THE SPATIAL C-TREE:
AN INDEX METHOD WITH HIGH CONCURRENCY AND EFFICIENT RECOVERY

A DISSERTATION
SUBMITTED TO THE COLLEGE OF COMPUTER SCIENCE
OF NORTHEASTERN UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

by
Jing Shan
May 2009
Abstract

Indexes are used in database management systems as a mechanism to speed up data retrieval. A good index structure should ensure that the data access through it is efficient and transactional. For one dimensional data, the $B^+$-tree and its variants, such as the $B^{link}$-Tree and the II-tree, are used widely because of their good query performance and concurrency control and recovery. However, structures for multidimensional data, despite the intense research for more than two decades, have not been so successful. Very few of the many proposed structures are implemented in commercial databases. For those that have been used, users are confronted with either poor performance or low concurrency.

We proposed a new spatial access method called the Spatial C-Tree. Our approach extends the C-Tree framework, which provides efficient concurrency control and robust recovery mechanism. We have implemented the Spatial C-Tree and compared it with several R-Tree variants. Our performance results show that the Spatial C-Tree performs very well in terms of insertion and exact match queries with various data sets. Our range query performance in some cases is comparable to the R-Tree variants and in other cases are outperformed by the R-Tree variants. We have identified possible improvements over the range query performance. The potential improvement on the range query performance and the high concurrency guarantee make the Spatial C-Tree a promising access method for spatial data.
Acknowledgments

First and foremost, I want to express my sincere gratitude and appreciation to my advisor, Prof. Betty Salzberg, for her invaluable guidance and support during my graduate studies. Through her dedication to research and her particular focus on fundamental questions, she inspired her students to make an impact in their fields, and through her constant feedback, she made me a better communicator. I will forever be grateful for her help and belief in me, as she is a true role model for me.

I would like to thank other members in my thesis committee - Dr. David Lomet, Professor Rajmohan Rajaraman and Professor Donghui Zhang. Dr. Lomet provided me with insightful advice and helped guide my thesis work since the very beginning. This thesis would not be possible without him. I would like to thank Professor Rajaraman for serving on both my proposal and thesis committee and for his thoughtful comments. I would also like to thank Professor Zhang for all the helpful advice and for not only being a professor, but also a great friend.

Thanks go to my fellow students Xiaowei Sun, Rui Wang, Tian Xia, Panfeng Zhou, Huanmei Wu and friends Xiaqin Ma, Emine Yilmaz, Robbie Ye, Feng Zhu, Guolong Lin, Ye Wang, Da Lin and Jun Gong, as well as many others. They have been true friends and companions.

Last but not least, I would like to thank my parents for their infinite love, persistent encouragement, and unconditional support in every aspect of my life. My mother has served as my role model in her kindness to others, optimism and hard-working nature. My father has taught me how to pursue my goals with confidence and to face and overcome obstacles with courage. Their high expectations have always been and will be the motivation behind my academic and professional achievements. Mom and Dad, this is for you!
# Contents

1 Introduction 3
   1.1 Importance of Index Structures .......................... 3
   1.2 Motivations ................................................. 4
   1.3 Our Approach ............................................... 5
   1.4 Outline ...................................................... 5

2 Related Work 7
   2.1 One-dimensional Indexing ................................. 7
   2.2 Multidimensional Indexing ................................ 7
      2.2.1 Transformed space .................................... 8
      2.2.2 Original space ......................................... 10
   2.3 Concurrency Control and Recovery on Access Methods ................. 11
      2.3.1 Consistency Requirements on Index Structures .............. 11
      2.3.2 The $\Pi$-Tree ........................................... 13
      2.3.3 Concurrency Control and Recovery in the R-Tree Family ...... 14

3 The C-Tree Framework 21
   3.1 Search Spaces of the C-Tree Pages ........................ 21
   3.2 The Containment Tree ....................................... 24
      3.2.1 Definitions .............................................. 24
      3.2.2 Properties .............................................. 26
      3.2.3 Operations on a Containment Tree ......................... 30
      3.2.4 Searches in a Containment Tree .......................... 33
   3.3 Generalizations of Containment Trees ........................ 36
      3.3.1 Definitions .............................................. 36
<table>
<thead>
<tr>
<th>Chapter/Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 Conclusion and Future Work</td>
<td>145</td>
</tr>
<tr>
<td>6.1 Summary</td>
<td>145</td>
</tr>
<tr>
<td>6.2 Future Work</td>
<td>146</td>
</tr>
<tr>
<td>Bibliography</td>
<td>135</td>
</tr>
<tr>
<td>Appendices</td>
<td>153</td>
</tr>
<tr>
<td>A Number of Pages</td>
<td>155</td>
</tr>
<tr>
<td>B Areas of Data Pages of MG Data Set</td>
<td>159</td>
</tr>
<tr>
<td>C Intersection Query (Page Size = 2KB)</td>
<td>165</td>
</tr>
<tr>
<td>D Contained Query (Page Size = 2KB)</td>
<td>173</td>
</tr>
<tr>
<td>E Containing Query (Page Size = 2KB)</td>
<td>175</td>
</tr>
<tr>
<td>F Data Sets</td>
<td>177</td>
</tr>
<tr>
<td>G Intersection Query Performance Comparison</td>
<td>183</td>
</tr>
<tr>
<td>H Contained Query Performance Comparison</td>
<td>187</td>
</tr>
<tr>
<td>I Containing Query Performance Comparison</td>
<td>191</td>
</tr>
</tbody>
</table>
List of Figures

1.1 Spatial object example. (a) spatial objects (b) use MBRs to approximate spatial objects. 5

2.1 A B-Tree example. ........................................ 7
2.2 Transform one-dimensional spatial objects and query range to two dimensional space. $a_i$'s are objects and Q is the query range. ........................................ 9
2.3 An example of the hB* -Tree. ................................ 9
2.4 An example of the $R$-tree. ................................ 10
2.5 An example of phantom in the R-Tree. ...................... 11
2.6 An example that shows the insertion and search for the same object might have different paths in the R-Tree. ........................................ 12
2.7 An example of page split in a B-Tree. .......................... 13
2.8 An example of split and post in a II-Tree. After 5 is inserted, $N_1$ is split into $N_1$ and $N'_1$. 14
2.9 Insert $R_4$ in an R-Tree. $N_1$ is split into $N_1$ and $N'_1$. ................. 16
2.10 An example of the $R^{link}$-Tree. ............................. 16
2.11 Each page in GiST has a predicate list. $Q_1$ and $Q_2$ are query ranges. ............................. 18

3.1 Search space predicates. ........................................ 22
3.2 Declared space, delegated space and home space of the $B$-Tree pages. .................. 24
3.3 A containment tree. ........................................ 25
3.4 A figure to illustrate Property 2. ............................. 26
3.5 A figure to illustrate Property 3. ............................. 27
3.6 A figure to illustrate Property 4. ............................. 28
3.7 An example that shows how a new node $n$ can be added to a containment tree. .............. 31
3.8 An example node merge in a containment tree. .................. 34
3.9 Only the sets indicated in (a) are partial subtrees. .................. 37
3.10 Node $a_1$ in $r_2$ is a generalization node of subtree $R_1$ in (a). ............................. 38
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.11</td>
<td>Generalization and refinement subtree of containment trees.</td>
<td>39</td>
</tr>
<tr>
<td>3.12</td>
<td>A well-formed containment hierarchy.</td>
<td>40</td>
</tr>
<tr>
<td>3.13</td>
<td>A figure to illustrate Property 9.</td>
<td>41</td>
</tr>
<tr>
<td>3.14</td>
<td>Searches in a containment hierarchy.</td>
<td>44</td>
</tr>
<tr>
<td>3.15</td>
<td>Different ways to perform searches in a containment hierarchy.</td>
<td>45</td>
</tr>
<tr>
<td>3.16</td>
<td>Searches in a containment hierarchy.</td>
<td>48</td>
</tr>
<tr>
<td>3.17</td>
<td>Add a new node in the refinement subtree of $k$ (Node addition case 1).</td>
<td>50</td>
</tr>
<tr>
<td>3.18</td>
<td>Add a new generalization node $n$ (Node addition case 2).</td>
<td>50</td>
</tr>
<tr>
<td>3.19</td>
<td>Remove node $n$ from $T_i$ (Node removal case 1).</td>
<td>51</td>
</tr>
<tr>
<td>3.20</td>
<td>Remove node $n$ from $T_i$ (Node removal case 2).</td>
<td>52</td>
</tr>
<tr>
<td>3.21</td>
<td>An example of the C-Tree.</td>
<td>54</td>
</tr>
<tr>
<td>3.22</td>
<td>The containment trees of the C-Tree shown in Figure 3.21.</td>
<td>55</td>
</tr>
<tr>
<td>3.23</td>
<td>The containment hierarchy corresponding to the C-Tree in Figure 3.21.</td>
<td>55</td>
</tr>
<tr>
<td>3.24</td>
<td>Index terms in the C-Tree.</td>
<td>56</td>
</tr>
<tr>
<td>3.25</td>
<td>The index entry containment trees of the C-Tree in Figure 3.21.</td>
<td>56</td>
</tr>
<tr>
<td>3.26</td>
<td>Data pages in the B-Tree.</td>
<td>59</td>
</tr>
<tr>
<td>3.27</td>
<td>Data pages and the level 1 containment tree.</td>
<td>62</td>
</tr>
<tr>
<td>3.28</td>
<td>Index entries of the data pages shown in Figure 3.27.</td>
<td>62</td>
</tr>
<tr>
<td>3.29</td>
<td>An example of index page.</td>
<td>63</td>
</tr>
<tr>
<td>3.30</td>
<td>Index term refinement subtree in page $I_1$ is generalization of the level 1 containment tree.</td>
<td>63</td>
</tr>
<tr>
<td>3.31</td>
<td>An example of index entry posting.</td>
<td>64</td>
</tr>
<tr>
<td>3.32</td>
<td>Index page split.</td>
<td>64</td>
</tr>
<tr>
<td>3.33</td>
<td>A new root page $R$ is created after the split in Figure 3.32</td>
<td>65</td>
</tr>
<tr>
<td>3.34</td>
<td>Page consolidation in the C-Tree.</td>
<td>66</td>
</tr>
<tr>
<td>3.35</td>
<td>Index page consolidation.</td>
<td>66</td>
</tr>
<tr>
<td>4.1</td>
<td>An example of object space and search space.</td>
<td>74</td>
</tr>
<tr>
<td>4.2</td>
<td>Descriptor regions of simple spatial predicates.</td>
<td>75</td>
</tr>
<tr>
<td>4.3</td>
<td>Simple spatial containment tree.</td>
<td>76</td>
</tr>
<tr>
<td>4.4</td>
<td>$R_6$ can not be the descriptor region of a new node in the simple spatial containment tree in Figure 4.3.</td>
<td>77</td>
</tr>
<tr>
<td>4.5</td>
<td>Spatial search predicate.</td>
<td>78</td>
</tr>
<tr>
<td>4.6</td>
<td>A containment tree using spatial search predicate.</td>
<td>81</td>
</tr>
</tbody>
</table>
4.7 Descriptor regions of predecessor siblings. .......................... 83
4.8 A spatial containment tree. .............................................. 84
4.9 The full path from \( n_0 \) to \( n_5 \) and the minimum path from \( n_0 \) to \( n_5 \). .................................................. 87
4.10 The full path from \( n_0 \) to \( n_i \) ........................................... 88
4.11 A new node \( n_6 \) is added to the spatial containment tree. Node \( n_3 \) and \( n_4 \) are descended nodes. ................................................... 90
4.12 Node \( n_6 \) is added as a sibling node of \( n_3 \), \( n_4 \) and \( n_5 \). .................................................. 90
4.13 A spatial containment tree and its descriptor regions. ................. 91
4.14 Generalization of the spatial containment tree in Figure 4.13. .......... 91
4.15 A spatial C-Tree. .......................................................... 93
4.16 (a) Spatial layout of the descriptor regions of the data pages in Figure 4.15. (b) The level 1 containment tree of the Spatial C-Tree in Figure 4.15. ................... 94
4.17 Object \( O_1 \) and \( O_2 \) are in page \( P_2 \) and \( P_5 \) in Figure 4.15 respectively. .............................................. 94
4.18 Post index entry for the new data page \( C_1 \). ............................ 95
4.19 A new root page \( R_{new} \) is created due to the split of the existing root page \( R \). ......... 96
4.20 Page \( C \) is an index page in a Spatial C-Tree. Rectangle \( R_i \) is the descriptor region of node \( n_i \). .......................................................... 97
4.21 The minimum path from \( n_1 \) to \( n_6 \) is posted to the parent page \( P \). ................................. 97
4.22 Posting the minimum path from \( n_1 \) to \( n_8 \) to page \( P \). ......................... 97
4.23 Post the minimum path from \( n_1 \) to \( n_4 \) to the parent page \( P \). ......................... 98
4.24 Page \( P \) is the parent page of \( C \) in a Spatial C-Tree. .................. 98
4.25 Merge node \( n_3 \) to parent page \( P \). .................................. 100
4.26 Merge node \( n_6 \) to parent page \( P \). .................................. 101
4.27 Data page in a Spatial C-Tree. ......................................... 106
4.28 Data page split in a Spatial C-Tree. .................................. 106
4.29 Data page split in a Spatial C-Tree. .................................. 107
4.30 Index pages in a Spatial C-Tree. ....................................... 108
4.31 Index page split in a Spatial C-Tree. .................................. 109
4.32 Containment split of index pages in a Spatial C-Tree. .................. 109
4.33 Split data page by hyperplane. ........................................ 111
4.34 Optimized containment tree for hyperplane split. ..................... 111
4.35 The upper edge and the left edge of \( R_1 \) are “expandable” edges. .... 114
5.1 Influence of splitting parameter on the number of pages. ........................................ 122
5.2 Influence of splitting parameter on the number of pages. ........................................ 123
5.3 Influence of splitting parameter on the areas of data page regions. ............................... 124
5.4 The smallest 70% data pages in Figure 5.3 .............................................................. 125
5.5 Influence of splitting parameter on the areas of data page regions ............................... 126
5.6 Average areas of data pages in MG data set (2nd and 5th 10 percentile). Page size is 2KB. 127
5.7 Average areas of data pages in MG data set (9th and 10th 10 percentile). Page size is 2KB. 128
5.8 Average areas of the largest 10 percentile data pages in fractal data set. Page size is 2KB. 129
5.9 Average areas of the largest 10 percentile data pages in LB data set. Page size is 2KB. ...... 129
5.10 Intersection query result of MG data set. Query area is 0.01% of the object space. Page size is 2KB. Buffer size holds up to 500 pages .............................................................. 130
5.11 Intersection query result of LB data set. Query area is 0.01% of the object space. Page size is 2KB. .............................................................. 131
5.12 Intersection query result of LB data set. Query area is 0.01% of the object space. Page size is 2KB. .............................................................. 131
5.13 Intersection query result of MG data set. Query area is 10% of the object space. Page size is 2KB. .............................................................. 132
5.14 Containing query performance of MG data set. Query area is 0.01% of the object space. Page size is 2KB. .............................................................. 133
5.15 Contained query performance of MG data set. Query area is 0.01% of the object space. Page size is 2KB. .............................................................. 133
5.16 Total number of pages. .............................................................................................. 134
5.17 Comparison of data page areas. .................................................................................. 135
5.18 Performance of insertions. .......................................................................................... 136
5.19 Exact match query performance. .................................................................................. 137
5.20 Performance comparison (number of pages accessed) of intersection queries using the MG data set. .............................................................................................. 138
5.21 Comparison of data page areas. .................................................................................. 139
5.22 Comparison of data page areas. .................................................................................. 139
5.23 Performance comparison (number of pages accessed) of intersection queries using fractal data set. .............................................................................................. 140
5.24 Performance comparison (IOs) of intersection queries using fractal data set. .............. 141
5.25 Comparison of data page areas. .................................................................................. 141
List of Tables

5.1 A summary of data sets used in our experiments. .......................... 120
5.2 The first 10 values in Figure 5.3 ................................................. 125
5.3 Intersection query performance compared to the R*_{NR}-Tree. ............. 138
5.4 Contained query performance compared to the R*_{NR}-Tree. ................. 144

......
Chapter 1

Introduction

This thesis proposes a new spatial access method called the Spatial C-Tree. This work is motivated by the high demand for spatial indexing from many new modern applications and the unsatisfactory performance of existing methods. In this chapter, we first review the importance of index structures in general. We then introduce our research motivation by discussing the difficulty in spatial indexing and analyzing problems with existing methods. In Section 1.3, we present a high-level overview of our approach. Finally, we outline the rest of the dissertation.

1.1 Importance of Index Structures

Indexes are an important building block of database management systems. Databases store a large amount of data in secondary storage (usually disk). A straightforward way to retrieve data is to scan the whole table and check each record against the query predicates. However, due to mechanical delays, IO operations are very slow, compared to CPU instructions ($10^{-3}$s vs $10^{-9}$s), hence they become a bottleneck. Access methods are designed as auxiliary structures to reduce disk accesses. While CPU speed, memory size, and disk capacity doubles every 18 months, the disk positioning rate doubles every ten years [3]! Thus, no matter how fast the technology development is, as long as the gap between CPU speed and disk speed continues to grow exponentially, disk access will remain a bottleneck for operations in database systems. Hence, index structures that optimize IOs are a crucial part of DBMS research.

Index structures are used not only to guarantee efficient data access, but to ensure high concurrency. If data records are accessed through a table scan, locks have to be placed on tables and data pages to avoid conflicting concurrent data accesses. Such large locking granules may
cause unnecessary blocking, hence low concurrency. Instead, if records are accessed through an index, such unnecessary blocking can often be avoided. Transactions conflict only if they access the same pages and locks on index pages are usually of shorter duration than data page locks, therefore higher concurrency.

1.2 Motivations

Classical database applications deal with one-dimensional data. The keys of these data can be numerical (e.g. salary), or alphabetical (e.g. last name). Current DBMS’s are well suited for such linearly ordered data.

The emergence of new modern applications, such as GIS (Geographic Information Systems), mapping, CAD, VLSI design, urban planning, transportation planning, resource management, geomarketing, archaeology and environmental modeling \cite{5}, require database systems to deal with spatial data.

The structure of spatial data is more complicated than traditional one-dimensional data. A spatial object usually has non-zero extent and a spatial location. One common technique is to use a Minimum Bounding Rectangle (MBR) to describe the extent and location of a spatial object. An example of spatial objects and their MBRs is illustrated in Figure 1.1. Figure 1.1(a) shows spatial objects with irregular shape. Figure 1.1(b) shows the MBRs of these objects. MBRs like these are normally used as the primary keys of spatial data.

Traditional index methods, such as the B-Tree, do not work well for spatial data since such methods cluster data by one dimension only while rectangles are multidimensional. Figure 1.1(c) and (d) describes an example that shows such inadequacy. Assume a data page can store 2 objects. If objects are clustered according to smaller $x$ coordinates of the rectangles, for example, object $O_1$ and $O_2$ will be considered closer than object $O_1$ and $O_4$. $O_1$ and $O_2$ would be in one page while $O_4$ would be in another page. Suppose we wish to find all objects contained in the rectangle labeled $R$ in Figure 1.1(c). This query needs to access two disk pages. As another example, if larger $y$ coordinates of the rectangles are used, object $O_2$ and $O_3$ would be in one page while object $O_1$ and $O_4$ would be in another page. Query $Q$ pictured in Figure 1.1(d) will be inefficient. However, in the optimal situation, where objects are clustered according to their spatial proximity, only one disk access is needed for query $R$ or query $Q$. Such multidimensional property of spatial data raises new challenges to conventional database systems.

Many index structures proposed in the past can be used to index spatial objects. Generally
1.3 Our Approach

We propose the Spatial C-Tree, a new access method for spatial data. It is an instance of the C-Tree [21, 17], which is an index framework that can be instantiated to index different data types. The C-Tree adopts the simple and efficient concurrency control and recovery mechanism of the Π-Tree and provides performance guarantees for insertions and exact match queries. The Spatial C-Tree instantiates the C-Tree by defining index terms, page splitting policies and an index term posting algorithm. By instantiating the C-Tree framework, the Spatial C-Tree achieves good concurrency control and recovery performance. Through the definition of splitting policies, the Spatial C-Tree achieves good query performance.

1.4 Outline

In Chapter 2, we first review some existing representative index methods for traditional linearly ordered data and spatial data. We then discuss concurrency control for these structures.

Chapter 3 introduces the C-Tree index framework. In this chapter, we define the general tree structure, search space for the C-Tree pages and abstract methods for the C-Tree. Our Spatial
C-Tree realizes this framework.

The Spatial C-Tree structure is introduced in Chapter 4. In this chapter, we not only define the Spatial C-Tree in general, but also give a detailed implementation of a Spatial C-Tree instance.

Chapter 5 tests the Spatial C-Tree with several representative data sets. We also compare the query results with three R-Tree variants. An optimization for the range intersection query is also presented in this chapter. Finally, in Chapter 6, we summarize this thesis work and outline some directions for future research.
Chapter 2

Related Work

2.1 One-dimensional Indexing

The B-Tree and its variants [4] are widely used as an index method for one-dimensional data. In this proposal, the B-Tree refers to the “B+-Tree” unless specified otherwise. The B+-Tree is a height balanced multi-way tree with data located in the leaf pages, also called data pages. Figure 2.1 (1) shows a B-Tree example. New data entries are inserted into data pages. If there is no room in the data page, a new data page is allocated, half of the data is moved into the new page and information about the split boundary is placed in the parent page. This split process might be propagated to upper levels of the tree until a page with enough space is reached. An example of data page split is illustrated in Figure 2.1 (2). Deletion of entries reverses this process when data pages become too sparsely populated. Figure 2.1 (3) shows an example of deletion.

2.2 Multidimensional Indexing

In the past two decades, many structures have been proposed to index multidimensional data. An overview of multidimensional access methods can be found in [8]. Existing access methods for

Figure 2.1: A B-Tree example.
multidimensional data can be divided into two categories: point access methods (PAM) and spatial access methods (SAM). Point access methods index multidimensional point data, i.e. objects without extent. Spatial access methods index multidimensional objects with non-zero extent, called spatial objects. Such objects are usually approximated by a minimum bounding rectangle (MBR), which is the smallest hyper-rectangle that covers the objects. Queries first find the qualified MBRs and then a refinement step eliminates false positive objects that do not intersect the query range although their MBRs do. Point access methods are a special case of spatial access methods.

All multidimensional indexing methods divide space into subspaces small enough that objects in that subspace can fit in one data page. Also, they all aim at preserving spatial proximity in pages as much as possible. In spite of these similarities, these methods work in quite different ways and demonstrate significant performance variability when used to answer queries.

Spatial access methods are more complicated than point access methods. The extent of spatial objects tend to cause overlaps among sibling pages of an index structure. Such overlaps are unavoidable as long as the indexed objects have extent.

In this thesis, I focus on spatial object indexing. Spatial access methods can be divided into two groups, in transformed space or in original space.

### 2.2.1 Transformed space

A spatial object represented by its MBR in a d-dimensional space can be transformed to a point data in a 2d-dimensional space. For example, a 2D rectangle $< [1, 1], [2, 2] >$, where $[1, 1]$ is its lower left corner and $[2, 2]$ is the upper right corner, can be mapped to point $[1, 1, 2, 2]$ in a 4D space. Such a transformed 2d-dimensional point can then be indexed by a multidimensional point access method. The advantage of this approach is that overlapping can be avoided.

Figure 2.2 shows an example that transforms one-dimensional hyper-rectangle objects, i.e. intervals, into two dimensional points\(^1\). $Q$ is a one-dimensional query range. Its corresponding intersection query range in 2D space is shown as the shadowed area in Figure 2.2(2). Query $Q$ looks for all point data in this 2-dimensional space whose $x_{low} \geq 5$ and $x_{high} \leq 10$. Note that the transformed query range is infinite along the $x_{high}$ axis. All transformed 2D points are above the main diagonal $x_{high} = x_{low}$ [8].

---

\(^1\)For the ease of visibility, we use a transformation from 1D to 2D. Spaces with dimension higher than 3 are hard to draw.
2.2. MULTIDIMENSIONAL INDEXING

Figure 2.2: Transform one-dimensional spatial objects and query range to two dimensional space. \(a_i\)'s are objects and \(Q\) is the query range.

The hB\(\pi\)-Tree  Point access methods (PAMs) can be used in transformed space. An example of a PAM is the hB\(\pi\)-Tree [6]. The hB\(\pi\)-Tree combines together the B-Tree, for a height balanced structure, the kd-Tree, for fast internal node retrieval, and the \(\Pi\)-Tree, for concurrency control and recovery. It is a height balanced, hierarchical tree-like structure.

An example of the hB\(\pi\)-Tree is shown in Figure 2.3 (1). The root, \(R\), of the hB\(\pi\)-Tree is responsible for the entire search space. The contents of \(R\) are organized by a kd-Tree as in Figure 2.3 (2). This kd-Tree describes the next level of the hB\(\pi\)-Tree, which includes three data pages \(N_1\), \(N_2\) and \(N_3\). Some kd-Tree nodes have decorations. They are pointers to child hB\(\pi\)-Tree pages. In search, we always visit the page indicated by the last decoration on the search path. The kd-Tree in Figure 2.3 indicates that \(N_1\) is responsible for the two dimensional space where \(y > y_1\), \(N_2\) is responsible for those objects \((x_i, y_i)\) where \(x_i \leq x_1\) and \(y_i \leq y_1\) and \(N_3\) is responsible for the space \(x > x_1\) and \(y \leq y_1\).

In index pages, searches are directed by the kd-trees located in those pages. One compares the query predicate with the kd-tree nodes. In Figure 2.3, the search for object \((x_2, y_2)\) \((x_2 > x_1\) and \(y_2 < y_1)\) follows the path \((y_1\text{-left, } x_1\text{-right})\) and visits \(N_3\).

Figure 2.3: An example of the hB\(\pi\)-Tree.
2.2.2 Original space

Many spatial access methods are in the original space that the spatial objects physically locate. If the indexed data are d-dimensional objects, these access methods will use this d-dimension. Most methods in this class [13, 1, 25] are based on the R-Tree [10]. The R-Tree is a balanced tree structure. Methods in the R-tree family all adopt the “object bounding” idea. The MBR (minimum bounding rectangle) of a leaf page is the smallest rectangle that contains the MBRs of each object in that leaf and the MBR of an index node is the smallest rectangle that includes the MBRs of its children. Figure 2.4 is an example of the R-tree. R is the root page. N_i’s are data pages. The MBR of each data page is the smallest rectangle that contains the spatial objects in that page. The MBR of page R is the smallest rectangle that contains the MBRs of its child pages, which are N_1, N_2 and N_3. The MBRs of pages at the same level in the R-Tree may overlap e.g. the MBR of N_1 and the MBR of N_2 have a overlapping region.

A range query that searches for objects intersecting the query range starts at the root page of the R-Tree. In an index page, all entries are checked against the query predicate. When a query range intersects more than one MBR, multiple paths need to be traversed. This could worsen the query performance. For example, in Figure 2.4 (2), although the query range Q intersects an object in N_2 only, both N_1 and N_2 are visited. This situation deteriorates as the size of the spatial data gets larger and more overlap occurs. The query performance of the R-Tree relies highly on the data distribution.

Figure 2.4: An example of the R-tree.

An insertion in the R-Tree, in contrast to searches, follows only a single path from the root page to a data page. If the new object is not contained in the MBR of any page at one level of the R-Tree, an optimal page will be chosen using some criterion such as minimal area expansion of the MBR. This is repeated until we reach a data page, where the new object is inserted. If the MBR of that data page does not contain the new object, we will expand the MBR. Such an MBR adjustment may be propagated upwards several levels. Figure 2.4 (3) is an insertion example.
2.3. CONCURRENCY CONTROL AND RECOVERY ON ACCESS METHODS

Object $a$ is not spatially contained in any leaf page. If we use the “smallest area enlargement” as a criterion to choose the desired data page, $a$ will be inserted in $N_3$. The MBR of $N_3$ needs to be expanded to include $a$ and the MBR of $R$ should be expanded as well to cover the new MBR of $N_3$. Such MBR expansions along the insertion path complicate the concurrency control and is the Achilles heel of the R-tree family. Details will be discussed later in Section 2.3.3.

2.3 Concurrency Control and Recovery on Access Methods

A general purpose DBMS should allow multiple users to proceed simultaneously. Data should be protected from system failures. After a system failure, it must be possible to recover to a consistent state.

2.3.1 Consistency Requirements on Index Structures

Index structures should make sure that each transaction sees a consistent state of the database, which includes data consistency and the consistency of index structures themselves.

**Data Consistency** The “phantom” is a typical violation of data consistency. **Phantoms** are those records that either appear or disappear from the search ranges of concurrent transactions [9].

Figure 2.5 is an example of a phantom. A transaction $T_1$ performs a range query $Q$ in the R-Tree. It looks for all objects that intersect $Q$. At this point, there is only one object qualified. Before $T_1$ commits, another transaction $T_2$ inserts an object $a$, which intersects query range $Q$. Object $a$ is called phantom. If $T_1$ performs the same search again, it will see two objects, which is a different set from the previous time. $T_1$ does not have a consistent view of the database.

![Figure 2.5: An example of phantom in the R-Tree.](image)

Locks on data items or data pages are not enough to prevent phantoms. The purpose of locks is to raise a conflict when two transactions access data in an incompatible way. Locks on existing records do not conflict with locks on phantoms. Figure 2.5 shows such an example. In index
structures like the R-Tree, where the insertion path and search path for the same object might be different and the inserter and searcher could access different data pages, locks on data pages do not help, either. For example, in Figure 2.6, assume there are two inserters and one searcher. Transaction $T_1$ inserts object $a$, $T_2$ inserts object $b$ and transaction $T_3$ searches for object $a$. Assume $T_1$ starts first. After $T_1$ visits page $R$, $T_2$ starts and finishes before $T_1$ visits page $N_2$. The insertion of $b$ enlarges the MBR of page $N_1$ and the new MBR happens to contain $a$. Now assume $T_3$ starts the search at this point (after $T_2$ is finished and before $T_1$ visits $N_2$). The search path of $T_3$ is $R$ and $N_1$ while the insertion path of the same object by $T_1$ is $R$ and $N_2$. Locks on $N_1$ and $N_2$ do not raise any conflict between $T_1$ and $T_3$. Therefore, a phantom prevention mechanism needs to be provided in the index level.

Figure 2.6: An example that shows the insertion and search for the same object might have different paths in the R-Tree.

**Index Integrity** In addition to consistency of data items, index structures themselves need to be integral. First, changes made by user transactions should be reflected in indexes once the user transactions commit. If a record is inserted by a user transaction $T_1$, other transactions that arrive after $T_1$ commits should be able to access this record through indexes. Second, structural changes of indexes should be done in an atomic way such that concurrently running transactions have a consistent view of the database. An example is shown in Figure 2.7. Assume a transaction $T_1$ searches for records with key larger than 11 and $T_2$ inserts a record 9. Ideally, these two transactions do not conflict and $T_1$ should find 15. However, in Figure 2.7 (2), if the insertion of $T_2$ triggers a page split, but before the new page is posted to its parent page, $T_1$ will find 0 qualified record. This is not correct. Such an inconsistent scenario should be prevented. Figure 2.7 shows a one-dimensional case. The problem becomes more challenging for multi-dimensional space, as shown later in Section 2.3.3.

In conclusion, index structures should meet two requirements regarding concurrency and re-
2.3. CONCURRENCY CONTROL AND RECOVERY ON ACCESS METHODS

Figure 2.7: An example of page split in a B-Tree.

covery. One is to prevent phantoms. The other is to maintain its own structure consistency, at normal run time and after recovery from crashes.

In the rest of this section, I will briefly review two types of concurrency and recovery mechanisms in multidimensional index methods. One is the Π-Tree structure. The other is the R-Tree family concurrency.

2.3.2 The Π-Tree

The Π-Tree [19, 20], based on the B\textsuperscript{link}-Tree, mainly addresses the problem of maintaining index tree structure consistency. It can be used in both one-dimensional methods (e.g. the B-Tree) and multidimensional methods, such as the hB\textsuperscript{π}-Tree. Phantom prevention in the Π-Tree is resolved through locks inside data pages (given an object, insertion and exact match query of that object follow the same path in the Π-Tree. The index structure does not expose any phantom hazard. So phantom preventions inside data pages only are enough for the Π-Tree. For example, for one-dimensional data, incremental key range locking on data items can be used). The Π-Tree achieves high concurrency by placing short latches on index pages. The Π-Tree structure changes are separated from the user transactions that trigger such changes and are decomposed into a sequence of short-term recoverable atomic actions so that index pages only need to be latched for a short period of time. Each atomic action keeps the tree in a well-formed state.

The Π-Tree is a balanced tree-like structure like the B-Tree. Each node in the Π-Tree contains index entries pointing to child pages and side pointers pointing to sibling nodes. A Π-Tree node can have multiple parents and contain multiple side pointers.

Side pointers are created during page splits and indicate a “containment” relationship among siblings. When a page \( N \) is split into \( N \) and \( N' \), part of \( N \)’s contents are moved to \( N' \). \( N \) is \textit{directly responsible} for part of the key space and the other part is \textit{delegated} to \( N' \). A side pointer pointing to \( N' \) is placed in \( N \). The side pointer contains both the key space and the address of \( N' \).
$N$ is called the containing page and $N'$ is called the contained page. The contents of a Π-Tree index node are organized to indicate the containment ordering among its children pages. The side pointers in each level of the Π-Tree form a partial containment order.

