for the degree of
Master of Science

June 2008

Thesis Committee:

Björn Þór Jónsson, supervisor
Associate Professor, Reykjavík University

Laurent Amsaleg
Research Scientist, IRISA-CNRS

Magnús Már Halldórsson
Professor, Reykjavík University

Copyright
Arnar Ólafsson
June 2008

The undersigned hereby certify that they recommend to the School of Computer Science at Reykjavík University for acceptance this thesis entitled **Dynamic Behavior of Balanced NV-trees** submitted by Arnar Ólafsson in partial fulfillment of the requirements for the degree of **Master of Science**.

____

Date

____

Björn Þór Jónsson, supervisor
Associate Professor, Reykjavík University

____

Laurent Amsaleg
Research Scientist, IRISA-CNRS

____

Magnús Már Halldórsson
Professor, Reykjavík University

The undersigned hereby grants permission to the Reykjavík University Library to reproduce single copies of this thesis entitled **Dynamic Behavior of Balanced NV-trees** and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as herein before provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatsoever without the author's prior written permission.

____
Date

____
Arnar Ólafsson
Master of Science

Dynamic Behavior of Balanced NV-trees

by

Arnar Ólafsson

June 2008

Abstract

In recent years, some approximate high-dimensional indexing techniques have shown promising results by trading off quality guarantees for improved query performance. While the query performance and quality of these methods has been well studied, however, the performance of index maintenance has not yet been reported in any detail. In this thesis we focus on the dynamic behavior of the NV-tree, which is a disk-based approximate index for very large collections. The NV-tree has several configuration and implementation options that affect the performance of index maintenance. We report on an initial study of the effects of these options on the dynamic behavior of the balanced NV-tree, and show that with appropriate implementation, significant performance improvements are possible. We implemented flexible configuration into the balanced NV-tree and ran detailed query search experiments on live data. We show that our configurations not only reduce maintenance cost, but can also improve search performance significantly with minimal loss of search quality.

Kvik hegðun jafnvægra NV-trjáa

eftir

Arnar Ólafsson

Júní 2008

Útdráttur

Á undanförnum árum hafa komið fram nýjar gerðir margvíðra nálgunarvísa sem hafa gefið góða raun með því að fórna ábyrgð á leitargæðum fyrir aukinn leitarhraða. Margskonar rannsóknir hafa verið gerðar á slíkum vísum með það að markmiði að skoða og auka afköst og gæði þeirra, en til þessa hefur lítið verið rannsakað hvernig best er að viðhalda slíkum vísum. Í þessari ritgerð er kvik hegðun NV-trjáa skoðuð, en NV-tré eru nálgunarvísar fyrir mjög stór gagnasöfn. NV-tré bjóða upp á marga stilli- og útfærslumöguleika, sem hafa mismunandi áhrif á hversu skilvirkt viðhald þeirra er. Við birtum niðurstöður rannsókna á áhrifum þessara stillinga á kvika hegðun NV-trjáa og sýnum að með vandaðri útfærslu má ná fram umtalsverðri afkastaaukningu. Við útfærðum einnig nýjar sveigjanlegar stillingar í NV-tréð og keyrðum ítarlegar myndaleitir á raungögn. Þessar stillingar bæta ekki einungis skilvirkni viðhaldsaðgerða, heldur auka einnig leitarafköst verulega með aðeins smávægilegum áhrifum á leitargæði.

To my father Ólafur, may he rest in peace,
and to my baby boy Ólafur Regin, the pearl in our life.

Acknowledgements

I wish to thank Herwig Lejsek and Friðrik H. Ásmundsson for the discussions of the properties of NV-trees as well as access to their implementation. I would also like to thank Björn Þór Jónsson and Laurent Amsaleg for their contribution to this work as co-authors of the related paper. And special thanks to my better half Solja Klargaard for enduring my frequent stays at school and a big hug to my mother for all her help.

Publications

Part of the material in this thesis was published in the paper *Dynamic Behavior of Balanced NV-trees* which was accepted to the Sixth International Workshop on Content-Based Multimedia Indexing (CBMI, 18-20th June, 2008, London, UK). The paper introduces flexible configurations, describes the simulation experiments and analyzes the results gained from our simulations. Most of Chapters 1 to 4 and 6.1 to 6.4 are part of the paper. Co-authors are Björn Þór Jónsson (Reykjavík University) and Laurent Amsaleg (IRISA-CNRS). While Björn and Laurent contributed significantly to the writing of the paper, the model implementation and experimentation is entirely my work.

Contents

1 Introduction **1**

1.1 Requirements of Content-Based Indexing 1

1.2 Current Index Strategies 2

1.3 Contribution of this Thesis 3

1.4 Overview of this Thesis 4

2 The NV-tree **5**

2.1 NV-tree Creation 5

2.2 NV-tree Nearest Neighbor Retrieval Process 6

2.3 Projection Strategies 7

2.4 Partitioning Strategies 7

2.5 Overlap Strategies 8

2.6 NV-tree Nodes 8

2.7 Insertions and Deletions 9

2.8 Summary 9

3 Implementing Redundancy **10**

3.1 Defining Partial Overlap 10

3.2 Non-Overlapping Configuration 12

3.3 Overlapping Configuration 13

3.4 Summary 14

4 Modeling Inserts **15**

4.1 Simulation Model Basics 15

4.2 Cost of Insert 17

4.2.1 Direct Insertions 17

4.2.2 Buffered Insertions 18

4.3 Cost of Index Maintenance 18

4.3.1 No Splits 19

4.3.2 Leaf Splits 20
4.3.3 Parent Splits 20
4.3.4 Hybrid Splits 21
4.3.5 Re-Generation 21
4.4 Partition Files 22
4.5 Summary 23

5 Implementing the Simulation Model 24
5.1 Overview 24
5.2 Implementing Insertions 25
5.2.1 Direct Insertions 26
5.2.2 Buffered Insertions 27
5.3 Implementing Index Maintenance 28
5.3.1 No Splits 28
5.3.2 Leaf Splits 28
5.3.3 Parent Splits 29
5.3.4 Hybrid Splits 31
5.3.5 Re-Generation 32
5.4 Implementing Partition Files 32
5.5 More Efficient Approach 33
5.6 Summary 35

6 Simulation Results 36
6.1 Simulation Environment 36
6.2 Experiment 1: Direct vs. Buffered Inserts 37
6.3 Experiment 2: Split Policies 39
6.4 Experiment 3: Partition Files 42
6.5 Experiment 4: Buffer Sizes 44
6.6 Summary 45

7 Effect of τ on Search Quality and Performance 47
7.1 Experimental Environment 47
7.2 Experiment 1: Search Quality 50
7.3 Experiment 2: Search Performance 54
7.4 Summary 58

8 Conclusion 59

Bibliography 61

List of Figures

3.1 Partial overlap configurations 11

4.1 Effect of split policies when the central leaf node overflows: (a) original partitions before split; (b) with No split; (c) with Leaf split; and (d) with Parent split. 19

6.1 Insertion cost for Re-Generation with and without the insertion buffer (varying τ; no partition files; 512MB buffer). 38

6.2 NV-tree size for Re-Generation with and without the insertion buffer (varying τ; no partition files; 512MB buffer). 38

6.3 Insertion and split cost of the different split policies ($\tau = 0.0$; buffered inserts; no partition files). 40

6.4 Insertion and split cost of the different split policies ($\tau = 0.25$; buffered inserts; no partition files). 40

6.5 Insertion and split cost of the different split policies ($\tau = 0.5$; buffered inserts; no partition files). 41

6.6 NV-tree size for Leaf, Parent and Hybrid splits without partition files (varying τ; buffered inserts). 41

6.7 Insertion and split cost of different split policies with and without partition files ($\tau = 0.0$; buffered inserts). 42

6.8 Insertion and split cost of different split policies with and without partition files ($\tau = 0.25$; buffered inserts). 43

6.9 Insertion and split cost of different split policies with and without partition files ($\tau = 0.5$; buffered inserts). 43

6.10 NV-tree size for Leaf, Parent and Hybrid splits with partition files (varying τ; buffered inserts). 44

6.11 Insertion and split cost for Hybrid split (varying τ; with and without partition files; varying buffer). 45

7.1 Distribution of CPU ticks 49

7.2 Average top score result for 3,120 images (varying τ; 1–3 indexes). 50
7.3 Image modification ranking 51
7.4 Average descriptor ratio for 3,120 images (varying τ; 1–3 indexes). 52
7.5 Number of descriptors needed for 3,120 images using stop rules (varying τ; 1–3 indexes). 53
7.6 Total NV-tree size (varying τ; 1–3 indexes). 55
7.7 Total reads needed (varying τ; 1–3 indexes). 55
7.8 Percentage read from disk vs. from memory (varying τ; 1–3 indexes). . . 56
7.9 Query search time for 3,120 images (varying τ; 3 indexes). 57

List of Tables

3.1 Partial lookup table for actual overlap. 12

4.1 Simulation model parameters. 17

4.2 Index maintenance cost. 19

6.1 User defined simulation model parameters. 37

7.1 Image modifications variants. 49

Chapter 1

Introduction

Content-based multimedia indexing has been an active area of research and development for the last two decades. Typically, multimedia content, such as sound, images or video, is mapped into one or many high-dimensional vectors of numbers, which are then stored in a high-dimensional index. Multimedia queries are likewise mapped into vectors, which are typically used to conduct nearest neighbor queries to the high-dimensional index. These queries return the most similar vectors, which are mapped back into multimedia data.

1.1 Requirements of Content-Based Indexing

Since the number and size of multimedia collections have been growing exponentially in recent years, the requirements for high-dimensional indexing have been changing very dramatically in at least three important ways.

First, multimedia data is of higher quality and complexity, requiring much more intricate description methods than before. While color histograms were considered potentially useful in early systems, recent state-of-the-art systems have adopted local descriptors such as the SIFT descriptors, which describe the content in great detail (Lowe, 2004).

Second, multimedia collections have grown in size by orders of magnitude and are still growing. Flickr, for example, currently holds more than two billion images in its collection. It has been shown quite conclusively that in such large-scale environments, exact methods cannot cope with the application requirements (Amsaleg & Gros, 2001). Approximate methods are therefore required to cope with the ever-increasing quantity of data.

Third, unlike assumptions made by early systems, multimedia collections are now subject to extremely high update activities. YouTube, for example, currently receives about 100,000 new videos per day. Search engine users already expect useful answers on current events and they will expect no less of tomorrow's multimedia applications. Furthermore, in some cases, such as for copyright protection applications, new material is actually more valuable than older material.

As multimedia collections are updated with new content, the index strategies used on the collections must handle inserts and deletions dynamically. With growing collections, the search strategies must be able to retrieve results both efficiently and effectively.

1.2 Current Index Strategies

Index strategies for low dimensional collections have been around for many years. One-dimensional data is frequently used in our daily life using relational databases. These databases use indexing strategies that have provided highly effective and efficient search strategies. The indexes are usually based on *hashing* and *trees*.

As the dimensionality of data increases, we need to consider other indexing strategies. Techniques for medium range dimensionality include VA-Files (Weber, Schek, & Blott, 1998), IQ-trees (Böhm, Berchtold, & Keim, 2001), R-trees (Guttman, 1984) and the Pyramid-Technique (Berchtold, Böhm, & Kriegel, 1998), along with many other techniques. Most of these index techniques use exact k-NN search strategy that has shown good results for dimensionality lower than $10 - 16$. For higher dimensional data it has been shown that these search strategies perform worse than a sequential scan of the whole collection (Amsaleg & Gros, 2001).

Recently, some approximate high-dimensional indexing techniques have shown promising results by trading off result quality guarantees for improved query performance. Perhaps the most popular technique is Locality Sensitive Hashing (LSH) (Datar, Indyk, Immorlica, & Mirrokni, 2006), which has been used for some applications (e.g., see (Casey & Slaney, 2006)). LSH has been shown to be an effective search method but a single hashtable bucket can become very large in LSH and this can lead to unpredictable IO costs (Lejsek, Ásmundsson, Jónsson, & Amsaleg, 2008).

More recently, however, the NV-tree, which is a disk-based approximate index for very large collections (Lejsek et al., 2008), has been shown to outperform LSH for large-scale retrieval (Lejsek et al., 2008). While the query performance of NV-tree has been well

studied, however, the performance trade-offs of the index maintenance and redundancy has not yet been reported in any detail.

1.3 Contribution of this Thesis

In this thesis we focus on the dynamic behavior of the NV-tree. The NV-tree has several configuration and implementation options that affect the performance of index maintenance.[1] We report on an initial study of the effects of these options on the dynamic behavior of the balanced NV-tree. We chose to use a balanced NV-tree as it is easier to simulate. To advance the NV-tree in high end applications, we work on four new aspects of the NV-tree.

- First, we introduce a flexible overlap configuration into the balanced NV-tree. The flexibility controls the redundancy of the NV-tree by adjusting the index to different overlap configurations. With less redundancy the index becomes smaller and is more likely to fit in memory, thus improving the performance of index creation, inserts and search.
- Second, as values are inserted, the redundant spaces in the leaf nodes are filled up and the tree requires maintenance. We introduce five different maintenance policies and study the effects of each policy on the index by simulating insertion of 250,000 images into the NV-tree.
- Third, we simulate a *buffered insert* strategy and compare the benefits to *direct inserts*. As disk reads and writes are the most expensive operations, we analyze the effect of using the buffer to assist with insertions to increase performance.
- Fourth, we analyze the effect of using *partition files* with the NV-tree. The partition files are sorted subsets of the object collection. When maintenance tasks are performed the whole collections needs to be sequentially scanned for each leaf partition used in the index. By reducing the sequential scans to subsets of the collection, performance is improved.

We have created a simulation model, using the Python language, to simulate the above functionality in the NV-tree. Our simulation model simulates 250,000 image insertions into a pre-existing collection of approximately 30,000 images. We simulate different variations of the flexible configurations, maintenance policies, buffering and partitions files.

[1] Note that concurrency control is outside the scope of this thesis.

The flexible configuration is a major factor in our dynamic behavior and has large impact on the index. In a second performance study, we therefore studied how the flexible configurations affect search quality, by running a large set of query images on live data using a balanced NV-tree. In this study, we focused on two key factors: a) the search quality and b) the search performance. We use 3,120 modified images from original images in the collection, based on 26 different modifications.

Our results show that our configurations not only reduce maintenance cost, but can also improve search performance significantly with minimal loss of search quality.

1.4 Overview of this Thesis

First, we describe the NV-tree and its main features in Section 2. We then demonstrate how redundancy can be implemented in a flexible manner in balanced NV-trees in Section 3. In Section 4 we define a simulation model to study the effects of various configuration and implementation options. In Section 5 we explain the implementation of our simulation model. In Section 6 we perform a detailed performance study, analyzing the index maintenance performance, and show that with appropriate implementation, significant performance improvements are observed. In Section 7 we perform detailed performance and quality measurements on the flexible NV-tree using 26 image modifications created from 120 query images. We show that search performance can be improved considerably with a small trade-off in search quality. We then conclude our findings in Section 8.

Chapter 2

The NV-tree

The NV-tree is a disk-based data structure designed to provide efficient approximate k-nearest neighbor search in very large high-dimensional collections. In essence, it transforms costly nearest neighbor searches in high-dimensional space into efficient uni-dimensional accesses using a combination of projections of data points to lines and (redundant) partitioning of the projected space.

This chapter describes several aspects of the NV-tree, focusing particularly on the aspects that are important for understanding the remainder of this thesis. We therefore describe first the two main operations of index creation and search. We then briefly outline different strategies for projecting the descriptors, partitioning the data collection, and introducing redundancy. Section 3 is entirely dedicated to investigating new strategies for redundancy. The internal data structures of the NV-tree are then described before presenting the index maintenance operations. A more detailed description of the NV-tree and its operations can be found in (Lejsek et al., 2008).

2.1 NV-tree Creation

Overall, an NV-tree is a tree index consisting of: a) a hierarchy of small *inner nodes*, which are kept in memory during query processing and guide the descriptor search to the appropriate leaf node; and b) larger *leaf nodes*, which are stored on disk and contain references to actual descriptors.

When the construction of an NV-tree starts, all descriptors are considered to be part of a single temporary partition. Descriptors belonging to the partition are first projected onto

a single *projection line* through the high-dimensional space. Strategies for selecting the projection lines are discussed in Section 2.3.

Next, the projected values are partitioned into disjunct sub-partitions based on their position on the projection line. Information about all these sub-partitions, such as the partition borders on the projection line, form the inner node of the first level of the NV-tree. Strategies for partitioning are described in Section 2.4.

Since descriptors which are close to partition borders are likely to be similar to descriptors in the adjacent partition, the NV-tree partitions are allowed to overlap for redundant coverage. An overlap parameter is used to control the amount of redundancy between partitions. In the extreme case, for each pair of adjacent partitions, an overlapping sub-partition is created which covers 50% of both partitions. Strategies for overlap are described in Section 2.5.

To build subsequent levels of the NV-tree, this process of projecting and partitioning is repeated for all the new sub-partitions using a new projection line at each level, creating a hierarchy of inner nodes. The process stops when the number of descriptors in a sub-partition falls below a specified limit designed to be no more than single I/O. A new projection line is then used to order the descriptor identifiers of the sub-partition, and the ordered identifiers are written to a leaf node on disk.

2.2 NV-tree Nearest Neighbor Retrieval Process

During query processing, the query descriptor first traverses the hierarchy of inner nodes of the NV-tree. At each level of the tree, the query descriptor is projected to the projection line associated with the current node. The search is then directed to the sub-partition with the center-point closest to the projection of the query descriptor. This process of projection and choosing the right sub-partition is repeated until the search reaches a leaf node.

