Query Workload-Aware Index Structures for Range Searches in 1D, 2D, and High-Dimensional Spaces

by

Parth Nagarkar

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved April 2017 by the
Graduate Supervisory Committee:

Kasim Selçuk Candan, Chair
Hasan Davulcu
Maria Luisa Sapino
Mohamed Sarwat

ARIZONA STATE UNIVERSITY

August 2017

ABSTRACT

Most current database management systems are optimized for single query execution. Yet, often, queries come as part of a query workload. Therefore, there is a need for index structures that can take into consideration existence of multiple queries in a query workload and efficiently produce accurate results for the entire query workload. These index structures should be scalable to handle large amounts of data as well as large query workloads.

The main objective of this dissertation is to create and design scalable index structures that are optimized for range query workloads. Range queries are an important type of queries with wide-ranging applications. There are no existing index structures that are optimized for efficient execution of range query workloads. There are also unique challenges that need to be addressed for range queries in 1D, 2D, and high-dimensional spaces. In this work, I introduce novel cost models, index selection algorithms, and storage mechanisms that can tackle these challenges and efficiently process a given range query workload in 1D, 2D, and high-dimensional spaces. In particular, I introduce the index structures, *HCS* (for 1D spaces), *cSHB* (for 2D spaces), and *PSLSH* (for high-dimensional spaces) that are designed specifically to efficiently handle range query workload and the unique challenges arising from their respective spaces. I experimentally show the effectiveness of the above proposed index structures by comparing with state-of-the-art techniques.

DEDICATION

To my parents, and

the Thatte brothers, Bhagwan and Vikram

ACKNOWLEDGEMENTS

I would like to take this opportunity and express my gratitude to my advisor, Dr. K. Selçuk Candan. I would not have been to complete this journey without his mentorship and endless support. His feedback and insights were extremely vital in reaching my goal, and I will always be grateful to him for that.

I would specifically like to thank my committee member, Dr. Maria Luisa Sapino. I would also like to thank my committee members, Dr. Hasan Davulcu and Dr. Mohamed Sarwat. I would also like to wholeheartedly appreciate Dr. Paolo Papotti for his help on my defense day. I would also like to thank the graduate advisors at CIDSE and its technical staff members Lincoln Slade and Brint MacMillan for always being very helpful.

A big shout out to all the wonderful members of EmitLab that I was fortunate to interact with, namely, Xinxin Wang, Renwei Yu, Wei Huang, Juan Pablo Cedeno, Xilun Chen, Xinsheng Li, Mijung Kim, Yash Garg, Sicong Liu, Shengyu Huang, Silvestro Poccia, Ashish Gadkari, Mao-Lin Li, and Hans Behrens. I would also like to thank Jung Hyun Kim for his help. I am also very grateful for the help of Geoffrey Barbier. A special thank you to Aneesha Bhat and Mithila Nagendra for their helpful and positive attitude. I would like to take this opportunity to say how grateful I was to be a part of the ASU Graduate and Professional Student Association.

I am very thankful for Bhagwan Thatte, Vikram Thatte, Aparna Thatte, and Shachi Kale. They were my support pillars throughout my stay in Phoenix, and I could always count on them for any help. Lastly, my eternal gratitude towards my parents for always being supportive and being there for me whenever I needed them. I wouldn't have been able to finish my Ph.D. without them.

TABLE OF CONTENTS

CHAPTER Page

CHAPTER Page

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

Figure Page

Figure Page

Figure Page

Chapter 1

INTRODUCTION

Due to the ever increasing amount of data, there is a need for current database systems to be able to process large amounts of data in an efficient and scalable manner. This need gave rise to newer database systems, as well as newer paradigms [30, 89, 27, 21]. Each of these data processing systems try to tackle different problems in the big field of *big data*. As everything becomes digital in today's world, the amount of incoming queries have also increased along with the data load. Index structures are often used in database systems to speed up query processing. As the amount of data increases, there is a need for new efficient index structures that can tackle this ever increasing amount of data and queries. Index structures need to be designed in a scalable approach such that it can work on very large amount of data and queries. Index structures in existing database systems are created and optimized for single query execution [81]. Often times, multiple queries need to be executed as part of a query workload. This could be because of several reasons: (a) the amount of data in data warehouses can be large, and tables in these data warehouse environment can consist of several columns. Often, many queries have to be executed on separate columns for data analysis in data warehouses [22], (b) large number of users exist in a multi-client environment [10] with one or more queries associated to each user, or (c) continuous queries from a stream are batched as a query workload [81], etc. Queries in a query workload often have overlaps that can be leveraged for faster query processing. One of the main reasons this happens is due to overlapping regions of interests between users in a single environment [10].

Figure 1.1 shows a sample range query workload for a 2D space. As seen in the

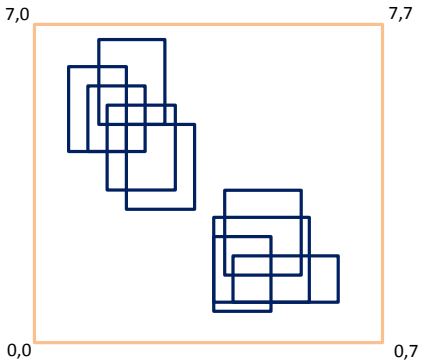


Figure 1.1: Sample Range Query Workload in a 2D Space

figure, often times the queries in a query workload have common overlaps between them. There is a need for an index structure that can take into consideration these overlaps and be able to process queries in an effective and scalable approach. There has been considerable research done to process query workloads consisting of specific types of queries, namely: top-k queries [81], range queries [25], group-by queries [22], and so on.

1.1 Shortcomings of Existing Techniques

The main goal of query workload optimization is to leverage the common data overlaps between different queries. There are several ways to tackle the problem of query workload optimization: (a) One way is to rewrite multiple queries into a single query [25] or using views [76, 52, 53]. In these type of approaches, the goal is to find ways to combine overlapping queries into a single query. Due to this, it is difficult to find the results of individual queries. (b) Another approach is to leverage the cache to effectively use intermediate results [71, 46]. Research in this field includes improving the performance of storage and retrieval of past query results for future queries that are stored in the cache. Approaches like these are not scalable since query results can easily exceed the cache size, especially when the query workloads are large. (c) One

more way is to identify common sub-expressions among different queries and then create an optimal global query plan based on these commonalities [77]. This problem has been shown to be NP-Hard [45], and heuristics-based approaches need to be used for query workloads larger than 10.

Another way of executing query workloads efficiently is to identify the most important parts of the index that will be used by multiple queries and bring those parts of the index into the memory for faster query execution. Given a query workload, the challenge is to develop effective query plans and efficient algorithms in order to identify the important parts of the index to be brought into the memory for fast query processing. These algorithms have to be fast and scalable in order to be effective. In my works [69] and [70], I have introduced novel query plans and index selection algorithms that can efficiently execute range query workloads for 1D spaces and 2D spaces respectively. I analyze the given query workloads using novel cost models and query plans, and then identify the most important parts of the index that need to be brought into the main memory and cached for efficient query processing. I experimentally show that existing database systems are not optimized for query workload execution, and my proposed cost models and algorithms are very effective and scalable.

Range queries are one of the most important general queries. Queries such as point or partial match queries are special types of range queries [78]. Due to their importance in different domains, considerable research has been done for improving range query performance. In the data warehouse environment, range queries are used as part of the select and aggregate query workload. Range queries in multi-dimensional spaces have also become very important as spatial and mobile applications gain popularity, thanks to the wide-spread use of mobile devices, coupled with increasing availability of very detailed spatial data (such as Google Maps and OpenStreetMap [73]),

and location-aware services (such as FourSquare and Yelp). For implementing spatial range queries (range queries in 2D spaces), many of these applications and services rely on spatial database management systems, which represent objects in the database in terms of their coordinates in 2D space. Queries in this 2D space are then processed using multidimensional/spatial index structures that help quick access to the data.

Query processing workloads in data warehousing environments often includes data selection and aggregation operations. When this is the case, column-oriented data systems are often the preferred choice of data organization. In a column-oriented system, each attribute is stored in a separate column. All the values of the attribute are stored successively on the disk. This is different from the traditional row-oriented systems where data belonging to the same tuple (from different attributes) are stored consecutively. This leads to the ability of compressing the data in a column that leads to reduction in I/O, which further leads to faster query processing [8]. Due to this benefit of compression, column-oriented systems have gathered a lot of attention in the research community as well as in the commercial world [86, 63, 101]. Bitmap indexes [93, 96] have been shown to be highly effective in answering queries in data warehouses [97] and column-oriented data stores [8]. There are two chief reasons for this: (a) first of all, bitmap indexes provide an efficient way to evaluate logical conditions on large data sets thanks to efficient implementations of the bitwise logical "AND", "OR", and "NOT" operations; (b) secondly, especially when data satisfying a particular predicate are clustered, bitmap indexes provide significant opportunities for compression, enabling either reduced I/O or, even, complete in-memory maintenance of large index structures. In addition, (c) existence of compression algorithms [31, 94] that support compressed domain implementations of the bitwise logical operations enables query processors to operate directly on compressed bitmaps without having to decompress them until the query processing is over and the results are to be

fetched from the disk to be presented to the user. Often times, the column domains in data warehousing environments are hierarchical in nature (e.g., geographical data, biological taxonomies, etc.). In such cases, using a hierarchy of bitmaps often leads to faster query processing due to the ability to choose higher bitmaps in the hierarchy for executing larger range queries [23]. Since bringing the entire hierarchy of bitmaps into the main memory would incur in a large I/O cost, there is a need to choose the right set of bitmaps from the hierarchy. The challenge then is, if given a query workload and a hierarchy of bitmaps, to come up with novel query plans and algorithms to choose the most optimal subset from the given hierarchy of bitmaps that can efficiently execute the given query workload. In this dissertation, I introduce the problem of range query workload execution in 1D and 2D spaces, and then introduce novel query plans and index selections algorithms that effectively execute a given range query workload in these spaces.

Range queries in high-dimensional spaces are a very important set of queries with applications in content-based systems of multimedia such as photos, videos, audio recordings, and sensor data [62]. Due to the well-known curse of dimensionality, most exact range query algorithms become slower than a linear scan as the number of dimensions increases. Hence, often times, an *approximate* range search is good enough to get results for a query, within an error bound, and is much faster than finding the exact results [29]. Due to these reasons, the problem of approximate range search has garnered significant amount of attention in the research community [90, 87, 44, 29, 62, 88]. A popular approach to solving approximate range queries is to represent the high dimensional data in a lower dimensional space, and do the query processing in this lower dimensional space. The data is then hashed to hash buckets in this space, with the idea that data points closer in the original space will be mapped to same bucket in the lower dimensional space. Locality Sensitive Hashing

(LSH)[44] is one of the most commonly used hashing techniques to solve approximate range queries. While Locality Sensitive Hashing has been studied and improved upon extensively [62, 98, 48, 59, 11, 29, 36], all the existing approaches are optimized for single query execution. In multimedia, it is more beneficial to represent data by a select group of points that can help identify the data easily. It is not necessary to store the entire data. These group of points are often generated by localized feature extraction algorithms such as SIFT [60] or SURF [12]. When a user is interested in finding a similar data object, a similarity query needs to executed on each of the features that represent this data object. These individual similarity query points can be collectively viewed as a set query. In traditional and state-of-the-art LSH-based techniques, users input a success guarantee for each individual query point, instead of a guarantee for the entire set query. A lower guarantee on each of these individual query points can lead to overall misses, and a stricter guarantee on these query points can lead to redundant and wasteful work for the entire set query (which can lead to slower query processing times). In Chapter 5, I present an index structure, *Point Set LSH (PSLSH)*, that is specifically designed to give guarantees for an entire set query while minimizing wasted and redundant work (thus improving the overall query execution time).

1.2 Research Contributions

The goal of this dissertation is to design and present novel index structures that are optimized for execution of range query workloads. I describe the unique challenges that occur in different types of spaces: namely, 1D, 2D, and high-dimensional spaces. I present innovative algorithms and storage mechanisms that are unique to solving the challenges occurring in the above mentioned 1D, 2D, and high-dimensional spaces. These index structures that are presented in this dissertation are shown to be scalable

for large data and large query workloads. In this dissertation, I will explain in detail these algorithms and storage mechanisms for each of the 1D, 2D, and high-dimensional spaces. I outline the research contributions of each of these works in the next sections.

1.2.1 Range Query Workloads in 1D Spaces

Data in data warehousing environments are often hierarchical in nature. Bitmaps are known to be very effective due to their ability to store data in a compressed manner which can reduce the IO cost. In earlier works [23, 24], the performance benefit of storing data as hierarchically organized bitmaps has been shown. Existing literature focuses on creating query plans by combining relevant nodes in a hierarchy for faster query execution (which I refer to as *inclusive* query plans). These plans are only suitable for smaller query ranges. In my work [69], I presented additional novel query plans (namely, *exclusive* and *hybrid*) that can answer query ranges of different sizes efficiently. These algorithms are explained in detail in Chapter 3. I further present algorithms that can choose a subset of the hierarchy that can most effectively execute a given range query workload. In this work, I also look at cases where there are no memory constraints and also real-life cases where there memory constraints. Experimental results show the effectiveness and efficiency of our proposed algorithms to execute a range query workload.

1.2.2 Range Query Workloads in 2D Spaces

Due to the popularity of geospatial applications such as Google Maps, Foursquare, etc., spatial range queries are very important. While there has been a lot of research in creating index structures to speed up spatial range queries, most of these index structures are not optimized for executing spatial range query workloads. In order to process spatial range query workloads, I leverage space-filling curves to convert the

2D data into a 1D space. Further, I create a hierarchy of bitmaps in this 1D space. Since these bitmaps can be large in number, I present a novel block-based storage mechanism for efficient storage. I further present novel block-based algorithms that choose the most effective subset of bitmaps that can help efficiently execute a given spatial range query workload. Experimental results of this novel index structure, called *compressed Spatial Hierarchical Bitmaps* (cSHB), show the efficiency of my approach when compared with spatial extensions of popular DBMSes.

1.2.3 Range Query Workloads in High-Dimensional Spaces

One of the most important techniques for dealing with the similarity search problem in high-dimensional spaces is Locality Sensitive Hashing (LSH). Locality Sensitive Hashing solves the approximate query problem in order to effectively answer range queries or similarity search queries in high-dimensional spaces. In most LSH-based works, the user has to input a success guarantee for individual query points, instead of a set of query points. An underestimation of this success guarantee on individual query points can lead to misses (and low accuracy) and an overestimation can lead to wasted work. In this dissertation, I present a novel index structure, *Point Set LSH (PSLSH)*, that uses a multi-level design to give a guarantee on an entire set of query points (instead of individual query points). I also present efficient cost models and design strategies that can effectively allocate resources such that the overall execution time is minimized, while guaranteeing a user-input success guarantee for the entire set of query points. Experimental evaluation shows the effectiveness and efficiency of PSLSH when compared to its alternatives.

1.3 Dissertation Outline

This dissertation is organized in the following way:

- In Chapter 2, I give an overview of the existing works in query workload optimization and range query processing in 1D, 2D, and high-dimensional spaces.
- In Chapter 3, I introduce and provide detailed novel query plans and algorithms (*HCS*) for efficient execution of range query workloads in 1D spaces.
- In Chapter 4, I identify the challenges for processing range query workloads in 2D spaces and present an innovative index structure (*cSHB*) to tackle the challenges.
- In Chapter 5, I describe the challenges for processing range query workloads in high-dimensional spaces and present a novel index structure (*PSLSH*) that effectively solves these challenges for efficient execution of range query workloads in high-dimensional spaces.
- In Chapter 6, I conclude this dissertation.

Chapter 2

RELATED WORK

In this chapter, I give an overview of the existing work in the fields of query workload optimization and range query processing in 1D, 2D, and high-dimensional spaces. Range queries are an important type of queries. In this dissertation, I look at the problem of executing range query workloads in 1D, 2D, and high-dimensional spaces. Each of these types of spaces have their own challenges. I give an overview of the works and different methodologies that are published to tackle the problem of query workload optimization. There has also been a lot of research in the area of developing index structures to efficiently execute range queries in these different types of spaces. I present an overview of these index structures in this chapter.

2.1 Query Workload Optimization

Often times, queries come as part of a query workload, and it is advantageous to execute them as part of the query workload instead of executing them individually. Considerable research has been done in the area of multi-query optimization in order to process query workloads faster. There are different proposed methods to achieve this optimization. It can be broadly divided into the following categories: (a) Given a set of queries in a query workload, one way to process the set of queries is to rewrite multiple queries into a single query [25, 34]. In [25], the idea is to find the common ranges between multiple range queries, and rewrite them into a single query. The drawback of approaches like this is that it is not easy to get the results of individual queries, which in many cases is necessary. In our works, our goal is to also get the

results of individual queries. Similar to this approach, another way of multiple query optimization is to rewrite the queries using views [76], [52] or a set of views [53]. (b) Another approach is to use an active cache that stores past query results that can be reused for future queries [46, 71]. The main goal of these approaches is to improve the storing and retrieval of intermediate results from the cache. While our works also use the cache for improved query processing, our goal behind using the cache is to save the most important parts of the index that will be used by multiple queries. My proposed index structures do not need to save intermediate results, which can require the size of the cache to be large. Like I show in this work, even for large amounts of data, our work only requires about the size of an L3 cache, which is available in most modern systems. (c) Another method is to identify common sub-expressions for a given set of queries, and create an optimal global execution plan based on the found commonalities [79, 77]. Any two sub-expressions of different queries can potentially have nothing in common, can be exactly same, can have some parts in common. In [45], this problem of finding common sub-expression has been shown to be NP-Hard. As the number of queries goes beyond 10, this problem can only be solved using heuristic methods. While their goal is also to reduce the cost of processing a query workload, their methodology is very different. In this work, I identify the important portions of the index that are going to be used by the query workload using novel cost models and query plans. I bring these portions of the index into the main memory and store in the cache in order to efficiently processing a query workload.

2.2 Range Query Workload Execution over Data Columns in 1D Spaces

Range queries are often used in data warehouse environments to perform operations that include aggregating a certain subset of data. In data warehouse environments, due to the ability of effective data compression, column-store architectures

are the preferred choice of data organization [8]. Compression of the data leads to reduction in the I/O cost, thereby speeding up the querying process. Building index structures over these data columns can further speed up the querying process. In [55], the authors propose a hierarchical index for execution of range-sum queries. The key idea of this work is to improve the update performance by reducing the number of cell accesses per query. This work focuses on improving the performance for a data cube. In [42], the authors describe a hierarchical data structure wherein they store the aggregate values of the leaf nodes in appropriate internal nodes. By doing this, they are able to stop searching at an internal node if all the values under the internal node are included in the range query. The main problem of this approach is that the aggregate values are stored in the internal nodes even for queries that do not need to do any aggregation. This results in wasted storage and computation thus reducing the performance of the queries. The work presented in [33] is an extension of [42]. In [33], the authors present a generalized index structure where they are able to use the information stored in the upper levels of the hierarchy. For queries that do not require aggregation, the aggregate values are not explicitly stored. The main focus of [66] is to cache aggregate results efficiently such as to be able to process both analytical and transaction query workloads in one system. In [41], the authors have presented novel techniques to execute range queries for different types of aggregation operations such as sum and max. In order to do this effectively, they use precomputed max over balanced hierarchical tree structures. In [23] and [24], the authors propose hierarchically organized bitmaps for efficient execution of OLAP queries for data with hierarchical domains. They identify internal nodes of the hierarchy that can speed up the query execution of OLAP queries. Their proposed query plan is what I term as an *inclusive* query plan. In [69], I present novel cost models and algorithms to choose a subset of bitmaps (and cache them in the memory) from a given hierarchy, to efficiently exe-

cute a given query workload. In addition to the *inclusive* query plan, I also propose *exclusive* and *hybrid* query plans to speed up the execution of range queries. Their work is also not optimized for range query workloads, whereas in this work, I identify nodes of the hierarchy that can efficiently execute a range query workload.

2.3 Range Query Workload Execution in 2D Spaces

2.3.1 Multi-level Index Structures using Bitmap Indexes

Bitmap indexes have been used in column store architectures for their benefit of compression, which further leads to improved I/O. There has been a lot of work on improving the performance and compression ratios of bitmap indexes [93], [95], [94], [75], [31], [26], [56], [50], [82]. Majority of the newer bitmap indexes use a compression scheme known as run-length encoding [93]. There are two benefits of using run-length encoding: 1) it has a good compression ratio, and 2) bitwise operations can be done on compressed bitmaps without having to decompress them [94]. Due to these benefits, bitmap indexes have been used in data warehousing environments and column-oriented architectures [95]. With these enhancements, bitmap indexes have also been shown to be effective for high cardinality data [96].

There has been some work done in the field of multi-level index structures that use bitmaps [82], [68], [83]. As mentioned earlier, in data warehouse environments, column-store architectures are a preferred choice of data organization. Bitmap indexes are used in column-store architectures for their benefit of compression and ability to do bitwise operations even on compressed bitmaps. It has been shown that bitmap indexes outperform the traditional index structures like B-tree in these environments [93], [97]. In [69] and [70], our goal is to efficiently choose a subset of bitmaps from a given hierarchy to execute a given query workload. I introduce different query

plans (namely, *inclusive, exclusive, and hybrid*). To the best of my knowledge, all the existing works focus on using the *inclusive* query plan. The main idea of the *inclusive* query plan is to choose higher level bitmaps that entirely satisfy the given query range and then choose the leaf level bitmaps for the boundary nodes in the query range to return the exact answer [75]. In [15], the authors also deal with the challenge of choosing the appropriate subset of bitmaps for multiple attributes, but their focus is on using data mining techniques in order to find them. In our work, our goal is the find the appropriate subset of bitmaps for a given domain hierarchy. There has been some work done on the creation of the hierarchies in a data warehouse environment for efficient execution of range queries [23], [24]. I assume that the hierarchies are given in the work presented in this chapter.

Bitmaps have also been used in spatial query processing. In [84], the authors propose an MBR-based spatial index structure named HSB-index. In this index, the leaves of the tree are encoded in the form of bitmaps. Just like in an R-tree [38], given a query, the HSB-index traverses down the hierarchical structure in a top-down manner to choose the appropriate bitmaps that are needed to be combined. In our work [70], I recognize that even internal nodes of the hierarchy can be encoded as bitmaps, which leads to improved performance for spatial range query workloads. Our work focuses on choosing the appropriate subset of bitmaps from the hierarchy (internal and leaf nodes) that can efficiently solve the given spatial range query workload. In [37], the authors describe algorithms for storing and retrieving large multidimensional HDF5 files. They use bitmap indexes in order to do the operations efficiently. They do support range queries on their architecture, but unlike our work, they don't leverage space-filling curves or hierarchical bitmaps. Also, none of the above mentioned works in spatial query processing are optimized for processing spatial range query workloads.

2.3.2 Multi-Dimensional Space Partitioning

Based on the partitioning strategy, multi-dimensional space partitioning can be broadly split into two categories: In the first category, a bounded region of the space is divided into multiple *"open"* partitions. Each of these partitions borders a boundary of the input region. Index structures such as Quadtree [35], G-tree [54], and k-d tree [17] fall into this category. In the second category, some of the boundary partitions are *"closed"*, i.e. these regions do not necessarily border any boundary of the input region. These partitions are often called *minimum bounding regions (MBRs)*. These MBRs can tightly cover the input data objects. Index structures such as R-tree [38] and its variants (R+-tree [80], R*-tree [14], Hilbert R-tree [49], etc.) are included in this category. There are two main drawbacks of these multi-dimensional index structures: 1) Overlaps between partitions (which causes redundant I/O), and 2) empty spaces within partitions (which causes wasted I/O). These two issues still exist in these data structures even after significant amount of research has been done [72]. One way to solve this challenge is to parallelize the index structures. In [9], the authors create a data warehousing system on top of Hadoop. This main goal of this system is the parallelization of the building of the R*-tree index structure and the subsequent query processing using Hadoop.

Most of the above mentioned index structures, as mentioned earlier, are optimized for single query processing. There has been some work done for the execution of query workloads as well. In [25], the authors extend the R-tree index structure to execute multiple range queries. But in their work, the authors combine adjacent queries into a single query. Hence, their algorithm is not able to distinguish the results of the individual queries in the query workload. In [74], the authors find the Hilbert values of the centroid of the rectangles formed by the given set of range queries. These

Hilbert values of the queries are then sorted, and the queries are then grouped in order to execute them over an R-tree.

2.3.3 Space Filling Curve based Indexing

As described in Chapter 4, I use a space-filling curve called Z-order [67] to convert the multi-dimensional space into a 1-dimensional space, in order to do the indexing and query processing on the 1-dimensional space. There are two commonly used space filling curves, one is the Z-order curve that I use, and the other is the Peano-Hilbert curve [39]. The Z-order curve maps the multi-dimensional space into a 1-dimensional space using a process called bit-shuffling. This process is very simple and efficient. On the other hand, even though the Hilbert curve generates a better mapping from the multidimensional space into a 1-dimensional space, it uses a more complicated and costly process [72]. Therefore, the Z-order curve has been used to tackle multi-dimensional problems. Space-filling curves have been used in several index structures to process spatial queries. In [72], the authors propose an index structure called BLOCK. The key assumption of this work is that the data and the index structure fit in the main memory. Hence, their goal is to reduce the number of checks that are done between the query range and the data. They use the Z-order to convert the data points into a 1-dimensional space, and then sort the list of the Z-order values. They start at the coarsest level for a given query. If a block in the index is included entirely in the query range, they retrieve the entries in this block, else they go to the next granular level. In our work [70], I don't assume that the data and the index can fit into the main memory. Hence, our cost model takes into account the I/O cost as well. In [100], the authors build a system called VegaGiStore on top of Hadoop to process spatial queries in parallel. This system creates a bi-level index structure. In the first level, they create a quadtree-based global index that is used for finding

the required part of the data. In the second level, they use a local index that uses the Hilbert curve for finding the spatial objects in the retrieved part of the data. In [13], the authors create an index structure called UB-tree. This index structure uses the Z-order curve to store multi-dimensional data in a B-tree, and do subsequent query processing on the B-tree. In [85], the authors present a range query algorithm that is specifically optimized for this UB-tree. The UB-tree is also used in the work presented in [65]. In this work, the authors create a hierarchical clustering scheme for the fact table of a data warehouse. The data in this fact table is stored using the UB-tree. Unlike our work [70], none of the above approaches are optimized to handle a spatial range query workload.

2.4 Range Query Workload Execution in High-dimensional Spaces

Efficient implementations of range and nearest neighbor queries are critical in many large data applications. Tree-based indexing methods (such as KD-tree [17], X-tree [18], SR-tree [51], etc.) have been shown to be effective for lower dimensions (dimensions less than 10), but suffer from the *curse of dimensionality* as the number of dimensions increases – in fact, they are often outperformed even by a linear scan [44]. One solution to address this problem is to look for *approximate* results using approximate indexing techniques, such as VA-files [92] and LSH [44].

2.4.1 Locality Sensitive Hashing (LSH)

The problem of approximate range searches and nearest neighbor queries gained importance as it is much efficient to retrieve *good enough* results in much lesser time. Locality Sensitive Hashing (LSH), first introduced in [44], is a popular technique that hashes similar data points into same buckets than dissimilar points. Since when using only one hash function, a lot of false positives may be generated, in order to reduce

the number of false positives, k different hash functions are conjunctively combined. Then, in order to satisfy the recall guarantee, multiple layers of such hash functions (called hash layers) are created. For any query point q, all the points in the hash buckets that the query point q hashes to are retrieved for all hash layers. These candidate points potentially contain false positives, and hence in order to remove the false positives, a filtering step is required where the distance of the candidate point and the query point q is calculated.

While the original LSH scheme [44] was proposed for binary Hamming spaces, the authors then extended the scheme for Euclidean spaces [29], and since then LSH has also been proposed for other distance and similarity measures [91]. While LSH was originally designed to solve the (r, c)-Near Neighbor problem (introduced in Section 5.1.4), it has also been used to solve other related problems. In fact, most of the following works (except the notable exceptions of E2LSH[1], C2LSH [36], and QALSH [43]) have been primarily designed to solve the c-Approximate Nearest Neighbor problem [87, 62] (c-k-ANN), where the goal is to find k neighbors such that the distance of the query point and the ith nearest neighbor is at most $(1 + c)$ times the distance from the query point to its true ith nearest neighbor.

2.4.2 Efficient Variants of LSH

Due to its effectiveness in supporting range and nearest neighbor queries in high-dimensional spaces, there are several widely used implementations of LSH. In [11], the authors propose to create a prefix-tree of hash functions for each hash layer. By using a prefix-tree, the authors are able to decide during query processing on how many hash functions to use in order to return the desired number of top-m results. In [62], the authors propose a probing sequence for the hash buckets in order to get

[1] `http://www.mit.edu/~andoni/LSH/`

the desired number of top-m results. The intuition is that similar data points lie in neighboring hash buckets, and hence by probing neighboring buckets they are able to retrieve more results, while creating less number of hash layers. In [88], the authors project the original points into a new space using hash functions. Each point in this space is represented using a z-order code, and the points are further retrieved based on the similarities between their z-order codes. In [59], the goal of the authors is to reduce the number of random I/O operations that are needed to retrieve candidate data points. They propose a distance measure between the compound hash keys (CHKs). They also propose a linear order on these CHKs, which are then stored in the ascending order on the secondary storage. Their idea is that if similar CHKs are stored on the same page, the total number of I/O operations will be reduced. In QALSH [43], the authors propose to build hash functions that are "query-aware", i.e., the bucket of the hash functions are created based on an input query point. In C2LSH [36], the authors propose to build a base of LSH functions, and then the points that collide most frequently (based on a count threshold) with the query point in these base functions are chosen as candidate points. By using this concept of "collision counting", the number of candidate points are reduced without having the need to have large number of layers. This approach has been shown to be more effective by generating less number of candidates, using smaller index structures, than the original LSH index structure [36, 43]. In this work, using effective parameters based on our proposed cost models, I build upon this concept of *collision counting*, and further reduce the number of candidates and the query processing time for processing a query set.

To the best of my knowledge, no existing work tries to solve the problem of giving guarantees on entire query sets in high-dimensional spaces. In [99], the authors propose an LSH-based index structure for solving multiple queries with different distance metrics. None of these works are designed to provide a guarantee for the entire query set, and hence can generate excessive candidates resulting in slow processing times.

Chapter 3

EXECUTION OF RANGE QUERY WORKLOADS IN 1D SPACES

3.1 Introduction

Range selection queries are frequent in many applications, including online analytical processing (OLAP) scenarios, where an aggregation operation needs to be applied over a certain range of data [41]. When data are large and the query processing workloads consist of such data selection and aggregation operations, column-oriented data stores are generally the preferred choice of data organization, especially because they enable effective data compression, leading to significantly reduced IO [8].

Recently, many databases have leveraged bitmap-indices, which themselves can be compressed, for efficiently answering queries [3], [2]. When column-domains (e.g., geographical data, categorical data, biological taxonomies, organizational data) are hierarchical in nature [24], it is often more advantageous to create *hierarchical bitmap indices* to efficiently answer queries over different sub-ranges of the domain. [24] for example proposes a hierarchically organized bitmap index (HOBI) for answering OLAP queries over data with hierarchical domains.

In this chapter, I also focus on hierarchically organized bitmap indices for answering queries over column-oriented data and present efficient algorithms for selecting the subset of bitmap indices to answer queries efficiently over compressed data columns.

3.1.1 Contributions of this Work

Since IO is often the main bottleneck in processing OLAP workloads over large data sets, given a query or a workload consisting of multiple queries, the main chal-

lenge in leveraging hierarchically organized bitmap indices is to choose the appropriate subset of bitmap indices from the given hierarchy to process the query. [24], for example, proposes a (what I term as an "*inclusive*") strategy which leverages bitmap indices associated to the internal nodes along with the bitmap indices associated to the data leaves to bring together the data elements needed to answer the query.

In this chapter, I note that such inclusive strategies can be sub-optimal. In fact, [24] shows that the inclusive strategy is effective mainly for small query ranges. Therefore, I introduce a more general *cut-selection* problem, which aims to help identify a subset (referred to as a *cut*) of the nodes of the domain hierarchy, which contain the operations nodes with the appropriate bitmap indices to efficiently answer queries. In particular, I discuss *inclusive*, *exclusive*, and *hybrid* strategies for cut-selection (Section 3.3.1) and experimentally show that the so-called *exclusive strategy* provides gains when the query ranges are large and that the *hybrid strategy* provides best solutions across all query range sizes, improving over the *inclusive* strategy even when the ranges of interest are relatively small (Section 3.4.1). I also show that the *hybrid strategy* can be efficiently computed for a single query or a workload of multiple queries and also that it returns optimal (in terms of IO) results in cases where there are no memory constraints (Section 3.4.2).

However, in cases where the memory is constrained, the cut-selection problem becomes difficult to solve. To deal with these cases, in Section 3.2.3, I present efficient cut-selection strategies that return close to optimal results, especially in situations where the memory limitations are very strict (i.e., the data and the hierarchy are much larger than the available memory).

Experiment results presented in Section 3.4 confirm the efficiency and effectiveness of the proposed *cut-selection* algorithms.

3.2 Problem Specification

In this section, I first introduce the relevant concepts and notations, provide a cost model, and introduce the *cut-selection* problem for identifying a subset of the nodes of the domain hierarchy, containing the nodes with the bitmap indices to efficiently answer a given query or a query workload.

3.2.1 Key Concepts, Parameters, and Notations

I first provide an overview of the concepts and parameters necessary to formulate the problem described in this chapter and introduce the relevant notations.

Columns and Domain Hierarchies

A database consists of relations, $\mathcal{R} = \{R_1, \ldots, R_{maxr}\}$. Each relation, R_r, consists of a set of attributes, $\mathcal{A}_r = \{A_{r,1}, \ldots, A_{r,maxa_r}\}$, with domains $\mathcal{D}_r = \{D_{r,1}, \ldots, D_{r,maxa_r}\}$ In this chapter, without loss of generality, I associate to each attribute, $A_{r,a}$, a corresponding hierarchy, $H_{r,a}$, which consists of a set of nodes, $\mathcal{N}_{r,a} = \{N_{r,a,1}, \ldots, N_{r,a,maxn_{r,a}}\}$. Also, since the goal is to efficiently answer queries over a single data column, unless necessary, I omit explicit references to relation R_r and attribute $A_{r,a}$; hence, when I do not need to refer to a specific relation and attribute, I simply omit the relation and attribute subscripts; e.g., I refer to H instead of $H_{r,a}$.

In this chapter, when talking about the nodes of a domain hierarchy H, I use the following notations:

- **Parent of a node:** For all N_*, $parent(N_*)$ denotes the parent of N_* in the corresponding hierarchy; if N_* is the root, then $parent(N_*) = \perp$.
- **Descendants of a Node:** The set of descendants of node n in the corresponding hierarchy is denoted as $desc(n)$.

- **Leaves:** L_H denotes the set of leaf nodes of the hierarchy H. Any other node in H that is not a leaf node is called an internal node. The set of internal nodes of H is denoted by I_H. *I assume that only the leaves of a hierarchy occur in the database.*

- **Leaf Descendants of a Node:** Leaf descendants of a node are the set of nodes such that they are leaf nodes as well as descendants of the given node; i.e., for a node n, $leafDesc(n)$ returns a set of nodes such that

$$\forall_{b \in leafDesc(n)} b \in L_H \wedge b \in desc(n).$$

Query Workload

In this chapter, I focus on query workloads with range queries on an attribute (i.e., column) of the database relations:

- **Range Specification:** Given an attribute A_a and the start and end points, i and j, I denote the corresponding range specification as, $rs_{a,i,j}$.

  Given two range specifications, $rs_{a,i,j}$ and $rs_{a,k,l}$,

  - if $k > j$, then these two range specifications are **disjoint**,
  - if $(i < k, l) \wedge (j > k) \wedge (j < l)$, then the two range specifications are **intersecting**, and
  - if $(i < k, l) \wedge (j > k, l)$, then the two range specifications are **overlapping**.

- **Range Queries:** Each query q involves fetching one or more sets of column values, such that each set of values belongs to a continuous range over the domain hierarchy of the attribute.

  A query, q, can have multiple range specifications. The set of range specifications for a query q is denoted as RS_q. Without loss of generality, I assume that

all range specifications in RS_q are *disjoint*. If a query has two intersecting or overlapping range specifications, $rs_{a,i,j}$ and $rs_{a,k,l}$, then I partition the query into two subqueries, q_1 and q_2, such that range specification for q_1 is $rs_{a,i,j}$ and specification for q_2 is $rs_{a,k,l}$. In Sections 3.3.2 and 3.3.3, I discuss algorithms for handling multiple queries.

- **Range Nodes:** Given a range specification, $rs_{a,i,j}$, the set of *leaf nodes* that fall in this range is denoted as, $RN_{a,i,j}$. These nodes are also referred to as *range nodes*.

  Given a query, q, and a node n, $G_{q,n} \in \{0, 1\}$ denotes whether the node n is a range node for query q. More specifically, if node n is a range node for any range specification in RS_q, then $G_{q,n} = 1$ and otherwise, $G_{q,n} = 0$.

  The set of all range nodes for any range specification of query q is denoted as RN_q. If RN_q is empty, the query returns null, whereas if RN_q has the exact same nodes as L_H, then the query returns the entire database content for the attribute on which H is defined.

Hierarchically Organized Bitmap Indices

As described above, the query workload includes queries that fetch ranges of values from columns of relations in the database, before performing further operations on these ranges. When bitmap indices are available, these operations are implemented in terms of bitmap manipulations [75]: for example, intersection of two range queries can be performed as bitwise-AND of two bitmap indices representing the database values in the two ranges. This ensures that those data objects that will be pruned as a result of the query processing are never fetched into memory. In this work, I assume that indices are organized hierarchically; i.e., every node n in H has a corresponding

bitmap B_n denoting which of the leaf nodes of n occur in attribute A of the database.

- **Bitmap Density:** Each bitmap B_n has a bit density, $0 \leq D_{B_n} \leq 1$, denoting ratio of bits set to 1 to the size in the bitmap.

Note that bitmap B_n *may or may not* have been materialized in the form of a bitmap index in the database.

3.2.2 Cost Model and Query Plans

Especially when the data sets are large, the bitmaps are often stored in a compressed manner and the various bit-wise operations are performed on compressed versions of the bitmap indices, further boosting the query performance [93]. In general, the time taken to read the bitmaps from secondary storage into the memory dominates the overall bitwise manipulation time [75], [31]. The cost of this process is proportional to the size of the bitmap file on the secondary storage; the larger the size of a bitmap file on a secondary storage, the longer it takes to bring the bitmap into the physical memory.