The Π-Tree has three basic operations: search, insertion and deletion. Exact match query (find records with the given key) follows a unique path from the root node of a Π-Tree to a leaf node. In each level of the Π-Tree, more than one page might be accessed. But only one of them is directly responsible for a given key. If an index page is directly responsible for the query predicate, the search will go down to the next level of the Π-tree by following one of its child entries. Otherwise, a side pointer with key space containing the query predicate will be followed.

Insertion also follows a single path. If a data page, say $N_1$, overflows after the insertion, $N_1$ will be split into $N_1$ and $N'_1$, as in Figure 2.8. A side pointer of $N'_1$ will be placed in $N_1$. Unlike in a regular B-Tree, the information about $N'_1$ is not posted to $N_1$'s parent $P$ immediately. The insertion transaction commits after $N_1$ is split. Posting to the parent is performed separately. A posting is scheduled whenever a side pointer traversal happens. The posting from level $i$ to level $i + 1$ is performed in an atomic and recoverable fashion. If $N_1$ has more than one parent, several post operations will be scheduled. Deletions, node consolidations and recovery of the Π-Tree are not discussed here. Details can be found in [20, 18]. In the Π-Tree, at most three index pages are locked at the same time, which gives the Π-Tree higher concurrency than methods reviewed in the next section.

![Figure 2.8: An example of split and post in a Π-Tree. After 5 is inserted, $N_1$ is split into $N_1$ and $N'_1$.](image)

### 2.3.3 Concurrency Control and Recovery in the R-Tree Family

The first paper that considers concurrency control of the R-Tree was written by Ng and Kameda [23]. It proposes three methods to handle the concurrency. In the first one, each transaction uses one lock on the root node. This obviously has no concurrency when an inserter is present. The second one defers structure change such as split or consolidation till no transaction accesses the tree. A separate transaction is used to perform such changes. And the third method locks a safe
node (a safe node is the index page whose child page on the insertion path has enough space and this page will not be split due to the insertion) and its subtree for insertions. These methods have relatively limited performance and concurrency.

The R_{link}-Tree The R_{link}-Tree [15] is an extension of the R-Tree. It is used in most existing R-Tree concurrency algorithms to maintain the index structural integrity.

The R_{link}-Tree uses the side pointer technique introduced in the B_{link}-Tree. Each page has a side pointer pointing to its right sibling page. When a page, say $N$, is split into $N$ and $N'$, the side pointer in $N$ is moved to $N'$ and a new side pointer pointing to $N'$ is placed in the new page $N$. The key idea behind side pointers is that, for those splits that haven’t been posted, the contents of new pages can be found by following the right links.

The side pointer technique in the R_{link}-Tree is more complicated than in the B_{link}-Tree. Any method that uses the right link concept should answer two questions: (1) whether a split has happened and, if so, (2) when to stop following the right link. In the B_{link}-Tree, it is the semantics of the key space that is used to answer the above two questions. If the key range of a page does not cover the query predicate, the page has been split. The search should follow right links until the search predicate is fully covered by the key spaces of pages visited. For example, in Figure 2.8 (2). Transaction $T_1$ inserts 5 in $N_1$. $N_1$ is split into $N_1$ and $N'_1$. Before $N'_1$ is posted to the parent page. $T_2$ searches for 7. When $T_2$ sees that page $N_1$ has a key space $(-\infty, 3)$, which does not contain the search predicate 7, it follows the right link and visits $N'_1$. $T_2$ stops following right links at $N'_1$ because $N'_1$’s key space is $(3, 7]$, which covers the search predicate. To summarize, pages in each level of the B-Tree family partition the entire key space. There is a unique page in each level whose key space contains the given key. Side pointers in the B_{link}-Tree always point to pages with higher key spaces. In the R_{link}-Tree, however, such a fact does not hold. Right links in the R_{link}-Tree have no spatial meanings. The key spaces of sibling pages in the R_{link}-Tree may overlap. By looking at the key space of a page and the query range, one can not tell whether the page has been split or not. Figure 2.9 shows such an example. $T_1$ inserts $R_4$ and causes $N_1$ to split. Before the new page $N'_1$ is posted to its parent page. $T_2$ issues a search $Q$, which looks for objects intersecting $Q$, i.e. $R_3$ in this example. When $T_2$ visits $N_1$, it can not detect the fact that $N_1$ has been split, and therefore can not find $R_3$.

To solve the above problem, the R_{link}-Tree uses a logical sequence number (LSN) in each page. This LSN is also stored in the corresponding index entry. A global monotonically increasing LSN counter is used. A LSN in an index entry serves as a “time stamp”. It remembers the “time”
CHAPTER 2. RELATED WORK

Figure 2.9: Insert $R_4$ in an R-Tree. $N_2$ is split into $N_1$ and $N'_1$.

when the last split of the corresponding page was posted. Figure 2.10 is an example of the R^link-Tree. When a page, say $N_1$ with LSN 1, is split into $N_1$ and $N'_1$, the old LSN of $N_1$ is assigned to $N_1$ and a new larger LSN 4 is assigned to $N_1$. The new LSN (4) in $N_1$ means that the latest split of $N_1$ happens at “time (LSN) 4”. The LSN of $N_1$’s index entry in $P$ is 1, which means the latest posting of $N_1$’s split happens at “time (LSN) 1”. When a searcher sees such inconsistent LSNs, it knows the $N_1$ has been split since “time 1”.

Figure 2.10: An example of the R^link-Tree.

Search operations, which look for all objects intersecting with the given query range, are performed in a similar depth first style as in regular R-Trees. An auxiliary stack is used. The root entry is pushed to a stack if it intersects the given query range. Each time an entry $I$ in the form of $(MBR, P, LSN)$, is popped from the stack, the corresponding page, $P$, is accessed. If the entries in $P$ are objects, they are reported. Otherwise, the qualified entries are pushed to the stack. This procedure is repeated until the stack is empty. If the LSN in the index entry popped from the stack is smaller than the LSN stored in the corresponding page, the searcher can detect that the page has been split since the searcher last visits the parent of that page. For example, in the right part of Figure 2.10, the index entry of $N_1$ has LSN 1 while the page $N_1$ has LSN 4. A searcher that looks for $R_3$, visits $N_1$, detects a larger LSN and keeps following the right link until a page with LSN 1 is reached, which is page $N'_1$ in this example. So the searcher sees the correct result, $R_3$.

When an insertion causes a node $N$ to overflow, $N$ is split into $N$ and $N'$. Information about $N'$ is posted to $N$’s parent although it may not be the real geometrically optimal parent. Such a
requirement deteriorates query performance in the R^{link}-Tree.

Predicate locks are used in the R^{link}-Tree to prevent phantoms. However, as pointed out in [9], predicate locking requires more resources and is expensive to maintain. The GiST concurrency method [16] adopts the R^{link}-Tree idea and proposes a modified predicate locking algorithm with higher concurrency.

**The GiST Concurrency** The currency control and recovery mechanism developed in [16] is proposed for the GiST, a generalized search tree [11].

GiST is a balanced tree structure with data on leaf pages. The index pages in GiST can have overlapping key spaces while objects are partitioned in data pages. The R-Tree and the B-Tree belong to GiST while the hB^{π}-Tree is not. In the following, I will use the R-Tree to illustrate the GiST concurrency idea since the R-Tree is an instance of the GiST.

To maintain index structural integrity, the GiST uses a similar strategy as the R^{link}-Tree. Each page has one right link pointing to its right siblings. Each page has a Node Sequence Number (NSN), which is monotonically increased by a global counter. Index entries in GiST pages, unlike those in the R^{link}-Tree, do not store such NSNs. Instead, when a searcher pushes index entries to its auxiliary stack, it pushes together with each entry the global NSN value at that time. The GiST guarantees that, if a searcher is going to visit \( N \) and \( N \) has been split into \( N \) and \( N' \), the searcher can either detect the split from \( N \)'s NSN or it has seen the information about \( N' \) in the parent page. This is achieved by combining the split (and the increase of the global NSN counter) and the post of \( N' \)'s information into an atomic action.

Here is how it works. Normally, if there is no split, the global NSN value that a searcher remembers with an index entry should be larger than or equal to the NSN in the corresponding page. When the increase of the global NSN counter and the post of information about the new page are put in one atomic action, a searcher either visits page \( N \)'s parent before the posting (i.e. before the global counter increase, hence remembers a smaller NSN with \( N \)'s index entry) or after the posting (hence, see the information about \( N' \)). If the searcher visits \( N \)'s parent before the posting, the NSN it remembers with the index entry of \( N \) in its stack is smaller than the NSN that the searcher sees later in page \( N \). So the searcher can detect the split. If the searcher visits \( N \)'s parent after the posting, it is able to see the information about \( N' \). This is a regular search as in the R-Tree.

The atomic action mentioned above is performed as follows. Before \( N \) is split, a X-latch needs to placed on \( N \)'s parent page \( P \) [9]. After the X-latch is granted, \( N \) is split. The global NSN
is increased and assigned to $N$. The old NSN of $N$ is assigned to the new page $N'$. After the information about $N'$ is posted to $P$, the X-latch on $P$ can be released.

To prevent phantoms, GiST maintains a list of predicates (a main memory structure) for each page. The predicate lists of the data pages are used later by inserters to detect conflicts. The predicate lists are created and maintained as follows. Whenever a query reads a page, the predicate of that query is attached to the page’s predicate list. Figure 2.11 is an example. $Q_1$ and $Q_2$ are query ranges. The intersection query $Q_1$ reads pages $N$, $P_1$, $P_2$, $R_1$ and $R_3$. $Q_2$ visits pages $N$, $P_2$, $R_3$ and $R_4$. There are two times when a page’s predicate list is changed. One is when the page’s MBR is updated, the other is when a new page is created through a split. If $P'$ is split off from $P$, $P'$ needs to copy from $P$ all the predicates that are consistent with the MBR of $P'$. If a node $P$’s MBR is enlarged, $P$ needs to copy all the “qualified” predicates of its parent node and “percolate” these new predicates to $P$’s children if needed.

The GiST requires the above changes to be reflected in the index before the new record is actually inserted. After (1) the MBRs are expanded to cover the new object, (2) the index structure is changed if there is any split and (3) the predicate lists along the insertion path are updated if necessary, an inserter checks the predicate list of the destination leaf page, which contains all search predicates that could conflict with the inserter. If there is any conflict, the inserter blocks itself until all conflicting transactions commit.

To summarize, this method requires an inserter to X-latch index pages, possibly at several upper levels, and to update index page MBRs inside the user transaction. In the worst case, when the data page split propagates up till the root page of the tree, the whole insertion path need to be X-latched at the same time, to adjust index entry MBRs as well as propagate splits along the path. And this worst case happens quite often in the R-Tree family during dynamic insertions when the new data arrives which is not in any MBR.

![Figure 2.11: Each page in GiST has a predicate list. $Q_1$ and $Q_2$ are query ranges.](image)
Conclusion of the R-tree Family Concurrency  
In the R-Tree family, the index structure changes can not be separated from the user transactions that trigger such changes. The index entries along insertion or deletion paths need to be updated to reflect the insertion or deletion of a record. For insertions, such updates have to be finished before the user transaction commits. Otherwise, other transactions can not see the insertion during the time after the insertion commits and before the updates are finished. The R-Tree uses MBRs to describe key space of each page. Each level of the R-tree does not cover the entire key space. The “entire key space” is the space that an object could ever be inserted. It includes the space that has objects present and the space that might be occupied by objects inserted in the future. In the R-Tree, MBRs of pages are determined only by the objects present. If a new key is not in the space reflected by existing MBRs, other search transactions can not see the new record. For deletions, the MBRs along the path should be updated immediately to minimize overlaps of MBRs and to improve performance of other queries.

When index structure changes are performed inside user transactions, the user transactions get longer. The locks on index nodes by long transactions decrease concurrency dramatically. Low concurrency is inherent in the R-Tree structure.

Extension of the R-Tree concurrency to the R*-Tree  
The main difference between the R-Tree and the R*-Tree, other than their splitting policies, is the “forced reinsertion” in the R*-Tree. In the R*-Tree [1], if a node overflows, it is not split immediately. Part of the entries (usually 30%) from this node are reinserted into the tree. If the node is an index page, the reinserted entries are inserted back into the nodes at the same level as the original page. If the reinsertion causes a node to overflow, that node is split as a regular split. The forced reinsertion improves the query performance as well as storage utilization in the R*-Tree. Such reinsertion is non-trivial when it comes to concurrency control. None of the algorithms described above handles such reinsertion. It is not clear how the forced reinsertion can be done without sacrificing concurrency.

The R-Tree is more static compared to the R*-Tree, in the sense that the R-Tree involves fewer structure changes in the index, while the reinsertion of the R*-Tree needs a lot of structure modifications. As explained earlier, all structure changes in the R-Tree family need to be associated with the corresponding user transaction. Therefore, the R*-Tree requires longer and more locks on the index nodes. Hence its concurrency is lower.
Chapter 3

The C-Tree Framework

This chapter introduces the C-Tree framework, which extends the concurrency control and recovery mechanism from the Π-Tree [19, 20]. In this chapter, we first define the search spaces of the C-Tree pages in Section 3.1. We then define the containment tree, its properties and tree transformations in Section 3.2. In Section 3.3, we define the concept of containment hierarchies. Finally, in Section 3.4, we define the C-Tree using the concept of containment tree and containment hierarchy.

3.1 Search Spaces of the C-Tree Pages

There are several types of search space associated with both index and data pages in the C-Tree. In this section, we first describe the definitions of search spaces and their relations. After that, we introduce the three search spaces associated with a C-Tree page.

Search spaces are defined by predicates. An object is in a search space if it satisfies the predicate of the search space. For example, suppose objects are data records with (one-dimensional) keys which are integers. A predicate for search space $S$ might be “key $\leq 10$” meaning that a data record $D$ is in $S$ if the key of $D$ is smaller than or equal to 10. Given a search space $S$, $\text{predicate}(S)$ denotes the predicate defining $S$. The entire search space is defined by a true predicate.

For any two search spaces $S_1$ and $S_2$, we define the containment and intersection relations as follows.

**Definition 1 (Containment).** A search space $S_1$ contains a search space $S_2$ if any object that satisfies $\text{predicate}(S_2)$ also satisfies $\text{predicate}(S_1)$. $S_1$ is called a container of $S_2$. 

21
For example, if predicate($S_1$) is “less than or equal to 10” and predicate($S_2$) is “less than or equal to 5”, we say $S_1$ contains $S_2$ because any object with key less than or equal to 5 will also satisfy “less than or equal to 10”.

**Definition 2 (Intersection).** Two search spaces $S_1$ and $S_2$ intersect if there could exist an object that satisfies both predicate($S_1$) and predicate($S_2$).

For example, let predicate($S_1$) to be “less than or equal to 10”, predicate($S_2$) to be “less than or equal to 5” and predicate($S_3$) to be “greater than 5”, as shown in Figure 3.1. Again, we assume the keys of objects are integers. According to definition 2, predicate($S_1$) intersects predicate($S_2$) while predicate($S_2$) and predicate($S_3$) do not intersect. This is because there is no integer that can be “less than or equal to 5” and “greater than 5” at the same time, hence there does not exist an object that can satisfy both predicate($S_2$) and predicate($S_3$).

<table>
<thead>
<tr>
<th>predicate($S_1$): $\leq 10$</th>
<th>predicate($S_3$): $&gt; 5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>predicate($S_2$): $\leq 5$</td>
<td>predicate($S_4$): $&gt; 5$ and $\leq 10$</td>
</tr>
</tbody>
</table>

Figure 3.1: Search space predicates.

For a set of search spaces, we define the following two concepts.

**Definition 3 (Well-Nested Search Spaces).** A set of search spaces is **well-nested** if for any two search spaces $S_i$ and $S_j$ either (1) $S_i$ and $S_j$ do not intersect or (2) one contains the other. In other words, if $S_i$ and $S_j$ intersect, either $S_i$ contains $S_j$ or $S_j$ contains $S_i$.

In the running example in Figure 3.1, if we add predicate($S_4$) as “greater than 5 and less than or equal to 10”, then $S_1$, $S_2$ and $S_4$ form a well-nested set while $S_1$, $S_2$ and $S_3$ do not. Any two spaces in the first set either “not intersect” (e.g. $S_2$ and $S_4$) or when they do intersect, one is larger than the other (e.g. $S_1$ and $S_4$). The spaces in the second set do not have such property. $S_1$ and $S_3$ intersect. But $S_1$ is not larger than $S_3$ and $S_3$ is not larger than $S_1$ either.

**Definition 4 (Union).** The **union** of search spaces $S_1$, $S_2$... $S_i$ ($i \geq 1$) is a search space defined by “predicate($S_1$) $\lor$ predicate($S_2$) $\lor$ ... $\lor$ predicate($S_i$)”. That is, an object is in the union if it is in any of the underlying search spaces.

The above definitions are for search spaces in general. There are three types of search spaces used for C-Tree pages, **declared (search) space**, **delegated (search) space** and **home (search)**
space. Given a page \( P \), its three search spaces can be written as \( \text{declared}(P) \), \( \text{delegated}(P) \) and \( \text{home}(P) \) respectively. A page is assigned its declared space when it is allocated and it retains that space until it is deallocated. When a page splits, it delegates some of its space. The remaining space is its home space. An object \( o \) is stored in a page \( P \) iff \( o \) satisfies the predicate of \( P \)’s home space.

**Definition 5 (Declared Space).** The declared space of a page \( P \) is defined by a predicate assigned to \( P \) when \( P \) is first created. \( \text{Declared}(P) \) is static. It does not change over time.

**Definition 6 (Delegated Space).** The delegated space of a page \( P \) is a search space contained in \( \text{declared}(P) \), but has been assigned to other pages. It is the union of its initial value and the declared spaces of the pages split off from \( P \).

When \( P \) is first created, it is assigned an initial value of \( \text{delegated}(P) \). When \( P \) splits, the delegated space of \( P \) becomes larger. The new value of \( \text{delegated}(P) \) after the split is the union of the \( \text{delegated}(P) \) before the split and the declared space of the new page. The delegated space of \( P \) is a proper subset of the declared space of \( P \).

**Definition 7 (Home Space).** The predicate of \( \text{home}(P) \) is defined as “satisfies the predicate of \( \text{declared}(P) \) and does not satisfy the predicate of \( \text{delegated}(P) \).”

\( \text{Home}(P) \) describes the data that can be stored in \( P \). That is, if an object \( o \) is stored in \( P \), \( o \) satisfies \( \text{predicate}(\text{home}(P)) \). For example, B-Tree pages can be described using these terminologies. When a B-Tree page \( P \) is first allocated, its declared space has a predicate which is a key range \( R \). Any record which has a key satisfying \( R \) is in \( P \)’s declared space. Suppose \( R \) is key range \([5, 15)\), meaning any record with key \( k \) satisfying \( 5 \leq k < 15 \) is in the declared space of \( P \). When \( P \) is first created, its delegated space is empty. Its home space is the same as its declared space, as shown in Figure 3.2. Assume \( P \) splits into \( P \) and \( P' \) later. Key range \([10, 15)\) is delegated to the new page \( P' \). Then the declared space of \( P' \) has predicate “key in \([10, 15)\)”, which is also the delegated space of \( P \) after the split. The delegated space of \( P' \) is empty. The home space of \( P' \) is the same as its declared space. The home space of \( P \) would be described by predicate “key satisfies \( \text{predicate}(\text{declared}(P)) \), but does not satisfy \( \text{predicate}(\text{delegated}(P)) \)”, i.e. the home space of \( P \) is described by the predicate “keys in \([5, 10)\)”. Figure 3.2(b) illustrates the scenario after the split.

For convenience, when an entry \( I \) points to a page \( P \), we say the search space of \( I \) is the same as the search space of \( P \). Here \( I \) can be either an index entry or a side pointer entry. We will define
index entry and side pointer formally later when we define the C-Tree in Section 3.4.

With the search space definitions in place, we are now ready to define the concept of containment trees in the following sections. We will first look at how to form a containment tree in an abstract level. After that, we will show how C-Tree pages relate to containment trees.

3.2 The Containment Tree

3.2.1 Definitions

A containment tree is a tree structure that represents the containment relationship of the C-Tree pages. Before we define a containment tree, we need to define “immediate containment”. Immediate containment is introduced through page splits. When a page \( P_1 \) is split off from an overflowing page \( P \), \( P \) becomes an immediate container of \( P_1 \).

**Definition 8 (Container/Contained Page).** If \( \text{declared}(P) \) contains \( \text{declared}(P_1) \), then we say that \( P \) is a \textbf{container page} of \( P_1 \) and \( P_1 \) is a \textbf{contained page} of \( P \).

**Definition 9 (Immediate Containment).** Given pages \( P \) and \( P_1 \), we say \( P \) \textbf{immediately contains} \( P_1 \) if (1) \( \text{declared}(P) \) contains \( \text{declared}(P_1) \) and (2) for any other data page \( P' \) whose declared space contains \( \text{declared}(P_1) \), \( \text{declared}(P') \) must contain \( \text{declared}(P) \).

**Definition 10 (Immediate Container/Contained Page).** If page \( P \) immediately contains page \( P_1 \), we say \( P \) is the \textbf{immediate container} of \( P_1 \) and \( P_1 \) is an \textbf{immediate contained} page of \( P \). When a page \( P' \) splits off from page \( P \), \( P \) becomes an immediate container of \( P' \).

Now we can define the containment tree.

**Definition 11 (Containment Tree).** A \textbf{containment tree} is a tree over a set of nodes which satisfy the following conditions:
3.2. THE CONTAINMENT TREE

1. A node in a containment tree corresponds to a page (either a data page or an index page). A node stores the predicate of the declared space of its corresponding page. When we refer to “the declared space of a node”, we mean the declared space of its corresponding page.

2. No two nodes in a containment tree can have the same declared space.

3. No node can have empty declared space.

4. Node $n_1$ is the parent node of $n_2$ iff the corresponding page of $n_1$ is the immediate container of the corresponding page of $n_2$.

5. If node $n$ and $n'$ have the same parent node, then $n$ and $n'$ are called sibling nodes. The intersection of the declared spaces of sibling nodes must be empty.

The corresponding pages of sibling nodes are called containment sibling pages.

**Definition 12 (Containment Sibling Pages).** Two pages $P_1$ and $P_2$ are called containment sibling pages if their corresponding nodes $n_1$ and $n_2$ are sibling nodes in a containment tree.

Figure 3.3 shows an example of a containment tree. The table on the left shows the page numbers and predicates of their declared spaces. On the right side is the containment tree corresponding to these pages. Node $n_1$ corresponds to page $P_1$. Page $P_0$ is the immediate container page of $P_1$, $P_2$ and $P_3$. $P_3$ is the immediate container page of $P_4$ and $P_5$. $P_4$ is the immediate container page of $P_6$. Node $n_1$, $n_2$ and $n_3$ are sibling nodes because they are all child nodes of $n_0$. Nodes $n_4$ and $n_5$ are also sibling nodes because they share the same parent node $n_3$.

<table>
<thead>
<tr>
<th>Page number</th>
<th>Predicate of declared space</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_0$</td>
<td>[0, 100)</td>
</tr>
<tr>
<td>$P_1$</td>
<td>[0, 20)</td>
</tr>
<tr>
<td>$P_2$</td>
<td>[90, 100)</td>
</tr>
<tr>
<td>$P_2'$</td>
<td>[60, 90)</td>
</tr>
<tr>
<td>$P_3$</td>
<td>[80, 90)</td>
</tr>
<tr>
<td>$P_4$</td>
<td>[60, 75)</td>
</tr>
<tr>
<td>$P_5$</td>
<td>[86, 90)</td>
</tr>
</tbody>
</table>

Figure 3.3: A containment tree.
3.2.2 Properties

Given the definitions discussed in the last section, we can derive the following six important properties. These properties guarantee the correctness of searches in a containment tree discussed later. We will use “delegated space of a node” to refer to the delegated space of its corresponding page and similarly “home space of a node” refers to the home space of its corresponding page.

Property 1. Declared space of any node in subtree rooted at \( n \) is contained in \( \text{declared}(n) \).

By the definition of containment tree (Definition 11), if a node \( n_1 \) is in the subtree rooted at \( n \), which means \( n_1 \) is a descendant of \( n \), declared(\( n \)) must contain declared(\( n_1 \)).

Property 2. For any two nodes in a containment tree, if none of them is an ancestor of the other, then their declared spaces do not intersect.

![Figure 3.4: A figure to illustrate Property 2.](image)

Assume \( n_i \) and \( n_j \) are two nodes in a containment tree such that \( n_i \) is not an ancestor of \( n_j \) and \( n_j \) is not an ancestor of \( n_i \). Let \( n \) be their common ancestor node whose declared space is the smallest one that contains declared(\( n_i \)) and declared(\( n_j \)). As shown in Figure 3.4, assume \( n_1 \) and \( n_2 \) are child nodes of \( n \) such that \( n_1 \) is a container of \( n_i \) and \( n_2 \) is a container of \( n_j \). According to the definition of containment tree (Definition 11), \( n_1 \) and \( n_2 \) are sibling nodes and declared(\( n_1 \)) \( \cap \) declared(\( n_2 \)) = \( \phi \). We also know declared(\( n_1 \)) contains declared(\( n_i \)); declared(\( n_2 \)) contains declared(\( n_j \)) (container definition). Therefore, declared(\( n_i \)) \( \cap \) declared(\( n_j \)) = \( \phi \).

Corollary 1. For any two nodes \( n_1 \) and \( n_2 \) in a containment tree, if declared(\( n_1 \)) contains declared(\( n_2 \)), then \( n_2 \) must be in the subtree rooted at \( n_1 \).

From the converse of Property 2, we know that one of \( n_1 \) and \( n_2 \) is an ancestor of the other. Since declared(\( n_1 \)) contains declared(\( n_2 \)), from the containment tree definition, \( n_1 \) is an ancestor of \( n_2 \). Thus, \( n_2 \) is in the subtree rooted at \( n_1 \).
Property 3. In a containment tree, the delegated space of a node \( n \) is the union of declared space of \( n \)'s child nodes, i.e. \( \text{delegated}(n) = \bigcup \text{declared}(n_i) \), where \( n_i \) is a child node of \( n \).

(1) We first show that \( \text{delegated}(n) \) contains the union of \( \text{declared}(n_i) \), i.e. \( \text{delegated}(n) \supseteq \bigcup \text{declared}(n_i) \). Given the fact that \( n \) is the parent node of \( n_i \), we know that \( P \) is an immediate container of \( P_i \) where \( P \) is the corresponding page of \( n \) and \( P_i \) is the corresponding page of \( n_i \). According to the definition of delegated space (Definition 6), \( \text{declared}(P_i) \) is part of \( \text{delegated}(P) \), i.e. \( \text{declared}(n_i) \subseteq \text{delegated}(n) \).

(2) We now show by contradiction that \( \text{delegated}(n) \) is a subset of \( \bigcup \text{declared}(n_i) \). Assume \( \text{delegated}(n) \) is not a subset of \( \bigcup \text{declared}(n_i) \). Then there must exist a search space \( S \) such that \( S \subseteq \text{delegated}(n) \) and \( S \not\subseteq \bigcup \text{declared}(n_i) \). Now let's consider the node whose declared space is the smallest one that contains \( S \). (By smallest, we mean there does not exist another node \( n' \) such that \( \text{declared}(n') \) contains \( S \) and \( \text{declared}(n) \) contains \( \text{declared}(n') \)). We call this node \( n_s \).

Node \( n_s \) must be in the subtree rooted at \( n \), as shown in Figure 3.5. This is because \( S \subseteq (\text{declared}(n) \cap \text{declared}(n_s)) \), from Property 2, one of these two nodes must be an ancestor of the other. Since we assume \( S \) is in \( \text{delegated}(n) \) and \( n_s \) is the smallest that contains \( S \), so \( n_s \) can not be an ancestor of \( n \). Therefore, \( n \) is an ancestor of \( n_s \).

Since \( n_s \) is in the subtree rooted at \( n \) and \( n_i \)'s are children of \( n \), \( \text{declared}(n_s) \subseteq \text{declared}(n_i) \). From the assumption that \( S \subseteq \text{declared}(n_s) \), \( S \subset \text{declared}(n_i) \), which contradicts to the assumption that \( S \not\subseteq \text{declared}(n_i) \). Thus, \( \text{delegated}(n) \) must be a subset of \( \bigcup \text{declared}(n_i) \).

![Figure 3.5: A figure to illustrate Property 3.](image)

From (1) and (2), we know that \( \text{declared}(n) \subseteq \bigcup \text{declared}(n_i) \) and \( \bigcup \text{declared}(n_i) \subseteq \text{declared}(n) \). Therefore, \( \text{delegated}(n) \) is the union of \( \text{declared}(n_i) \), where \( n_i \) is a child node of \( n \).

By replacing the containment tree nodes in Property 3 with their corresponding pages, we have Corollary 2

**Corollary 2.** \( \text{Delegated}(P) \) is the union of \( \text{declared}(P_j) \) for all pages \( P_j \) that are immediately contained pages of \( P \).
**Property 4.** In a containment tree, the declared space of a node $n$ is the union of $\text{home}(n_i)$ for all nodes $n_i$ in the subtree rooted at $n$.

We show the correctness of Property 4 by induction on the height of the subtree. (The height is defined as the maximum number of edges on a path from the root of the subtree to a leaf node of the subtree).

Step 1, the basis: we will show the property holds when $height = 0$. When height = 0, it means there is only one node in the subtree. That is $n$. Since $n$ has no child node, we know $\text{delegated}(n)$ is empty and $\text{declared}(n) = \text{home}(n)$. Hence, property 4 is true. (Note that we use $\text{declared}(n)$, $\text{home}(n)$ and $\text{delegated}(n)$ to represent the declared space, home space and delegated space of $n$’s corresponding page $P$.)

Step 2, the inductive step: assume the property holds for $height = t$ ($t \geq 0$), we show it is also true when $height = t + 1$. With the condition $height = t + 1, t \geq 0$, we know that $n$ has at least one child node. Let’s consider all the child nodes of $n$. Let’s call them $n_1, ..., n_m$, where $m \geq 1$.

(1) From the induction hypothesis, we know that, for each child node $n_k$, Property 4 holds because the height of the subtree rooted at $n_k$ is at most $t$. That is $\text{declared}(n_k) = \bigcup \text{home}(n_{kj})$ where $n_{kj} (j = 1...i_k)$ is in the subtree rooted at $n_k$. This is described in Figure 3.6

![Figure 3.6: A figure to illustrate Property 4.](image)

(2) From Property 3, we know $\text{delegated}(n)$ equals the union of $\text{declared}(n_k)$, where $n_k$ is a child node of $n$, i.e. $\text{delegated}(n) = \bigcup \text{declared}(n_k)$.

(3) According to the definition of home space, we know $\text{declared}(n) = \text{home}(n) \cup \text{delegated}(n)$. Replace $\text{delegated}(n)$ with $\bigcup \text{declared}(n_k)$ from (2), we have $\text{declared}(n)$ equals $\text{home}(n) \cup (\bigcup_k \text{declared}(n_k))$. Now replace $\text{declared}(n_k)$ with $\bigcup \text{home}(n_{kj})$ from (1), we have $\text{declared}(n) = \text{home}(n) \cup (\bigcup \text{home}(n_{kj}))$, where $n_{kj}$ is a node in the subtree rooted at $n_k$ and $n_k$ is a child node $n$. Therefore, $\text{declared}(n)$ is the union of $\text{home}(n_i)$ for all nodes $n_i$ in the subtree rooted at $n$. Thus, the property is true.
3.2. THE CONTAINMENT TREE

Given a node $n$ and its corresponding page $P$, there is a one to one mapping from a node in $n$’s subtree to a contained page of $P$. Therefore, we can rewrite Property 4 with Corollary 3.

**Corollary 3.** $\text{Declared}(P) = \text{home}(P) \cup (\cup_i \text{home}(P_i))$ where $P_i$ is a contained page of $P$.

**Property 5.** The spaces $\text{home}(n_i)$ for all nodes $n_i$ in a containment tree are disjoint.

We prove this property in a similar fashion as the proof of Property 4. We also use induction on the height of the subtree. Let $n$ be the root node of the containment tree.

Step 1, The base case is when $\text{height} = 0$. In this case, $n$ is the only page in the subtree. $\text{Home}(n)$ is the only element. Therefore, the property holds.

Step 2, Assume Property 5 is true when $\text{height} = t$ ($t \geq 0$). Now let’s consider the case where $\text{height} = t + 1$. Again we use $n_1 \ldots n_m$ to represent the child nodes of $n$ as shown in Figure 3.6.

We discuss the home space of $n$, the home space of $n$’s child nodes, and the home space of $n$’s descendant nodes respectively in the following 3 steps.

Step 2.1, The home space of $n$ and the home space of $n$’s descendant nodes are disjoint. Let $n_{kj}$ be a node in the subtree rooted at $n_k$ ($k = 1 \ldots m$). From the home space definition, $\text{home}(n) \cap \text{delegated}(n) = \phi$. From Corollary 2, $\text{delegated}(n) = \cup \text{declared}(n_k)$. Therefore, $\text{home}(n) \cap \text{declared}(n_k) = \phi$. $\text{Home}(n_{kj}) \subset \text{declared}(n_k)$. Therefore $\text{home}(n) \cap \text{home}(n_{kj}) = \phi$.