The leaf node is fetched into memory and the query descriptor is projected onto the projection line of the leaf node. The search then starts at the position of the query descriptor projection. The two descriptor identifiers on either side of the projected query descriptor are returned as the nearest neighbors, then the second two descriptor identifiers, etc. Thus, the $k/2$ descriptor identifiers found on either side of the query descriptor projection are alternated to form the ranked k approximate neighbors of the query descriptor.

Note that since leaf partitions have a fixed size, the NV-tree guarantees query processing time of a single disk read regardless of the size of the descriptor collection. Larger collections need deeper NV-trees but the intermediate nodes fit easily in memory and tree traversal cost is negligible.

2.3 Projection Strategies

In the NV-tree, projection lines are used at each level of the tree, and hence a strategy is needed for selecting those lines. There are two alternative strategies. First, we can use random line from a pool of random lines, that are created by generating isotropic random lines requiring a minimal angle between pairs of lines. Using random lines is independent of data, but may lead to sub-optimal partitioning. Second, we can select the "best" line using Principal Component Analysis (PCA), which is very costly. In (Lejsek et al., 2008), however, an *approximate PCA* strategy is proposed, which selects the best line from a large line pool. This strategy proved to yield search results of better quality than random lines; the simulation model for inserts therefore assumes well chosen lines from a line pool.

2.4 Partitioning Strategies

A partitioning strategy is likewise needed at every level of the NV-tree. Three strategies were proposed: *Balanced*, *Unbalanced* and *Hybrid*.

The *Balanced* strategy partitions data based on cardinality. Therefore, each sub-partition gets the same number of descriptors, and eventually all leaf partitions are of the same size. Although node fanout may vary from one level to the other, the NV-tree becomes balanced as each leaf node is at the same height in the tree.

The *Unbalanced* partitioning strategy adjusts to the data distribution, by using distances instead of cardinalities. In this case, sub-partitions are created such that the absolute distance between their boundaries is equal. All the data points in each interval belong to the associated sub-partition. With this strategy, however, the normal distribution of the projections leads to a significant variation in the cardinalities of sub-partitions. Due to the repeated application of the partitioning strategy, the NV-tree becomes unbalanced as dense areas are partitioned more often than sparse areas.

The *Hybrid* strategy first follows the *Unbalanced* strategy until a sub-partition is of a size that could fit into around six leaf partitions. Then the *Balanced* strategy is used to construct the leaf partitions. As a result, leaf partitions are better utilized and the tree is shallower, both of which result in smaller space requirements.

We have chosen to focus on the *Balanced* strategy in our work since it is much easier to implement and model.

2.5 Overlap Strategies

An overlap strategy is needed at each level of the NV-tree. One option is the *No Overlap* strategy, where each descriptor is only inserted into a single sub-partition, as described above. A second option is the *Full Overlap* strategy, where each descriptor is inserted into two partitions, except for the descriptors at both ends of the projection line. We can also chose intermediate values to control the overlap. Much of the remainder of this thesis explores the implementation and effect of partially overlapping partitions.

2.6 NV-tree Nodes

The intermediate nodes of the NV-tree are used for two purposes: to guide the search for a descriptor to the single appropriate leaf node, and to guide the insertion of a descriptor to all appropriate leaf nodes, as described below. Typically, in non-redundant tree structures, such as the traditional B^+-tree, this can be achieved by storing an array of partitioning values in each intermediate node. Due to the potential redundancy of the NV-tree, however, these two purposes of the intermediate nodes must be achieved using different arrays of values. The *partition borders* array stores values guiding the insertion process to all appropriate leaf nodes. The *search borders* array keeps track of the values which are mid-way between the partition borders and are used to guide the search process. Intermediate nodes are small and are kept in memory during query processing.

Each leaf node is the size of a disk block and stores (*projected value*, *descriptor identifier*) pairs. For efficiently finding the pair of the leaf, which has its projected value closest to the projection of the query descriptor, leaves are organized by the projected values in a sorted look-up table.

2.7 Insertions and Deletions

Consider first insertions to the NV-tree, with no overlapping partitions. In this case, the insertion process must first descend the tree in the same manner as the search, to find the appropriate leaf to insert the descriptor into. Then the final projection line is used to find the appropriate location for the descriptor identifier within that leaf.

With overlapping partitions, on the other hand, each descriptor must potentially be inserted into many leaf nodes, due to the redundancy arising from the overlapping partitions. Unlike the search, the insertion process must then, at each level, descend into either one or two sub-partitions which contain the projection of the descriptor. In the worst case of full overlap, a descriptor may thus need to be inserted into 2^h leaf nodes, where h is the number of levels in the tree.

Note that until the descriptor has been inserted into all the appropriate partitions, it cannot be considered fully inserted and may not show up in certain query results. Needless to say, insertion can be an expensive operation that: 1) can affect the choices of index creation strategies, and 2) must be implemented carefully. In Sections 4 through 6, we analyze cost models for several implementation choices.

Deletion is implemented in a similar manner as insertion. It is possible, however, to keep a table of recently deleted descriptors and filter them out of the result. In this way, deleted descriptors can be removed from query results, although they may still be found in one or more partitions on disk. For this reason, we do not address deletion further in this thesis.

2.8 Summary

Overall, an NV-tree consists of a hierarchy of small inner nodes, which fits in memory, and larger leaf nodes, which are stored on disk and contain descriptor identifiers. In this section, we have described the processes for index creation, index search, and index maintenance, as well as alternative strategies for the index creation and search. As we have pointed out, insertion can be an expensive operation that: 1) can affect the choices of index creation strategies, and 2) must be implemented carefully. In this thesis, we examine the effects of various strategies and implementation options on the insertion performance of balanced NV-trees.

Chapter 3

Implementing Redundancy

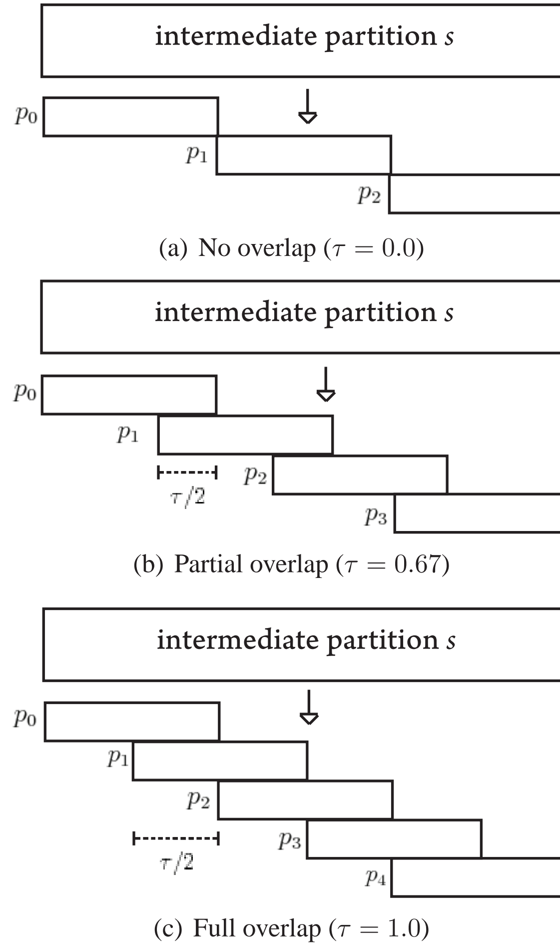
In this chapter we propose a methodology for implementing redundancy within the balanced NV-tree in a flexible manner, ranging from no overlap to full overlap. We assume that the administrator of the NV-tree index supplies five configuration values: tree height h; collection size d (in tuples); partition size p (in tuples); desired leaf node utilization u; and the desired overlap factor $\tau \in [0, 1]$. This section describes how these five values are used to compute an *index level configuration* $L^\tau = \left[l_0^\tau, \ldots, l_{h-1}^\tau\right]$, which describes the number of sub-partitions l_i^τ at each level i of the balanced NV-tree.

3.1 Defining Partial Overlap

During the NV-tree construction, a key issue is how to partition intermediate sub-partitions. At each stage of the construction, the overlap factor τ is used to decide what fraction of each partition should overlap with other partitions. For $\tau = 0.5$, for example, half of the descriptor identifiers of the partition must also be present in other partitions. As we shall see in the following discussion, it may not be possible to partition in such a way that the overlap is exactly τ (as then the last partition would be smaller than the others). Therefore, we now describe a general partitioning method, where the overlap between partitions is at least τ.

Without loss of generality, we assume in the following discussion that we are splitting an intermediate sub-partition s of size lp into leaf partitions,[1] and that the desired utilization is $u = 100\%$. The goal is then to split s into a number of partitions, such that each

[1] The size of intermediate partitions is always an integer multiple of pu.



(a) No overlap ($\tau = 0.0$)

(b) Partial overlap ($\tau = 0.67$)

(c) Full overlap ($\tau = 1.0$)

Figure 3.1: Partial overlap configurations

partition gets exactly p descriptors and the actual overlap between leaf partitions is at least τ.

EXAMPLE 1.
Figure 3.1 shows three possible sub-partition configurations for an intermediate sub-partition s of size $3p$. These configurations have: a) three leaf partitions and no overlap ($\tau = 0$); b) four leaf partitions and partial overlap ($\tau = 2/3$); and c) five leaf partitions and full overlap ($\tau = 1$). As the figure shows, since each partition (aside from the end partitions) is adjacent to two partitions, the overlap between any two adjacent partitions is $\tau/2$.

Since the size of each partition must be exactly p, only a few values of τ yield distinct partial overlaps in each case. These values depend on the size of the intermediate sub-partition, lp, and the number of leaf partitions, k, and are therefore denoted by $\tau_{l,k}$. In Figure 3.1, those values are $\tau_{3,3} = 0$, $\tau_{3,4} = 2/3$ and $\tau_{3,5} = 1$. Having $\tau = 1/2$, for

$\tau_{l,k}$		k												
		6	7	8	9	10	11	12	13	14	15	16	17	18
l	6	0.00	0.33	0.57	0.75	0.89	1.00	-	-	-	-	-	-	-
	7	-	0.00	0.29	0.50	0.67	0.73	0.83	0.92	1.00	-	-	-	-
	8	-	-	0.00	0.25	0.44	0.60	0.73	0.83	0.92	1.00	-	-	-
	9	-	-	-	0.00	0.22	0.40	0.55	0.67	0.77	0.86	0.93	1.00	-
	10	-	-	-	-	0.00	0.20	0.36	0.50	0.62	0.71	0.80	0.88	0.94
	11	-	-	-	-	-	0.00	0.18	0.33	0.46	0.57	0.67	0.75	0.82
	12	-	-	-	-	-	-	0.00	0.17	0.31	0.43	0.53	0.63	0.71

Table 3.1: Partial lookup table for actual overlap.

example, should yield the same partitioning as with $\tau = 2/3$. When partitioning a sub-partition s of size lp, the goal is therefore to find the number of partitions k which gives the smallest overlap factor $\tau_{l,k} \geq \tau$.

The minimum value for k is when $\tau = 0$. In this case, only l sub-partitions can be defined. Conversely, the maximum value for k is when $\tau = 1$; in this case $2l - 1$ sub-partitions can be created. Generally, for any l and k, the amount of overlap between the k leaf partitions is $(k-1)\frac{\tau_{l,k}}{2}$, yielding the following equation:

$$k - (k-1)\frac{\tau_{l,k}}{2} = l \tag{3.1}$$

Solving for $\tau_{l,k}$ results in:

$$\tau_{l,k} = \frac{2(k-l)}{k-1} \tag{3.2}$$

EXAMPLE 2.
Solving Equation 3.2 for all values of l and k yields a look-up table of $\tau_{l,k}$ values as partially shown in Table 3.1. If $l = 8$ then k ranges from 8 to 15. When $\tau = 0.5$ is desired, for example, then k must be 11, leading to an actual overlap of $\tau_{8,11} = 0.6$.

Given p and $\tau_{l,k}$, the partition borders can be easily determined. The lower partition border of sub-partition p_i can be found at rank $p(i - i\tau_{l,k}/2)$ in the intermediate node about to be partitioned, while the upper partition border can be found at rank $p(i+1-i\tau_{l,k}/2)$. Search borders are determined, as before, by finding the mid-point between partition borders of adjacent sub-partitions.

3.2 Non-Overlapping Configuration

The computation of the index configuration proceeds in two steps. First, a non-overlapping configuration L^0 is found, corresponding to $\tau = 0$. Then the index configuration L^τ is de-

termined by applying the procedure described above at each level of the non-overlapping configuration.

In the absence of overlap, the expected number of leaf partitions is simply d/pu. In order to get an initial configuration $L^0 = \left[l_0^0, \ldots, l_{h-1}^0\right]$ with at least d/pu partitions, we define:

$$l^0 = \left\lceil \sqrt[h]{d/pu} \right\rceil \tag{3.3}$$

and initialize $l_i^0 = l^0$. The total number of leaf partitions given by this configuration is denoted by:

$$|L^0| = \prod_i l_i^0 \tag{3.4}$$

As this configuration may give more than d/pu leaf partitions, we must seek a configuration which better approximates the desired d/pu partitions. In essence, we seek the configuration $L^0 = [l^0, \ldots, l^0, l^0 - 1, \ldots, l^0 - 1]$ which gives the smallest number of leaf partitions $|L^0| \geq d/pu$.

EXAMPLE 3.
Given $h = 4$*,* $d = 35,484,770$*,* $p = 16,384$ *and* $u = 67\%$*, the expected number of partitions is* $d/pu = 3,233$*. The initial partitioning estimate is* $l^0 = \left\lceil \sqrt[4]{3,233} \right\rceil = 8$*. Since* $8 \times 8 \times 8 \times 8 = 4,096$*, however, a better configuration can be found. Next,* $8 \times 8 \times 8 \times 7 = 3,584$*, while* $8 \times 8 \times 7 \times 7 = 3,136$*, which is smaller than* $3,233$*. Therefore, the configuration* $L^0 = [8,8,8,7]$ *is chosen.*

3.3 Overlapping Configuration

Next, the overlap factor τ is used to determine the initial index configuration L^τ based on the non-overlapping configuration L^0. At each level i, we use l_i^0 to solve Equation 3.1 of Section 3.1, yielding the number of overlapping sub-partitions, l_i^τ, at that level. This process is efficiently implemented using the look-up table shown in Table 3.1. The vector $L^\tau = \left[l_0^\tau, \ldots, l_{h-1}^\tau\right]$ then describes the balanced NV-tree index configuration which has the smallest overlap greater than τ.

Note that the number of partitions, and thus index utilization, may differ from the intended configuration parameters. This is due to the approximation of the overlap factor. As before, the number of leaf partitions is denoted by:

$$|L^\tau| = \prod_i l_i^\tau \tag{3.5}$$

EXAMPLE 4.
Assume the non-overlapping configuration $[8, 8, 8, 7]$ *of Example 3 and a desired overlap factor of* $\tau = 1/2$*. The resulting index configuration is* $L^\tau = [11, 11, 11, 9]$*, which has* $|L^\tau| = 11 \times 11 \times 11 \times 9 = 11,979$ *leaf partitions.*

3.4 Summary

In this section we have proposed a general method for partial overlap. We have defined equations to configure the correct number of leaf partitions based on the collection size and user defined values such as the desired overlap factor. We have also described how the desired overlap factor is only a proposed lower limit to the overlap needed. In Section 4, we will describe the cost models for insertions and maintenance needed based on the partial overlapping method proposed in this section.

Chapter 4

Modeling Inserts

In this chapter, we describe the simulation model which we have implemented in order to study the expected performance of various insertion strategies for NV-trees. In this section, we focus on the cost model, while the implementation is described in Chapter 5. We start by giving the simulation model basics in Section 4.1. Then we detail the cost formulas behind individual implementation options in Sections 4.2 through 4.4.

4.1 Simulation Model Basics

In Section 2, three major strategic choices were discussed for the NV-tree. In this work, we have focused on the balanced partitioning strategy, as it is more amenable to modeling than the unbalanced and hybrid strategies. The projection strategy is not modeled as such, but the model assumes well chosen lines from a large set of random lines. Finally, the overlap strategy is fundamental to the model and follows the implementation described in Section 3.

Table 4.1 shows the input parameters of the model, as well as the instantiation used for examples and in the experimental section. First, an initial balanced NV-tree configuration L^{τ} is created, based on the first half of Table 4.1, and used to initialize the appropriate data structures. Then, as multiple descriptor insertions are simulated, the simulation model accumulates insertion costs, based on the second half of Table 4.1 as described in detail below, and updates the data structures appropriately. Insertions are paused every 10,000 images and the various performance metrics written to disk.

It has been shown that the NV-tree copes well with the requirements of image and video copyright protection applications using powerful local descriptors (Lejsek et al., 2008).

The most popular such descriptors are the SIFT descriptors (Lowe, 2004), which we simulate in our model. SIFT descriptors are 128-dimensional and can be stored in 132 bytes (1 byte per dimension plus 4 bytes to store the descriptor identifiers), so $D = 132d$. Note that the number of descriptors d in the initial collection corresponds to the set of SIFT descriptors for about 30 thousand photo-agency images.