Read Cost of Compressed Bitmap Indices

Therefore, I model the cost of a bitmap operation as proportional to the size of the corresponding (compressed) bitmap file, which in turn determines the time taken to read a bitmap into the memory. Note that in general the query performance of a bitmap index with density greater than 0.5 is equivalent to the performance of a bitmap with density complement to the original [94]. For example, performance of a bitmap with density 0.7 is often equivalent to the performance of a bitmap with density 0.3. This is because a bitmap with density 0.7 can be negated and stored as a bitmap with density 0.3. I also include this behavior in our cost model, $readCost(B_n)$,

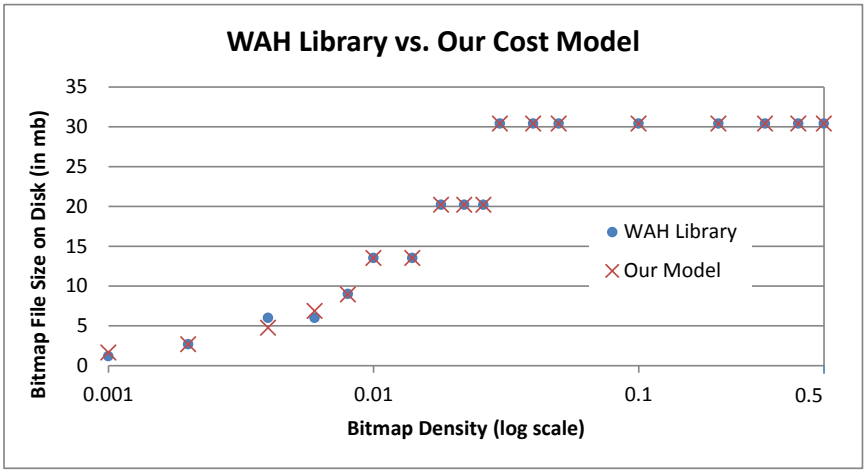


Figure 3.1: Comparison of Our Cost Model and WAH Library Model. $D_{x_1} = 0.01$, $D_{x_2} = 0.015$, $D_{x_3} = 0.03$, and $a = 1043, b = 0.5895$, on a 500 GB SATA Hard Drive with 7200 RPM, and 16 MB Buffer Size.

of reading a bitmap index, B_n, as follows:

$$\begin{cases} 0 & \text{if } D_{B_n} = 0 \vee D_{B_n} = 1 \\ aD_{B_n} + b & \text{if } (0 < D_{B_n} \leq D_{x_1}) \vee (1 - D_{x_1} \leq D_{B_n} < 1) \\ k_1 & \text{if } (D_{x_1} < D_{B_n} \leq D_{x_2}) \vee \\ & (1 - D_{x_2} \leq D_{B_n} < 1 - D_{x_1}) \\ k_2 & \text{if } (D_{x_2} < D_{B_n} \leq D_{x_3}) \vee \\ & (1 - D_{x_3} \leq D_{B_n} < 1 - D_{x_2}) \\ k_3 & \text{otherwise} \end{cases}$$

Here D_{B_n} is the bit density, $0 < D_{x_1} < D_{x_2} < D_{x_3} < 0.5$ are three bit density thresholds, and a, b, k_1, k_2, and k_3 are constants.

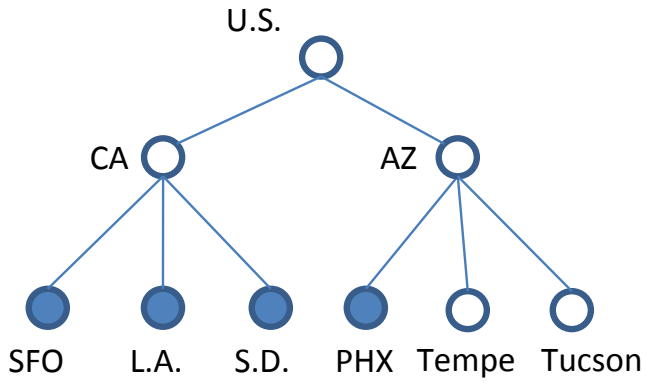
Intuitively, when the bit density of a bitmap is 0 or 1, the size of the bitmap on the disk is very negligible due to the high-level of compression. Hence, I assume the size of these bitmaps as non-existant on the secondary storage. The bit density thresholds,

D_{x_1}, D_{x_2}, and D_{x_3}, and the constant values, a, b, k_1, k_2, and k_3, are specific to the implementation of the bitmap library.

Figure 3.1 shows alignment of our cost model with the read cost of the WAH library for different bit densities.

Inclusive, Exclusive, and Hybrid Query Plans

For a query plan for q, I define the set of nodes that are required to execute q as its *operation nodes*. Naturally, a given query can be executed in various different ways, each with a different set, ON_q, of *operation nodes*. In particular, I consider two distinct types of query plans: *inclusive* and *exclusive* plans.



Consider the 3-level location hierarchy, H, shown above. Here, the leaf nodes (cities in U.S.) are the actual values in the database. The node *U.S.* is the root node of the hierarchy. Let us consider a query q that has a set of range nodes (shaded nodes in the figure) RN_q =[SFO, $L.A.$, $S.D.$, PHX]. Assume that I have bitmap indices for all the nodes of H. There are at least two different plans of executing q:

- *Inclusive query plans:* The first plan is to combine a subset of the bitmaps of H. In the above example, one inclusive way to do this would be to combine the bitmaps of RN_q.

Another inclusive plan would be to combine the bitmaps of CA and PHX (i.e. CA OR PHX). Note that this strategy is similar to what was reported in the literature [24].

- *Exclusive query plans:* Alternatively, I can remove the bitmaps of the non-range nodes of q from the relevant internal nodes of H. For instance, in this example, I can achieve this by first performing a bitwise-OR operation on the bitmaps of $Tempe$ and $Tucson$ and then doing a bitwise-ANDNOT operation between the bitmap of $U.S$ and the resultant bitmap from the OR operation (i.e. $U.S$ ANDNOT ($Tempe$ OR $Tucson$)).

  Another exclusive plan would be to do the following:
  CA OR (AZ ANDNOT ($Tempe$ OR $Tucson$).

It is easy to see that all four plans would return the same result; however, these plans have different operation nodes: for the first inclusive query plan, the operation nodes are $ON_q = [SFO, L.A., S.D., PHX]$, whereas for the second inclusive query plan, $ON_q = [CA, PHX]$. Similarly, for the first exclusive query plan $ON_q = [U.S., Tempe, Tucson]$, and for the second exclusive query plan $ON_q = [CA, AZ, Tempe, Tucson]$. Since different nodes are required, each execution plan also requires different amount of data being read.

In this chapter, I consider inclusive and exclusive strategies for answering range queries using hierarchical bitmaps. I also consider *hybrid* strategies, which combine inclusive and exclusive strategies (that may make inclusive or exclusive decisions at different nodes of the hierarchy) for better performance.

3.2.3 Cut Selection Problem

As described above, any range query, q, on hierarchy H, can be answered (through inclusive, exclusive, and hybrid strategies) using bitmap indices for the leaves of the hierarchy. I note however that, if the bitmap indices are also given for a subset of the internal nodes of the hierarchy, I may be able to reduce the overall cost of the query significantly by also leveraging the bitmap indices for these internal nodes. I refer to these subsets as *cuts* of the hierarchy.

Query Processing with Cuts

A *cut*, c, is defined as a subset of internal nodes (including the root node) in a hierarchy, H, satisfying the following two conditions:

- *validity:* there is exactly one node on any root-to-leaf branch in a given cut (note that, by this definition, the set containing only the root node of the hierarchy by itself is a cut); and
- *completeness:* the nodes in c collectively cover every possible root-to-leaf branch in the given hierarchy, H.

If a set of internal nodes of H only satisfies the first condition, then the cut is referred to as an *incomplete cut*.

The challenge of course is to select the appropriate cut c of the hierarchy H that will minimize the query processing cost, but will not add significant memory overhead (if the memory is a constraint). I discuss the alternative formulations of the cut-selection problem, next.

Cut Selection Case 1: Single Query without Memory Constraints

The simplest scenario is identifying the cut necessary to execute a single range query. As explained earlier, the cost for executing a query is proportional to the size of the bitmaps that are read into the memory from the secondary storage. Thus, given a query q and cut c on H, problem

$$cost(c,q) = \underset{ON_q \subseteq (c \cup L_H)}{MIN} \left\{ \sum_{n \in ON_q} readCost(B_n) \right\} \tag{3.1}$$

denotes the *best execution cost* for query q given the bitmaps for the leaves, and the cut c. The cut-selection problem for a given query q on hierarchy H can be formulated as *finding a cut c such that $cost(c,q)$ is the smallest among all cuts of H.*

Cut Selection Case 2: Multiple Queries without Memory Constraints

In general, I am not given a single range query, but a set of range queries that need to be executed on the same data set. Therefore, the above formulation needs to be generalized to scenarios with multiple range queries. If a set, Q, of queries is given on a hierarchy H, then one way to formulate the cut-selection problem is *to search for a cut c such that $cost(c,Q)$, defined as*

$$cost(c,Q) = \sum_{q \in Q} cost(c,q) \tag{3.2}$$

is the smallest among all cuts of the hierarchy H.

Note, however, that this formulation treats each query independently and implicitly assumes that each query plan accesses the bitmaps of its operation nodes from the secondary storage; i.e. it pays the cost of reading a bitmap from the secondary storage every time the node is needed for query processing. This will obviously be redundant when the different queries can be processed using the same operation nodes:

in such a case, *it would be best to bring the bitmap for the operation nodes to the memory and keep it to process all the relevant queries.*

This, however, changes the problem formulation significantly; in particular, I now need *to search for a cut c such that cost′(c, Q), defined as*

$$\left(\sum_{n \in c} readCost(B_n)\right) + \left(\sum_{n \in (\cup_{q \in Q} ON_q)/c} readCost(B_n))\right) \tag{3.3}$$

is the smallest among all cuts of the hierarchy H. Intuitively, the cut is read into the memory once and for each query in Q the remaining operation nodes are brought to the memory as needed. The first term in equation 3.3 is the cost of reading the bitmaps of the nodes in c from the secondary storage into the memory. Once these bitmaps have been read into the memory, I reuse them for further query processing, i.e. the bitmaps of the cuts need to be read into the memory only once. The second term denotes the cost of reading remaining bitmaps from the secondary storage every time it is needed to execute a query. These remaining bitmaps are also read only once and cached subsequently for further re-use for queries in the workload.

Cut Selection Case 3: Multiple Queries with Memory Constraints

The above formulations do not have any memory availability constraints; i.e., as many bitmaps as needed can be read and cached in memory for the given workload. In general, however, there may be constraints on the amount of data I can cache in memory. Therefore, I next consider scenarios where I have a constraint on the amount of memory that can be used during query processing. Let us assume that I have a memory availability constraint S_{total}. Every bitmap has a size associated to it, S_{B_n}, denoting the memory requirement of the bitmap file of node n in the main memory. Given a query workload Q and S_{total}, *I want to find a (potentially incomplete) cut c*

that minimizes the following cost:

$$\left(\sum_{n\in c} readCost(B_n)\right) + \left(\sum_{q\in Q}\sum_{m\in ON_q/c} readCost(B_m)\right) \tag{3.4}$$

subject to

$$\sum_{n\in c} S_{B_n} \leq S_{total} \tag{3.5}$$

Note that the bitmaps for c are read into the memory once and for each query in Q the remaining operation nodes are brought to the memory as needed. The major difference from before is that due to the constraint on the size of the nodes that can be maintained in memory, c may be an incomplete cut. Moreover, the operation nodes that are not in the cut cannot be cached in memory for reuse (unless $S_{total} > \sum_{n\in c} S_{B_n}$).

3.3 Cut Selection Algorithms

As described in the previous section, query execution times can be reduced if I am also given the bitmap indices for a subset of the nodes in the domain hierarchy of the column. A key challenge is to select the appropriate subset (or *cut*) of the hierarchy H to minimize the query processing cost, without adding significant memory overhead. In this section, I present algorithms that search for a cut, c, given a query q or a workflow of queries Q. It is important to note that these algorithms do not directly return the operation nodes required to execute q; instead they aim to find a cut, c, such that there exists a set of operation nodes $ON_q \subseteq (c \cup L_H)$ with a small cost. Once a good cut of hierarchy is found, the necessary operation nodes ON_q are identified in post-processing by searching within the cut c.

3.3.1 Case 1: Single Query without Memory Constraints

As described in Section 3.2.2, queries can be processed using inclusive, exclusive, and hybrid strategies. In this subsection, I first provide three algorithms, corresponding to the previously mentioned strategies, for the basic scenario where there is a single query without any memory constraints.

Inclusive Cut Selection (I-CS) Algorithm

The *inclusive cut selection (I-CS)* algorithm associates an *inclusive cost* to all nodes of the hierarchy and selects the cut using these inclusive costs. Given node v of the hierarchy H, let $l(v) = \{m \| (m \in leafDesc(v)) \wedge (G_{q,m} = 1)\}$. Formally, given a query q and a node n on hierarchy H, I define the inclusive cost, $nodeInclCost(n, q)$, of the node in the cut as follows:

$$\begin{cases} \infty & \text{if } \forall_{m \in leafDesc(n)} G_{q,m} = 0 \\ readCost(B_n) & \text{if } \forall_{m \in leafDesc(n)} G_{q,m} = 1 \\ \sum\limits_{\substack{m \in leafDesc(n) \\ \wedge G_{q,m}=1}} readCost(B_m) & \text{otherwise} \end{cases}$$

Note that the inclusive cost is only applicable for the internal nodes of a hierarchy; it is undefined for a leaf node.

In Alg. 1, I present the outline of the proposed algorithm which uses the above definition of inclusive cost to find a cut c that gives the optimal cost to execute a single range query q. Note that since a valid cut does not include any leaf nodes, the algorithm considers only the set of internal nodes, I_H, of hierarchy H.

The *inclusive cut selection algorithm* presented in Alg. 1 is a dynamic programming solution that traverses the nodes in the hierarchy in a bottom-up manner:

- In line 5 of the pseudo-code, the set *children* is empty for a node on the second-

Algorithm 1 Inclusive Cut Selection Algorithm

```
1: Input: Hierarchy H, Set of internal nodes I_H, Query q
2: Output: Set of nodes c
3: Initialize: Node n = root, c
4: procedure findNodeInclusiveCut(n)
5:     Set children = findChildren(n, I_H);
6:     if children is empty then
7:         add n to c;
8:         return nodeInclCost(n, q);
9:     else
10:        costChildren = 0;
11:        for each child m of n do
12:            costChild = findNodeInclusiveCut(m);
13:            if costChild ≠ ∞ then
14:                costChildren = costChildren + costChild;
15:            end if
16:        end for
17:        if costChildren = 0 then
18:            costChildren = ∞;
19:        end if
```

```
20:         costCurrNode = nodeInclCost(n, q);
21:         if costCurrNode ≤ costChildren then
22:             remove all descendants of n from c;
23:             add n to c;
24:         end if
25:         return min(costCurrNode, costChildren)
26:     end if
27: end procedure
```

to-last level of the hierarchy H, since the input to the function $findChildren$ is the set of internal nodes I_H. Whenever the set *children* is empty, I add the current node to the cut c, and return the inclusive cost of the current node.

- The condition on line 13 makes sure that the cost of children of n does not include the cost when a child m has the cost ∞. This will happen when none of the nodes in $leafDesc(m)$ is a range node, i.e. q does not want the contents of m to be included in the result of the query.

- The condition on line 17 will be true if for every child m of n, $nodeInclCost(m) = \infty$. This also means that no node in $leafDesc(n)$ is a range node. In such a case, I want the total cost of all the children of n to be equal to ∞.

- The algorithm then compares the inclusive cost of the parent with the inclusive cost of the set of its children. If the inclusive cost of the parent is cheaper than the combined inclusive cost of its children, then I remove the descendants of n from c and add n to c. Otherwise, I keep the cut as it is, since using the children of n is cheaper than using n.

If the resulting c contains only the root node of the hierarchy, then it means that

using the leaves is the cheapest option.

Note that the algorithm is very efficient: each internal node in the hierarchy is considered only once and for each node only its immediate children need to be considered; moreover, the function $nodeInclCost()$, which is called for each node, itself has a bottom-up implementation with $O(1)$ cost per node assuming that node densities for each internal node has been computed ahead of time Consequently, the cost of this algorithm is linear in the size of the hierarchy, H.

Exclusive Cut Selection (E-CS) Algorithm

Above, I considered the inclusive strategy which uses bitwise OR operations among the selected bitmaps to execute the query q. As I see in Section 3.4.1, this option may be costly when the query ranges are large. Alternatively, I can identify query results using an exclusive strategy: For a given query q, consider a leaf node m such that $G_{q,m} = 0$. That means that this node is not a range node. I call the leaf nodes (like m), which are outside of the query range, the *non-range nodes* and denote them as NS_q. The values of these leaf nodes are part of the actual data that q does not want to be displayed in the result. The exclusive strategy, initially introduced in Section 3.2.2, would first identify the non-range leaf nodes and then use the rest to identify the query results.

Like the inclusive cost, I associate an exclusive cost to all internal nodes of the hierarchy. Consider an internal node n of the hierarchy. If every node in $leafDesc(n)$ is a range node, that means that the q wants the content of n to be included in the result of the query, i.e. $leafDesc(n)$ does not contain any non-range node. In this case, I do not need to remove any node from n, and thus, the exclusive cost of n is the read cost of the node n. Note, that in the same scenario, the inclusive cost of n is also the read cost of n. If, in contrast, none of the leaf descendants of n is a range

node, then the query results will not include n and in this case, the *node exclusive cost* of n can be said to be ∞. The main difference is the scenario when only some of $leafDesc(n)$ are non-range nodes. In this case, the exclusive strategy removes the non-range nodes from n, and thus, the exclusive cost of n is the read cost of reading all the non-range nodes under n, in addition to the read cost of n. Based on these, I can formulate the node exclusive cost, $nodeExclCost(n, q)$ as follows:

$$\begin{cases} \infty & \text{if } \forall_{m \in leafDesc(n)} G_{q,m} = 0 \\ readCost(B_n) & \text{if } \forall_{m \in leafDesc(n)} G_{q,m} = 1 \\ readCost(B_n) + & \sum_{m \in leafDesc(n) \wedge G_{q,m}=0} readCost(B_m) \\ & \text{otherwise} \end{cases}$$

Given these node exclusive costs (which can again be computed in O(1) time per node using a bottom-up algorithm), an optimal exclusive cut can be find using a linear time algorithm similar to the node inclusive cut algorithm presented in Alg. 1; the main difference being that each internal node in the hierarchy is associated with an exclusive cost, instead of an inclusive cost. In this case, the results would be a cut c such that reading every node in $ON_q \subseteq (c \cup NS_q)$, I can execute the query q optimally using the exclusive strategy. If the output cut c is the root node of the hierarchy, then every node in NS_q has to be removed, i.e. an ANDNOT operation has to be done between the root node and the nodes in NS_q.

Hybrid Cut Selection (H-CS) Algorithm

So far, I have considered inclusive and exclusive strategies independently from each other. However, we could consider both inclusive and exclusive strategies for each node in the hierarchy and associate the better strategy to that node. In other words, I could modify the linear-time, bottom-up algorithm presented in Alg. 1 using the following cost function for each internal node of the hierarchy, H:

$$nodeHybridCost(n, q) = min(\quad nodeInclCost(n, q), \\ nodeExclCost(n, q)).$$

Unlike when searching for the inclusive or exclusive cuts of the hierarchy, during the traversal, I also need to mark each node as an *inclusive-preferred* or *exclusive-preferred* node based on the contributor to the hybrid cost. Naturally, in this case the resulting cut, c, can be partitioned into two: an *inclusive cut*, c_i (whose nodes are considered in an inclusive way), and an *exclusive cut*, c_e (whose nodes are considered under the exclusive strategy). Those nodes that have a lower inclusive cost are included in c_i, whereas those that have a lower exclusive cost are included in c_e.

- If no $leafDesc(n)$ is in the range, then I call n, an *empty node.* An empty node is not used in any query processing and is ignored.

- If all of $leafDesc(n)$ are in the range, then I call n, a *complete node.* A complete node indicates that all the leaf descendants of the node are needed for query processing. Hence, both the inclusive and the exclusive costs of a complete node are same.

- If only some of the $leafDesc(n)$ are part of the range, then I call n, a *partial node.* Note that the only time n will have potentially different inclusive and

exclusive costs is when n is a *partial node*. If a node is a partial node, I find both the inclusive and exclusive costs, and choose the minimum of the two costs. Subsequently, whichever cost is chosen, I label the node accordingly as part of the inclusive or the exclusive cut. This helps us in efficiently finding the operation nodes as described further.

Algorithm 2 Finding the Operation Nodes

```
1: Input: Set of nodes c, Query q
2: Output: Set of operation nodes ON_q
3: Initialize: ON_q
4: procedure findOperationNodes(c, q)
5:     for each node n in c do
6:         if n is a complete node then
7:             add n to ON_q;
8:         else if n is a partial node then
9:             inclusiveCost = nodeInclCost(n, q);
10:            exclusiveCost = nodeExclCost(n, q);
11:            if inclusiveCost ≤ exclusiveCost then
12:                add every node from nodeInclusiveCut(n, q) to ON_q;
13:            else
14:                add every node from nodeExclusiveCut(n, q) to ON_q;
15:            end if
16:        end if
17:    end for
18:    return ON_q
19: end procedure
```

As I mentioned earlier, the algorithms described in this section return a cut c, but not the specific operation nodes that are required to optimally execute the query q. Given a cut c, I need an additional step in order to find the necessary operation nodes. Alg. 2 provides the pseudo-code for finding the operation nodes following execution of the H-CS algorithm. Here, the functions $nodeInclusiveCut(n, q)$ and $nodeExclusiveCut(n, q)$ return the set of operation nodes required to execute the relevant part of the query q at an internal node n based on inclusive or exclusive strategies, respectively and the algorithm follows the minimal cost strategy to identify the operation nodes for the hybrid execution. I explained our marking strategy earlier in this section. Based on the marking of each node in the cut, I call the respective function to get the corresponding inclusive or exclusive operation nodes. Note that if the cut, c, includes the root of the hierarchy, then either reading the nodes as part of the query range, or removing the non-range nodes from the root is the cheapest option. This decision is again made based on whether the root node was labeled as part of the inclusive or exclusive cut. I do not need to recompute the two individual costs to make that decision.

3.3.2 Case 2: Multiple Queries without Memory Constraint

In this previous section, I have shown that the simple case where there is a single query to be executed can be handled in linear time in the size of the hierarchy. In general, however, I may be given a set of range queries and need to identify a cut of the hierarchy to help process this set of queries efficiently. In this subsection, I present an algorithm to find a cut for multiple queries without any memory constraints. I consider the more realistic case with memory constraints in the next subsection.

Assume I am given a query workload Q that contains more than one query (each with its corresponding range). Since I do not have memory constraints, if a bitmap

node in the hierarchy has been read into the memory, it can also be cached to be reused by other queries, without incurring any further read costs.

Remember that in Section 3.3.1 I have discussed how to find a hybrid cut and the corresponding operation nodes given a single query. Let us first assume that I use the algorithms discussed in Section 3.3.1 to find the hybrid costs and the appropriate labeling for each query in the workload, Q, separately. In order to see how important a particular node n is relative to a particular query workload. Let us consider, *Sub-Operation Nodes*, $SN_{n,q}$, which denote the operation nodes required to execute the part of q (in Q) that is under n. Hence, $SN_{n,q}$ will contain nodes that are in $n \cup leafDesc(n)$. In order to decide which nodes to choose in the set $n \cup leafDesc(n)$ given q, I use the same hybrid logic as explained in Algorithm 2.

I associate to each node, n, in the hierarchy a new cost, called *no constraint node cost* ($NCNodeCost(n, Q)$), defined as the cost to perform the query workload such that (a) first the node is read and cached into the memory and (b) the remaining nodes in each query's corresponding $SN_{n,q}$ are read:

$NCNodeCost(n, Q) =$

$$(readCost(B_n)) + \left(\sum_{m \in (\cup_{q \in Q} SN_{n,q})/n} readCost(B_m) \right).$$

Intuitively, this cost tells us how important a particular node, n, is relative to the query workload Q: If there are two nodes, n_a and n_b, such that n_a appears in $SN_{n_a,q}$ for more than one query $q \in Q$ and n_b does not appear in any $SN_{n_b,q}$ for any $q \in Q$, then the $NCNodeCost(n_a, Q)$ will be lower than $NCNodeCost(n_b, Q)$. Consequently, I can say that a node that is included in the $SN_{n,q}$ is more important (caching it would impact more queries) and such important nodes have small $NCNodeCost$ values. I use this as the basis of our algorithm, shown in Alg. 3, to find the relevant hybrid cut given multiple queries. This bottom-up traversing algorithm is similar to the Hybrid

Algorithm 3 Hybrid Cut Multiple Query Algorithm

1: **Input:** Hierarchy H, Set of internal nodes I_H, Query Workload Q
2: **Output:** Set of nodes c
3: **Initialize:** Node n = root, c
4: **procedure** FINDHYBRIDCUT(n)
5: Set $children = findChildren(n, I_H)$;
6: **if** *children* is empty **then**
7: add n to c;
8: return $NCNodeCost(n, Q)$;
9: **else**
10: $costChildren = 0$;
11: **for** each child m of n **do**
12: $costChildren = costChildren + costChild$;
13: **end for**
14: $costCurrNode = NCNodeCost(n, Q)$;
15: **if** $costCurrNode \leq costChildren$ **then**
16: remove all descendants of n from c;
17: add n to c;
18: **end if**
19: **end if**
20: **return** $min(costCurrNode, costChildren)$
21: **end procedure**

Cut Algorithm explained in the previous section. The main difference is that I use the cost $NCNodeCost(n, Q)$ for each node, which is derived using the hybrid logic as explained in the previous section.

3.3.3 Case 3: Multiple Queries with Memory Constraint

In the previous subsection, I introduced a node cost (based on the cost model as described in 3.2.3.) to capture the importance of a node in a multiple query scenario without a memory constraint. In this section, I relax the assumption of unlimited memory availability and consider the more general situation where we have a memory constraint, limiting how many bitmaps I can keep in memory at a time. More specifically, in this section, I present two algorithms, namely *1-Cut Selection Algorithm* and *k-Cut Selection Algorithm*, that find a cut given a query workload and a memory constraint. Note that, as discussed in Section 3.2.3, due to the memory constraint, the resulting cuts may be incomplete.

Let us consider a set of nodes for each query and each n, called *Constraint Operation Nodes*, denoted by $CON_{n,q}$. Here, $CON_{n,q} \subseteq n \cup L_H$. $CON_{n,q}$ chooses the set of nodes from $n \cup L_H$ that are required to execute q in the cheapest possible manner given n and the set of leaf nodes.

$CON_{n,q}$ consists of two sets of nodes. The first set is the set of nodes that includes n and its leaf descendants. I have to decide which nodes to choose in the set $n \cup leafDesc(n)$ given q. In order to make this decision, I use the same hybrid logic as explained in Algorithm 2. The second set of nodes, consists of the set of leaf nodes that are not descendants of n , i.e. $L_H \cap leafDesc(n)$. In order to execute q, all the query range nodes in this set have to be read, and hence we include them in $CON_{n,q}$.

As I have done in Case 2 (without memory constraints), I introduce a node cost to capture the importance of each internal node in the hierarchy relative to query

workload Q. This cost, called *constrained node cost* ($CNodeCost(n, Q)$), reflects the cost of performing the query in such a way that (a) only nodes with low cost, and that can fit into the memory within the given constraint, are read and cached into the memory and (b) the remaining nodes in each query's $CON_{n,q}$ are read from the secondary storage as needed.

$CNodeCost(n, Q) =$

$$(readCost(B_n)) + \left(\sum_{q \in Q} \sum_{m \in CON_{n,q}/n} readCost(B_m) \right)$$

Intuitively, if more queries can reuse a node for further query processing when the node is cached, the lower the *constrained node cost* of the node is relative to the query workload Q.

1-Cut Selection Algorithm

In Alg. 4, I present the pseudo-code of *1-Cut Selection Algorithm*, for Case 3 with multiple queries in the presence of a memory constraint.

Here, $S_{available}$ denotes the amount of memory available for adding nodes to a cut and S_{B_n} denotes the size of the bitmap index of node n on the secondary storage. The first time the algorithm is called, I initialize $S_{available}$ to the memory available for the whole process, i.e., S_{total}; in subsequent calls, the amount is reduced as new bitmaps are added to the cut. Note that

- In line 6, I choose a node that has the lowest node cost and the size of the node is lesser than or equal to the remaining memory availability.
- In line 9, I ensure that the returned cut does not contain any two nodes that are on the same root-to-leaf branch.

Algorithm 4 1-Cut Selection Algorithm

1: **Input:** Hierarchy H, Set of internal nodes I_H, Query Workload Q, $S_{available}$

2: **Output:** Set of nodes c

3: **Initialize:** $S_{available} = S_{total}$

4: **procedure** FINDCUTCONSTRAINT($D_H, S_{available}$)

5: **while** I_H is not empty OR there exists a node n such that $S_{B_n} \leq S_{available}$ **do**

6: choose node n such that n has the lowest $CNodeCost(n, Q)$ among nodes in I_H & $S_{B_n} \leq S_{available}$;

7: add n to c;

8: remove n from I_H;

9: remove ancestors and descendants of n from I_H;

10: update $S_{available} = S_{available} - S_{B_n}$;

11: **end while**

12: **return** c

13: **end procedure**

The stopping condition of the greedy process is reached when the input set of nodes is empty (i.e. a complete cut is found) or when each of the remaining nodes have sizes larger than $S_{available}$. Note that it is possible that in some cases the optimal subset of nodes required to execute the given query workload may all fit in the available memory. Our algorithm adds nodes until all nodes are seen or no nodes can be added further due to memory constraints. In order to avoid adding nodes that are not going to be used in query processing, I introduce a new node label, *unused*, applied while calculating the $CNodeCost(n, Q)$ indicating that the node as *unused* if the node is not used by any query. This is easy to find out if for every q in Q, $P_{n,q}$ does not include n, then the node is an unused node.

It is important to note that the above algorithm does not necessarily return a cut that has the optimal cost. As I see in Section 3.4.3, the sub-optimality of the algorithm is most apparent in situations where I have plenty (yet still insufficient amount of) memory and, consequently, the cost-sensitive greedy algorithm over-prunes the solution space (though it still provides cuts that are significantly more efficient than a naïve execution plan). In situations where the memory constraints are tight, however, the algorithm returns very close to optimal or optimal cuts, proving the effectiveness of the cost model and the proposed approach.

k-Cut Selection Algorithm

In this subsection, I note that the key weakness of the above algorithm is that it considers only a single cut of the hierarchy: When I choose to include a node in the cut, I remove all the ancestors and descendants of the node from further consideration; however, it is possible that a node can have the lowest cost, but two or more of its ancestors or descendants *combined* can lead to a better execution plan. A node n may be chosen before its ancestor m, because cost(n) is lesser than cost(m). But, it is also possible that choosing m could be a better choice than choosing n if m can be used to execute a larger portion of the range nodes of the query.

Therefore, in Alg. 5, I present a variation of the algorithm, called the *k-Cut Selection Algorithm*. In this variation, the algorithm considers k different cuts. When a node, n, is added to a cut, the algorithm does not eliminate its ancestors and descendants from further consideration; instead, it simply does not add these ancestors and descendants to the same cut as n to follow the rules of validity as described in Section 3.2.3. These ancestors and descendants however may be added to the other k-1 cuts.

In Algorithm 5, the i^{th} cut has a corresponding memory requirement, $S_{c_i,available}$.

Algorithm 5 k-Cut Selection Algorithm

1: **Input:** Hierarchy H, Set of internal nodes I_H, Query Workload Q, $S_{available}$, $cutList$

2: **Output:** Set of nodes c

3: **Initialize:** $\forall_{c \in cutList} S_{c_i,available} = S_{total}$.

4: **procedure** FINDKCUTCONSTRAINT(H)

5: **while** each node n in I_H is seen OR there exists a node n such that $S_{B_n} \leq S_{c_i,available}$ for $i \leq k$ **do**

6: choose node n such that n has the lowest $CNodeCost(n, Q)$ among nodes in H;

7: mark n as seen;

8: **for** each cut c in $cutList$ **do**

9: **if** $S_{B_n} \leq S_{c_i,available}$ **then**

10: **if** there is no conflict in c for node n **then**

11: **if** n has not been added to any empty cut **then**

12: add n to c;

13: update $S_{c_i,available} = S_{c_i,available} - S_{B_n}$;

14: **end if**

15: **else**

16: copy each node in c to the next available empty cut;

17: replace the conflicting node with node n;

18: **end if**

19: **end if**

20: **end for**

21: Sort the *cutList* based on the lowest cost for each cut;

22: **end while**

23: **return** the cut c in cutList that has the lowest total cost;

24: **end procedure**

- In the algorithm, line 11 ensures that a node is not added more than once to an empty cut. This prevents two cuts containing identical nodes.

- Lines 16 and 17 are part of the replacement procedure. According to Section 3.2.3, a cut cannot have two nodes on the same root-to-leaf branch. Hence, n cannot be added to the existing cut if there is such a *conflict*. In these lines, when I detect a conflict, I add the nodes of a cut to an empty cut and replace the conflicting node with the current node. This lets us construct multiple conflicting cuts that are individually conflict-free.

  Note that if after replacing the conflicting node with the current node, the size of the cut exceeds the size of available memory, then this node and the corresponding conflicting cut is ignored.

- In Line 21, I sort the *cutList* in ascending order based on the overall cost of each discovered cut. I do this in order to give more preference to the cuts with a lower cost during the next iteration.

Auto Selection of k

As I see in the next section, in practice it is sufficient to consider fairly small number of cuts to significantly improve the effectiveness of the proposed greedy algorithm (returning very close to optimal cuts), without increasing the cost of the optimization step significantly. However, in cases where it is difficult for the user to set the value

of k ahead of the time, I propose a δ *auto-stop* condition: after finding the i'th cut, I evaluate if $cost_{i-1} - cost_i < \delta$, for a user provided per-iteration cost gain value, δ. The algorithm auto-stops when the condition is satisfied (i.e., when the cost gain of the iteration drops below the predetermined gain). In Section 3.4.3, the auto-stop condition is effective, even when I simply set $\delta = 0$; i.e., I stop when the cost of the new cut has the same cost as the previous cut (note that, for any two integers $l, m > 1$, and $l > m$, the cost of l-greedy cut will always be equal to or lesser than the cost of m-greedy cut; this is because whatever cut that is returned by the m-greedy cut algorithm will always be enumerated and considered by the l-greedy cut algorithm).

3.4 Evaluation

In order to evaluate the cut-selection algorithms presented in this work, I considered two datasets: (a) a synthetically generated dataset (with normal value distribution) and (b) the TPC-H dataset [28], each with 150 million records. In particular, in the TPC-H dataset, I focused on the *account balance* attribute whose values demonstrate a near-uniform distribution, with spikes in the occurrences for some values.

In this section, I have two main evaluation criteria: (1) query execution IO cost and (2) optimization time. I compared the results of our cut-selection algorithms against (a) leaf-only query execution, (b) random cut-selection, and (c) exhaustive cut-search strategies.

For both of the above data sets, I considered (balanced) attribute hierarchies of different depth and internal-node fanout: these were generated for different numbers of leaf nodes and maximum possible fanouts of the internal nodes of the hierarchy. Since finding the optimal cut using an exhaustive strategy *for comparison purposes* is prohibitively expensive, I initially considered small hierarchies, with 20, 50, and 100 leaf nodes and heights of 4, 5, and 4 respectively (the root of the hierarchy being

considered at height 1).

In Section 3.4.4, I consider hierarchies of larger sizes and higher number of queries to study the scalability of the cut-selection algorithms against the hierarchy size.

Bitmap indices were generated for the nodes of these hierarchies using the Java library, WAH bitset [1] as explained in [93]. The parameters of the read cost model presented in Section 3.2.2, and shown in Figure 3.1 were computed based on these bitmap indices.

I have also created query workloads with different target range sizes. For example, for a hierarchy of 100 leaf nodes, 10% query range size indicates that each range query covers 10 consecutive leaf nodes.

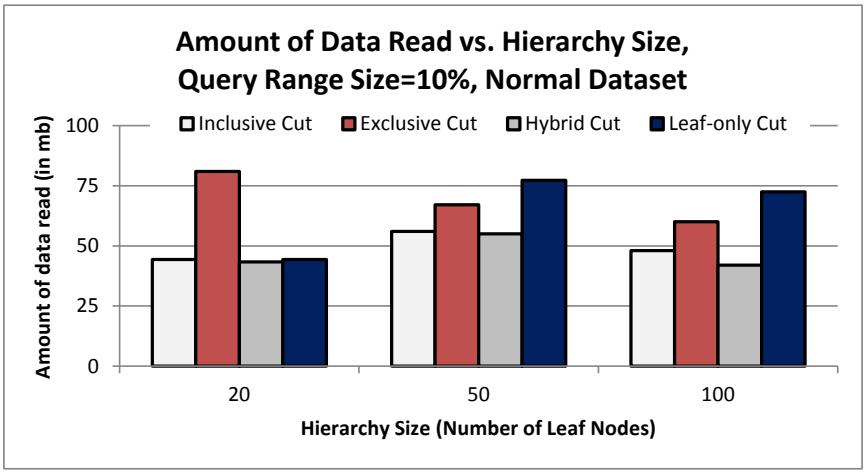
I ran the experiments on a quad-core Intel®Core™i5-2400 CPU @ 3.10GHz machine with 8.00GB RAM. All codes are implemented and run using Java v1.7.

3.4.1 Case 1: Single Query without Memory Constraints

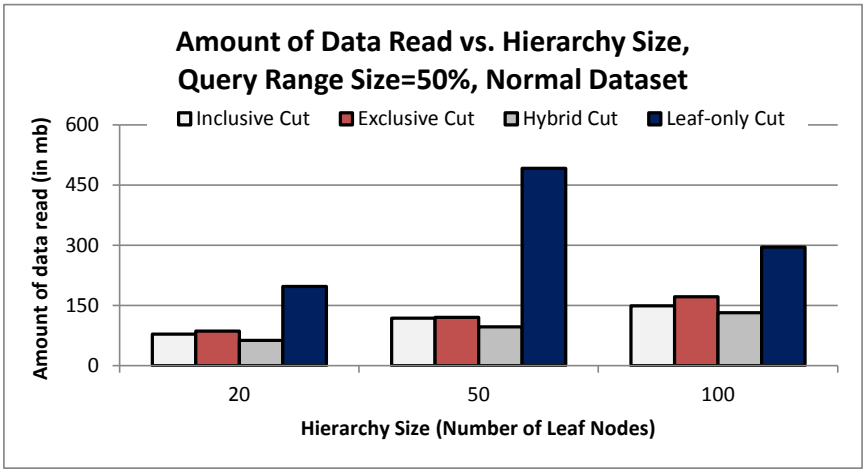
I first evaluate the cut-selection algorithm for the single query without memory constraints scenario. All reported costs are averages of the costs for 10 different runs.

Figures 3.2(a) through (f) compares the three different cut-selection algorithms (I-CS, E-CS, and H-CS) presented in Section 3.3.1 for different data sets and varying hierarchy and range query sizes. As I see in these charts, the inclusive strategy is efficient when the query ranges are small; this is consistent with the observation in [24]. The exclusive strategy, however, is more efficient than the inclusive strategy when the query ranges are larger. Most importantly, in all cases, the hybrid strategy (H-CS) returns the best cuts.