Step 2.2, The home spaces of nodes in the subtrees rooted at different child nodes of $n$ are also disjoint. Let $n_k$ and $n_{k'}$ be two child nodes of $n$. Let $n_{kj}$ be a node in the subtree rooted at $n_k$ and $n_{k'j'}$ be in the subtree rooted at $n_{k'}$. Node $n_k$ and $n_{k'}$ are sibling nodes, so their declared spaces are disjoint by definition, i.e. $\text{declared}(n_k) \cap \text{declared}(n_{k'}) = \phi$. $\text{Home}(n_{kj}) \subset \text{declared}(n_k)$ and $\text{home}(n_{k'j'}) \subset \text{declared}(n_{k'})$, therefore $\text{home}(n_{kj}) \cap \text{home}(n_{k'j'}) = \phi$.

Step 2.3, From the induction hypothesis, we also know that the home spaces of two nodes inside the subtree rooted at $n_k$ are disjoint.

Step 2.1, 2.2 and 2.3 have shown that the Property 5 is true when $\text{height} = t$.

From Step 1 and Step 2, we can conclude that $\text{home}(n_i)$ for all nodes in the subtree rooted at $n$ are disjoint. Hence, Property 5 holds.

From Property 5, we can derive Corollary 4 and Corollary 5 by replacing node $n$ and $n_i$ with their corresponding pages.

**Corollary 4.** For any $P_i$ that is a contained page of $P$, $\text{home}(P)$ and $\text{home}(P_i)$ are disjoint.

**Corollary 5.** For any two pages $P_i$ and $P_j$ which correspond to nodes in the same containment tree, $\text{home}(P_i) \cap \text{home}(P_j) = \phi$. 

Property 6. The home spaces of all nodes in a subtree rooted at node $n$ partitions $\text{declared}(n)$.

Proof sketch: (1) From Property 4, we know $\text{declared}(n) = \bigcup \text{home}(n_i)$ for all nodes $n_i$ in the subtree of $n$. (2) From Property 5, we know that $\text{home}(n_i)$ are disjoint. Combine (1) and (2), we know that $\text{home}(n_i)$ partition $\text{declared}(n)$.

Corollary 6. $\text{declared}(P)$ is partitioned by $\text{home}(P)$ and $\text{home}(P_i)$ for all $P_i$ that is a contained page of $P$.

Property 7. The declared spaces of all nodes in a containment tree form a well-nested set.

For any two nodes in a containment tree, if one is an ancestor node of the other, then the declared space of the ancestor node contains the declared space of the descendant node; if none of them is an ancestor of the other, then according to Property 2, their declared spaces are disjoint. Therefore, a containment tree is well-nested.

Now that we have discussed the static properties of a containment tree, we will look at the tree transformations in the next section.

3.2.3 Operations on a Containment Tree

In general, there are two types of operations that one can perform on a containment tree: (1) static operations such as searching over the tree for a node and (2) dynamic operations that change the tree structure, such as adding a new node or removing an existing node.

Static searches are developed based on the properties discussed Section 3.2.2. We will explain how to perform a search operation in detail in the next section.

In this section, we focus on the structural changes in a containment tree. These changes should obey the definitions provided in Section 3.2.1, which are the base for all the properties used in a search. If these definition are violated, the correctness of the tree properties and the correctness of the search operations can not be guaranteed anymore.

Section 3.2.1 can be summarized into the following two conditions:

i Any node in a containment tree can have at most one immediate container.

ii The declared space of sibling nodes do not intersect.

In the following, we will show how to add or remove a node following these two conditions. To maintain the correctness of the containment tree, certain conditions need to be met during the
operations. We summarize these conditions as “requirements”. Note that these requirements are directly derived from the definitions of the containment tree. They are not something that need to be maintained in addition to the definitions.

**Adding a New Node**

When a node $n$ is added to a containment tree, there are three possibilities: (1) $n$ is the first node in the tree, (2) $n$ is added as a leaf node, and (3) $n$ is added in the middle of the tree, i.e. $n$ is not a root node or a leaf node. (Note that a new node $n$ will not be added as the root node unless $n$ is the first node. If $n$ is the first node, then the above two conditions are trivially satisfied.) In any case, the declared space of $n$ can not be empty, which is part of the containment tree definition (Definition 11).

Assuming both conditions are satisfied by every node in the tree before the new node $n$ is added, in the following, we will show case by case what the requirements are to add $n$ into the containment tree.

**Case 1**, if $n$ is the only node in a containment tree, $n$ has no container nor sibling node. So the two conditions are trivially satisfied. There is no requirement on $n$.

**Case 2**, $n$ is added as a non-root leaf node. The scenario where $n$ is added as both a leaf node and a root node is covered in Case 1. Adding $n$ as a non-root leaf node means $n$ has one container node. Let it be $n_1$. Assume $b_1, b_2, ..., b_k$ ($k \geq 0$) are the child nodes of $n_1$, as shown in Figure 3.7(2). The nodes whose immediate container or sibling nodes have changed are $n$ and $b_i$ ($i = 1, ..., k$). $n$ has a new container node $n_1$, so declared($n$) should be contained in declared($n_1$). $n$ becomes the new sibling node of $b_i$, so declared($n$) should not intersect declared($b_i$). This is stated

![Figure 3.7](image-url)
CHAPTER 3. THE C-TREE FRAMEWORK

in Requirement 1.

**Requirement 1.** When a node $n$ is added as a child node of $n_1$ and $n$ is a leaf node, (1) the declared space of $n$ can not be empty, (2) the declared space of $n$ must be contained in the declared space of $n_1$, and (3) the declared space of $n$ can not intersect any existing child node of $n_1$.

**Case 3,** $n$ is added in the middle of the tree. $n$ is not a root node nor leaf node. This means $n$ has an immediate container and at least one child node. Assume node $b$ is $n$’s parent node. Assume $n_1$ ... $n_i$ are the child nodes of $n$ and $b_1...b_j$ are the child nodes of $b$ after $n$ is added.

Node $n$ is a child node of $b$, so the declared space of $n$ needs to be contained in the declared space of $b$. Node $n_1$ ... $n_i$ are the child nodes of $n$, so declared($n$) should contain declared($n_i$). Node $n$ is also the new sibling nodes of $b_1$ ..$b_j$, so the declared space of $n$ and the declared space of $b_k$ ($k = 1$..$j$) should be disjoint.

In the following we will show that node $n_1$... $n_i$ should also be the child nodes of $b$ before $n$ is added. Since declared($n_k$) $\subset$ declared($n$) and declared($n$) $\subset$ declared($b$), declared($n_k$) $\subset$ declared($b$). So before $n$ is added, $n_k$ was in the subtree rooted at $b$ (Corollary 1). If $n_k$ was not a child node of $b$, then $n_k$ must be a descendant of $b_t$. This would mean that declared($n_k$) is contained in both declared($b_t$) and declared($n$), which can not be true because $n$ and $b_t$ are sibling nodes. Therefore, $n_k$ must be a child node of $b$ before $n$ is added.

The above changes are summarized in Requirement 2.

**Requirement 2.** Suppose $n$ is added to a containment tree, such that $n$ is a child node of $b$ and $n$ has at least one child node. After $n$ is added to the tree, assume $n$’s child nodes are $n_1$ ... $n_i$ ($i \geq 1$) and $b$’s child nodes are $n$, $b_1$ ..$b_j$ ($j \geq 0$). The following requirements must be satisfied (1) $n_1$...$n_i$ must be $b$’s child nodes before $n$ is added, (2) declared($n$) $\subset$ declared($b$), (3) ($\cup_{k=1...i}$ declared($n_k$)) $\subset$ declared($n$), and (4) declared($n$) $\cap$ declared($b_t$) = $\phi$, where $b_t$ is a child node of $b$ after $n$ is added. The new delegated space of $b$ is the old delegated space of $b$ union the declared space of $n$.

**Removing a Node from the Containment Tree**

When a node $n$ is deleted from a containment tree, two questions need to be answered: (1) what happens to the space that node represents and (2) what happens to the child nodes of $n$. A merge operation takes care of both questions. We use the merge operation defined in Definition 13 to delete a node from a containment tree.
Definition 13 (Node Merge). When node $n_2$ is merged into a node $n_1$, $n_2$ is removed from the tree, the child nodes of $n_2$ become child nodes of $n_1$ and $\text{home}_{\text{new}}(n_1) = \text{home}_{\text{old}}(n_1) \cup \text{home}(n_2)$, $\text{declared}_{\text{new}}(n_1) = \text{declared}_{\text{old}}(n_1) \cup \text{declared}(n_2)$.

Combining the above definition of node merge and the definition of declared space (Definition 5), we can infer that a node can only be merged into its parent node. The definition of declared space states that a declared space once created should remain unchanged, which means the declared space of $n_1$ should be the same before and after declared($n_2$) is merged into it, i.e. declared$_{\text{old}}(n_1) \cup \text{declared}(n_2) = \text{declared}_{\text{old}}(n_1)$. Therefore, declared($n_2$) must be contained in declared$_{\text{old}}(n_1)$. So node $n_1$ must be an ancestor node of $n_2$. Node $n_2$ can not be merged into a non-parent ancestor node, because declared($n_2$) is in the declared space of its parent node, which is part of the delegated space of $n_2$’s non-parent ancestor node. Thus, to maintain the invariance of declared space, a node can be merged into its parent node only. This is stated in Requirement 3.

Requirement 3. In a containment tree, a node can be merged into its parent node only. The root node can not be deleted unless it is the only node in the containment tree.

Note that Requirement 3 not only guarantees the invariance of declared space, but also ensures that the containment tree remains well-nested after the merge operation. Figure 3.8 illustrates an example of node merge. Assume node $n_2$ is merged into its parent $n_1$. Node $a_i$s are the child nodes of $n_1$ before the merge. And $b_j$s are the child node of $n_2$. After the merge, node $a_i$s and $b_j$s are the only nodes in this containment tree whose parent and/or sibling nodes have changed. Assuming the containment tree is well-nested before the merge, we have (1) the declared spaces of $a_i$s and $b_j$s are all contained in declared($n_1$) and (2) the declared spaces of $a_i$s do not intersect the declared space of $n_2$, therefore they do not intersect the declared spaces of $b_j$s either. These two facts still hold after the merge. Therefore, the containment tree is still well-nested after $n_2$ is merged into $n_1$.

In this section, we have shown what requirements need to be satisfied during tree transformation to maintain the correctness of the containment tree. In the next section, we will show how to perform searches over the containment tree using the properties defined in Section 3.2.2.

### 3.2.4 Searches in a Containment Tree

In this section, we show how to perform searches in a containment tree.
Definition 14 (Containment Tree Search). Given a search predicate $Q$, a containment tree search looks for a set of nodes $N$ such that, for every node $n_i \in N$, the predicate of the home space of $n_i$ intersects $Q$.

A containment tree search starts at the root node of the containment tree. We first look at the home space of the root node $n$. If the home space of $n$ intersects $Q$, $n$ is added to the query result list. We then look at the declared spaces of $n$’s child nodes. Those child nodes whose declared spaces satisfy $Q$ are pushed into a stack as to-be-visited nodes. The above steps are repeated for every node in the stack. The search follows possibly multiple paths down the containment tree and visit all the nodes whose home spaces satisfy $Q$. The pseudo-code is given in Algorithm 1.

To justify the correctness of a search, one needs to demonstrate two things (1) all desired results are found by the search and (2) the results returned by the search procedure are all desired ones. We will show that these two statements are true by examining the properties of a containment tree. We use $Q$ to represent the query predicate based on search space $q$.

Step 1: All qualified nodes are found by the search procedure. We will show this by contradiction.

Assume node $n$ is the node whose home space satisfies $Q$ but is not found by the search. Let’s assume the path from the root node $n_0$ to $n$ is $n_0$, $n_1$, $n_2$, ..., $n_k$, $n$. Since $n_0$ is visited, but $n$ is not visited, we know that there must exist a node $n_i$ ($i \geq 0$) such that $n_i$ is visited and $n_{i+1}$ is not added to the stack (i.e. Line 7 in Algorithm 1 is not satisfied for $n_{i+1}$). This means the declared space of $n_{i+1}$ does not satisfy $Q$.

On the other hand, since $n_{i+1}$ is a container node of $n$, the declared space of $n_{i+1}$ must contain the home space of $n$. Since the home space of $n$ intersects $Q$, the declared space of $n_{i+1}$ must also
**Algorithm 1** Search(Node $n$, QueryPredicate $Q$). Given a node $n$ in a containment tree and a query predicate $Q$, find all nodes $m$ in the subtree rooted at $n$ such that the home space of $m$ satisfies $Q$. Let $S$ be a stack, which is empty when the program is first initialized. Let $L$ be a list that stores the query result. $L$ is empty at the beginning.

1: if $\text{home}(n) \cap Q \neq \emptyset$ 
2: add $n$ to $L$; 
3: end if

4:

5: for each child node $n_i$ of $n$ do
6: if declared($n_i$) $\cap Q \neq \emptyset$ then
7: push $n_i$ to stack $S$.
8: end if
9: end for

10: if $S$ is not empty then
11: $n \leftarrow S$.pop;
12: Search($n$, $Q$);
13: else
14: return $L$.
15: end if
16: end if
intersects $Q$, which contradicts to the analysis in the previous step. Therefore, the assumption is wrong. Node $n$ must be found by the search procedure.

**Step 2:** The result set does not include any node whose home space does not satisfy $Q$.

In Algorithm 1, list $L$ stores the query result. Since a node is added to $L$ (Line 2 in Algorithm 1) only when its home space satisfies $Q$, it is trivial to show that all the reported nodes are qualified results.

Note that the above search procedure is a generalized method. Other queries, such as exact match query or range query, are special cases of this generalized method and might be further optimized. As an example, in the following, we briefly describe how to optimize Algorithm 1 for exact match queries.

An **exact match query** in a containment tree looks for a node whose home space contains the given search space. In this case, the intersection of the search predicate and the home space of the desired node includes one single object. Since the home spaces of all nodes in a containment tree are disjoint, there is at most one node in a containment tree that satisfies the exact match search predicate $Q$. So once we find a node whose home space satisfies the query, we do not need to look further down the tree (Line 2 in Algorithm 1 can be optimized as “return $n$”). Since the declared spaces of sibling nodes are disjoint, given a set of sibling nodes, there is at most one node whose declared space satisfies $Q$ (in Line 5 to Line 9 in Algorithm 1, there is at most one $n_i$ that satisfies $Q$). So once we find the node $n_i$ whose declared space satisfies $Q$, we don’t need to look at other sibling nodes any more. The generalized search method can be optimized for exact match query as shown in Algorithm 2.

### 3.3 Generalizations of Containment Trees

In the last section, we discuss the properties, searches and operations of a single containment tree. In this section, we will discuss the properties, searches and operations of a series of containment trees. We start by introducing the related definitions.

#### 3.3.1 Definitions

**Definition 15 (Partial Subtree).** Given a containment tree $T$, we say nodes $n_1...n_k$ form a **partial subtree** of $T$ rooted at $n_1$ if (1) node $n_1...n_k$ are all in the subtree rooted at $n_1$, and (2) for any two nodes $n_i$ and $n_j$ in the partial subtree, if there is a path from $n_i$ to $n_j$ in $T$, then all nodes along this path are also in the partial subtree.
3.3. GENERALIZATIONS OF CONTAINMENT TREES

Algorithm 2 *ExactMatch*(Node $n$, QueryPredicate $Q$). Given a node $n$ and a query predicate $Q$ defined as “contains $q$”, find the node $m$ in the subtree rooted at $n$ such that the home space of $m$ satisfies $Q$.

1: if $\text{home}(n) \cap Q \neq \emptyset$ then
2: return $n$;
3: end if
4:
5: for each child node $n_i$ of $n$ do
6: if $\text{declared}(n_i) \cap Q \neq \emptyset$ then
7: return $\text{ExactMatch}(n_i, Q)$;
8: end if
9: end for
10:
11: return nil;

Figure 3.9: Only the sets indicated in (a) are partial subtrees.

Figure 3.9 illustrates examples of partial subtrees. In Figure 3.9(a), region $R_1$, $R_2$, and $R_3$ are all partial subtrees of $T$. $R_4$ and $R_5$ in Figure 3.9(b) and Figure 3.9(c), however, are not partial subtrees of $T$. In $R_4$ in Figure 3.9(b), node $n_{10}$ does not have a common ancestor as the others. $R_5$ in Figure 3.9(c) does not satisfy condition (2) in the partial subtree definition. Node $n_3$ and $n_9$ are both in $R_5$, but node $n_5$, which is on the path from $n_3$ to $n_9$ is not part of $R_5$.

Note that a partial subtree is different from a normal subtree. A subtree normally includes all nodes below a certain node in the tree. If $R$ is a subtree of $T$, then the leaf nodes of $R$ must also be leaf nodes of $T$. But the leaf nodes of a partial subtree of $T$ can be internal nodes of $T$. A subtree must be a partial subtree, but a partial subtree may not be a subtree.
**Definition 16 (Leaf Nodes of a Partial Subtree).** Let $R$ be a partial subtree of $T$. Given a node $n$ in $R$, $n$ is called a **leaf node** of $R$ if $n$ does not have any child node or none of $n$’s children is in the partial subtree $R$.

In the running example in Figure 3.9(a), node $n_7$ and $n_9$ are leaf nodes of $R_1$, node $n_3$ and $n_4$ are leaf nodes of $R_2$, and $n_{10}$ and $n_{11}$ are leaf nodes of $R_3$.

**Definition 17 (Child Nodes of a Partial Subtree).** Child nodes of the leaf nodes of a partial subtree are called the **child nodes** of the partial subtree.

In the above example in Figure 3.9(a), the child node of $R_1$ is $n_8$, the child nodes of $R_2$ are $n_5$, $n_7$, and $n_6$, and $R_3$ does not have any child node.

**Definition 18 (Refinement Subtree/Generalization Node).** $T_1$ and $T_2$ are two containment trees. Let $m$ be a node in $T_2$ and $R$ be a partial subtree of $T_1$ rooted at $n$, node $m$ is called a **generalization node** of $R$ if (1) $\text{declared}(n) = \text{declared}(m)$, (2) $m$ and $R$ have the same number of child nodes, and (3) for every child node of $m$, there exists a child node of $R$ with the same declared space. If $m$ is a generalization node of $R$, then $m$ is a generalization node of every node in $R$ and $R$ is called a **refinement subtree** of $m$.

Figure 3.10: Node $a_1$ in $r_2$ is a generalization node of subtree $R_1$ in (a).

Figure 3.10 illustrates an example of Definition 18. In Figure 3.10(a), $R_1$ is a partial subtree of $T$. Node $a_1$ in $T_2$ in Figure 3.10(b) is a generalization node of $R$, because (1) the declared space of $a_1$ is the same as the declared space of $n_1$, which is the root node of $R$, (2) $n_1$ and $a_1$ both have only one child node, and the declared spaces of their child nodes ($n_4$ and $a_2$) are the same.
3.3. GENERALIZATIONS OF CONTAINMENT TREES

Definition 19 (Generalization/Refinement). Given two containment trees $T_1$ and $T_2$, if every node in $T_2$ is a generalization node of a partial subtree of $T_1$, then $T_2$ is called a generalization of $T_1$, and $T_1$ is called a refinement of $T_2$.

Figure 3.11: Generalization and refinement subtree of containment trees.

Figure 3.11 shows an example of Definition 19. $T_1$ is a refinement of $T_2$ and $T_2$ is a generalization of $T_1$. This is because, for every node $m_i$ in $T_2$, there is a partial subtree of $T_1$ that considers $m_i$ as a generalization node.

So far we have discussed the generalization and refinement between two containment trees. This relationship is transitive and can be applied to multiple containment trees.

Definition 20 (Well-Formed Containment Hierarchy). Given a list of containment trees $< T_1, ... T_k >$, if $\forall i (i = 1...k - 1)$, $T_i$ is a refinement of $T_{i+1}$, then $< T_1, ..., T_k >$ is called a well-formed containment hierarchy. We say $T_i$ is in the $i^{th}$ level of the hierarchy. Level 1 is in the bottom of the hierarchy and is also called the leaf level.

Definition 21 (Refinement Pointer). Let $T_i$ and $T_{i+1}$ be two adjacent containment trees in a well-formed containment hierarchy with $T_i$ being a refinement of $T_{i+1}$. Let $m$ be a node in $T_{i+1}$ and $n$ be the root node of $m$’s refinement subtree in $T_i$. A pointer from $m$ to $n$ is called a refinement pointer.

Figure 3.12 shows a well-formed containment hierarchy $< T_1, T_2, T_3 >$ with refinement pointers. Note that refinement pointers only exist between two adjacent levels of a containment hierarchy. If we look at the containment hierarchy from bottom up (in the order of $T_1$, $T_2$, $T_3$), then every level is a refinement of the next level. If we traverse the hierarchy in a top down fashion, then a level becomes a generalization of the next level.
CHAPTER 3. THE C-TREE FRAMEWORK

3.3.2 Properties

Property 8. Given two containment trees \( T_1 \) and \( T_2 \), let \( T_1 \) be a refinement of \( T_2 \). Every node in \( T_2 \) corresponds to a refinement subtree in \( T_1 \). These refinement subtrees partition \( T_1 \).

To prove that the refinement subtrees partitions \( T_1 \), we need to show that (1) every node of \( T_1 \) is in a refinement subtree, and (2) no node of \( T_1 \) is in two refinement subtrees.

(1) Every node of \( T_1 \) is in a refinement subtree.

Let \( n \) be a node in \( T_1 \), if there exists a node \( m \) in \( T_2 \) with the same declared space as \( n \), then \( n \) is in the refinement subtree of \( m \). If there does not exist any node in \( T_2 \) with the same declared space as \( n \), then we consider node \( m \) in \( T_2 \) such that the declared space of \( m \) is the smallest declared space in \( T_2 \) that contains the declared space of \( n \). Since \( \text{declared}(m) \) is the smallest declared space in \( T_2 \) that contains \( \text{declared}(n) \), \( \text{declared}(n) \) is not contained in the declared space of any child node of \( m \). Hence, the corresponding nodes of \( m \)'s child nodes in \( T_1 \) can not be a container node of \( n \). By definition, \( n \) must be in the refinement subtree of \( m \).

(2) We use proof by contradiction to show that no node in \( T_1 \) is in more than one refinement subtrees. Assume \( n \) is a node in \( T_1 \) and \( n \) belongs to two refinement subtrees \( R_1 \) and \( R_2 \). Let \( n_1 \) be the root node of \( R_1 \) and \( n_2 \) be the root node of \( R_2 \). The intersection of \( \text{declared}(n_1) \) and \( \text{declared}(n_2) \) is not empty, since both of them contain \( \text{declared}(n) \). Therefore, \( n_1 \) and \( n_2 \) must have a containment relationship. Without loss of generality, assume \( n_1 \) is a container of \( n_2 \).

Let \( m_1 \) be the node in \( T_2 \) with the same declared space as \( n_1 \) and \( m_2 \) be the node in \( T_2 \) with the same declared space as \( n_2 \). Node \( m_1 \) is an ancestor node of \( m_2 \). According to the refinement subtree definition, \( n_2 \) must be in the subtree of a child node of \( R_1 \). So the subtree rooted at \( n_2 \) is
outside $R_1$. Node $n$ is in $R_2$, so $n$ is in the subtree rooted at $n_2$, therefore $n$ is also outside of $R_1$. This contradicts with the assumption that $n$ is in both $R_1$ and $R_2$. Therefore, the contradiction assumption does not hold. There does not exist any node in $T_1$ that is in more than one refinement subtree.

**Property 9.** A generalization node’s home space is partitioned by home spaces of nodes in its refinement subtree.

To show the correctness of Property 9, we define the following notations for convenience. Figure 3.13 illustrates these notations.

- $m$ is a generalization node in $T_2$.
- $d_1...d_k$ are child nodes of $m$.
- $R$ is $m$’s refinement subtree in $T_1$.
- $n$ is the root node of $R$ and $a_1 ...a_t$ are non-root nodes of $R$.
- $c_1...c_k$ are child nodes of the refinement subtree $R$.
- $b_1...b_m$ are descendant nodes of $c_i$s ($i = 1..k$).

![Figure 3.13: A figure to illustrate Property 9.](image)

Using the above notation, Property 9 can be rewritten as “home($m$) is partitioned by home($n$) and home($a_i$)s for $i = 1..t$”. We show the correctness in two steps.

**Step 1:** $\text{home}(m) = \text{home}(n) \cup (\bigcup \text{home}(a_i))$. 
(1) In $T_2$, from Property 3 of containment tree $(\text{delegated}(n) = \bigcup \text{declared}(n_i)$ where $n_i$ is a child node of $n$), we have:

$$\text{declared}(m) = \text{home}(m) \cup (\bigcup_{i=1..k} \text{declared}(d_i)). \quad (3.1)$$

(2) In $T_1$, according to Property 4 of containment tree (the declared space of a node $n$ is the union of home($n_i$) for all $n_i$ in the subtree rooted at $n$), we have:

$$\text{declared}(n) = \text{home}(n) \cup (\bigcup_{i=1..t} \text{home}(a_i)) \cup (\bigcup_{i=1..m} \text{home}(b_i)) \cup (\bigcup_{i=1..k} \text{home}(c_i)) \quad (3.2)$$

(3) In $T_1$, apply Property 4 to those subtrees rooted at node $c_i$, we have the following:

$$(\bigcup_{i=1..k} \text{declared}(c_i)) = (\bigcup_{i=1..k} \text{home}(c_i)) \cup (\bigcup_{i=1..m} \text{home}(b_i)) \quad (3.3)$$

(4) Apply equation (3.3) to equation (3.2), it follows

$$\text{declared}(n) = \text{home}(n) \cup (\bigcup_{i=1..t} \text{home}(a_i)) \cup (\bigcup_{i=1..k} \text{declared}(c_i)) \quad (3.4)$$

Combining equation (3.1), (3.4) and the fact that $\text{declared}(m) = \text{declared}(n)$ and $\text{declared}(c_i) = \text{declared}(d_i)$, we have $\text{home}(m) = \text{home}(n) \cup (\bigcup_{i=1..t} \text{home}(a_i))$.

**Step 2:** According to Property 5 of containment tree, the home spaces of all nodes in a containment tree are disjoint. Therefore, home($n$) and home($a_i$)s are disjoint.

From Step 1 and Step 2, we know that home($m$) is partitioned by home($n$) and home($a_i$)s.

Property 9 is the building block of searches in the C-Tree. It shows that the nodes in a generalization form of a containment tree are in “coarser granularity” in terms of home spaces. A refinement subtree partitions the home space of a generalization node into “finer granularity”, hence the name “refinement subtree”. To search for a node in a containment tree $T$, not only can one traverse the “fine-grained” structure starting from the root node of $T$, but one can also make use of the “coarse-grained” generalizations of $T$ and visit possibly less nodes. Such searches via a containment hierarchy are discussed next.

**Property 10.** Given any two containment tree $T_i$ and $T_j$ in a containment hierarchy $<T_1, ..., T_k>$, the declared space of the root node of $T_i$ must be the same as the declared space of the root node of $T_j$.

Without loss of generality, assume $T_i$ is a generalization of $T_j$. Let $R_i$ be the root node of $T_i$ and $R_j$ be the root node of $T_j$. 
From Definition 18 (refinement subtree/generalization node) and Definition 19 (refinement/generalization), we know that \( R_i \) is a generalization node of a partial subtree in \( T_j \). Therefore, there must exist a node \( n \) in \( T_j \) such that the declared space of \( n \) is the same as the declared space of \( R_i \).

Now we use proof by contradiction to show that \( n \) must be \( R_j \). Assume \( n \) is not \( R_j \). Since \( R_j \) is the root node of \( T_j \), \( n \) must be either a child node or a descendant of \( R_j \). Therefore, the declared space of \( n \) is contained in the declared space of \( R_j \) and they can not be the same (i.e. \( \text{declared}(n) \subset \text{declared}(R_j) \)). According to the definition of generalization/refinement (Definition 19), there must be a node \( m \) in \( T_i \) that is the generalization node of \( R_j \). The declared space of \( m \) is the same as the declared space of \( R_j \). Since \( \text{declared}(m) = \text{declared}(R_j) \), \( \text{declared}(n) = \text{declared}(R_i) \), and \( \text{declared}(n) \subset \text{declared}(R_j) \), we have \( \text{declared}(R_i) \subset \text{declared}(m) \), which contradicts the fact that \( R_i \) is the root node of \( T_i \) and \( m \) is a node in \( T_i \). Therefore, the assumption is wrong. Node \( n \) must be \( R_j \).

So the declared space of \( R_i \) is the same as the declared space of \( R_j \).

### 3.3.3 Searches

Searches in a containment hierarchy are based on two observations:

- If a search space \( S \) satisfies a query \( Q \), then all containers of \( S \) also satisfy \( Q \). (Definition ?? and Property ??)

- The home space of a node \( n \) has two types of containers:

  1. the declared spaces of \( n \)'s ancestor nodes in the same containment tree. We call this the **child pointer containment** for convenience.

  2. the home space of \( n \)'s generalization nodes in the upper level containment trees in a containment hierarchy (Property 9). We call this the **refinement pointer containment**

Searches using child pointer containment, i.e. searches in a single containment tree, are discussed in Section 3.2.4. In this section, we will look at searches in a containment hierarchy, i.e. the refinement pointer containment. The rest of this section is organized as follows. (1) We first give an example of searches using these two types of containment. (2) We then define the generalized search procedure in a containment hierarchy. After that, we show the correctness of the search. (3) At the end, we discuss the pros and cons of searches in a single containment tree and searches using a containment hierarchy.
Examples

We start with an example illustrated in Figure 3.14. \(< T_1, T_2, T_3 >\) is a containment hierarchy. Search space \(q\) is \([135, 140)\). Let \(Q\) be a query predicate defined as “contains \(q\)”. We want to find the node in \(T_1\) whose home space satisfies \(Q\).

To search within \(T_1\), we use Algorithm 2 defined in Section 3.2.4. We start from the root node of \(T_1\), visit every node on the path from \(n_1\) to \(n_{14}\) and report \(n_{14}\) as the query result. The query path is shown using highlighted solid line in Figure 3.15. All 6 nodes along this path are visited.

Alternatively, we can use the containment hierarchy and refinement pointers. If node \(n_{14}\) is a qualified query result, i.e. the home space of \(n_{14}\) satisfies \(Q\), then the home space of \(n_{14}\)’s generalization nodes, which is \(t_2\), and \(m_4\) in this case, must also satisfies \(Q\). So we can start with the root level containment tree \(T_3\). In the refinement subtree in every level, we look for the nodes whose home spaces satisfy \(Q\) and then follow refinement pointers of these nodes until we reach the qualified nodes in \(T_1\). The highlighted dash line in Figure 3.15 illustrates the search path in the containment hierarchy. Node \(t_1, t_2, m_4\) and \(n_{14}\) are visited in this case.
3.3. GENERALIZATIONS OF CONTAINMENT TREES

Figure 3.15: Different ways to perform searches in a containment hierarchy.

Searches in a Containment Hierarchy

Now we define the generalized search procedure in a containment hierarchy. Given a query predicate $Q$ and a containment hierarchy $< T_1, \ldots, T_k >$, to search for the nodes in leaf level containment tree $T_1$, we recursively identify in every level of the containment hierarchy the nodes whose home spaces satisfy $Q$ until we reach the leaf level. We first start with the root node of $T_k$ and perform a search as defined in Algorithm 1 within $T_k$. After we find all the qualified nodes in $T_k$, we follow their refinement pointers to the next level and perform the search in Algorithm 1 within their refinement subtrees. The above steps are repeated until we find all the qualified nodes in $T_1$. The pseudo code is given in Algorithm 3.

Note that, in Line 11 in Algorithm 3, the search is limited to nodes inside $n$’s refinement subtree only. Given a node $t$ in the subtree rooted at $m$, one can decide whether $t$ is inside is $n$’s refinement subtree or not by remembering the home space of $n$ or the declared space of $n$’s child nodes. If the home space of $t$ is not contained in the home space of $n$, or if the declared space of $t$ is contained in the declared space of any of $n$’s child nodes, then $t$ is not inside the refinement
subtree of \( n \) (these conditions can be added to Line 6 of Algorithm 1).

**Algorithm 3** *Search*(ContainmentHierarchy \(< T_1, \ldots, T_k >\), QueryPredicate \( Q \)). Given a containment hierarchy \(< T_1, \ldots, T_k >\) and a query predicate \( Q \), find all nodes \( m \) in \( T_1 \) such that the home space of \( m \) satisfies \( Q \). Let \( S \) be a stack, which is empty when the program is first initialized. Let \( L \) be a list that stores the query result. \( L \) is empty at the beginning.

1: \( n \leftarrow \) the root node of \( T_k \);
2: \( L_i \leftarrow \) Search\((n, Q)\); \{defined in Algorithm 1\}
3: push every node in \( L_i \) to \( S \).
4: 
5: while \( S \) is not empty do
6: \( n \leftarrow S\.pop.\)
7: if \( n \) is in the leaf level containment tree \( T_1 \) then
8: add \( n \) to \( L \).
9: else
10: \( m \leftarrow n\.refinement\.pointer \); \{i.e. \( m \) is the root node of \( n \)’s refinement subtree in the next level.\}
11: \( L_i \leftarrow \) Search\((m, Q)\) \{as defined in Algorithm 1 and the search is limited in the refinement subtree of \( n \) only. \}
12: push every node in \( L_i \) to \( S \).
13: end if
14: end while
15: 
16: return \( L \).

As usual, we show the correctness of the above search procedure in 2 steps.