The main structure of the simulation model is an array of partitions, $\mathcal{P} = \left[p_0, \ldots, p_{|L^\tau|}\right]$. Each partition p_j holds local information about: the number of descriptors in the partition (p_j.*count*); the probability of insertions to the partition (p_j.*prob*); the parent node of the leaf; and other book-keeping elements. When the index is created, the partition size is uniform as the index creation is rank based, making $p_j.\mathit{count} = \lceil d/|L^0| \rceil$. During insertion, each descriptor may be inserted into more than one leaf partition for $\tau > 0$. While in the worst case it may be inserted into 2^h partitions, the expected number of inserted partitions is:

$$V = |L^\tau|/|L^0| \tag{4.1}$$

If the distribution of the descriptors to insert is identical to the distribution of descriptors in the original collection, which is likely, the inserted descriptors will be uniformly distributed into the partitions. The initial probability of insertion into a given partition is thus $p_j.\mathit{prob} = 1/|L^0|$. During index insertion and maintenance, the p_j.*count* and p_j.*prob* values are then maintained depending on the insertion and splitting policies.

The cost model of the simulator focuses on disk cost, as the CPU cost of traversing the index is generally only 1–3% of disk cost (Lejsek et al., 2008). We assume all disk accesses to transfer P bytes, but distinguish between random and sequential I/Os in our model; we assume that for any file read of more than $10P$, sequential reads can be achieved through a combination of pre-fetching, buffering and blocked I/O. While true sequential access is typically two orders of magnitude faster than true random access, we use a ratio of 1/10 to account for other disk traffic which may interrupt long sequential reads. The simulation model ignores effects of buffer management entirely, as we have observed that the uniform distribution of accesses reduces the effectiveness of buffer management.[1] We do, however, consider using a buffer of size B for inserted descriptor identifers and their projected values.

In the remainder of this section, we develop and argue for the cost formulas of the simulation model. We have broadly split the insertion cost into two parts. First, there is the cost of insertion to a leaf partition (Section 4.2). Second, as partitions overfill, there is an additional cost of index maintenance (Section 4.3). Note that since any method must ap-

[1] The smallest NV-trees are likely to fit in memory, leading to an overestimation of I/O cost; we note this effect when analyzing the results.

Description	Notation	Value
Initial tree height	h	4
Initial collection size	d	35,484,770 tuples
Desired leaf size	p	16,384 tuples
Desired leaf utilization	u	67%
Desired overlap factor	τ	[0, ..., 1]
Collection size on disk	D	$d \times 132\text{b} = 4{,}36$ GB
Size of a single disk I/O	P	$p \times 8\text{b} = 128$ KB
Cost of random I/Os	C_R	12.5 ms
Cost of sequential I/Os	C_S	$C_R/10$
Buffer size	B	512 MB

Table 4.1: Simulation model parameters.

pend the descriptors to the actual descriptor collection, that cost is not included. We also discuss an implementation strategy called partition files, where the descriptors themselves are stored redundantly for each partition (Section 4.4).

4.2 Cost of Insert

When a descriptor must be inserted, the $\mathcal{P}$ vector is traversed and, for each partition, the p_j*.prob* value is used in a random trial to determine whether a descriptor should be inserted into that partition. We propose two strategies for insertion: one where the descriptor is inserted directly into the appropriate partitions, and one where insertions are buffered. The buffer is assumed to be organized as a hash table on the partition identifiers. Thus, all descriptors belonging to the same partition are stored together. The search is modified to search not only the partition on disk, but also the in-memory structure.

4.2.1 Direct Insertions

Direct insertion of a descriptor into a partition involves reading in the appropriate partition, modifying it and writing back to disk, for a cost of $2C_R$.

EXAMPLE 5.
Consider the collection described in Table 4.1. With full overlap a descriptor will on average be inserted into $V = 12.24$ *partitions, and inserting a single image with 500 descriptors therefore requires* $500 \times 12.24 \times 2 \times 12.5$ *ms* $= 153$ *seconds. With no overlap* $V = 1$ *and the same insertion will take* $500 \times 2 \times 12.5$ *ms* $= 12.5$ *seconds.*

4.2.2 Buffered Insertions

In this case, there are two scenarios which lead to disk activity. First, when a specific bucket has reached a size of $2p/3$, it is flushed to disk to avoid situations where a single bucket causes multiple splits. The cost of such a flush is the same as that of a direct insert, $2C_R$. Second, if the entire buffer B is full, all partitions in memory are flushed to disk in a sequential manner, incurring a cost of $2C_S|L^\tau|$. This method is chosen since the uniform distribution of inserts makes it likely that many buckets are filled to a similar capacity.

The precise savings of buffering depend on the size of the buffer and the distribution of inserts. Note that a clever implementation would opportunistically update partitions when they are read in by a search process and flush buffers when disk activity is low. Our simulation model, however, cannot capture such details as it does not model buffer management.

4.3 Cost of Index Maintenance

When the utilization of a particular leaf node reaches 100%, the node must be split. Unlike many other tree structures, however, splitting is a complex operation in the NV-tree and there are many potential splitting policies. The key differentiator between these policies is the amount of local re-organization of the index required by the policy. Furthermore, as a result of the re-organization, some leaf partitions may acquire a new random line necessitating re-projections of a part of the descriptor collection. The cost of each policy is thus composed of the following two components:

$$C_{split} = C_{reorg} + C_{repro} \tag{4.2}$$

In the simulation model, we have considered the following five policies: *No split*; *Leaf split*; *Parent split*; *Hybrid split* (*Leaf split* or *Parent split*); and *Re-Generation*. These policies and the associated cost formulas are described in detail in the following subsections. The cost formulas are summarized in Table 4.2, while Figure 4.1 is used to illustrate the differences between the policies.

Maintenance Policy	C_{reorg}	C_{repro}
No Splits	C_R	0
Leaf Splits	$C_R(\Delta l^{\tau}_{h-1})$	C_{scan}
Parent Splits	$C_R(2l^{\tau}_{h-1} + \Delta l^{\tau}_{h-1} - 2)$	C_{scan}
Re-Generation	C_{regen}	0

Table 4.2: Index maintenance cost.

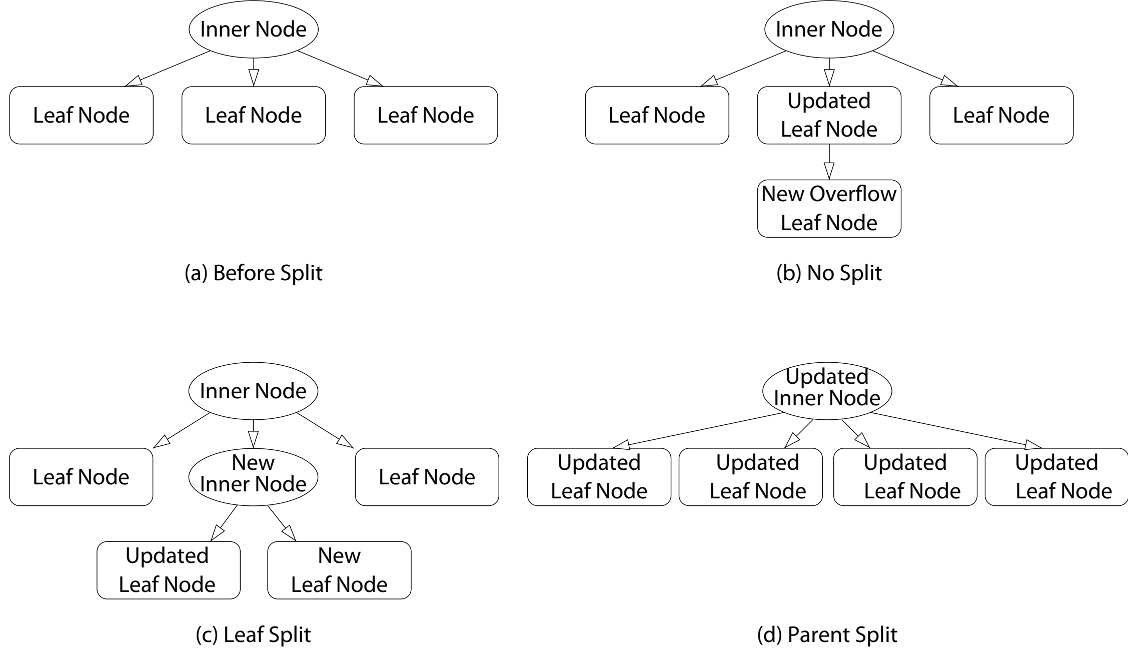


Figure 4.1: Effect of split policies when the central leaf node overflows: (a) original partitions before split; (b) with No split; (c) with Leaf split; and (d) with Parent split.

4.3.1 No Splits

The straight-forward way to deal with a partition overflow is to simply allocate a second disk block (or third, etc.) and maintain all disk blocks as a single partition (see Figure 4.1(b)). In this case, the cost of the reorganization consists of reading the original partition and writing it, as well as writing the second disk block. As the reading and writing of the original disk block are already accounted for through the insertion cost, only a single random disk write C_R must be counted. Needless to say, there is no cost of reprojecting the collection.

Two observations are in order, however. First, this policy also affects the cost of insertions to that partition; the cost formulas of Section 4.2 must be multiplied by $\lceil p_j.count/p \rceil$. Second, this policy also affects query costs as many blocks must potentially be read at retrieval time. This policy negates the fundamental property of the NV-tree that query results are always returned in a single disk read; as query costs are not accounted for in

the simulation, all results must be analyzed in view of that. But this approach serves, in a sense, as a lower bound on the cost of index maintenance.

4.3.2 Leaf Splits

The *Leaf split* policy works as follows (see Figure 4.1(c)). First, a new intermediate node is created, which replaces the leaf node in the NV-tree. Then two or more new (partially overlapping) leafs are created for the new intermediate node, thus extending the local depth of the NV-tree. Note that this policy leads to locally unbalanced trees and rapidly increasing space requirements. The search time, however, is not affected.

In the simulation model, the p_j*.prob* and p_j*.count* values of the partitions are maintained appropriately. The index maintenance cost is then calculated as follows. First the old node must be read and then the new nodes must be written to disk (one node replaces the old leaf node). Since, as with *No split*, two disk operations are already accounted for through the insertion cost, the cost of re-organization is the write cost of the new leafs:

$$C_{reorg} = C_R(\Delta l^{\tau}_{h-1}) \tag{4.3}$$

Since the new leafs are associated with a new random line, however, it is necessary to re-project the descriptors. The most efficient method is to scan the entire descriptor collection and compute the projections.[2] Thus the cost of re-projections is:

$$C_{scan} = C_S D/P \tag{4.4}$$

4.3.3 Parent Splits

With the *Parent split* policy (see Figure 4.1(d)), all immediate siblings of the leaf node to be split are considered as a set and re-organized together. The *Parent split* adds one or more leaf nodes to the sibling set and using the partitioning process of Section 3 to populate the partitions.

EXAMPLE 6.
Consider a newly created non-overlapping NV-tree ($\tau = 0$) with $l^{\tau}_{h-1} = 4$ and desired utilization $u = 67\%$. Assume that all four leaf nodes of a particular parent are nearly full,

[2] Since the split partition contained at least p descriptors, as many as p random disk reads may be required to find the descriptors. Unless the collection is very large, p random reads cost far more than a sequential scan.

when the first leaf must be split. Then there are nearly $4p$ distinct descriptor identifiers in the four leaves. After splitting, there should be $4/u \approx 6$ partitions, each with about pu descriptors.

When the original leafs overlap, care must be taken to remove that redundancy before populating the new partitions. Unlike the *Leaf split* policy, the *Parent split* policy creates wider trees but retains the original depth of the tree, resulting in lower space requirements. When the parent node is split repeatedly, however, the partitions may become "narrow" leading to potentially reduced result quality.

As before, the p_j*.prob* and p_j*.count* values of the partitions are maintained appropriately. The re-organization cost is then derived as follows. Assume a newly created tree with partitioning $L^\tau = \left[l_0^\tau, \ldots, l_{h-1}^\tau\right]$. Upon a split, l_{h-1}^τ leaf nodes must be read and $l_{h-1}^\tau + \Delta l_{h-1}^\tau$ written. As before, two disk operations are accounted for in the insertion cost, leading to the formula in Table 4.2. When the tree is modified, sibling counts are updated to correctly account for re-organization cost. As with *Leaf split*, the collection must then be scanned to re-project all the descriptors.

4.3.4 Hybrid Splits

This policy works as follows. If a leaf has few siblings, then the *Parent split* policy is employed. Once the leaf is one of l_0^τ or more siblings, however, the *Leaf split* policy is used. The resulting new leaf nodes then have one or few siblings, and the *Parent split* policy is used again. This policy should yield the lower space requirements of the *Parent split* and the higher result quality of the *Leaf split*. The cost associated with each split, of course, depends on the split policy used.

4.3.5 Re-Generation

With this policy, no splits are performed. Instead, the index is built from scratch once the first leaf overflows, resulting in a new index configuration. Assuming a uniform distribution of inserts, many leaf nodes are likely to be nearly full and thus *Re-Generation* can potentially avoid a long string of expensive splits. While this policy may not be appropriate in many high-availability situations, it is nevertheless interesting to understand the associated costs.

In order to model the costs, we must recall the index creation process. Essentially, it is a depth-first process of creating temporary files containing ever smaller sub-partitions,

until the appropriate leaf size is found. The cost calculation is made easier by the fact that the index creation process results in balanced trees. Once the tree configuration $L^\tau = \left[l_0^\tau, \ldots, l_{h-1}^\tau\right]$ has been computed, the total size D_i of the intermediate files at level i can be computed as follows:

$$\begin{aligned} D_0 &= D \\ D_{i+1} &= D_i \left(l_i^\tau / l_i^0\right) \end{aligned} \tag{4.5}$$

For each tree level i, the index creation process must then read D_i/P disk blocks and write D_{i+1}/P disk blocks, for a total cost of:

$$C_{regen} = \frac{C_S}{P} \sum_{i=0}^{h-1} (D_{i+1} + D_i) \tag{4.6}$$

4.4 Partition Files

The bulk of the cost of the *Leaf split* and *Parent split* policies is due to the cost of scanning the collection to re-project the descriptors. In order to minimize this cost, we consider introducing even further redundancy into the index by storing the descriptors for each leaf node or leaf parent node in special *partition files*. When splitting the partitions, the partition files can then be read instead of scanning the whole collection.

For *Leaf split*, one partition file corresponds to each leaf node. In this case, there are initially $f_l = |L^\tau|$ partition files and the size of each partition file is D_h/f_l. For *Parent split*, partition files are maintained one level higher in the tree, resulting in $f_p = |L^\tau|/l_{h-1}^\tau$ partition files of size D_{h-1}/f_p. The cost of scanning is thus reduced to $D_h/f_l P$ for *Leaf split* and $D_{h-1}/f_p P$ for *Parent split*. For *Hybrid split*, the partition files alternate between parent nodes and leaf nodes.

When inserting a descriptor identifier to a leaf partition, the descriptor must also be inserted into the corresponding partition file. As with inserts into partitions, there are two alternative implementations. First, the descriptors may be inserted directly into the partition file, resulting in a cost of $2C_R$; as we observe in the next section, this cost is too high to be feasible. Second, the buffer B may also be used to hold the descriptors until the buffer is flushed; this is the approach taken in our model. In this case, however, the buffer will fit many fewer descriptors, as each descriptors now requires $132 + 8 = 140$ bytes of storage compared to the 8 bytes required to store the descriptor identifier and projected

value. When flushing the buffers, the partition files must be read and written, leading to costs of $2C_S D_h/P$ and $2C_S D_{h-1}/P$ for *Leaf split* and *Parent split*, respectively.

4.5 Summary

In this section, we have defined and described the cost models of insertion and maintenance. The cost formulas are based on the I/O cost produced when inserting. When a partition reaches 100% utilization, maintenance must be performed to keep the NV-tree available for further inserts. The maintenance cost is based on re-organizing subset of the NV-tree and re-projecting that same subset to new partitions. In Section 5 we describe how we implement the cost formulas and the proposed partial overlapping method into the NV-tree simulator.

Chapter 5

Implementing the Simulation Model

In this chapter, we describe how the simulation model was implemented and the basic data structures used in the model. The simulator was written using Python 2.5 and the Eclipse development software. No additionally installed Python packages where used, creating portable code for multiple machines on different platforms.

In Section 5.1 we describe how data structures in the model are initialized and how the NV-tree is created. In Section 5.2 we explain how the data structures are maintained when records are inserted. In Section 5.3 we describe how index maintenance tasks are implemented and how they effect the data structures. In Section 5.4 we describe how partition files are implemented into the model and in Section 5.5 we describe a more efficient implementation of our proposed uniform distribution.

5.1 Overview

The code was designed to be object-oriented to encapsulate different data structures and create readable and reusable code. One external file was used containing descriptor counts for approximately 250,000 images used for the insertion simulation. The file was created by counting SIFT descriptors from a large image collection of press photos provided by one of the major Icelandic newspapers, *Morgunblaðið*.

A special *Run* class was designed to create flexible simulations for batch runs. Each experiment is defined by a list of five simulation parameters: a) maintenance strategy, b) buffer flag, c) buffer size, d) partition file flag, and e) τ factor. Not all variations are simulated, e.g., using partition files with *Re-Generation* does not result in meaningful experiment.

The *Run* class also handles error handling and messaging. Since each experiment can run for days, depending on combination of settings and hardware, we implemented the error handling to use the python *pickle* library to serialize classes when a run-time error or user cancellation was raised. This means that erroneous simulations can be executed from the same state after the error has been fixed. Re-executing failed simulations only applies if the data has not been altered because of the failed code. Completion and error messaging was also implemented in the *Run* class. Three options are possible: a) no messages, b) e-mail message, or c) SMS message to a mobile phone. The SMS service was implemented so that response to errors could be handled quickly to minimize wasted idle CPU time.