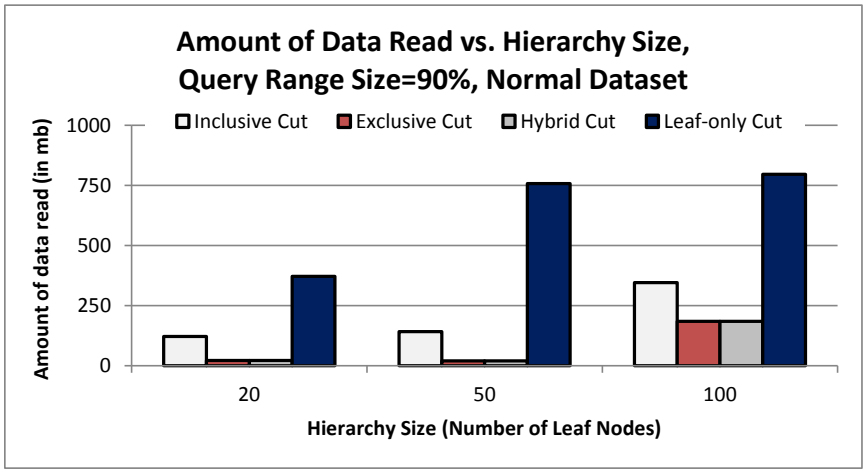
In Figure 3.3, I compare the hybrid (H-CS) strategy against (exhaustively found) optimal and average cuts. The figure also shows the performance of the worst cut. As expected, the H-CS strategy returns optimal cuts. On the average, randomly



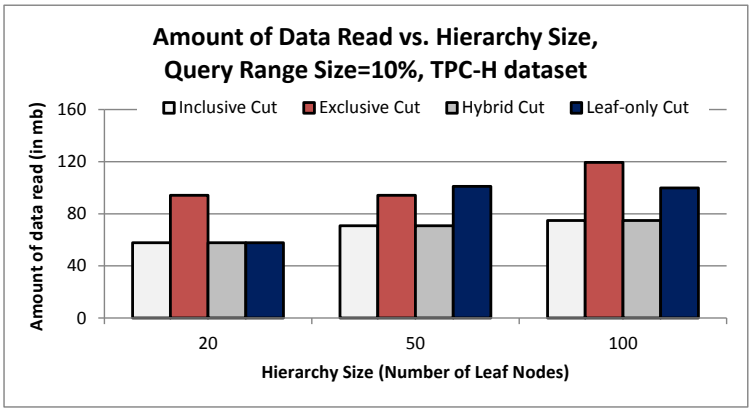
(a) synthetic data, 10% query range



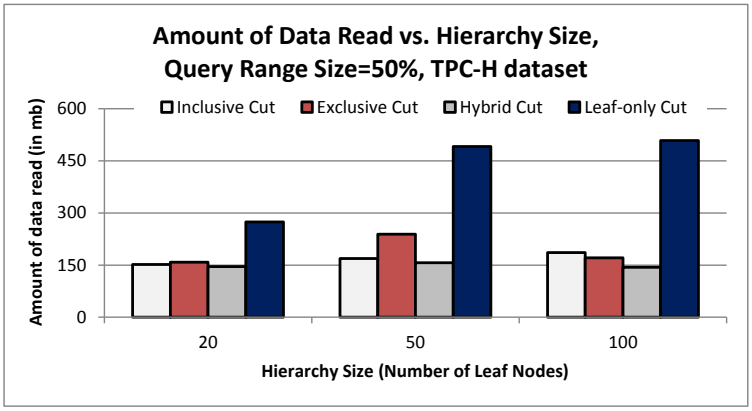
(b) synthetic data, 50% query range



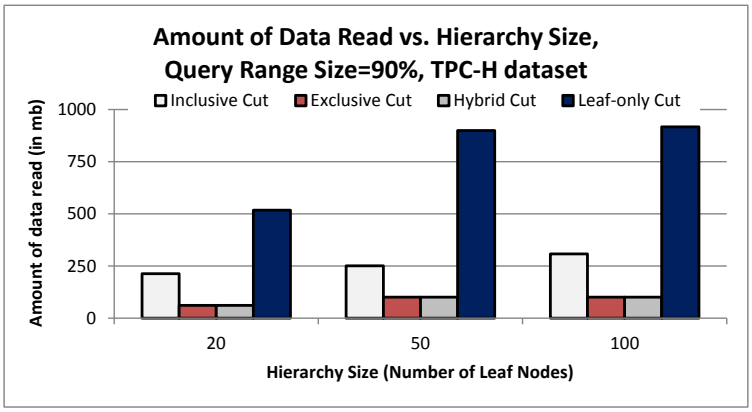
(c) synthetic data, 90% query range



(d) TPC-H data, 10% query range



(e) TPC-H data, 50% query range



(f) TPC-H data, 90% query range

Figure 3.2: Case 1, Single Query without Memory Constraints: Effects of Varying Hierarchy and Range Sizes on the Amount of Data Read by the Three Different Cut-Selection Algorithms

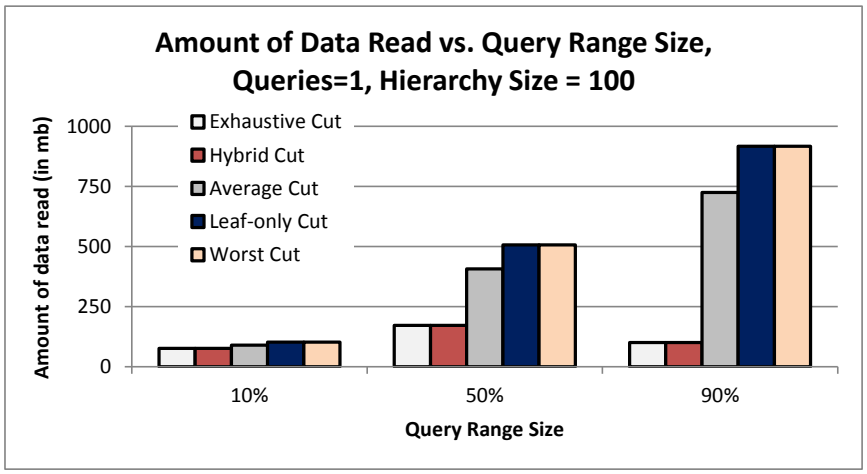


Figure 3.3: Case 1, Single Query without Memory Constraints: Comparing the Proposed Cut Algorithm to (Exhaustively Found) Optimal, Average, and Worst Cuts (TPC-H Data)

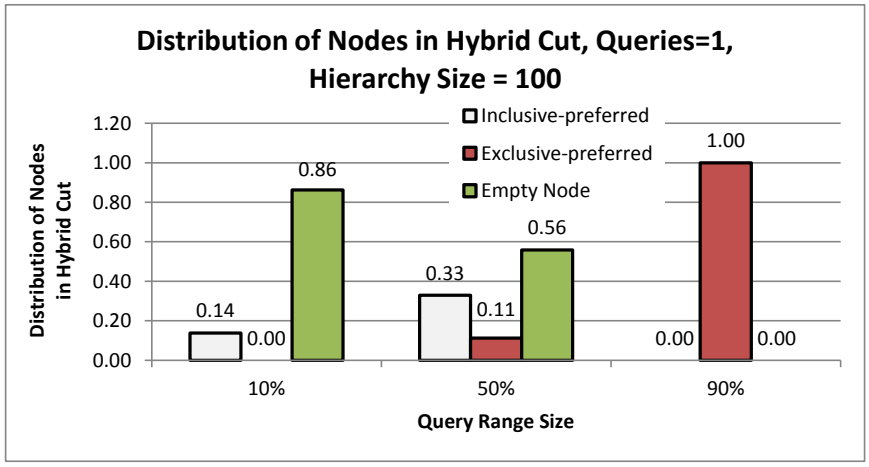


Figure 3.4: Case 1, Single Query without Memory Constraints: Percentages of Nodes in Each Strategy (TPC-H Data)

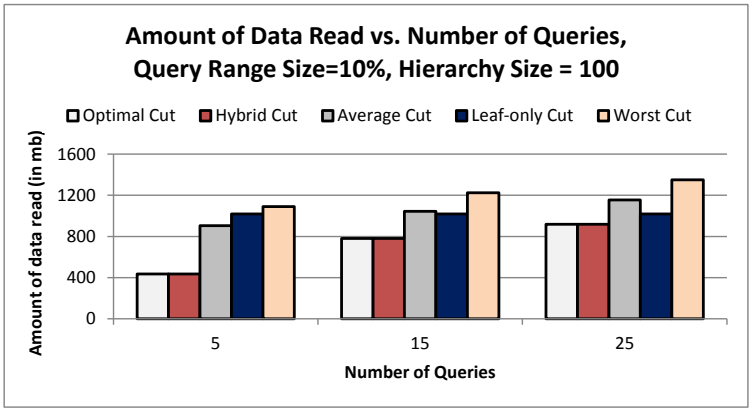
selecting a cut performs quite poorly (almost as bad as selecting the worst possible cut), especially as the query range sizes increase. This highlights the importance of utilizing an effective (hybrid) cut-selection algorithm for answering queries.

In Figure 3.4, I show the percentages of nodes that are labeled inclusive-preferred or exclusive-preferred in a hybrid cut, as explained in Section 3.3.1, for different query ranges. As defined in Section 3.3.1, empty nodes are nodes that are not used in query processing. When the query range size is small, most of the query processing can be done using the leaf nodes. Hence, I see in the figure that most of the nodes in the cut are empty nodes. As expected, when the query range is small, the inclusive strategy dominates and when the range is large, the exclusive strategy dominates. For ranges that are neither small nor large, the hybrid algorithm leverages a mix of inclusive and exclusive strategies.

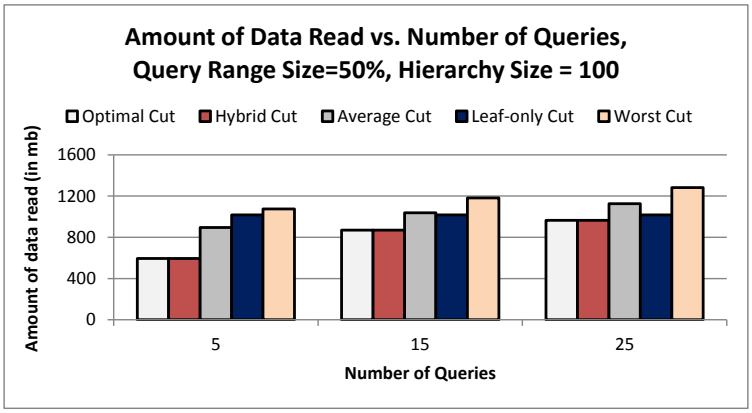
3.4.2 Case 2: Multiple Queries without Memory Constraints

In this section, I evaluate the hybrid cut selection algorithm (Alg. 3) for query workloads with multiple queries. For our evaluations, we considered query workloads of different sizes (and with different ranges). All reported costs are averages of the costs for 10 different runs.

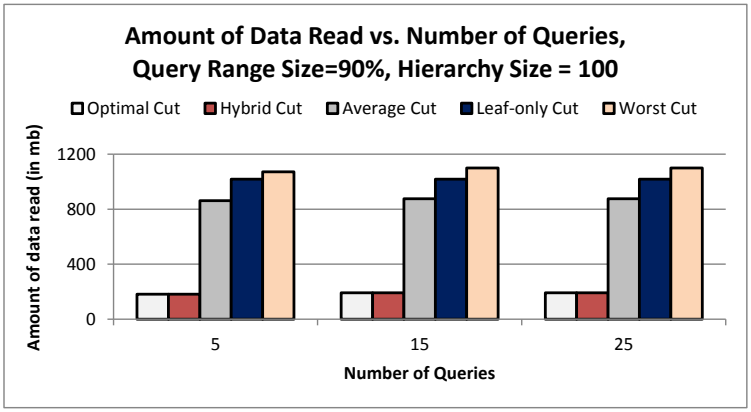
Figure 3.5 shows the impact of using the proposed hybrid cut selection algorithm for different numbers of queries. As I see in this figure, as expected, the hybrid cut selection algorithm returns the optimal cut. The impact of the proposed cut selection algorithm is especially strong when the query includes large ranges as when there are large overlaps among the queries, the query evaluation algorithm has more opportunities for reusing cached nodes, and the proposed hybrid cut strategy is able to leverage these opportunities most effectively.



(a) 10% query range



(b) 50% query range



(c) 90% query range

Figure 3.5: Case 2, Multiple Queries without Memory Constraints (TPC-H Data)

3.4.3 Case 3: Multiple Queries under Memory Constraints

In this section, I evaluate the effectiveness of the proposed k-hybrid cut algorithm (Alg. 5, described in Section 3.3.3), for multiple queries, but under memory constraints. I report the memory availability in terms of the percentage of the memory needed to store the bitmap indices corresponding to the maximum cut of the given hierarchy. The presented results are averages of 10 different runs.

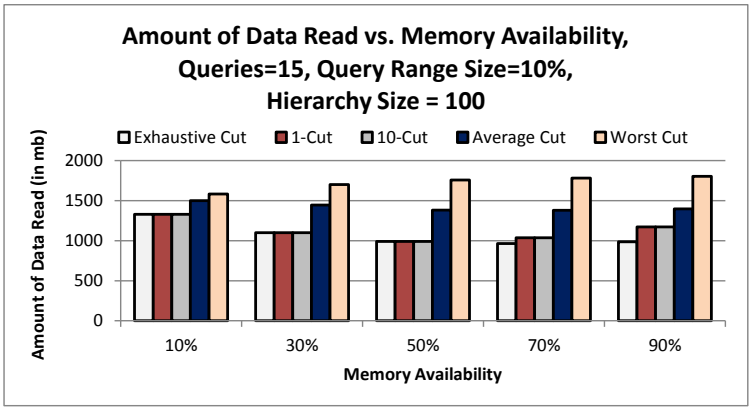
Once again, I compare the proposed cut selection algorithm against solutions found through exhaustive enumeration, average solutions representing randomly selected cuts, and also the worst solution. Remember, that under memory limitations, I need to consider also the incomplete cuts of the input hierarchies.

Note that the number of incomplete cuts that an exhaustive algorithm would need to consider grows very fast:

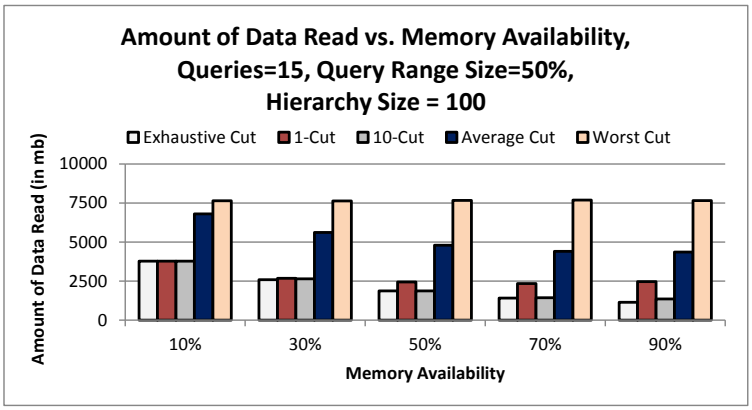
Num. of leaves	Height	Incomplete cuts
20	4	154
50	5	296,381
100	4	1,185,922

However, since the number of incomplete cuts grow even faster than the number of complete cuts, enumerating all incomplete cuts for the exhaustive algorithm (which I use to locate the optimal cut for comparison purposes), becomes prohibitive beyond hierarchies with 100 leaf nodes.

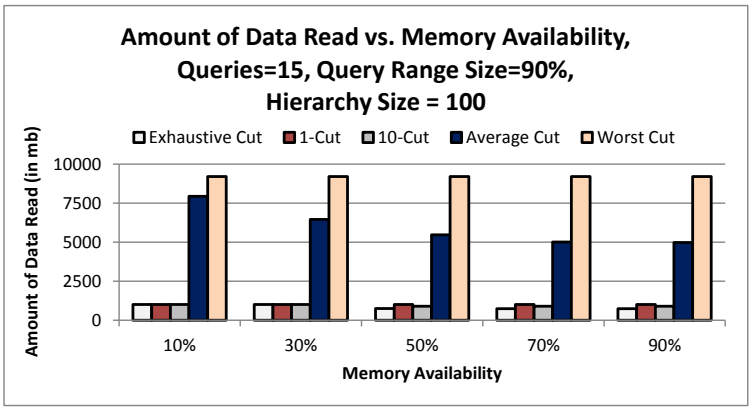
Figure 3.6 shows that, in this case, the proposed hybrid cut selection algorithms are not optimal; however, they return cuts that are very close to optimal. In fact, especially when the memory availability is very restricted (which is the expected situation in most realistic deployments), even the 1-Cut algorithm is able to return optimal or very close to optimal answers. As the available memory increases, the



(a) 10% query range



(b) 50% query range



(c) 90% query range

Figure 3.6: Case 3, Multiple Queries with Varying Memory Availability (TPC-H Data

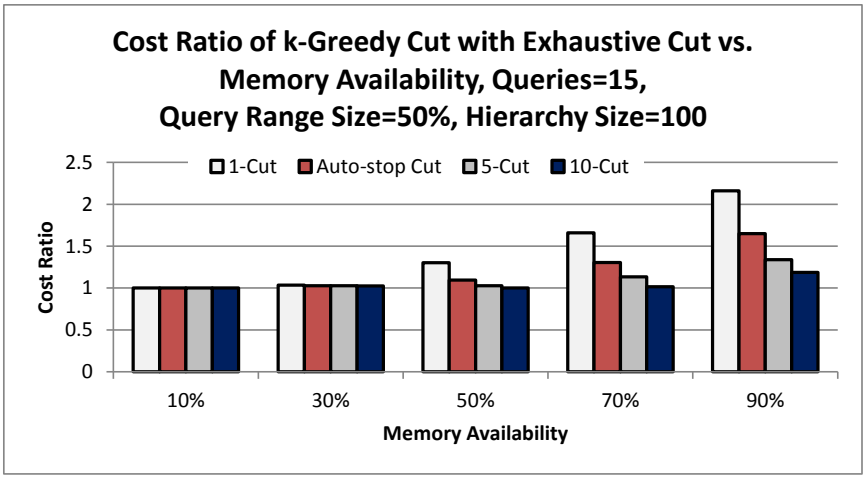


Figure 3.7: Case 3, Multiple Queries with Varying Memory Availability (TPC-H Data): Impact of Different k

optimal cost decreases as there are more caching opportunities, but 1-Cut strategy may not be able to leverage this effectively, especially for larger query ranges. However, I see that the multi-cut strategy (10-Cut in this figure) performs quite close to optimal. Figure 3.7, which plots the ratio of the cost of the solutions found by the multi-cut strategy (for different values of k) to the cost of the optimal cut found through an exhaustive search, confirms this observation: note the figure also shows that the *auto-stop* strategy described in Section 3.3.3 is effective in reducing the cost, without having to fix the value k ahead of time.

Figures 3.8 through 3.10 further confirm that the proposed multi-cut strategy is robust against changes in the size of the query ranges, number of queries, and hierarchy sizes.

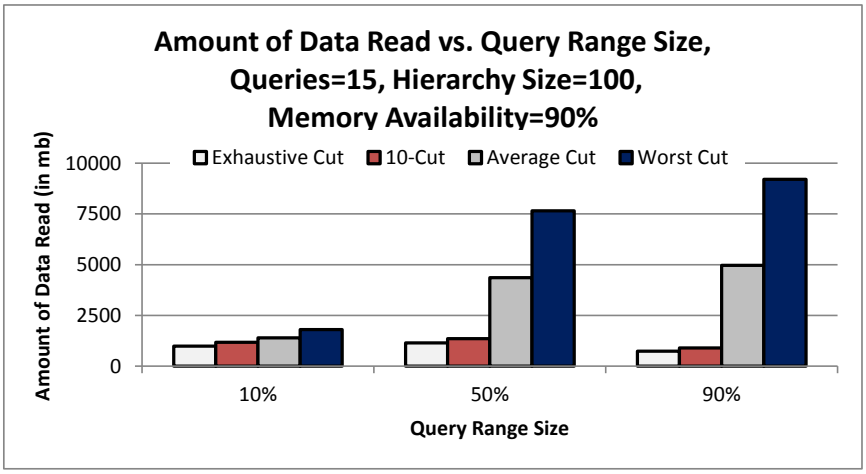


Figure 3.8: Case 3, Effect of Different Query Range Sizes (TPC-H Data, 90% Memory Availability)

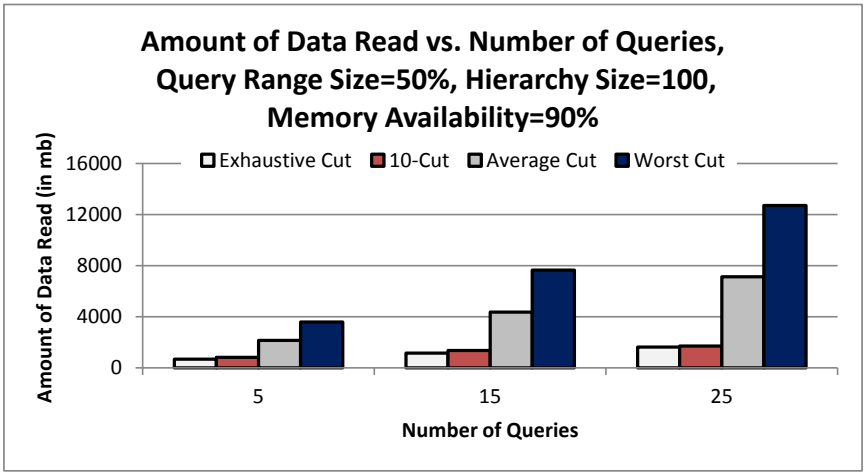


Figure 3.9: Case 3, Effect of Different Number of Queries (TPC-H Data, 90% Memory Availability)

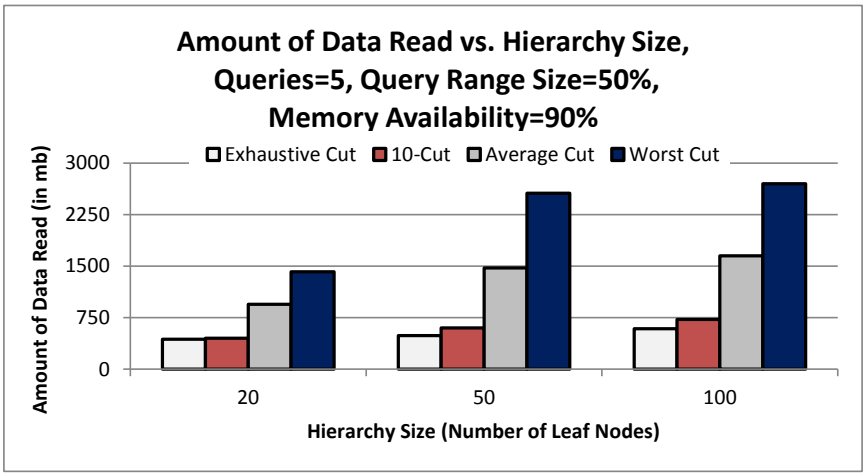


Figure 3.10: Case 3, Effect of Different Hierarchy Sizes (TPC-H Data, 90% Memory Availability)

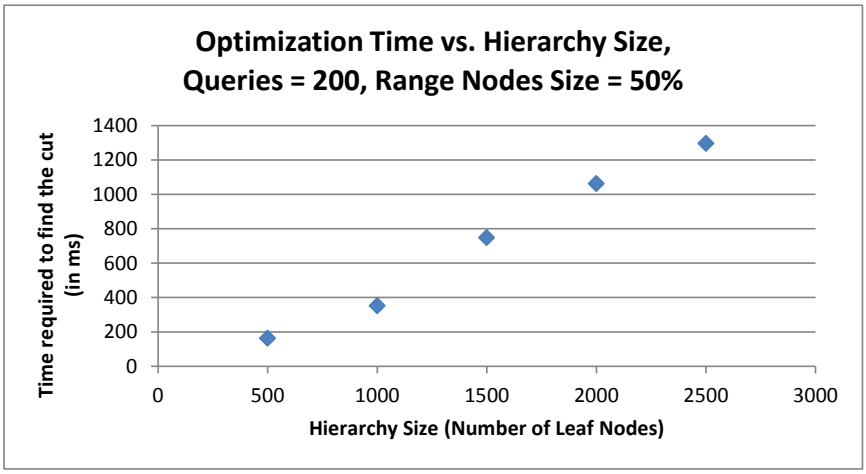


Figure 3.11: Effect of Different Hierarchy Sizes on Time Taken to Find the Hybrid Cut

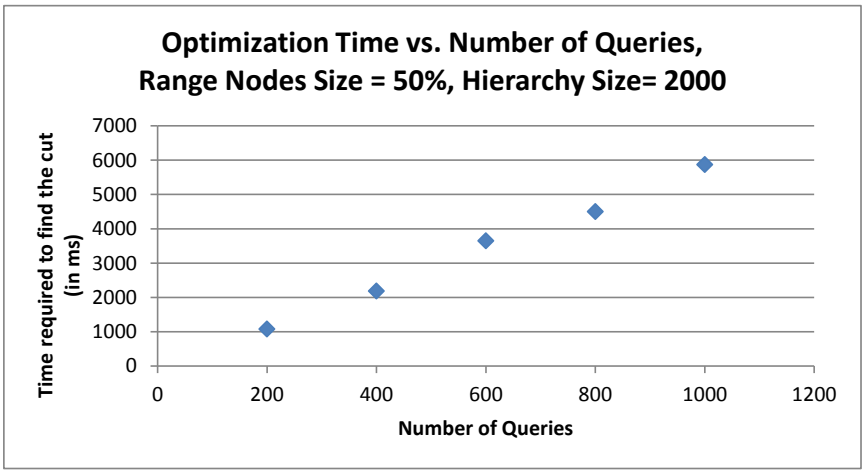


Figure 3.12: Effect of Different Number of Queries on Time Taken to Find the Hybrid Cut

3.4.4 Cut-Selection Time

Up to now, I considered query processing cost using cuts. I now focus on the time needed to select cuts for hierarchies of different sizes. In Figures 3.11 and 3.12, I see the cut selection time as a function of the size of the hierarchy (number of leaf nodes; i.e., the size of the domain) and the number of queries, respectively. Please note that, in these figures, I do not compare our algorithm with exhaustively found cuts, and hence are able to consider larger hierarchy sizes and higher number of queries. The figures confirm that the time taken to find the cut increases linearly with size of the attribute domain and the number of queries.

Chapter 4

EXECUTION OF RANGE QUERY WORKLOADS IN 2D SPACES

4.1 Introduction

Spatial and mobile applications are gaining in popularity, thanks to the widespread use of mobile devices, coupled with increasing availability of very detailed spatial data (such as Google Maps and OpenStreetMap [4]), and location-aware services (such as FourSquare and Yelp). For implementing range queries (Section 4.2.1), many of these applications and services rely on spatial database management systems, which represent objects in the database in terms of their coordinates in 2D space. Queries in this 2D space are then processed using multidimensional/spatial index structures that help quick access to the data [78].

4.1.1 Spatial Data Structures

The key principle behind most indexing mechanisms is to ensure that data objects closer to each other in the data space are also closer to each other on the storage medium. In the case of 1D data, this task is relatively easy as the total order implicit in the 1D space helps sorting the objects so that they can be stored in a way that satisfies the above principle. When the space in which the objects are embedded has more than one dimension, however, the data has multiple degrees of freedom and, as a consequence, there are many different ways in which the data can be ordered on the storage medium and this complicates the design of search data structures. One common approach to developing index structures for multi-dimensional data is to partition the space hierarchically in such a way that (a) nearby points fall into

the same partition and (b) point pairs that are far from each other fall into different partitions. The resulting hierarchy of partitions then can either be organized in the form of trees (such as quadtrees, KD-trees, R-trees and their many variants [78]) or, alternatively, the root-to-leaf partition paths can be serialized in the form of strings and these strings can be stored in a string-specific search structure. Apache Lucene, a highly-popular search engine, for example, leverages such serializations of quadtree partitions to store spatial data in a *spatial prefix tree* [61].

An alternative to applying the partitioning process in the given multi-dimensional space is to map the coordinates of the data into a 1D space and perform indexing and query processing on this 1D space instead. Intuitively, in this alternative, one seeks an embedding from the 2D space to a 1D space such that (a) data objects closer to each other in the original space are also closer to each other on the 1D space, and (b) data objects further away from each other in the original space are also further away from each other on the 1D space. This embedding is often achieved through fractal-based *space-filling curves* [20, 40]. In particular, the Peano-Hilbert curve [40] and Z-order curve [67] have been shown to be very effective in helping cluster nearby objects in the space. Consequently, if data are stored in an order implied by the space-filling curve, then the data elements that are nearby in the data space are also clustered, thus enabling efficient retrieval. In this work, I leverage these properties of space-filling curves to develop a highly compressible bitmap-based index structure for spatial data.

4.1.2 Bitmap-based Indexing

Bitmap indexes [83, 93] have been shown to be highly effective in answering queries in data warehouses [97] and column-oriented data stores [8]. There are two chief reasons for this: (a) first of all, bitmap indexes provide an efficient way to evaluate

logical conditions on large data sets thanks to efficient implementations of the bitwise logical "AND", "OR", and "NOT" operations; (b) secondly, especially when data satisfying a particular predicate are clustered, bitmap indexes provide significant opportunities for compression, enabling either reduced I/O or, even, complete in-memory maintenance of large index structures. In addition, (c) existence of compression algorithms [31, 94] that support compressed domain implementations of the bitwise logical operations enables query processors to operate directly on compressed bitmaps without having to decompress them until the query processing is over and the results are to be fetched from the disk to be presented to the user.

4.1.3 Contributions of this Work

In this work, I show that bitmap-based indexing is also an effective solution for managing spatial data sets. More specifically, I first propose *compressed spatial hierarchical bitmap (cSHB)* indexes to support spatial range queries. In particular, I (a) convert the given 2D space into a 1D space using Z-order traversal, (b) create a hierarchical representation of the resulting 2D space, where each node of the hierarchy corresponds to a (sub-)quadrant (i.e., effectively creating an implicit "quadtree"), and (c) associate a *bitmap* file to each node in the quadtree representing the data elements that fall in the corresponding partition. I present efficient algorithms for answering range queries using a select subset of bitmap files stored in a given cSHB index.

I then consider a service provider that has to answer multiple concurrent queries over the same spatial data and, thus, focus on query workloads involving multiple range queries. Since the same set of queries can be answered using different subsets of the bitmaps in the cSHB index structure, I consider the problem of identifying the appropriate bitmap nodes for processing the given query workload. More specifically, as I visualize in Figure 4.1, (a) I develop cost models for range query processing over

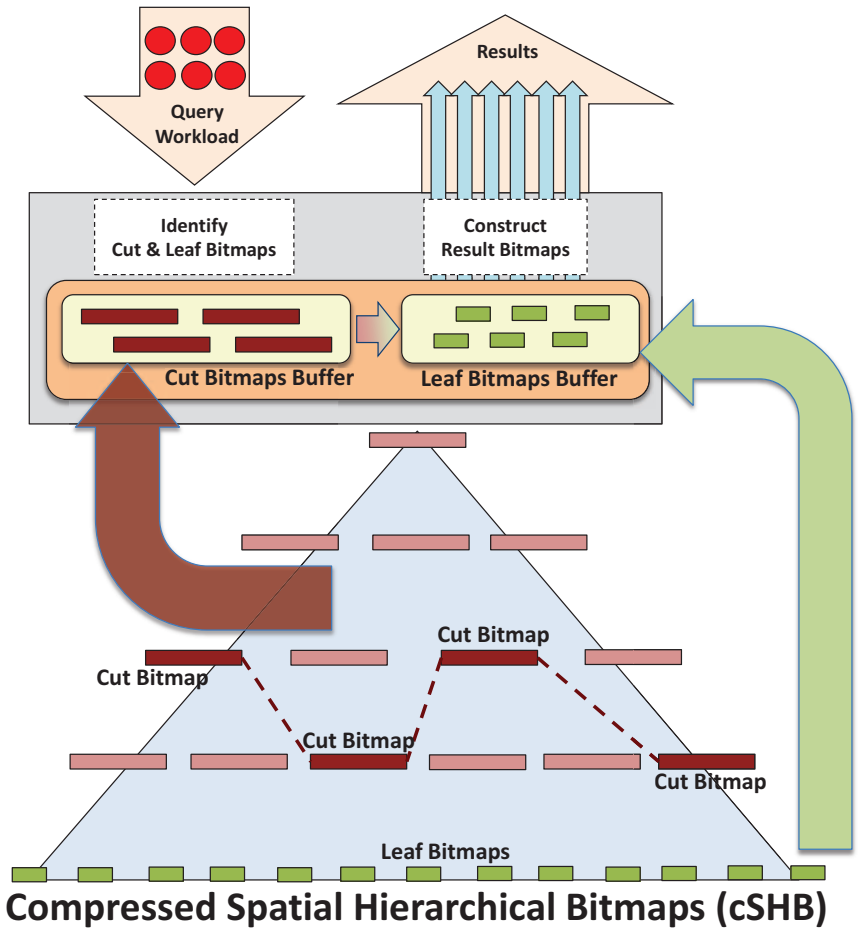


Figure 4.1: Processing a Range Query Workload using *compressed Spatial Hierarchical Bitmap (cSHB)*

compressed spatial hierarchical bitmap files and (b) propose efficient *bitmap selection* algorithms that select the best bitmap nodes from the cSHB index structure to be fetched into the main-memory for processing of the query workload. In this chapter, I also present an efficient disk-based organization of compressed bitmaps. To my best knowledge, this is the first work that provides an efficient index structure to execute a query workload involving multiple spatial range queries by using bitmap indexes. Experimental evaluations of the cSHB index structure and the bitmap selection algorithms show that cSHB is highly efficient in answering a given query workload.

4.2 Compressed Spatial Hierarchical Bitmap (cSHB) Indexes

In this section, I present the key concepts used in this work and introduce the *compressed spatial hierarchical bitmap (cSHB)* index structure for answering spatial range queries.

4.2.1 Key Concepts and Notations

Spatial Database

A multidimensional database, $\mathcal{D}$, consists of points that belong to a (bounded and of finite-granularity) multidimensional space $\mathcal{S}$ with d dimensions. A spatial database is a special case where $d = 2$. I consider rectangular spaces such that the boundaries of $\mathcal{S}$ can be described using a pair of south-west and a north-east corner points, c_{sw} and c_{ne} ($c_{sw}.x \leq c_{ne}.x$ and $c_{sw}.y \leq c_{ne}.y$ and $\forall_{p \in \mathcal{S}}\ c_{sw}.x \leq p.x \leq c_{ne}.x$ and $c_{sw}.y \leq p.y \leq c_{ne}.y$).

Spatial Query Workload

In this chapter, I consider query workloads, Q, consisting of a set of *rectangular* spatial range queries.

- **Spatial Range Query:** A range query, $q \in Q$, is defined by a corresponding range specification $q.rs = \langle q_{sw}, q_{ne} \rangle$, consisting of a south-west point and a north-east point, such that $q_{sw}.x \leq q_{ne}.x$ and $q_{sw}.y \leq q_{ne}.y$.

  Given a range query, q, with a range specification, $q.rs = \langle q_{sw}, q_{ne} \rangle$, a data point $p \in \mathcal{D}$ is said to be contained within the query range (or is a *range point*) if and only if $q_{sw}.x \leq p.x \leq q_{ne}.x$ and $q_{sw}.y \leq p.y \leq q_{ne}.y$.

Spatial Hierarchy

In cSHB, we associate to the space $\mathcal{S}$ a hierarchy H, which consists of the node set $\mathcal{N}(H) = \{n_1, \ldots, n_{maxn}\}$:

- **Nodes of the hierarchy:** Intuitively, each node, $n_i \in \mathcal{N}(H)$ corresponds to a (bounded) subspace, $S_i \subseteq S$, described by a pair of corner points, $c_{i,sw}$ and $c_{i,nw}$.

- **Leaves of the hierarchy:** L_H denotes the set of leaf nodes of the hierarchy H and correspond to all potential point positions of the finite space S. *Assuming that the database, $\mathcal{D}$, contains only points, only the leaves of the spatial hierarchy occur in the database.*

- **Parent of a node:** For all n_i, $parent(n_i)$ denotes the parent of n_i in the corresponding hierarchy; if n_i is the root, then $parent(n_i) = \perp$.

- **Children of a node:** For all n_i, $children(n_i)$ denotes the children of n_i in the corresponding hierarchy; if $n_i \in L_H$, then $children(n_i) = \emptyset$. In this work, I assume that the children induce a partition of the region corresponding to the parent node:

$$\left(\underset{n_h \neq n_j \in children(n_i)}{\forall} S_h \cap S_j = \emptyset \right) \text{and} \left(S_i = \bigcup_{n_h \in children(n_i)} S_h \right).$$

- **Descendants of a Node:** The set of descendants of node n_i in the corresponding hierarchy is denoted as $desc(n_i)$. Naturally, if $n_i \in L_H$, then $desc(n_i) = \emptyset$.

- **Internal Nodes:** Any node in H that is not a leaf node is called an internal node. The set of internal nodes of H is denoted by I_H. Each internal node in the hierarchy corresponds to a (non-point) sub-region of the given space. If

$\mathcal{N}(H, l)$ denotes the subset of the nodes at level l of the hierarchy H, then I have

$$\left(\forall_{n_i \neq n_j \in \mathcal{N}(H,l)}\ S_i \cap S_j = \emptyset\right) \text{ and } \left(\mathcal{S} = \bigcup_{n_i \in \mathcal{N}(H,l)} S_i\right).$$

The root node corresponds to the entire space, $\mathcal{S}$.

- **Leaf Descendants of a Node:** Leaf descendants, $leafDesc(n_i)$, of a node are the set of nodes such that

$$leafDesc(n_i) = desc(n_i) \cap L_H.$$

4.2.2 Compressed Spatial Hierarchical Bitmap (cSHB) Index Structure

In this section, I introduce the proposed *compressed spatial hierarchical bitmap (cSHB)* index structure:

Definition 4.2.1 (cSHB Index Stucture) *Given a spatial database $\mathcal{D}$ consisting of a space, $\mathcal{S}$, and a spatial hierarchy, H, a cSHB index is a set, $\mathcal{B}$ of bitmaps, such that for each $n_i \in \mathcal{N}(H)$, there is a corresponding bitmap, $B_i \in \mathcal{B}$, where the following holds:*

- *if n_i is an internal node (i.e., $n_i \in I_H$), then $\left(\exists_{o \in \mathcal{D}} \exists_{n_h \in leafDesc(n_i)}\ located\_at(o, n_h)\right) \leftrightarrow (B_i[o] = 1)$, whereas*
- *if n_i is a leaf node (i.e., $n_i \in L_H$), then $\left(\exists_{o \in \mathcal{D}}\ located\_at(o, n_i)\right) \leftrightarrow (B_i[o] = 1)$*

Our Implementation of cSHB

A cSHB index structure can be created based on any hierarchy satisfying the requirements[1] specified in Section 4.2.1.

[1] In fact, cSHB can be created even when some of the requirements are relaxed – for example children do not need to cover the parent range entirely (as in R-trees).

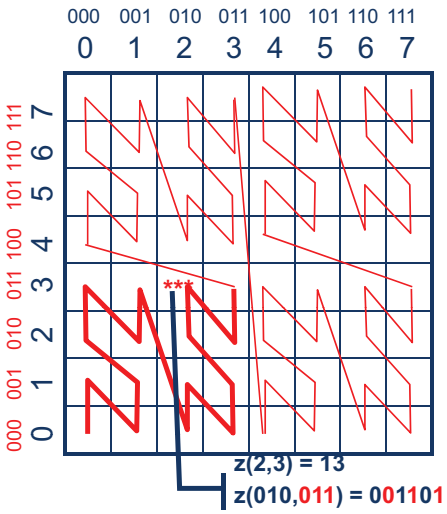


Figure 4.2: Z-order Curve for a Sample 2D Space.