**Step 1**: All qualified nodes are found.

Assume node \( a_1, a_2, \ldots, a_h \) are the nodes whose home spaces satisfy \( Q \). From Property ??, the home spaces of generalization nodes of every \( a_i \) also satisfy \( Q \). Since we have shown the correctness of Algorithm 1, we know that all of these generalization nodes will be identified as the temporary query result in each level of the containment tree hierarchy (Line 2 and Line 11 of Algorithm 3). And eventually all qualified \( a_i \)s will be identified (Line 8 of Algorithm 3) and returned (Line 16 of Algorithm 3).

**Step 2**: The result set does not include any node whose home space does not satisfy \( Q \).
From the correctness of Algorithm 1, we know that a node $m$ is added to $L_i$ and pushed to stack $S$ only if the home space of $m$ satisfies $Q$. Therefore, all nodes returned as the final result in $L$ have home spaces that satisfy $Q$.

**Analysis**

Assume we are interested in qualified nodes in the leaf level containment tree $T_1$ only. Now let’s compare the performance of searching within $T_1$ and searching through the containment hierarchy. In general, we define the performance of a query procedure as the number of qualified results divided by the number of nodes visited. The higher the ratio, the better the performance is.

In the example illustrated in Figure 3.14, we should choose to search through the containment hierarchy, because the ratio of qualified nodes over visited nodes for searching through the containment hierarchy is $1/4$ while the ratio for searching within $T_1$ is $1/6$.

Note that the two ratio numbers in this example happen to be the inverse of the total number of nodes visited. But this is not always true. What matters to a query performance is the ratio, not the number of nodes visited. One should analyze the ratio number before decide which approach to use. Sometimes, a combination of both might be a good optimization.

Figure 3.16 illustrates an example of such a hybrid approach. Let $< T_1, T_2 >$ be a containment hierarchy of numeric data ($T_1$ and $T_2$ are containment trees corresponding to B-Tree pages). Let $q$ be a search space defined by $[28, 42]$. Assume we are looking for nodes in $T_1$ whose home space intersects $q$. We can define query predicate $Q$ as “intersects $q$”. To search within $T_1$ using Algorithm 1, we need to visit every node along the path from $n_1$ to $n_6$, among which 3 nodes are qualified results. The performance ratio is $3/6$. To search through the containment hierarchy as defined in Algorithm 3, we will need to visit all 3 nodes in $T_2$, then follow the refinement pointers of $m_2$ and $m_3$, and visit $n_4$, $n_5$ and $n_6$. The ratio in this case is also $3/6$.

Now if we look at the query and the containment definition closer, we will notice that in this example if the home space of a node say $n_6$ intersects $q$, then the home space of an ancestor of $n_6$ also intersects $q$ unless the delegated space of the ancestor node contains $q$. This means if we can find the node in $T_1$ with the smallest declared space containing $q$, (which is $n_4$ in this case), then all the nodes from $n_4$ to $n_6$ will be qualified result. Therefore, this search can be divided into 2 steps (1) search for the node $t$ in $T_1$ with the smallest declared space that contains $q$, (2) search for all nodes in the subtree rooted at $t$ whose home spaces intersects $q$.

Assume the depth of a containment tree is usually much larger than the number of levels in a
containment hierarchy. Then to search for \( t \) in step 1, we should search through the containment hierarchy. Once we find \( t \), we should search within the leaf level containment tree, because the generalization nodes of \( t \)'s child nodes only contribute to the denominator of the performance ratio, not the numerator. Therefore, we can map the two steps into 2 searches: the first search is a containment hierarchy search (Algorithm 3) with query predicate \( Q_1 \) defined as “contains \( q \)”, and the second search is a containment tree search (Algorithm 1) within the subtree rooted at \( t \) with query predicate \( Q_2 \) defined as “intersects \( q \). Following these steps, we visit 5 nodes in total. They are \( n_1, n_2, n_4, n_5 \) and \( n_6 \). The performance ratio of this hybrid approach is \( \frac{3}{5} \), which is better than using either one alone.

The above optimization is based on the observation that, in the example illustrated in Figure 3.16, if the home space of a node \( n \) intersects the query search \( q \), then the home space of \( n \)'s ancestor must also intersect \( q \) unless the delegated space of the ancestor node contains \( q \). This observation may not be true for every containment hierarchy. Since such optimization relies on the detailed definition of search spaces, we will not discuss this any further.

### 3.3.4 Operations

In this section, we discuss node addition and removal in a containment hierarchy. When a node is added to or removed from the level \( i \) containment tree of a containment hierarchy, not only should the well-nested property of the level \( i \) containment tree be kept, but the overall containment hierarchy should also remain in a well-formed state (i.e. the changes should comply with
3.3. GENERALIZATIONS OF CONTAINMENT TREES

Definition 20). In Section 3.2.3, we have discussed the requirements one should follow when perform operations within a single containment tree. In this section, we discuss what it takes to maintain the containment hierarchy well-formed.

Containment hierarchy is defined based on the generalization/refinement relationship of level $T_i$ and $T_{i+1}$ ($i = 1 \ldots k - 1$). Therefore, to maintain the correctness of the containment hierarchy is to keep the generalization/refinement relationship between adjacent levels.

Let $< T_1, T_2, ..., T_k >$ be a containment hierarchy. Assume $T_i$ is the containment tree where a node $n$ is added or removed. In the following we focus on the operations in $T_{i-1}$ and $T_{i+1}$ that are triggered by changes in $T_i$. Operations in other levels triggered by this change can be recursively derived.

Without loss of generality, we assume that after a node is added to or removed from $T_i$, $T_i$ remains well-formed (i.e. Requirement 1, 2, 3 for operations within a single containment tree are all satisfied).

Node Addition

When a node $n$ is added to $T_i$, we consider three cases (1) required operations in the upper level containment tree $T_{i+1}$, (2) required operations in the lower level containment tree $T_{i-1}$, and (3) when a new level is created due to the addition of $n$.

**Case 1**: Required operations in the upper level containment tree, i.e. $T_{i+1}$ (for $i = 1$ to $k - 1$).

Assume node $k$ is the generalization node of $n$’s parent node, as illustrated in Figure 3.17. Node $n$ cannot be the root node of the refinement subtree (because there does not exist a node in $T_{i+1}$ with the same declared space as $n$) and it cannot be child nodes of the refinement subtree either. Therefore, according to Definition 18, after $n$ is added, the new partial subtree remains as a refinement subtree of the generalization node $k$. Thus, no operation is required in $T_{i+1}$.

**Case 2**: Required operations in the lower level containment tree, i.e. $T_{i-1}$ (for $i = 2$ to $k$).

When a new node $n$ is added to $T_i$, for $T_i$ to be a generalization of $T_{i-1}$, according to Definition 18, 19, there must be a node in $T_{i-1}$ with the same declared space as $n$. Let’s call this node $t$. Figure 3.18 illustrates this case. Assume node $p$ in $T_i$ is the generalization node of $t$ before $n$ is added. After $n$ is added, $t$ is the root node of the new refinement subtree and it becomes a child node of $p$’s refinement subtree. According to Definition 18, $n$ must be a child node of $p$. We summarize this case as Requirement 4.

**Requirement 4.** When a new node $n$ is added to $T_i$, (1) there must be a node $t$ in $T_{i-1}$ with the...
same declared space as \( n \) has (\( t \) becomes the root node of the \( n \)'s refinement subtree), and (2) \( n \) must be added as the child node of \( t \)'s original generalization node before \( n \) is added.

**Case 3:** Adding a new level \((i = k + 1)\).

This is a special instance of case 2. When the newly added node \( n \) is a new level by itself, \( n \) must be a generalization node of the root node of \( T_k \).

**Requirement 5.** *When a new level \( T_{k+1} \) is created, the first node in \( T_{k+1} \) must be the generalization node of the root node of \( T_k \).*
3.3. GENERALIZATIONS OF CONTAINMENT TREES

Node Removal

When a node \( n \) is removed from \( T_i \), we consider two cases: (1) the required operation in \( T_{i-1} \) and the requirement operation in \( T_{i+1} \).

**Case 1**: Required operations in \( T_{i-1} \) (\( i = 2...k \)).

When a node \( n \) is deleted from \( T_i \), the child nodes of \( n \) become the child nodes of \( n \)’s parent node. According to refinement subtree definition (Definition 18), a refinement subtree is defined by the root node and its child nodes. So the refinement subtree of \( n \) becomes part of the refinement subtree of \( n \)’s parent node after \( n \) is removed. Figure 3.19 illustrates this case.

![Figure 3.19: Remove node \( n \) from \( T_i \) (Node removal case 1).](image)

**Requirement 6.** After a node \( n \) is deleted from \( T_i \), the refinement subtree of \( n \) in \( T_{i-1} \) becomes part of the refinement subtree of \( n \)’s parent node.

**Case 2**: Required operations in \( T_{i+1} \) (\( i = 1...k - 1 \)).

Assume node \( m \) in \( T_{i+1} \) is the generalization node of node \( n \). If \( n \) is the root node of \( m \)’s refinement subtree as shown in Figure 3.20, then after \( n \) is deleted in \( T_i \), node \( m \) must also be removed from \( T_{i+1} \). Otherwise, if \( m \) is kept after \( n \) is removed, then there does not exist any node in \( T_i \) with the same declared space as \( m \), which violates Definition 18.

If \( n \) is not the root node of \( m \)’s refinement subtree, then the removal of \( n \) does not change the declared space of the root node or child nodes of \( m \)’s refinement subtree. So no change is required in \( T_{i+1} \).

**Requirement 7.** When node \( n \) is removed from \( T_i \), if \( n \) is the root node of a refinement subtree, then the generalization node of \( n \) must also be removed from \( T_{i+1} \).
3.4 The C-Tree

In this section, we first introduce the C-Tree definition in general, then we describe data pages and index pages in detail, and at the end we will present properties of the C-Tree.

The C-Tree is a balanced search tree of disk pages with the following distinctive features:

1. The declared spaces of pages in every level of the C-Tree form a containment tree. Side pointers of the C-Tree pages map to child pointers of the containment tree.

   **Definition 22 (The Level \( i \) Containment Tree).** The containment tree corresponding to the pages in the \( i \)th level of the C-Tree is called the **level \( i \) containment tree**.

   For convenience, we use \( \text{node}(P) \) to represent the containment tree node corresponding to page \( P \). The home space of \( P \) is the same as the home space of \( \text{node}(P) \).

2. The containment trees of all the levels in the C-Tree form a containment hierarchy.

   **Definition 23 (Parent/Child Page).** Given two adjacent levels \( i \) and \( i + 1 \) (\( i = 1..k-1 \)), assume \( P \) is a page in level \( i + 1 \) and \( C \) is a page in level \( i \), if \( \text{node}(P) \) in level \( i + 1 \) containment tree is a generalization node of \( \text{node}(C) \), then \( P \) is called the **parent page** of \( C \) and \( C \) is called a **child page** of \( P \).

   **Definition 24 (The Child Level Refinement Subtree).** The containment tree nodes corresponding to the child pages of \( P \) are a refinement subtree of \( \text{node}(P) \). We call this the **child level refinement subtree** of \( \text{node}(P) \).
3. *Index terms in page* $P$ *are organized as a refinement subtree of node*($P$), *which is called the index term refinement subtree of node*($P$).

When all the side pointers in $P$’s child pages in the one level lower in the C-Tree have been installed, the index term refinement subtree of node($P$) is the same as the child level refinement subtree of node($P$). If there is any side pointer in $P$’s child pages that has not been posted, then the index term refinement subtree of node($P$) is a generalization of the child level refinement subtree of node($P$).

4. *Side pointers in* $P$ *are stored as child nodes of the index term refinement subtree of node*($P$). The index term refinement subtree and the side pointers are well-nested.

**Definition 25.** In the C-Tree, a **side pointer** pointing to page $P$ is created when $P$ is split off from its container page, and it consists of two parts: (1) a pointer to $P$, and (2) a description of the declared space of $P$.

**Definition 26 (Level $i$ Index Entry Containment Tree).** If all side pointers in level $i$ index pages are mapped to the root nodes of their corresponding pages, the index terms in level $i$ index pages form a containment tree. We call this containment tree the **level $i$ index entry containment tree**.

As an immediate result of the above definitions, the level $i$ index entry containment tree is a refinement of the level $i$ containment tree and is a generalization of the level $i - 1$ containment tree. We will prove the correctness of this statement as a property of the C-Tree in Section 3.4.2.

5. The declared space of the root page of the C-Tree is the entire search space, defined by a true predicate.

The above features distinguish the C-Tree from other index structures. On the other hand, the C-Tree also shares many common characteristics with some other index structures, such as the B-Tree, the $hB^{π}$-Tree, the R-tree etc. It is a balanced search tree of disk pages. The leaf pages are called **data pages**. All data items are stored in the data pages. The non-leaf pages are called **index pages**. Index pages store index terms. A **child pointer** is part of an index term. It is stored in the parent page and points to the child page. Two pages are in the same **level** of the C-Tree if they are the same distance from the root page, where distance is measured by the number of child pointers on the path from the root page. Data pages are in level 1. Pages closer to the root page of
the C-Tree have a higher level. When a page $P_1$ is split off from a page $P_0$, a side pointer pointing to $P_1$ will be placed in $P_0$.

### 3.4.1 An Example of the C-Tree

Now let’s look at an example of the C-Tree in Figure 3.21. The numeric intervals next to the pages define their declared spaces. This structure is a C-Tree because, (1) the declared spaces of pages in every level of the C-Tree correspond to a containment tree. Figure 3.22 illustrates the three containment trees, (2) these containment trees form a containment hierarchy as shown in Figure 3.23, and (3) we define the index terms and side pointers as in Figure 3.24. We use solid circles to show regular containment nodes and dashed circles to represent side pointers. The page number is marked next to the node that represents it. The index terms within page $I_i$ form a refinement subtree of node $m_i$ in $T_2$. Note that the index term refinement subtree in page $R$ in Figure 3.24 is a generalization of $T_2$ in Figure 3.23, which is the refinement subtree of node $k_1$ in Figure 3.23. This happens because the side pointer from $I_1$ to $I_2$ has not been posted yet. Figure 3.25 illustrates level 2 index entry containment tree $D_2$ and the level 3 index entry containment tree $D_3$. $D_2$ in Figure 3.25 is a refinement of $T_2$ in Figure 3.21 and is the same as $T_1$. $D_3$ is a refinement of $T_3$ and is a generalization of $T_2$.

![Figure 3.21: An example of the C-Tree.](image)

### 3.4.2 Properties of the C-Tree

With the above definitions of the C-Tree, we can derive the following properties.

**Property 11.** The level $i$ index entry containment tree is a refinement of the level $i$ containment tree and is a generalization of the level $i - 1$ containment tree.

Let $T_i$ and $T_{i-1}$ be the level $i$ and level $i - 1$ containment tree respectively. Let $D_i$ be the level $i$ index entry containment tree.
Figure 3.22: The containment trees of the C-Tree shown in Figure 3.21.

Figure 3.23: The containment hierarchy corresponding to the C-Tree in Figure 3.21.

(1) $D_i$ is a refinement of $T_i$.

According to the definition of index entry containment tree (Definition 26), for every page $P$ in the $i$th level of the C-Tree, the index term refinement subtree of node($P$) is a partial subtree of $D_i$. From the definition of level $i$ containment tree (Definition 22), we know that every node $n$ in $T_i$ corresponds to a page $P$ in the $i$th level of the C-Tree. Therefore, for every node $n$ in $T_i$ ($n = \text{node}(P)$), there exists a partial subtree of $D_i$ (i.e. the index term refinement subtree of node($P$)) such that $n$ is the generalization node of this partial subtree. Thus, according to the definition of generalization/refinement (Definition 19), $T_i$ is a generalization of $D_i$ and $D_i$ is a refinement of $T_i$.

(2) $D_i$ is a generalization of $T_{i-1}$.

From item 3 of the C-Tree definition, we know that the index term refinement subtree of node($P$) is either the same as or is a generalization of the child level refinement subtree of node($P$). Therefore, for every node $m$ in the index term refinement subtree of node($P$), there exists a node or a partial subtree of $T_{i-1}$ that considers $m$ as its generalization node.

From Definition 26 (the definition of index entry containment tree), we know that for every
node $m$ in $D_i$, there is a page $P_i$ in the $i$th level of the C-Tree such that $m$ is a node in the index term refinement subtree of node($P_i$).

Combine the above two facts and we can conclude that for every node $m$ in $D_i$, there exists a node or a partial subtree in $T_i$ that considers $m$ as its generalization node. Therefore $D_i$ is a generalization of $T_{i-1}$.

**Corollary 7.** Let $T_i$ ($i = 1..k$) be the level $i$ containment tree and $D_i$ ($i = 2..k$) be the level $i$ index entry containment tree. $< T_1, D_2, T_2, D_3, ..., D_k, T_k >$ form a containment hierarchy.

Now let’s map this containment hierarchy to the C-Tree. Nodes in $T_i$ map to delimited spaces of pages in level $i$ of the C-Tree. Nodes in $D_i$ are index terms inside level $i$ index pages. Child pointers in $T_i$ correspond to side pointers in $i^{th}$ level of the C-Tree. Child pointers in $D_i$ are child pointers from index nodes to its child nodes inside level $i$ index pages. Following refinement
pointers from $T_i$ to $D_i$ is to identify the root node in level $i$ index pages. Following refinement pointers from $D_i$ to $T_{i-1}$ is to follow the pointers associated with index nodes in level $i$ index pages to child pages in level $i - 1$.

**Property 12.** Let $P$ be a parent page of $C$ in the C-Tree, the home space of $P$ contains the home space of $C$.

From the definition of parent/child page (Definition 23), node($P$) is a generalization of node($C$). According to Property 9 (the home space of a generalization node is partitioned by the home spaces of nodes in its refinement subtree), the home space of node($P$) contains the home space of node($C$). Therefore, the home space of page $P$ contains the home space of page $C$.

**Property 13.** The home spaces of pages in the same level of the C-Tree partition the entire search space.

From Property 10 (the declared spaces of root nodes of any two containment trees in a containment hierarchy are the same) and the fact that the declared space of the root page of the C-Tree is the entire search space, we know that the declared space of the root node of the level $i$ containment tree is the entire search space.

By definition of the C-Tree, pages in the same level of the C-Tree form a containment tree. The home spaces of these pages are the home spaces of the containment tree. From Property 6 (the home spaces of all nodes in a subtree rooted at node $n$ partitions declared($n$)), we can conclude that the home spaces of pages in the same level of the C-Tree partition the entire search space.

The above properties are derived from the definition of the C-Tree and they are crucial for searches in the C-Tree. In the rest of this section, we will complete the definition of the C-Tree by defining splits and consolidations of data pages and index pages of the C-Tree. After that we discuss search procedures in the C-Tree.

### 3.4.3 The C-Tree Data Pages

Data pages in the C-Tree are defined as follows:

1. A data page has a list of data entries and a list of side pointers.
2. A data entry $E$ can be added to a data page $P$ iff $E$ satisfies the predicate of home($P$).
3. When an overflowing page $P$ splits, it follows the following steps:
(a) Part of $P$’s declared space, which is decided by a split algorithm, is delegated to a new page $P'$. This space becomes the declared space of $P'$.

(b) The declared space of $P'$ can not be empty and must be properly contained (not equal to) the declared space of $P$. For any existing side pointers in $P$, its declared space should be either contained in $\text{declared}(P')$ or does not intersect $\text{declared}(P')$ at all. (Requirement 1, 2)

(c) Existing data entries and side pointers in $P$ that satisfy the predicate of $\text{declared}(P')$ are moved into $P'$.

If the declared space of $P'$ contains at least one existing side pointer in $P$, then the split of $P'$ is called a containment split.

**Definition 27.** When a page $P_1$ is split off from page $P_0$, if the declared space of $P_1$ contains at least one declared space of existing side pointers in $P_0$, the split is called a containment split.

If the total number of side pointers in an overflowing page $P$ exceeds a predefined upper bound, then the split of $P$ must be a containment split.

(d) A side pointer pointing to $P'$ is placed in the side pointer list of $P$.

4. The consolidation of an underflowing page is realized through page merges. Two pages can be merged only if

(a) One is the immediate container page of the other. (Requirement 3)

(b) The two pages have the same parent page.

When two pages $P_0$ and $P_1$ are merged, assuming $P_0$ is the immediate container page of $P_1$, the following steps need to be followed:

(a) If node($P_1$) is the root node of the refinement subtree of any node (including nodes in other containment tree and index entry containment trees), the generalization nodes of node($P_1$) needs to be deleted. (Requirement 7).

(b) The data entries and side pointers in $P_1$ are added to the data entry list and the side pointer list of $P_0$ respectively.

(c) The side pointer in $P_0$ that points to $P_1$ is removed.

(d) Page $P_1$ is deallocated.
Data page split corresponds to adding a new node in the level 1 containment tree, while data page consolidation maps to removing a node from the level 1 containment tree. To maintain the correctness of the C-Tree, the split and consolidation need to follow the requirements of operations in a single containment tree and in a containment hierarchy (i.e. Requirement 1 to 7). Among these, Requirement 1 and 2 are directly related to the data page split and Requirement 3 and 7 are related to data page consolidation. This is why these requirements are part of the above data page definition. Since the level 1 containment tree does not have any refinement, the other requirements (Requirement 4, 5, 6) are trivially satisfied. Therefore, all the requirements of node addition and removal in a containment hierarchy are followed by the data page operations in the C-Tree. So the C-Tree are kept in the well-formed state after these operations.

With the above definition in mind, now let’s look at an example. Figure 3.26 illustrates an example of data pages in the B-Tree using the C-Tree concepts. Initially, there are two data pages $P_1$ and $P_2$, as shown in Figure 3.26(a). $P_2$ is split off from $P_1$. The declared space of $P_1$ is $[0, 100)$ and the declared space of $P_2$ is $[50, 100)$.

In Figure 3.26(b), assume $P_2$ underflows after a data record $R_4$ is deleted. Assume $P_1$ and $P_2$ have the same parent page. Following the above definition, we merge $P_2$ into $P_1$. Figure 3.26(c) shows how $P_1$ looks like after the consolidation.

Figure 3.26(d) shows the two pages after a new record $R_7$ is inserted in $P_1$ in Figure 3.26(a). Assume $P_1$ overflows after the insertion. $P_3$ is split off from $P_1$. Since the declared space of $P_3$ is $[20, 100)$, which contains the $[50, 100)$, the side pointer of $P_2$ is moved to $P_3$. The two records that are in range $[20, 100)$ are also moved to $P_3$. Figure 3.26(e) shows the pages after the split.

Figure 3.26: Data pages in the B-Tree.
3.4.4 The C-Tree Index Pages

An index page $P$ in the $i$th level of the C-Tree ($i > 1$) is defined as follows:

1. $P$ stores index entries of its child pages and side pointers.

**Definition 28 (Index Entry).** Let $C$ be a page in the $k^{th}$ level of the C-Tree. The index entry of $C$ consists of (1) the node corresponding to $C$ in the level $k$ containment tree, i.e. $\text{node}(C)$, which is called the index node of $C$ in $P$, (2) the child nodes of node($C$) in the level $k$ containment tree, and (3) a pointer pointing to page $C$. The triplet (node($C$), node($C$)'s child nodes, child pointer to page $C$) is called the index entry of $C$.

The index entries and the side pointers are organized into a containment tree. The index entries form the index term refinement subtree of node($P$) and the side pointers are the child nodes of the index term refinement subtree.

2. The declared space of $P$ is the same as the declared space of the root node of index term refinement subtree stored in $P$.

3. Insertion in page $P$ is triggered by page splits in level $i - 1$. An index entry of page $C$ can be added to $P$ only if $C$ is a child page of $P$.

When the index entry of page $C$ is added to $P$, these steps need to be followed:

(a) The index node of $C$ (the declared space of $C$) and the pointer pointing to $C$ are added (the child nodes of node($C$) do not need to be posted).

(b) Assuming $C$ is split off from page $C_0$, the index node of $C$ must be added as a child node of node($C_0$) in $P$.

(c) In $P$, existing child nodes of node($C_0$) whose declared spaces are contained in declared($C$) become child nodes of node($C$).

4. When the root page $R$ of the C-Tree splits, assuming $R$ is in the $k^{th}$ level of the C-Tree, a new root page $R'$ is created, and the level $k$ containment tree, which consists of two nodes, are stored in $R'$ (Requirement 5).

5. When $P$ overflows, a subtree of the containment tree stored in $P$ (including both index nodes and side pointers) are split off to a new page $P'$. A copy of the root node of the
subtree is kept in $P$ as the side pointer to $P'$. An index node for page $P'$ is scheduled to be posted to $P$'s parent page.

Note that if we allow an arbitrary number of child nodes, it may not always be possible split an index page. An extreme scenario is that if all the non-root nodes in $P$ are child nodes of the root node, then no matter how we split $P$, the page will still be full afterwards.

In general, a k-nary tree always has a subtree whose size is between $\frac{1}{k+1}$ and $\frac{k}{k+1}$ of the entire tree size. Let $m$ and $M$ be the minimum and maximum capacity of a page. As a heuristic, we limit the number of side pointers in a C-Tree page to $\lfloor \frac{M}{m} - 1 \rfloor$ so that a containment tree node has no more than $\lfloor \frac{M}{m} - 1 \rfloor$ child nodes.

Similar to the data page split, if the number of side pointers in an overflowing page $P$ exceeds the maximum number ($\lfloor \frac{M}{m} - 1 \rfloor$), we need to force the next split to be a containment split. In other words, the declared space of the new page that is split off from $P$ should contains the declared space of at least one existing side pointer in $P$.

6. A node $n$ is deleted from page $P$ when the page that $n$ corresponds to in level $i - 1$ is merged into its container page. The child nodes of $n$, if exists, become child nodes of $n$’s parent node after $n$ is deleted.

The root node of an index page can not be deleted. This means if page $C$ in level $i - 1$ corresponds to the root node in $P$, when $C$ underflows, $C$ can not be merged into its container page. Instead, an immediate contained page of $C$ should be merged into $C$.

7. When an index page $P$ underflows, $P$ can be merged with another page $P'$ if the following conditions are satisfied:

   (a) $P$ and $P'$ have the same parent page.

   (b) $P$ is the container page of $P'$ or $P$ is an immediate contained page of $P'$.

   (c) If the index node of $P$ is the root node in $P$’s parent page, then $P$ can only be merged with an immediate contained page.

When $P$ and $P'$ are merged, without loss of generality, assuming $P'$ is the container page of $P$, the following steps need to be performed.

   (a) The index node of $P$ is deleted from $P$’s parent page.

   (b) The side pointer in $P'$ that points to $P$ is replaced with the tree structure in $P$. 
(c) \( P \) is deallocated.

8. If there is only one node in the root page of the C-Tree, then the child page this node points to becomes the new root page and the original root page is deallocated.

In the following, before we show the above definition keeps the C-Tree in a well-formed state, we look at examples of the C-Tree index pages.

We first start with the data pages. Figure 3.27 shows four data pages and the corresponding level 1 containment tree. According to Definition 28, we can derive the index entries of these data pages as shown in Figure 3.28. An index entry includes not only the corresponding containment node, but also its child nodes. With the containment node and the child nodes, one can derive the home space of a page from its index entry.

![Figure 3.27: Data pages and the level 1 containment tree.](image)

![Figure 3.28: Index entries of the data pages shown in Figure 3.27](image)

The index entries in Figure 3.28 are organized into an index term refinement subtree as shown in Figure 3.29. This index term refinement subtree is the same as the level 1 containment tree because all the side pointers in the data page level have been posted. Now assume page \( P_3 \) overflows and is split into \( P_3 \) and \( P_5 \). Before the index node of \( P_5 \) is posted to its parent page \( I_1 \). The index term refinement subtree in \( I_1 \) is a generalization of the level 1 containment tree. This is shown in Figure 3.30. Figure 3.31 illustrates the C-Tree after the index node of \( P_5 \) is posted.

Now let’s look at examples of index page split. Assume page \( I_1 \) in Figure 3.32 is full. Assume the subtree rooted at node \( a_3 \) is split off to a new page \( I_2 \). The declared space of the new page \( I_2 \)
3.4. THE C-TREE

Figure 3.29: An example of index page.

Figure 3.30: Index term refinement subtree in page $I_1$ is generalization of the level 1 containment tree.

will be the same as the declared space of its root node $a_3$. A side pointer pointing to $I_2$ is placed in $I_1$. To keep the tree structure in $I_1$ well-formed, the side pointer donated by a dashed circle must be a child node of $a_1$. Figure 3.32(b) shows the two pages after the split. A new root page $R$ is created after the split, as shown in Figure 3.33.

Now assume we want to consolidate page $P_2$. Since a consolidation must be of two pages with the same parent, $P_2$ can be merged into page $P_1$ only. But before the merge, the index node for $P_2$ must be deleted from its parent page first. This is shown in Figure 3.34. Assume after $a_2$ is deleted from $I_1$, $I_1$ underflows. Page $I_2$ will be merged into $I_1$. Again, before the merge, index node of the contained page, i.e. $I_2$, needs to be removed from its parent page first. This is illustrated in Figure 3.35(a). After the merge, $I_1$ becomes the only child of the root page $R$, so $R$ is deleted and $I_1$ becomes the new root page, which is shown in Figure 3.35(b).

To prove the above index page definition keeps the C-Tree in a well-formed state, we need to show the index page definition is consistent with the C-Tree definition given in Section 3.4, which
includes (1) the declared spaces of pages in every level of the C-Tree form a containment tree, (2) all containment trees form a containment hierarchy, (3) the index terms in an index page $P$ is a refinement subtree of node($P$), and (4) the side pointers in $P$ are child nodes of the index term refinement subtree in $P$.

The index page definition, on the other hand, specifies the following information inductively: (a) the declared space of an index page, which is the base case of a containment tree and the containment tree hierarchy, (b) the index page split and consolidation, which are the inductive steps of the containment tree and the containment tree hierarchy, (c) the index term, which is the base case of index term refinement subtree, and (d) the insertion and deletion of index nodes, which are the inductive steps on the index term refinement subtree.

In the following, we use proof by induction to show three cases respectively, case 1: the declared spaces of pages in the $i^{th}$ level form a containment tree ($i > 1$) (item 1 of the C-Tree definition), case 2: all containment trees form a containment hierarchy (item 2 of the C-Tree definition), case 3: index terms in page $P$ form an index term refinement subtree of node($P$) and
3.4. THE C-TREE

Figure 3.33: A new root page \( R \) is created after the split in Figure 3.32

the side pointers in \( P \) are the child nodes of this refinement subtree (item 3 and 4 of the C-Tree definition).

**Case 1:** The declared spaces of pages in every level of the C-Tree form a containment tree.

We use double induction in this proof, one on the level number, and the other induction on the number of pages in one level.

(1.1) Base case: level \( i = 2 \) and number of pages \( j = 1 \).

When there is only one page in level \( i \), it is trivially true that the declared space of this page forms a containment tree with one node.

(1.2) Induction on number of pages in level \( i \).

(1.2.1) Assuming the hypothesis is true when there are \( j \) pages in level \( i \), we will show that it is also true when there is \( j + 1 \) pages in level \( i \). This maps to the index page split scenario.

To add a new node in a containment tree, two requirements must be met, as discussed in Section 3.2.3. Requirement 1 is for non-containment split. And Requirement 2 is for containment split.

Assume page \( P \) is the overflowing page in level \( i \) and a new page \( P' \) is split off from \( P \). According to item 5 in the index page definition, a subtree of the containment tree stored in \( P \) is split to \( P' \). Assume \( n \) is the root node of the subtree. The declared space of \( P' \) is the same as the declared space of \( n \), which is not empty (item (1) of Requirement 1) and is contained in the declared space of \( P \) (item (2) of Requirement 1 and Requirement 2). Now let’s consider the side pointers stored in \( P \) before the split. These side pointers correspond to the child nodes of
node(P) in the level i containment tree. Since the index terms and side pointers stored in P form a well-nested containment tree before the split and the side pointers are always leaf nodes of this containment tree, we know that for any side pointer node s in P, the declared space of n must either contain the declared space of s or not intersect with the declared space of s at all, which means that, for any existing child node of node(P) in the level i containment tree, its declared space should either be contained in the declared space of the new page \( P' \) or it does not intersect the declared space of \( P' \) at all (item(3) of Requirement 1 and item(3) and (4) of Requirement 2). And it is trivially true that the side pointers moved to page \( P' \) must originally be side pointers in P (item (1) of Requirement 2).

Combine the above and we know that the index page split satisfies the requirements of node addition in a single containment tree discussed in Section 3.2.3. Therefore, the level i containment
tree remains well-nested after a level $i$ index page split.

(1.2.2) Now we will look at the index page consolidation scenario.

To maintain the well-nestedness of the level $i$ containment tree, Requirement 3 must be met. This requirement states that a node can only be merged into its parent node and the root node of the containment tree can not be deleted unless it is the only node in the tree. This is consistent with item 7 and item 8 in the index page definition. Item 7 defines that during consolidation it is always the contained page that is merged into its container page. And item 8 defines that the only scenario when the root node of the level $i$ containment tree can be deleted is when the root node of the level $i$ containment tree is the only node and that there is only one index node in its corresponding page.

(1.3) Induction on the number of levels. Assuming the hypothesis is true in level $i$ or lower, we will show that it is also true in the $i + 1$ level.

Again we will use induction on the number of pages in the $i + 1^{th}$ level.

The base case is when there is only one page in the the $i + 1^{th}$ level. This is when a new root page is created as defined in item 4. Since there is only one node in the level, it is a trivial case of containment tree.

The induction on the number of pages is the same as shown in (1.2).