The *Run* class implements a batch of experiments. Each batch consists of multiple experiments grouped together. As each experiment is highly CPU intensive, it is recommended to distribute the experiments over multiple CPU's or machines.

Each experiment creates a new instance of the *Simulation* class. The *Simulation* class encapsulates a single experiment; it implements the insertion calls based on an external descriptor file and creates the result files for post-processing.

The *Simulation* instance creates an instance of the *NVTree* object class. The class is the main operation class of the model. The main functionality of *NVTree* is: a) create the partition nodes and its partitions, b) execute maintenance tasks when needed, c) keep reference to buffer, d) create the partition files, and e) simulate inserts into partitions.

When the *NVTree* class is initialized it uses the same initialization configuration as specified in Table 4.1. The only user provided values are the simulations parameters.

A bookkeeping class, *LeafLevelCreator*, is used by the *NVTree* to create the initial configurations and return a python list (P) containing the partition classes. The list simulates the partitions stored on disk. The leaf list is calculated by first finding L^0 based in the initialization configuration and then L^τ. The size of P is found to be $|L^\tau|$. The *LeafLevelCreator* class also provides additional functions to maintain the tree and its configurations.

5.2 Implementing Insertions

Each partition on disk is represented by the *Partition* class. The *Partition* is a simple bookkeeping class storing internal parameters which define the partition instance. As *value pairs* $(\textit{projected value}, \textit{descriptor identifier})$ are inserted, an internal counter p_j*.count* reg-

isters the insertions and calculates the local utilization of the partition $u_j = p_j.count/p$. Each partition also keeps track of the insertion probability $p_j.prob$.

When simulating insertion into P we use a uniform distribution. The estimated number of values to insert into the tree for each descriptor is initially considered as the value $V = |L^\tau|/|L^0|$. When the tree is created, all partitions are assigned with probability of insertion $p_j.prob = 1/|L^0|$. This means that all partitions have the same probability to have values inserted on creation. If the tree would never grow, we could simply select V partitions using random integers between $[0, |P|-1]$ to approximate uniform distribution. The NV-tree is simulated as a balanced flexible tree that reacts to maintenance events, however, and when adding the new partitions to the tree the likelihood of inserting will change.

Our first approach to the uniform distribution was to sequentially go through all partitions (in Section 5.5, we describe a more efficient implementation). For each partition, the probability is retrieved and compared to new random value between $[0, 1]$. If the random value is less than $p_j.prob$, then p_j is selected as partition for insertion.

EXAMPLE 7.
Assume we have $L^0 = [8, 8, 8, 7]$. *If* $\tau = 0.0$, *we have* $|P| = 3,584$ *and* $p_j.\text{prob} = 1/3,584$. *Given that probability we should select a random number lower than* $p_j.\text{prob}$ *approximately* $V = 1$ *times for each pass through* P. *If* $\tau = 1.0$, *we have* $|P| = 43,875$ *and* $p_j.\text{prob} = 1/3,584$. *We should now select a random number lower than* $p_j.\text{prob}$ *approximately* $V = 12.24$ *times for each pass through* P.

5.2.1 Direct Insertions

Algorithm 1 shows how the direct insert scenario is implemented. For all descriptors in image we traverse all partitions in P. As we traverse the partition list we compare the probability $p_j.prob$ of each partition p_j with our newly created random number. If $p_j.prob$ is larger than the random number, we insert *value pair* into the partition.

Algorithm 1 Inserting directly to disk

```
1: procedure DirectInsert(P, Image)
2:     for all desc ∈ Image do
3:         for j ← 0, |P| do
4:             if P[j].prob > new Random() then
5:                 P[j].insert()
6:             end if
7:         end for
8:     end for
9: end procedure
```

5.2.2 Buffered Insertions

Algorithm 2 shows how the buffered insert scenario is implemented. Inserting into the buffer follows the same functionality as direct inserts. Instead of inserting to disk, however, the values are inserted into the appropriate bucket in the buffer. After each insert we check for two possible scenarios to flush: a) if the buffer has reached its maximum limit, we flush all buckets in the buffer and b) if the current bucket is $2p/3$ full, we flush the bucket. The reason for flushing when the bucket has reached $2p/3$ is to avoid multiple splits of a single partition. The $2p/3$ flushing does not apply to *Re-Generation* and *No split*.

Algorithm 2 Inserting using buffer

```
1: procedure BufferedInsert(Buffer, P, Image)
2:     for all desc ∈ Image do
3:         for j ← 0, |P| do
4:             if P[j].prob > new Random() then
5:                 Buffer[j].insert()
6:                 if Buffer.Full == True then
7:                     Buffer.Flush(P)
8:                 else if Buffer[j].TwoThirdsFull == True then
9:                     Buffer.Flush(P[j])
10:                end if
11:            end if
12:        end for
13:    end for
14: end procedure
```

5.3 Implementing Index Maintenance

When new partitions are added to the tree it does not make them all equally considered when inserting. We now detail the maintenance of the p_j*.prob* and p_j*.count* values, as the index is updated.

5.3.1 No Splits

The *No split* increments the p_j*.count* with each inserted *value pair* with overfill option. This means that more than p values can be stored in the partition ($u > 1$). The additional values are stored in the partition but are represented as more than a single IO (see Section 4.3.1). Since no splitting occurs, the p_j*.prob* is the same from the start and the value of V does not change.

5.3.2 Leaf Splits

When *Leaf split* is used the partition records p_j*.count* until $u = 1$. At this point the partition returns a *split flag* to the *NVTree* class. When the flag is received the *NVTree* performs a *Leaf split* on the flagged partition.

When *Leaf split* is triggered, we grow the tree *downwards* from the flagged partition. We mark the flagged partition as the internal partition and then add t new partitions to the leaf list P. The new partition count t is calculated by finding how many partitions fit using the determined utilization.

The flagged partition is now internal and not applicable to insertions. The t new partitions are considered as child partitions and can be traversed to from the internal partition. To maintain the correct probability of insertions as the tree grows downwards we update the probability of the new child partitions as shown in equation 5.1.

$$p_{new}.prob = \frac{p_j.prob}{t - (t-1)(\tau/2)} \tag{5.1}$$

At the same time all values of the internal partition must now be divided unto the t new partitions. How they are divided depends on the τ value. If $\tau = 0$, the values are divided equally between the new partition. If $\tau > 0$, then the overlap needs to be taken into count. Equation 5.2 shows how the values are distributed over the new partitions.

$$p_{new}.count = \frac{p_j.count}{t - (t-1)(\tau/2)} \tag{5.2}$$

The basic implementation of the *Leaf split* functionality is described in algorithm 3.

Algorithm 3 Leaf Split Method

```
 1: procedure LEAFSPLIT(P, NodeList, p, t)
 2:     count ← p.count/(t − (t − 1)(τ/2))
 3:     prob ← p.prob/(t − (t − 1)(τ/2))
 4:     p.internal ← 1
 5:     node ← new Node()
 6:     for i ← 1, t do
 7:         p_new ← new Partition()
 8:         p_new.count ← count
 9:         p_new.prob ← prob
10:         P.add(p_new)
11:         node.add(p_new)
12:     end for
13:     NodeList.add(node)
14: end procedure
```

5.3.3 Parent Splits

All *partitions* $p_j \in P$ are contained within a *partition node*. The number of children in each partition node is initialized as l^{τ}_{h-1}. The partition nodes are used to group together partitions and keep track of local node configurations as the tree changes.

The partition records p_j*.count* until $u = 1$. As described in the *Leaf split* section, the partition returns a *split flag* to the *NVTree* class. At this point the *NVTree* triggers a *Parent split* on the partition node containing the flagged partition. The parent node is split in such way that Δl^{τ}_{h-1} additional partitions are added to the existing partition node such that the tree grows *sideways*. If *Parent split* is used with buffered inserts, all values stored in the buffer are calculated into the partition node before splitting. The amount of partitions to add depends on four factors: a) number of partitions currently in the node, b) the initial utilization of the tree, c) total number of values in each partition, and d) the τ value used. We first need to find how many unique values are stored in each partition node n.

$$n.unique = \left\lceil \frac{(\sum_{j \in n} p_j.count) \times (n.count - (n.count - 1) \times \frac{\tau}{2})}{n.count} \right\rceil \tag{5.3}$$

Equation 5.3 defines how we determine the number of unique values found in the current partition node. We start by summing up the number of values in each $p_j \in n$. For example, if $\tau = 0$, we return all values as unique. If $\tau \geq 0$, we filter out any doubled overlap values.

The unique values can be considered a subset of the collection. Using the number of values we can calculate the l^0 given the desired utilization. The τ and l^0 is then used to look up the correct l^τ using values from Table 3.1.

Now we need to focus on $p_j.count$ and $p_j.prob$ for all partitions in the partition node. Equations 5.4 shows how the probability is calculated for each partitions in the node and then all partitions in the node are updated with the new value.

$$p_j.prob = \frac{p_j.prob \times l^\tau_{h-1}}{l^\tau_{h-1} + \Delta l^\tau_{h-1}} \tag{5.4}$$

At the same time values from all partitions in the node must now be divided to $l^\tau_{h-1} + \Delta l^\tau_{h-1}$ partitions. How the Δl^τ_{h-1} is calculated depends on the τ value and desired utilization. Equation 5.5 shows how the values are distributed to each partition in the new node.

$$p_j.count = \frac{p_j.count \times l^\tau_{h-1}}{l^\tau_{h-1} + \Delta l^\tau_{h-1}} \tag{5.5}$$

Algorithm 4 describes how the *Parent split* is implemented. We start by getting the parent node of the partition to be split. From the parent node we can retrieve the partition count (partition siblings). We then calculate how many partitions are needed using the unique values found from the current partition node, the τ and desired utilization. We then calculate how values are distributed and assign the new probability to each partition. The current partitions must be updated and the new partitions added to the current partition node.

Algorithm 4 Parent Split Method

```
 1: procedure PARENTSPLIT(NodeList, p_j)
 2:     node ← NodeList[p_j]
 3:     size ← node.PartitionCountInNode()        // size=old partition count=l^τ_{h−1}
 4:     delta ← GetDeltaCount(node)        // delta=Δl^τ_{h−1}
 5:     p_new.count ← |node.values|×size/(size+delta)
 6:     p_new.prob ← p_j.prob×size/(size+delta)
 7:     for i ← 1, size do
 8:         p_i ← node[i]
 9:         p_i.count ← p_new.count
10:         p_i.prob ← p_new.prob
11:         node.update(p_i)
12:     end for
13:     for i ← 1, delta do
14:         p_i ← new Partition()
15:         p_i.count ← p_new.count
16:         p_i.prob ← p_new.prob
17:         node.add(p_i)
18:     end for
19: end procedure
```

5.3.4 Hybrid Splits

The *Hybrid split* is a combination of the *Leaf split* and *Parent split* strategies. We implemented the *Hybrid split* to use the partition count in each node as the deciding factor for maintenance tasks to use. We initialize each partition node with l^{τ}_{h-1} partitions. If $l^{\tau}_{h-1} < l^{\tau}_{0}$, the *Hybrid split* will chose *Parent split* as first split. As the *Leaf split* generates a partition node containing two, or a few, partitions, we have space to grow the node *sideways*. Since the *Parent split* can grow more than a single partition each time, we turn to leaf partition when the partition count in the node is equal or larger than l^{τ}_{0}.

Algorithm 5 shows how we access the partition count in the node to determine what task to use. The *Hybrid split* simply decides between the *Leaf split* and *Parent split* strategies.

Algorithm 5 Hybrid Split Method

```
1: procedure HybridSplit(p_j)
2:     count ← Node(p_j).getPartitionCountInNode()
3:     if count >= l_0^τ then
4:         LeafSplit(p_j)
5:     else
6:         ParentSplit(p_j)
7:     end if
8: end procedure
```

5.3.5 Re-Generation

When *Re-Generation* is used, the partition only records the p_j.*count* until $u = 1$. At this point the whole tree is re-computed and re-built based on the current collection size. The same rules apply to p_j.*prob* as with *No split*, since no actual partition split is done. The p_j.*prob* stays unchanged until it is re-calculated based on the new overlap configurations.

5.4 Implementing Partition Files

The partition files are created and stored using the *PartitionFiles* class. The class stores two dictionary objects; each dictionary holds buckets for different maintenance tasks. When using *Leaf split* we store descriptors inserted into each partition and for *Parent split* we store descriptors inserted into all partitions in specific partition node. When using *Hybrid split* both dictionaries are used, depending on the state of the current partition node. When we switch from *Leaf* to *Parent* split we must also re-organize the descriptors in the dictionaries.

When partition files are used with buffering we go through the same procedure as inserting values into partitions. We store the descriptors the same way as values, only they will take more space in the buffer. Using partitions with the buffer leaves smaller buffer space for values. For each partition touched with insert, we insert 132 bytes into the buffer for the descriptor, 128 bytes are stored for the 128 dimensions of the descriptor and an additional 4 bytes for the descriptor identifier. The actual value pair stored only takes 8 bytes, which means that the buffer fills up more quickly using partition files.

5.5 More Efficient Approach

Our proposed approximation uniform distribution method generates an extremely high number of random values, as we sequentially traverse all partitions for each inserted descriptor and generate a random number in every partition.

EXAMPLE 8.
Assume we have configuration [8,8,8,7] and $\tau = 1$. *We insert using 250,000 images having on average 513 descriptors, for a total of 128,250,000 descriptors. For each descriptor we need to sequentially traverse a minimum of 43,875 partitions. We therefore need at least 128,250,000* $\times$ *43,875 random numbers for this single experiment. The average time of generating single random number was gained by profiling the time taken to execute 1,000,000,000 random numbers. Single random float execution time is about* 4.5×10^{-6} *seconds. This experiment would thus take a minimum of about 7,000 hours, just to generate the random numbers.*

Our more efficient approach was to use V to estimate the number of inserts as the tree grows. The only maintenance tasks effected by V is the *Leaf* and *Hybrid split*. This is due to the tree height only growing for these tasks. Using *No split*, *Re-Generation* and *Parent split* does not change V throughout the experiments. For the *Leaf* and *Hybrid split*, we have to adjust V as the tree grows downwards.

Since V is not an integer, we sum up the *leftover* ($V - \lfloor V \rfloor$) for each inserted descriptor until the *leftover* is larger than 1. At this point we increment the number of estimated values to $\lfloor V \rfloor + 1$ for that insertion and decrement *leftovers* by *one*. This adjusts the V to include the *leftovers*.

The actual size of V also needs to change during *Leaf Split*. Equation 5.6 shows how we calculate the estimation of V as the tree grows using *Leaf split*. The increase of V is in relation to τ; when $\tau = 0$ we have no increase in V. This is as expected since the insertion path through the tree never encounters overlapped nodes. The value t in eq. 5.6 the number of partitions we initialize the new partition node.

$$
\begin{aligned}
V &= V + \left(\sum p.new - \sum p.old\right) \\
&= V + \left(\frac{t \times p.old}{t - (t-1)\frac{\tau}{2}} - p.old\right) \\
&= V + \left(p.old \times \left(\frac{t}{t - (t-1)\frac{\tau}{2}} - 1\right)\right)
\end{aligned} \tag{5.6}
$$

The changes to V are local in the tree so the estimate only changes slightly with each *Leaf split*. Part of implementing the new method was adding a new list (A) containing $|P|$ float numbers. The value $a_j \in A$ represents the odds for a *value pair* being inserted into p_j. All values in A are initialized as 1 and will store float numbers in the range 0 to 1.

When using *Leaf*, *Parent* and *Hybrid split*, we need to change the odds of access in a_j based on how p_j*.prob* changes. We use the same method to calculate a_j as we do p_j*.prob* only here a_j is based on the initial value 1, not $1/|L^0|$ as with p_j*.prob*. The probability of each partition is used to calculate V when splitting as defined in equation 5.6.

Let us look at algorithm 6 for implementation of the *More Efficient Method*. The method starts by iterating V times to search for partitions to insert. We start by selecting a random integer between $[0, |P| - 1]$. We use this integer to retrieve a partition from the partition list P. Since the *Leaf split* generates internal partitions, we check for internal state in the selected partition. If the partition is internal, we select a new random integer and check again until we find leaf partition. When a leaf partition is found we generate a random float number and compare that number to a_j of the selected partition. It the random number is smaller than a_j, then this partition is selected for insertion; otherwise, we repeat the procedure. Using this method, large scale experiments became feasible. Note that this method does not affect the simulation results.

Algorithm 6 More Efficient Method

```
 1: procedure FINDPARTITIONS(V)
 2:     for i ← 1, V do
 3:         while found=0 do
 4:             insertid ← random.randint(0, |P| − 1)
 5:             if P[insertid].internal=1 then
 6:                 found ← 0
 7:             else
 8:                 if A[insertid] > new Random() then
 9:                     found ← 1
10:                     InsertIntoPartition(insertid)
11:                 else
12:                     found ← 0
13:                 end if
14:             end if
15:         end while
16:     end for
17: end procedure
```

5.6 Summary

In this section we have described the implementation of our NV-tree simulator. The simulator inserts descriptors based on the definitions in Sections 3 and 4. We described the implementation of buffered inserts vs. direct inserts and how partition files would assist maintenance. We also define more effective implementation to simulate the uniform distribution into growing NV-tree. In Section 6 we describe our simulation experiments using the NV-tree simulator.