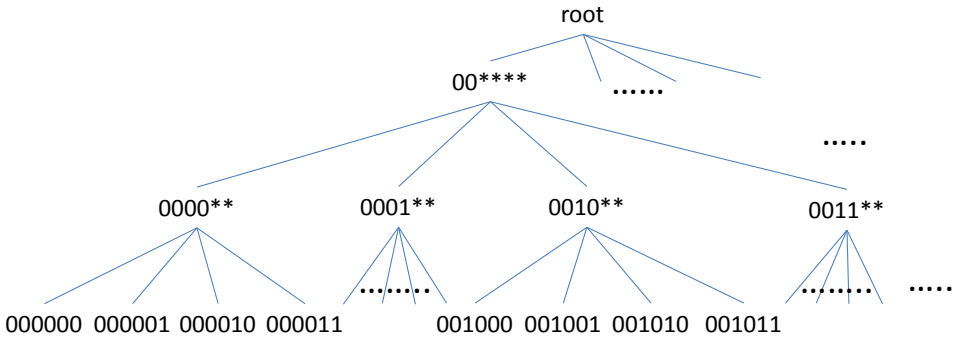


Figure 4.3: A Sample 4-level Hierarchy Defined on the Z-order Space Defined in Figure 4.2 (The String Associated to Each Node Corresponds to its Unique Label)

In this work, *without loss of generality*, I discuss a Z-curve based construction scheme for cSHB. The resulting hierarchy is analogous to the MX-quadtree data structure, where all the leaves are at the same level and a given region is always partitioned to its quadrants at the center [78]. As introduced in Sections 4.1.1 and 2.3.3, a space-filling curve is a fractal that maps a given finite multidimensional data space onto a 1D curve, while preserving the locality of the multidimensional data points (Figure 4.2): in other words nearby points in the data space tend to be mapped to

nearby points on the 1D curve. As I also discussed earlier, Z-curve is a fractal commonly used as a space-filling curve (thanks to its effectiveness in clustering the points in the data space and the efficiency with which the mapping can be computed).

A key advantage of the Z-order curve (for my work) is that, due to the iterative (and self-similar) nature of the underlying fractal, the Z-curve can also be used to impose a hierarchy on the space. As visualized in Figure 4.3, each internal node, n_i, in the resulting hierarchy has four children corresponding to the four quadrants of the space, S_i. Consequently, given a 2^h-by-2^h space, this leads to an $(h+1)$-level hierarchy, (analogous to an MX-quadtree [78]) which can be used to construct a cSHB index structure[2]. As I show in Section 4.4, this leads to highly compressible bitmaps and efficient execution plans.

Blocked Organization of Compressed Bitmaps

Given a spatial database, $\mathcal{D}$, with a corresponding hierarchy, H, I create and store a *compressed bitmap* for each node in the hierarchy, except for those that correspond to regions that are *empty*. These bitmaps are created in a bottom-up manner, starting from the leaves (which encode for each point in space, $\mathcal{S}$, which data objects in $\mathcal{D}$ are located at that point) and merging bitmaps of children nodes into the bitmaps of their parents. Each resulting bitmap is stored as a *compressed* file on disk.

It is important to note that, while compression provides significant savings in storage and execution time, a naive storage of compressed bitmaps can still be detrimental for performance: in particular, in a data set with large number of objects located at unique points, there is a possibility that a very large number of leaf bitmaps need to be created on the secondary storage. Thus, creating a separate bitmap file for

[2]Without loss of generality, I assume that the width and height are 2^h units for some integer $h \geq 1$.

Algorithm 6 Writing blocks of compressed bitmaps to disk

1: **Input:**

- A spatial database, $\mathcal{D}$, defined over 2^h-by-2^h size space, $\mathcal{S}$ and a corresponding $(h+1)$-level (Z-curve based) hierarchy, H, with set of internal nodes, I_H
- Minimum block size, K

2: **procedure** WRITEBITMAPS

3: Block $T = \emptyset$

4: $availableSize = K$

5: **for** level $l = (h+1)$ (i.e., leaves) to 0 (i.e., root) **do**

6: **for** each node n_i in l in increasing Z-order **do**

7: **if** $l == (h+1)$ **then**

8: Initialize a compressed bitmap B_i

9: **else**

10: $B_i = \underset{n_j \in children(n_i)}{\texttt{OR}} B_j$

11: **end if**

12: **if** $size(B_i) \geq K$ **then**

13: write B_i to disk;

14: **else**

15: $T = append(T, B_i)$

16: $availableSize = availableSize - size(B_i)$

17: **if** ($availableSize \leq 0$) or (n_i is the last node at this level) **then**

18: write T to disk;

19: Block $T = \emptyset$

20: $availableSize = K$

21: **end if**

22: **end if**

23: **end for**

24: **end for**

25: **end procedure**

each node may lead to inefficiencies in indexing as well as during query processing (as directory and file management overhead of these bitmaps may be non-negligible).

To overcome this problem, cSHB takes a *target block size*, K, as input and ensures that all index-files written to the disk (with the possible exception of the last bitmap file in each level) are at least K bytes. This is achieved by concatenating, if needed, compressed bitmap files (corresponding to nodes at the same level of hierarchy). In Algorithm 6, I provide an overview of this block-based bottom-up cSHB index creation process. In Line 10, I see that the bitmap of an internal node is created by performing a bitwise OR operation between the bitmaps of the children of the node. These OR operations are implemented in the compressed bitmap domain enabling fast creation of the bitmap hierarchy. As it creates compressed bitmaps, the algorithm packs them into a block (Line 15). When the size of the block exceeds K, the bitmaps in the block are written to the disk (Line 18) as a single file and the block is re-initialized.

Example 4.2.1 *Let us assume that* $K = 10$ *and also that I am considering the following sequence of nodes with the associated (compressed) bitmap sizes:*

$$\langle n_1, 3\rangle; \langle n_2, 4\rangle; \langle n_3, 2\rangle; \langle n_4, 15\rangle; \langle n_5, 3\rangle; \ldots$$

This node sequence will lead to following sequence of bitmap files materialized on disk:

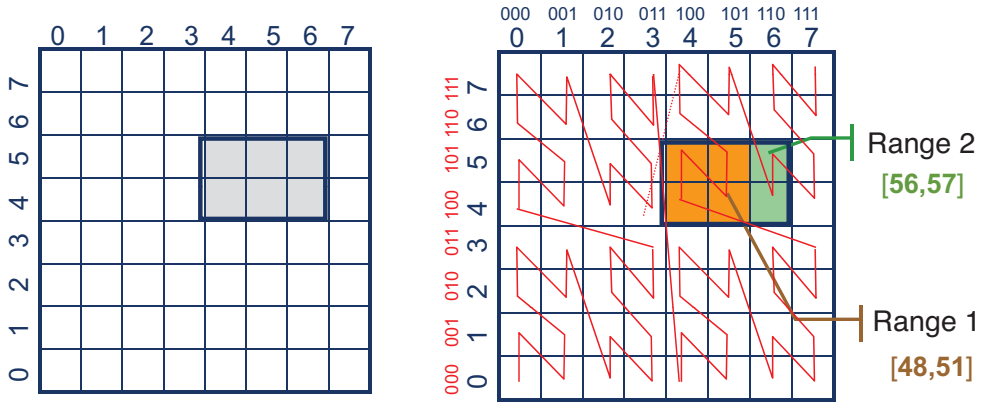
$$\underbrace{[B_4]}_{size=15} \; ; \; \underbrace{[B_1 \| B_2 \| B_3 \| B_5]}_{size=3+4+2+3=12} \; ; \; \ldots$$

Note that, since the bitmap for node n_4 *is larger than the target block size,* B_4 *is written to disk as a separate bitmap file; on the other hand, bitmaps for nodes* n_1*,* n_2*,* n_3*, and* n_5 *need to be concatenated into a single file to obtain a block larger than* $K = 10$ *units.*

Note that this block-based structure implies that the size of the files and the number of bitmap files on the disk will be upper bounded, but it also means that the cost of the bitmap reads will be lower bounded by K. Therefore, to obtain the best performance, repeated access to a block to fetch different bitmaps must be avoided through bitmap buffering and/or bitmap request clustering. In the next section, I discuss the use of cSHB index for range query processing. In Section 4.4, I experimentally analyze the impact of block-size on the performance of the proposed cSHB index structure.

4.3 Query Processing with the cSHB Index Structure

In this section, I describe how query workloads are processed using the cSHB index structure. In particular, I consider query workloads involving multiple range queries and propose *spatial bitmap selection* algorithms that select a subset of the bitmap nodes from the cSHB index structure for efficient processing of the query workload.



(a) 2D query range (b) Corresponding 1D ranges

Figure 4.4: Mapping of a Single Spatial Range Query to Two 1D Ranges on the Z-order Space: (a) A Contiguous 2D Query Range, $[sw = (4, 4); ne = (6, 5)]$ and (b) the Corresponding Contiguous 1D Ranges, [48,51] and [56,57], on the Z-Curve

4.3.1 Range Query Plans and Operating Nodes

In order to utilize the cSHB index for answering a spatial range query, I first need to map the range specification associated with the given query from the 2D space to the 1D space (defined by the Z-curve). As I see in Figure 4.4, due to the way the Z-curve spans the 2D-space, it is possible that a single contiguous query range in the 2D space may be mapped to multiple contiguous ranges on the 1D space. Therefore, given a 2D range query, q, I denote the resulting set of (disjoint) 1D range specifications, as RS_q.

Let us be given a query, q, with the set of 1D range specifications, RS_q. Naturally, there may be many different ways to process the query, each using a different set of bitmaps in the cSHB index structure, including simply fetching and combining only the relevant leaf bitmaps:

Example 4.3.1 (Alternative Range Query Plans) *Consider a query q with $q.rs =$*

$\langle(1,0),(3,1)\rangle$ *on the space shown in Figure 4.2. The corresponding 1D range,* $[2,11]$*, would cover the following leaf nodes of the hierarchy shown in Figure 4.3:* $RS_q = (000010, 000011, 001000, 001001, 001010, 001011)$*. The following are some of the alternative query plans for q using the proposed cSHB index structure:*

- Inclusive query plans: *The most straightforward way to execute the query would be to combine (bitwise OR operation) the bitmaps of the leaf nodes covered in 1D range,* $[2,11]$*. I refer to such plans, which construct the result by combining bitmaps of selected nodes using the OR operator, as* inclusive plans*.*

  An alternative inclusive plan for this query would be to combine the bitmaps of nodes $000010, 000011, 0010**$*:*

  $$B_{000010} \texttt{ OR } B_{000011} \texttt{ OR } B_{0010**}.$$

- Exclusive query plans: *In general, an exclusive query plan includes removal of some of the children or descendant bitmaps from the bitmaps of a parent or ancestor through the ANDNOT operation. One such exclusive plan would be to combine the bitmaps of all leafs nodes, except for* B_{000010}*,* B_{000011}*,* B_{001000}*,* B_{001001}*,* B_{001010}*,* B_{001011}*, into a bitmap* B_{non_result} *and return*

  $$B_{root}\texttt{ANDNOT}B_{non\_result}.$$

- Hybrid query plans: *Both inclusive and exclusive only query plans may miss efficient query processing alternatives. Hybrid plans combine inclusive and exclusive strategies at different nodes of the hierarchy. A sample hybrid query plan for the above query would be*

  $$\big(B_{0000**} \texttt{ ANDNOT } (B_{000000} \texttt{ OR } B_{000001})\big) \texttt{ OR } B_{0010**}.$$

As illustrated in the above example, a range query, q, on hierarchy H, can be answered using different query plans, involving bitmaps of the leaves and certain internal nodes of the hierarchy, collectively referred to as the *operating nodes of a query plan*. In Section 4.3.3, I present algorithms for selecting the operating nodes for a given workload, Q; but first we discuss the cost model that drives the selection process.

4.3.2 Cost Models and Execution Strategies

In cSHB, the bitwise operations needed to construct the result are performed on compressed bitmaps directly, without having to decompress them.

Cost Model for Individual Operations

I consider two cases: (a) logical operations on disk-resident compressed bitmaps and (b) logical operations on in-buffer compressed bitmaps.

Operations on Disk-Resident Compressed Bitmaps In general, when the logical operations are implemented on compressed bitmaps that reside on the disk, the time taken to read a bitmap from the secondary storage to the main memory dominates the overall bitwise manipulation time [31]. The overall cost is hence proportional to the size of the (compressed) bitmap file on the secondary storage.

Let us consider a logical operation on bitmaps B_i and B_j. Let us assume that $T(B_i)$ and $T(B_j)$ denotes the blocks in which B_i and B_j are stored, respectively. Since multiple bitmaps can be stored in a single block, it is possible that B_i and B_j are in the same block. Hence, let us further assume that $\mathcal{T}_{(B_i,B_j)}$ is the set of unique blocks that contain the bitmaps, B_i and B_j. Then the overall I/O cost is:

$$cost_{io}(B_i \texttt{ op } B_j) = \alpha_{IO}\big(\sum_{T \in \mathcal{T}_{(B_i,B_j)}} size(T)\big),$$

where α_{IO} is an I/O cost multiplier and `op` is a binary bitwise logical operator. A similar result also holds for the unary operation NOT.

Operations on In-Buffer Compressed Bitmaps When the compressed bitmaps on which the logical operations are implemented are already in-memory, the disk access cost is not a factor. However, also in this case, the cost is proportional to the sizes of the compressed bitmap files in the memory, independent of the specific logical operator that is involved [94], leading to

$$cost_{cpu}(B_i \texttt{ op } B_j) = \alpha_{cpu}\big(size(B_i) + size(B_j)\big),$$

where α_{cpu} is the CPU cost multiplier. A similar result also holds for the unary operation NOT.

Cost Models for Multiple Operations

In this section, a cost model is considered which assumes that blocks are disk-resident. Therefore, I consider a storage hierarchy that consists of a disk (which stores all bitmaps), RAM (as a buffer that stores all relevant bitmaps), and L3/L2 caches (that stores currently needed bitmaps).

Buffered Strategy In the *buffered* strategy, visualized in Figure 4.1, the bitmaps that correspond to any leaf or non-leaf operating nodes for the query plan of a given query workload, Q, are brought into the buffer once and cached for later use. Then, for each query $q \in Q$, the corresponding result bitmap is extracted using these buffered operating node bitmaps. Consequently, if a node is an operating one for more than one $q \in Q$, it is read from the disk only once (and once for each query from the memory). Let us assume that $\mathcal{T}_{ON_Q}$ denotes the set of unique blocks that contains

all the necessary operating nodes given a query workload Q(ON_Q). This leads to the overall processing cost, $time\_cost_{buf}(Q, ON_Q)$, of

$$\underbrace{\alpha_{IO}\left(\sum_{T\in\mathcal{T}_{ON_Q}} size(T)\right)}_{read\ cost} + \underbrace{\alpha_{cpu}\left(\sum_{q\in Q}\sum_{n_i\in ON_q} size(B_i)\right)}_{operating\ cost}.$$

Since all operating nodes need to be buffered, this execution strategy requires a total of $storage\_cost_{buf}(Q, ON_Q) = \sum_{n_i\in ON_Q} size(B_i)$ buffer space. Note that, in general, $\alpha_{IO} > \alpha_{cpu}$. However, in Section 4.4, I see that the number of queries in the query workload and query ranges determine the relative costs of in-buffer operations vs. disk I/O.

The buffered strategy has the advantage that each query can be processed individually on the buffered bitmaps and the results for each completed query can be pipelined to the next operator without waiting for the results of the other queries in the workload. This reduces the memory needed to temporarily store the result bitmaps. However, in the buffered strategy, the buffer needed to store the operating node bitmaps can be large.

Incremental Strategy The *incremental* strategy avoids buffering of all operating node bitmaps simultaneously. Instead, all leaf and non-leaf operating nodes are fetched from the disk one at a time *on demand* and results for each query are constructed incrementally. This is achieved by considering one internal operating node at a time and, for each query, focusing only on the leaf operating nodes under that internal node. For this purpose, a *result accumulator bitmap*, Res_j, is maintained for each query in $q_j \in Q$ and each operating node read from the disk is applied directly on this result accumulator bitmap.

While it does not need buffer to store all operating node bitmaps, the incremental strategy may also benefit from partial caching of the relevant blocks. This is because,

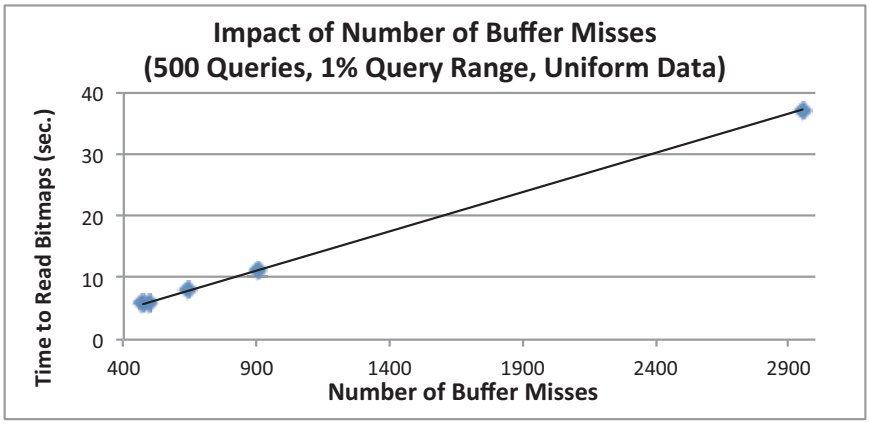


Figure 4.5: Buffer Misses and the Overall Read Time (Data and Other Details are Presented in Section 4.4)

while each internal node needs to be accessed only once, each leaf node under this internal node may need to be brought to the memory for multiple queries. Moreover, since the data is organized in terms of blocks, rather than individual nodes (Section 4.2.2), a single block may serve multiple nodes to different queries. When sufficient buffer is available to store the *working set* of blocks (containing the operating leaf nodes under the current internal node), the execution cost, $time\_cost_{inc}(Q, ON_Q)$, of the incremental strategy is identical to that of the buffered strategy. Otherwise, as illustrated in Figure 4.5, the read cost component is a function of buffer misses, $\alpha_{IO} \times \#\_buffer\_misses$, which itself depends on the size of the buffer and the clustering of the data.

The storage complexity[3] is $storage\_cost_{inc}(Q, ON_Q) = \sum_{q_j \in Q} size(Res_j)$ plus the space needed to maintain the most recently read blocks in the working set. Experiments reported in Section 4.4 show that, for the considered data sets, the sizes of the working sets are small enough to fit into the L3-caches of many modern hardware.

4.3.3 Selecting the Operating Bitmaps for a Given Query Workload

To process a range query workload, Q, on a data set, $\mathcal{D}$, with the underlying cSHB hierarchy H, I need to select a set of operating bitmap nodes, ON_Q, of H from which I can construct the results for all $q_j \in Q$, such that $time\_cost(Q, ON_Q)$ is the minimum among all possible sets of operating bitmaps for Q. It is easy to see that the number of alternative sets of operating bitmaps for a given query workload Q is exponential in the size of the hierarchy H. Therefore, instead of seeking the set of operating bitmaps among all subsets of the nodes in H, I focus the attention on the *cuts* of the hierarchy, defined as follows:

Definition 4.3.1 (Cuts of H Relative to Q) *A* complete cut, C, of a hierarchy, H, relative to a query load, Q, *is a subset of the* internal nodes (including the root) *of the hierarchy, satisfying the following two conditions:*

- validity: *there is exactly one node on any root-to-leaf branch in a given cut; and*
- completeness: *the nodes in C collectively cover every possible root-to-leaf branch for all leaf nodes in the result sets for queries in Q.*

If a set of internal nodes of H only satisfies the first condition, then I refer to the cut as an incomplete cut.

[3]The space complexity of the incremental strategy can be upper-bounded if the results for the queries in Q can be pipelined to the next set of operators *progressively* as partial results constructed incrementally.

As visualized in Figure 4.1, given a cut C, cSHB queries are processed by using **only the bitmaps of the nodes in this cut, along with some of the leaf bitmaps necessary to construct results of the queries in Q.** In the rest of this subsection, I first describe how queries are processed given a cut, C, of H and then present algorithms that search for a cut, C, given a workload, Q.

Range Query Processing with Cuts

It is easy to see that any workload, Q, of queries can be processed by any (even incomplete) cut, C, of the hierarchy and a suitable set of leaf nodes: Let R_q denote the set of leaf nodes that appear in the result set of query $q \in Q$ and $\bar{R}_q$ be the set of leaf nodes that do not appear in the result set. Let also R_q^C be the set of the result leaves covered by a node in C. Then, one possible way to construct the result bitmap, B_q, is as follows:

$$B_q = \left(\left(\underset{n_i \in C}{\texttt{OR}} B_i \right) \texttt{OR} \underbrace{\left(\underset{n_i \in R_q \setminus R_q^C}{\texttt{OR}} B_i \right)}_{inclusions} \right) \underset{n_j \in R_q^C \cap \bar{R}_q}{\texttt{ANDNOT}} \underbrace{B_j}_{exclusions} .$$

Intuitively any result nodes that are not covered by the cut need to be *included* in the result using a bitwise OR operation, whereas any leaf node that is not in any result needs to be *excluded* using an ANDNOT operation. Consequently,

- if $C \cap R_q = \emptyset$, an *inclusion-only* plan is necessary,
- an *exclusion-only* plan is possible only if C covers R_q completely.

Naturally, given a range query workload, Q, different query plans with different cuts will have different execution costs. The challenge is, then,

- to select an appropriate cut, C, of the hierarchy, H, for query workload, Q, and

- to pick, for each query $q_j \in Q$, a subset $C_j \in C$ for processing q_j,

in such a way that these will minimize the overall processing cost for the set of range queries in Q. Intuitively, I want to include in the cut, those nodes that will not lead to a large number of exclusions and cannot be cheaply constructed by combining bitmaps of the leaf nodes using OR operations.

Cut Bitmap Selection Process

Given the above cut-based query processing model, in this section we propose a cut selection algorithm consisting of two steps: (a) a per-node cost estimation step and (b) a bottom-up cut-node selection step. I next describe each of these two steps.

Node Cost Estimation First, the process assigns an estimated cost to those hierarchy nodes that are relevant to the given query workload, Q. For this, the algorithm traverses through the hierarchy, H, in a top-down manner and identifies part, R, of the hierarchy relevant for the execution of at least one query, $q \in Q$ (i.e., for at least one query, q, the range associated with the node and the query range intersect). Note that this process also converts the range in 2-D space into 1-D space by identifying the relevant nodes in the hierarchy. Next, for each internal node, $n_i \in R$, a cost, $cost_i$, is estimated assuming that this node and its leaf descendants are used for identifying the matches in the range S_i. The outline of this process is presented in Algorithm 7 and is detailed below:

• *Top-Down Traversal and Pruning.* Line 5 indicates that the process starts at the root and moves towards the leaves. For each internal node, n_i, being visited, first, the set, $Q(n_i) \subseteq Q$, of queries for which n_i is relevant is identified by intersecting the ranges of the queries relevant to the parent (i.e., $Q(parent(n_i))$) with the range of n_i.

Algorithm 7 Cost and Leaf Access Plan Assignment Algorithm

1: **Input:** Hierarchy H, Query Workload Q

2: **Outputs:** Query workload, $Q(n_i)$, and cost estimate, $cost_i$, for each node, $n_i \in H$; leaf access plan, $E_{i,j}$, for all node/query pairs $n_i \in H$ and $q_j \in Q(n_i)$; a set, $R \subseteq I_H$, or relevant internal nodes

3: **Initialize:** $R = \emptyset$

4: **procedure** COST_AND_LEAFACCESSPLANASSIGNMENT

5: **for** each internal node $n_i \in I_H$ in top-down fashion **do**

6: **if** $n_i =$ "$root$" **then**

7: $Q(n_i) = Q$

8: **else**

9: $Q(n_i) = \{q \in Q(parent(n_i))\ s.t.\ (q.rs \cap S_i) \neq \emptyset\}$

10: **end if**

11: **if** $Q(n_i) \neq \emptyset$ **then**

12: add n_i into R

13: **end if**

14: **end for**

15: **for** each node $n_i \in R$ in a bottom-up fashion **do**

16: **for** $q_j \in Q(n_i)$ **do**

17: Compute $icost(n_i, q)$

18: Compute $ecost(n_i, q)$

19: Compute the leaf access plan, $E_{i,j}$, as

$$E_{i,j} = [\texttt{ecost}(\texttt{n}_\texttt{i}, \texttt{q}_\texttt{j}) < \texttt{icost}(\texttt{n}_\texttt{i}, \texttt{q}_\texttt{j})]$$

20: **end for**

21: Compute the leaf access cost, $leaf\_cost_i$, as

$\left(\sum_{q_j \in Q(n_i)} E_{i,j} \times ecost(n_i, q_j) + (1 - E_{i,j}) \times icost(n_i, q_j)\right)$

22: **end for**

23: **end procedure**

More specifically,

$$Q(n_i) = \{q \in Q(parent(n_i)) \; s.t. \; (q.rs \cap S_i) \neq \emptyset\}.$$

If $Q(n_i) = \emptyset$, then n_i and all its descendants are ignored, otherwise n_i is included in the set R.

• *Inclusive and Exclusive Cost Computation.* Once the portion, R, of the hierarchy relevant to the query workload is identified, next, the algorithm re-visits all internal nodes in R in a bottom-up manner and computes a cost estimate for executing queries in $Q(n_i)$: for each query, $q \in Q(n_i)$, the algorithm computes inclusive and exclusive leaf access costs:

- *Inclusive leaf access plan (Line 17):* If query, q, is executed using an inclusive plan at node, n_i, this means that the result for the range $(q.rs \cap S_i)$ will be obtained by identifying and combining (using bitwise ORs) all relevant leaf bitmaps under node n_i. Therefore, the cost of this leaf access plan is

$$icost(n_i, q) = \sum_{(n_j \in leafDesc(n_i)) \wedge ((q.rs \cap S_j) \neq \emptyset)} size(B_j).$$

  This value can be computed incrementally, simply by summing up the inclusive costs of the children of n_i.

- *Exclusive leaf access plan (Line 18):* If query, q, is executed using an exclusive leaf access plan at node, n_i, this means that the result for the range ($q.rs \cap S_i$) will be obtained by using B_i and then identifying and excluding (using bitwise ANDNOT operations) all irrelevant leaf bitmaps under node n_i. Thus, I compute the exclusive leaf access plan cost, $ecost(n_i, q)$, of this query at node n_i as

$$\begin{aligned} ecost(n_i, q) &= size(B_i) \\ &+ \sum_{(n_j \in leafDesc(n_i)) \wedge ((q.rs \cap S_j) = \emptyset)} size(B_j) \end{aligned}$$

or equivalently as

$$\begin{aligned} ecost(n_i, q) &= size(B_i) + \left(\sum_{n_j \in leafDesc(n_i)} size(B_j) \right) \\ &- icost(n_i, q) \end{aligned}$$

Since the initial two terms above are recorded in the index creation time, the computation of exclusive cost is a constant time operation.

• *Overall Cost Estimation and the Leaf Access Plan.* Given the above, I can find the best strategy for processing the query set $Q(n_i)$ at node n_i by considering the overall estimated cost term, $cost(n_i, Q(n_i))$, defined as

$$\underbrace{\left(\sum_{q_j \in Q(n_i)} E_{i,j} \times ecost(n_i, q_j) + (1 - E_{i,j}) \times icost(n_i, q_j) \right)}_{leaf\ access\ cost\ for\ all\ relevant\ queries}$$

where $E_{i,j} = 1$ means an exclusive leaf access plan is chosen for query, q_j, at this node and $E_{i,j} = 0$ otherwise.

Cut Bitmap Selection Once the nodes in the hierarchy are assigned estimated costs as described above, the cut that will be used for query processing is found

Algorithm 8 Cut Selection Algorithm

1: **Input:** Hierarchy H; per-node query workload $Q(n_i)$; per-node cost estimates $cost_i$; and the corresponding leaf access plans, $E_{i,j}$, for node/query pairs $n_i \in H$ and $q_j \in Q(n_i)$; the set, $R \subseteq I_H$, or relevant internal nodes
2: **Output:** All-inclusive, C_I, and Exclusive, C_E, cut nodes
3: **Initialize:** $Cand = \emptyset$
4: **procedure** FINDCUT
5: **for** each relevant internal node n_i in R in a bottom-
6: up fashion **do**
7: Set $internal\_children = children(n_i) \cap I_H$;
8: **if** $internal\_children = \emptyset$ **then**
9: add n_i to $Cand$;
10: $rcost_i = cost_i$
11: **else**
12: $costChildren = \sum_{n_j \in internal\_children} rcost_j$
13: $rcostIO_i = findBlockIO(n_i)$
14: **for** each child n_j in $internal\_children$ **do**
15: $costChildrenIO = costChildrenIO + findBlockIO(n_j)$
16: **end for**
17: **if** $(rcost_i + rcostIO_i) \leq (costChildren + costChildrenIO)$ **then**
18: **for** each descendant n_k of n_i in $Cand$ **do**
19: remove n_k from $Cand$;
20: **if** n_k is the only node to read from $T(B_k)$ **then**
21: mark $T(B_k)$ as "not-to-read";
22: **end if**
23: **end for**

24: add n_i to $Cand$;

25: $rcost_i = cost_i$

26: mark $T(B_i)$ as "to-read";

27: **else**

28: $rcost_i = costChildren$

29: **end if**

30: **end if**

31: **end for**

32: $C_E = \{n_i \in Cand\ s.t.\ \exists_{q_j \in Q(n_i)} E_{i,j} == 1\}$

33: $C_I = Cand / C_E$

34: **end procedure**

by traversing the hierarchy in a bottom-up fashion and picking nodes based on their estimated costs[4]. The process is outlined in Algorithm 8. Intuitively, for each internal node, $n_i \in I_H$, the algorithm computes a revised cost estimate, $rcost_i$, by comparing the cost, $cost_i$, estimated in the earlier phase of the process, with the total revised costs of n_i's children:

- In Line 13, the function $findBlockIO(n_i)$ returns the cost of reading the block $T(B_i)$. If this block has already been marked "to-read", then the reading cost has already been accounted for, so the cost is zero. Otherwise, the cost is equal to the size of the block $T(B_i)$, as explained in Section 4.3.2.

- It is possible that a block T is first marked "to-read" and then, later in the process, marked "not-to-read", because for the corresponding nodes in the cut, more suitable ancestors are found and the block is no longer needed (Line 21).

[4]Note that this bottom-up traversal can be combined with the bottom-up traversal of the prior phase. I am describing them as separate processes for clarity.

- If $cost_i$ is smaller (Line 17), then n_i and its leaf descendants can be used for identifying the matches to the queries in the range S_i. In this case, no revision is necessary and the revised cost, $rcost_i$ is equal to $cost_i$. Any descendants of n_i are removed from the set, $Cand$, of cut candidates and n_i is inserted instead.

- If, on the other hand, the total revised cost of n_i's children is smaller than $cost_i$, then matches to the queries in the range S_i can be more cheaply identified by considering the descendants of n_i, rather than n_i itself (Line 27). Consequently, in this case, the revised cost, $rcost_i$, is set to

$$rcost_i = \sum_{n_j \in children(n_i)} rcost_j.$$

As I experimentally show in Section 4.4, the above process has a small cost. This is primarily because, during bottom-up traversal, only those nodes that have not been pruned in the previous top-down phase are considered. Once the traversal is over, the nodes in the set, $Cand$, of cut candidates are reconsidered and those that include exclusive leaf access plans are included in the exclusive cut set, C_E, and the rest are included in the all-inclusive cut set, C_I.

Caching of Cut and Leaf Bitmaps During query execution, the bitmaps of the nodes in C_E are read into a cut bitmaps buffer, whereas the bitmaps for the nodes in C_I do not need to be read as the queries will be answered only by accessing relevant leaves under the nodes in C_I. The blocks that contain bitmaps of these relevant leaves are stored in an LRU-based cache so that leaf bitmaps can be reused by multiple queries.

Complexity

The bitmap selection process consists of two steps: (a) a per-node cost estimation step and (b) a cut bitmap selection step. Each of these steps visit only the *relevant* nodes of the hierarchy. Therefore, if I denote the set of nodes of the hierarchy, H, that intersect with any query in Q, as $H(Q)$, then the overall work is linear in the size of $H(Q)$.

During the cost estimation phase, for each visited node, n_i, an inclusive and exclusive cost is estimated for any query that intersects with this node. Therefore, the worst case time cost of the overall process (assuming that all queries in Q intersect with all nodes in $H(Q)$) is $O(|Q| \times |H(Q)|)$.

4.4 Experimental Evaluation

In this section, I evaluate the effectiveness of the proposed *compressed spatial hierarchical bitmap (cSHB)* index structure using spatial data sets with different characteristics, under different system parameters. To assess the effectiveness of cSHB, we also compare it against alternatives.

I ran the experiments on a quad-core Intel Core i5-2400 CPU @ 3.10GHz machine with 8.00GB RAM, and a 3TB SATA Hard Drive with 7200 RPM and 64MB Buffer Size, and in the same Windows 7 environment. All codes were implemented and run using Java v1.7.

4.4.1 Alternative Spatial Index Structures and the Details of the cSHB Implementation

As alternatives to cSHB, I considered different systems operating based on different spatial indexing paradigms. In particular, we considered spatial extensions of PostgreSQL called PostGIS [64], of a widely used commercial DBMS (which we refer

to as DBMS-X), and of Lucene [61]:

- PostGIS [64] creates spatial index structures using an R-tree index implemented on top of GiST.
- DBMS-X maps 2D space into a 1D space using a variation of Hilbert space filling curve and then indexes the data using B-trees.
- Apache Lucene [61, 47], a leading system for text indexing and search, provides a spatial module that supports geo-spatial range queries in 2D space using quadtrees and prefix-based indexing. Intuitively, the space is partitioned using a MX-quadtree structure (where all the leaves are at the same level and a given region is always partitioned to its quadrants at the center [78]) and each root-to-leaf path is given a unique path-string. These path-strings are then indexed (using efficient prefix-indexing algorithms) for spatial query processing.

Since database systems potentially have overheads beyond pure query processing needs, I also considered disk-based implementations of R*-tree [14] and the Hilbert R-tree [49]. For this purpose, I used the popular XXL Java library [19]:

- A packed R*-tree, with average leaf node utilization $\sim 95\%$ (page size 4MB).
- A packed Hilbert R-tree, with average leaf node utilization $\sim 99\%$ (page size 4MB).

I also implemented the proposed cSHB index structure on top of Lucene. In particular, I used the MX-quadtree hierarchy created by Lucene as the spatial hierarchy for building cSHB. I also leveraged Lucene's (Java-based) region comparison libraries to implement range searches. The compressed bitmaps and compressed domain logical operations were implemented using the JavaEWAH library [56].

Data set	**#points**	**#points per (non-empty) cell** ($h = 10$)		
		Min.	Avg.	Max.
Synthetic (Uniform)	100M	54	95	143
Gowalla (Clustered)	6.4M	1	352	312944
OSM (Clustered)	688M	1	3422	1.2M

Table 4.1: Data Sets and Clustering

As described in Section 4.3.2, I have introduced two different execution strategies, namely buffered and incremental strategies, and presented the corresponding cost models. The buffered strategy assumes that all the bitmaps corresponding to any leaf or non-leaf operating nodes for the query plan can be brought into the buffer once and cached for later use. The incremental strategy, however, relaxes this assumption and consequently, does not need sufficient buffer space to store all operating node bitmaps; instead, the incremental strategy relies on partial caching of only the relevant blocks. While the data sets considered in our experiments could all benefit from the buffered strategy given a sufficiently modern hardware, in the experiments presented in this thesis, our goal is to see whether the cSHB method is still advantageous and competitive against other systems when this is not the case. Therefore, in the experiments presented in this thesis, I consider those cases where cSHB cannot leverage a full buffered strategy for high performance and, instead, needs to rely on partial caching through the proposed incremental strategy.

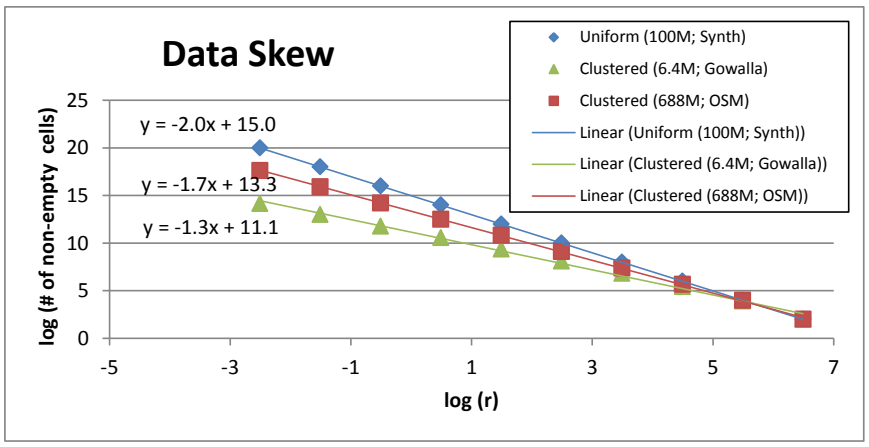


Figure 4.6: Data Skew

4.4.2 Data Sets

For the experiments, I used three data sets: *(a) a uniformly distributed* data set that consists of 100 million synthetically generated data points. These points are mapped to the range $\langle -180, -90\rangle$ to $\langle 180, 90\rangle$, *(b) a clustered* data set from Gowalla, which contains the locations of check-ins made by users. This data set is downloaded from the Standford Large Network Dataset Collection [57], and *(c) a clustered* data set from OpenStreetMap (OSM) [4] which contains locations of different entities distributed across North America. The OSM data set consists of approximately 688 million data points in North America. I also normalized both the real data sets to the range $\langle -180, -90\rangle$ to $\langle 180, 90\rangle$. In order to obtain a fair comparison across all index structures and the data sets, all three data sets are mapped onto a $2^h \times 2^h$ space and the positions of the points in this space are used for indexing. Table 4.1 provides an overview of the characteristics of these three very different data sets. Figure 4.6 re-confirms the data skew in the three data sets using the box-counting method proposed in [16]: in the figure, the lower the negative slope, the more skewed the data. The figure shows that the clustered Gowalla data set has the largest skew.

Parameter	Value range
Block Size (MB)	0.5; **1**; 2.5; 5; 10
Query range size	0.5% **1**%; 5%
$\|Q\|$	100; **500**; 1000
h	9; **10**; 11
Buffer size (MB)	2; 3; 5; 10; **20**; 100

Table 4.2: Parameters and Default Values (in bold)

4.4.3 Evaluation Criteria and Parameters

I evaluate the effectiveness of the proposed *compressed spatial hierarchical bitmap (cSHB)* index structure by comparing its (a) index creation time, (b) index size, and (c) query processing time to those of the alternative index structures described above under different parameter settings. Table 4.2 describes the parameters considered in these experiments and the default parameter settings.

Since the goal is to assess the contribution of the index in the cost of the query plans, all index structures in the comparison used index-only query plans. More specifically, I executed a count($*$) query and configured the index structures such that only the index is used to identify the relevant entries and count them to return the results. Consequently, only the index files are used and data files are not accessed. Note that all considered index structures accept square-shaped query ranges.
The range sizes indicated in Table 4.2 are the lengths of the boundaries relative to the size of the considered 2D space. These query ranges in the query workloads are generated uniformly.