From (1.1), (1.2) and (1.3), we know that according to the index page definition, the declared spaces of pages in every level of the C-Tree form a containment tree and this containment tree is maintained in well-nested state during index page splits and consolidations.

Case 2: all containment trees in a C-Tree form a containment hierarchy.

We use induction on the number of levels.

(2.1) Base case: when there is only one level in the C-Tree. Level $i = 1$. A single containment tree is a trivial case of the containment hierarchy.

(2.2) Induction on the number of levels.

Assume when there are $k$ levels in the C-Tree, all the containment trees form a containment hierarchy. Now we will show that the level $(k + 1)^{th}$ containment tree is also part of the containment hierarchy.

We use proof by induction on the number of pages in level $k + 1$.

(2.2.1) When there is only one page in level $k + 1$, this page must be the new root page created according to item 4 in the index page definition. The declared space of this page is the same as the root node stored in it, which is the declared space of the root node of level $k$ containment tree. So the level $(k + 1)$ containment tree is generalization of the level $k$ containment tree, i.e. all the
containment trees in the C-Tree form a containment hierarchy. This also satisfies Requirement 5 of node addition in a containment hierarchy, which states that when a new level $T_{k+1}$ is created, the first node in $T_{k+1}$ must be the generalization node of the root node of $T_k$.

(2.2.2) Assume the level $k + 1$ containment tree is part of the containment hierarchy when there are $j$ pages in it. Now we will show this is also true there are $j + 1$ pages in level $k + 1$. This corresponds to index page split defined in item 5 of the index page definition.

According to Requirement 4 discussed in Section 3.3.4, when a new node $n$ is added to a containment tree $T_{k+1}$ in a containment hierarchy, (1) there must be a node $t$ in $T_k$ with the same declared space as $n$ has, and (2) $n$ must be added as the child node of $t$’s original generalization node before $n$ is added.

When a new page $P'$ is split off from $P$ in level $k + 1$ in the C-Tree, the declared space of $P'$ is the same as the root node stored in $P'$, which is index node for a child page of $P'$. Therefore, there must exist a page $C$ in level $k$ with the same declared space as $P'$. So Requirement 4(1) is satisfied.

Since $P'$ is split off from $P$, page $C$ is a child page of $P$ before the split, i.e. node($P$) in level $k + 1$ containment tree is a generalization node of node($C$) before the split. When $P'$ is split off, a side pointer of $P'$ is placed in $P$ and node($P'$) becomes a child node of node($P$). So Requirement 4(2) is also satisfied.

(2.2.3) Assume the hypothesis is true when there are $j$ pages $(j > 2)$ in level $k + 1$. Now let’s look at the scenario when there is $j - 1$. This maps to index page consolidation.

To remove a node from a containment hierarchy, two requirements need to be met. They are Requirement 6 and Requirement 4.

Requirement 6 states that when a node $n$ is deleted from $T_{k+1}$, the refinement subtree of $n$ in $T_k$ becomes part of the refinement subtree of $n$’s parent node. Before an index page $P$ is deleted, the tree structure stored in $P$ is placed into its container page to replace the side pointer of $P$. After this merge, the tree structure in $P$’s container page is still well-nested and every index node has a corresponding child page. The child pages of $P$ become child pages of $P$’s container page. So the refinement subtree of node($P$) becomes the refinement subtree of the parent node of node($P$), which is consistent with Requirement 6.

Requirement 7 states that when a node $n$ is the root node of the refinement subtree of its generalization node, then when $n$ is deleted, its generalization nodes must also be deleted. For the declared space of a C-Tree index page to be the root node of the refinement subtree of its generalization node, the index node corresponding to this page must be the root node in its parent
3.4. THE C-TREE

page. Such page can not be deleted (item 6 of the index page definition). Therefore, Requirement 7 is satisfied.

**Case 3:** Index terms in page $P$ form an index term refinement subtree of node($P$) and the side pointers in $P$ are the child nodes of this refinement subtree.

(3.1) We use induction on the number of index nodes in $P$ to prove the first part of case 3.

Base case: when the number of index nodes in $P$ is 1. This index node must be the root node of $P$ because it is there when $P$ is first created and it is the last node removed from $P$. The declared space of $P$ is the same as the declared space of its root node. Therefore, this root node is a refinement subtree of node($P$).

Induction: assume when there are $i$ index nodes in $P$, they form a refinement subtree of node($P$). Now let’s consider $i + 1$ (addition of a new node) and $i - 1$ (removal of an existing from $P$).

A new node $n$ is added to $P$ when an index entry insertion occurs (item 3 of the index page definition). Assume $n$ is an index node for page $C$. Node $n$ is added in $P$ as the child node of $C$’s container page. Therefore the declared space of $n$ is not empty and it is contained in the declared space of its parent node (i.e. Requirement 1(1), (2), and Requirement 2(2) are satisfied). When $C$ is split off from its container page $C_0$, the declared spaces of side pointers in $C_0$ either do not intersect the declared space of $C$ or are contained in the declared space of $C$ (item 3 of data page definition and item 5 of index page definition). These side pointers correspond to child nodes of node($C_0$) in $P$. Therefore, the declared spaces of these child nodes either do not intersect the declared space of $n$ or are contained in the declared space of $n$ (Requirement 1(3) and Requirement 2(1), (3) and (4) are satisfied).

Since both Requirement 1 and Requirement 2 are met, we know that adding a new node in $P$ keeps the index term tree structure in $P$ well-nested. Since the declared space of $P$ is the same as the root node of this tree structure, this tree is a refinement subtree of node($P$).

Removal of an existing node from $P$ corresponds to node deletion in index page (item 6 of index page definition). When a node $n$ is deleted, its child nodes become child nodes of $n$’s parent node and the root node in $P$ can not be deleted, which is consistent with Requirement 3. Therefore, the index term refinement subtree in $P$ is well-nested after an existing node is removed.

(3.2) We will show that the side pointers in $P$ are always child nodes of its index term refinement subtree.

The only time when a new side pointer is added to $P$ is when $P$ splits as defined in item 5 of index page definition. Since an entire subtree is split off to the new page, the side pointer when
first created does not have any child. A child node can be added to a node \( n \) only when \( n \) is an index node and its corresponding page in one level lower splits. So no node will be added as child node of a side pointer. Therefore, side pointers, if exists, are always leaf nodes in the tree structure in \( P \). If there does not exist any side pointer, then the hypothesis is trivially true.

With case 3 and Corollary 7, we know that the level \( i \) containment trees (for \( i = 1 \) to \( k \)) and the level \( i \) (\( i = 2 \) to \( k \)) index entry containment tree altogether form a containment hierarchy. This is an important property for searches in the C-Tree discussed next.

### 3.4.5 Searches in the C-Tree

Searches in the C-Tree are equivalent to searches in the containment hierarchy \(< T_1, D_2, T_2, ..., D_k, T_k >\), where \( T_i \) is the level \( i \) containment tree and \( D_i \) is the level \( i \) index entry containment tree. So the search algorithm discussed in Section 3.3.3 applies to the C-Tree.

Recall that following refinement pointers from \( T_i \) to \( D_i \) is to identify the root node in a level \( i \) index page and following refinement pointers from \( D_i \) to \( T_i \) is to follow the child pointers from index nodes in level \( i \) index pages to child pages in level \( i - 1 \). Given a level \( i \) index page \( P \), \( \text{node}(P) \) is the node corresponding to \( P \) in \( T_i \). To search within the refinement subtree of \( \text{node}(P) \) is to search the index nodes in \( P \). Let \( d \) be an index node in \( D_i \). Assume \( d \) is the index node of page \( C \) in level \( i - 1 \). The refinement subtree of \( d \) in \( T_{i-1} \) includes only \( \text{node}(C) \) if all side pointers \( C \) have been installed. If there exists any side pointer in \( C \) that has not been posted, then the search of the refinement subtree need to follow the not-yet-posted side pointers and visit the corresponding pages. In general, one can tell whether a side pointer in page \( C \) has been posted or not by remembering the home space of the index node of \( C \) when visit \( C \)’s parent page and compare it with the declared spaces of the side pointers stored in \( P \). If the declared space of a side pointer intersects the home space of the index node of \( C \), then this side pointer has not been posted.

Taking possible side pointer traversal and the refinement pointers into account, the search algorithm for the C-Tree can be written as Algorithm 4. Since we have shown the correctness of searches in a containment hierarchy and a C-Tree is a containment hierarchy, the searches in a C-Tree is also correct.

This chapter has illustrated the C-Tree framework. In the next chapter, we will show how to extend the C-Tree to index spatial objects.
Algorithm 4 *Search*(Page $R$, QueryPredicate $Q$). Given the root page $R$ of a C-Tree and a query predicate $Q$, find all data pages $D$ in the C-Tree such that the home space of $D$ satisfies $Q$. Let $S$ be a stack, which is empty when the program is first initialized. Let $L$ be a list that stores the query result. $L$ is empty at the beginning.

1: let $r$ be the pointer pointing to root page $R$;
2: push $r$ to $S$;
3:
4: while $S$ is not empty do
5: \hspace{1em} $n \leftarrow$ S.pop.
6: \hspace{1em} follow pointer of $n$ to allocate page $P$; \{following refinement pointer from $D_i$ to $T_{i-1}$ if $n$ is an index node\}
7: \hspace{1em} for every side pointer $m$ in $P$ do
8: \hspace{2em} if the declared space of $m$ intersects the home space of $n$ then
9: \hspace{3em} push $m$ to $S$ \{the index node corresponding to $m$ has not been posted yet\}
10: \hspace{2em} end if
11: \hspace{1em} end for
12: \hspace{1em} if $P$ is a data page then
13: \hspace{2em} add $P$ to $L$
14: \hspace{1em} else
15: \hspace{2em} $n \leftarrow$ the root node in $P$ \{following refinement pointers from $T_i$ to $D_i$\}
16: \hspace{2em} $L_i \leftarrow$ *Search*($n$, $Q$); \{defined in Algorithm 1\}
17: \hspace{2em} push every node in $L_i$ to $S$; \{along with each node stores its home space.\}
18: \hspace{1em} end if
19: \hspace{1em} end while
20:
21: return $L$. 

Chapter 4

The Spatial C-Tree

In this chapter, we apply the framework discussed in the previous chapter to index spatial objects. The major difference between the C-Tree and the Spatial C-Tree lies in the definition of their search spaces. The search spaces in the C-Tree are defined by abstract predicates. The search spaces in the Spatial C-Tree, on the other hand, are more concrete. They are defined by point sets (regions) in the “object space”, where the spatial objects naturally locate.

This chapter is organized as follows. First, in Section 4.1, we introduce the concept of object spaces and how to use it to represent search space. Such mappings are used in the following sections to define spatial search predicates. In Section 4.2, we start with a simple version of spatial predicates. Its corresponding containment tree is given in Section 4.3. This “simple” definition is a straightforward extension of the containment concept. It can only be used in certain cases. In Section 4.4, we define the “real” spatial search predicates used in spatial C-Trees. Its corresponding containment tree is introduced in Section 4.5. In Section 4.6, we define the abstract Spatial C-Tree without specifying the detailed split and consolidation algorithms. An implementation of such spatial C-Tree using hyperplane splits is discussed in Section 4.7.

4.1 Map Object Spaces to Search Spaces

Spatial objects normally exist in a multidimensional space. We call this space the object space, to distinguish it from search spaces. Object space is a concrete natural space for spatial objects. It is a multidimensional space represented by coordinates. Search space is an abstract logical space represented by predicates. In the object space, a spatial object is a set of points. On the other hand, the spatial object is in a search space if it satisfies the search space predicate.
An example that illustrates the difference between the object space and the search space is shown in Figure 4.1. Figure 4.1(a) is an example of a 2 dimensional object space. It is denoted by $x$ and $y$ axis. In the object space, an object $o$ is described by its coordinates only. Figure 4.1(b) shows four predicates that can be used to define a search space. A search space predicate can be defined as a boolean-valued function of any attribute of the objects. The attributes include not only coordinates in the object space, but also other characteristics such as color, weight, shape etc.

![Figure 4.1: An example of object space and search space.](image)

Object space of spatial objects are defined using regions.

**Definition 29 (Region).** A spatial region is a point set in a multidimensional object space.

**Definition 30 (Containment of Regions).** A region $R_1$ contains a region $R_2$ if every point in $R_2$ is in $R_1$.

**Definition 31 (Intersection of Regions).** A region $R_1$ intersects a region $R_2$ if there exists at least one point that is in both $R_1$ and $R_2$.

### 4.2 “Simple” Spatial Predicates

In this section, we define “simple” spatial predicates. A simple spatial predicate is defined by a region called its descriptor region. An object satisfies a simple spatial predicate if the object is fully contained in the descriptor region.

**Definition 32 (Simple Spatial Predicate).** A spatial object $o$ satisfies a simple spatial predicate $Pr$ if object $o$ is fully contained in a region associated with $Pr$, which is called the Descriptor Region of $Pr$.

In Figure 4.2(a), assuming region $R_1$ is the descriptor region of a simple spatial predicate $Pr_1$, then object $O_3$ and $O_4$ do not satisfy $Pr_1$ because they are not fully contained in $R_1$. In
Figure 4.2(b), assuming region $R_1$ is the descriptor region of $Pr_1$ and $R_2$ is the descriptor region of $Pr_2$, then object $O_1$, $O_2$, $O_5$ and $O_6$ satisfy $Pr_1$ and object $O_2$ and $O_6$ satisfy $Pr_2$.

![Diagram of descriptor regions](image)

Figure 4.2: Descriptor regions of simple spatial predicates.

We have defined in the previous chapter that, for any two search spaces $S_1$ and $S_2$, $S_1$ contains $S_2$ if any object satisfying $\text{predicate}(S_2)$ also satisfies $\text{predicate}(S_1)$ (Definition 1), and $S_1$ intersects $S_2$ if there exists an object that satisfies both $\text{predicate}(S_1)$ and $\text{predicate}(S_2)$ (Definition 2). As a consequence, in the following, we discuss containment and intersection properties of simple spatial predicates and their descriptor regions.

**Property 14.** A simple spatial predicate $Pr_1$ contains a simple spatial predicate $Pr_2$ iff the descriptor region of $Pr_1$ contains the descriptor region of $Pr_2$.

We use proof by contradiction to show the correctness of both directions. Let $R_1$ be the descriptor region of $Pr_1$ and $R_2$ be the descriptor region of $Pr_2$.

1. If $Pr_1$ contains $Pr_2$, then $R_1$ contains $R_2$. Assume $R_1$ does not contain $R_2$. Then there exists at least one object $o$ (e.g. an object whose shape is exactly the same as $R_2$) such that $o$ is contained in $R_2$, but not $R_1$. According to the definition of simple spatial predicate, $o$ satisfies $Pr_2$, but not $Pr_1$. This contradicts to the fact that $Pr_1$ contains $Pr_2$. Therefore, $R_1$ must contain $R_2$.

2. If $R_1$ contains $R_2$, then $Pr_1$ contains $Pr_2$. Assume $Pr_1$ does not contain $Pr_2$. Then there exists at least one object $o$ such that $o$ satisfies $Pr_2$, but not $Pr_1$. This means object $o$ is contained in $R_2$, but not $R_1$. This contradicts to the fact that $R_1$ contains $R_2$. Therefore the contradiction assumption is wrong. $Pr_1$ contains $Pr_2$.

We can show in a similar fashion as the above proof that a simple spatial predicate intersects another simple spatial predicate iff their descriptor regions intersect. This is stated in Property 15.

**Property 15.** Two simple spatial predicates intersect iff their descriptor regions intersect.
4.3 Simple Spatial Containment Tree

A “simple” spatial containment tree is a containment tree in which declared spaces of nodes are defined using simple spatial predicates. To maintain the well-nested property of the containment tree, the declared space of a node in the simple spatial containment tree should be contained by the declared spaces of its ancestor nodes, and not intersect the declared space of any of its sibling nodes. This means, according to Property 14 and Property 15, that for every node in a simple spatial containment tree, its descriptor region must (1) be contained in the descriptor regions of its ancestor nodes, and (2) not intersect the descriptor region of any of its sibling nodes.

Definition 33 (Simple Spatial Containment Tree). A simple spatial containment tree is a containment tree such that the declared spaces of containment tree nodes are defined using simple spatial predicates.

Figure 4.3 is an example of simple spatial containment tree. $R_i$s are the descriptor regions of the declared spaces of containment tree nodes $n_i$s. The descriptor region of a node is stored along with the node in the containment tree. The declared space of node $n_i$ includes all objects that are completely contained in region $R_i$. The home space of $n_i$ includes those objects that are in the declared space of $n_i$ but are not in the declared space of any of $n_i$’s child nodes. So object $O_1$ and $O_6$ are in the home space of $n_0$. Object $O_2$ is in the home space of node $n_2$, object $O_3$ is in the home space of $n_3$, object $O_4$ is in the home space of $n_4$, and object $O_5$ is in the home space of $n_5$.

Figure 4.3: Simple spatial containment tree.

In simple spatial containment tree, searching for a node whose home space contains an object $o$ is to look for the node whose descriptor region is the smallest one that contains $o$. Let $n$ be the desired node. Since $o$ satisfies home$(n)$, $o$ is not in the descriptor region of any child node or descendent node of $n$. The descriptor regions of all the other nodes either contain or are disjoint from the descriptor region of $n$. So $n$ has the smallest descriptor region that contains $o$. 
A simple spatial containment tree does not apply to certain complicated scenarios. For example, in the containment tree shown in Figure 4.3, we can not create a new child node of node $n_0$ to contain object $o_1$ because its descriptor region would intersect the descriptor regions of already existing children of $n_0$, which does not satisfy the definition of a simple spatial containment tree. As an example, if we create a new child with descriptor region $R_6$, as shown in Figure 4.4(a), then $R_6$ intersects $R_1$ and $R_2$. This means that the declared space of the new child intersects the declared spaces of $n_1$ and $n_2$. Then the tree structure would not be a proper containment tree anymore.

If we allow two children to have intersecting descriptor regions in a simple spatial containment tree, e.g. $R_1$ and $R_6$ in Figure 4.4(b), then for objects in the intersection, say $o_7$, there is no way to tell which child it is in. Ideally we would like to express that objects like $o_7$ belong to the declared space of $n_1$, not the declared space of the new child node.

It is for the above reasons, we introduce the “regular” spatial search predicate in the next Section, which has more expressive power than the simple spatial search predicate. The regular spatial search predicate can also be used to describe home spaces in a containment tree, which can not be done using simple spatial predicates.

![Figure 4.4: $R_6$ can not be the descriptor region of a new node in the simple spatial containment tree in Figure 4.3.](image)

### 4.4 Spatial Search Predicates

By observing the example in Figure 4.4 and home spaces in a simple spatial containment tree, we can see that we need not only a descriptor region to describe what objects are included in the search space but also a way to point out that certain regions inside the descriptor region have been taken out from the search space already. We use *preceding regions* to describe such exclusion.
Definition 34 (Preceding Region). A preceding region of a spatial search predicate $S$ is a region that intersects but does not contains the descriptor region of $S$ such that objects that are fully contained in this region have been assigned to other search spaces, hence do not belong to $S$.

Definition 35 (Spatial Search Predicate). A spatial predicate $Pr$ consists of a descriptor region and a set of preceding regions such that an object $o$ satisfies $Pr$, if $o$ is fully contained in the descriptor region of $Pr$ and is not completely contained in any of the preceding regions of $Pr$.

For convenience, we use $[R, (a_1, \ldots, a_k)]$ to represent a spatial search predicate whose descriptor region is $R$ and preceding regions are $a_1, \ldots, a_k$.

An example of spatial search predicate is illustrated in Figure 4.5. Let $Pr$ be a spatial search predicate. Assume $R_1$ is the descriptor region of $Pr$ and $R_2$ is a preceding region of $Pr$. Object $O_1$ satisfies $Pr$ because it is fully contained in $R_1$, but not in $R_2$. Object $O_2$ does not satisfy $Pr$ because $O_2$ is not inside the descriptor region of $Pr$. Object $O_3$ does not satisfy $Pr$ either although it is inside $R_1$. This is because $O_3$ is fully contained by $R_2$ which is a preceding region of $Pr$.

![Figure 4.5: Spatial search predicate.](image)

With the above definition in mind, let’s look at the properties of spatial search predicate. We first look at properties related to predicate containment. And then we examine properties regarding predicate intersections.

In the following properties, spatial search predicates $S_1$ and $S_2$ are defined as $S_1 = [R_1, (a_1, \ldots, a_k)]$ and $S_2 = [R_2, (b_1, \ldots, b_t)]$.

Property 16. If $S_1$ contains $S_2$, then $R_1$ must contain $R_2$.

Without loss of generality, assume a preceding region of a spatial search predicate is not the same as its descriptor region. We use proof by contradiction to show the correctness of Property 16.

Assume $R_1$ does not contain $R_2$. Then there could exist at least one object $o$ that is fully contained in $R_2$ and not in preceding regions $b_1\ldots b_t$, but not $R_1$, hence satisfies $S_2$, but not $S_1$. 


This contradicts to the fact that $S_1$ contains $S_2$. Therefore, the assumption is wrong. $R_1$ must contain $R_2$.

**Property 17.** If $S_1$ contains $S_2$, then the intersection of $R_2$ and any of $S_1$’s preceding regions must be contained in one of $S_2$’s preceding regions, i.e. $\forall a_i, \exists b_j, (a_i \cap R_2) \subseteq b_j (i = 1..k, j = 1...t)$.

We use proof by contradiction. Let $R$ be the intersection of $a_i$ and $R_2$. Assume $R$ is not fully contained in any of $b_j$s.

$R$ is inside a preceding region of $S_1$, therefore $R$ does not satisfy $S_1$. $R$ is inside $R_2$ and is not fully contained in any of $S_2$’s preceding regions. So $R$ satisfies $S_2$. This contradicts to the fact that $S_1$ contains $S_2$, therefore the assumption is wrong. $R$ must be fully contained in at least one of $S_2$’s preceding regions.

**Property 18.** If $R_1$ contains $R_2$ and the intersection of $R_2$ and each one of $S_1$’s preceding regions is fully contained in at least one preceding region of $S_2$, then $S_1$ contains $S_2$. That is, if $R_2 \subset R_1$ and $\forall a_i, \exists b_j$ such that $(a_i \cap R_2) \subset b_j$, then $S_2 \subset S_1$.

According to Definition 35, for any object $o$ that is in $S_2$, $o$ is fully contained $R_2$ and is not fully contained in any preceding region of $S_2$. (1) Since $R_2$ is contained in $R_1$, object $o$ is contained in $R_1$. (2) For any preceding region $a_i$ of $S_1$, $a_i$ does not contain $o$ because the intersection of $a_i$ and $R_2$ is contained in $b_j$ and $b_j$ does not fully contain $o$. Therefore, from (1) and (2), object $o$ is also in $S_1$. Any object that is in $S_2$ is also in $S_1$. Therefore, $S_1$ contains $S_2$.

**Property 19.** If two spatial search predicates intersect, then their descriptor regions must intersect, i.e. if $S_1 \cap S_2 \neq \phi$, then $R_1 \cap R_2 \neq \phi$.

We also use proof by contradiction. Assume $R_1 \cap R_2 = \phi$, then there doesn’t exist any object that is inside both $R_1$ and $R_2$. Hence, there doesn’t exist any object that satisfies both $S_1$ and $S_2$, which contradicts to the fact that $S_1$ intersects $S_2$. Therefore, their descriptor regions must intersect.

When two descriptor regions intersect, their corresponding spatial search predicates do not necessarily intersect. This is stated in the following property.

**Property 20.** If $R_1$ intersects $R_2$, then $S_1$ intersects $S_2$ iff the intersection of $R_1$ and $R_2$ is not contained in any preceding region of $S_1$ or $S_2$. In other words, if $R_1 \cap R_2 \neq \phi$, then $S_1 \cap S_2 \neq \phi$ iff $\forall a_i \forall b_j (R_1 \cap R_2) \not\subseteq a_i$ and $(R_1 \cap R_2) \not\subseteq b_j$. 
(1) If \( S_1 \) intersects \( S_2 \), then the intersection of \( R_1 \) and \( R_2 \) is not contained in any preceding region of \( S_1 \) or \( S_2 \).

We use proof by contradiction to show the correctness of this direction. Without loss of generality, assume the intersection of \( R_1 \) and \( R_2 \) is contained in a preceding region of \( S_1 \). Let’s call this region \( a_i \). Any object that is inside both \( R_1 \) and \( R_2 \) is fully contained in \( a_i \). According to Definition 35, for any object \( o \) to satisfy both \( S_1 \) and \( S_2 \), \( o \) must be in both \( R_1 \) and \( R_2 \), hence inside the intersection of \( R_1 \) and \( R_2 \). Since the intersection is contained in \( a_i \), there does not any object inside the intersection that satisfy \( S_1 \). Hence \( S_1 \) and \( S_2 \) do not intersect, which contradicts to the fact that \( S_1 \) intersects \( S_2 \). Therefore, the assumption is wrong. The intersection of \( R_1 \) and \( R_2 \) is not contained in any preceding region of \( S_1 \) or \( S_2 \).

(2) If the intersection of \( R_1 \) and \( R_2 \) is not contained in any preceding region of \( S_1 \) or \( S_2 \), then \( S_1 \) intersects \( S_2 \).

Since the intersection of \( R_1 \) and \( R_2 \) is not contained in any preceding region of \( S_1 \) and \( S_2 \), there exists at least one object \( o \) that is fully contained in the intersect, but is not contained in any preceding region of \( S_1 \) or \( S_2 \). According to the definition of spatial search predicate \( o \) is inside both \( S_1 \) and \( S_2 \), hence \( S_1 \) intersects \( S_2 \).

4.5 Spatial Containment Tree

Definition 36 (Spatial Containment Tree). A spatial containment tree is a containment tree such that the declared space of the containment tree nodes are defined using spatial predicates.

With the well-nested property of a containment tree and the properties of spatial search predicates discussed in the previous section, we have the following requirements of a spatial containment tree.

When a new node \( n \) is created,

1. the descriptor region of declared\((n)\) must be contained in the descriptor region of \( n \)'s container node,

2. the declared space of \( n \) “inherits” the preceding regions of \( n \)'s container node, i.e. the preceding regions of \( n \)'s container node that intersect the descriptor region of declared\((n)\) are also preceding regions of \( n \).

3. if a sibling node \( m \) is created earlier than \( n \) and the descriptor region of declared\((m)\) intersects the descriptor region of declared\((n)\), then the descriptor region of declared\((m)\) must
be a preceding region of declared($n$).

According to Property 18, the first two items in the above requirement guarantee that the declared space of $n$ is contained in the declared space of its container node. According to the definition of preceding regions (Definition 18), the last item in the above requirement guarantees that the declared spaces of sibling nodes are disjoint.

Figure 4.6 illustrates an example of a spatial containment tree. Node $n_0$ is the root node. It has three child nodes, $n_1$, $n_2$ and $n_3$ in creation order. Node $n_1$ is created earlier than $n_2$, so its descriptor region is a preceding region of $n_2$. Similarly, $R_1$ and $R_2$ are preceding regions of $n_3$. Node $n_4$ and $n_5$ are child nodes of $n_3$. The descriptor region of $n_4$ does not intersect any of $n_3$’s preceding region. So $n_4$ does not “inherit” any preceding region from its container node. Node $n_5$, on the other hand, intersects $R_2$. Therefore, $n_5$ “inherits” its preceding region $R_2$ from its parent node $n_3$. Node $n_4$ is created earlier than $n_5$ and their descriptor regions intersect, so $R_4$ is a preceding region of $n_5$.

![Spatial Containment Tree Diagram](image)

Figure 4.6: A containment tree using spatial search predicate.

Note that in the running example in Figure 4.6, the preceding regions of a node are descriptor regions of either its sibling nodes or the sibling nodes of its ancestors. By the time we visit a node from the root node, we already know the preceding regions of its parent node and the preceding regions of its ancestor nodes. If we have a way to identify the preceding regions coming from its sibling nodes, we do not need to store the preceding regions with a node anymore. For this purpose, we introduce “sibling edges” in the spatial containment tree, which organize all sibling nodes into a list ordered by their creation order. When a new node $n$ is added to the spatial containment tree, if $n$ is not the only child node, then a sibling edge pointing from the last node in the sibling list to $n$ is added.
With the above in mind, we now define the implementation of a spatial containment tree as follows:

**Implementation of Spatial Containment Tree** A spatial containment tree is implemented as follows:

1. Every node in a spatial containment tree stores its descriptor region. Without lose of generality, we assume the descriptor region of every node is unique.

2. The descriptor region of a node \( n \) is fully contained in the descriptor region of \( n \)'s parent node.

3. The child nodes of a node \( n \) is organized into an ordered list. This ordered list is linked together via sibling links.

4. A node \( n \) has one child link that points from \( n \) to the first node in \( n \)'s child list.

5. The preceding regions of a node \( n \) include (a) the preceding regions of \( n \)'s parent node that intersect the descriptor region of \( n \), and (b) the descriptor regions of \( n \)'s predecessor sibling nodes that intersect the descriptor region of \( n \). For convenience, we define the predecessor nodes of a node \( n \) as follows:

**Definition 37 (Predecessor Sibling).** A node \( m \) is a predecessor of its sibling node \( n \) if \( m \) is created earlier than \( n \), and one of the following condition is satisfied:

- (a) The descriptor region of \( m \) intersects the descriptor region of \( n \), and the descriptor region of \( m \) does not contain the descriptor region of \( n \) and vice versa.

- (b) Node \( m \) is a predecessor sibling of a node \( t \) and node \( t \) is a predecessor sibling of node \( n \). In this case, the descriptor region of \( n \) may contain the descriptor region of \( m \).

Note that in Definition 37, we do not consider the case where the descriptor region of \( m \) contains the descriptor region of \( n \). If node \( m \) is created first and its descriptor region contains the one of \( n \), then \( n \) cannot be a sibling of \( m \). It will be a descendant of \( m \).

Let's look at an example of predecessor siblings. Assume region \( R_1, R_2, R_3 \) and \( R_4 \) in Figure 4.7 are the descriptor regions of four sibling nodes \( n_1, n_2, n_3 \) and \( n_4 \) in creation order. Node \( n_1 \) is a predecessor sibling of \( n_2 \) because \( n_1 \) is created earlier than \( n_2 \) and \( R_1 \).
intersects \( R_2 \) (part (a) of Definition 37). Similarly \( n_2 \) is a predecessor sibling of \( n_3 \), and \( n_3 \) is a predecessor sibling of \( n_4 \). Note that \( n_1 \) is also predecessor siblings of \( n_3 \) and \( n_4 \). This is due of part (b) of Definition 37. Node \( n_1 \) is a predecessor sibling of \( n_3 \) because \( n_1 \) is a predecessor sibling of \( n_2 \) and \( n_2 \) is a predecessor sibling of \( n_3 \). Similarly \( n_1 \) is a predecessor sibling of \( n_4 \) because \( n_1 \) is a predecessor sibling of \( n_2 \) and \( n_2 \) is a predecessor sibling of \( n_3 \), which is a predecessor sibling of \( n_4 \).

![Figure 4.7: Descriptor regions of predecessor siblings.](image)

**Definition 38 (Predecessor).** A node \( m \) is a **predecessor** of a node \( n \) if \( m \) is a predecessor sibling of \( n \) or if \( m \) is a predecessor sibling of an ancestor node of \( n \). Node \( n \) is called a **successor** of node \( m \).

If \( m \) is a predecessor node of \( n \) and their descriptor regions intersect, then the descriptor region of \( m \) is a preceding region of \( n \).

Using Definition 4.5, the containment tree in the running example in Figure 4.6 can be organized into a spatial containment tree as shown in Figure 4.8. Node \( n_0 \) has three children \( n_1, n_2 \) and \( n_3 \). They are linked together via sibling links. Node \( n_0 \) has only one child link that points from \( n_0 \) to \( n_1 \), which is the first node in the list. Similarly, there is a child link from \( n_3 \) to \( n_4 \) and a sibling link from \( n_4 \) to \( n_5 \). As an example, let’s look at the preceding regions of node \( n_5 \). Assume \( n_0 \) does not have any preceding region. Node \( n_1, n_2 \) and \( n_4 \) are predecessors of \( n_5 \). These three nodes are all created earlier than \( n_5 \). The descriptor regions of \( n_2 \) and \( n_4 \) intersect the descriptor region of \( n_5 \). Node \( n_1 \) is a predecessor node of \( n_2 \), which is a predecessor of \( n_5 \). Therefore, \( n_1 \) is also a predecessor of \( n_5 \). Among the predecessors of \( n_5 \), \( n_2 \) and \( n_4 \) are the ones whose descriptor region intersects the descriptor region of \( n_5 \). So the preceding regions of \( n_5 \) are \( R_2 \) and \( R_4 \).

Compared with the example in Figure 4.8 and the simple spatial containment tree in Figure 4.3, we can see that the simple spatial containment tree is a spatial containmenttree that happens to have no predecessor. This is because descriptor regions of sibling nodes in the simple spatial containment tree are all disjoint, hence no predecessor. The spatial containment tree is a generalization of the simple spatial containment tree with more flexible declared space definition.
In the following, we will show that the structure defined in Definition 4.5 is actually a well-nested containment tree.

**Property 21.** The spatial containment tree implementation (Definition 4.5) defines a well-nested containment tree.

We show the correctness of Property 21 in two steps.

**Step 1:** For any two nodes $a$ and $b$ in a spatial containment tree, if $a$ is the parent node of $b$, then the declared space of $a$ contains the declared space of $b$.

1.1) According to item (2) in Definition 4.5, the descriptor region of $a$ contains the descriptor region of $b$.