Chapter 6

Simulation Results

In this chapter, we describe the experimental results obtained from the simulation model described in Chapters 4 and 5. In Section 6.1 we detail the simulation environment and initial configurations for our experiments. In Sections 6.2 to 6.5 we focus on key results from our experiments and describe the results. Section 6.2 analyzes the effect of buffering inserts on a baseline strategy of *Re-Generation*. Section 6.3 examines the effect of our split policies. We study the effect of using partition files to limit the cost of re-projections in Section 6.4. Finally, in Section 6.5 we study the effects of different buffer sizes on the *Hybrid split* policy. Then we summarize the results in Section 6.6.

6.1 Simulation Environment

All experiments were run on a Beowolf Cluster with Rocks Distribution containing *nine* compute nodes. Each node consists of dual Intel Pentium(III) 1400 MHz CPUs and 2GB main memory.

The workload models a collection of high-quality press photos, which starts out at about 30 thousand photos or approximately 36 million descriptors. The inserted images consist of approximately 250,000 high-quality press photos provided by a local newspaper, *Morgunblaðið*. SIFT descriptors were extracted from the images and an external lookup file was created with the exact number of descriptors for each image. The lookup file is used as the basis for our inserted object collection. The total number of descriptors extracted from our workload images were approximately 125 million.

The experimental setup was described in Table 4.1 on page 17. To control the experiment we supply the simulation with *five* configurations defined in Table 6.1. All maintenance

Description	Values
Maintenance task	no-split, re-generation, leaf split, parent split, hybrid split
Buffer flag	0, 1
Buffer size	32MB, 64MB, 256MB, 512MB (default), 768MB, 1024MB
Partition flag	0, 1
τ factor	0.00, 0.25, 0.50, 1.00

Table 6.1: User defined simulation model parameters.

tasks are simulated with and without the buffer additions. For all simulations the buffer is set to 512MB (exception in Section 6.5, where additional buffer size simulations are conducted with *Hybrid split* only). Partition files are only applicable with *Leaf*, *Parent* and *Hybrid splits*. The four τ factors are used for all simulations.

6.2 Experiment 1: Direct vs. Buffered Inserts

In this experiment, we ran a baseline policy of *Re-Generation*. Descriptors are inserted, either directly to the index or indirectly through the descriptor buffer. Once the first partition splits, processing is halted and the index is rebuilt from the collection. During the index construction, each leaf node is filled to 67% of capacity, leaving room for insertions. The assumption behind this policy is that since the inserts are uniform, many partitions will be about to be split, and hence rebuilding the index can save many collections scans. Here we focus on the costs of the insertion and ignore the cost of the index builds, as we wish to understand the effects of buffering on insertion performance.

Figure 6.1 shows the insertion costs of this policy, for four different values of $\tau \in \{0.0, 0.25, 0.5, 1.0\}$, with and without the insertion buffer. The x-axis shows the number of descriptors that have been inserted, in millions, in addition to the nearly 36 million descriptors in the original index, while the y-axis shows the total time of the insertions (note the logarithmic scale). As the figure shows, the performance difference is enormous, as without buffering each insert requires two expensive random disk operations, while with buffering the index is occasionally scanned and written sequentially, resulting in fewer and less expensive disk operations. We therefore only consider buffered inserts in the remainder of this thesis.

It is interesting to observe the evolution of the index size during this experiment, shown in Figure 6.2. The x-axis shows the number of inserted descriptors as before, while the y-axis shows the size of the NV-tree index at each time, for each value of τ for both direct and buffered inserts, as well as the size of the collection. As the index is periodically

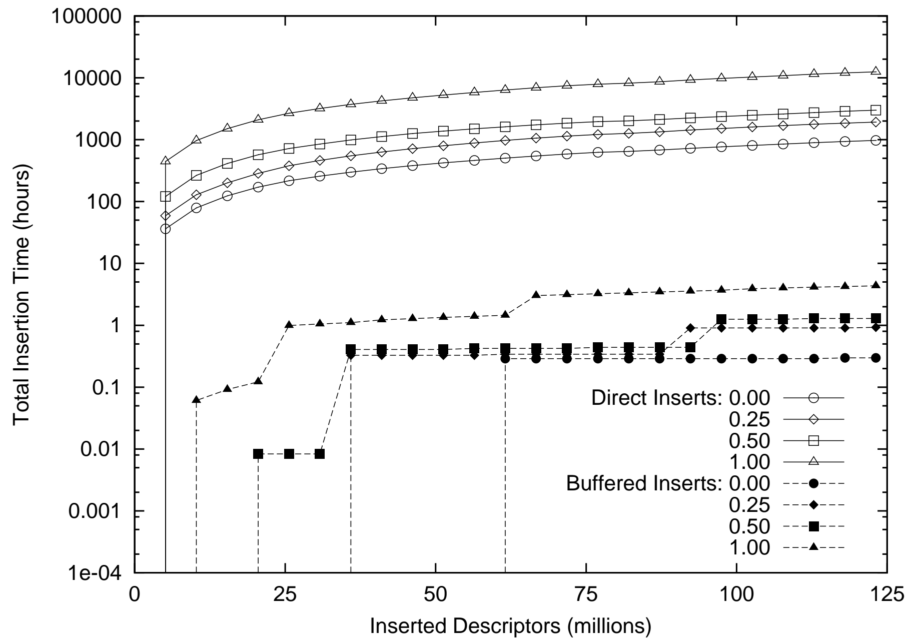


Figure 6.1: Insertion cost for Re-Generation with and without the insertion buffer (varying τ; no partition files; 512MB buffer).

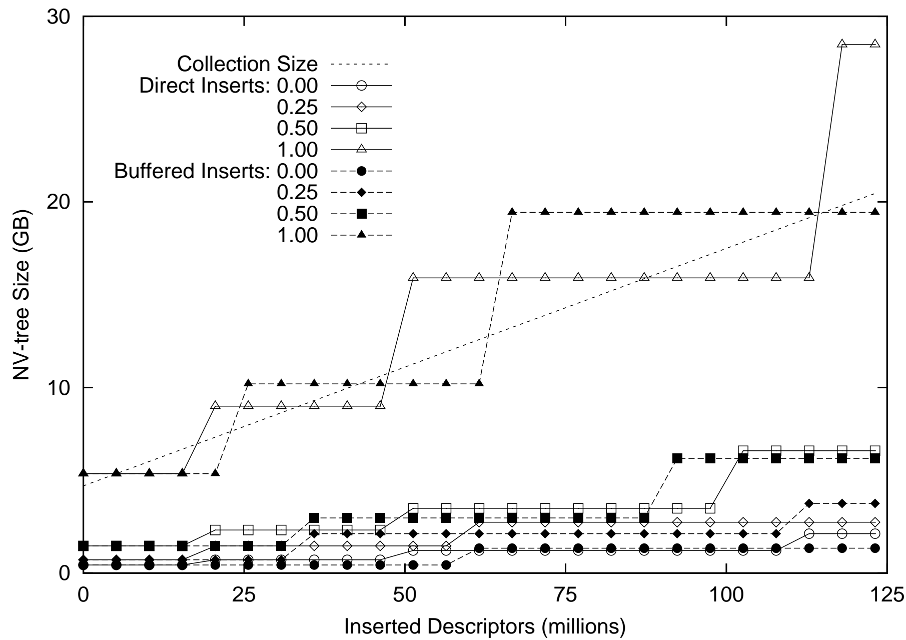


Figure 6.2: NV-tree size for Re-Generation with and without the insertion buffer (varying τ; no partition files; 512MB buffer).

rebuilt, the size and utilization are regularly reset to the desired value; these points are seen as a step-wise growth of the NV-tree index. In between, the partially empty partitions are slowly filling up until a split is required, triggering the index rebuild. There are two key observations to be made.

First, as Figure 6.2 shows, the index size is highly dependent upon the value of τ. For $\tau = 1.0$, the index is roughly as large as the collection, while for $\tau = 0.0$, the index remains at less than 2.5 GB, or about 12% of the collection size. While our simulation model does not capture the effects of buffer management, having a smaller index will result in much improved buffer management performance, in particular when the index can fit in memory. In our setting, a server with 4 GB of memory could easily fit both the index and the insertion buffer, leading to excellent performance.

Second, with buffered inserts, the index growth invariably occurs later in the insertion process. In the case of $\tau = 0.0$, this is particularly obvious, as then the insertion buffer can store the first 60 million descriptors or so, while without buffering the index starts growing after about 20 million descriptors. As a result, the collection is larger when the index is re-built, leading to a larger index. With $\tau = 1.0$, relatively fewer descriptors fit in the insertion buffer as each descriptor goes into many partitions and index *re-builds* are only postponed briefly. As we shall see later, postponing the splits may have an adverse effect on the overall performance. In this experiment, however, the effect is only positive.

6.3 Experiment 2: Split Policies

Turning to the effect of split policies, Figures 6.3, 6.4 and 6.5 show the total cost of insertions and splits for the five different split policies for $\tau = 0.0$, $\tau = 0.25$ and $\tau = 0.5$, respectively. Before analyzing the performance of the different policies, a few effects are worth noting. First, as these figures only consider buffered inserts, the total cost is much lower than in Figure 6.1. Second, due to the effect of buffering and since the simulation model only considers the cost of disk operations, no cost is registered until after 60, 35 and 20 million descriptors have been inserted, respectively.

The two simple policies of *No split* and *Re-Generation* are included as baseline references. *No split* affects query costs negatively, as it negates the key feature of the NV-tree that each query is answered in a single disk read. Since query costs are expected to dominate most applications, the *No split* policy should not be used. The *Re-Generation* policy may not be feasible in many applications, as it requires halting all processing while index re-construction takes place. Figures 6.3 through 6.5 show, however, that these policies are very efficient for insertions.

Turning to the three main split policies, we observe that *Leaf split* generally has the worst performance. Consider first Figure 6.3, where $\tau = 0.0$. In this case, no partitions overlap

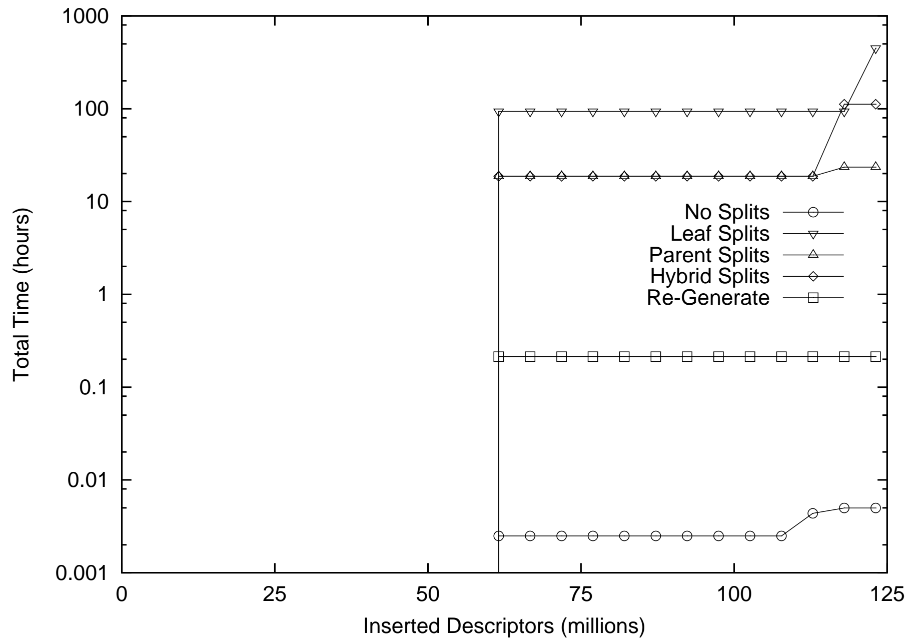


Figure 6.3: Insertion and split cost of the different split policies ($\tau = 0.0$; buffered inserts; no partition files).

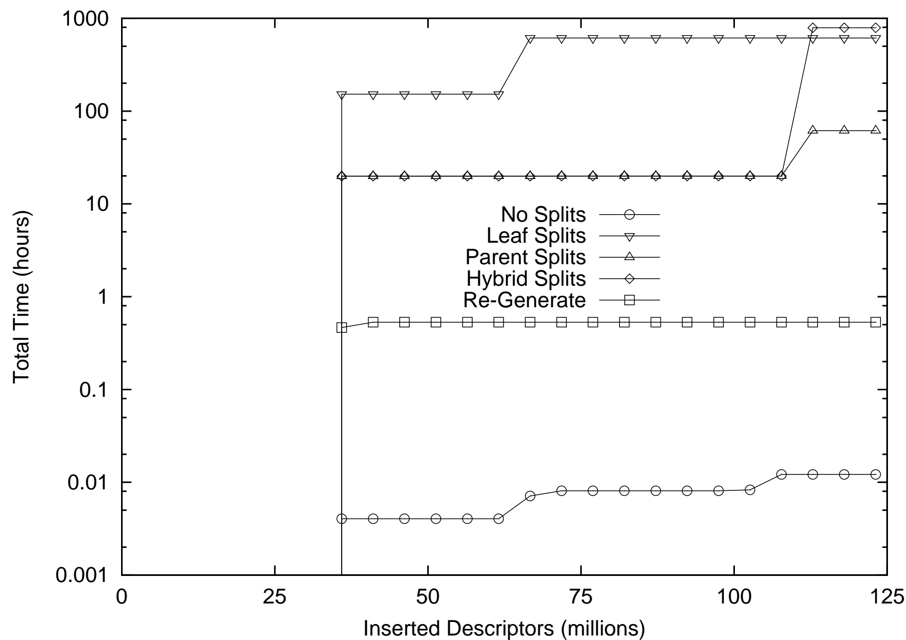


Figure 6.4: Insertion and split cost of the different split policies ($\tau = 0.25$; buffered inserts; no partition files).

and each descriptor is inserted into a single partition. The reason for the higher insertion cost of *Leaf split* is that when the insertion buffer is flushed, many partitions are split, resulting in a significantly larger index. Therefore, the insertion buffer fills up more rapidly,

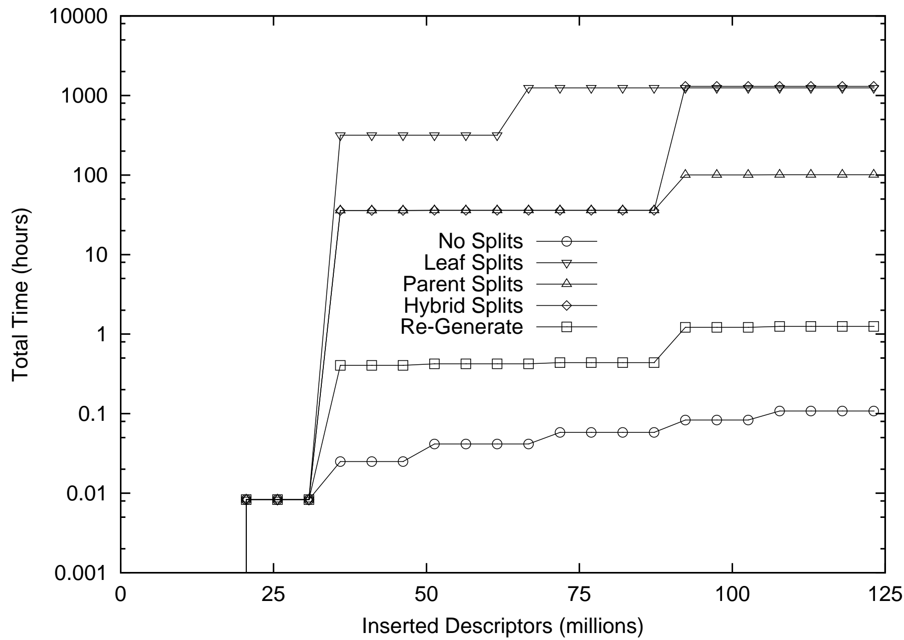


Figure 6.5: Insertion and split cost of the different split policies ($\tau = 0.5$; buffered inserts; no partition files).

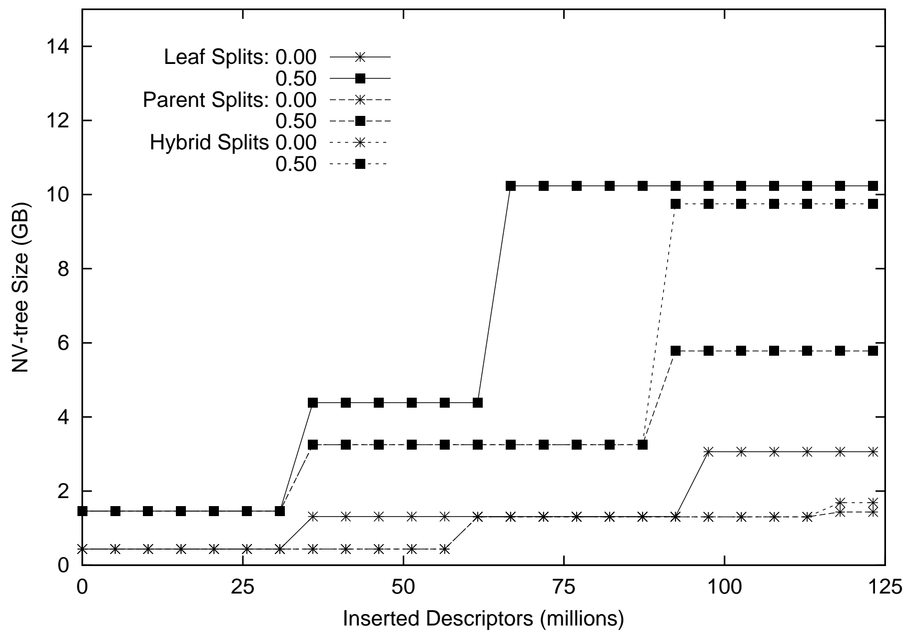


Figure 6.6: NV-tree size for Leaf, Parent and Hybrid splits without partition files (varying τ; buffered inserts).

leading to further splits, and so on. In Figures 6.4 and 6.5, where $\tau = 0.25$ and $\tau = 0.5$, respectively, very similar effects are seen.

