FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Randomized Indexing for Approximate Selection Queries on Multidimensional Arrays

by

Luboš Krčál

A dissertation thesis submitted to
the Faculty of Information Technology, Czech Technical University in Prague,
in partial fulfilment of the requirements for the degree of Doctor.

Dissertation degree study programme: Informatics
Department of Theoretical Computer Science

Prague, August 2021

Supervisor:

Prof. Ing. Jan Holub, Ph.D.
Department of Theoretical Computer Science
Faculty of Information Technology
Czech Technical University in Prague
Thákurova 9
160 00 Prague 6
Czech Republic

Abstract

Multidimensional data, either in the form of dense arrays, or sparse relational data are a common data structure for effective storage, access, management, querying, disseminating, analysis, and visualization of scientific datasets. Array data are being used in many scientific domains, including computational fluid dynamics, oceanography, spatiotemporal climate analysis, forecasting, medical, biomedical, astronomical and satellite data processing. Efficient processing of high dimensional data is difficult due to the arbitrary size, cardinality, and so called curse of dimensionality.

Bitmap indices are widely used in commercial databases for processing complex queries, due to the efficient use of hardware accelerated bit-wise operations and their space-efficiency. Compressed, hierarchical, multi-component bitmap indices have also been used for relational data.

Inverted indexing is another commonly used technique in a variety of high dimensional data applications, such as exact search, similarity search, or machine learning. Inverted index maps multidimensional data points into lists based on some discretization of their dimension values. Similarly, a column index of a relational database may map individual column values to lists of corresponding rows. To evaluate a query, inverted lists are usually intersected to obtain a list of points satisfying all the constraints. Our interest is in a more generalized approach, where each query evaluates similarity based on the number of matched dimensions.

In this work, we have designed, implemented and evaluated two methods of indexing multidimensional array data for selection queries, and an extension of data parallel inverted index as part of generic framework for similarity search on the GPU. Following is a list of individual contributions:

For the purpose of efficient execution of various spatiotemporal selection queries in large distributed array databases, we have designed a multidimensional array inverted index based on grid transformations. We demonstrate the efficiency of our multidimensional array index on a complete, large-scale satellite dataset. The work was implemented and integrated as an extension of a distributed open-source array database SciDB.

Next, we have proposed a hierarchical indexing scheme for multidimensional arrays that overcomes the dimensionality-induced inefficiencies of standard spatial and bitmap indexing techniques on dense multidimensional arrays. The index is based on novel n-dimensional sparse trees for dimension partitioning, with bound number of individual, adaptively binned indices for attribute partitioning. This indexing performs well on queries involving both dimensions and attributes constraints, as it prunes the search space early.

Lastly, we have improved query performance of generic similarity search in *GENIE* (Generic inverted index on GPU) by incorporating compressed inverted index on GPU with data parallel decoding. Multiple decoding schemes were designed, implemented, and evaluated for a fully data parallel decoding and query execution. The implementation has sped up total query processing time in 3-4 times on real world datasets. All the components were integrated into publicly available similarity search framework GENIE in a robust and modular architecture, with configurable query compiler and index management components. The extensions of GENIE were designed for multi-GPU and multi-node distributed deployment with an implementation of the distributed functionality publicly available.

Keywords:
Multidimensional Arrays, Array Database, SciDB, Scientific Computing, Bitmap Index, Inverted Index, Similarity Search, Approximate Nearest Neighbors, GPU Accelerated Database, Index Compression, Data Parallel Decoding, GENIE

Acknowledgements

First and foremost, the completion of my dissertation would not be possible without my supervisor, professor Jan Holub, who has been guiding me throughout my research track for the last almost 10 years. He started as my master's thesis supervisor, then continues as my doctorate supervisor and the principal investigator of several research projects. He has always shown relentless support, exceptional stability and resolve, while providing me with enough freedom to explore related research topics of interest.

I would also like to express deep gratitude to my supervisor at Nanyang Technological University in Singapore, professor Shen-Shyang Ho, who has introduced me to the research environment in Singapore. Shen-Shyang always had constructive advice, and always displayed great compassion, support, and guidance. He also went far out of his way to help me find more research opportunities after his departure from Singapore.

Many thanks go to all my research colleagues at NTU: Jianjun Zhao, Tianyi Zhou, Pei-Hung Chen, Woon Huei Chai, Thet Mon Htwe, Tzu-Yi Hung, Vidhya Natarajan, and Cheng Seng Low. And to my friends from the CIR lab: Tim Muller, Ali Alizadeh Mansouri and Sun Zhu. You all have created an amazing and friendly work environment. Additionally, I would like to also thank the ACM ICPC head coaches and coordinators, Rui Fan and Kevin Anthony Jones, with whom we traveled to many programming competitions.