1.2) According to item (5) in Definition 4.5, the preceding regions of $a$ that intersect the descriptor region of $b$ are preceding regions of $b$. That means the intersection of any of $a$’s preceding region and the descriptor region of $b$ is contained in at least one preceding region of $b$.

From (1.1), (1.2) and Property 18, the declared space of $a$ contains the declared space of $b$.

**Step 2:** If $a$ and $b$ are sibling nodes in a spatial containment tree, then the declared space of $a$ and the declared space of $b$ do not intersect.

Without loss of generality, assume node $a$ precedes node $b$ in the child list of their parent node.

2.1) If the descriptor region of $a$ does not intersect the descriptor region of $b$, then according to Property 19, the declared space of $a$ does not intersect the declared space of $b$.

2.2) If the descriptor region of $a$ intersects the descriptor region of $b$, then according to item (5) in Definition 4.5, the descriptor region of $a$ is a preceding region of the declared space of $b$. Therefore the intersection of the descriptor regions of $a$ and $b$ are contained in a preceding region of $b$ (i.e. the descriptor region of $a$). According to Property 20, the declared space of $a$ does not intersect the declared space of $b$. 
From (2.1) and (2.2), the declared spaces of sibling nodes do not intersect. Hence a spatial containment tree is a well-nested containment tree.

**Property 22.** In a spatial containment tree, when a new node \( n \) is added as the child node of \( m \) (with \( n \)'s descriptor region being fully contained in the descriptor region of \( m \)), node \( m \) is the only node whose home space changes due to the addition of \( n \) and the home space of \( n \) does not intersect the home space of any node except the home space of \( m \) before \( n \) is added.

(1) We first show that the declared space of \( n \) is contained in the declared space of \( m \).

The descriptor region of \( n \) is contained in the descriptor region of \( m \). The predecessor nodes of \( m \) are also predecessor nodes of \( n \), i.e. the preceding regions of \( m \) that intersect \( n \) are also preceding regions of \( n \). From Property 18, the declared space of \( n \) is contained in the declared space of \( m \), i.e. declared\((n) \subset \) declared\((m)\).

(2) The home space of \( n \) is contained in the home space of \( m \) before \( n \) is created.

Assume that before \( n \) is added, \( m_1...m_k \) are child nodes of \( m \) and the home space of \( m \) is (declared\((m) - (\cup \text{ declared}(m_i)))\).

(2.1) If \( n \) is added as a leaf node, then \( m_1...m_k \) are all sibling nodes of \( n \). The descriptor regions of those \( m_i \)'s that intersect the descriptor region of \( n \) are preceding regions of \( n \). Therefore, the declared space of \( n \) does not intersect the declared space of any of its sibling node \( m_i \). That is, declared\((n) \cap \) declared\((m_i) = \phi \). Therefore, declared\((n) \subset (\text{declared}(m) - (\cup \text{ declared}(m_i)))\).

Since \( n \) is leaf node, it does not have any child. Hence declared\((n) = \text{home}(n) \). We have home\((n) \subset (\text{declared}(m) - (\cup \text{ declared}(m_i)))\).

(2.2) Node \( n \) is added as the parent node of some existing node. Without loss of generality, assume node \( m_1...m_i \) are sibling nodes of \( n \) and node \( m_{i+1}...m_k \) are child nodes of \( n \).

(2.2.1) Using the same idea as in (2.1), we can show that the declared space of \( n \) does not intersect the declared space of \( m_1...m_i \). Therefore, declared\((n) \subset (\text{declared}(m) - (\cup_{j=1,...i} \text{declared}(m_j)))\).

(2.2.2) The home space of \( n \) is declared\((n) - \cup_{j=i+1...k} \text{declared}(m_j)\).

Combine (2.2.1) and (2.2.2), we have (declared\((n) - \cup_{j=i+1...k} \text{declared}(m_j)) \subset (\text{declared}(m) - (\cup_{j=1...k} \text{declared}(m_j))). That is, home\((n) \subset (\text{declared}(m) - (\cup_{j=1...k} \text{declared}(m_j)))\).

From the above, we can conclude that the home space of the newly added node \( n \) is part of the original home space of its parent node \( m \).

(3) Now we will show that node \( m \) is the only node whose home space has changed due to the addition of \( m \).
The declared space of all existing nodes stay the same.

The descriptor regions of all existing node are not changed. Node \( n \) is added as the last one in the child node list of \( m \). So \( n \) is not a predecessor of any other node, i.e. there is no addition of predecessor node.

The nodes whose relative locations might have been possibly changed are \( m_{i+1}\ldots m_k \) (as in (2.2)). Node \( m_1\ldots m_i \) were sibling nodes of \( m_{i+1}\ldots m_k \) and now become sibling nodes of their parent node \( n \). According to the definition of predecessor node (Definition 38), if one of \( m_1\ldots m_i \) was a predecessor node of any of \( m_{i+1}\ldots m_k \), then after \( n \) is added, it still remains as predecessor of that node. Therefore, the declared spaces of \( m_{i+1}\ldots m_k \) also remains the same.

(3.2) Node \( m \) is the only node whose delegated space has changed, because it is the only node whose child node list has been changed (with addition of node \( n \)).

Combine (3.1) and (3.2), we know that the home spaces of all nodes except \( m \) remain the same after \( n \) is added and from Step (2), these home spaces doe not intersect the home space of \( n \). Therefore, Property 22 is correct.

Property 22 is important for our spatial C-Tree discussion later on, because it shows that, if a node in the spatial containment tree maps to an index entry of a page in the spatial C-Tree, then adding a new page \( P \) and its corresponding index node \( n \) does not require objects in existing pages to be moved except those that were in the container page of \( P \).

The difference between a spatial containment tree and a regular containment tree lies in the declared spaces of its nodes. In a regular containment tree, the declared space of a node is stored within the node. In the spatial containment tree, on the other hand, only the descriptor region of a node is stored within the node. The declared space of a spatial containment tree node is derived from its parent node and some of its sibling nodes.

Next we define the concepts of full path and minimum path of a spatial containment tree for convenience.

**Definition 39 (Full Path).** Given a spatial containment tree rooted at node \( n_0 \), for any node \( n_i \) in the tree, there is a unique path from \( n_0 \) to \( n_i \) via either child pointers or side pointers. This path is called the full path from \( n_0 \) to \( n_i \).

**Definition 40 (Minimum Path).** Given a spatial containment tree rooted at node \( n_0 \), for any node \( n_i \) in the tree, the minimum path from \( n_0 \) to \( n_i \) is the minimum set of nodes in the full path from \( n_0 \) to \( n_i \) that store preceding regions of \( n_i \).

That is, the full path from \( n_0 \) to \( n_i \) excluding (1) the nodes that are not predecessors of \( n_i \).
except $n_0$, (2) the predecessor nodes of $n_i$ whose descriptor regions do not intersect the one of $n_i$, and (3) the intersecting predecessor nodes whose descriptor regions are contained by at least one other intersecting predecessor node of $n_i$.

In the spatial containment tree in Figure 4.8, the full path from $n_0$ to $n_5$ and the minimum path from $n_0$ to $n_5$ are shown in Figure 4.9(a) and (b) respectively.

![Spatial Containment Tree Diagram](image-url)

Figure 4.9: The full path from $n_0$ to $n_5$ and the minimum path from $n_0$ to $n_5$.

**Property 23.** Given a minimum path from $n_0$ to $n_i$, the preceding regions of $n_i$ are the preceding regions of $n_0$ and the descriptor regions of nodes in the minimum path from $n_0$ to $n_i$ except $n_0$.

We show the correctness of Property 23 in two steps. We first show that the preceding regions of $n_i$ are the preceding regions of $n_0$, the intersecting descriptor regions of its ancestors’ predecessor sibling nodes, and the intersecting descriptor region of $n_i$’s own predecessor sibling nodes. We then show that the contributing sibling nodes of $n_i$’s ancestors are the nodes in the minimum path from $n_0$ to $n_i$.

**Step 1:** The preceding regions of $n_i$ include the preceding regions of $n_0$, the descriptor regions of $n_i$’s ancestors’ predecessor sibling nodes that intersect the descriptor region of $n_i$, and the intersecting descriptor region of $n_i$’s predecessor sibling nodes.

Assume $P_1 \ldots P_k$ are ancestor nodes of $n_i$. Node $P_j$ is $j$ child links away from the root node $n_0$, with $P_1$ being a child node of $n_0$ and $P_k$ being the parent node of $n_i$. This is illustrated in Figure 4.10.

We use induction on $j$ (the distance from $n_0$ to $P_j$) to show that the preceding regions of $n_i$ are the preceding regions of $P_j$, the descriptor regions of $P_j$’s predecessor sibling nodes ($t = j + 1 \ldots k$) and the descriptor regions of $n_i$’s predecessor sibling nodes that intersect the descriptor region of $n_i$. 


CHAPTER 4. THE SPATIAL C-TREE

Figure 4.10: The full path from $n_0$ to $n_i$

Base case: When $j = k$, this is trivially true.

Node $P_k$ is the parent node of $n_i$, according to item (5) of Definition 36, the preceding regions of $n_i$ include (a) the preceding regions of $P_k$ that intersect the descriptor region of $n_i$, and (b) the descriptor regions of $n_i$’s predecessor sibling nodes that intersect the descriptor region of $n_i$.

Induction: Assume that when $j = m$, the preceding regions of $n_i$ include (a) the preceding regions of $P_m$, (b) the descriptor regions of $P_t$’s predecessor sibling nodes that intersect the descriptor region of $n_i$ (for $t = m$ to $k$), and (c) the descriptor regions of $n_i$’s predecessor sibling nodes, we show it is also true when $j = (m - 1)$ ($m > 1$).

Node $P_{m-1}$ is the parent node of node $P_m$. According to item (5) of Definition 36, the preceding region of $P_m$ includes (i) the preceding regions of $P_{m-1}$, and (ii) the descriptor regions of $P_m$’s predecessor sibling nodes that intersect the descriptor region of $P_m$. Replace item (a) in the induction assumption with (i) and (ii), we have that the preceding regions of $n_i$ include (a) the preceding regions of $P_{m-1}$, (b) the descriptor regions of $P_t$’s predecessor sibling nodes (for $t = m - 1$ to $k$) that intersect the descriptor region of $n_i$, and (c) the descriptor region of $n_i$’s predecessor sibling nodes, i.e. the induction is true for $j = (m - 1)$.

**Step 2**: All the above nodes whose descriptor regions contribute to the preceding region of $n_i$ are in the minimum path from $n_0$ to $n_i$.

First, according to item (4) of Definition 36 and the definition of full path (Definition 39), all predecessor sibling nodes of $P_j$ are in the full path from $n_0$ to $n_i$. If a node (other than $n_0$ and $n_i$) is in the full path from $n_0$ to $n_i$, then it is either an ancestor of $n_i$ or a predecessor sibling of an ancestor of $n_i$.

Second, according to Definition 40, if the descriptor region of a node $a$ intersects the descriptor
region of \( n_i \) and \( a \) is not a parent node of \( n_i \), then \( a \) is in the minimum path from \( n_0 \) to \( n_i \). Therefore, all the predecessor sibling nodes of \( P_i \) whose descriptor regions intersect the descriptor region of \( n_i \) are in the minimum path from \( n_0 \) to \( n_i \). Also, there does not exist any node \( a \) in the minimum path from \( n_0 \) to \( n_i \) such that \( a \) is not a predecessor sibling node of \( n_0 \) or \( P_j \).

Combine the above, and we can conclude that the preceding regions of \( n_i \) are the preceding regions of \( n_0 \) and the descriptor regions of nodes in the minimum path from \( n_0 \) to \( n_i \) except \( n_0 \) and \( n_i \).

With Property 23, we know that, given the declared space of an ancestor of \( n_i \), to calculate the declared space of \( n_i \) is to identify the minimum path from the ancestor node to \( n_i \).

The main difference between a spatial containment tree and a regular containment tree is that the declared space of a regular containment tree node is stored within the node, while a spatial containment tree node stores its descriptor region only and its declared space is derived from an ancestor node and the minimum path from this ancestor node. Such difference explains the following variations of containment tree operations.

**Adding a node** \( n \) to a spatial containment tree: in general, the declared space of the new node should not intersect the declared space of existing nodes, i.e. it should either be contained or contains or disjoint from other nodes’ declared spaces. The same applies to the spatial containment tree with the following additional requirement:

When a new node \( n \) is added to a spatial containment tree,

1. There is only one node \( R \) in the spatial containment tree whose declared space contains the descriptor region of \( n \). Node \( n \) should be added as a child of such node \( R \).

2. Node \( n \) is added as the last node in its sibling list.

3. If a sibling node \( m \) is not a predecessor of \( n \) and the descriptor region of \( m \) is contained in the descriptor region of \( n \), then after \( n \) is added, \( m \) should become a child node of \( n \). Such child nodes of \( n \) are called **descended node**. If there are more than one descended node, their order in the new sibling list should be the same before \( n \) is added.

Figure 4.11 illustrated an example of node addition in a spatial containment tree. Figure 4.11(a) shows a spatial containment tree and the descriptor regions of its nodes. In Figure 4.11(b), a new node \( n_6 \) is added with descriptor region \( R_6 \). Although \( R_6 \) is contained in both \( R_1 \) and \( R_2 \), \( n_6 \) is a child node of \( n_1 \), not \( n_2 \), because the declared space of \( n_1 \) contains \( R_6 \) while the declared space of \( n_2 \) does not. After \( n_6 \) is added to the end of the sibling list after \( n_5 \), node \( n_3 \) and \( n_4 \)
are identified as “descended node” because they are not predecessors of \( n_6 \) and their descriptor regions are contained in \( R_6 \). Node \( n_3 \) is a predecessor sibling of \( n_4 \) both before and after \( n_6 \) is added.

![Diagram](image)

**Figure 4.11:** A new node \( n_6 \) is added to the spatial containment tree. Node \( n_3 \) and \( n_4 \) are descended nodes.

Note that if \( n_3 \) and \( n_4 \) are not moved to be child nodes of \( n_6 \), the result is still a valid spatial containment tree, shown in Figure 4.12. This does not mean that given a new node, there are more than one way to add it. The declared space of \( n_6 \) in Figure 4.11(b) is different from the declared space of \( n_6 \) in Figure 4.12. The former one contains the declared spaces of \( n_3 \) and \( n_4 \) while the latter one does not.

For any given node (with a fixed declared space), there is only one way to add it to a containment tree. The reason there are two variations of adding \( n_6 \) in this running example is because it is the “descriptor region” not the “declared space” of \( n_6 \) that was given. Identifying descended nodes is part of the process to define the declared space of \( n_6 \). Having descended nodes corresponds to “containment split” (Definition 27) in data pages.

![Diagram](image)

**Figure 4.12:** Node \( n_6 \) is added as a sibling node of \( n_3, n_4 \) and \( n_5 \).

**Deleting a node** \( n \) from a spatial containment is possible only if the descriptor region of \( n \) does not serve as a preceding region of any of sibling nodes. If the descriptor region of \( n \) is a preceding region of a node \( m \), then removing \( n \) would change the declared space of \( m \), which by
definition should remain the same once \( m \) is created.

**Query** algorithms of a regular containment tree still applies to a spatial containment tree.

**Containment hierarchy** of spatial containment trees also differs from the regular containment hierarchy. This is because a node in a spatial containment tree does not carry all the information needed to describe its declared space anymore. Its declared space may rely on other nodes.

Let’s look at an example illustrated in Figure 4.13 and Figure 4.14. Figure 4.13(a) shows a spatial containment tree \( T_1 \). The descriptor regions of the nodes in \( T_1 \) are shown in Figure 4.13(b). The spatial containment tree \( T_2 \) in Figure 4.14(a) is not a generalization of \( T_1 \), because the declared space of \( n_3 \) in \( T_2 \) is “fully contained in \( R_3 \)” which does not correspond to the declared space of any node in \( T_1 \). The declared space of \( n_3 \) in \( T_1 \) is “fully contained in \( R_3 \), but not completely contained in \( R_1 \) and \( R_2 \)”.

![Figure 4.13](image1)

Figure 4.13: A spatial containment tree and its descriptor regions.

![Figure 4.14](image2)

Figure 4.14: Generalization of the spatial containment tree in Figure 4.13.

\( T_3 \) in Figure 4.14 is a generalization of \( T_1 \) with \( n_0 \) and \( n_3 \) being two generalization nodes. Note that node \( n_1 \) and \( n_2 \) are present in \( T_3 \) to describe the declared space of \( n_3 \). They are not “real” generalization node. They are added to a generalization merely to describe the preceding regions of other nodes. We call such nodes **ghost nodes**. A “real” generalization node is called a **real node** to be distinguished from ghost nodes. Dashed circles are used to represent ghost nodes.
Given a spatial containment hierarchy < $T_1$...$T_k$ > with $T_i$ being a refinement of $T_{i+1}$, we define the following three concepts.

**Definition 41 (Corresponding Node).** Given two spatial containment tree $T_i$ and $T_j$ with $T_i$ being a refinement of $T_j$, node $m$ in $T_i$ and node $n$ in $T_j$ are called corresponding nodes if their descriptor regions are the same.

Every node in $T_j$ must have a corresponding node in its refinement $T_i$. For two nodes $a$ and $b$ in $T_j$, if $a$’s corresponding node in $T_i$ is a predecessor of $b$’s corresponding node $T_i$, then $a$ must be a predecessor of $b$ in $T_j$.

**Definition 42 (Real Node).** A node $n$ in $T_i$ ($i = 2...k$) is called a real node if the declared space of $n$ is the same as the declared space of its corresponding node in $T_1$.

Nodes in $T_1$ are all real nodes. Side pointers are also real nodes.

**Definition 43 (Ghost Node).** A node $n$ in $T_i$ is called a ghost node if its declared space is not the same as the declared space of its corresponding node in $T_1$.

Ghost nodes are present merely as predecessor nodes of real nodes. If a node $n$ in $T_i$ is a ghost node, then its corresponding node in a generalization of $T_i$ must also be a ghost node. A ghost node can not be a generalization node because its declared space is not the same as its corresponding node. (Definition 18).

A ghost node is present not only when it is added to describe the declared space of a generalization node, but also when nodes are deleted from a spatial containment tree. If a node $n$ is a predecessor sibling of a node $m$ and their descriptor regions intersect, then removing $n$ from the spatial containment tree would change the declared space of $m$. Therefore, when it is time to delete node $n$, instead of removing the node completely, we change node $n$ from a real node to a ghost node. Such change marks $n$ as being deleted and maintains the declared space of $m$ to be the same.

### 4.6 Spatial C-Tree

The Spatial C-Tree is an instance of the C-Tree and it is designed to index spatial objects.

**Definition 44 (Spatial C-Tree).** The Spatial C-Tree is a C-Tree such that
1. Its level i containment trees and level i index entry containment trees are spatial containment trees.

2. The descriptor region of a node in a spatial containment tree is a rectangle.

Figure 4.15 illustrates an example of a spatial C-Tree. \(P_i\)'s are data pages. We use \(I_i\)'s to indicate index pages. \(R\) is the root page of the C-Tree. Figure 4.16(a) shows the spatial layout of the data page descriptor regions. Rectangle \(R_i\) is the descriptor region of data page \(P_i\). Figure 4.16(b) is the level 1 containment tree of the spatial C-Tree shown in Figure 4.15.

Now let’s look at declared spaces of the pages in Figure 4.15. The descriptor region of the root page \(R\) is \(R_1\). It does not have any preceding region. So the predicate of the declared space of page \(R\) is “completely contained in \(R_1\)”. This is also the declared space of page \(I_1\). Page \(I_2\) is split off from \(I_1\). Node \(n_2\) is a predecessor sibling of node \(n_3\) in \(I_1\). So the declared space of \(I_2\) is “completely contained in \(R_3\) but not completely contained in \(R_2\)”. Similarly, the declared space predicate of \(I_3\) is “completely contained in \(R_6\) but not completely contained in \(R_2\) or \(R_5\)”. Now coming to data pages. The descriptor region of page \(P_i\) is \(R_i\). Page \(P_1\), \(P_2\), \(P_4\) and \(P_5\) do not any
Figure 4.16: (a) Spatial layout of the descriptor regions of the data pages in Figure 4.15. (b) The level 1 containment tree of the Spatial C-Tree in Figure 4.15.

preceding region. The preceding region of $P_3$ is $R_2$. Page $P_6$ also inherits $R_2$ as its preceding region from its container page $P_3$. $R_5$ is also a preceding region of $P_6$ because it is $P_5$ is a sibling of $P_6$ and $R_5$ intersects $R_6$. The preceding region of $P_7$ is $R_5$. Therefore, object $O_1$ shown in Figure 4.17 should be stored in page $P_2$ and object $O_2$ should be in page $P_5$.

Figure 4.17: Object $O_1$ and $O_2$ are in page $P_2$ and $P_5$ in Figure 4.15 respectively.

Note that in a general C-Tree, we define the index entry of a page $C$ in the $k^{th}$ level of the C-Tree as (1) $C$’s corresponding node in level $k$ containment tree, and (2) the child nodes of this node (Definition 28). In a spatial C-Tree, an index entry requires not only the above information, but also a minimum path to describe the declared space of $C$, because the node corresponding to page $C$ does not store the declared space of page $C$ anymore. Instead it only stores the descriptor region of $C$. It is the declared space of an ancestor node, the minimum path and the descriptor region all together that decide the declared space of $C$. Therefore, when the index entry for page $C$ is posted to its parent page, both its index node and the nodes on the minimum path from an ancestor node $n$ should be posted and the declared space of $n$ should be known in the parent page.

In the rest of this section, we discuss the posting algorithm of the Spatial C-Tree. We do not discuss the splitting algorithms here as different implementations of the Spatial C-Tree have
4.6. **SPATIAL C-TREE**

different splitting algorithms. Their posting algorithms, on the other hand, all follow the discussion below.

In general, there are three posting scenarios: (1) post the index entry for a new data page, (2) create a new root page and post the index entries for the old root page and its new sibling page to the new root page when the root page splits, (3) post the index entry of a non-root index page to its parent page.

(1) To post the index entry for a new data page $C_1$, assuming $C_1$ is split off from page $C$, we need to first identify the index node of the container page $C$ in its parent page $P$, and then post the index node of the new page $C_1$ to the end of the child node list of $C$’s index node. Figure 4.18 illustrates this posting. Figure 4.18(a) shows the Spatial C-Tree before $C$ splits. Figure 4.18(b) shows the Spatial C-Tree after the index node of the new page $C_1$ is posted.

![Figure 4.18: Post index entry for the new data page $C_1$.](image)

(2) Figure 4.19 shows the scenario when the existing root page $R_0$ splits into $R_0$ and $R_1$ and a new root page $R_{new}$ is created. In Figure 4.19(a), root page $R_0$ is full. The subtree rooted at $n$ is identified as the subtree to be split off to a new page $R_1$. In Figure 4.19(b), a new root page $R_{new}$ is created. Node $n_0$ and $n$ in $R_{new}$ are the index nodes for page $R_0$ and $R_1$ respectively. Note that in addition to $n_0$ and $n$, there are also two ghost nodes in $R_{new}$. They are posted to illustrate the declared space of node $n$ in $R_{new}$. These nodes are in the minimum path from $n_0$ to
n in Figure 4.19(a).

To summarize, when a new root page $R_{\text{new}}$ is created, the index node for the existing root page $R_0$, which is also the root node $n_0$ in $R_0$, is posted to $R_{\text{new}}$ as the root node in $R_{\text{new}}$. Let $n$ be the root node of the subtree that is split off to a new page $R_1$ from $R_0$. All nodes in the minimum path from $n_0$ to $n$ are also posted to $R_{\text{new}}$ as the index entry for $R_1$.

![Figure 4.19: A new root page $R_{\text{new}}$ is created due to the split of the existing root page $R$.](image)

(3) Posting the index entry of a non-root index page is more complicated than the above two scenarios due to the fact that two index entries (minimum paths) may share same ghost nodes. For example, let’s look at the containment tree in page $C$ in Figure 4.20(a). The descriptor region of $n_i$ is rectangle $R_i$ in Figure 4.20(b). Assume page $C$ first splits, the subtree rooted at $n_6$ is split off. To post the index entry of the new page, the minimum path from $n_1$ to $n_6$ needs to be posted. Figure 4.21(b) shows the parent page $P$ after the posting. Now assume page $C$ splits again. This time, it is the subtree rooted at $n_8$ that is split off. So the minimum path from $n_1$ to $n_8$ needs to be posted, which includes $n_2$ and $n_7$. Since $n_2$ has been previously posted, we should merge the minimum path from $n_1$ to $n_8$ into the existing subtree of $n_1$ in page $P$. Figure 4.22(b) shows page $P$ after the merge.

In addition to the common ghost node like $n_2$ described above, a new index entry needs to be
merged into the containment tree in the parent page $P$ in order to maintain the relative position of index nodes in the containment hierarchy. In the running example, assume page $C$ splits the third time. The subtree rooted at node $n_4$ is split off, as illustrated in Figure 4.23(a). Node $n_3$ and $n_4$ need to be posted to $P$ because they are in the minimum path from $n_1$ to $n_4$. Both of them have child nodes that have been posted to $P$ already. So we need to make sure those child nodes also appear as child nodes of $n_3$ and $n_4$ in page $P$ to maintain the well-formed containment hierarchy. Figure 4.23(b) shows page $P$ after the minimum path from $n_1$ to $n_4$ is posted.

Such merge is not needed in the first two posting scenarios because (1) to post the index entry
of a data page, one only needs to post one single node instead of a minimum path, and (2) when index entries are posted to a new root page $R_{\text{new}}$, there is no existing index entry in $R_{\text{new}}$.

In the following, we discuss how to merge a minimum path into the existing containment tree in the parent page. For convenience, we use the notation illustrated in Figure 4.24 in the following discussion. Page $C$ is the index page that splits. Page $P$ is the parent page of $C$. This is the page where the new index entry will be posted to. Node $n_i$s are the nodes in the containment tree in $C$. Node $n_1$ is the root node in $C$. Node $m_1$ is the index node of $C$ in page $P$. Node $m_i$s are nodes in the subtree rooted at $m_1$ in $P$. Node $m_i$ in $P$ has the same descriptor region as node $n_i$ in $C$.

The merge is based on the fact that (i) the nodes in the subtree rooted at $m_1$ in $P$ is a subset of the nodes in the subtree rooted at $n_1$ in $C$, (ii) given two nodes $m_i$ and $m_j$ in $P$, $m_i$ is an ancestor of $m_j$ iff $n_i$ is an ancestor of $n_j$ in $C$, (iii) if $m_i$ precedes $m_j$ in $P$, then either $n_i$ precedes $n_j$ or
n_i precedes an ancestor of n_j.

Assume we want to post the minimum path from n_1 to n_k to P. During the merge, we use the full path from n_1 to n_k as auxiliary information to match the relative position of existing nodes in P and the nodes in the minimum path from n_1 to n_k. We use pointer f to the current node in the full path and pointer p to keep track of the current node in the subtree rooted at m_1 in P. At the beginning, f points to n_1 in the full path and p points to node m_1 in P.

According to the relative position of f and p, there are three different cases: 1) p matches one of the nodes in the full path, 2) p is in the subtree of a node in the full path, and 3) p is preceded by a node in the full path. In the running example in Figure 4.24, assume we want to post the minimum path from n_1 to n_6. The full path from n_1 to n_6 is n_1, n_2, n_3, n_4, n_5 and n_6. Then node m_2 is an example of case 1) because m_2 corresponds to node n_2 in the full path. Node m_7 and m_8 belong to case 2) because they are in the subtree rooted at n_3. Node m_{10} belongs to case 3) because m_{10} is preceded by the nodes (n_2, n_3 and n_4) in the full path. Now let’s discuss how to identify each case and what to post.

Case 1): The node that p points to corresponds to a node in the full path.

This case can be identified when the descriptor regions of f and p are the same. This means that the node corresponding to f already exists in the parent page, either as a ghost node or a real node. If f points to n_k, then p must be a ghost node, change p from ghost node to real node and the posting procedure is done. Otherwise, if f is not the last node in the full path, then f moves onto the next node in the full path and p moves in the same direction as f, either to its child node or its sibling node.

For example, in the running example in Figure 4.24. Assume we want to post the minimum path from n_1 to n_7 to page P, which includes node n_1, n_2 and n_7. At the beginning, pointer p points to m_1 in P and f points to n_1 in the full path. The descriptor regions of m_1 and n_1 are the same. Since n_1 is not the last node in the full path, we move onto the next node. The next node on the full path is a child node of n_1, we move both f and p point to their child nodes, i.e. f points to n_2 and p points to m_2. Again the descriptor regions of m_2 and n_2 are the same. Pointer f moves to node n_3 and p moves to node m_7. Their descriptor regions are the same and n_7 is the last node in the full path. Therefore, we change node m_7 from ghost node to real node and the posting is done. Node m_7 exists in P because it is in the minimum path from n_1 to n_8, hence was posted when n_8 is posted to P.

Case 2): The node that p points to is in the subtree of a node in the full path.

In this case, the descriptor region of p must be contained in the descriptor region of f. We
remember the node that $p$ points to (let’s called it $m_i$) and iterate through its sibling list until we find a sibling node $m_j$ whose descriptor region is not contained in the descriptor region of $f$. That is, the descriptor regions of the sibling nodes between $m_i$ and $m_j$ are all contained in the descriptor region of $f$.

If $f$ is in the minimum path (must move to sibling node), then the node $n$ that $f$ points to is posted as the parent node of all the nodes between $m_i$ and $m_j$ (including $m_i$ excluding $m_j$) and their subtrees. Node $m_j$ becomes a sibling node of $n$. And then we move onto the next node. Pointer $f$ points to the original sibling node of $n$ in the full path and $p$ points to $m_j$.

When $f$ is not in the minimum path, if the next node in the full path is a sibling of the current node, then $f$ is moved to point to the next node and $p$ points to $m_j$; otherwise if the next node in the full path is a child node of $f$, then $f$ points to the child node and $p$ stays the same (i.e. points to $m_i$).

For example, in Figure 4.24, assume we want to post the minimum path from $n_1$ to $n_4$, which includes $n_1$, $n_2$, $n_3$ and $n_4$. Assume $f$ points to $n_3$ and $p$ points to $m_7$. Since the descriptor region of $m_7$ is contained in the descriptor region of $n_3$, we remember $m_7$ and iterate through the sibling list of $m_7$ until we find $m_{10}$ whose descriptor region is not contained by $n_3$. Since $n_3$ is in the minimum path, $n_3$ is posted as the parent node of $m_7$ and $m_8$. After that, $f$ points to $n_4$ and $p$ points to $m_{10}$. Figure 4.25(a) shows page $P$ and the full path before $n_3$ is merged into the containment tree in $P$ and Figure 4.25(b) illustrates the scenario after the merge.

![Diagram](https://via.placeholder.com/150)

**Figure 4.25**: Merge node $n_3$ to parent page $P$.

**Case 3)**: The node that $p$ points to is preceded by the current node in the full path that $f$ points to.

This is when the descriptor region of $f$ neither contains nor equals to the descriptor region of
4.7 \textit{PAGE SPLITS IN THE SPATIAL C-TREE}

In this case, either $p$ corresponds to or is contained by a subsequent node in the full path, which requires the next node in the full path is a sibling of the current node, or $p$ is preceded by all the sibling nodes of $f$ in the full path.

If the next node in the full path is a child node of $f$, then $f$ can not be in the minimum path. Post all the subsequent nodes that are in the minimum path as preceding siblings of the node that $p$ points to. If the next node in the full path is sibling node, then move $f$ to point to the next sibling node and $p$ stays the same. Before $f$ is moved, if $f$ is in the minimum path, post $f$ as a sibling node before $p$.

Using the example in Figure 4.24, assume we want to post the minimum path from $n_1$ to $n_6$, pointer $p$ points to $m_{10}$ and $f$ points to $n_4$. We can tell that the descriptor region of $m_{10}$ is not contained nor equals to the descriptor region of $n_4$, so it falls into case 3). The next node in the full path is $n_5$, which is a child node of $n_4$. So we examine all the nodes in the full path from $n_4$ to $n_6$ and post those that are in the minimum path from $n_1$ to $n_6$, which is $n_6$ in this example. Node $n_6$ is added as a preceding sibling of $m_{10}$. Figure 4.26 illustrates page $P$ after $n_6$ is merged.

![Figure 4.26: Merge node $n_6$ to parent page $P$.](image)

The above posting algorithm is summarized in Algorithm 5.

4.7 \textit{Page Splits in the Spatial C-Tree}

In an index tree, page splitting algorithms are closely related to its query performance. Due to different characteristics of the data, e.g. data distribution pattern, data density etc, splitting algorithms can be tuned to achieve better indexing performance. Same applies to the Spatial C-Tree. In this section, we focus on the general rules that apply to all Spatial C-Tree page splits and guarantee the correctness of the C-Tree. At the end, we introduce two variations of data page split.
Algorithm 5 Post(Page C, Page C₁ IndexPage P). Page C₁ is split off from C. Page P is the parent page of C and C₁. This algorithm posts the index entry of C₁ to P. P is null if C is the root page.