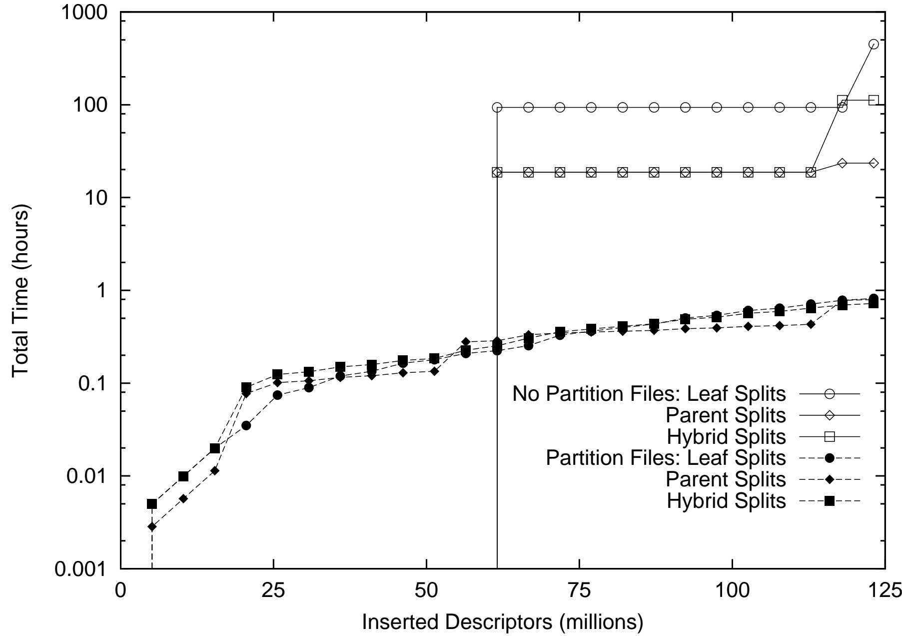


Figure 6.7: Insertion and split cost of different split policies with and without partition files ($\tau = 0.0$; buffered inserts).

Figures 6.3, 6.4 and 6.5 also show that *Parent split* generally has the best overall performance. As described in Section 4, however, the aggressive splitting of this policy is likely to lead to lower result quality. The *Hybrid split* policy is therefore recommended.

Figure 6.6 shows how the tree size increases with our main split policies, for $\tau \in \{0.0, 0.5\}$. We observe that *Leaf split* shows the largest increase in tree size, and as the τ is larger, the greater is the jump from other policies. The *Parent split* shows the lowest increase in size after 125 million inserted descriptors. The *Hybrid split*, on the other hand, shows lower index sizes until around 85 million inserts where it jumps in between the Leaf and the *Parent split*. A similar effect is observed for all τ factors. This observation supports the *Hybrid split* as the recommended split policy.

6.4 Experiment 3: Partition Files

Figures 6.7, 6.8 and 6.9 show the performance of three split policies with and without partition files for $\tau = 0.0$, $\tau = 0.25$ and $\tau = 0.5$, respectively. Using partition files is much more efficient in all cases for two reasons. First, when splits are performed, only a relatively small partition file must be read instead of scanning the whole collection. Second, when the actual descriptors must be stored in the insertion buffer, it is effectively about 90% smaller than before; as was mentioned above, having a smaller insertion buffer

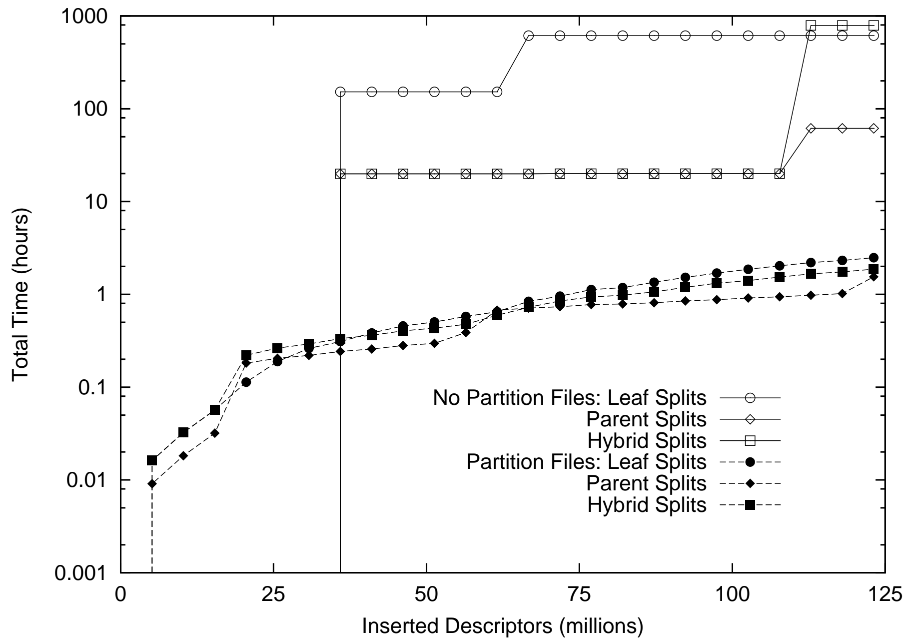


Figure 6.8: Insertion and split cost of different split policies with and without partition files ($\tau = 0.25$; buffered inserts).

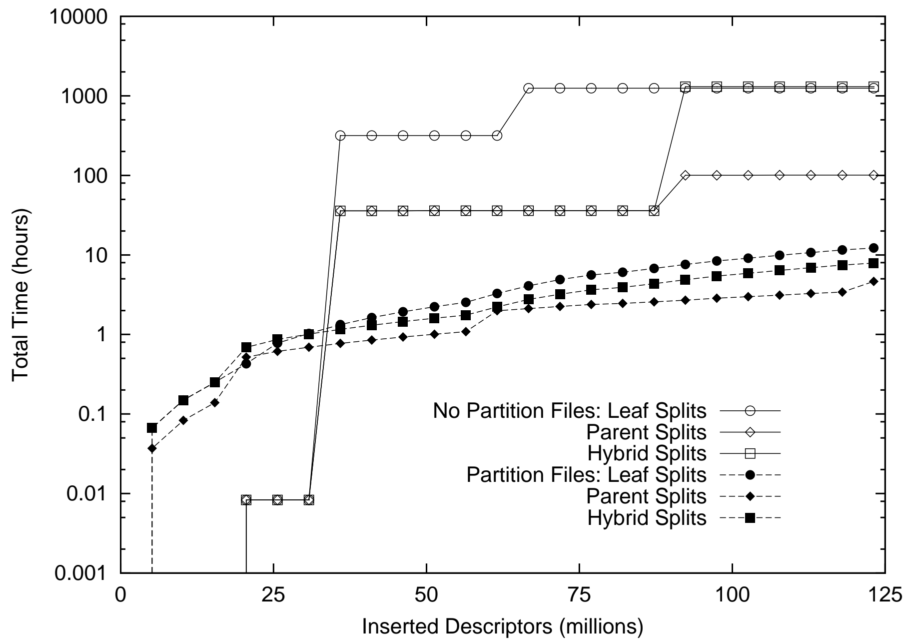


Figure 6.9: Insertion and split cost of different split policies with and without partition files ($\tau = 0.5$; buffered inserts).

can actually improve performance. Due to these effects, the savings in total execution time are about 97.7% for $\tau = 0.0$ and over 99% for $\tau = 0.5$. Note that for an index with no overlap, partition files can actually replace the descriptor collection, resulting in even further savings.

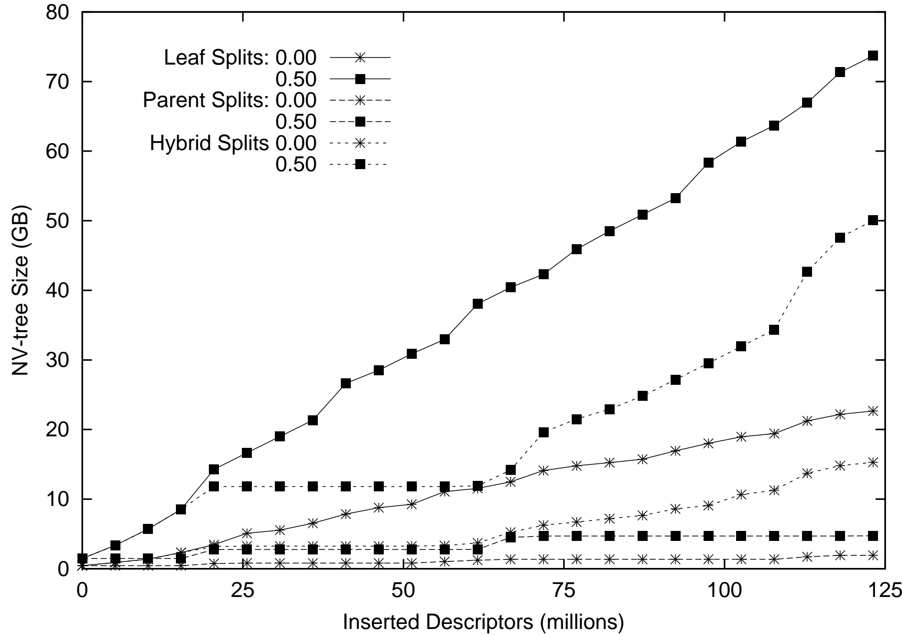


Figure 6.10: NV-tree size for Leaf, Parent and Hybrid splits with partition files (varying τ; buffered inserts).

Figure 6.10 shows the tree size of the three main split policies using the partition files; it can be compared to Figure 6.6. As can be seen, the *Leaf split* policy needs far greater space than the other policies needing around 80 GB of space after 150 million inserted descriptors using $\tau = 0.5$. As before, *Parent split* generally perform best and *Leaf split* worst, but due to quality concerns and moderate disk space requirements the *Hybrid split* is the recommended policy.

6.5 Experiment 4: Buffer Sizes

Turning to different buffer sizes, Figure 6.11 shows the total cost of insertions and splits for six different buffer sizes, for $\tau \in \{0.0, 0.25, 0.5\}$, for the *Hybrid split* policy only. The x-axis shows the different buffer sizes in megabytes, while the y-axis shows the total cost of the insertions (note the logarithmic scale on the x and y-axis). As the figures shows, the performance difference with and without partition files is very high as described in Section 6.4. There are two key observations to be made.

First, without the partition files there is a small increase in total cost as the buffer size is increased. This shows that smaller buffer sizes can be more beneficial. It is interesting to see how using 256MB and 512MB buffer sizes with $\tau = 0.0$, in particular, shows

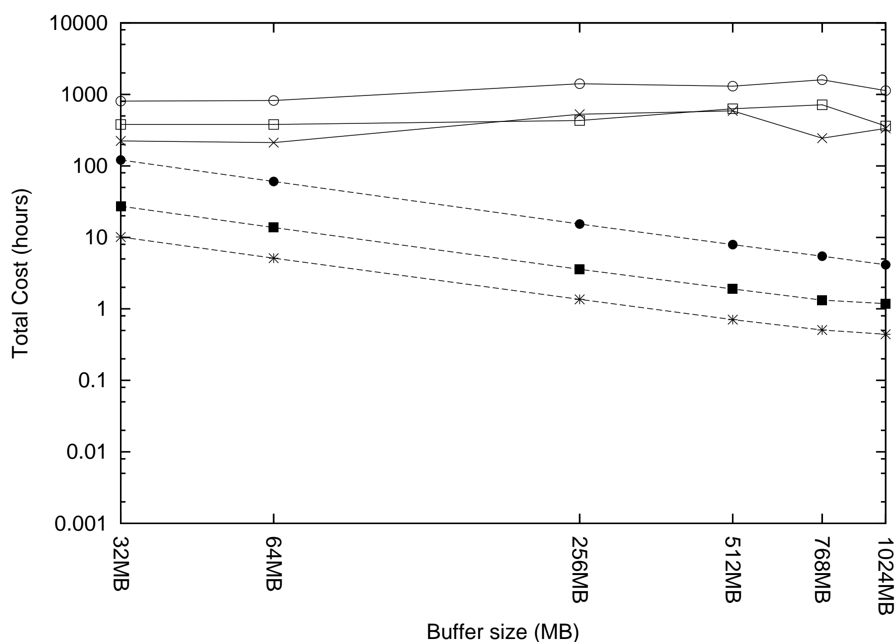


Figure 6.11: Insertion and split cost for Hybrid split (varying τ; with and without partition files; varying buffer).

increased total cost. This is explained by additional split costs generated very late in the insertion process, when the collection is larger.

Second, increasing the buffer size has a positive effect on the total cost using partition files. Using $\tau = 0.5$ and 32MB buffer takes around 120 hours to insert 250,000 images. Increasing the buffer size to 1,024MB decreases the total cost to 4 hours.

6.6 Summary

In our simulations we have seen that using the buffer improves the performance of insertions by orders of magnitude. We have also seen that using the *Hybrid split* policy is the best choice for inserts. The *Leaf split* policy generates very large and deep tree with a small number of partitions in each leaf node. The *Parent split* policy creates a smaller tree but with a very high number of partitions in each leaf node. We therefore recommend the *Hybrid split* policy, since it will work to maintain the balanced structure of the tree and we expect the search quality to be maintained. Using *Re-Generation* should be investigated further to see, whether, ways to maintain the index availability during re-generation can be implemented. This could make *Re-Generation* the most efficient split policy. Using partition files shows a large decrease in total cost of insertion due to lower re-projection cost when split policies are executed but increases the disk space needed. If disk space is

not an issue, we recommend using partition files. Using partition files creates a redundant database collection. It could be argued that the partition files could be substituted for the actual database collection to limit the redundancy further. As the NV-tree grows downwards and sideways using *Leaf split* and *Parent split*, respectively, it can affect the search results. While we do not investigate the effects of split policies any further in this thesis, however, we experiment with the effect of the τ value on search quality and performance, as the τ value determines the partial overlap and is a major factor in the index size and insertion performance. In Section 7 we therefore analyze the effect of partial overlapping on the search performance and quality.

Chapter 7

Effect of τ on Search Quality and Performance

In this chapter we run detailed search experiments on live data using the flexible configuration, as described in Section 3. The flexible configuration was implemented into the balanced NV-tree to investigate the effect of partial overlapping on search quality and performance. The query searches were conducted without being affected by insertions and maintenance. Our simulation model was designed to simulate a single index. Running our experiments in this chapter using a single index resulted in poor quality results, however, we therefore added experiments using two and three indexes. In (Lejsek et al., 2008), single index search has been solved using the *unbalanced* NV-tree; by combining smaller leafs, better line selections and other configurations, the *unbalanced* NV-tree is quite effective for single index searches. A single index can return more false positives, but adding more indexes to the NV-tree has shown to decrease false positives. In Section 7.1, we detail the experimental environment and initial configurations for our experiments. In Section 7.2 we run extensive query searches and analyze the results. In Section 7.3, we analyze the search performance of the flexible configuration. We then summarize the results in Section 7.4.

7.1 Experimental Environment

The data collection was created using 29,277 high-quality photo images, generating a total of 35,484,770 local descriptors. Each descriptor is a 128 dimensional SIFT feature vector. We created three different NV-trees using each of the five values of $\tau \in \{0.0, 0.25, 0.5, 0.75, 1.0\}$. Each of the NV-trees is unique in two ways; a) different lines

are selected for each node in the index and b) different overlap is created, based on the τ value.

The workload consists of 120 images extracted from the data collection. For each original image, 26 image modifications were created. Each modification is described in Table 7.1. The total workload, thus consisted of 3,120 query images.

All experiments were run on an Intel Xeon 3.20GHz with 1024KB L2 cache and 2GB memory. The same computer was used to minimize hardware influence on timing results. No special *memory warming* was conducted before or between queries. Since different NV-trees were used each time, this was not considered necessary.

The NV-tree internal buffer size was set to 1,000 partitions or 128MB. The internal buffer manager uses LRU (Least Recently Used) algorithm to exchange partitions to and from the buffer manager.

The NV-tree offers stopping rules to increase performance. We repeated all experiments using stop rules for all τ values. More detailed description on the stopping rules algorithm can be found in (Lejsek, Ásmundsson, Jónsson, & Amsaleg, 2006).

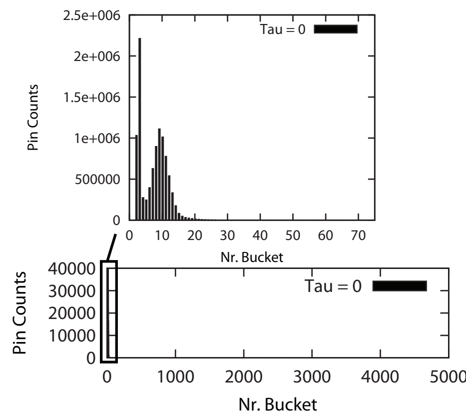
When more than one NV-tree is used, the descriptor distance is computed in the NV-tree using median rank distance. The NV-tree evaluates every descriptor as nearest neighbor using the *median rank aggregation* (MEDRANK) (Fagin, Kumar, & Sivakumar, 2003). The MEDRANK needs to find the same *descriptor id* in more than half of the indexes to return it as a positive result. When the same *descriptor id* is found, the image containing that descriptor receives a *vote*. As more descriptors are found, the more *votes* the image collects. All images that receive a *vote* are written to a result file. The image with the highest vote score is considered the winner and the next competitor is the image that scores second most votes.