Dataset	cSHB	Lucene	DBMS-X	PostGIS	R*-tree	Hilbert R-tree
Synthetic	1601	2396	3865	4606	2160	2139
Gowalla	24	114	232	112	22	20
OSM	2869	12027	30002	76238	18466	17511

Table 4.3: Index Creation Time (in seconds)

Dataset	cSHB	Lucene	DBMS-X	PostGIS	R*-tree	Hilbert R-tree
Synthetic	10900	5190	1882	8076	3210	1510
Gowalla	44	220	121	600	211	100
OSM	2440	22200	12959	61440	22100	10400

Table 4.4: Index Size on Disk (MB)

4.4.4 Discussion of the Indexing Results

Indexing Time. Table 4.3 shows the index creation times for different systems and index structures, for different data sets (with different sizes and uniformity): cSHB index creation is fastest for the larger Synthetic and OSM data sets, and competitive for the smaller Gowalla data set. As the data size gets larger, the alternative index structures become significantly slower, whereas cSHB is minimally affected by the increase in data size. The index creation time also includes the time spent on creating the hierarchy for cSHB.

Index Size. Table 4.4 shows the sizes of the resulting index files for different systems and index structures and for different data sets. As I see here, cSHB provides a competitive index size for uniform data (where compression is not very effective).

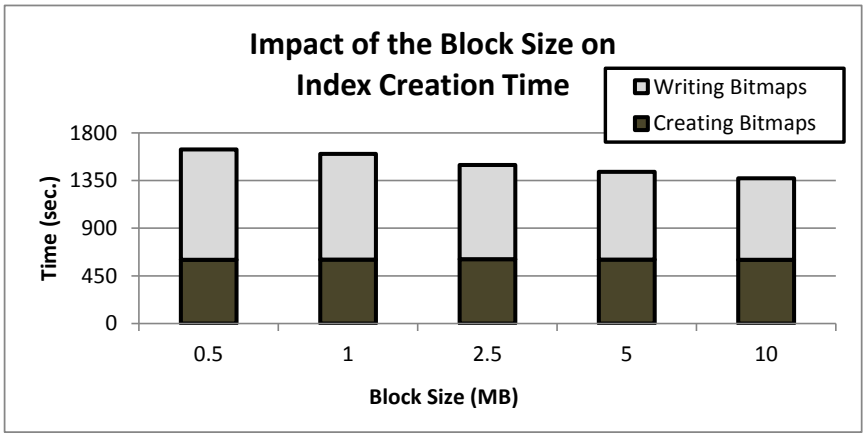


Figure 4.7: Impact of the Block Size on Index Creation Time of cSHB (Uniform Data Set)

On the other hand, on clustered data, cSHB provides very significant gains in index size – in fact, even though the clustered data set, OSM, contains more points, cSHB requires less space for indexing this data set than it does for indexing the uniform data set.

Range	cSHB	Lucene	DBMS-X	PostGIS	R*-tree	Hilbert R-tree	cSHB-LO
	Synthetic (Uniform; 100M)						
0.5%	35	123	414	12887	2211	4391	52
1%	42	131	345	28736	2329	4480	59
5%	137	187	368	72005	2535	4881	1700
	Gowalla (Clustered; 6.4M)						
0.5%	2	2	24	19	8	24	2
1%	3	3	29	34	11	26	3
5%	3	48	37	194	20	45	5
	OSM (Clustered; 688M)						
0.5%	13	23	303	1129	3486	4368	13
1%	15	30	645	4117	3889	5599	14
5%	28	66	15567	18172	4626	6402	78

Table 4.5: Comparison of Search Times for Alternative Schemes and Impact of the Search Range on the Time to Execute 500 Range Queries (in seconds)

Impact of Block Size. As I discussed in Section 4.2.2, cSHB writes data on the disk in a blocked manner. In Figure 4.7, I see the impact of the block sizes on the time needed to create the bitmaps. As I see here, one advantage of using blocked storage is that the larger the blocks used, the faster the index creation becomes.

4.4.5 Discussion of the Search Results

Impact of the Search Range. Table 4.5 shows the impact of the query range on search times for 500 queries under the default parameter settings, for different

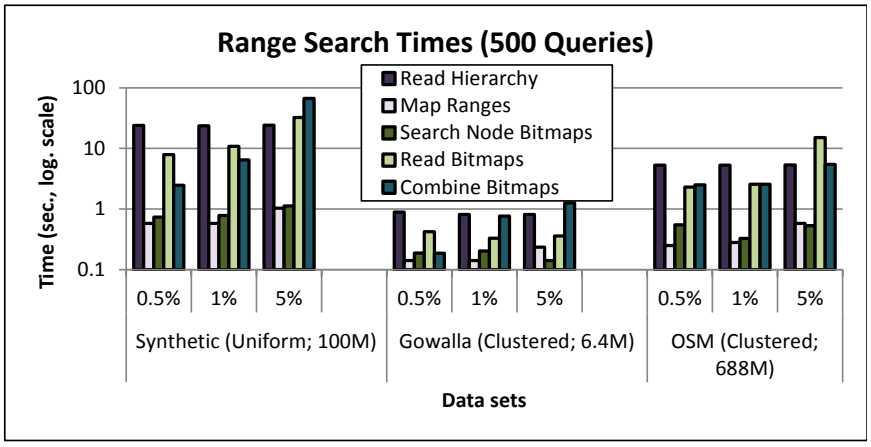
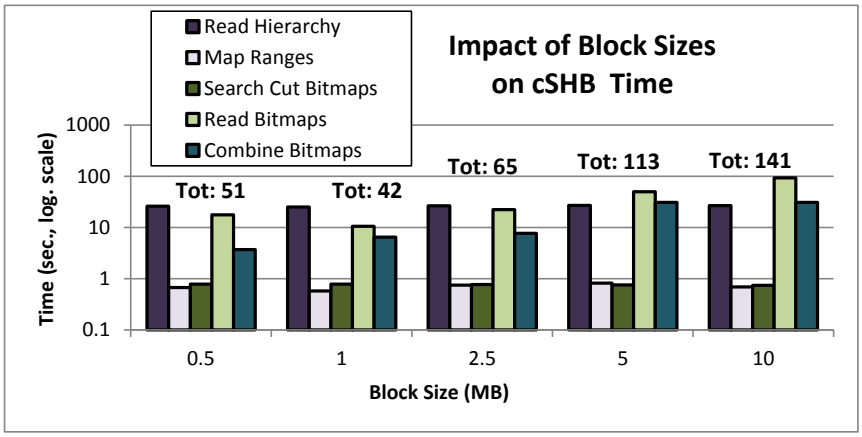


Figure 4.8: cSHB Execution Breakdown

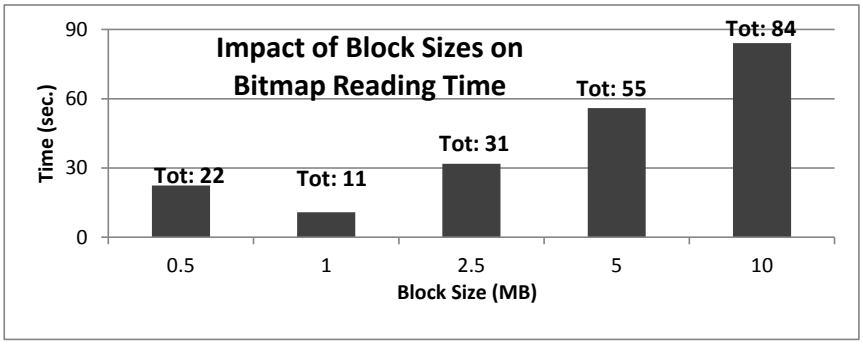
systems. As I expected, as the search range increases, the execution time becomes larger for all alternatives. However, cSHB provides the best performance for all ranges considered, especially for the clustered data sets. Here, I also compare cSHB with its leaf-only version (called cSHB-LO), where instead of a cut consisting of potentially internal nodes, I only choose the leaf nodes for query processing. As you can see from the figure, while cSHB-LO is a good option for very small query ranges (0.5% and 1%), it becomes very slow as the query range increases (since the number of bitwise operations increases, and it is not able to benefit from clustering).

Execution Time Breakdown. Figure 4.8 provides a breakdown of the various components of cSHB index search (for 500 queries under the default parameter settings): The bitmap selection algorithm presented in Section 4.3.3 is extremely fast. In fact, the most significant components of the execution are the times needed for reading the hierarchy into memory[5], and for fetching the selected bitmaps from the disk into the buffer, and performing bitwise operations on them. As expected, this component sees a major increase as the search range grows, whereas the other costs are more or

[5]Once a hierarchy is read into the memory, the hierarchy does not need to be re-read for the following queries.



(a) Impact of block size on overall cSHB execution time



(b) Impact of block size on bitmap reading time

Figure 4.9: Impact of the Block Size (500 queries, 1% Query Range, Uniform Data)

less independent of the sizes of the query ranges.

Impact of the Block Sizes. As I see above, reading bitmaps from the disk and operating on them is a major part of cSHB query execution cost; therefore these need to be performed as efficiently as possible. As I discussed in Section 4.2.2, cSHB reads data from the disk in a blocked manner. In Figure 4.9, I see the impact of the block sizes on the execution time of cSHB, including the time needed to read bitmaps from the disk. As I see here, small blocks are disadvantageous (due to the directory management overhead they cause). Very large blocks are also disadvantageous as, the larger the block gets, the larger becomes the amount of redundant data read

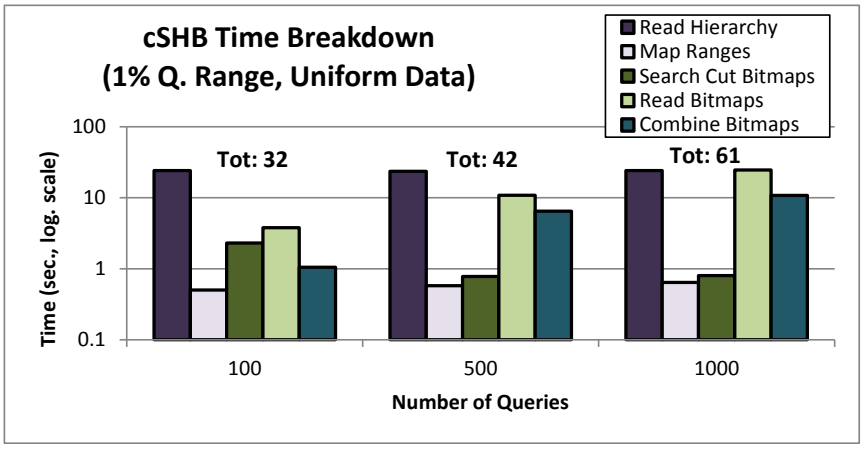


Figure 4.10: Impact of the Number of Queries on the Execution Time of cSHB (1% Query Range, Uniform Data)

for each block access. As I see in the figure, for the configuration considered in the experiments, 1MB blocks provided the best execution time.

Impact of the Number of Queries in the Workload. Figure 4.10 shows the total execution times as well as the breakdown of the execution times for cSHB for different number of simultaneously executing queries. While the total execution time increases with the number of simultaneous queries, the increase is sub-linear, indicating that there are savings due to the shared processing across these queries. Also, in Section 4.3.2, I had observed that the number of queries in the query workload and query ranges determine the relative costs of in-buffer operations vs. disk I/O. In Figures 4.8 and 4.10, I see that this is indeed the case.

Impact of the Depth of the Hierarchy. Figure 4.11 shows the impact of the hierarchy depth on the execution time of cSHB: a $4\times$ increase in the number of cells in the space (due to a 1-level increase in the number of levels of the hierarchy) results in $< 4\times$ increase in the execution time. Most significant contributors to this increase are the time needed to read the hierarchy and the time for bitmap operations.

Impact of the Cache Buffer. As we discussed in Section 4.3.2, the incremen-

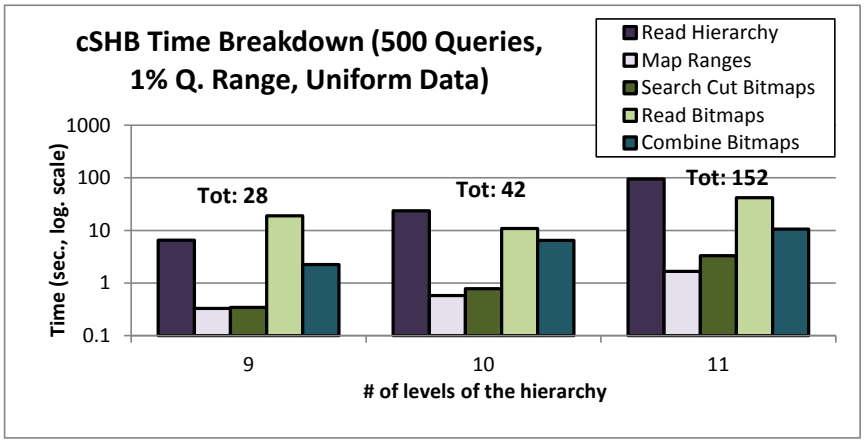


Figure 4.11: Impact of the Depth of the Hierarchy (500 queries, 1% Query Range, Uniform Data)

Q.Range (on 100M data)	Min	Avg.	Max.
0.5%	1	2.82	36
1%	1	2.51	178
5%	1	1.02	95

Table 4.6: Working Set Size in Terms of 1MB Blocks

tal scheduling algorithm keeps a buffer of blocks containing the working set of leaf bitmaps. As Table 4.6 shows, the average size of the working set is fairly small and can easily fit into the L3 caches of modern hardware. Table 4.7 confirms that a small buffer, moderately larger than the average working set size, is sufficient and larger buffers do not provide significant gains.

Query Range	**Buffer Size**					
	2MB	**3MB**	**5MB**	**10MB**	**20MB**	**100MB**
0.5%	11.8	11.3	10.9	10.6	10.5	10.2
1%	24.2	19.1	18.1	17.5	17.3	16.3
5%	823.8	399.9	155.9	105.8	101.6	94.9

Table 4.7: Impact of the Buffer Size on Execution Time (in seconds, for 500 queries, 100M data)

Chapter 5

EXECUTION OF RANGE QUERY WORKLOADS IN HIGH-DIMENSIONAL SPACES

5.1 Introduction

The similarity search problem in high dimensional spaces is a very well-known problem. Tree-based indexing structures (KD-tree [17], X-tree [18], SR-tree [51], etc.) have been shown to be effective for only up to ten dimensions [44]. Beyond dimensions greater than this, tree-based index structures are often out-performed even by linear scans (a problem popularly known as the *curse of dimensionality*. One solution to address this problem for higher dimensions is to look for *approximate* results instead of exact results. Especially given that, in many applications, 100% accuracy is not needed, searches that return points that are *close enough* to the query point rather than the closest ones, are often times, more effective and much faster than solutions that attempt to find exact query results [29].

One of the most popular solutions for approximate searching is called *Locality Sensitive Hashing (LSH)*, first proposed in [44]. The idea behind locality sensitive hashing is to map high dimensional data to lower dimensional representations, in such a way that searches are nevertheless reasonably accurate. In the lower dimensional space, which is obtained through *random projections*, data points are mapped to individual buckets based on a hash function. The intuition behind this method is that data points closer in the original space will be mapped to the same buckets with a higher probability in the lower dimensional space than dissimilar points. This may, however, lead to *misses* as well as *false positives*. Given a distance metric and a

corresponding locality sensitive hashing family (detailed in Section 5.1.4), LSH data structures control their precision and recall by using multiple independently chosen hash functions organized into several hash layers: intuitively, *conjunctively* combined hashes at each layer reduce *false positives*, whereas *disjunctively* combined layers of hash functions help avoid *misses* – often, at the expense of identifying candidate data elements that needs to be eliminated during post-processing.

Locality Sensitive Hashing has been studied and improved upon extensively in the research community [62, 98, 48, 59, 11, 29, 36]. In particular, most existing solutions follow the original framework proposed in the seminal LSH work [44], which aimed to address the (r, c)-near neighbors problem: every point p that lies within a distance of r from query point q should be reported with a probability guarantee of at least $1 - \delta$ (where δ is a user-specified error probability), whereas points that lie beyond distance of $c \times r$, for some $c > 1$, from the query point q should have a very low likelihood of being included in the query result[1]. Intuitively, the user provides a *success probability* that decides the balance between the accuracy of the results and query processing speed for a given radius. Higher the target success probability, higher is the accuracy for the results, but slower the query processing speed, and vice-versa. Since using more layers helps eliminate misses, this may increase the number of *candidate points* that need to be enumerated and potentially eliminated during post-processing. The parameters of the index structure need to be selected carefully to achieve the target accuracy for the given target radius.

[1]While the original work address searches for Hamming distance, this was quickly extended to Euclidean and other distances [29].

5.1.1 Motivation

Similarity search queries are a crucial set of queries in multimedia applications. Multimedia data is often represented by the most important points that can help identify the data, without having to store the entire data. These points, also called *features*, are extracted using popular localized feature extraction algorithms such as SIFT [60] or SURF [12]. A user may want to find all the data objects in the database that are similar to a particular data object. Each object is represented by a set of features, and a similarity query has to be performed on each of these features in order to find the objects in the database that have similar features. Collectively, these individual query points form a query set. In traditional and state-of-the-art LSH-based techniques, users input a success guarantee for each individual query point, instead of a guarantee for the entire query set. A lower guarantee on these individual query points can lead to overall misses, and a higher guarantee can lead to redundant and wasteful work for the whole set. Returning results (or features) that satisfy all the query points is an expensive process. Returning approximate results can save time as well as return "good enough" results. Hence, users may only be interested in features that satisfy a certain number of query points instead of all the points in the query set. Then the challenge is to design an index structure that can take a target number of points and a guarantee on the entire set as an input, and return the data points, that satisfy at least "target" number of query points in the query set while satisfying the success guarantee, in an efficient and waste-avoiding manner.

5.1.2 Research Contributions

To deal with the challenge of giving a guarantee for a set query (instead of individual query points), I design and develop a novel index structure, *Point Set LSH*

(PSLSH), which creates a *layer* of an LSH index structure based on the Hamming distance (called PSLSH_H), on top of the existing LSH index structure (that is based on the Euclidean distance - called PSLSH_E). Since, especially for secondary-storage based implementations, the post-processing step (where the precise positions for the enumerated candidates are fetched from storage to identify and eliminate false positives[2]) is the most expensive step, the goal of PSLSH is to design the index structure such that less candidates are generated. In particular, given a total budget of hash functions to use, I present a novel strategy that decides how many hash functions PSLSH_E can use and how many hash functions PSLSH_H can use. The existing existing collision counting approach [36] is not designed to work effectively in a multi-level index structure. I present an extension to the collision counting approach so that it can effectively remove false positives in a multi-level index structure. I also present novel cost models that can effectively predict the total query set execution time for different strategies, and thus choose the most efficient design strategy. To the best of my knowledge, this is the first work that presents an index structure that can give guarantees for an entire set query instead of guarantees on individual query points. Experimental evaluations of PSLSH shows the effectiveness of the proposed index structure and the design strategies in avoiding wasted and redundant work while giving a guarantee on the entire set query.

5.1.3 Organization of the Chapter

The rest of the chapter is organized in the following way: in Section 5.2, I describe the relevant works that try to solve the similarity search problem using LSH. I describe the (r, c)-Near Neighbor Problem and the preliminaries necessary to understand Point

[2]If the computed precise distance is more than the query radius, the candidate is a false positive and it is eliminated.

Set LSH in Section 5.1.4. In Section 5.4, I describe the design of Point Set LSH and provide the theoretical analysis behind the proposed novel design. I evaluate the proposed index structure in Section 5.5.

5.1.4 Background and Preliminaries

In this section, I briefly describe the key concepts underlying LSH, relying primarily on the terminology and formulations in E2LSH[1] and C2LSH [36], and then formally describe the problem is solved in this chapter.

5.1.5 Key Concepts

Hash Functions. A hash function family H is said to be (r, c, P_1, P_2)-sensitive if it satisfies all the following conditions for any two points x and y in a data set $D \subset \mathbb{R}^d$:

- if $|x - y| \leq r$, then $Pr[h(x) = h(y)] \geq P_1$, and
- if $|x - y| > cr$, then $Pr[h(x) = h(y)] \leq P_2$

Here, c is an approximation ratio, P_1 and P_2 are probabilities, and in order for the definition to work, $c > 1$ and $P_1 > P_2$. The above definition states that the two points x and y are hashed to the same bucket with a very high probability $\geq P_1$ if they are close to each other (i.e. the distance between the two points is less than or equal to r), and if they are not close to each other (i.e. the distance between the two points is greater than cr), then they will be hashed to the same bucket with a low probability $\leq P_2$.

In the original LSH scheme for Euclidean distance, each hash function is defined as $h_{a,b}(v) = \frac{a.v+b}{w}$, where a is a d-dimensional random vector with entries chosen independently from the standard normal distribution and b is a real number chosen uniformly from $[0, w)$, such that w is the width of the hash bucket [29]. This leads to

the following collision probability function [59]:

$$P(r) = \int_0^w \frac{1}{r}\frac{2}{\sqrt{2\pi}} e^{\frac{-t^2}{2r^2}}(1 - \frac{t}{w})dt. \tag{5.1}$$

Note that the collision probability is governed by the width, w, of the hash bucket: if the size is chosen to be much larger than the query radius, then there can be a lot of candidates generated. If the size is chosen to be much smaller than the query radius, then there can be potentially several misses.

Controlling Accuracy through Layer Structure. To control false positives, LSH concatenates multiple hash functions to create a compound hash function for a single hash table. For k hash functions $h_1(x), h_2(x), ..., h_k(x)$, it creates a compound hash function $g(x) = (h_1(x), h_2(x), ..., h_k(x))$. For a data point x, the answer of this compound hash function $g(x)$ is used as the bucket id for the given hash table (which is made up of k hash functions).

On the other hand, as the number of hash functions increases, the recall of the entire index drops. In order to increase the recall, LSH creates multiple hash layers each consisting of these k hash functions. Let m be the total number of hash layers in the LSH index and let us assume that the user is interested in objects within distance r. The probability that points x and y fail to collide in all m hash layers is $(1 - P_1^k)^m$. The probability that the two points x and y collide in at least one hash layer (which is the same as the expected recall for the index) is

$$1 - \delta \geq 1 - (1 - P_1^k)^m, \tag{5.2}$$

where δ is a user-provided input denoting the expected miss probability. In other words, the number, k, of hash functions per layer and the number, m, of layers can together be used to control the accuracy of the index structure.

5.1.6 C2LSH Method

The C2LSH method [36] relies on the concept of *collision counting*. [36] theoretically shows that two close points x and y (i.e., $|x - y| \leq r$) collide in at least l layers with a probability $1 - \delta$, when the total number, m, of layers in the hash structure is set to be

$$m = \lceil \frac{\ln(\frac{1}{\delta})}{2(p_1 - p_2)^2}(1 + z)^2 \rceil, \tag{5.3}$$

where $z = \sqrt{\ln(\frac{2}{\beta})/\ln(\frac{1}{\delta})}$, and β is the percentage of points whose distance with a query point is greater than cr[3]. Given this, the authors suggest that only those points that collide above a collision count threshold, l, with the query point are considered as candidate points. The collision count threshold is defined as a ratio of the total number of layers; more specifically, $l = \lceil \alpha \times m \rceil$, where the collision threshold percentage, α, is

$$\alpha = \frac{zp_1 + p_2}{1 + z}. \tag{5.4}$$

Note that, in C2LSH, since there is only one hash function per hash layer, the total number of hash functions are equal to the total number of available hash layers. In further discussions, the terms hash functions and hash layers are used interchangeably.

5.2 Problem Specification

Given a multidimensional database $\mathcal{D}$ that consists of points that belong to a bounded multidimensional space $\mathcal{S}$, a single query representing a point in $\mathcal{S}$ is called a **point query**.

Definition 5.2.1 (Positive Point Query) *A data point x* satisfies *a positive point query, q_i (and its corresponding radius r_{q_i}), if $dist(x, q_i) \leq r_{q_i}$.*

[3]C2LSH sets $\beta = \frac{100}{n}$, where n is the cardinality of the dataset.

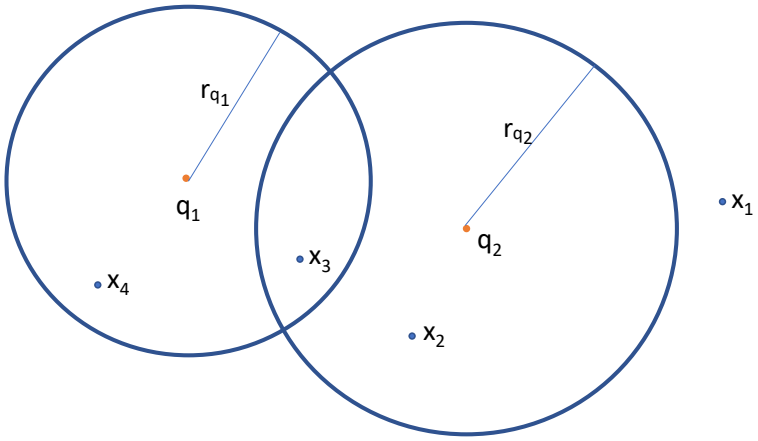


Figure 5.1: Illustration of Two Point Queries (q_1 and q_2)

Similarly,

Definition 5.2.2 (Negative Point Query) *A data point x* satisfies *a negative point query, q_i (and its corresponding radius r_{q_i}), if $dist(x, q_i) > r_{q_i}$.*

A set of point queries is referred to as a **set query** Q. Figure 5.1 illustrates two point queries and their respective radiuses. Let us consider q_1 is a positive point query and q_2 is a negative point query. In this example, data points x_3 and x_4 *satisfy* q_1, while data points x_1 and x_4 satisfy q_2. Data point x_4 satisfies both point queries q_1 and q_2.

Definition 5.2.3 (Set Query Satisfaction) *Consider a set query Q that consists of s point queries. Each ith query in Q is also represented by its corresponding radius r_{q_i} and a flag f_{q_i}. The flag $f_{q_i} \in (0, 1)$ denotes whether a point query is a positive point query ($f_{q_i} = 1$) or a negative point query ($f_{q_i} = 1$). A point x* satisfies *q_i,*

- *if $f_{q_i} = 1$ and $dist(x, q_i) \leq r_{q_i}$, or*
- *if $f_{q_i} = 0$ and $dist(x, q_i) > r_{q_i}$.*

A data point x satisfies *a given set query Q if x* satisfies *at least θ point queries (where $1 \leq \theta \leq s$) of Q.*

In Figure 5.1, if the set query Q consisted of q_1 and q_2 ($\theta = 1$, $f_{q_1} = 1$, and $f_{q_2} = 0$), then data points x_1, x_3, and x_4 satisfy Q, whereas x_2 satisfies no query. If $\theta = 2$, then only the data point x_4 satisfies Q.

Definition 5.2.4 (Set Query Guarantee) *Every data point x that satisfies at least θ point queries in a given set query Q, should be reported with a probability of at least $1 - \delta$ probability (where δ is a user-specified probability).*

In particular, in this work, two key questions are answered:

- Given a budget of total hash functions, how are the hash functions distributed to PSLSH$_E$ and PSLSH$_H$ such that the number of candidates are reduced while satisfying the user-input guarantee, $1 - \delta$, for the entire set query?

- As noted earlier, the precise distance computation is an expensive process. Once the candidates from individual queries are computed, the next step is to remove those points that do not satisfy at least θ queries. In the recently introduced *collision counting* approach, only the points that collide[4] are considered as candidates. Since PSLSH$_H$ is built on top of PSLSH$_E$, the input to PSLSH$_H$ is the output of PSLSH$_E$ (which can potentially contain misses and false positives). The existing collision threshold [36] does not consider the scenario when the input data potentially contains misses and false positives. The second question that is answered is: how can the collision counting approach be extended such that it accounts for the input data to contain misses and false positives?

[4]Two points are said to collide if they hash to the same set of hash functions in multiple layers

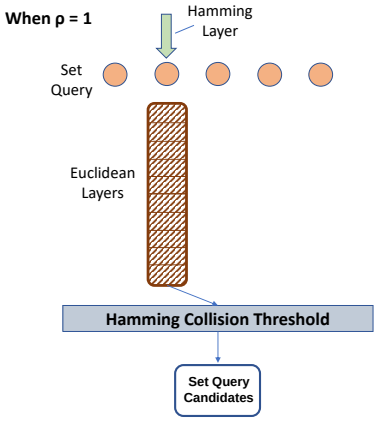
5.3 Naive Solution

The naive solution to this problem consists of three steps:

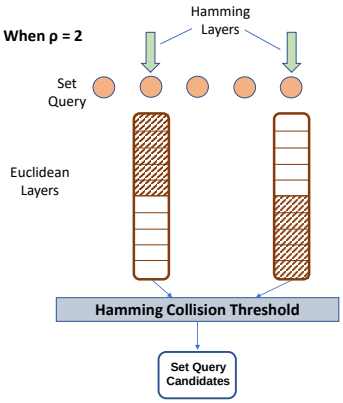
- **Step 1:** for each point query q_i in the set query Q, determine a success guarantee $1 - \delta$,
- **Step 2:** using the standard LSH index based on the above success guarantee, find the candidates for each point query q_i in Q,
- **Step 3:** for each candidate point x, find all the point queries satisfied by x (as explained in Section 5.2). Let us consider a list V_Q that contains the flags of each query in the given set query (i.e. the ith value of V_Q (V_Q^i) would be equal to the flag value of query q_i). Let us also consider that for every point x in the database, we have a list V_x where the ith value of V_x (V_x^i) $= 1$ if $dist(x, q_i) \leq r_i$, or $V_x^i = 0$ if $dist(x, q_i) > r_i$. In order for point x to be considered in the final result set of Q, x has to satisfy at least θ query points in the set query Q. Note that, this is equivalent to the following: point x is to be considered in the final results if the Hamming distance between V_Q and V_x is $\leq s - \theta$.

There are two main problems with the naive approach: a) The user is unable to input an overall guarantee for the set query, and instead has to input a guarantee on individual point queries, b) Underestimating the guarantee on individual point queries can lead to overall misses, while overestimating the guarantee on individual point queries can lead to redundant and wasteful work!

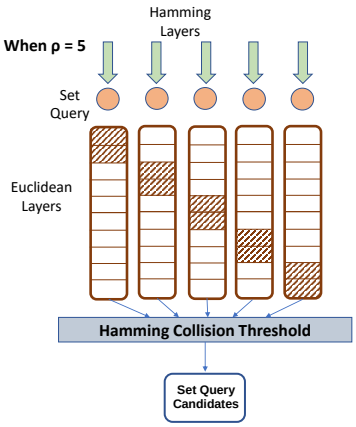
There is additional computation necessary for the Hamming distance in order to find the final results. By leveraging this need for the Hamming distance computation, I introduce an LSH index structure based on the Hamming distance, in order to give a guarantee on the set query.



(a) When $\rho = 1$



(b) When $\rho = 2$



(c) When $\rho = 5$

Figure 5.2: Architecture of PSLSH (for scenarios with different Splitting Factors (ρ))

5.4 Point Set LSH (PSLSH)

In this section, I describe the proposed index structure, *Point Set LSH* (PSLSH). In order to provide guarantee for the entire set query, PSLSH introduces an additional index structure based on the Hamming distance on top of the existing LSH index structure. In PSLSH, the goal is to find the points in a database that satisfy at least θ queries from a given set query Q.

5.4.1 Design of PSLSH

In this section, I describe the design and the intuition behind the design of PSLSH. The design of PSLSH can be summarized as follows:

- Similar to the original LSH [44], PSLSH is built for a user-specified m number of layers.

- In the naive solution, once the candidates for each individual point query are found, the false positives need to be removed (for each individual point query) by fetching the data from the secondary storage, which is an expensive operation. Instead, in PSLSH, an additional index structure based on the Hamming distance (called PSLSH$_H$) is created in order to provide a guarantee on the entire set query, and in process, avoid removing the false positives after the candidates for each query are found. Given a set query (that consists of s point queries), PSLSH randomly chooses ρ point queries to be processed (which is similar to randomly choosing a dimension in the Hamming space [44]). Each of these ρ point queries are further processed using LSH-indexes based on the Euclidean distance (called PSLSH$_E$). Given a total budget of hash layers (which in Figure 5.2 = 10), the goal is to appropriately allocate these hash layers to PSLSH$_E$ and PSLSH$_H$. In other words, we want to split the m total layers into ρ "sub-

indexes" (hence, ρ is called the *Splitting Factor*), such that each sub-index will process an individual query using $\frac{m}{\rho}$ layers. PSLSH$_E$ generates candidates from individual point queries which are further pruned by using the novel modified Hamming Collision Threshold (further explained in Section 5.4.2). Figure 5.2 (a) shows the scenario where $\rho = 1$, i.e. 1 point query is randomly chosen to be processed and there is only one sub-index. All the m (which in this Figure is equal to 10) layers are assigned to this sub-index. When the splitting factor is 2 (Figure 5.2 (b)), there are two point queries to be processed on two sub-indexes (where each of the sub-indexes have 5 layers assigned to it). The following discussion is based on the observation that as the number of layers assigned to PSLSH$_E$ decrease, the accuracy of PSLSH$_E$ decreases [36]. Similarly, as the number of layers assigned to PSLSH$_H$ increases, the accuracy of PSLSH$_H$ increases.

  - When $\rho = \mathbf{1}$, there will be m layers assigned to PSLSH$_E$, whereas only 1 layer assigned to PSLSH$_H$. In this option, PSLSH$_E$ will have high accuracy, but the accuracy of PSLSH$_H$ will be low.
  - On the contrary, when $\rho = \mathbf{m}$, PSLSH$_E$ will only have 1 layer per sub-index resulting in low accuracy, and PSLSH$_H$ will have m layers resulting in high accuracy.

  Our goal is to choose a ρ such that the resultant accuracy of PSLSH is high.

- Since false positives are not removed from the candidates generated from PSLSH$_E$, one has to appropriately modify the collision threshold for PSLSH$_H$. Also, since the input to PSLSH$_H$ is the output of PSLSH$_E$, it can further contain false negatives. The collision threshold for PSLSH$_H$ is based on ρ, and each ρ has a different positive and negative collision probabilities for the Hamming distance.

I present the theoretical analysis behind the modified positive and negative collision probabilities for the Hamming distance in Section 5.4.2. By using this modified collision threshold for the Hamming distance, the number of candidates needed to find the final results are further reduced, thus improving the execution time of the given set query.

- If the total number of layers (m) are not completely divisible by ρ, then there will be an uneven distribution of layers among the sub-indexes. Uneven distribution of layers can lead to unpredictable behavior of the individual sub-indexes. In order to avoid this, I present the distribution strategy of PSLSH in detail in Section 5.4.4.

5.4.2 Theoretical Analysis of the Positive and Negative Colliding Probabilities

The naive way to find the results for the set query would be to calculate the distance of each candidate with each query s in Q, and then remove the points that do not satisfy θ of the s point queries in the set query Q. As noted earlier, LSH was originally designed for the Hamming distance metric. In PSLSH, I propose to add a layer of LSH based on the Hamming distance (PSLSH_H) in order to give a guarantee on the entire set query. But this process is not trivial, since in the original formulation [44], the input does not consider the possibility of having false positives and false negatives. Note that, once the candidates for each of the ρ point queries are found, the exact Euclidean distance of each candidate point with its corresponding query (in order to remove the false positives) are not computed. These candidates can also have misses resulting in false negatives. These candidates are an input to the PSLSH_H. Hence, the original LSH formulation for the Hamming distance needs to be extended such that it accounts for the input data to possibly contain false positives and false negatives.

First, I briefly describe the original LSH formulation as described in [44]. Let us denote the probability that two close points in the Hamming space are hashed to the same bucket as P_{1_H}, and the probability that two distant objects are hashed to the same bucket as P_{2_H}. For a given Hamming radius r_H, a d_H-dimensional Hamming space, and an approximation factor c_H, $P_{1_H} = 1 - \frac{r_H}{d_H}$ and $P_{2_H} = 1 - \frac{c_H r_H}{d_H}$. As described earlier, the number of Hamming dimensions is equal to the number of queries in the set query (i.e. s), and since our goal is to satisfy at least θ queries, the Hamming radius $r_H = s - \theta$.

The input to PSLSH$_H$ (which are the candidate sets generated by PSLSH$_E$ for each of the ρ queries) can contain false negatives (i.e. V_x^i should have been equal to 1 instead of 0) or false positives (i.e. V_x^i should have been equal to 0 instead of 1). Let us denote the error probability of a false negative generated by PSLSH$_E$ as δ_{FN_E}, and the error probability that a false positive is generated by PSLSH$_E$ as δ_{FP_E}.
There are four possible cases that need to be taken into consideration, which are described as follows:

- agr_{PRIN}**:** The case where V_Q^i and V_x^i *should be* returned with the same value, i.e. they agree in principle,

- dis_{PRIN}**:** The case where V_Q^i and V_x^i *should be* returned with different values, i.e. they disagree in principle,

- agr_{PRAC}**:** The case where V_Q^i and V_x^i *are* returned with the same value, i.e. they agree in practice,

- dis_{PRAC}**:** The case where V_Q^i and V_x^i *are* returned with different values, i.e. they disagree in practice.

In order to find P_{1_H} and P_{2_H} while accounting for the false positives and false negatives, I consider the following scenarios:

- **Scenario 1:** V_Q^i and V_x^i agree in practice and also agree in principle (i.e. $P(agr_{PRAC} \cap agr_{PRIN})$),
- **Scenario 2:** V_Q^i and V_x^i agree in practice but disagree in principle (i.e. $P(agr_{PRAC} \cap dis_{PRIN})$),
- **Scenario 3:** V_Q^i and V_x^i disagree in practice but agree in principle (i.e. $P(dis_{PRAC} \cap agr_{PRIN})$), and
- **Scenario 4:** V_Q^i and V_x^i disagree in practice and also disagree in principle (i.e. $P(dis_{PRAC} \cap dis_{PRIN})$).

Set Query that Consists of Only Positive Point Queries

Let us denote the probability that two close points are hashed to the same bucket (while accounting for false positives and false negatives) as P'_{1_H}. As described in Section 5.2, a set query can contain both positive and negative point queries. In order to explain better, I first present the analysis where a set query contains only positive point queries, and then extend it to include both positive and negative point queries.

$$\begin{aligned} P'_{1_H} = {} & P(agr_{PRAC} \cap agr_{PRIN}) + P(agr_{PRAC} \cap dis_{PRIN}) \\ & + P(dis_{PRAC} \cap agr_{PRIN}) + P(dis_{PRAC} \cap dis_{PRIN}) \end{aligned} \tag{5.5}$$

$$
\begin{aligned}
P'_{1_H} = & P(agr_{PRAC}|agr_{PRIN}).P(agr_{PRIN}) \\
& + P(agr_{PRAC}|dis_{PRIN}).P(dis_{PRIN}) \\
& + P(dis_{PRAC}|agr_{PRIN}).P(agr_{PRIN}) \\
& + P(dis_{PRAC}|dis_{PRIN}).P(dis_{PRIN})
\end{aligned} \tag{5.6}
$$

$$
\begin{aligned}
P'_{1_H} = & P(agr_{PRIN}) \left[P(agr_{PRAC}|agr_{PRIN}) + P(dis_{PRAC}|agr_{PRIN})\right] \\
& + P(dis_{PRIN}) \left[P(agr_{PRAC}|dis_{PRIN}) + P(dis_{PRAC}|dis_{PRIN})\right]
\end{aligned} \tag{5.7}
$$

In the original LSH formulation, $P(agr_{PRAC}|agr_{PRIN}) + P(dis_{PRAC}|agr_{PRIN}) = 1$ and $P(agr_{PRAC}|dis_{PRIN}) + P(dis_{PRAC}|dis_{PRIN}) = 0$ because it assumes that there are no errors in the input data. Since the input data can have false negatives and false positives, $P(agr_{PRAC}|agr_{PRIN}) + P(dis_{PRAC}|agr_{PRIN}) < 1$ and $P(agr_{PRAC}|dis_{PRIN}) + P(dis_{PRAC}|dis_{PRIN}) > 0$. Let us denote the number of true positives in the candidate set generated by PSLSH$_E$ (for a single query q_i) as TP, the number of false positives generated as FP, the number of true negatives as TN, and the number of false negatives as FN.