1: if C₁ is a data page then
2:     n ← a new containment node that contains the descriptor region of C₁;
3:     L ← a list of side pointers in C that are split off to C₁;
4:     PostDataPageEntry(n, L, P);
5: else
6:     n₁ ← the root node in C;
7:     nₖ the side pointer to C₁;
8:     FP ← the full path from n₁ to nₖ in C;
9:     MP ← the minimum path from n₁ to nₖ in C;
10:    f ← the root node of FP; {i.e. n₁}
11:    if P is null then
12:        allocate a new page for P; {P is the new root page.}
13:        post MP to P;
14:        exit;
15:    else
16:        p ← the index node of C in P.
17:        PostIndexPageEntry(nₖ, P, f, p);
18:    end if
19: end if

Algorithm 6 PostDataPageEntry(IndexNode n, List L, IndexPage P) Node n is the index node of a data page C₁ which is split off from page C. L is a list of side pointers that are split off from C to C₁. P is the parent page of C and C₁.

1: n₁ ← index node of C in P. {n₁ can be found by searching for the node in P whose home space contains the descriptor region of n.}
2: remove the nodes in L from the child list of n₁;
3: Add n to the end of the child list of n₁;
4: add nodes in L as child list of n;
Algorithm 7 PostIndexPageEntry(IndexNode \( n_k \), IndexPage \( P \), Pointer \( f \), Pointer \( p \))

Node \( n_k \) is the index node of an index page \( C_1 \) which is split off from page \( C \). \( P \) is the parent page of \( C \) and \( C_1 \). Pointer \( f \) points to a node in the full path from \( n_1 \) to \( n_k \). Initially \( f \) is \( n_1 \). Pointer \( p \) points to a node in \( P \). Initially \( p \) points to the index node of \( C \) in \( P \).

1: \textbf{if} the descriptor region of \( p \) is the same as the descriptor region of \( f \) \textbf{then}
2: \{ — case 1 — \}
3: MergeCase1(\( n_k \), \( P \), \( f \), \( p \));
4: \textbf{else}
5: \textbf{if} the descriptor region of \( p \) is contained in the descriptor region of \( f \) \textbf{then}
6: \{ — case 2 — \}
7: MergeCase2(\( n_k \), \( P \), \( f \), \( p \));
8: \textbf{else}
9: \{ — \( p \).descriptor_region \( \subseteq \) \( f \).descriptor_region — \}
10: \{ — case 3 — \}
11: MergeCase3(\( n_k \), \( P \), \( f \), \( p \));
12: \textbf{end if}
13: \textbf{end if}

Algorithm 8 MergeCase1(IndexNode \( n_k \), IndexPage \( P \), Pointer \( f \), Pointer \( p \))

1: \textbf{if} \( f \) points to \( n_k \) \textbf{then}
2: change the node \( p \) points to from ghost node to real node,
3: change the page pointer of the node that \( p \) points to to \( C_1 \),
4: exit.
5: \textbf{end if}
6: \{ — move onto the next node — \}
7: \textbf{if} next node in the full path is a sibling of \( f \) \textbf{then}
8: \( f \leftarrow f \).sibling;
9: \( p \leftarrow p \).sibling;
10: \textbf{else}
11: \( f \leftarrow f \).child;
12: \( p \leftarrow p \).child;
13: \textbf{end if}
14: PostIndexPageEntry(\( n_k \), \( P \), \( f \), \( p \));
Algorithm 9  \textit{MergeCase2}(IndexNode }n_k\textit{, IndexPage }P\textit{, Pointer }f\textit{, Pointer }p\textit{)}

1: \{ – iterate through the sibling list of }p\textit{ and find all nodes contained in }f\textit{ — } \}
2: \( s \leftarrow p; \)
3: \textbf{while} \( s\)’s descriptor region is contained in \( f\)’s descriptor region and \( s \) is not null \textbf{do}
4: \( l \leftarrow s; \)
5: \( s \leftarrow s.\text{ sibling}; \)
6: \textbf{end while}
7: \textbf{if} \( f \) is in the minimum path \textbf{then}
8: \{ – add all }p\textit{’s immediate siblings that are contained in }f\textit{ as children of }f\textit{ – } \}
9: connect the previous node visited before }p\textit{ to }f\textit{;
10: \( f.\text{child} \leftarrow p, \)
11: \( l.\text{ sibling} \leftarrow \text{null}; \)
12: \( f.\text{ sibling} \leftarrow s; \)
13: \textbf{end if}
14: \textbf{if} next node in the full path is }f\textit{’s sibling \textbf{then}
15: \( f \leftarrow f.\text{ sibling}; \)
16: \( p \leftarrow s; \)
17: else
18: \( f \leftarrow f.\text{ child}; \{ p \text{ stays the same} \}
19: \textbf{end if}
20: \text{PostIndexPageEntry}(n_k, P, f, p);
4.7. PAGE SPLITS IN THE SPATIO-TEMPORAL C-TREE

Algorithm 10 MergeCase3(IndexNode \( n_k \), IndexPage \( P \), Pointer \( f \), Pointer \( p \))

1: \( \text{if} \) next node in the full path is \( f \)'s child \( \text{then} \)
2: \hspace{1em} post all subsequent nodes that in the minimum path;
3: \hspace{1em} \( n_k \).sibling \( \leftarrow p; \)
4: \hspace{1em} \text{exit;}
5: \text{else}
6: \hspace{1em} \text{if} \( f \) is in the minimum path \( \text{then} \)
7: \hspace{1em} \hspace{1em} add \( f \) before \( p; \)
8: \hspace{1em} \hspace{1em} \text{end if}
9: \hspace{1em} \hspace{1em} \( f \).sibling \( \leftarrow p; \)
10: \hspace{1em} \hspace{1em} \( f \leftarrow \) next node in the full path; \{ \( p \) stays the same. \}
11: \hspace{1em} \text{PostIndexPageEntry}(n_k, P, f, p);
12: \hspace{1em} \text{end if}

4.7.1 Data Page Split

When a new data page \( P' \) is split off from an existing data page \( P \), the following rules must be met.

1. The descriptor region of the new page \( P' \) must be completely contained in the descriptor region of \( P \).

2. All existing objects in \( P \) that are fully contained in the descriptor region of \( P' \) must be moved to the new page \( P' \).

3. Let \( [S_1... S_t] \) be the side pointer list in page \( P \). For each existing side pointer \( S_i \) in \( P \), if the descriptor region of \( S_i \) is completely contained in \( R' \), and \( S_i \) is not a preceding sibling of any side pointer \( S_j \) \((j > i)\) that is not contained in \( R' \), then \( S_i \) is moved to page \( P' \) as a side pointer and is removed from the side pointer list in \( P \). If any side pointer is moved from \( P \) to \( P' \), then the split becomes a containment split. If more than one side pointers are moved to \( P' \), then their order in the sibling list in \( P \) is also the order in \( P' \).

4. A side pointer pointing to \( P' \) is added to the end of the side pointer list of \( P \). It consists of the descriptor region of \( P' \) and a pointer to page \( P' \).

The first rule is to guarantee that the declared space of the new page \( P' \) is contained in the declared space of its container page \( P \). (Property 17, Property 18). The second rule keeps ob-
jects in the page whose home space contains them. Rule 3 assures that predecessor siblings are maintained correctly.

We use data page $P$ shown in Figure 4.27(b) as an example to clarify the above rules. Page $P$ stores a list of spatial objects $o_i$s and a list of side pointers pointing to page $P_1$, $P_2$ and $P_3$. The descriptor region of page $P$ is $R$ and the descriptor regions of page $P_i$s are $R_i$s. Figure 4.27(a) shows the spatial layout of the descriptor regions. The shadowed rectangles represent spatial objects. Index page $I$ in Figure 4.27(c) is the parent page of $P$, $P_1$, $P_2$ and $P_3$.

![Figure 4.27: Data page in a Spatial C-Tree.](image)

Now assume page $P$ is full and a new page $P_4$ with descriptor region $R_4$ is split off from $P$. There are many ways to decide $R_4$. In the following, we show two variations and how they differ according to rule 3 above.

Figure 4.28(a) shows a straightforward split. $R_4$ does not contain the descriptor region of any existing side pointers in $P$. Therefore, no side pointer will be moved to the new page $P_4$ and this is not a containment split. A side pointer to $P_4$ is added to the end of the sibling list in $P$. Those objects completely contained in $R_4$ are moved to $P_4$. Figure 4.28(b) illustrates page $P$ and $P_4$ after the split. In Figure 4.28(c), an index node corresponding to $P_4$ is posted to its parent page $I$.

![Figure 4.28: Data page split in a Spatial C-Tree.](image)
4.7. PAGE Splits IN THE SPATIAL C-TREE

Figure 4.29(a) shows an example of containment split. The descriptor region \( R_4 \) of the new page contains \( R_1 \) and \( R_3 \). According to the splitting rule, the side pointer to \( P_1 \) can not be moved to \( P_4 \) because \( R_1 \) is a preceding region of \( R_3 \) and \( R_3 \) is not contained in \( R_4 \). The side pointer to \( P_2 \), on the other hand, is “contained” in the new page and will be moved to \( P_4 \). This is because \( R_2 \) is fully contained in \( R_4 \) and there is no other side pointer in \( P \) that considers it as a preceding sibling. Figure 4.29(b) shows the two pages after the split. And Figure 4.29(c) illustrates the parent page \( I \) after the index node for \( P_4 \) is posted.

The above splitting rules does not specify how the descriptor region of a new data page is chosen. We will give two examples on the selection of the descriptor region at the end of this section.

4.7.2 Index Page Split

When an index page \( P \) is split into \( P \) and \( P' \), a subtree rooted at node \( m \) is moved from \( P \) to \( P' \) and the following rules apply:

1. Node \( m \) must be a real node in \( P \).
2. Node \( m \) must have at least \( k \) (\( k > 1 \)) nodes in its subtree, where \( k \) is a predefined threshold.
3. After \( m \) and its subtree is moved to \( P' \), a pointer pointing to page \( P' \) is stored with \( m \) and node \( m \) becomes a side pointer of \( P' \).
4. The minimum path from the root node in \( P \) to node \( m \) is posted to \( P' \)’s parent page.

In the first rule, we require that the root node of the subtree to be split off is a real node. This is because only real node carries complete information of its declared space. Adding a new
index page in the \( i^{th} \) level of the Spatial C-Tree is actually adding a new generalization node in the level \( i \) containment tree. A generalization node “represents” its refinement subtree in lower level containment trees. So the generalization node needs to have the same declared space as its corresponding node in lower level.

In the second rule above, we require node \( m \) not to be a leave node. This guarantees that after the subtree is moved to \( P' \), page \( P \) has more space available. Otherwise, if \( m \) is a leave node, after \( m \) is moved to \( P' \), the node itself still stays as a side pointer and there will be no space gain in the original page \( P \). This defeats the original purpose of page split. Therefore, we require that the subtree rooted at \( m \) is a non-trivial subtree.

Figure 4.30 illustrates an example of index page split. Page \( P, P_1 \) and \( I \) are all index pages. Page \( I \) is the parent page of \( P \) and \( P_1 \). \( P_1 \) was split off from \( P \). We use double circle to represent a side pointer node. Node \( m_1 \) in page \( I \) is a corresponding node of \( n_1 \) in page \( P \). \( R_i \)s are descriptor regions of node \( n_i \)s and \( m_i \)s. Figure 4.30(b) shows the spatial layout of some of these descriptor regions.

![Diagram](image.png)

Figure 4.30: Index pages in a Spatial C-Tree.

Now assume page \( P \) is full. We will split the subtree rooted at \( n_4 \) to a new page \( P_2 \). Node \( n_4 \) in page \( P \) becomes a side pointer of \( P_2 \). The minimum path from \( n_1 \) to \( n_4 \) includes \( n_1, n_2, n_3 \) and \( n_4 \). So \( n_2 \) and \( n_4 \) will be posted to the parent page \( I \) (\( n_3 \) already exists). Figure 4.31 shows the pages after the split.
When the subtree that is split off contains side pointers, the split becomes a containment split. In the running example in Figure 4.30, if the subtree rooted at $n_8$ instead of $n_4$ is split off to the new page $P_2$, then the side pointer to $P_1$ will be moved to $P_2$ and this split is a containment split. Figure 4.32 shows the pages after the split.

![Containment split of index pages in a Spatial C-Tree.](image)

The selection of the subtree to be split off follows the following heuristics, assuming the subtree is rooted at node $m$. 

![Index page split in a Spatial C-Tree.](image)
1. Fewer nodes in the minimum path from the root node to \( m \) means less ghost nodes to be posted, hence less overhead in the parent page.

2. The subtree to be split off should have at least \( k \) nodes, where \( k \) is a tuning threshold. If there doesn’t exist such a subtree, then a subtree whose size is the closest to \( k \) should be chosen.

3. Depth first traversal to identify a desired subtree normally means less time to find a desired split candidate node and such node probably has less nodes in its minimum path.

4.7.3 Variations of Data Page Split

The Hyperplane Approach

The hyperplane approach, as the name indicates, is to split data pages by a hyperplane. That is the descriptor region of the newly created page and the descriptor region of the container page differ by one edge only. When a data page \( P \) overflows, we need to split it into \( P \) and \( P' \) such that both pages have at least \( k \) objects. We perform a plane sweep along both directions of each dimension to look for a split candidate such that the descriptor region of the new page is the smallest.

Figure 4.33 illustrates an example of the hyperplane split. Assume page \( P_1 \) is a data page with descriptor region \( R_1 \). Page \( P_2 \), \( P_3 \), \( P_4 \) and \( P_5 \) are split off \( P_1 \) in order. \( R_i \)'s are descriptor region of \( P_i \)'s. Node \( n_i \)'s are containment tree nodes corresponding to page \( P_i \)'s. When \( P_2 \) is split off from \( P_1 \), we identify the splitting hyperplane to be \( x = x_1 \) in Figure 4.33(a). Therefore, \( R_2 \) in Figure 4.33(b) becomes the descriptor region of the new page \( P_2 \). Similarly \( R_3 \) in Figure 4.33(c) is the descriptor region of \( P_3 \). Since \( R_2 \) intersects \( R_3 \), node \( n_2 \) (the containment tree node corresponding to \( P_2 \)) becomes a predecessor sibling of \( n_3 \). When \( P_4 \) is created, hyperplane \( y = y_2 \) in Figure 4.33(d) is decided to be the splitting hyperplane. Note that the splitting dimension and direction of \( P_4 \) are the same as \( P_3 \), and there is no other sibling in between \( P_3 \) and \( P_4 \), therefore this is a containment split. When \( P_5 \) with descriptor region \( R_5 \) in Figure 4.33(e) is split off from \( P_1 \), although \( R_5 \) contains \( R_2 \), it is not a containment split. Instead, node \( n_2 \) is a predecessor sibling of \( n_5 \). This is because \( n_2 \) is a predecessor sibling of \( n_4 \) and \( n_4 \) is a predecessor sibling of \( n_5 \). According to part (b) of the predecessor sibling definition (Definition 37), \( n_2 \) is a predecessor sibling of \( n_5 \). Figure 4.33(f) shows the corresponding containment tree.

One quality of hyperplane split is that in a \( n \) dimensional space the descriptor regions of the newly created page and its container page share \( n - 1 \) edges. This means that in the corresponding
containment tree, the descriptor regions of a node and its parent differ by one edge. Therefore, we can optimize the containment tree to be more space efficient to store only the different edge in the child node, which includes the split dimension, the split value, and the split direction. Figure 4.34(a) shows the optimized version of the containment tree in Figure 4.33(f).

Storing the “three part” splitting information (dimension, value and direction) is also how $kd$-tree organizes information. So for people who are familiar with $kd$-tree, the containment tree with hyperplane split can be rewrite into a $kd$-tree like structure. Note that this structure is not the same as a $kd$-tree, because (1) the semantics of space represented by a node has changed (without using the spatial predicate, a $kd$-tree can only index point data), and (2) “extra” information needs to be stored in the $kd$-tree. That is which child of a $kd$-tree node is the “real” child of the containment tree. Figure 4.34(b) shows an example that represents the containment tree in Figure 4.34(a). We
use a thicker line to indicate the containment tree child node of a \( kd \)-tree node.

Using hyperplane split, as in any other splitting algorithm, one needs to decide how many objects to split off to the new page. In our experiments, we used a \( kd \)-tree like structure. The result shows that when \( k \) (the number of objects to be split off) is close to half of the page size, page utilization and query performance such as intersection queries are better than using a smaller value.

There are two benefits to use the hyperplane split.

First, the containment trees are “compact” in the sense that in most containment tree nodes less amount of information needs to be stored than a rectangle. So the “fanout” of an index page is higher than storing a rectangle as descriptor region in each containment tree node.

Second, the number of nodes in the minimum path from a node \( n \) to a node \( m \) has an upper bound. Even though there can be arbitrary number of nodes in the full path from \( n \) to \( m \), in a \( d \)-dimensional space, there are at most \( 2d \) nodes in the minimum path, one node per splitting dimension and splitting direction. This is because using hyperplane split, after every at node \( 2d \) split of the same node, the newly created descriptor region must contain the descriptor region(s) of at least one existing sibling node (but the declared space of the new node may not contain the declared space of any sibling). According to part (3) of minimum path definition (Definition 40), when two nodes \( n_1 \) and \( n_2 \) are both intersecting predecessor siblings of a node \( m \), if \( n_2 \) contains \( n_1 \), then only \( n_2 \) will be in the minimum path from \( n \) to \( m \). Therefore, hyperplane splits limit the number of nodes in a minimum path to at most \( 2d \).

In the example in Figure 4.33, with 2 dimensions and 2 splitting directions, there are 4 possible combinations of splitting dimensions and directions. Therefore, after every 4 splits, the descriptor region of the 5th split must contain the descriptor region of at least one existing sibling nodes. Note that this is the containment of descriptor regions, not declared spaces. So the split may not be a containment split and a containment node can much child nodes than \( 2d \).

The drawback of using hyperplane split includes that (1) the areas of the descriptor regions tend to be large, hence it is more likely for an intersection query to access the corresponding page, and (2) the descriptor regions of siblings are more likely to intersect. If the split dimensions of two sibling nodes are different, then their descriptor regions must intersect and one becomes a predecessor sibling of the other. When a new node is created, even if its descriptor region contains the descriptor region of an existing node, as long as a sibling in between splits at different dimension, the split can not be a containment split. In such case, the containment tree is “flat” and it is not good for index page split in upper level of the Spatial C-Tree. The only way to have a
containment split is to use the same split dimension and direction as the last sibling node. In this case the descriptor region of the new node is forced to have a large area. Either one is not good for query performance. Experiments show that with this approach, the area of the data pages tend to be large, hence a poor query performance. Therefore, we focus on the following MBR approach in our experiments.

**The MBR Approach**

An alternative way to split data pages is to use the “MBR” approach. We first pick the objects and/or side pointers to split. And then we use the minimum bounding rectangle (MBR) of these objects and/or side pointers as the descriptor region of the new page, hence the name the MBR approach. During the split, we try to make the newly created page “dense”, where density is defined as the number of objects divided by the area of the MBR. Intuitively, making the area of the region of the newly create page small is another way to split. We evaluate both “density” based split and “area” based split in our experiments.

Now let’s look at the details of the split. Let \( k \) be the smallest number of objects that a data page contains. Let \( t \) be a threshold for the number of side pointers in a page. We will use experiments to decide the best choices of \( k \) and \( t \). Given an overflowing page \( P \) with \( s \) side pointers and \( n \) objects, if \( s \) is smaller than \( t \), then we will perform a “regular” split, else if \( s \) is equal to or larger than \( t \), then we will force a “containment split” to reduce the side pointer numbers. Note that a regular split may or may not be a containment split. If it happens to be a containment split, the contained side pointers need to be moved to the new page. So at the end of the regular split, we check whether the new side pointer contains any existing side pointer. In the following, we discuss these two splits respectively.

**Case 1:** Regular Split happens when the number of side pointers in \( P \) is less than the predefined threshold.

We use “plane sweep” as illustrated in Algorithm 11, to find the best split candidate. We first need to sort all objects along each dimension and also along each direction of the dimension. The sorting will speed up later operations. Along each dimension, we perform “plane sweep” using the sorted lists. In each iteration, we try to include \( m \) to \( M - m \) objects and look for the best split candidate (with largest density or smallest area). Once the new bounding rectangle is decided, we check whether this split is a containment split. If it is, then the qualified side pointers are moved to the new page as well.
Case 2: When the number of side pointers in \( P \) is \( t \) (or greater than \( t \)), a containment split will be performed when possible.

In a containment split, we constantly refer to the “expandable” edges of the descriptor regions of a side pointer. If an edge of a side pointer \( s \) does not intersect with any side pointer in \( P \) that considers \( s \) as its intersecting predecessor sibling, then we say this edge can be “expanded” because the expansion of this edge does not change the preceding regions of any other side pointers.

All edges of the last side pointer in the side pointer list are expandable. In Figure 4.35, \( R_1 \) and \( R_2 \) are descriptor regions of two side pointers \( Pr_1 \) and \( Pr_2 \). \( Pr_1 \) is a predecessor sibling of \( Pr_2 \). The upper and left edges of \( R_1 \) are “expandable” because expanding these two edges to include more objects will not change preceding regions and the declared space of \( Pr_2 \). All four edges of \( R_2 \) are expandable because \( R_2 \) has no successor.

Algorithm 12 (ContainmentSplit) describes how the containment split works. We consider every side pointer that has “expandable” edge as a possible side pointer that can be contained in the new descriptor region (Line 2 to 11). We use Algorithm 13 to calculate the highest density to include this side pointer. And we pick the best out of all possible splits.

In Algorithm 13, for a given side pointer, to find a good split candidate, we only need to look at those objects that expand only the “expandable” edges of this side pointers (Line 2 to 4). We first find \( k \) objects that increase the MBR the least amount (Line 5 to 10). And then we try out \( n - 2k \) splits to find the best with the highest density (Line 11 to 29).

So far we have seen the splitting and posting algorithms of the Spatial C-Tree. The next chapter presents the experimental work to evaluate the performance of the Spatial C-Tree.
Algorithm 11 RegularSplit(List of objects $o_i$, List of side pointers $s_j$, int $k$)

1: sort objects according to the lower value and higher value respectively along each dimension.
2: for each dimension do
3:    {plane sweep using the sorted list of the lower values}
4:    $i \leftarrow 0$
5:    while $i \leq M - k$, where $M$ is the total number of objects in the list {make sure there are at least $k$ objects in the newly created page} do
6:        for $x$ from $m$ to $M - m$ do
7:            {there may not be as many as $M - m$ objects after $i$}
8:            calculate the density of $k$ objects, from $i$th objects to $(i+k)$th, in the sorted list. {when using “area” based split, calculate the area of the MBRs of these objects.}
9:            if the density (or area) is better than the best so far then
10:               { for density, we look for the largest one, for area, we look for the smallest }
11:                  remember the density and the MBR as a split candidate
12:           end if
13:        end for
14:    end while
15: repeat the above using the sorted list of the higher values along this dimension.
16: end for
17: using the MBR $R$ remembered as split candidate to split the current page.
18: objects contained in $R$ are split off to the new page
19: qualified side pointers, if there is any, is also split off to the new page.
Algorithm 12 ContainmentSplit(List of objects $o_i$, List of side pointers $S_j$, int $k$)

1: $density = 0$. $M = \text{nil}$.
2: $\textbf{for each side pointer } S_j \textbf{ do}$
3:   find the edges of $S_j$ that do not intersect the descriptor region of any pointer that considers $S_j$ as a predecessor sibling.
4:   add the edge of list $D$.
5:   $R ←$ the descriptor region of $S_j$.
6:   FindSplitCandidateByExpanding($R$, $D$, $o_i$, $k$). Let $d$ be the returned density and $R$ be the returned bounding rectangle. \{Algorithm 13\}
7:   $\textbf{if } d > density \textbf{ then}$
8:      $density = d$.
9:      $M = R$.
10: $\textbf{end if}$
11: $\textbf{end for}$
12: $\textbf{if } density > 0 \textbf{ then}$
13:   Split all objects contained in $M$ to the new page. $M$ is the descriptor region of the new page.
   Move all side pointers whose descriptor regions are contained in $M$ and do not intersect with any predecessor sibling of $M$ to $M$.
14: $\textbf{else}$
15:   $R ←$ the MBR of all objects.
16:   $\text{SplitByShrinking}(R, o_i, k)$
17: $\textbf{end if}$
Algorithm 13 FindSplitCandidateByExpanding(Rectangle $R$, List of edges of $R$, List of objects $o_i$, int $k$) Given a rectangle $R$, a list of edges of $R$ that can be “expanded”, a list of objects, Find the best split candidate according to density.

1: $L \leftarrow$ list of object $o_i$.
2: for each edge of $R$ that can not be expanded do
3: remove objects from $L$, $LX_{low}$, $LY_{low}$, $LX_{high}$, and $LY_{high}$ that are outside this edge (if the edge is a lower edge along a dimension, then remove all objects that are on the left hand side of the edge. if the edge if a higher edge along a dimension, then remove all objects on the right hand side of the edge).
4: end for
5: $i = 0$.
6: while $i \leq k$ do
7: $o \leftarrow$ FindExpandingCandidate($R$, $LX_{low}$, $LY_{low}$, $LX_{high}$, $LY_{high}$) \{Algorithm 14\}.
8: $R \leftarrow$ the MBR of $R$ and $o$.
9: Remove $o$ from $LX_{low}$, $LY_{low}$, $LX_{high}$, and $LY_{high}$.
10: end while
11: $n = \text{number of remaining objects in the current list}$.
12: $m = k$.
13: $a \leftarrow$ area of $R$
14: $\text{density} = m/a$.
15: $M \leftarrow R$
16: while $n > k$ do
17: $o \leftarrow$ FindExpandingCandidate($R$, $LX_{low}$, $LY_{low}$, $LX_{high}$, $LY_{high}$).
18: $R \leftarrow$ the MBR of $R$ and $o$.
19: Remove $o$ from $LX_{low}$, $LY_{low}$, $LX_{high}$, and $LY_{high}$.
20: $n = n - 1$.
21: $m = m + 1$.
22: $a = \text{area of } R$.
23: $d = m/a$.
24: if $d > \text{density}$ then
25: $\text{density} = d$.
26: $M = R$.
27: end if
28: end while
29: return $M$ and $\text{density}$.
Algorithm 14 \textit{FindExpandingCandidate}(Rectangle $R, LX_{\text{low}}, LY_{\text{low}}, LX_{\text{high}}, LY_{\text{high}}$)

Given a list of objects sorted according to each dimension and a rectangle $R$, return the object from the list such that the MBR of $R$ and this object is the smallest.

1: $i = 1$. \{ — find the object that is the closest to lower $x$ edge of $R$ — \}
2: while the lower $x$ coordinate of $LX_{\text{low}}[i+1]$ is no larger than the lower $x$ coordinate of $R$ do
3: $i = i + 1$.
4: end while
5: $o_{\text{slow}} \leftarrow LX_{\text{low}}[i]$.
6: $i = 1$. \{ — find the object that is the closest to higher $x$ edge of $R$ — \}
7: while the higher $x$ coordinate of $LX_{\text{high}}[i+1]$ is no smaller than the higher $x$ coordinate of $R$ do
8: $i = i + 1$.
9: end while
10: $o_{\text{xhigh}} \leftarrow LX_{\text{high}}[i]$.
11: Repeat step 1 to 4 on $LY_{\text{low}}$. Assign the object whose lower $y$ coordinate is the largest one that no larger than the lower $y$ coordinate of $R$ to $o_{\text{ylow}}$.
12: Repeat step 6 to 9 on $LY_{\text{high}}$ and assign the object whose high $y$ coordinate is the smallest that is no smaller than the higher $y$ coordinate of $R$ to $o_{\text{yhigh}}$.
13: $L \leftarrow \{o_{\text{slow}}, o_{\text{xhigh}}, o_{\text{ylow}}, o_{\text{yhigh}}\}$.
14: $r = 0$.
15: $o = \text{nil}$.
16: $i = 1$.
17: while $i \leq 4$ do
18: $r' \leftarrow$ area of the minimum bounding rectangle of $R$ and $L[i]$.
19: if $r' > r$ then
20: $r = r'$.
21: $o = L[i]$.
22: end if
23: end while
24: return $L[i]$. 
Chapter 5

Performance Results

We performed extensive experiments to compare the Spatial C-Tree with some representative techniques. This chapter shows the experimental results. We first present the characters of the spatial data sets we used. We then use experimental results to show the influence of data page minimum capacity on the Spatial C-Tree structures and the query performance in Section 5.3. Section 5.4 presents the insertion and query performance to compare the Spatial C-Tree with the R-Tree variants.

Note that currency control and recovery related performance is not included in this chapter since the Spatial C-Tree uses the Π-Tree concurrency control algorithm, which is shown in Chapter 2 to be more efficient than the existing ones for the R-Tree family. Thus, in this chapter, we focus on the insertion and query performance.

5.1 Experimental Setup and Workloads

We compared the Spatial C-Tree with (1) the original R-Tree (with quadratic splitting algorithm), (2) the R*-Tree and (3) a variant of the R-Tree that uses the same splitting algorithm as the R*-Tree but without reinsertion. We call this variant the R$_{NR}^*$-Tree (the R*-Tree without reinsertion). We chose to use the R$_{NR}^*$-Tree because the splitting algorithm of the R*-Tree outperforms the original R-Tree to some extent and the concurrency of the R*-Tree degenerates dramatically with the reinsertion. In the rest of this chapter, we present the query performance for all three R-Tree variants. However, the R*-Tree query performance should not be considered as a competitive result because the R*-Tree has very low concurrency due to reinsertion.

We implemented all of these structures in Java. Our experiments were run on a Dell Op-
tiplex GX620 PC with a 3.2GHz Pentium(R)D processor. An LRU (least recently used) buffer replacement policy is used.

We run our experiments on both synthetic and real data sets. All of them are non-point data, represented by their MBRs.

We use two different ways to generate synthetic data sets. (1) One is to randomly generated uniformly distributed squares. Given two values \( l_{\text{min}} \) and \( l_{\text{max}} \) \((l_{\text{min}} < l_{\text{max}})\), the side of the squares follows a normal distribution, which centers at \( \frac{l_{\text{min}}+l_{\text{max}}}{2} \). (99.7% of the squares have sides between \( l_{\text{min}} \) and \( l_{\text{max}} \).) (2) Since real life data tend to be fractals [22, 7, 2], we also generated fractal data sets using “Levy Flight”, a non-standard random walk process whose step length \( z \) (flight paths) are cumulative distributed according to \( P(z) = \text{const} \cdot z^{-D} \) [22, 24]. Here \( D \) is the Hausdorff fractal dimension. In addition to geometry, the size of these non-point objects are also generated following a self-similar power-law distribution \( N_r(A > a) = F \cdot a^{-B} \) [22, 24]. The shape of these objects are also square.

For real data sets, we chose to use the TIGER database [27] of the US Bureau of Census and polygon data in SEQUOIA2000 [26]. In the TIGER data set, we chose ‘MGCounty’ and ‘LBCounty’ (road intersections of the Montgomery county, MD and Long Beach county, CA), since they are popular data set used in spatial data structure testing [2, 7]. For the SEQUOIA2000 data set, we used MBRs to approximate the polygon data. The objects in SEQUOIA2000 are more squarish than the ones in TIGER data sets. In particular, there are almost 14% objects in the MG data set are vertical or horizontal lines and more than 37% in the LB data set.

The following table is a summary of the data sets we used. The spatial layouts of these data sets are attached in Appendix F.

<table>
<thead>
<tr>
<th>data set name</th>
<th>description</th>
<th>number of entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unif20K</td>
<td>Uniformly distributed data</td>
<td>20000</td>
</tr>
<tr>
<td>levy15</td>
<td>Fractal data with Hausdorff fractal dimension 1.5</td>
<td>10000</td>
</tr>
<tr>
<td>MG</td>
<td>Montgomery county road intersection from Tiger</td>
<td>39231</td>
</tr>
<tr>
<td>LB</td>
<td>Long Beach county roads from Tiger</td>
<td>53145</td>
</tr>
<tr>
<td>islands</td>
<td>Islands data set from Sequoia2000</td>
<td>25050</td>
</tr>
</tbody>
</table>

Table 5.1: A summary of data sets used in our experiments.

Throughout the experiments, we used different page sizes (2KB, 4KB, 8KB) and observed consistent performance.
5.2 Typical Queries

We compare performance of the following typical spatial queries.

1. **Exact match** query finds spatial objects, whose spatial regions are exactly the same as the query region.

2. **Intersection range** query finds objects whose spatial properties intersect the given query range.

3. **Containing query** searches for all objects that contain the given query range.

4. **Contained query** finds objects that are completely contained in the given query range.

In a Spatial C-Tree, an exact match search begins at the root page and traverses downward the tree. For each index page it encounters, it searches the containment tree inside the page for a containment tree node whose home space contains the query region and follow its pointer to the next level of the spatial C-Tree until a data page is reached.

An intersection query in a Spatial C-Tree searches for data pages whose descriptor regions intersects the query region. According to Property 16, if the descriptor region of a data page intersects the query region, then the descriptor region of its container page must also intersect the query region. Therefore, in the Spatial C-Tree, an intersection query is performed within the level 1 containment tree and look at data pages only. Within each data page, we search for objects that intersect the query range, and also side pointers whose descriptor regions intersect the query range. We traverse to the descendant pages via the side pointers.

The containing query is analogous to the intersection query. We also start with the root page of the level 1 containment tree and visit via side pointers whose descriptor regions completely contain the query range.

For contained query in a Spatial C-Tree, we start at the root page and search within an index for containment tree node whose home space intersects the query range. That is, the descriptor region intersects the query range and the intersection is not contained in the descriptor region of any predecessor or child node.