For the query results we focused on: a) the percentage of images found over all image variants, b) number of descriptors needed to finish the search, c) number of descriptors found for the winning image, d) number of descriptors found for the next competitor, e) total pin counts and CPU ticks for each pinned partition, and f) the total search time.

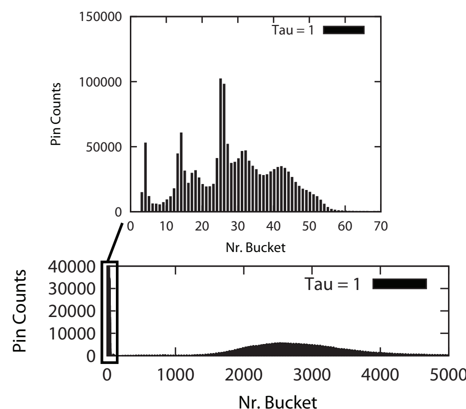
The NV-tree uses its own internal buffer manager. When a partition is requested it is read into memory and pinned. The operating system uses an additional buffer manager that can use performance techniques to transfer data to and from disk such as pre-fetching. When the NV-tree requests a partition into its own memory manager it may have already been moved into memory from disk by the operating system.

Modif.	Description	Class.	Modif.	Description	Class.
AFFINE 1	Shear in X	Easy	MEDIAN 9	9x9 median filter	Med.
AFFINE 2	Shear in Y	Easy	NOISE 5	Applied 5% noise	Med.
AFFINE 3	Shear in X and Y	Easy	PSNR	Watermark removal	Easy
CONV 1	Low brightness	Med.	RESC 200	Image scaled to 200%	Med.
CONV 2	High brightness	Med.	RESC 75	Image scaled to 75%	Easy
CONV 3	Sharpen	Med.	ROT 10	10° rotation	Easy
CONV 4	Strong sharpen	Hard	ROT 90	90° rotation	Easy
CONV 5	Emboss filter	Hard	ROTCROP 2	2° rotation and crop.	Easy
COTR 1	High contrast	Med.	ROTCROP 5	5° rotation and crop.	Easy
COTR 2	Low contrast	Hard	ROTSCAL 2	ROTCROP 2 + scaling	Med.
CROP 75	Crop 75% from cent.	Easy	ROTSCAL 2	ROTCROP 5 + scaling	Med.
JPEG 15	15% quality	Med.	SS 1	Change in color space	Med.
JPEG 80	80% quality	Easy	SS 2	Change in color space	Easy

Table 7.1: Image modifications variants.



(a) 3 indexes and $\tau = 0.0$



(b) 3 indexes and $\tau = 1.0$

Figure 7.1: Distribution of CPU ticks

To see whether a partition had been read from disk or memory, we added a time measurement of CPU ticks around all pin requests. We created a histogram using the CPU ticks to determine the local boundary between the pins *from memory* and *from disk*. A mod-

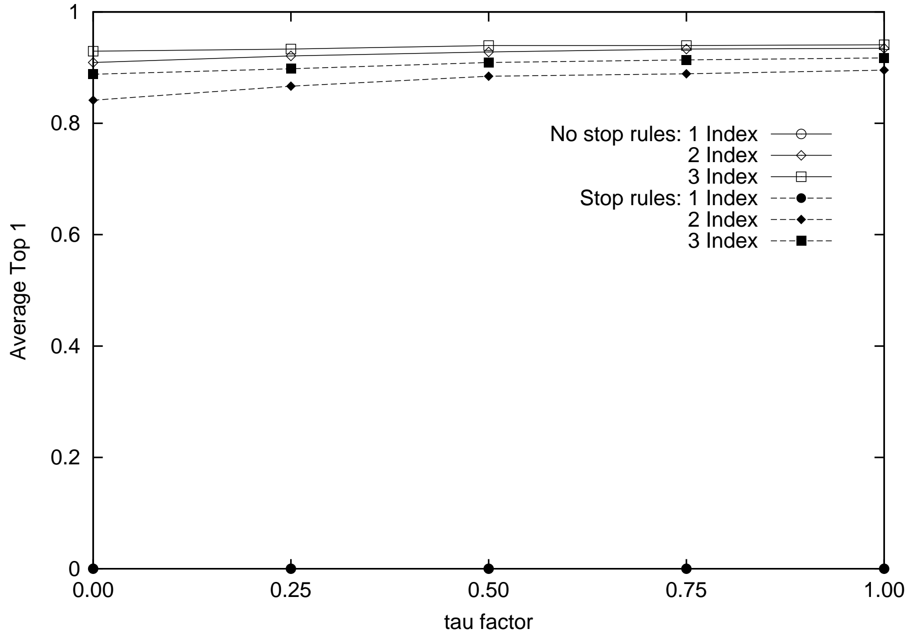


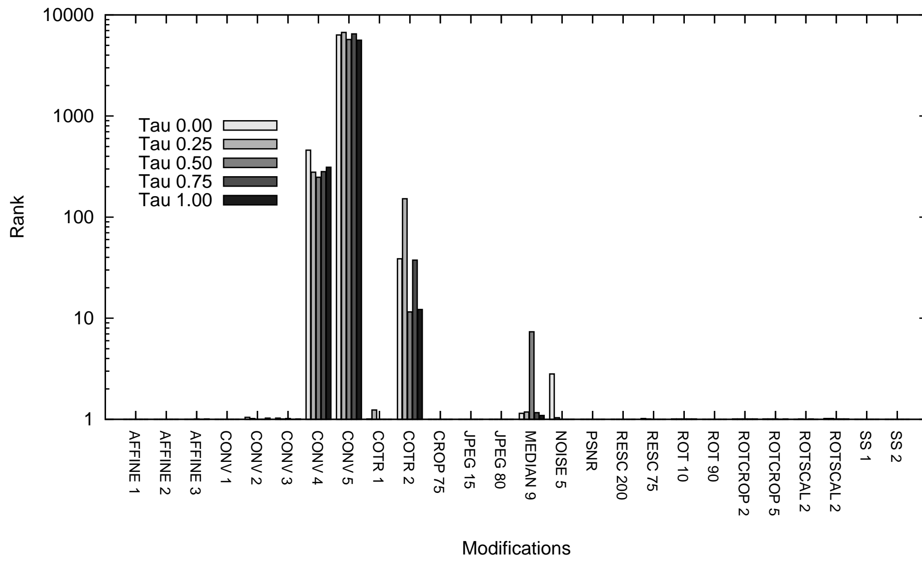
Figure 7.2: Average top score result for 3,120 images (varying τ; 1–3 indexes).

erate value was picked as a local minimum threshold value and pin counts were divided into groups of *from memory* or *from disk* pins. We choose to use 3,000,000 CPU ticks ($\approx 9x10^{-4}$ seconds) as a divider between the two groups. Figures 7.1(a) and 7.1(b) show the distribution of CPU tick counts using $\tau = 0.0$ and $\tau = 1.0$, respectively. Both figures show both the overall distribution, and the detail of the fastest reads. Our chosen value is located in bucket number 75, but as Figures 7.1(a) and 7.1(b) show the results are not sensitive to that value.

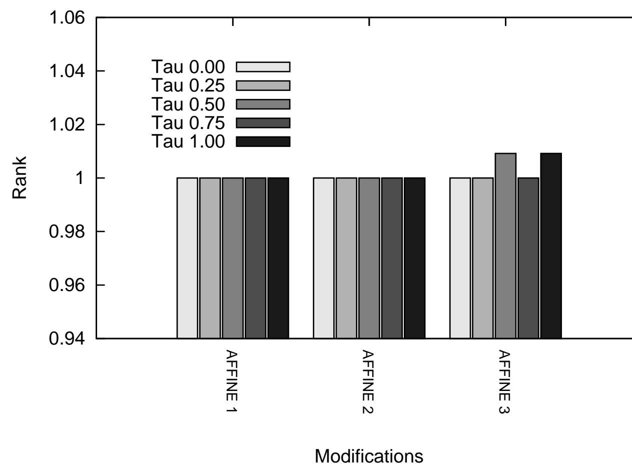
7.2 Experiment 1: Search Quality

In this experiment, we ran 3,120 query images on each of the *five* balanced NV-tree variants. Image types like the CONV4, CONV5 and COTR2 are extreme modifications and have been shown be the least effective query modifications using the NV-tree. Each query was repeated using *1*, *2* and *3* indexes.

Figure 7.2 shows the aggregated average top score over the five different values of $\tau \in \{0.0, 0.25, 0.5, 0.75, 1.0\}$. The x-axis shows different τ values, while the y-axis shows the average top score. For each modification we aggregated the top score and divided by the number of modifications. As the figure shows, the number of images at top rank



(a) All modifications (varying τ; 3 indexes).



(b) Easy modifications (varying τ; 3 indexes).

Figure 7.3: Image modification ranking

increases slightly as the τ value increases. In this figure there are two key observations to be made.

First, using only a single index returns a very low top score; this is due to configuration in the *balanced* NV-tree. As explained in the chapter introduction, this has been solved using the *unbalanced* NV-tree.

Second, the difference in overall quality of the query search for *two* indexes is 2.5% between $\tau = 0.0$ and $\tau = 1.0$. For *three* indexes the difference is only 1.1%. Going from two indexes to three indexes increases the quality on average by 1.1%. Adding the stop rules decreases the overall quality using *two* indexes by an average of 5% and for *three* indexes the decrease is only 3.1%.

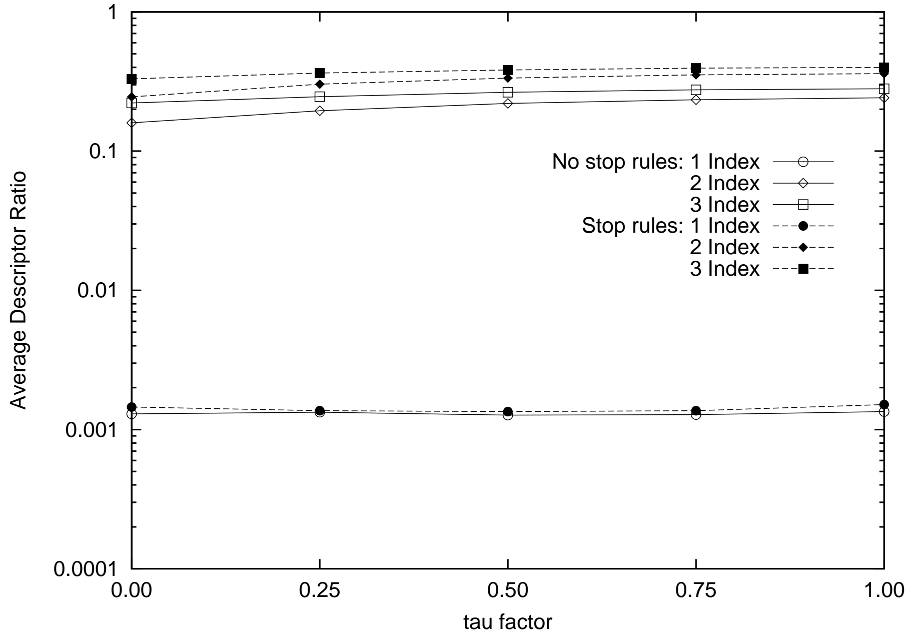


Figure 7.4: Average descriptor ratio for 3,120 images (varying τ; 1–3 indexes).

Figure 7.3 shows the average rank of each modification by τ value using *three* indexes. The x-axis shows τ values grouped by image modifications, while the y-axis shows the average search rank (note the logarithmic scale). As the figure shows, the ranking is very high, almost always close to 1 for the easy and medium modifications. We also see that the hard modifications have a very low rank.

In Figure 7.3 we have one key observation, namely that different τ values do not significantly affect the ranking of the result images. For modifications like COTR2 and MEDIAN9 we see that some τ values show lower ranking with higher τ values. Figure 7.3(b) shows in more detail three easy modifications (note the y-range) found in Figure 7.3(a). As seen the query images are found in almost all cases. For $\tau \in \{0.5, 1.0\}$ for the AFFINE3 modification we did not have the correct image in top rank in two cases of 120. The difference between the indexes is a) line selection on creation and b) τ value. It is more likely that the lower rank is due to worse lines found for that modification. Since lower τ indexes return the images with higher rank, we cannot assume that only the τ value is causing the images not to be found.

Figure 7.4 shows the aggregated descriptor ratio over the five different values of $\tau \in \{0.0, 0.25, 0.5, 0.75, 1.0\}$. The x-axis shows different τ values, while the y-axis shows the average descriptor ratio. The descriptor ratio is aggregated for each image modification and the average found. As the figure shows, the descriptor ratio increases slightly with a higher τ value. In this figure there are two key observations to be made.

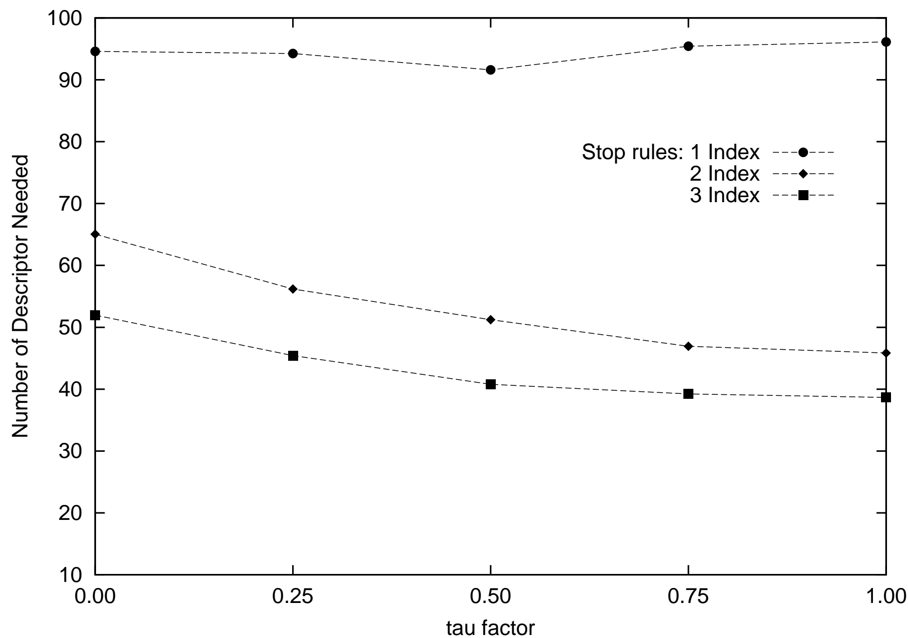


Figure 7.5: Number of descriptors needed for 3,120 images using stop rules (varying τ; 1–3 indexes).

First, as expected, the ratio for a single index is very low. This corresponds to the result in Figure 7.2 as the single index fails to find any result, and the average ratio between winner and competitor descriptors is very low. Using only a single index with a small collection introduces noise on into the search. Descriptors are being returned as false positives and the correct images are not being found. Adding the second index to the NV-tree reduces this noise and filters out false positive descriptors. With larger collections the noise level should decrease.

Second, it is interesting to see that using stop rules results in a higher descriptor ratio than searching all descriptors in the query image. When using stop rules we can experience that larger set of meaningfully nearest neighbors can be positioned either; a) early in the search or b) late in the search. When the set is positioned early, we get a higher descriptor ratio since either the correct image or false positive has been found and many of the competitors will not be searched. If the set is positioned late in the search we may *miss* the image in the search. The higher descriptor ratio in our experiments is most likely due to meaningful descriptors found early in the search.

Figure 7.5 shows the aggregated descriptors needed using stop rules over the five different values of $\tau \in \{0.0, 0.25, 0.5, 0.75, 1.0\}$. Note that without stop rules all descriptors are searched, and therefore we only focus on stop rules for descriptors needed. The x-axis shows different τ values, while the y-axis shows the average number of descriptors needed to finish the search using stop rules. The results are aggregated for each image modifi-

cation and the average found. As the figure shows, the number of descriptors needed to return either found or not found decreases considerably with a higher τ value using the two and three indexes. In this figure there are two key observations to be made.

First, using single index results in a high number of descriptors needed. This is expected as the search hardly finds any images matching the query images. This leads to the search continuing until the NV-tree stops the search confident that the correct image will not be found.

Second, as more indexes are used, the stop rule gains confidence earlier in the search. Using $\tau = 0.0$ with *three* indexes need on average 52 descriptors to stop searching and only 39 descriptors with $\tau = 1.0$. Using two indexes the same values increase to 65 descriptors and 45 descriptors, respectively. On average 25% to 30% fewer descriptors are needed using $\tau = 1.0$ than $\tau = 0.0$.

7.3 Experiment 2: Search Performance

In this experiment, we ran 3,120 query images on each of the *five* balanced NV-tree variants. For each query image we recorded the pin count and the CPU ticks needed to fetch the partition into the internal buffer manager. We also recorded the total number of partitions needed to finish searching all 3,120 images and the total time needed for all images.