- $P(agr_{PRAC}|agr_{PRIN})$, the probability of true positives occurring, $= \frac{TP}{TP+FN} = 1 - \delta_{FN_E}$, and
- $P(agr_{PRAC}|dis_{PRIN})$, the probability of false positives occurring, $= \frac{FP}{FP+TN} = \delta_{FP_E}$

Note that, from [44], we know that $P(agr_{PRIN}) \leq 1 - \frac{r_H}{s}$ and $P(dis_{PRIN}) > \frac{r_H}{s}$. Due to the opposing inequalities, the two terms in Equation 5.7 cannot be simply added. Since the Hamming Radius (r_H) is a discrete variable, each of the probabilities of the possible values of Hamming Radius can be summed up to get the following:

$$P'_{1_H} = \left[\sum_{h=0}^{r_H-1} P(h|h \leq r_H). \left(1 - \frac{h}{s}\right). (1 - \delta_{FN_E}) \right] + \left[\sum_{h=0}^{r_H-1} P(h|h \leq r_H). \left(\frac{h}{s}\right). (\delta_{FP_E}) \right] \tag{5.8}$$

where $P(h|h \leq r_H) = \frac{1}{r_H}$.

$$P'_{1_H} = \frac{1}{r_H}. (1 - \delta_{FN_E}) \left[\sum_{h=0}^{r_H-1} \left(1 - \frac{h}{s}\right) \right] + \frac{1}{r_H}. (\delta_{FP_E}) \left[\sum_{h=0}^{r_H-1} \left(\frac{h}{s}\right) \right] \tag{5.9}$$

Similarly, for the calculation of the negative collision probability (P'_{2_H}) and an approximation ratio $c_H > 1$,

$$P'_{2_H} = \left[\sum_{h=c_H r_H}^{s} P(h|h > c_H r_H). \left(1 - \frac{h}{s}\right). (1 - \delta_{FN_E}) \right] + \left[\sum_{h=c_H r_H}^{s} P(h|h > c_H r_H). \left(\frac{h}{s}\right). (\delta_{FP_E}) \right] \tag{5.10}$$

where $P(h|h > c_H r_H) = \frac{1}{s - c_H r_H + 1}$.

$$P'_{2_H} = \frac{1}{s - c_H r_H + 1}. (1 - \delta_{FN_E}) \left[\sum_{h=c_H r_H}^{s} \left(1 - \frac{h}{s}\right) \right] + \frac{1}{s - c_H r_H + 1}. (\delta_{FP_E}) \left[\sum_{h=c_H r_H}^{s} \left(\frac{h}{s}\right) \right] \tag{5.11}$$

Set Queries that Contain Negative Point Queries

Let us now extend P'_{1_H} and P'_{2_H} to consider a set query that contains both positive (i.e. $V_Q^i = 1$) and negative point queries (i.e. $V_Q^i = 0$). V_Q is the list that contains

flags that denotes whether a point query in the set query Q is a positive or a negative query. For a negative point query q_i, as described in Section 5.2, a point query x satisfies q_i if $dist(x, q_i) > r_{q_i}$. Thus, for a candidate set generated by PSLSH$_E$ of a negative query q_i, a point x is a false positive if it is included in the candidate set of q_i but $dist(x, q_i) \leq r_{q_i}$. Thus, P'_{1_H} and P'_{2_H} for a set query with both positive and negative point queries:

$$\begin{aligned} P'_{1_H} = P(V_Q^i = 1) \Bigg[& \frac{1}{r_H} . (1 - \delta_{FN_E}) \left[\sum_{h=0}^{r_H - 1} \left(1 - \frac{h}{s} \right) \right] \\ & + \frac{1}{r_H} . (\delta_{FP_E}) \left[\sum_{h=0}^{r_H - 1} \left(\frac{h}{s} \right) \right] \Bigg] \\ + P(V_Q^i = 0) \Bigg[& \frac{1}{r_H} . (\delta_{FP_E}) \left[\sum_{h=0}^{r_H - 1} \left(1 - \frac{h}{s} \right) \right] \\ & + \frac{1}{r_H} . (1 - \delta_{FN_E}) \left[\sum_{h=0}^{r_H - 1} \left(\frac{h}{s} \right) \right] \Bigg] \end{aligned} \tag{5.12}$$

$$\begin{aligned} P'_{2_H} = P(V_Q^i = 1) \Bigg[& \frac{1}{s - c_H r_H + 1} . (1 - \delta_{FN_E}) \left[\sum_{h=c_H r_H}^{s} \left(1 - \frac{h}{s} \right) \right] \\ & + \frac{1}{s - c_H r_H + 1} . (\delta_{FP_E}) \left[\sum_{h=c_H r_H}^{s} \left(\frac{h}{s} \right) \right] \Bigg] \\ + P(V_Q^i = 0) \Bigg[& \frac{1}{s - c_H r_H + 1} . (\delta_{FP_E}) \left[\sum_{h=c_H r_H}^{s} \left(1 - \frac{h}{s} \right) \right] \\ & + \frac{1}{s - c_H r_H + 1} . (1 - \delta_{FN_E}) \left[\sum_{h=c_H r_H}^{s} \left(\frac{h}{s} \right) \right] \Bigg] \end{aligned} \tag{5.13}$$

Calculation of the Hamming Collision Threshold

By using Equations 5.12 and 5.13, the collision counting method (as proposed in [36] and described in Section 5.1.4) for PSLSH$_H$ can now be extended in order to handle the scenario where false positives and false negatives are introduced to the input data.

For a user-provided failure probability threshold δ_H, only the candidates that collide with V_Q at least l_H times are considered (where $l_H = \lceil \rho \times \alpha_H \rceil$, $\alpha_H = \frac{z_H P'_{1_H} + P'_{2_H}}{1+z_H}$, $z_H = \sqrt{\ln(\frac{2}{\beta_H}) / \ln(\frac{1}{\delta_H})}$, β_H is the percentage of points whose distance with the set query is greater than $c_H r_H$, and δ_H is the error percentage for the set query). By using this modified collision threshold for PSLSH$_H$, PSLSH is able to effectively reduce the final candidates and the overall set query processing time.

5.4.3 Finding the Optimal Splitting Factor

The idea behind PSLSH is that one can process a subset of queries from the set query due to the locality sensitive nature of the query candidates in the set query. In order to do that, the number of layers, m, of PSLSH are split into ρ sub-indexes, such that each sub-index has (approximately) $\frac{m}{\rho}$ layers. Hence, ρ is called the *Splitting Factor*. Then each of the ρ queries are processed on the corresponding ρth sub-index. As explained in Section 5.4.1, for a splitting factor ρ, the number of layers assigned to PSLSH$_E$ are (approximately) $\frac{m}{\rho}$, while the number of layers assigned to PSLSH$_H$ are ρ. Intuitively, for a very small ρ (e.g. $\rho = 2$), the number of candidates generated by PSLSH$_H$ are high because the number of layers assigned to PSLSH$_H$ are low, and hence the collision threshold is not able to effectively prune the false positives, which increases the total set query execution time. On the contrary, when ρ is very large (e.g. ρ close to m), then the number of layers assigned to PSLSH$_E$ are low, which causes PSLSH$_E$ to generate excessive false positives. Also, as ρ increases, more queries have to be processed which increases the time taken by PSLSH to identify the candidates by accessing the hash layers. Hence, the goal is to find the optimal value of the *Splitting Factor* ρ such that the total number of candidates generated by PSLSH$_E$ and PSLSH$_H$ are minimized, which in turn minimizes the set query execution time. Before the optimization function is presented, the breakdown of the

set query execution time is explained. Let us denote the set query execution time as $Time_Q$. The set query execution time is dominated by two main sub-costs:

- **Index Access cost:** This cost includes the time taken to access the index structure in order to find the candidates for all the queries. This cost is denoted by $Time_{IA}$. $Time_{IA}$ is proportional to the number of queries executed (i.e. the total number of sub-indexes). As the number of queries processed increases, the number of index accesses also increase, which increases the $Time_{IA}$, and vice-versa. Assuming λ_{IA} is an index access cost multiplier, we have $Time_{IA} = \lambda_{IA} \times f(\rho)$. As explained further in Section 5.4.5, λ_{IA} is a function of the size of the input dataset.

- **IO cost:** This cost (denoted by $Time_{IO}$) includes the time needed to bring each candidate into the memory from the secondary storage, compute the exact distance between the candidate and each query in the set query, and compute whether the candidate satisfies at least θ queries of the given set query. The number of candidates generated are related to the number of queries processed, but it is not a straightforward conclusion as further explained in Section 5.4.3. Assuming λ_{IO} is an IO cost multiplier, we have $Time_{IO} = \lambda_{IO} \times g(\rho)$.

  The goal is to find ρ such that $Time_Q$ (where $Time_Q = Time_{IA} + Time_{IO}$) is minimized, i.e.

$$\begin{aligned} &\text{minimize} && (\lambda_{IA} \times f(\rho)) + (\lambda_{IO} \times g(\rho)) \\ &\text{subject to} && 1 \leq \rho \leq m. \end{aligned}$$

Finding the Relationship between ρ and $Time_{IO}$

$Time_{IO}$ is proportional to the number of candidates generated by PSLSH. Each candidate point has to be brought from the specific location on the secondary storage into the main memory for further computation. Intuitively, as ρ increases (i.e. more hash functions are assigned to PSLSH$_H$ than PSLSH$_E$), more candidates would be generated by PSLSH$_E$ but less candidates would be generated by PSLSH$_E$. The main challenge then would be to choose the splitting factor (ρ) such that the least total number of candidates are generated by both PSLSH$_E$ than PSLSH$_H$.

However, the above hypothesis does not hold true. The total number of candidates generated by PSLSH are dependent on the collision thresholds used by PSLSH$_E$ and PSLSH$_H$ (i.e. l_E and l_H respectively). As explained in Section 5.1.6 and Section 5.4.2, $l_E = \lceil m_E \times \alpha_E \rceil$ and $l_H = \lceil \rho \times \alpha_H \rceil$. In the following discussion, for the sake of simplicity, we refer to l_E when the ceiling function is used (i.e. $l_E = \lceil m_E \times \alpha_E \rceil$), and refer to l'_E when the ceiling function is not used (i.e. $l'_E = m_E \times \alpha_E$). Thus, we have $l_E \geq l'_E$. The goal is to estimate $Time_{IO}$ for different values of ρ. In order to find this relationship, the number of candidates generated by PSLSH for different values of ρ needs to be estimated.

Let us consider two splitting factors, ρ_i and ρ_j, used by PSLSH$_{E_i}$ and PSLSH$_{E_j}$ respectively. Let us denote their corresponding collision thresholds (as explained in Section 5.1.6) as l_{E_i} and l_{E_j} respectively. Let us also denote the number of candidates generated by PSLSH$_{E_i}$ as $cand(PSLSH_{E_i})$, and by PSLSH$_{E_j}$ as $cand(PSLSH_{E_j})$. Let us consider the similar notations (l_{H_i}, l_{H_j}, $cand(PSLSH_{H_i})$, and $cand(PSLSH_{H_j})$) for PSLSH$_H$.

Observation 5.4.1 *If* $l_{E_i} - l'_{E_i} > l_{E_j} - l'_{E_j}$, *then* $cand(PSLSH_{E_i}) < cand(PSLSH_{E_j})$. *The same holds true for* $PSLSH_H$. *If* $l_{H_i} - l'_{H_i} > l_{H_j} - l'_{H_j}$, *then* $cand(PSLSH_{H_i}) < cand(PSLSH_{H_j})$.

In the collision counting approach (defined in [36]), the collision threshold has to be an integer since it counts the number of times a data point collides with a given query point. When the ceiling function is used on l'_E, the collision counting approach uses a more constrained collision threshold l_E than l'_E. As the difference between $l_E - l'_E$ increases, this over-constraint increases misses (while still satisfying the user-input error guarantee) but also reduces false positives. Thus, the value of $l_E - l'_E$ needs to be taken into consideration when predicting the number of candidates generated by PSLSH for a particular splitting factor. As explained in Section 5.4.1, PSLSH splits the total number of layers in ρ sub-indexes. In some cases, each ρ sub-index have exactly $\frac{m}{\rho}$ layers, but in some cases, different sub-indexes have different number of layers (as presented in Algorithm 9). Let us consider m_o are the number of layers assigned to the oth sub-index (and l_{E_o} is the collision threshold for m_o layers). In order to predict the number of candidates generated by PSLSH_E, we define

$$Diff_{E_\rho} = 1 - \frac{\sum_{o=1}^{\rho} l_{E_o} - l'_{E_o}}{\rho} \tag{5.14}$$

where for a given ρ, $Diff_{E_\rho}$ defines the average of the difference between l_E and l'_E over the ρ sub-indexes. Similarly, for PSLSH_H, we define

$$Diff_{H_\rho} = 1 - (l_{H_\rho} - l'_{H_\rho}) \tag{5.15}$$

Note that, since for any given ρ, there is only one collision threshold for PSLSH_H (and hence an average is not needed unlike PSLSH_E).

Observation 5.4.2 *For two splitting factors* i *and* j, *the total candidates generated by* $PSLSH_{\rho=i}$ *and* $PSLSH_{\rho=j}$ *depend on the values of* $Diff_{E_{\rho=i}}$, $Diff_{H_{\rho=i}}$, $Diff_{E_{\rho=j}}$, *and* $Diff_{H_{\rho=j}}$.

The total number candidates generated by PSLSH is dependent on the total number of candidates generated by PSLSH$_E$ and PSLSH$_H$. If $Diff_{E_\rho}$ is close to 1, then PSLSH$_E$ will produce less number of candidates than when $Diff_{E_\rho}$ is close to 0. Similarly, when $Diff_{H_\rho}$ is close to 1, then PSLSH$_H$ will produce less number of candidates than when $Diff_{H_\rho}$ is close to 0. Thus, PSLSH will generate less candidates when both $Diff_{E_\rho}$ and $Diff_{H_\rho}$ are close to 1. But it will not necessarily generate less candidates when only $Diff_{E_\rho}$ or $Diff_{H_\rho}$ are close to 1, and the other is close to 0. Thus, scenarios when there is a large difference between $Diff_{E_\rho}$ and $Diff_{H_\rho}$ need to be penalized. Hence, we define another metric, ϕ_{Diff_ρ}, which takes the harmonic mean between $Diff_{E_\rho}$ and $Diff_{H_\rho}$ (and thus further penalizes the scenarios where there is a large difference between $Diff_{E_\rho}$ and $Diff_{H_\rho}$) in order to predict the number of candidates PSLSH generates,

$$\phi_{Diff_\rho} = \frac{2}{\frac{1}{Diff_{E_\rho}} + \frac{1}{Diff_{H_\rho}}} \tag{5.16}$$

Note that, this harmonic mean is also called the F-score or the F-measure. In Section 5.5.4, the effectiveness of the proposed ϕ_{Diff_ρ} is evaluated.

Mapping ϕ_{Diff_ρ} to the Number of Candidates. By simply using Equation 5.16, the number of candidates generated by a particular splitting factor cannot be estimated. In order to estimate the number of candidates generated by different splitting factors, there has to be a mapping between at least one ϕ_{Diff_ρ} and the number of candidates generated by that particular ρ. In order to find this mapping, during runtime, PSLSH finds the number of candidates generated for $\rho = 1$ by executing 1 query from the given set query. Note that, this is an inexpensive step because there is no need to build any additional layers or index structures and there is no need for a disk access either (since we only need to find the number of candidates generated and we do not need to remove any false positives - which is usually the most expensive

part of the query execution time). Let us denote the number of candidates generated by PSLSH for ρ as $cand(PSLSH_{\rho})$. The number of candidates for $\rho > 1$ can then by estimated by considering the change between the W_{Diff} values for two adjacent splitting factors.

$$\forall_{i=2}^{m} cand(PSLSH_{\rho=i}) = \frac{\phi_{Diff_{\rho=i}}}{\phi_{Diff_{\rho=i-1}}} \times cand(PSLSH_{\rho=i-1}) \tag{5.17}$$

In Section 5.5.4, the accuracy of the proposed method of estimating the number of candidates for different splitting factors is evaluated. Once the number of candidates for different splitting factors are estimated, $Time_{IO}$ can then be estimated for different splitting factors. Note that, the values of ϕ_{Diff} for different splitting factors can be pre-calculated for a given dataset since the collision thresholds for PSLSH$_E$ and PSLSH$_H$ can be pre-calculated for a given dataset. As further explained in Section 5.5, this time to calculate candidates for $\rho = 1$ is included in the overall query processing time. Once $Time_{IA}$ and $Time_{IO}$ can be estimated, the splitting factor that minimizes the overall query execution time can be predicted.

5.4.4 Distribution of Layers to ρ Sub-Indexes

In PSLSH, the m layers of the LSH index are split into ρ sub-indexes. If m is not completely divisible by ρ, then there can be an uneven distribution of these layers to the individual sub-indexes. One naive solution would be to assign $\lfloor \frac{m}{\rho} \rfloor$ layers to each of the $\rho - 1$ sub-indexes, and the remaining layers $(m - (\rho - 1).\lfloor \frac{m}{\rho} \rfloor)$ are assigned to the ρth index. This can lead to unpredictable behavior of δ_{FN_E} and δ_{FP_E} as the last sub-index can have significantly more layers to process the sub-queries resulting in a smaller δ_{FN_E} and δ_{FP_E} relative to the other layers. In order to avoid this, a simple layer distribution strategy is presented in Algorithm 9. The remaining layers are further distributed among the ρ indexes (Lines 8-11) such that the maximum

Algorithm 9 Distribution of Layers to Sub-indexes

1: **Input:**

- Total number of layers (m)
- Splitting Factor (ρ)

2: **Output:** Mapping of layers to sub-indexes (i.e. $\forall_{i \in \rho} m_\rho^i$)

3: **procedure** MAPLAYERS

4: **for** sub-index $i = 1; i \leq \rho; i++$ **do**

5: **if** $m \mod \rho == 0$ **then**

6: $m_\rho^i = \frac{m}{\rho}$

7: **else**

8: **if** $i \leq m \mod \rho$ **then**

9: $m_\rho^i = \lfloor \frac{m}{\rho} \rfloor + 1$

10: **else**

11: $m_\rho^i = \lfloor \frac{m}{\rho} \rfloor$

12: **end if**

13: **end if**

14: **end for**

15: **end procedure**

difference in the number of layers mapped to any sub-index is always 1.

5.4.5 Space and Query Time Complexities

As noted earlier, PSLSH uses C2LSH to do the LSH query processing. The total space complexity needed for PSLSH is same as C2LSH, since the total number of layers needed for PSLSH is m layers. PSLSH accepts m as a user-input, and in general, $m \ll n$ [36, 29]. C2LSH requires space to store the m layers and the dataset.

For a dataset with d dimensions and n data points, the space consumption for the data is $O(dn)$. In each hash layer, there are n data point IDs. Hence, the total space consumption for PSLSH is $O(dn + mn)$.

PSLSH has three major costs: a) Locating the m buckets for each of the ρ queries. Assuming the worst case, where $\rho = m$, then the cost of locating m buckets for d-dimensional objects is $O(m^2 d)$. b) The second cost includes the cost of collision counting for PSLSH$_E$ and PSLSH$_H$. Collision counting has to be done for at most n data points over m layers for ρ queries (where $\rho = m$ in the worst case), and collision counting for PSLSH$_H$ can be done while doing the collision counting for PSLSH$_H$. Thus, the cost for collision counting is $O(m^2 n)$. c) The third cost includes the distance computation necessary to remove false positives to get the results for the set query execution. Let us consider $|cand_Q|$ is the number of candidates that PSLSH generates. The cost of computing the distance of $|cand_Q|$ with s queries is $O(|cand_Q|ds)$. Note that, if $\beta_H = \frac{100}{n}$ and the number of points that do not satisfy at least $c_H\theta$ queries in Q is denoted by $c_H r_H groundTruth(Q)$, then $|cand_Q| < |c_H r_H groundTruth(Q)| + 100$. Thus, the total query time cost is $O(m^2 d + m^2 n + |cand_Q|ds)$.

5.4.6 C2LSH vs. QALSH

PSLSH uses the collision counting method of C2LSH for performing query processing. In [43] (called QALSH), the authors build upon the collision counting method in order to create "query-aware" hash functions (where the buckets of the hash functions are created based on the input query point). In QALSH, the authors show that by building query-aware hash functions, they are able to improve the top-k query processing time of C2LSH. My work is focused on the (r, c)-near neighbor problem. the codes of C2LSH and QALSH were modified (with the help of their authors) in order to execute the (r, c)-near neighbor problem. After executing (r, c)-near neigh-

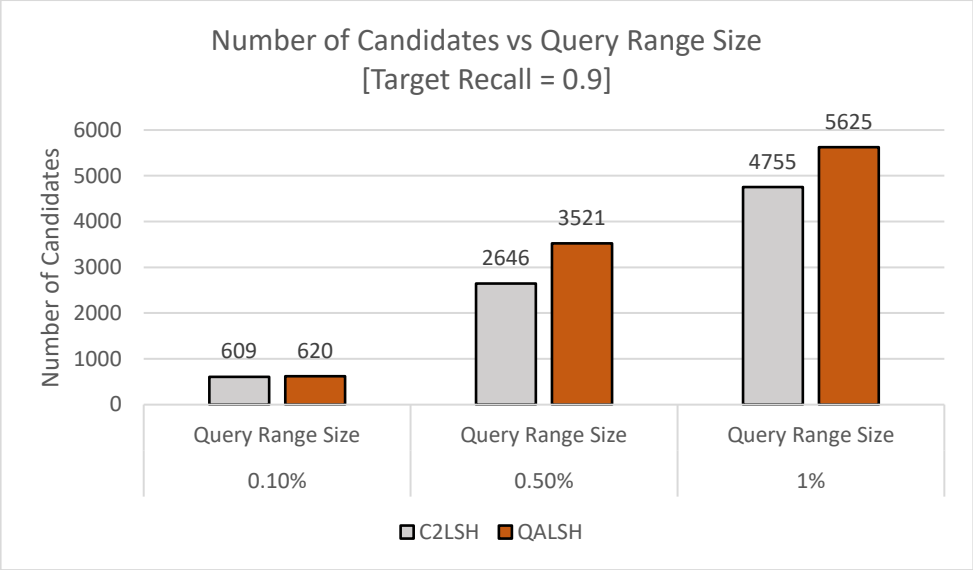


Figure 5.3: Number of Candidates vs. Varying Query Range Sizes [Data=Audio, Target Recall=0.9]

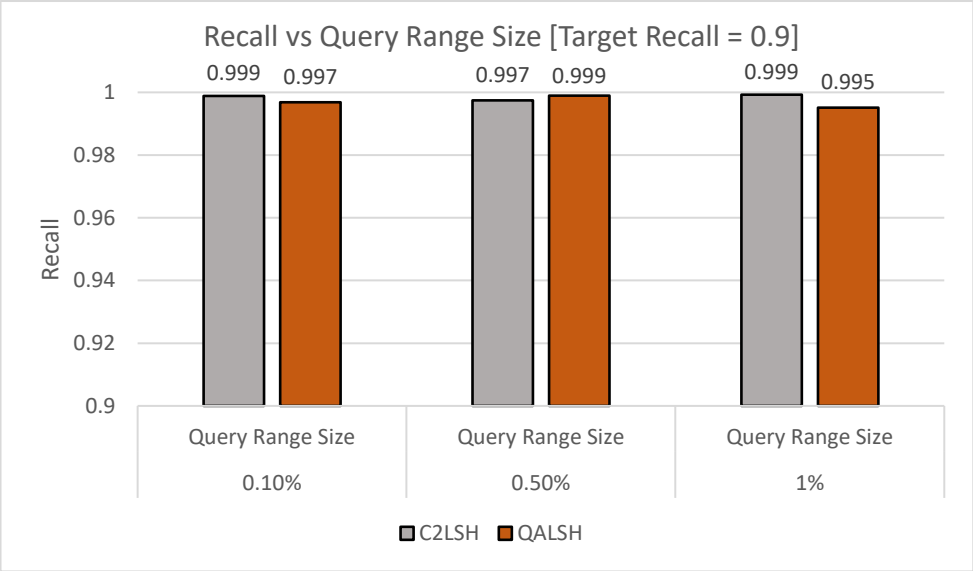


Figure 5.4: Recall vs. Varying Query Range Sizes [Data=Audio, Target Recall=0.9]

bor queries, it was found that C2LSH actually performs better than QALSH for the (r, c)-near neighbor problem by generating less candidates for different query sizes. I present these results in Figures 5.3 and 5.4. Figure 5.3 shows that for different query range sizes, C2LSH generates less candidates than QALSH, and Figure 5.4 shows that both C2LSH and QALSH return a very high recall for a target input recall of 0.9. It is also important to note that QALSH incurs extra runtime processing because it has to hash the data into query-aware buckets (in [43], the authors show that for the top-k problem, their approach is still faster than C2LSH even with this additional overhead). Since C2LSH generates less candidates than QALSH for the (r,c)-near neighbor problem, C2LSH was used instead of QALSH for performing high-dimensional range query processing in PSLSH.

5.5 Experimental Evaluation

In this section, I evaluate the effectiveness of the proposed index structure, PSLSH. In order to evaluate, several real data sets with different characteristics were used, under different system parameters. All the experiments were run on the academic cloud environment, Chameleon Cloud[5]. M1.large instances, consisting of an 8GB RAM and an 80GB non-volatile storage were used. All the experiments were run on an Ubuntu 16.04 operating system. All results presented in this section are an average of 20 runs. Under the guidance of the authors of C2LSH [36], their source code (that could only run top-k nearest neighbor (c-k-NN) queries) was modified to run radius search (r-c-NN) queries. Since there is no work that directly aims at solving my problem, PSLSH was evaluated against the following alternatives. In each of the alternatives, the state-of-the-art C2LSH implementation is used for finding the candidates of the point queries in the set query.

[5]https://www.chameleoncloud.org/

- **Naive-LSH-Pre:** In this alternative, the candidates for each point query in the set query are found, then the false positives that don't satisfy each of the point queries (by finding the exact distance between each candidate and the corresponding point query) are removed, and then the points that satisfy at least θ point queries in the set query are found. In the following figures, in this chapter, this alternative is referred to as Naive-Pre.

- **Naive-LSH-Post:** In this alternative, the candidates for each point query in the set query are found, then the candidates that satisfy at least θ point queries in the set query are found, and then the false positives (by finding the exact distance of each candidate with each of the point queries) that don't satisfy the set query are found to calculate the final result set. In the following figures, in this chapter, this alternative is referred to as Naive-Post.

- **Naive-LSH-Hamming:** This alternative is similar to PSLSH, except the proposed modified Hamming Collision Threshold (as explained in Section 5.4.2) is not used. In this alternative, no Hamming Collision Threshold is used. In the following figures, in this chapter, this alternative is referred to as Naive-Hamming.

5.5.1 Datasets

For the experiments, the following four datasets (of which 1 is synthetically generated - Epidemic - and the rest are real datasets) are used. Similar to the experimental setup in [36], each of the dimension values are normalized to be integers in the range of [0,10000].

- **Epidemic:** This dataset consists of 193,185 128-dimensional SIFT points that are extracted from time-series simulation data depicting the SEIR model. 128-

Parameter	Value range
Target Recall (δ_H)	0.85; **0.9**; 0.95;
# of Queries in Query Workload (s)	30; **40**; 50;
Ratio of Satisfied Queries (θ/s)	0.65; **0.75**; 0.85;
Number of Layers (m)	125; **150**; 200;
Ratio of Negative Queries	**0**; 0.05; 0.1;

Table 5.1: Parameters and Default Values (in bold)

dimensional SIFT feature points are extracted from the simulated time-series of the SEIR model. The simulation ensembles for the SEIR model are generated by using the Spatiotemporal Epidemiological Modeler [32] as described in [58]. The page size is set to be 4KB.

- **ColorHistogram[5]:** This dataset consists of 68,040 32-dimensional color histograms. The page size is set to be 4KB.

- **Mnist[6]:** This dataset consists of 60K objects of 50-dimensions. The page size is set to be 4KB.

- **P53[7]:** This dataset consists of 31,008 objects of 5409-dimensions. The page size is set to be 64KB.

5.5.2 Evaluation Criteria and Parameters

I evaluate the effectiveness of PSLSH by comparing the total set query execution time to those of the alternative strategies. Since C2LSH is used as the base LSH algorithm for all the alternative strategies, the index creation time or the index size is not presented (since the same total number of layers are created for all alternatives).

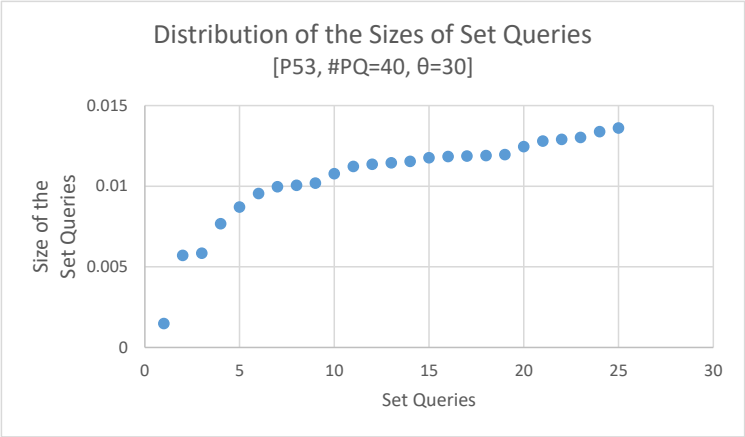


Figure 5.5: Distribution of the Sizes of Different Query Workloads [Data=P53, Number of Point Queries=40, $\theta = 30$]

Table 4.2 shows the parameters and their range values that are used in the evaluation of PSLSH. In the experiments, $c_H = 2$. Note that, the values for all cost models that are presented in this section are chosen by using the Epidemic dataset as the training dataset. I show the effectiveness of these models by comparing the estimated values of the cost models with the observed values for the P53 dataset. The observed values for the remaining datasets also follow similar behavior.

5.5.3 Distribution of Different Set Queries

In order to show its effectiveness, PSLSH is evaluated against set queries of varying sizes. The size of a set query is measured as the number of results in the set query with respect to the dataset size. The radiuses of the point queries are chosen such that the number of results for the point queries are between 0.1% and 2% of the dataset size, and the total number of results of the set queries are between 0.1% and 2% of the size of the dataset. These ranges were chosen so that the set query sizes will be diverse. Figure 5.5 shows the diverse distribution of sizes of different set queries for the P53 dataset.

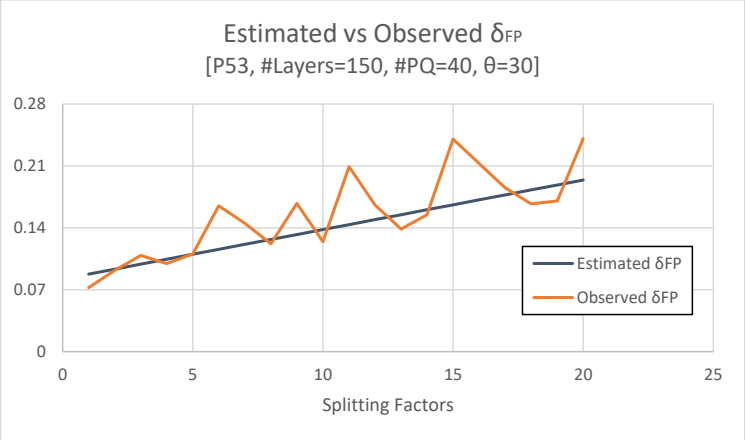


Figure 5.6: Estimated vs Observed δ_{FP_E} [Data=P53, Number of Point Queries=40, $\theta = 30$]

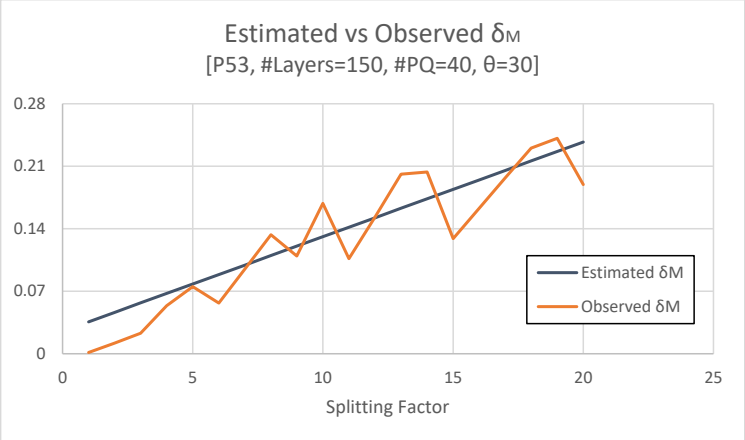


Figure 5.7: Estimated vs Observed δ_{FN_E} [Data=P53, Number of Point Queries=40, $\theta = 30$]

5.5.4 Analysis of Cost Models

Estimated vs Observed δ_{FP_E} and δ_{FN_E}

As mentioned, the models were generated using past statistics from the Epidemic dataset. Figures 5.6 and 5.7 show that these models are applicable for real datasets such as P53 as well. The variations that are seen in the Figures are due to the values

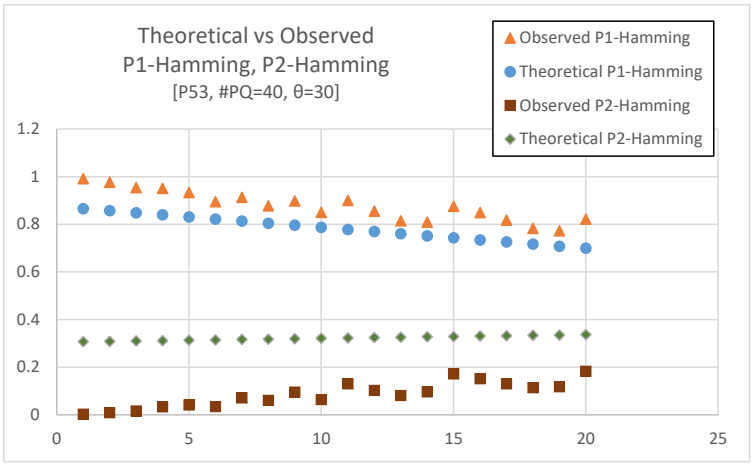


Figure 5.8: Splitting Factor (ρ) vs Theoretical/Observed P'_{1_H} and P'_{2_H} [Data=P53, Number of Layers=150, Number of Queries=40, $\theta = 30$]

of the Euclidean Collision Thresholds (as explained in Section 5.4.3). These variations are taken into consideration while finding the relationship between ρ and $Time_{IO}$.

Proposed Theoretical vs Observed Collision Probabilities

I compare the proposed theoretical collision probabilities (P'_{1_H} and P'_{2_H}) and observed collision probabilities in this section. Suppose $candList(q_i)$ is the list of candidates for a point query q_i generated by PSLSH$_E$, and the ground truth list for the set query Q (i.e. all the data points that satisfy at least θ point queries) is denoted by $groundTruth(Q)$. Then the observed P'_{1_H} w.r.t query q_i:

$$\frac{|candList(q_i) \cap groundTruth(Q)|}{|groundTruth(Q)|} \tag{5.18}$$

Similarly, suppose the $c_H r_H$-ground truth for Q (i.e. all the data points that do not satisfy at least $c_H\theta$ point queries) is denoted by $c_H r_H groundTruth(Q)$. Then the observed P'_{2_H} w.r.t query q_i:

$$\frac{|candList(q_i) \cap c_H r_H groundTruth(Q)|}{|c_H r_H groundTruth(Q)|} \tag{5.19}$$

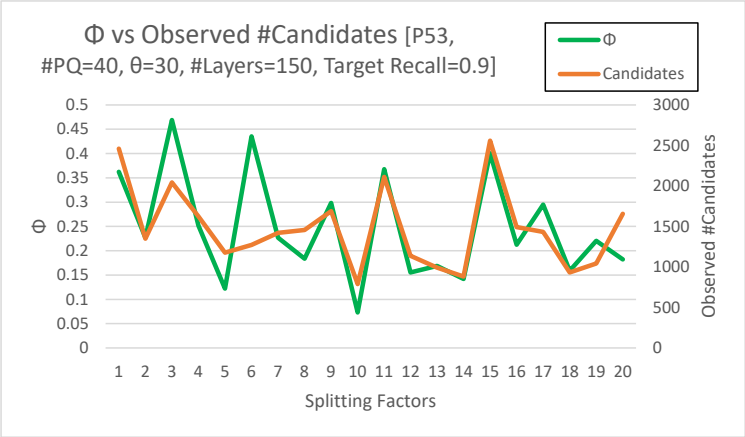


Figure 5.9: ϕ_{Diff_ρ} vs the Number of Candidates for Different Splitting Factors [Data=P53, Number of Layers=150, Number of Queries=40, $\theta = 30$]

In Figure 5.8, I show how the proposed theoretical positive and negative collision probabilities effectively estimate the observed positive and negative collision probabilities for PSLSH$_H$. These values are dependent on the effectiveness of the cost models of δ_{FP_E} and δ_{FN_E} (as explained in Section 5.4.2).

Estimated vs Observed Number of Candidates for Different Splitting Factors

Figure 5.9 shows the effectiveness of ϕ_{Diff_ρ} in predicting the behavior of the number of candidates generated by different splitting factors. It validates Observation 5.4.2 (explained in Section 5.4.3) that ϕ_{Diff_ρ} is dependent on the values of l'_E and l'_H. In Figure 5.10, I compare the estimated number of candidates against the observed number of candidates for different splitting factors. The figure shows the accuracy of the proposed design in estimating the number of candidates based on ϕ_{Diff_ρ}.