5.3 Parameters to Split Data Pages

There are a couple of parameters we reply on to perform a data page split. They are (1) the maximum number of side pointers allowed in a data page, and (2) the minimum capacity of a data
page. When the number of side pointers in a data page exceeds the maximum threshold, we will perform a containment split instead of a regular split. During split, we make sure the pages are at least as full as minimum capacity. We use experiments to evaluate best values for these two parameter. We also compare splits using “density” based approach and “area” based one.

To decide the minimum capacity in a data page in the Spatial C-Tree, we ran experiments with $m = 20\%, 25\%, 30\%, 35\%, 40\%$ and $45\%$ relative to $M$, the maximum capacity of a page. For the maximum number of side pointers, we used 2, 3, 5, 8 and 10. We did not use bigger numbers for side pointers because more side pointers could introduce more ghost nodes in index pages. In the experiments, we observe: (1) the number of pages created, (2) the areas of the data pages and (3) query performance change. The number of pages indicates the average page utilization. A smaller number means a better utilization. The area of a data page is a hint that shows how likely the page intersects with queries such as intersection query. Also a data page of large area could have more “dead space”. The query performance is the determine factor of which parameters to use in later experiments.

**Number of Pages**

Figure 5.1 illustrates the change in number of pages with different settings. This figure shows the MG data set result. Page size is set to 2KB. Other data sets and page sizes show similar results. They are listed in Appendix A.

![figure 5.1](image)

Figure 5.1: Influence of splitting parameter on the number of pages.

The x coordinate corresponds to the “split parameters”. To compare all variations of split parameters, we use the format $(n, m)$ along x axis, where $n$ is the maximum number of side pointers and $m$ is the minimum capacity. So the first 6 units along the x axis all have the same number of maximum side pointer 2 but different minimum capacities. And then the next 6 units
are for maximum side pointer 3 with different minimum capacities, and so on. We use such x coordinates in the rest of this section.

The undulation in Figure 5.1 shows that the number of pages decreases as the minimum capacity increases, while the number of side pointers has less an influence. This is consistent with intuition that bigger minimum capacity means better page utilization. The dotted blue line is for the “area” based split and the solid red line is for “density” based split. They have similar page numbers with the density based split slightly smaller.

Figure 5.2 shows the page numbers using islands data set with page size set to 8KB. It demonstrates the same result as in Figure 5.1. Note that when the page size is bigger, there are less number of data pages, hence fewer index pages. That’s why the lines that show the index page numbers have a different shape than the ones in Figure 5.2. There are either 1 or 3 index pages when the page size is set to 8KB.

---

**Area of Data Pages**

In the following, we first look at the overall trend of changes of data page area. And then we focus on one data set (MG data set) and analyze how different split parameters affect different category of data pages. The latter analysis is shown to be closely related to the performance trendy seen later on.

**Overall Trend**

Figure 5.3 shows the changes in the areas of data page regions with different splitting parameters. The blue solid lines are for area based split and the red dotted lines are for density based split. Instead of using the absolute value of the areas, we use relative values to the object space, which is shown as “percentage of object space” in the Y axis. Along X axis, for each value pair of (max
side pointer number, min capacity), we order the areas of data pages and look at the average value of the first 10 percent and so on in a 10 percent interval. For example, if there are 1000 pages, we order them according to their areas, and then look at average values of the first 100 pages (10 percentile), the average values of the next 100 pages (the next 10 percentile) etc... Such values provide us with an estimation of the areas without looking at 1000 actual numbers.

Figure 5.3: Influence of splitting parameter on the areas of data page regions.

Table 5.2 shows the first 10 values of the red column (density based split) in Figure 5.3. Their x coordinates are shown in the first column. The y coordinates are in the 2nd column. This means when the maximum number of side pointers is 2 and the minimum capacity value is 0.2, the average area of the smallest 10% data pages is 0.002915419%; the average area of the next 10% data pages is 0.006958678%; ... and the average area of the largest 10% data pages is 8.314470285%.

Figure 5.4 shows the smallest 70% of the data pages in Figure 5.3 as their area are much smaller than the top 30% pages and are hard to see when put together.

The result shown in Figure 5.3 and Figure 5.4 demonstrates that the area of data pages tend to be larger as the minimum capacity increases. And as the number of side pointers increases, the areas of data pages decreases. When minimum capacity increases, the number of data included in a page gets larger, so does the MBR of the page, hence larger area. As the number of side pointers increases, the split is more “flexible” (less containment split), so the best split candidate tend to have better density or smaller areas. The figures also show that the density based and the area based splits have about the same data page areas. The density based split have slightly smaller areas in most cases.

Figure 5.5 shows data page areas of fractal data set. The fractal data set is more “clustered”
Table 5.2: The first 10 values in Figure 5.3

<table>
<thead>
<tr>
<th>x coordinate</th>
<th>y coordinate</th>
</tr>
</thead>
<tbody>
<tr>
<td>[10%(2, 0.2)]</td>
<td>0.002915419</td>
</tr>
<tr>
<td>[20%(2, 0.2)]</td>
<td>0.006958678</td>
</tr>
<tr>
<td>[30%(2, 0.2)]</td>
<td>0.012657273</td>
</tr>
<tr>
<td>[40%(2, 0.2)]</td>
<td>0.023064014</td>
</tr>
<tr>
<td>[50%(2, 0.2)]</td>
<td>0.039635871</td>
</tr>
<tr>
<td>[60%(2, 0.2)]</td>
<td>0.065901333</td>
</tr>
<tr>
<td>[70%(2, 0.2)]</td>
<td>0.118744001</td>
</tr>
<tr>
<td>[80%(2, 0.2)]</td>
<td>0.232488405</td>
</tr>
<tr>
<td>[90%(2, 0.2)]</td>
<td>0.625526229</td>
</tr>
<tr>
<td>[100%(2, 0.2)]</td>
<td>8.314470285</td>
</tr>
</tbody>
</table>

Figure 5.4: The smallest 70% data pages in Figure 5.3

than the MG data set seen above. (The spatial layout of different data sets are shown in Appendix ) It seems the area based split (blue line) is more susceptible to the effect of clustering than the density based split. Other experiment results regarding data page areas shown similar result.
Analysis of Different Categories

In this part, we take a close-up look at each categories of data pages and how the splitting parameters effect every category.

We order the data pages by their area and look at the average area of each 10 percentile. In general, for data pages with smaller areas (e.g. the first 10 percentile, the 2nd 10 percentile etc), the minimum capacity affects the average area more than the maximum number of side pointer, as shown in Figure 5.6. As the area increases, effects of minimum capacity gets smaller. For the largest 10 percentile data pages, as shown in Figure 5.7, the impacts of the maximum number of side pointers is more than the impact of the minimum capacity of pages. (More details of this experiment can be found in Appendix B.

The above observation, that the page minimum capacity has more impacts on the pages with smaller areas and the maximum number of side pointers has more impact on the pages with larger area, is consistent for all data sets. Data pages with smaller areas for other data sets all look like the one shown in Figure 5.6. The charts for the largest 10 percentile are quite different. Figure 5.8 shows the largest 10 percentile and the influence of the splitting parameter of the fractal data set. Figure 5.9 shows the one for the LB data set.

Query Performance

When compare query performance of different data page split parameters, we only consider range queries as the query performance is the exact match query is fixed, in which the number of pages accessed is always the same as the height of the tree. For range queries, we use 7 sizes of query

Figure 5.5: Influence of splitting parameter on the areas of data page regions
Figure 5.6: Average areas of data pages in MG data set (2nd and 5th 10 percentile). Page size is 2KB.

areas (in terms of percentage of object space) : 0.01%, 0.05%, 0.1%, 0.5%, 1%, 5%, 10%. For each query size, we generate 50 queries and compare the average number of pages accessed.

For intersection queries, we have observed the following:

1. when the query area is small, the page splitting parameters have very similar impact on the average number of pages accessed as it does on the area of the largest 10 percentile data pages.

For example, Figure 5.10 shows the intersection query of MG data set with query area being 0.01% of the objects spaces. The top figure shows the average number of pages accessed per query. The x coordinate, same as what we have seen before, has format (n, m), where n is the maximum number of side pointers and m is the minimum capacity. The bottom figure
CHAPTER 5. PERFORMANCE RESULTS

Figure 5.7: Average areas of data pages in MG data set (9th and 10th 10 percentile). Page size is 2KB.

shows the average IOs of 50 consecutive queries.

Note that the trend of average number of pages accessed in Figure 5.10 is similar to the trend of the largest 10 percentile data pages in Figure 5.7. This observation applies to other data set as well. For example, for the LB data set, the average number of data pages in Figure 5.11 and the area of the largest 10 percentile data pages in Figure 5.9 show similar behavior, and for the fractal data set, Figure 5.12 and Figure 5.8 are also similar.

The average number of IOs shows similar behavior (bottom figure in Figure 5.10). But the impact of the maximum side pointer number is less dramatic on the IOs than it does on the number of pages accessed as a result of caching.

The above observation can be explained from two aspects: (i) when the query area is small,
there are a small number of data pages that contain qualified objects, and (ii) when a data page $P$ with qualified objects is visited, using the intersection query algorithm illustrated in Section 5.2, all data pages that are containers of $P$ in the data page containment tree are also visited and these container pages tend to have a large area (hence in the top 10 percentile of all data pages). Therefore, the effect of page splitting parameter on these top 10 percentile data pages dominates the query performance.

2. when the query area is large, the wavy trend of the query performance is similar to the charts of areas of most data pages (except the largest 10 to 20 percentile). For example, Figure 5.13 shows the average number of pages accessed using MG data set. This chart is similar to

Figure 5.8: Average areas of the largest 10 percentile data pages in fractal data set. Page size is 2KB.

Figure 5.9: Average areas of the largest 10 percentile data pages in LB data set. Page size is 2KB.
the areas of data pages shown in Figure 5.6. This is because, when the query area is large, more pages are accessed, and that there are so many pages that the ones with large areas in the top 10 percentile is just a small portion. The majority of the data pages accessed have a smaller area (falls below 80 or 90 percentile).

![Intersection Query: Avg number of pages accessed](image1)

![Intersection Query IOs](image2)

Figure 5.10: Intersection query result of MG data set. Query area is 0.01% of the object space. Page size is 2KB. Buffer size holds up to 500 pages.

The above two observations are important. They show that, to improve the intersection query performance of small query area, we should aim at improving the largest 10 percent data pages,
5.3. PARAMETERS TO SPLIT DATA PAGES

Figure 5.11: Intersection query result of LB data set. Query area is 0.01% of the object space. Page size is 2KB.

Figure 5.12: Intersection query result of LB data set. Query area is 0.01% of the object space. Page size is 2KB.

and to improve the intersection query performance of the larger query area, we should increase the minimum capacity. They also explain the different variation of query performance between the C-Tree and the R-Tree, as discussed later on in Section 5.4. More intersection query results can be found in Appendix C.

The query results of **containing queries** are similar to the intersection queries. This is because
the search procedure of intersection query and containing query are alike. They both start at the root data page, and in both queries, if a data page qualifies the search predicate, then its container pages also qualify, therefore should be visited. Figure 5.14 shows the containing query results of the MG data set, with query area being 0.01% of the object space. Again, this chart shows similar trend as the average areas of the top 10 percentile data pages shown in Figure 5.7. More containing query result can be found in Appendix E.

The contained query, on the other hand, is different from the above two types. It started at the root index page and only visits a page if the query intersects the descriptor region of the page and the intersection is not inside any predecessor. What’s more, if the query range is contained in a data page $P$, then the container page of $P$ will not be visited. Hence the “top 10” percentile data page effect does not exist in contained query performance. Instead, it shows a trend more like the areas of the majority of data pages. That is a larger minimum number of capacity tends to produce a better query result, with small variations from the maximum number of side pointers. As an example, Figure 5.15 shows the contained query performance using MG data set with query area being 0.01%. Appendix D lists more query results of contained queries.

Figure 5.13: Intersection query result of MG data set. Query area is 10% of the object space. Page size is 2KB.
5.4 Performance Comparison

In this section, we compare the query performance of the Spatial C-Tree and three R-Tree variants, the R-Tree (with quadratic split, the R*-Tree and the R*-NR-Tree, which is the R*-Tree without reinsertion). When construct the Spatial C-Tree, we use density based split, and we set the maxi-
mum number of side pointers to 10 and the minimum page capacity to 0.35.

Before move onto the performance, let’s first look at the characteristics of the constructed trees.

Figure 5.16 shows the total number of pages of each tree. The Spatial C-Tree has the largest number of pages while the R*-Tree has the smallest number. The number of pages is an indication of page utilization. It is not directly related to the query performance.

![Figure 5.16: Total number of pages.](image)

Figure 5.17 shows the comparison of average area of data pages. The \( x \) coordinates are percentiles of data pages ordered by area. The first unit along \( x \) axis is the first 10 percentile. 20% means the second 10 percentile, etc. The \( y \) coordinate shows the average areas of pages in each category in terms of percentage of the object space. For example, the first unit shows that the average area of the data pages in the 1st 10 percentile category in C-Tree is 0.006%. The corresponding number for the R-Tree is 0.012%.

Figure 5.17 shows that, (1) when the percentile is small (e.g. 10\%, 20\%), the average areas of the C-Tree data pages are smaller than the average areas of all R-Tree variants; (2) as the percentile number increases, the average area of data pages in the C-Tree increases faster than the R-Trees; and (3) in the largest 30 percentile, the average areas of C-Tree data pages are much larger than the ones in the R-Trees.

Recall the observation in the previous section that in a Spatial C-Tree, for intersection and containing queries, when the query area is small, the largest 10 percentile data pages tend to dom-


Figure 5.17: Comparison of data page areas.

imate the query performance, and as query area becomes larger, such dominance gets smaller. This is demonstrated in the query performance result shown later in Section 5.4.3 and Section 5.4.4.

5.4.1 Insertion

Figure 5.18 shows the performance comparison of data insertions. In this result, the page size is set to 2KB. Other page sizes show similar result. We insert the objects in a random order into an empty tree and observe the total number of pages accessed. The Spatial C-Tree has the smallest number of pages accessed, although its total number of pages is larger than the others (Figure 5.16). This is because in the Spatial C-Tree, an insertion always read $h$ pages, where $h$ is the height of the tree, and writes one page unless there is page split. However, in the R-Tree
variants, when an object is inserted, it is likely that all pages along the insertion path needs to be adjusted to reflect the minimum bounding rectangle change. This adjustment results in more page access. For the R*-Tree, the reinsertion introduces even more page accesses, hence the R*-Tree has the largest number of pages accessed among all trees.

![Chart showing insertion performance](image)

**Figure 5.18:** Performance of insertions.

### 5.4.2 Exact Match Queries

The exact match queries are generated following the data distribution. The query range size is the average size of all data entries. Another parameter for the query generation is the hit ratio $h$ defined as the probability that a data entry is found. We use a random number generator to generate a random number $r$ from 0 to 1. If the generated random number $r$ is bigger than $h$, we randomly generate a query. If $r$ is no bigger than $h$, we randomly choose a data entry in the data set to be the query range.

For an exact match query, regardless of the hit ratio, the Spatial C-Tree always visits a fixed number of pages. This number is the height of the tree. The R-Tree variants, on the other hand, is more likely to stop at upper level index pages when the hit ratio is smaller than 1, and when the hit ratio is 1, visit more than $h$ pages, where $h$ is the height of the tree.

Figure 5.19 shows the exact match query performance with hit ratio 1 (i.e. there is always an object that satisfies the query). As expected, the number of pages accessed in the Spatial C-Tree is constant. And the number of pages accessed in the R-Tree variants are larger than the number
of pages accessed in the Spatial C-Tree, with the R-Tree having the worst performance. Note that for the uniform data set, the number of pages accessed in the Spatial C-Tree is one page more than the ones in the R-Tree variants. This is because for this particular data set, the Spatial C-Tree has 3 levels with the root page having 2 entries and the R-Tree variants all have 2 levels. Hence for this data set, the number of pages accessed in the Spatial C-Tree is always 3 and the number of pages accessed in the R-Trees are slightly above 2.

![Exact Match Query Performance Graph](image)

Figure 5.19: Exact match query performance.

### 5.4.3 Intersection Queries

The query ranges for the intersection queries, the contained queries and the containing queries are generated following data distribution. Their areas are varied from 0.01% to 5% of the entire object space. The page size is set to be 4KB. The minimum capacity of a Spatial C-Tree data page is set to be 0.35M and the maximum number of side pointers allowed in a data page is set to 10. All the following query results are an average value of 50 different queries.

Figure 5.20 shows the intersection query results with various query ranges using the MG county data set, in terms of number of pages accessed. From the figure, we can observe the following: (1) As the query range increases, the query performance of all tree structures decreases. This is because when the query range becomes bigger, it is more likely for a page to intersect the query range. (2) The R-Tree variants constantly outperforms the Spatial C-Tree. And (3) as the query range increases, the difference between Spatial C-Tree and the R-Trees decreases. Table 5.3
shows the same result as in Figure 5.20 using the query result of the R\^{NR}\textsuperscript{*}-Tree as the baseline.

Figure 5.20: Performance comparison (number of pages accessed) of intersection queries using the MG data set.

<table>
<thead>
<tr>
<th>Query Area</th>
<th>C-Tree</th>
<th>R^{NR}\textsuperscript{*}-Tree</th>
<th>Original R-Tree</th>
<th>R^{*}-Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01%</td>
<td>159.8%</td>
<td>100.0%</td>
<td>139.4%</td>
<td>100.4%</td>
</tr>
<tr>
<td>0.05%</td>
<td>157.7%</td>
<td>100.0%</td>
<td>138.6%</td>
<td>93.9%</td>
</tr>
<tr>
<td>0.10%</td>
<td>165.8%</td>
<td>100.0%</td>
<td>130.0%</td>
<td>98.9%</td>
</tr>
<tr>
<td>0.50%</td>
<td>135.8%</td>
<td>100.0%</td>
<td>113.1%</td>
<td>94.3%</td>
</tr>
<tr>
<td>1.00%</td>
<td>124.3%</td>
<td>100.0%</td>
<td>115.5%</td>
<td>94.8%</td>
</tr>
<tr>
<td>5.00%</td>
<td>114.2%</td>
<td>100.0%</td>
<td>106.6%</td>
<td>92.6%</td>
</tr>
<tr>
<td>10.00%</td>
<td>109.6%</td>
<td>100.0%</td>
<td>104.9%</td>
<td>92.2%</td>
</tr>
</tbody>
</table>

Table 5.3: Intersection query performance compared to the R\^{NR}\textsuperscript{*}-Tree.

Recall that in the end of Section 5.3, we have shown that when the query range is small, the query performance trend of the Spatial C-Tree is “dominated” by the largest 10 percentile data pages. This observation may or may not be true for the R-Tree variants. But lowering the area of the largest 10 percentile data pages in the Spatial C-Tree may help close the gap between the Spatial C-Tree and the R-Tree. Figure 5.21 and Figure 5.22 show the average data page areas in each 10 percentile of all four tree structures. As we can see in Figure 5.21, the average area of the largest 5 percentile data pages in the Spatial C-Tree is more than 10 times larger than its
correspondence in the R-Trees, and the next 5 percentile data pages is about twice as much as the ones in the R-Tree. The query performance of the Spatial C-Tree, on the other hand, is less than twice as much as the R-Trees. Therefore, the intuition here is that the average areas of the largest 10 percentile data pages in the Spatial C-Tree might not have to be the same or smaller than the ones in the R-Tree. Lowering the areas to as much as half of its current value might help close the performance gap dramatically.

Figure 5.21: Comparison of data page areas.

Figure 5.22: Comparison of data page areas.
Other data sets show similar results as the MG data set discussed above. The query performance difference between the Spatial C-Tree and the R-Trees are proportional to the difference of average areas between the largest 10 percentile data pages in the Spatial C-Tree and the largest 10 percentile data pages in the R-Tree. In other words, when the query performance difference between the Spatial C-Tree and the R-Tree is large, the difference of average areas of the largest 10 percentile data pages between the Spatial C-Tree and the R-Tree tend to be large too.

One result set worth to mention is the one of fractal data set. Figure 5.23 and the Figure 5.24 show the query result in terms of number of pages accessed and I/Os respectively. In Figure 5.23, the number of pages accessed by the Spatial C-Tree is much bigger than the ones from the R-Trees. However, this gap becomes much smaller in Figure 5.24 where caching effect helps to keep the large data pages in memory. Note that in this data set, the objects are grouped into several clusters. The Spatial C-Tree is more sensitive to the insertion order. And with random insertion, the data pages of the Spatial C-Tree are much larger than the ones in the R-Tree. Hence the big performance difference. Figure 5.26 and Figure 5.25 show the average areas of data pages in the Spatial C-Tree and in the R-Trees. This set of result is consistent with the earlier conjecture that the size of the largest 10 percentile data pages might play an important role in the intersection query performance of the Spatial C-Tree.

Figure 5.23: Performance comparison (number of pages accessed) of intersection queries using fractal data set.
5.4. PERFORMANCE COMPARISON

141

Figure 5.24: Performance comparison (IOs) of intersection queries using fractal data set.

Figure 5.25: Comparison of data page areas.

5.4.4 Containing Queries

The performance results of containing queries are similar to the intersection queries. The R-Trees tend to outperform the Spatial C-Tree in most cases and the performance difference decreases as the query area increases. Figure 5.27 shows the containing query result using the MG data set.

The similarity between the containing query and the intersection query is because their search procedure are alike. They both start from the root data page and if a data page satisfies the query predicate, then its container page will also be visited, although some of these container pages might
not contain any object qualified. So in addition to lower the data page areas, another possibility to improve the query performance of the Spatial C-Tree is to "skip" some of these container pages by utilizing index pages.
5.4. PERFORMANCE COMPARISON

5.4.5 Contained Queries

For the contained queries, in general, the Spatial C-Tree is comparable to the original R-Tree. The performance of the R-Trees is constantly better than the ones of the Spatial C-Tree. Figure 5.28 shows the performance comparison using the MG data set. Table 5.4 shows the comparison using the result of the R*NR-Tree as the baseline. One observation is that when the query range is small (e.g. 0.01%) or large (e.g. 5%), the difference between the Spatial C-Tree and the R-Trees are smaller than the times when the query range is around 0.01%. Other data set are consistent with this finding. One guess is that the R-Trees are “best packed” for query ranges around 0.01%. But more experiments need to be done to produce more accurate reasoning.

Figure 5.28: Performance comparison of contained queries using MG data set.
<table>
<thead>
<tr>
<th>Query Area</th>
<th>C-Tree</th>
<th>R*NR-Tree</th>
<th>Original R-Tree</th>
<th>R*-Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01%</td>
<td>113.1%</td>
<td>100.0%</td>
<td>139.4%</td>
<td>100.4%</td>
</tr>
<tr>
<td>0.05%</td>
<td>134.8%</td>
<td>100.0%</td>
<td>138.6%</td>
<td>93.9%</td>
</tr>
<tr>
<td>0.10%</td>
<td>156.1%</td>
<td>100.0%</td>
<td>130.0%</td>
<td>98.9%</td>
</tr>
<tr>
<td>0.50%</td>
<td>139.4%</td>
<td>100.0%</td>
<td>113.1%</td>
<td>94.3%</td>
</tr>
<tr>
<td>1.00%</td>
<td>129.9%</td>
<td>100.0%</td>
<td>115.5%</td>
<td>94.8%</td>
</tr>
<tr>
<td>5.00%</td>
<td>119.6%</td>
<td>100.0%</td>
<td>106.6%</td>
<td>92.6%</td>
</tr>
<tr>
<td>10.00%</td>
<td>113.6%</td>
<td>100.0%</td>
<td>104.9%</td>
<td>92.2%</td>
</tr>
</tbody>
</table>

Table 5.4: Contained query performance compared to the $R^*_{NR}$-Tree.
Chapter 6

Conclusion and Future Work

We conclude this dissertation with a summary of our contribution and some directions of future work.

6.1 Summary

Spatial index has received a lot of attention for the past two decades due to the widely use of applications that deal with spatial data. These applications demand the underlying DBMS to provide both efficient data access and high concurrency during data operations. The existing methods have good performance in one of these two properties, but not both.

We proposed a method the Spatial C-Tree that utilizes the Π-Tree concurrency by extending the C-Tree framework. The concurrency is more efficient than existing methods in the R-Tree family. The insertion and the exact match queries have a disk access guarantee which is not always true in the R-Tree family. The performance of range queries, including intersection query, containing query and contained queries of the Spatial C-Tree, is outperformed by the R-Trees in most cases, but with much room to improve.

In Chapter 3, we define containment tree, containment tree hierarchy and the C-Tree framework. In Chapter 4, we introduce spatial containment tree and the Spatial C-Tree as an extension of the C-Tree framework. We also define in this chapter the page split and posting algorithm of the Spatial C-Tree with two variations in data page split.

In Chapter 5, we present experimental results. We compare various splitting parameters in the Spatial C-Tree and observe that, for queries like intersection query and containing query, when the query range is small, the page splitting parameters have similar impact as it does on the area of the
largest 10 percentile data pages, and when the query range is large, impact of splitting parameters on the query performance is similar to its impact on the areas of most data pages (except the top 10 percentile). In the second half of the chapter, we compare the Spatial C-Tree and the R-Tree variants. The Spatial C-Tree has the best performance in insertion and exact match. The performance of contained query is comparable between the Spatial C-Tree and the R-Tree variants. For intersection and containing queries, the R-Trees outperforms the Spatial C-Tree. This is likely because of the large areas in the top 10 or 20 percentile of the data pages in the Spatial C-Tree.

6.2 Future Work

In Chapter 5, we have mentioned the observation that the largest 10 percentile data pages seem to “control” the performance of queries such as intersection query and containing query. Therefore, taking down the area of the largest 10 percentile data page could be one potential solution to improve the query performance.

Another observation is that the intersection query and the containing query both visit unnecessary container pages. When a qualified data page is visited, its container pages are always visited. Such visits of container page can be avoided by utilizing the index pages and remembering statistics of the object such as the largest side along each dimension.

Assume we know the largest side of all objects in the Spatial C-Tree are $t_1, t_2, \ldots, t_d$ for all $d$ dimensions. For an intersection query, we can expand the query range $Q$ by $t_i$ along both directions of dimension $i$, as shown in Figure 6.1(a). In an index page, we search for those containment tree nodes whose home space “intersects” the expanded query range. The home space of node $n$ “intersects” the expanded query range when (i) the descriptor region of $n$ intersects $Q$, (ii) the intersection of $Q$ and the descriptor region of $n$ is not contained in any predecessor of $n$, (iii) $n$ does not have any descendant $m$ such that the descriptor region of $m$ contains the expanded query range, $m$ is a real node and the expanded query range is not contained in any predecessor of $m$.

We only need to visit those child pages indexed by such $n$ that satisfies all of the above 3 conditions. If $n$ contains any object that intersects $Q$, then the descriptor region of $n$ must contain such object (condition (i)) and the object can not be inside the predecessor node of $n$ (condition (ii)). Since $t_i$s are the largest side possible along the $i$-th dimension. For any object that intersections $Q$, the object must be inside the expanded query range. If such range is contained in a descendant node $m$, then there can not be any object in $n$ that intersections $Q$ (condition (iii)).

For an containing query, we can expand the query range as shown in Figure 6.1(b). Note that,
6.2. **FUTURE WORK**

Figure 6.1: Optimization on the intersection queries and the containing queries.

if $t_i$ is smaller than the side of query range $Q$ along $i$-th dimension, then we can stop the query at the root page because in this case there will not be any object that contains $Q$. In an index page, we search for those nodes $n$ such that (i) the descriptor region of $n$ contains $Q$, and $Q$ is not contained in any predecessor of $n$, and (ii) $n$ does not have any descendant $m$ where the descriptor region of $m$ contains the expanded query range in Figure 6.1(b) and the expanded query range is not contained in any predecessor of $m$.

The performance improvement of the above optimization depends on how accurate the “largest side” describes the objects. In extreme cases, e.g. there is one object that is substantially larger than the rest of the objects, the above optimization might not work. We might end up visit more pages than without such optimization. One compensation for such inaccuracy is to store the statistics information in each index page or maybe even in each index entry, which will of course introduce extra cost to update.

Another direction to improve the query performance is to explore bulk load in the Spatial C-Tree. Bulk loading could potentially pack the data pages in such a way that there are less “large” data pages. From the experiment results in Chapter 5, we have seen that the query performance is mostly affected by the largest 10% or 15% data pages. Keeping these data pages small and leaving maybe a constant number of “large” data pages should be the goal of bulk loading. Separating objects of different sizes and identifying clusters might be helpful in the bulk loading process.

Another future direction is to extend the C-Tree framework to index spatio-temporal data. Spatial attributes and temporal attributes both have containment properties. Yet their containment are different. That’s why the existing approaches that consider the temporal value as an extra dimension of the spatial part have been not been very successful. The C-Tree framework does not
require each dimension of the multi-dimensional data to be treated equally. As long as we can (1) define a containment relationship along each dimension and (2) define a semantics in a way such that, given two records $r = ([d_1, d_3, \ldots, d_n], t)$ and $r' = [d_1', d_3', \ldots, d_n']$, $t'$, if $[d_1, d_3, \ldots, d_n]$ contains $[d_1', d_3', \ldots, d_n']$ and $t$ contains $t'$, then $r$ contains $r'$, the C-Tree framework can be applied.
Bibliography


Appendices
Appendix A

Number of Pages
Figure A.1: Influence of maximum number of side pointers and minimum capacity on the number of data pages and index pages. (Page Size is 2KB)
Figure A.2: Influence of maximum number of side pointers and minimum capacity on the number of data pages and index pages. (Page Size is 4KB)
Figure A.3: Influence of maximum number of side pointers and minimum capacity on the number of data pages and index pages. (Page Size is 8KB)
Appendix B

Areas of Data Pages of MG Data Set
Figure B.1: a (page size = 2KB).
Figure B.2: a (page size = 2KB).
Figure B.3: a (page size = 2KB).
Figure B.4: a (page size = 2KB).
Appendix C

Intersection Query (Page Size = 2KB)
Figure C.1: Query area is 0.01% of the object space
Figure C.2: Query area is 0.05% of the object space
Figure C.3: Query area is 0.1% of the object space
Figure C.4: Query area is 0.5% of the object space
Figure C.5: Query area is 1% of the object space
Figure C.6: Query area is 5% of the object space
Figure C.7: Query area is 10% of the object space
Appendix D

Contained Query (Page Size = 2KB)
Figure D.1: Query area is 0.01% of the object space
Appendix E

Containing Query (Page Size =2KB)
Figure E.1: Query area is 0.01% of the object space
Appendix F

Data Sets
Figure F.1: SEQUOIA2000 islands data set.
Figure F.2: TIGER LB county data set.
Figure F.3: TIGER MG county data set.
Figure F.4: Synthetic (fractal) data set.
Figure F.5: Synthetic (uniform) data set.
Appendix G

Intersection Query Performance Comparison
Figure G.1: Intersection queries. Page size is 2KB.
Figure G.2: Intersection queries. Page size is 4KB.
Figure G.3: Intersection queries. Page size is 8KB.
Appendix H

Contained Query Performance Comparison
Figure H.1: Contained queries. Page size is 2KB.
Figure H.2: Contained queries. Page size is 4KB.
Figure H.3: Contained queries. Page size is 8KB.
Appendix I

Containing Query Performance Comparison
Figure I.1: Containing queries. Page size is 2KB.
Figure I.2: Containing queries. Page size is 4KB.
### Containing Query Performance Comparison

<table>
<thead>
<tr>
<th></th>
<th>R*-Tree</th>
<th>R*NR-Tree</th>
<th>R-Tree</th>
<th>C-Tree (density)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.54</td>
<td>1.14</td>
<td>1.08</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1.66</td>
<td>1.3</td>
<td>1.06</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1.88</td>
<td>1.3</td>
<td>1.16</td>
<td>1.02</td>
</tr>
<tr>
<td></td>
<td>1.59</td>
<td>1.59</td>
<td>1.65</td>
<td>1.04</td>
</tr>
<tr>
<td></td>
<td>1.72</td>
<td>1.72</td>
<td>1.8</td>
<td>1.02</td>
</tr>
<tr>
<td></td>
<td>1.56</td>
<td>1.56</td>
<td>1.62</td>
<td>1.06</td>
</tr>
<tr>
<td></td>
<td>1.32</td>
<td>1.32</td>
<td>1.38</td>
<td>1.10</td>
</tr>
<tr>
<td></td>
<td>1.32</td>
<td>1.32</td>
<td>1.38</td>
<td>1.10</td>
</tr>
<tr>
<td></td>
<td>0.64</td>
<td>0.64</td>
<td>0.44</td>
<td>0.30</td>
</tr>
<tr>
<td></td>
<td>0.64</td>
<td>0.64</td>
<td>0.44</td>
<td>0.30</td>
</tr>
<tr>
<td></td>
<td>0.54</td>
<td>0.54</td>
<td>0.36</td>
<td>0.22</td>
</tr>
<tr>
<td></td>
<td>0.54</td>
<td>0.54</td>
<td>0.36</td>
<td>0.22</td>
</tr>
<tr>
<td></td>
<td>0.64</td>
<td>0.64</td>
<td>0.44</td>
<td>0.30</td>
</tr>
<tr>
<td></td>
<td>0.64</td>
<td>0.64</td>
<td>0.44</td>
<td>0.30</td>
</tr>
<tr>
<td></td>
<td>0.64</td>
<td>0.64</td>
<td>0.44</td>
<td>0.30</td>
</tr>
<tr>
<td></td>
<td>0.64</td>
<td>0.64</td>
<td>0.44</td>
<td>0.30</td>
</tr>
</tbody>
</table>

**Figure I.3**: Containing queries. Page size is 8KB.