Figure 7.6 shows the total NV-tree size using 1, 2 and 3 indexes over the five different values of $\tau \in \{0.0, 0.25, 0.5, 0.75, 1.0\}$. The x-axis shows different τ values, while the y-axis shows the size in gigabytes. In this figure there are two key observations to be made.

First, the index difference between $\tau = 0.0$ and $\tau = 1.0$ is very large. Using a smaller τ value decreases redundancy in the NV-tree and creates in turn a smaller index. Using $\tau = 0.0$ will generate a 92% smaller index than $\tau = 1.0$. Adding indexes to the search results in a linear increase in the total disk size needed.

Second, 32-bit operating systems offer up to 4GB of main memory. This means that *four* indexes using $\tau = 0.25$ would fit into the main memory and *two* indexes using $\tau = 0.5$. Being able to fit the index into memory will greatly improve performance as we will see.

Figure 7.7 shows the average number of partitions needed to search using 1, 2 and 3 indexes over the five different values of $\tau \in \{0.0, 0.25, 0.5, 0.75, 1.0\}$. The x-axis shows

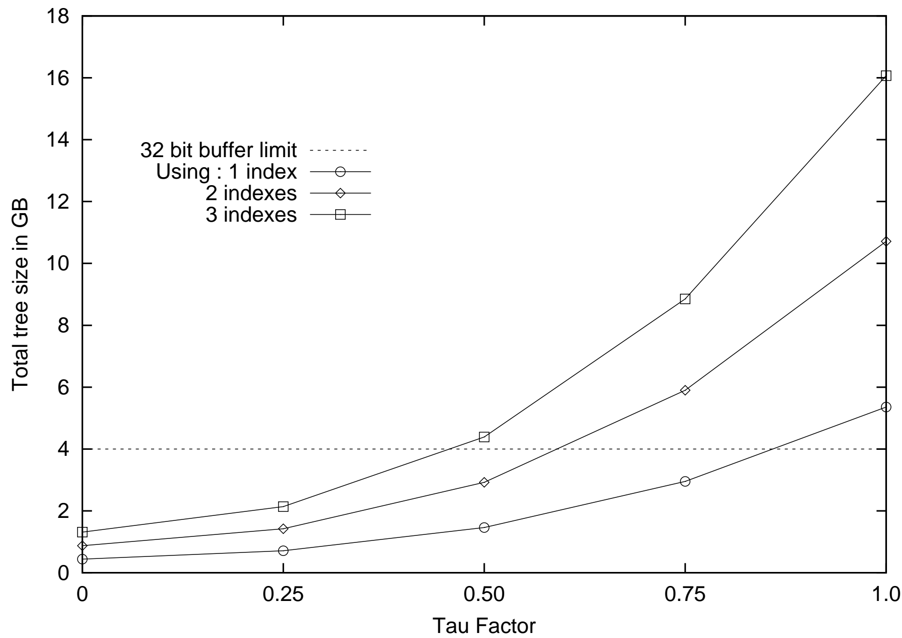


Figure 7.6: Total NV-tree size (varying τ; 1–3 indexes).

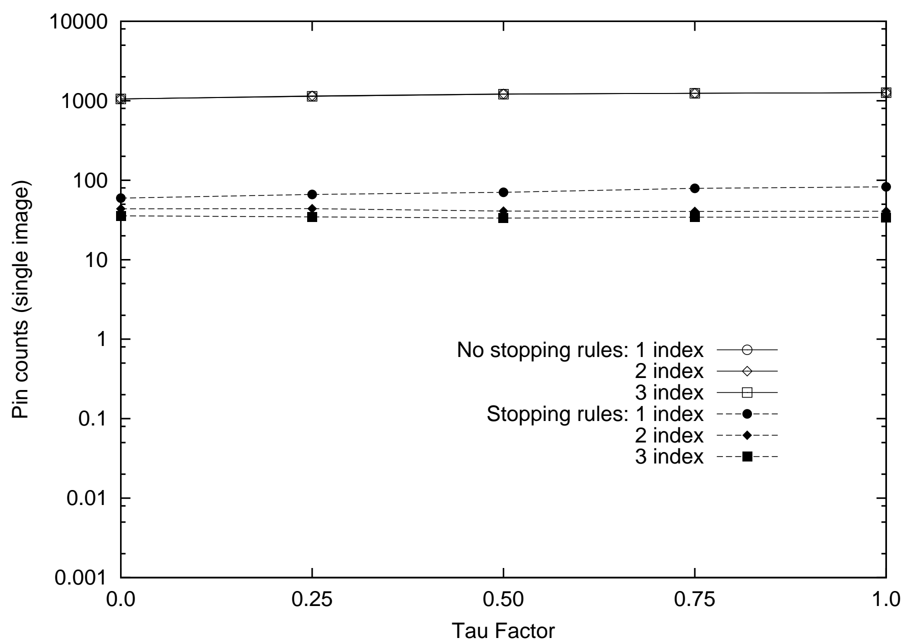


Figure 7.7: Total reads needed (varying τ; 1–3 indexes).

different τ values, while the y-axis shows average total number of partitions needed for each query image for each index. For *two* and *three* indexes, the data points are divided by the number of indexes and number of images. This is done to compare all indexes based on average value of single image.

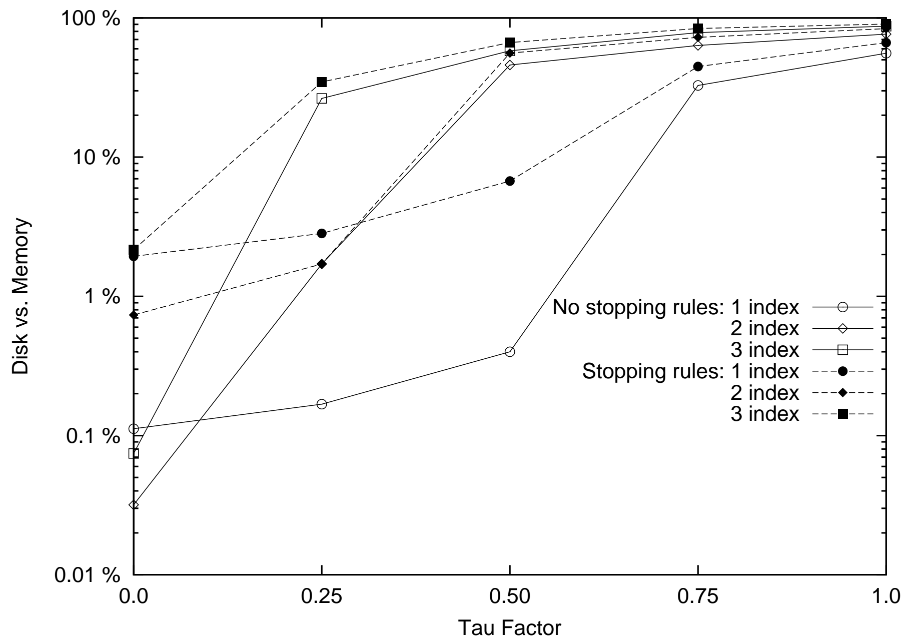


Figure 7.8: Percentage read from disk vs. from memory (varying τ; 1–3 indexes).

Figure 7.8 shows the percentage of pins from disk using 1, 2 and 3 indexes over the five different values of $\tau \in \{0.0, 0.25, 0.5, 0.75, 1.0\}$. The x-axis shows different τ values, while the y-axis shows the percentage of pins from disk. In the Figures 7.7 and 7.8 there are three key observations to be made.

First, as seen in Figure 7.7, when using fewer indexes and stop rules we need to search more partitions. This corresponds to Figure 7.5 where we see that we need a larger set of query descriptors to finish the search with fewer indexes. When searching all query descriptors we need approximately the same number of partitions to search. The only difference is that using a single index requires a single disk read and then an additional disk read for each index added.

Second, as figure 7.8 shows, searching with stop rules needs a higher percentage of reads from disk than searching all query descriptors. Using stop rules uses on average 96% fewer partitions over all τ values, than searching all query descriptors. Since we do not *warm* up the memory, large portition of the partitions come from disk using stop rules.

Third, using $\tau = 0.0$ and 3 indexes, each index size is 448 MB (1,344 MB total) and the internal buffer manager only requires 0.07% of the its partitions being read from disk, despite not warming the buffer. When the τ value is increased, the index gets larger and more disk reads are required during the search. For $\tau = 0.5$ and 3 indexes, the index is

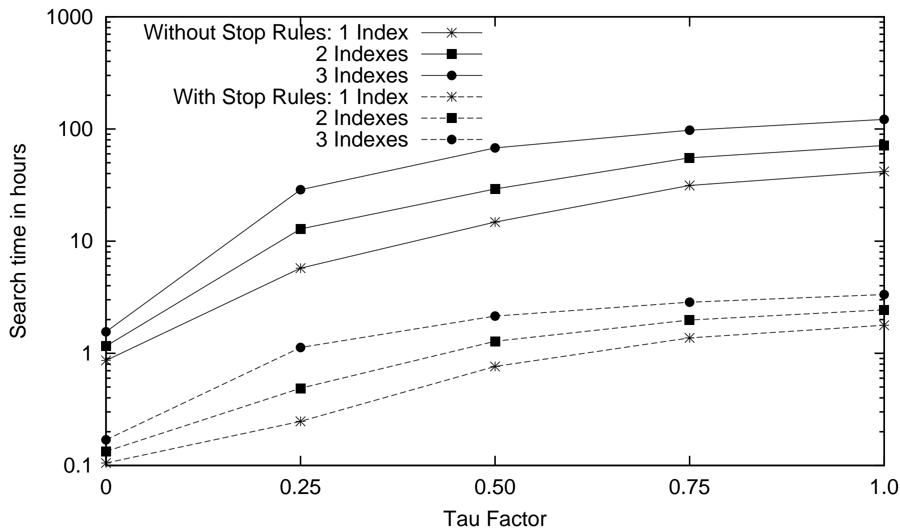


Figure 7.9: Query search time for 3,120 images (varying τ; 3 indexes).

1,500 MB (4,500 MB total) and we need approximately 58% of the required partitions to be read from disk and with $\tau = 1.0$ the index has reached 5,484 MB (16,452 MB total) and we need 87% of its partitions from disk. The internal buffer manager was configured to use 128 MB as its internal memory; even so the operating system is storing almost all partitions in the main memory when $\tau = 0.0$. As the τ value grows, the main memory is not able to keep all partitions in memory so we have increased disk access.

Figure 7.9 shows the time taken to search 3,120 images using 1, 2 and 3 indexes with and without stop rules for five different values of $\tau \in \{0.0, 0.25, 0.5, 0.75, 1.0\}$. The x-axis shows different τ values, while the y-axis shows the search time in hours. Note that real time measurements can be affected by access to disk and CPU by other processes. Running the same experiments again could result in slightly different values, but the overall trend should be the same. In Figure 7.9 there are three key observations to be made.

First, as fewer indexes are used, the better performance is measured. This is expected since fewer partitions need to be read from disk. As the figure shows, adding an index to the NV-tree increases the search time on average 35% for stop rules and 44% for full search.

Second, the performance difference between $\tau = 0.0$ and $\tau = 1.0$ is very high. Using $\tau = 0.0$ has 98.7% better performance than $\tau = 1.0$ using *three* indexes. The average search time is 1.8 seconds using $\tau = 0.0$ and 143 seconds for $\tau = 1.0$. This shows that minimizing redundancy in the NV-tree has enourmous effect on performance.

Third, using stop rules with the flexible configuration improves the performance even more. The average search time is 0.19 seconds using $\tau = 0.0$ and 3.85 seconds for

$\tau = 1.0$. Using $\tau = 0.0$ has 95% better performance than $\tau = 1.0$ using *three* indexes and stop rules.

7.4 Summary

In our experiments we have seen that reducing the τ value only slightly decreases the search quality in return for a huge improvement in search performance. Using *three* indexes we have a 1.1% decrease in quality, 92% smaller index and a 98.7% improvement in performance. We recommend adjusting the τ value to maximize the amount of partitions stored in the main memory. We have also seen that using stop rules improves performance even further. Using $\tau = 0.0$, we get 3.1% decrease in quality using *three* indexes but gain 94.9% improvement on performance. Using the memory as primary storage for the NV-tree is extreamly beneficial for performance. Redundancy is added to the NV-tree, both by using $\tau > 0$ and by adding additional indexes to the NV-tree. We recommend using as low τ value as possible. Using $\tau = 0$ is recommended, if enough buffer space is available. When the index is larger than the available buffer space, we recommend selecting the lowest possible $\tau > 0$ value, as it will continue to support single disk read for each query descriptor. Search quality can be improved by adding indexes to the NV-tree and selecting low τ value will assist in keeping larger portion of the indexes in the buffer.

Chapter 8

Conclusion

In this thesis we have addressed the implementation and performance of dynamic behavior of the balanced NV-tree, using a detailed simulation model. We have experimented with various parameters that effect the insertion cost and index size. In particular, we have observed how the overlap (τ value) has a major impact on the NV-tree. Based on this observation, we also investigated the effect of overlap on search quality and performance using live data. From our simulations and query experiments we make three key conclusions.

First, we observed that using an insertion buffer is a very efficient technique. Furthermore, using partition files results in very significant performance improvements. Note that for an index with no overlap, the partition files can actually replace the descriptor collection, resulting in even further savings. Interestingly, using a large buffer shows improved performance only when used with partition files. We recommend that buffered insertions and partition files should be used together to maximize the insertion and maintenance performance.

Second, although the *No split* and *Re-Generate* policies show good performance, they are not always applicable in practice, due to high query costs and significant unavailability, respectively. Using *Re-Generation* should be investigated further, however, to see whether ways to maintain the index availability during re-generation can be implemented. This could make *Re-Generation* the most efficient split policy. From our three basic split policies, we see that *Leaf split* shows the worst insertion cost and creates a large index. The *Parent split*, showed the lowest insertion cost, but due to repeated splitting this technique can lower the search quality. Therefore, we recommend using the *Hybrid split* policy, which performs similarly to *Parent split*, but avoids the lower search quality.

Finally, the simulations indicated that overlap is a key factor in determining performance. We showed that reducing the overlap improves the search performance significantly with minimal effect on search quality. When using the balanced NV-tree configuration, we need to use 2-3 indexes. Using *no overlap*, the search quality is 1-3% worse then using *full overlap*, but we have around 95% improvement in performance. The main reason for this performance improvement is improved buffer performance due to smaller indexes.

There is a trade-off using redundancy. First, reducing the τ value will decrease the index size and improve overall performance but will decrease the search quality. Second, adding additional indexes will increase the space requirements, but improves the search quality. Both scenarios can work together, adding additional indexes to improve search quality can be countered with a small τ value for performance.

We recommend selecting $\tau = 0$ for two main reasons. First, it maximizes the insertion performance and, secondly, we fit a higher proportion of the index in memory. When the collection is small, the whole index can actually fit into memory, resulting in an extremely efficient search. As the index grows, however, more and more disk reads are needed due to memory limitations. At this point, increasing the τ value could be beneficial, even if it increases redundancy and the index size gets larger.

Bibliography

Amsaleg, L., & Gros, P. (2001). Content-Based Retrieval Using Local Descriptors: Problems and Issues from a Database Perspective. *Pattern Analysis and Applications*, *4*(2/3), 108-124.

Berchtold, S., Böhm, C., & Kriegel, H.-P. (1998). The Pyramid-Technique: Towards Breaking the Curse of Dimensionality. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data* (pp. 142–153). New York, NY, USA: ACM Press.

Böhm, C., Berchtold, S., & Keim, D. A. (2001). Searching in High-Dimensional Spaces: Index Structures for Improving the Performance of Multimedia Databases. *ACM Computing Surveys*, *33*(3), 322–373.

Casey, M. A., & Slaney, M. (2006). Song Intersection by Approximate Nearest Neighbor Search. In *ISMIR 2006, 7th International Conference on Music Information Retrieval* (p. 144-149). Victoria, Canada.

Datar, M., Indyk, P., Immorlica, N., & Mirrokni, V. (2006). *Locality-Sensitive Hashing using Stable Distributions*. Cambridge, MA: MIT Press.

Fagin, R., Kumar, R., & Sivakumar, D. (2003). Efficient Similarity Search and Classification via Rank Aggregates. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data* (p. 301-312). San Diego, CA.: ACM.

Guttman, A. (1984). R-trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data* (pp. 47–57). New York, NY, USA: ACM Press.

Lejsek, H., Ásmundsson, F. H., Jónsson, B., & Amsaleg, L. (2008). *NV-tree: An Efficient Disk-Based Index for Approximate Search in Very Large High-Dimensional Collections* (Tech. Rep. No. RUTR-CS07001). Iceland: Reykjavík University. (Accepted to IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI))

Lejsek, H., Ásmundsson, F. H., Jónsson, B. T., & Amsaleg, L. (2006). Scalability of Local Image Descriptors: A Comparative Study. In *Proceedings of the 14th Annual ACM International Conference on Multimedia* (pp. 589–598). New York, NY, USA: ACM.

Lowe, D. G. (2004). Distinctive Image Features from Scale-Invariant Keypoints. *International Journal of Computer Vision*, *60*(2), 91-110.

Weber, R., Schek, H.-J., & Blott, S. (1998). A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces. In *Proceedings of the 24rd International Conference on Very Large Data Bases* (pp. 194–205). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

REYKJAVÍK UNIVERSITY
HÁSKÓLINN Í REYKJAVÍK

School of Computer Science
Reykjavík University
Kringlan 1, IS-103 Reykjavík, Iceland
Tel: +354 599 6200
Fax: +354 599 6301
http://www.ru.is