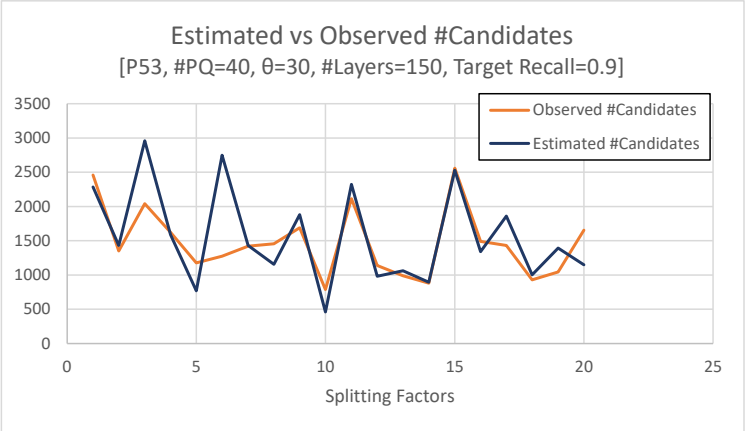


Figure 5.10: Estimated Number of Candidates vs Observed Number of Candidates for Different Splitting Factors [Data=P53, Number of Layers=150, Number of Queries=40, $\theta = 30$]

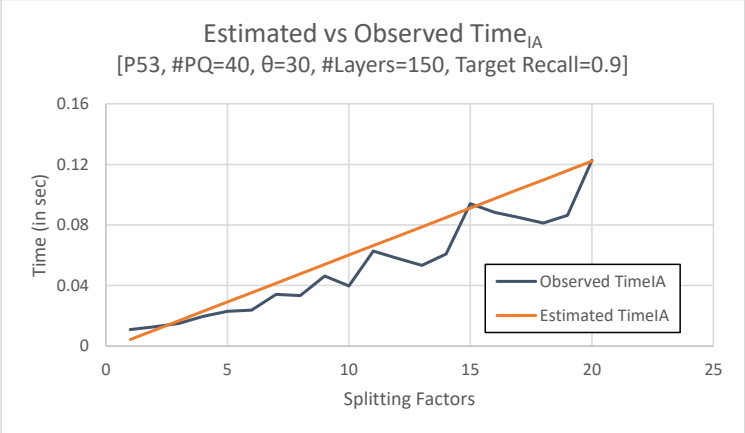


Figure 5.11: Estimated vs Observed $Time_{IA}$ for Different Splitting Factors [Data=P53, Number of Layers=150, Number of Queries=40, $\theta = 30$]

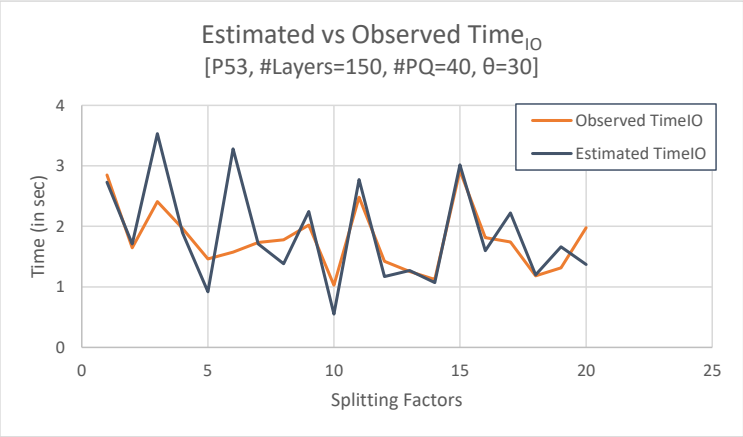


Figure 5.12: Estimated vs Observed $Time_{IO}$ for Different Splitting Factors [Data=P53, Number of Layers=150, Number of Queries=40, $\theta = 30$]

Estimated vs Observed $Time_{IA}$ and $Time_{IO}$

Figures 5.11 and 5.12 show the estimated $Time_{IA}$ and $Time_{IO}$ against the observed values. The models were created using statistics from the Epidemic dataset. Hence, in these figures, I show how the estimated $Time_{IA}$ and $Time_{IO}$ compare for the observed values for a different dataset (P53). As explained in Sections 5.4.3 and 5.4.5, $Time_{IA}$ is proportional to the splitting factor and the number of data points in the dataset. The time taken to access the index for 1 data point for different splitting factors is first calculated. For any given dataset, $Time_{IA}$ can then be estimated for any splitting factor. Similarly, $Time_{IO}$ is proportional to the number of candidates and the dimensionality of the dataset (as explained in Sections 5.4.3 and 5.4.5). The time taken to access a 1 1-dimensional data point is calculated. Since PSLSH is able to estimate the number of candidates (using the strategy presented in Section 5.4.3), PSLSH can then estimate the time it takes to access the number of estimated d-dimensional candidate points.

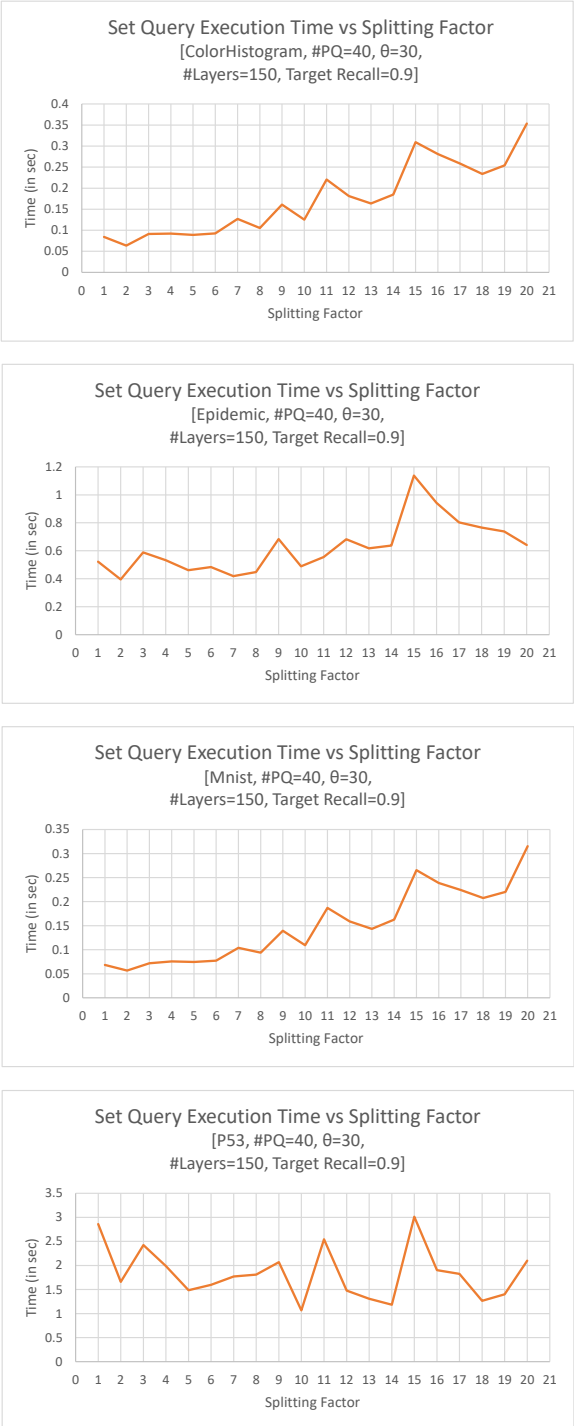


Figure 5.13: Set Query Execution Time of PSLSH for Different Splitting Factors (for Different Datasets)

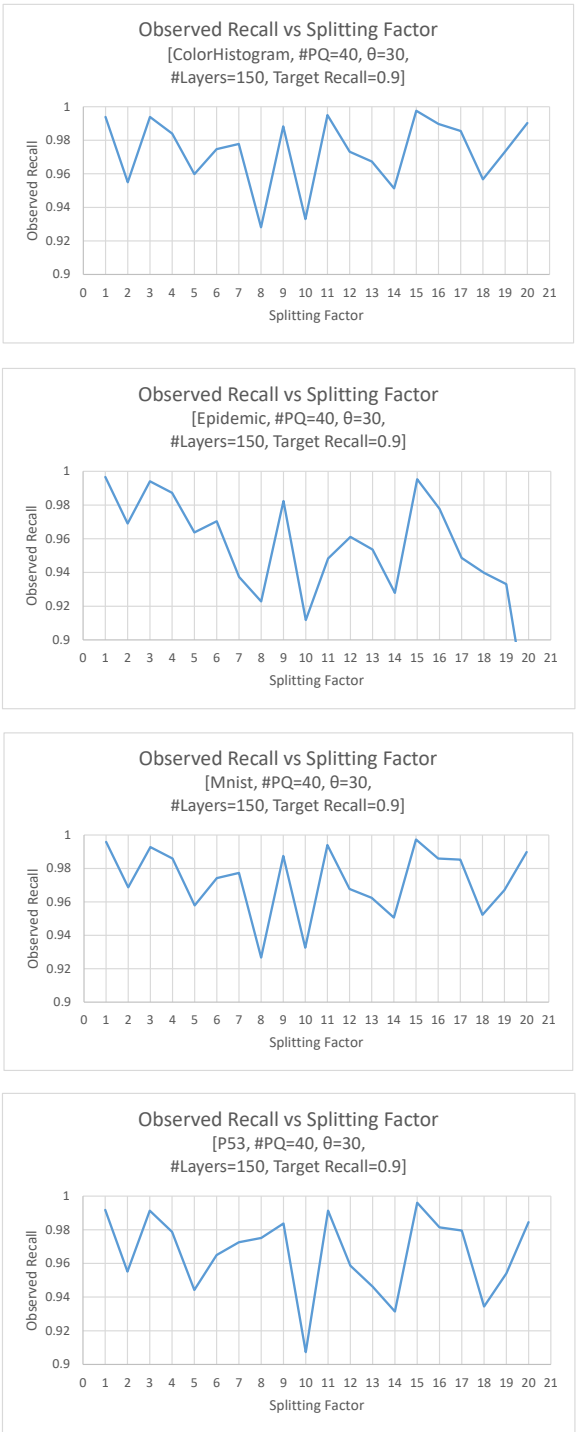


Figure 5.14: Observed Recall of PSLSH for Different Splitting Factors (for Different Datasets)

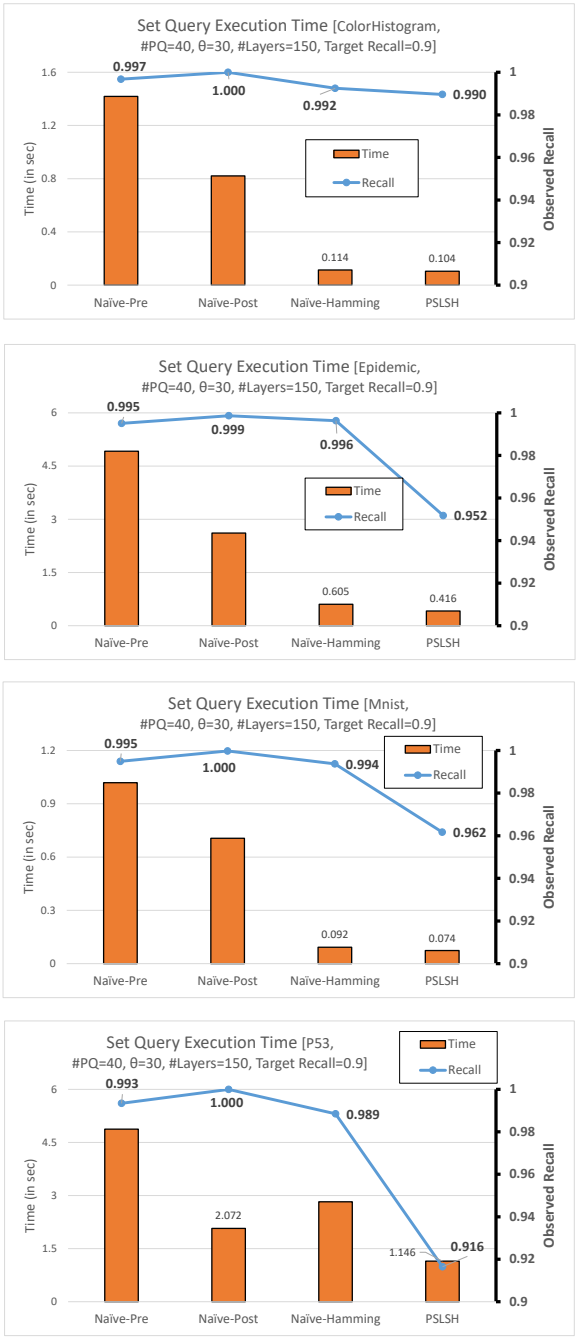


Figure 5.15: Comparison of PSLSH (Set Query Execution Time) against its Alternatives (for Different Datasets and Default Settings)

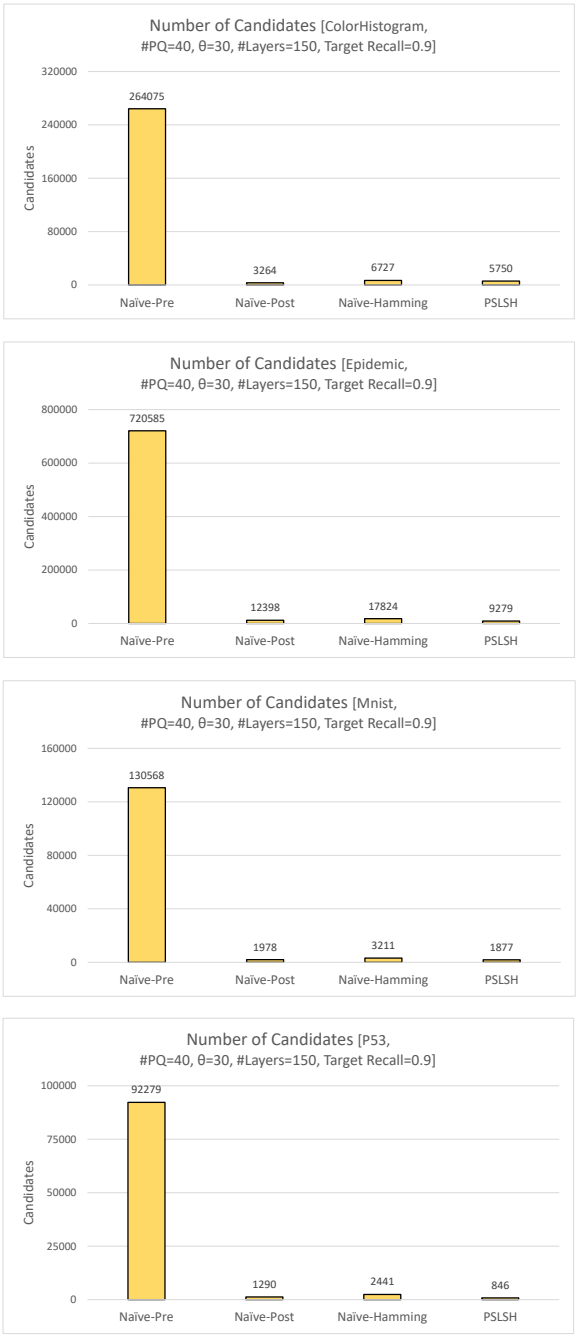


Figure 5.16: Comparison of PSLSH (Number of Candidates) against its Alternatives (for Different Datasets and Default Settings)

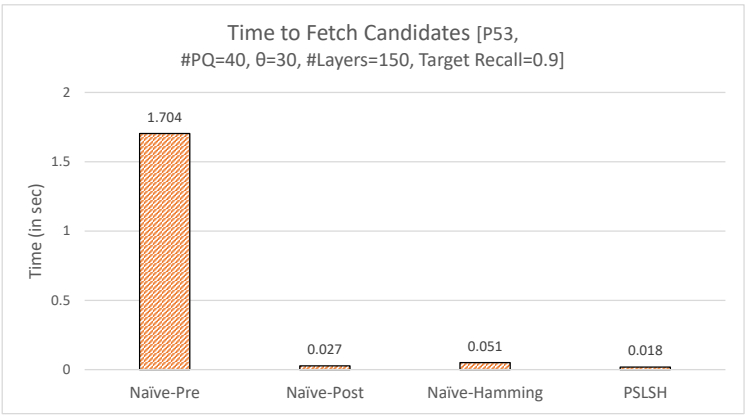


Figure 5.17: Time to Fetch Candidates [Data=P53, Number of Layers=150, Number of Queries=40, $\theta = 30$]

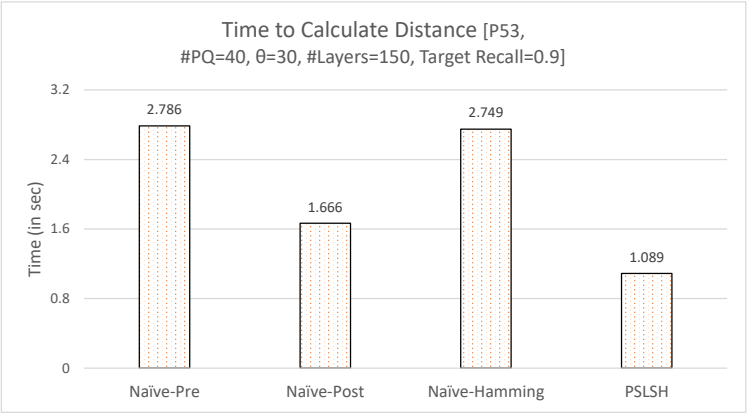


Figure 5.18: Time to Calculate Exact Distances between Candidates and Point Queries [Data=P53, Number of Layers=150, Number of Queries=40, $\theta = 30$]

5.5.5 Discussion of the Performance Results

In this section, I analyze the execution performance of PSLSH against its alternatives. I first compare the performance of PSLSH against an exhaustive search for the most efficient splitting factor. Figure 5.13 shows the execution times for different set queries for different datasets (for the default settings as described in Table5.1). For number of layers (<200; which are enough number of layers), splitting factors above (approximately) 20 are not very effective because large number of false positives and misses are generated by PSLSH_E (since very few layers are assigned to PSLSH_E). Hence, in the following discussions, the most efficient splitting factor is found from the first 20 splitting factors. In the following discussion, the datasets, ColorHistogram, Mnist, and Epidemic, are referred to as low-dimensional datasets, and P53 as a high-dimensional dataset. For low-dimensional datasets, it can be seen from Figure 5.13 that the most efficient splitting factor is a smaller splitting factor (e.g. for all these datasets, the splitting factor of 2 is the most efficient). In contrast, for the high-dimensional dataset, the most efficient splitting factor is 10. This is because, for low-dimensional datasets, $Time_{IA}$ dominates the overall set query execution cost. Since $Time_{IA}$ increases with the splitting factor, the most efficient splitting factor is a smaller splitting factor. For a splitting factor of 2, the Hamming Collision threshold is better able to prune false positives than a splitting factor of 1, and hence a splitting factor of 2 is better than a splitting factor of 1. For the high-dimensional dataset, $Time_{IO}$ dominates the overall set query execution cost, and hence a higher splitting factor (that minimizes the number of candidates) is chosen. Figure 5.14 shows the corresponding observed recalls for the set queries for different datasets across different splitting factors. This figure shows that, by using a splitting factor up to 20, PSLSH can also achieve the target recall (default=0.9). The only exception in the above

figure, is for when the splitting factor is 20 for the Epidemic dataset. This happens because the misses in PSLSH$_E$ are higher than expected because less number of layers are assigned to PSLSH$_E$.

Figure 5.15 shows the execution times for different set queries for different datasets for PSLSH and its alternatives. The splitting factor that was chosen by PSLSH always was the most efficient splitting factor when compared with the first 20 splitting factors. It is worth noting that the main difference between the execution times of PSLSH in Figures 5.13 and 5.15 is that the execution time of PSLSH in Figure 5.15 includes the time to estimate $Time_{IO}$ (as described in Section 5.4.3). Due to the accuracy and effectiveness of the cost models and the proposed design, PSLSH can determine the most efficient splitting factor. Figure 5.15 also shows that PSLSH is the most efficient when compared to its alternatives. Naive-LSH-Pre is always the slowest strategy across datasets. This is because it has to remove false positives after processing each point query in the set query (by fetching the data point from the secondary storage). Naive-LSH-Hamming generates more candidates than PSLSH across datasets (which can be seen from Figure 5.16), but the execution times are comparable for the low-dimensional datasets. As the dimensionality increases, $Time_{IO}$ becomes more dominant, and hence it is much slower for the high-dimensional dataset. Naive-LSH-Post generates less candidates than Naive-LSH-Hamming, but since it requires to process more point queries than Naive-LSH-Hamming, for low-dimensional datasets (where $Time_{IA}$ is the dominant cost), Naive-LSH-Hamming is still faster. For high-dimensional datasets, Naive-LSH-Post is much faster than Naive-LSH-Hamming because it generates less candidates. Figure 5.16 shows the number of candidates that are generated by PSLSH and the alternative strategies. It is worth noting that Naive-LSH-Pre generates substantially more candidates than PSLSH and other strategies. For instance, for the P53 dataset, Naive-LSH-Pre generates approximately 75 times

more candidates than Naive-LSH-Post, but the execution time (Figure 5.15) of Naive-LSH-Pre is approximately only 2.5 times more than that of Naive-LSH-Post. $Time_{IO}$ consists of two sub-costs: a) the time required to fetch the candidate data points from the secondary storage (Figure 5.17), and b) the time required to calculate the exact distance between each candidate point and the point queries to remove false positives that do not satisfy at least θ point queries (Figure 5.18). It can be seen in Figure 5.17 that the time to fetch the candidate points from the secondary storage for Naive-LSH-Pre is approximately 65 times the cost of Naive-LSH-Post. On the other hand, the time to calculate distances for Naive-LSH-Pre is approximately only 1.75 times than that of Naive-LSH-Post. This happens because of the way the Euclidean distance calculation function is implemented. If the target radius is known, the Euclidean distance calculation can be terminated if the distance between the two points is greater than the target radius. In this case, all the dimensions of the points are not compared with. For several candidates generated by Naive-LSH-Pre, the Euclidean distance calculation is able to terminate early. Hence, even though substantially higher number of candidates are generated by Naive-LSH-Pre when compared with other alternatives, the total time is not as high as one would expect. Yet, in all scenarios, Naive-LSH-Pre is the slowest alternative.

Impact of Varying Target Recall

Figures 5.19 and 5.20 show the impact of varying target recall for different datasets. In this figure, the secondary axis shows the achieved recall for the different alternative strategies. Naive-LSH-Pre and Naive-LSH-Post are unable to adapt to the varying recall. Both of these strategies return very high recall even when the target recall is 0.85, which results in wasted work and slower execution times. Naive-LSH-Post is always able to achieve the highest recall (at the expense of slower execution times).

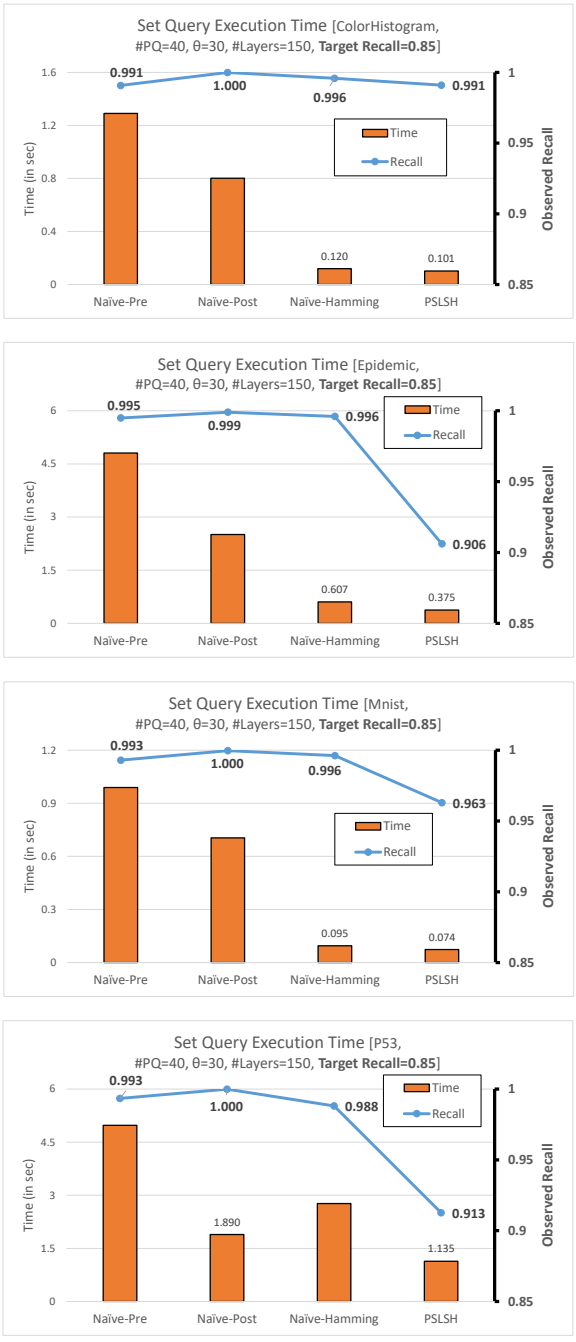


Figure 5.19: Impact of Varying Target Recall (0.85) (for Different Datasets)

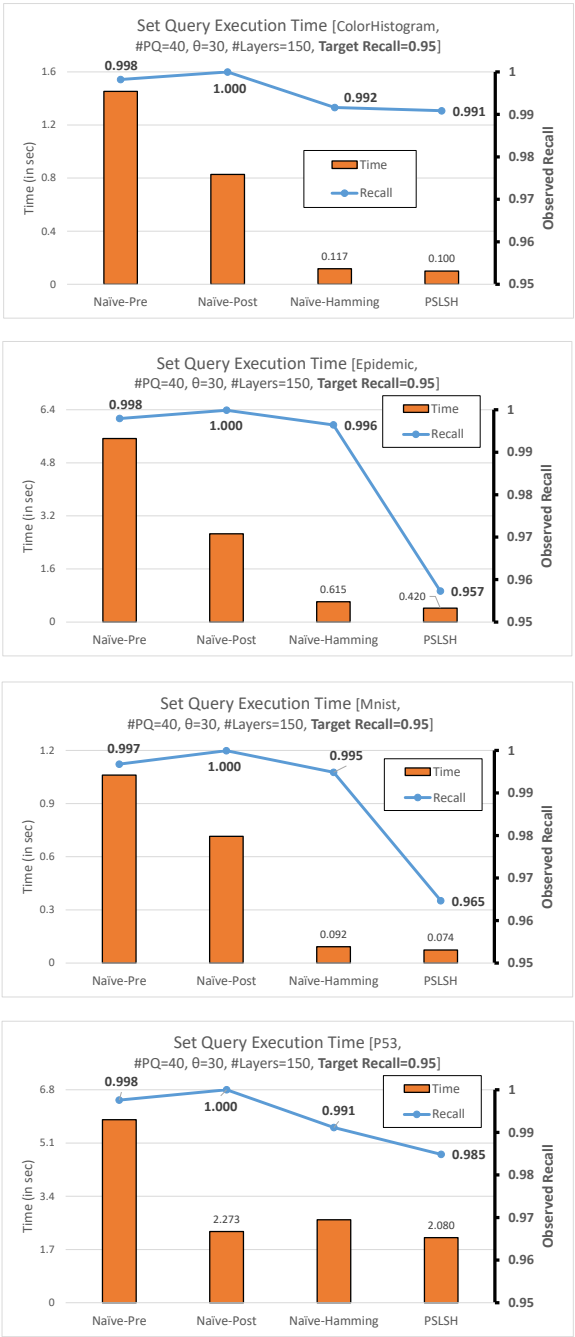


Figure 5.20: Impact of Varying Target Recall (0.95) (for Different Datasets)

PSLSH is able to adapt appropriately according to the input target recall for the set query. In all the cases, PSLSH is able to satisfy the target recall.

Impact of Varying Number of Layers

Figures 5.21 and 5.22 shows the impact of varying number of layers on PSLSH. When the number of layers changed significantly, I had to retrain the δ_{FP_E} and δ_{FN_E} models. As mentioned before, when there are less number of layers available to PSLSH$_E$, more false positives and false negatives are generated by PSLSH$_E$. Hence, for different number of layers, since the subsequent models and calculation of the Hamming Collision Threshold are dependent on the values of δ_{FP_E} and δ_{FN_E}, I retrained the models on the Epidemic dataset. These figures show that PSLSH can choose a splitting factor that is still faster than its alternatives for different number of layers, across all datasets.

Impact of Varying Number of Point Queries in the Set Query

Figures 5.23 and 5.24 show that PSLSH can still perform better than its competitors even when the number of point queries in the set query vary. For low-dimensional datasets, as consistent with the previous results, PSLSH is much faster than Naive-LSH-Pre and Naive-LSH-Post. For high-dimensional datasets, Naive-LSH-Post is the fastest among the alternative strategies, but still slower than PSLSH. As the number of point queries in the set query decrease or increase, the time gain of PSLSH over Naive-LSH-Post (for high-dimensional datasets) remains consistently high.

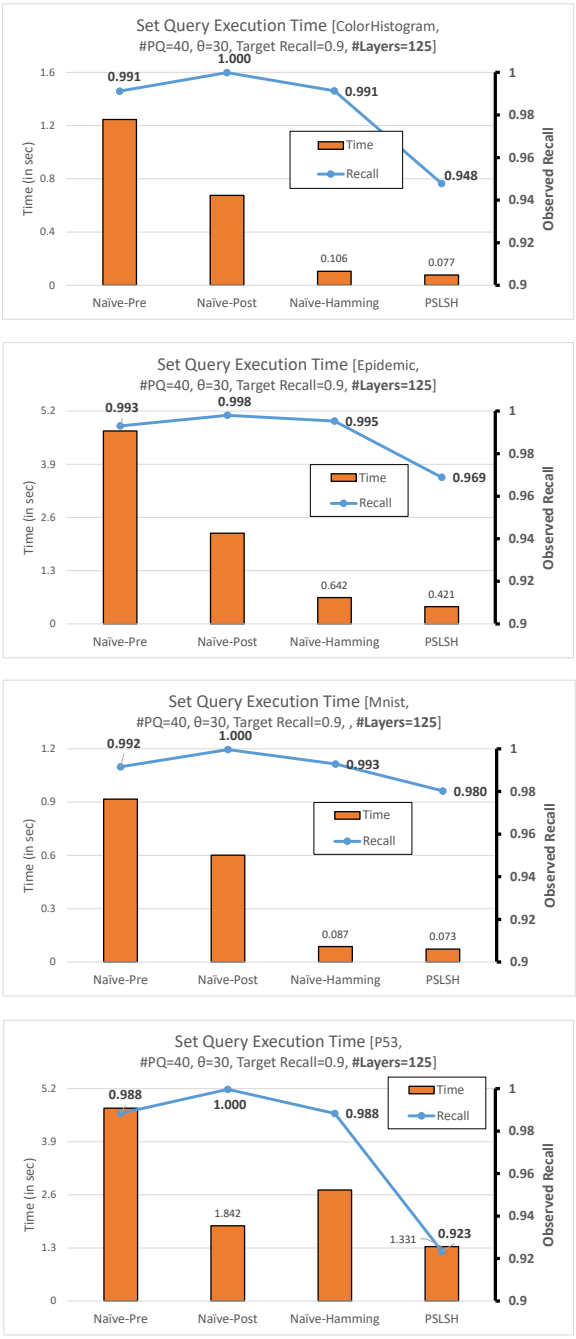


Figure 5.21: Impact of Number of Layers (125) (for Different Datasets)

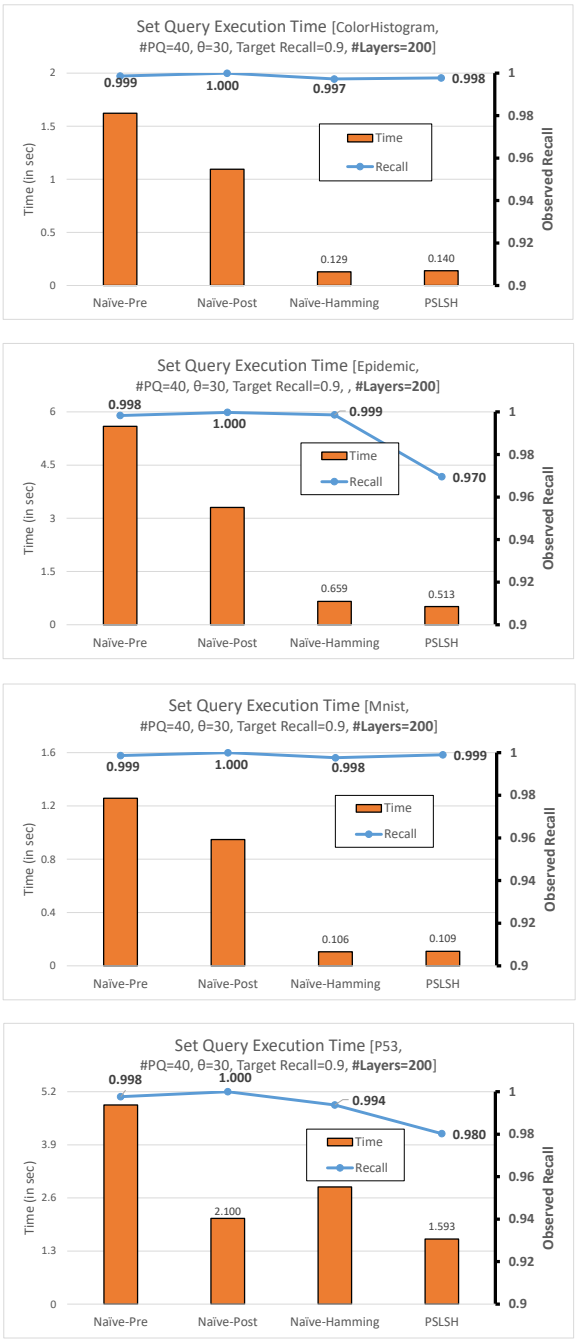


Figure 5.22: Impact of Number of Layers (200) (for Different Datasets)

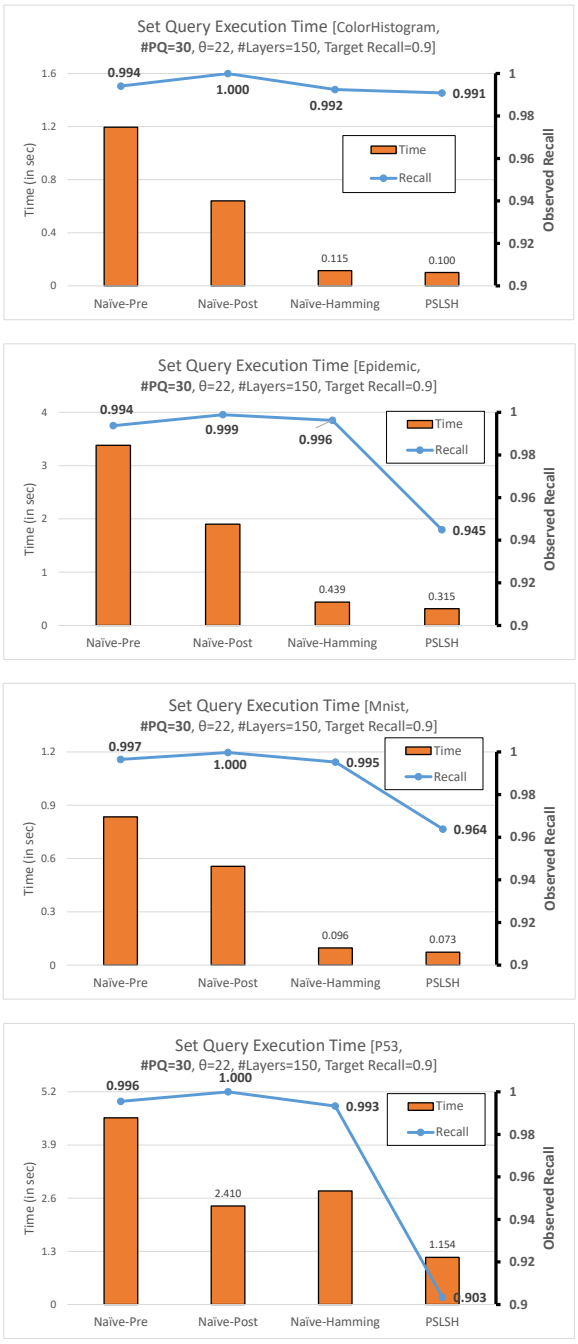


Figure 5.23: Impact of Varying Number of Point Queries (30) in the Set Query (for Different Datasets)

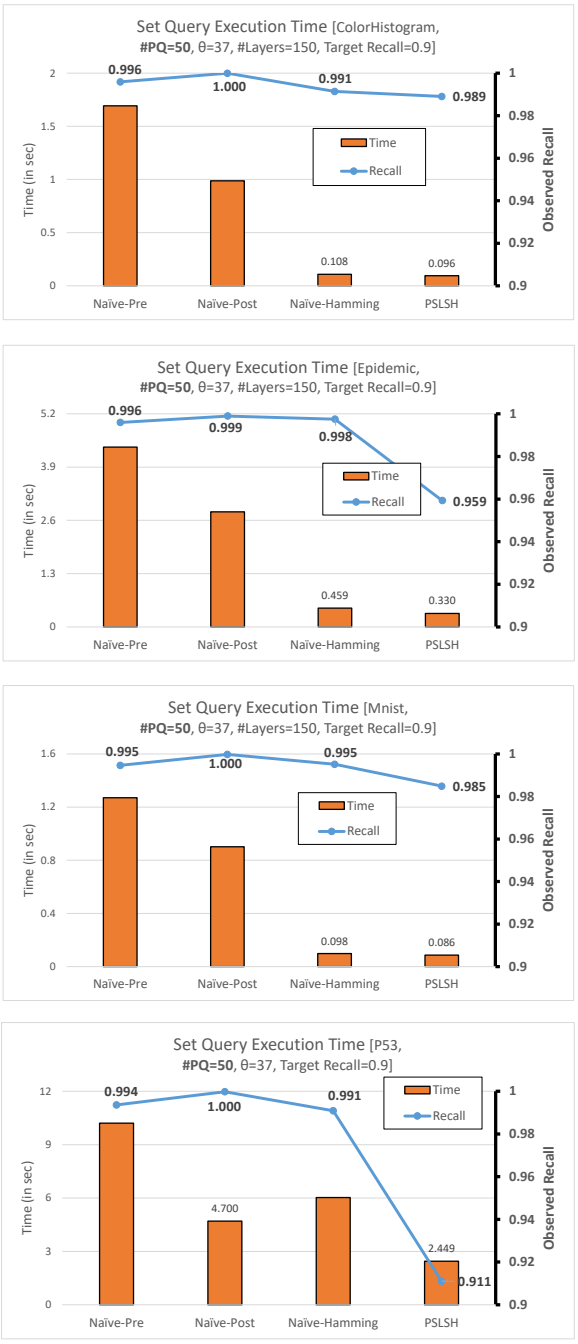


Figure 5.24: Impact of Varying Number of Point Queries (50) in the Set Query (for Different Datasets)

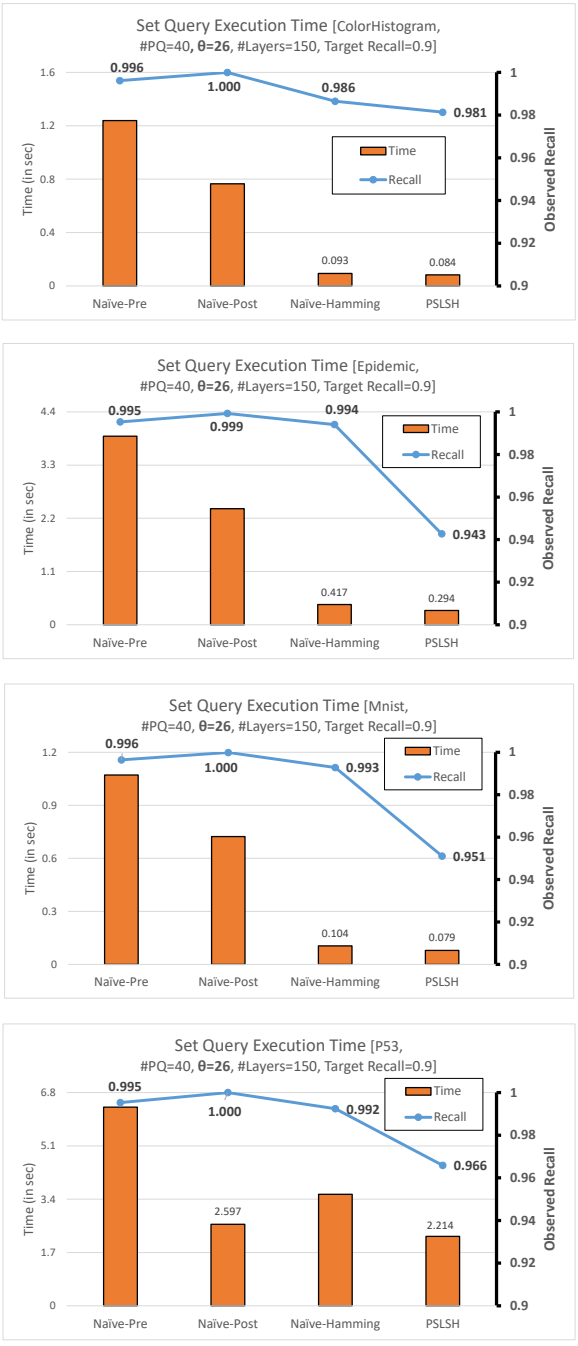


Figure 5.25: Impact of Varying % of Required Satisfied Point Queries (65%) in the Set Query (for Different Datasets)

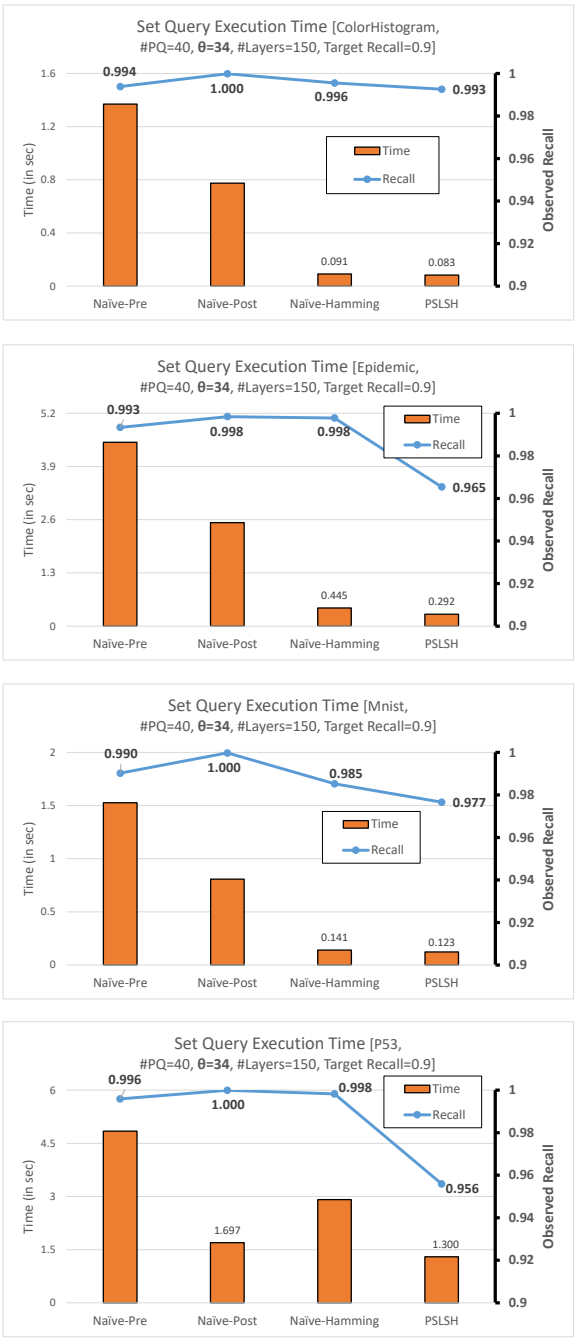


Figure 5.26: Impact of Varying % of Required Satisfied Point Queries (85%) in the Set Query (for Different Datasets)

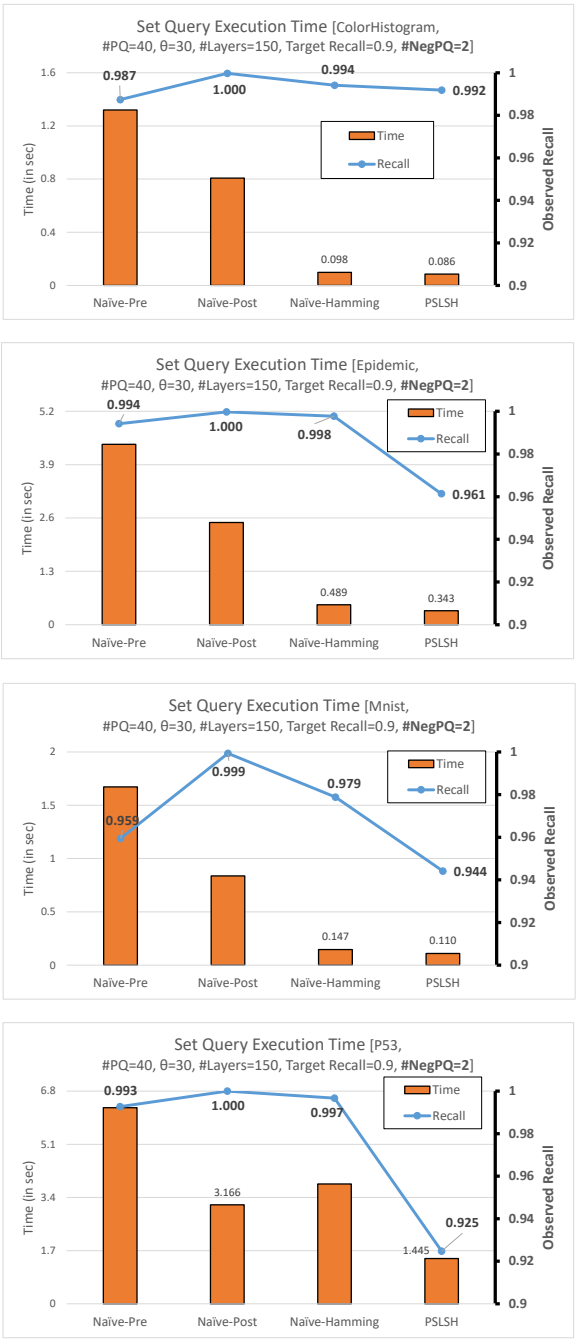


Figure 5.27: Impact of Varying % of Negative Point Queries (5%) in the Set Query (for Different Datasets)

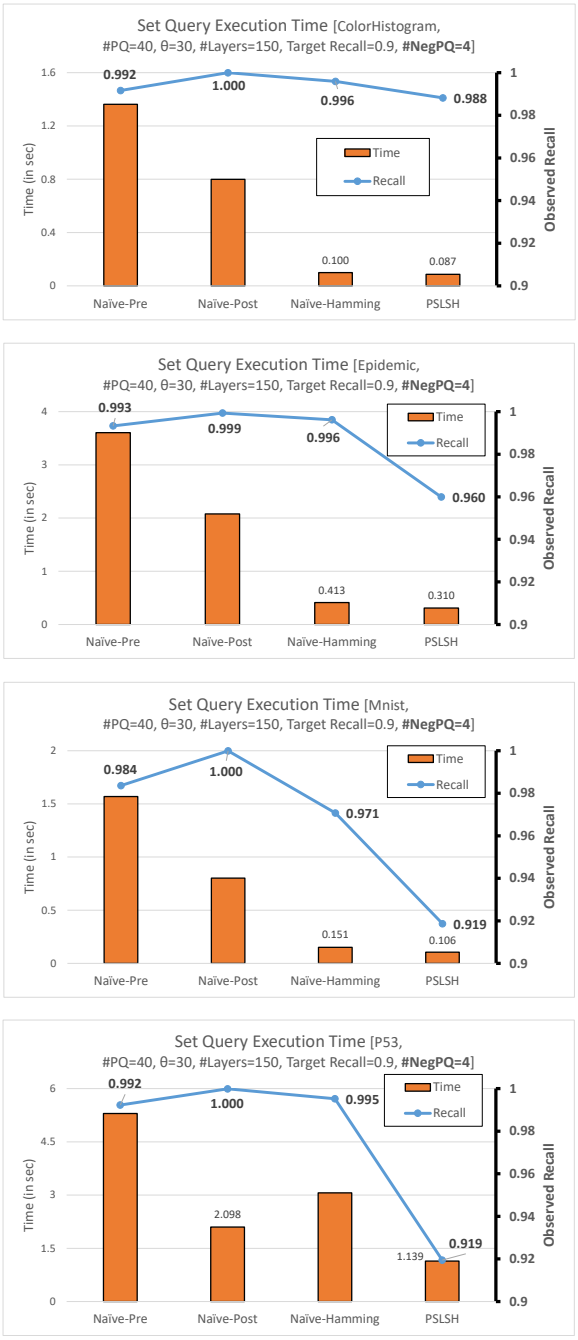


Figure 5.28: Impact of Varying % of Negative Point Queries (10%) in the Set Query (for Different Datasets)

Impact of Varying % of Required Satisfied Point Queries, Negative Point Queries in the Set Query

Figures 5.25 and 5.26 show the performance results of PSLSH and the competitors for varying % of required satisfied point queries in a given set query. PSLSH is again the fastest strategy across all datasets for different % of required satisfied point queries in the set query. Figure 5.26 shows that, for high-dimensional datasets, the time gain of PSLSH over Naive-LSH-Post is significantly higher when the % of required satisfied point queries in the set query is higher (85%). Figures 5.27 and 5.28 show the impact of negative point queries in the set query. Even when the set query contains negative point queries, PSLSH shows a time gain over the alternative strategies.

Chapter 6

CONCLUSION

The main goal of this dissertation is to design scalable index structures that are optimized for executing range query workloads. I particularly look at three different kinds of spaces: a) 1D spaces, b) 2D spaces, and c) high-dimensional spaces. Each of these spaces have their own unique challenges and different index structures need to be built to address these challenges. In this dissertation, I proposed index structures that tackled these challenges for each of the above mentioned different dimensional spaces. I presented unique cost models, storage mechanisms, and algorithms for these index structures to efficiently execute range query workloads.

6.1 Range Query Workloads in 1D Spaces

Column-stores use compressed bitmap-indices for answering queries over data columns. When the data domain is hierarchical, organizing the bitmap indices hierarchically can help answer queries over different sub-ranges of the attribute domain more efficiently. In Chapter 3, I showed that existing *inclusive* strategies for leveraging hierarchically organized bitmap indices can be sub-optimal in terms of their IO costs unless the query ranges are small. I also showed that an *exclusive (cut-selection) strategy* provides gains when the query ranges are large and that a *hybrid (cut-selection) strategy* can provide best solutions, improving over both strategies even when the ranges of interest are relatively small. I also presented algorithms for implementing the *hybrid strategy* efficiently for a single query or a workload of multiple queries, in scenarios with and without memory limitations. In particular, I showed that when the memory is constrained, selecting the right subset of bitmap indices becomes difficult;

but, I also showed that, even in this case, there exists efficient cut-selection strategies that return close to optimal results, especially in situations where the memory limitations are very strict. Experiment results confirmed that the *cut-selection* algorithms presented in Chapter 3 are efficient, scalable, and highly-effective.

6.2 Range Query Workloads in 2D Spaces

In Chapter 4, I showed that bitmap-based indexing can be highly effective for executing range query workloads on spatial data sets. I introduced a novel *compressed spatial hierarchical bitmap (cSHB)* index structure that takes a spatial hierarchy and uses that to create a hierarchy of compressed bitmaps to support spatial range queries. I also introduced a novel block-based storage mechanism for storing the hierarchy of compressed bitmaps effectively. Queries are processed on cSHB index structure by selecting a relevant subset of the bitmaps and performing *compressed-domain* bitwise logical operations. I also developed novel cost models and bitmap selection algorithms that identify the subset of the bitmap files in this hierarchy for processing a given spatial range query workload. These cost models and algorithms are further optimized to take into consideration the block-based storage mechanism. I compared the proposed *compressed spatial hierarchical bitmap (cSHB)* index structure with state-of-the-art spatial extensions of popular database management systems. Experiments on synthetic and real data sets showed that the proposed index structure is highly efficient in supporting spatial range query workloads.

6.3 Range Query Workloads in High-Dimensional Spaces

In Chapter 5, I presented the novel index structure, *Point Set LSH (PSLSH)* for solving range query workloads in high-dimensional spaces. Multimedia such as image data or time series data are represented by a set of important features that

are extracted using feature extraction algorithms. Similarity search queries over these features are an important part of several multimedia applications. Often times, points that satisfy certain number of queries are needed to answer these similarity search queries. Traditional LSH-based index structures require users to input a guarantee on individual query points instead of a guarantee on the entire set query. This can lead to potential misses and false positives, which lead to higher query processing times. In this work, I introduced a novel index structure, PSLSH, by designing a Hamming distance based LSH index structure (PSLSH_H) on top of the existing Euclidean distance base LSH structure (PSLSH_E) to solve the above challenge. I also presented the design and the theoretical analysis of PSLSH. The experimental analysis proves the effectiveness of PSLSH for different datasets under different settings.

REFERENCES

[1] "Compressedbitset - wah compressed bitset for java", https://code.google.com/p/compressedbitset/ (2007).

[2] "Luciddb - home", http://www.luciddb.org/html/main.html (2013).

[3] "Oracle database 10g", http://docs.oracle.com/cd/B13789_01/server.101/b10736/indexes.htm (2013).

[4] "Openstreetmap", http://www.openstreetmap.org/ (2015).

[5] "Colorhistogram", http://kdd.ics.uci.edu/databases/CorelFeatures/ (2017).

[6] "Mnist", http://yann.lecun.com/exdb/mnist/ (2017).

[7] "P53", http://archive.ics.uci.edu/ml/datasets/p53+Mutants (2017).

[8] Abadi, D., S. Madden and M. Ferreira, "Integrating compression and execution in column-oriented database systems", in "Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data", SIGMOD '06, pp. 671–682 (ACM, New York, NY, USA, 2006), URL http://doi.acm.org/10.1145/1142473.1142548.

[9] Aji, A., F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang and J. Saltz, "Hadoop gis: A high performance spatial data warehousing system over mapreduce", Proc. VLDB Endow. **6**, 11, 1009–1020, URL http://dx.doi.org/10.14778/2536222.2536227 (2013).

[10] Andrade, H., T. Kurc, A. Sussman and J. Saltz, "Efficient execution of multiple query workloads in data analysis applications", in "Proceedings of the 2001 ACM/IEEE Conference on Supercomputing", SC '01, pp. 53–53 (ACM, New York, NY, USA, 2001), URL http://doi.acm.org/10.1145/582034.582087.

[11] Bawa, M., T. Condie and P. Ganesan, "Lsh forest: Self-tuning indexes for similarity search", in "Proceedings of the 14th International Conference on World Wide Web", WWW '05, pp. 651–660 (ACM, New York, NY, USA, 2005), URL http://doi.acm.org/10.1145/1060745.1060840.

[12] Bay, H., T. Tuytelaars and L. Van Gool, "Surf: Speeded up robust features", in "European conference on computer vision", pp. 404–417 (Springer, 2006).

[13] Bayer, R., "The universal b-tree for multidimensional indexing: General concepts", in "Proceedings of the International Conference on Worldwide Computing and Its Applications", WWCA '97, pp. 198–209 (Springer-Verlag, London, UK, UK, 1997), URL http://dl.acm.org/citation.cfm?id=645965.674403.

[14] Beckmann, N., H.-P. Kriegel, R. Schneider and B. Seeger, "The r*-tree: An efficient and robust access method for points and rectangles", in "Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data", SIGMOD '90, pp. 322–331 (ACM, New York, NY, USA, 1990), URL http://doi.acm.org/10.1145/93597.98741.

[15] Bellatreche, L., R. Missaoui, H. Necir and H. Drias, "Selection and pruning algorithms for bitmap index selection problem using data mining", in "Proceedings of the 9th International Conference on Data Warehousing and Knowledge Discovery", DaWaK'07, pp. 221–230 (Springer-Verlag, Berlin, Heidelberg, 2007), URL http://dl.acm.org/citation.cfm?id=2391952.2391978.

[16] Belussi, A. and C. Faloutsos, "Self-spacial join selectivity estimation using fractal concepts", ACM Trans. Inf. Syst. **16**, 2, 161–201, URL http://doi.acm.org/10.1145/279339.279342 (1998).

[17] Bentley, J. L., "Multidimensional binary search trees used for associative searching", Communications of the ACM **18**, 9, 509–517, URL http://doi.acm.org/10.1145/361002.361007 (1975).

[18] Berchtold, S., D. A. Keim and H.-P. Kriegel, "The x-tree: An index structure for high-dimensional data", in "Proceedings of the 22th International Conference on Very Large Data Bases", VLDB '96, pp. 28–39 (Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996), URL http://dl.acm.org/citation.cfm?id=645922.673502.

[19] Bercken, J. V. d., B. Blohsfeld, J.-P. Dittrich, J. Krämer, T. Schäfer, M. Schneider and B. Seeger, "Xxl - a library approach to supporting efficient implementations of advanced database queries", in "Proceedings of the 27th International Conference on Very Large Data Bases", VLDB '01, pp. 39–48 (Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001), URL http://dl.acm.org/citation.cfm?id=645927.672371.

[20] Butz, A. R., "Alternative algorithm for hilbert's space-filling curve", IEEE Trans. Comput. **20**, 4, 424–426, URL http://dx.doi.org/10.1109/T-C.1971.223258 (1971).

[21] Candan, K. S., P. Nagarkar, M. Nagendra and R. Yu, "Rankloud: A scalable ranked query processing framework on hadoop", in "Proceedings of the 14th International Conference on Extending Database Technology", EDBT/ICDT '11, pp. 574–577 (ACM, New York, NY, USA, 2011), URL http://doi.acm.org/10.1145/1951365.1951444.

[22] Chen, Z. and V. Narasayya, "Efficient computation of multiple group by queries", in "Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data", SIGMOD '05, pp. 263–274 (ACM, New York, NY, USA, 2005), URL http://doi.acm.org/10.1145/1066157.1066188.

[23] Chmiel, J., T. Morzy and R. Wrembel, "Hobi: Hierarchically organized bitmap index for indexing dimensional data", in "Proceedings of the 11th International

Conference on Data Warehousing and Knowledge Discovery", DaWaK '09, pp. 87–98 (Springer-Verlag, Berlin, Heidelberg, 2009), URL http://dx.doi.org/10.1007/978-3-642-03730-6_8.

[24] Chmiel, J., T. Morzy and R. Wrembel, "Time-hobi: Indexing dimension hierarchies by means of hierarchically organized bitmaps", in "Proceedings of the ACM 13th International Workshop on Data Warehousing and OLAP", DOLAP '10, pp. 69–76 (ACM, New York, NY, USA, 2010), URL http://doi.acm.org/10.1145/1871940.1871957.

[25] Chovanec, P. and M. Krátký, "On the efficiency of multiple range query processing in multidimensional data structures", in "Proceedings of the 17th International Database Engineering & Applications Symposium", IDEAS '13, pp. 14–27 (ACM, New York, NY, USA, 2013), URL http://doi.acm.org/10.1145/2513591.2513656.

[26] Colantonio, A. and R. Di Pietro, "Concise: Compressed 'n' composable integer set", Information Processing Letters **110**, 16, 644–650, URL http://dx.doi.org/10.1016/j.ipl.2010.05.018 (2010).

[27] Cooper, B. F., R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver and R. Yerneni, "Pnuts: Yahoo!'s hosted data serving platform", Proc. VLDB Endow. **1**, 2, 1277–1288, URL http://dx.doi.org/10.14778/1454159.1454167 (2008).

[28] Council, T. P. P., "Tpc-h benchmark specification", http://www.tpc.org/tpch/ (2013).

[29] Datar, M., N. Immorlica, P. Indyk and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions", in "Proceedings of the Twentieth Annual Symposium on Computational Geometry", SCG '04, pp. 253–262 (ACM, New York, NY, USA, 2004), URL http://doi.acm.org/10.1145/997817.997857.

[30] Dean, J. and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters", Commun. ACM **51**, 1, 107–113, URL http://doi.acm.org/10.1145/1327452.1327492 (2008).

[31] Deliège, F. and T. B. Pedersen, "Position list word aligned hybrid: Optimizing space and performance for compressed bitmaps", in "Proceedings of the 13th International Conference on Extending Database Technology", EDBT '10, pp. 228–239 (ACM, New York, NY, USA, 2010), URL http://doi.acm.org/10.1145/1739041.1739071.

[32] Edlund, S. B., M. A. Davis and J. H. Kaufman, "The spatiotemporal epidemiological modeler", in "Proceedings of the 1st ACM International Health Informatics Symposium", IHI '10, pp. 817–820 (ACM, New York, NY, USA, 2010), URL http://doi.acm.org/10.1145/1882992.1883115.

[33] Feng, Y. and A. Makinouchi, "Ag-tree: A novel structure for range queries in data warehouse environments", in "Proceedings of the 11th International Conference on Database Systems for Advanced Applications", DASFAA'06, pp. 498–512 (Springer-Verlag, Berlin, Heidelberg, 2006), URL http://dx.doi.org/10.1007/11733836_35.

[34] Fenk, R., V. Markl and R. Bayer, "Improving multidimensional range queries of non rectangular volumes specified by a query box set", in "Proc. of International Symposium on Database, Web and Cooperative Systems (DWACOS", (1999).

[35] Finkel, R. A. and J. L. Bentley, "Quad trees a data structure for retrieval on composite keys", Acta Informatica **4**, 1, 1–9, URL http://dx.doi.org/10.1007/BF00288933 (1974).

[36] Gan, J., J. Feng, Q. Fang and W. Ng, "Locality-sensitive hashing scheme based on dynamic collision counting", in "Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data", SIGMOD '12, pp. 541–552 (ACM, New York, NY, USA, 2012), URL http://doi.acm.org/10.1145/2213836.2213898.

[37] Gosink, L., J. Shalf, K. Stockinger, K. Wu and W. Bethel, "Hdf5-fastquery: Accelerating complex queries on hdf datasets using fast bitmap indices", in "Proceedings of the 18th International Conference on Scientific and Statistical Database Management", SSDBM '06, pp. 149–158 (IEEE Computer Society, Washington, DC, USA, 2006), URL http://dx.doi.org/10.1109/SSDBM.2006.27.

[38] Guttman, A., "R-trees: A dynamic index structure for spatial searching", in "Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data", SIGMOD '84, pp. 47–57 (ACM, New York, NY, USA, 1984), URL http://doi.acm.org/10.1145/602259.602266.

[39] Hilbert, D., "Ueber die stetige abbildung einer linie auf ein flchenstck", Mathematische Annalen **38**, 459–460, URL http://eudml.org/doc/157555 (1891).

[40] Hilbert, D., "Ueber stetige abbildung einer linie auf ein flachenstuck", Mathematische Annalen (1891).

[41] Ho, C.-T., R. Agrawal, N. Megiddo and R. Srikant, "Range queries in olap data cubes", in "Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data", SIGMOD '97, pp. 73–88 (ACM, New York, NY, USA, 1997), URL http://doi.acm.org/10.1145/253260.253274.

[42] Hong, S., B. Song and S. Lee, "Efficient execution of range-aggregate queries in data warehouse environments", in "Proceedings of the 20th International Conference on Conceptual Modeling: Conceptual Modeling", ER '01, pp. 299–310 (Springer-Verlag, London, UK, UK, 2001), URL http://dl.acm.org/citation.cfm?id=647524.725898.

[43] Huang, Q., J. Feng, Y. Zhang, Q. Fang and W. Ng, "Query-aware locality-sensitive hashing for approximate nearest neighbor search", Proc. VLDB Endow. **9**, 1, 1–12, URL http://dx.doi.org/10.14778/2850469.2850470 (2015).

[44] Indyk, P. and R. Motwani, "Approximate nearest neighbors: Towards removing the curse of dimensionality", in "Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing", STOC '98, pp. 604–613 (ACM, New York, NY, USA, 1998), URL http://doi.acm.org/10.1145/276698.276876.

[45] Jarke, M., "Common subexpression isolation in multiple query optimization", in "Query Processing in Database Systems", edited by W. Kim, D. Reiner and D. Batory, Topics in Information Systems, pp. 191–205 (Springer Berlin Heidelberg, 1985), URL http://dx.doi.org/10.1007/978-3-642-82375-6_11.

[46] Jin, R., K. Sinha and G. Agrawal, "A framework to support multiple query optimization for complex mining tasks", in "Proceedings of the 6th International Workshop on Multimedia Data Mining: Mining Integrated Media and Complex Data", MDM '05, pp. 23–32 (ACM, New York, NY, USA, 2005), URL http://doi.acm.org/10.1145/1133890.1133893.

[47] Jing, Y., C. Zhang and X. Wang, "An empirical study on performance comparison of lucene and relational database", in "Proceedings of the 2009 International Conference on Communication Software and Networks", ICCSN '09, pp. 336–340 (IEEE Computer Society, Washington, DC, USA, 2009), URL http://dx.doi.org/10.1109/ICCSN.2009.96.

[48] Joly, A. and O. Buisson, "A posteriori multi-probe locality sensitive hashing", in "Proceedings of the 16th ACM International Conference on Multimedia", MM '08, pp. 209–218 (ACM, New York, NY, USA, 2008), URL http://doi.acm.org/10.1145/1459359.1459388.

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

[50] Kaser, O., D. Lemire and K. Aouiche, "Histogram-aware sorting for enhanced word-aligned compression in bitmap indexes", in "Proceedings of the ACM 11th International Workshop on Data Warehousing and OLAP", DOLAP '08, pp. 1–8 (ACM, New York, NY, USA, 2008), URL http://doi.acm.org/10.1145/1458432.1458434.

[51] Katayama, N. and S. Satoh, "The sr-tree: An index structure for high-dimensional nearest neighbor queries", in "Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data", SIGMOD '97, pp. 369–380 (ACM, New York, NY, USA, 1997), URL http://doi.acm.org/10.1145/253260.253347.

[52] Konstantinidis, G. and J. L. Ambite, "Scalable query rewriting: A graph-based approach", in "Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data", SIGMOD '11, pp. 97–108 (ACM, New York, NY, USA, 2011), URL http://doi.acm.org/10.1145/1989323.1989335.

[53] Konstantinidis, G. and J. L. Ambite, "Optimizing query rewriting for multiple queries", in "Proceedings of the Ninth International Workshop on Information Integration on the Web", IIWeb '12, pp. 7:1–7:6 (ACM, New York, NY, USA, 2012), URL http://doi.acm.org/10.1145/2331801.2331808.

[54] Kumar, A., "G-tree: A new data structure for organizing multidimensional data", IEEE Transactions on Knowledge and Data Engineering **6**, 2, 341–347, URL http://dx.doi.org/10.1109/69.277778 (1994).

[55] Lauer, T., D. Mai and P. Hagedorn, "Efficient range-sum queries along dimensional hierarchies in data cubes", in "Proceedings of the 2009 First International Conference on Advances in Databases, Knowledge, and Data Applications", DBKDA '09, pp. 7–12 (IEEE Computer Society, Washington, DC, USA, 2009), URL http://dx.doi.org/10.1109/DBKDA.2009.18.

[56] Lemire, D., O. Kaser and K. Aouiche, "Sorting improves word-aligned bitmap indexes", Data Knowl. Eng. **69**, 1, 3–28, URL http://dx.doi.org/10.1016/j.datak.2009.08.006 (2010).

[57] Leskovec, J. and A. Krevl, "SNAP Datasets: Stanford large network dataset collection", http://snap.stanford.edu/data (2014).

[58] Liu, S., S. Poccia, K. S. Candan, G. Chowell and M. L. Sapino, "epidms: Data management and analytics for decision-making from epidemic spread simulation ensembles", The Journal of Infectious Diseases **214**, 4, 427–432 (2016).

[59] Liu, Y., J. Cui, Z. Huang, H. Li and H. T. Shen, "Sk-lsh: An efficient index structure for approximate nearest neighbor search", Proc. VLDB Endow. **7**, 9, 745–756, URL http://dx.doi.org/10.14778/2732939.2732947 (2014).

[60] Lowe, D. G., "Distinctive image features from scale-invariant keypoints", International journal of computer vision **60**, 2, 91–110 (2004).

[61] Lucene, A., "Apache lucene", http://www.http://lucene.apache.org/ (2011).

[62] Lv, Q., W. Josephson, Z. Wang, M. Charikar and K. Li, "Multi-probe lsh: Efficient indexing for high-dimensional similarity search", in "Proceedings of the 33rd International Conference on Very Large Data Bases", VLDB '07, pp. 950–961 (VLDB Endowment, 2007), URL http://dl.acm.org/citation.cfm?id=1325851.1325958.

[63] MacNicol, R. and B. French, "Sybase iq multiplex - designed for analytics", in "Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30", VLDB '04, pp. 1227–1230 (VLDB Endowment, 2004), URL http://dl.acm.org/citation.cfm?id=1316689.1316798.

[64] Management, U. P. D. and Queries, http://postgis.net/docs/using$_$postgis$_$dbmanagement.html (????).

[65] Markl, V., F. Ramsak and R. Bayer, "Improving olap performance by multidimensional hierarchical clustering", in "Proceedings of the 1999 International Symposium on Database Engineering & Applications", IDEAS '99, pp. 165– (IEEE Computer Society, Washington, DC, USA, 1999), URL http://dl.acm.org/citation.cfm?id=850953.853877.

[66] Mller, S. and H. Plattner, "Aggregates caching in columnar in-memory databases", in "In Memory Data Management and Analysis", edited by A. Jagatheesan, J. Levandoski, T. Neumann and A. Pavlo, vol. 8921 of *Lecture Notes in Computer Science*, pp. 69–81 (Springer International Publishing, 2015), URL http://dx.doi.org/10.1007/978-3-319-13960-9_6.

[67] Morton, G., *A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing* (International Business Machines Company, 1966), URL https://books.google.com/books?id=9FFdHAAACAAJ.

[68] Morzy, M., T. Morzy, A. Nanopoulos and Y. Manolopoulos, "Hierarchical bitmap index: An efficient and scalable indexing technique for set-valued attributes", in "Advances in Databases and Information Systems", pp. 236–252 (2003), URL http://dx.doi.org/10.1007/978-3-540-39403-7_19.

[69] Nagarkar, P. and K. S. Candan, "HCS: hierarchical cut selection for efficiently processing queries on data columns using hierarchical bitmap indices", in "Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, March 24-28, 2014.", pp. 271–282 (2014), URL http://dx.doi.org/10.5441/002/edbt.2014.26.

[70] Nagarkar, P., K. S. Candan and A. Bhat, "Compressed spatial hierarchical bitmap (cshb) indexes for efficiently processing spatial range query workloads", PVLDB **8**, 12, 1382–1393, URL http://www.vldb.org/pvldb/vol8/p1382-nagarkar.pdf (2015).

[71] Nam, B., H. Andrade and A. Sussman, "Multiple range query optimization with distributed cache indexing", in "Proceedings of the 2006 ACM/IEEE Conference on Supercomputing", SC '06 (ACM, New York, NY, USA, 2006), URL http://doi.acm.org/10.1145/1188455.1188560.

[72] Olma, M., F. Tauheed, T. Heinis and A. Ailamaki, "Block: Efficient execution of spatial range queries in main-memory", Technical Report EPFL (2013).

[73] OpenStreetMap, "Openstreetmap", http://www.openstreetmap.org/ (2015).

[74] Papadopoulos, A. and Y. Manolopoulos, "Multiple range query optimization in spatial databases", in "Proceedings of the Second East European Symposium on Advances in Databases and Information Systems", ADBIS '98, pp. 71–82 (Springer-Verlag, London, UK, UK, 1998), URL http://dl.acm.org/citation.cfm?id=646041.678641.

[75] Rotem, D., K. Stockinger and K. Wu, "Optimizing candidate check costs for bitmap indices", in "Proceedings of the 14th ACM International Conference on Information and Knowledge Management", CIKM '05, pp. 648–655 (ACM, New York, NY, USA, 2005), URL http://doi.acm.org/10.1145/1099554.1099718.

[76] Roussopoulos, N., "View indexing in relational databases", ACM Trans. Database Syst. **7**, 2, 258–290, URL http://doi.acm.org/10.1145/319702.319729 (1982).

[77] Roy, P., S. Seshadri, S. Sudarshan and S. Bhobe, "Efficient and extensible algorithms for multi query optimization", in "Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data", SIGMOD '00, pp. 249–260 (ACM, New York, NY, USA, 2000), URL http://doi.acm.org/10.1145/342009.335419.

[78] Samet, H., *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)* (Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005).

[79] Sellis, T. K., "Multiple-query optimization", ACM Trans. Database Syst. **13**, 1, 23–52, URL http://doi.acm.org/10.1145/42201.42203 (1988).

[80] Sellis, T. K., N. Roussopoulos and C. Faloutsos, "The r+-tree: A dynamic index for multi-dimensional objects", in "Proceedings of the 13th International Conference on Very Large Data Bases", VLDB '87, pp. 507–518 (Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1987), URL http://dl.acm.org/citation.cfm?id=645914.671636.

[81] Shastri, A., Y. Di, E. A. Rundensteiner and M. O. Ward, "Mtops: Scalable processing of continuous top-k multi-query workloads", in "Proceedings of the 20th ACM International Conference on Information and Knowledge Management", CIKM '11, pp. 1107–1116 (ACM, New York, NY, USA, 2011), URL http://doi.acm.org/10.1145/2063576.2063736.

[82] Sinha, R. R., S. Mitra and M. Winslett, "Bitmap indexes for large scientific data sets: A case study", in "Proceedings of the 20th International Conference on Parallel and Distributed Processing", IPDPS'06, pp. 68–68 (IEEE Computer Society, Washington, DC, USA, 2006), URL http://dl.acm.org/citation.cfm?id=1898953.1899001.

[83] Sinha, R. R. and M. Winslett, "Multi-resolution bitmap indexes for scientific data", ACM Trans. Database Syst. **32**, 3, URL http://doi.acm.org/10.1145/1272743.1272746 (2007).

[84] Siqueira, T. L., C. D. D. Ciferri, V. C. Times and R. R. Ciferri, "The sb-index and the hsb-index: Efficient indices for spatial data warehouses", Geoinformatica **16**, 1, 165–205, URL http://dx.doi.org/10.1007/s10707-011-0128-5 (2012).

[85] Skopal, T., M. Krátký, J. Pokorný and V. Snášel, "A new range query algorithm for universal b-trees", Inf. Syst. **31**, 6, 489–511, URL http://dx.doi.org/10.1016/j.is.2004.12.001 (2006).

[86] Stonebraker, M., D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran and S. Zdonik, "C-store: A column-oriented dbms", in "Proceedings of the 31st International Conference on Very Large Data Bases", VLDB '05, pp. 553–564 (VLDB Endowment, 2005), URL http://dl.acm.org/citation.cfm?id=1083592.1083658.

[87] Sun, Y., W. Wang, J. Qin, Y. Zhang and X. Lin, "Srs: Solving c-approximate nearest neighbor queries in high dimensional euclidean space with a tiny index", Proc. VLDB Endow. **8**, 1, 1–12, URL http://dx.doi.org/10.14778/2735461.2735462 (2014).

[88] Tao, Y., K. Yi, C. Sheng and P. Kalnis, "Efficient and accurate nearest neighbor and closest pair search in high-dimensional space", ACM Trans. Database Syst. **35**, 3, 20:1–20:46, URL http://doi.acm.org/10.1145/1806907.1806912 (2010).

[89] Thusoo, A., J. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu and R. Murthy, "Hive - a petabyte scale data warehouse using hadoop", in "Data Engineering (ICDE), 2010 IEEE 26th International Conference on", pp. 996–1005 (2010).

[90] Wang, J., H. T. Shen, J. Song and J. Ji, "Hashing for similarity search: A survey", CoRR **abs/1408.2927**, URL http://arxiv.org/abs/1408.2927 (2014).

[91] Wang, J., H. T. Shen, J. Song and J. Ji, "Hashing for similarity search: A survey", arXiv preprint arXiv:1408.2927 (2014).

[92] Weber, R., H.-J. Schek and S. Blott, "A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces", in "Proceedings of the 24rd International Conference on Very Large Data Bases", VLDB '98, pp. 194–205 (Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998), URL http://dl.acm.org/citation.cfm?id=645924.671192.

[93] Wu, K., E. Otoo and A. Shoshani, "On the performance of bitmap indices for high cardinality attributes", in "Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30", VLDB '04, pp. 24–35 (VLDB Endowment, 2004), URL http://dl.acm.org/citation.cfm?id=1316689.1316694.

[94] Wu, K., E. J. Otoo and A. Shoshani, "An efficient compression scheme for bitmap indices", Tech. rep., ACM Transactions on Database Systems (2004).

[95] Wu, K., E. J. Otoo and A. Shoshani, "Optimizing bitmap indices with efficient compression", ACM Trans. Database Syst. **31**, 1, 1–38, URL http://doi.acm.org/10.1145/1132863.1132864 (2006).

[96] Wu, K., K. Stockinger and A. Shoshani, "Breaking the curse of cardinality on bitmap indexes", in "Proceedings of the 20th International Conference on Scientific and Statistical Database Management", SSDBM '08, pp. 348–365 (Springer-Verlag, Berlin, Heidelberg, 2008), URL http://dx.doi.org/10.1007/978-3-540-69497-7_23.

[97] Zaker, M., S. Phon-amnuaisuk and S. cheng Haw, "An adequate design for large data warehouse systems: Bitmap index versus b-tree index", (2008).

[98] Zhang, W., K. Gao, Y.-d. Zhang and J.-t. Li, "Data-oriented locality sensitive hashing", in "Proceedings of the International Conference on Multimedia", MM '10, pp. 1131–1134 (ACM, New York, NY, USA, 2010), URL http://doi.acm.org/10.1145/1873951.1874168.

[99] Zheng, Y., Q. Guo, A. K. Tung and S. Wu, "Lazylsh: Approximate nearest neighbor search for multiple distance functions with a single index", in "Proceedings of the 2016 International Conference on Management of Data", SIGMOD '16, pp. 2023–2037 (ACM, New York, NY, USA, 2016), URL http://doi.acm.org/10.1145/2882903.2882930.

[100] Zhong, Y., J. Han, T. Zhang, Z. Li, J. Fang and G. Chen, "Towards parallel spatial query processing for big spatial data", in "Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum", IPDPSW '12, pp. 2085–2094 (IEEE Computer Society, Washington, DC, USA, 2012), URL http://dx.doi.org/10.1109/IPDPSW.2012.245.

[101] Zhou, J. and K. Ross, "A multi-resolution block storage model for database design", in "Database Engineering and Applications Symposium, 2003. Proceedings. Seventh International", pp. 22–31 (2003).

APPENDIX A

SAMPLE RANGE QUERY WORKLOAD IN 2D SPACE

In this Appendix, I present sample data (Table A.1) and a sample range query workload (Table A.2) used in Chapter 4. Let us consider a range query workload Q that consists of 3 rectangular spatial range queries (q_1, q_2, q_3).

ID	xCoordinate	yCoordinate
1	50.2	62.8
2	32.5	16.4
3	12.6	41.3
4	53.1	87.6
5	65.2	10.5

Table A.1: A sample Table *PointsData* that consists of 2D points

As described in Section 4.2.1 of Chapter 4, the range specification of these queries is defined by a south-west point (q_i^{sw}) and a north-east (q_i^{ne}) point.

QueryID	sw.xCoor	sw.yCoor	ne.xCoor	ne.yCoor
1	50.0	50.0	60.0	90.0
2	40.5	52.8	62.4	73.4
3	45.5	5.8	68.4	70.3

Table A.2: Sample Range Queries and their Range Query Specifications

Let us consider the sample range specifications as presented in Table (Table A.2). For each query q_i in Q, the goal is to find IDs of points that would satisfy the following SQL query:

select ID **from** PointsData
where xCoordinate $\geq q_i^{sw}.xCoor$ AND xCoordinate $\leq q_i^{ne}.xCoor$
AND yCoordinate $\geq q_i^{sw}.yCoor$ AND yCoordinate $\leq q_i^{ne}.yCoor$

In the above example, q_1 would return Point 1 and Point 4, q_2 would return Point 1, whereas q_3 would return Point 1 and Point 5.