My gratitude also goes to my supervisors at National University of Singapore, professor Anthony Tung and professor Bingsheng He. Anthony has shown great leadership, support and valuable scientific insight. Bingsheng has forever motivated me with his outstanding ambition and exceptional knowledge. Both Anthony and Bingsheng have inspired me with their focus and intellectual capability, and it would have been great honor for me to keep working with them had I stayed in Singapore longer.

I also had great pleasure of working with my colleagues from NUS: Yuxin Zheng, Guo Qi, Yifan Lei, Fei Wang, Pingyi Luo, Jisong Yang, Christian von der Weth, Yueji Yang, Siyuan Liu, and Jae Ramon Bespinyowong. Thank you for all the support, awesome work environment, insightful lunches, and for all the fixes you did after I repeatedly broke the servers.

Big thank you to my colleagues from our research group at the Czech Technical University in Prague, namely Ondřej Cvacho and Petr Procházka. I am grateful for the opportunity to expand my knowledge and learn about other related topics from you.

Many thanks to my friends from Nyriad, New Zealand, especially my former seniors, John Mackenzie and Daein Choi, and colleagues Mitch Turnbull and Robbie Litchfield, who all supported me in one way or another after we departed the company. I have learned a great deal from my managers as well. Thank you for the experience.

Last, I would also like to thank my current coworkers from Edgeworx, USA, namely Rashmi Modhwadia, Christina Dang, Alex de Wergifosse, Serge Radinovich, Neha Naithani, Saeid Rezaei Baghbidi, Todd Papaioannou and Kilton Hopkins. With some I worked longer than with others, but I have learned something from everyone.

Throughout most of my doctorate studies, I have found reprieve in practicing martial arts. And I would like to thank my Brazilian jiu-jitsu coaches: Christian Rodrigues, Vincent Tan, Harvey Skinner and Errol Watson, especially for teaching me persistence and never quitting on my long term goals. Having the opportunity to teach kids classes myself, I have made an observation, that just a bit of focus and effort at the right time makes a huge difference.

Honorary mention at the end goes to Libor Buš, my former manager at Eccam in Prague. Unbeknownst to him, inspired me to pursue the highest form of education. Libor got his Ph.D., established a company, and started a family all at the same time, while preserving his humble attitude.

This research has been partially supported by the following:

- Czech Science Foundation (GAČR) project No. GA13-03253S (*Text and Tree Structures Processing and Their Applications*) by Ministry of Education, Youth, and Sport of the Czech Republic
- SGS project *SGS14/101/OHK3/1T/18* by Czech Technical University in Prague, Czech Republic
- Academic Research Fund (AcRF) Tier 1 Grant RG-18/14 (*Array-Based Database Technology for Large-Scale Satellite Data and Their Analysis*) by Ministry of Education, Singapore; at School of Computer Science and Engineering (SCSE), Nanyang Technological University (NTU), Singapore
- National Research Foundation Singapore under Indirect Research Cost (IRC) Funding at Interactive Digital Media Institute (IDMI), National University of Singapore (NUS), Singapore

Contents

Abstract **iii**

1 Introduction **1**
1.1 Contributions . . . 2
1.2 Structure of The Document . . . 3

2 Background and State-of-the-Art **5**
2.1 Mutlidimensional Array Model . . . 5
2.1.1 Multidimensional Selection Queries . . . 5
2.2 Multidimensional Spatial Indexing . . . 6
2.2.1 Point Access Methods . . . 7
2.2.2 Spatial Access Methods . . . 8
2.2.3 Tilings of the Domain Space . . . 10
2.2.4 Space-Filling Curves . . . 10
2.3 Spatial Indexing of Arrays . . . 11
2.3.1 Array Indexing of Arrays . . . 12
2.3.2 Tree Indexing of Arrays . . . 12
2.3.3 Tree Indexing of Array Blocks . . . 13
2.3.4 Indexing of Triangular Blocks on a Sphere . . . 13
2.4 Bitmap Indexing . . . 14
2.5 Bucket Indexing Methods . . . 16
2.6 Similarity Hashing . . . 16
2.7 Spatiotemporal Indexing and Databases . . . 16
2.8 Approximate Array Pattern Matching . . . 17

3 Inverted Regridding Indexing **19**
3.1 Indexing Spatiotemporal Data . . . 20
3.1.1 Regridding Index . . . 20

3.1.2 Cartesian Index 21
3.1.3 Hierarchical Structure of Indices 21
3.2 Use Case Scenario: Select QuikSCAT data given tropical cyclone trajectory 22
3.3 Retrieving Data Regions – Pointers into the original data 22
3.4 Time Complexity 23
3.5 Conclusion 24

4 Hierarchical Bitmap Index 25
4.1 Related Work and Previous Results 25
4.2 Problem Statement 27
4.3 Description of Hierarchical Bitmap Array Index 27
4.3.1 Partitioning of Arrays 28
4.3.2 Structure of the Array Chunk Index 29
4.3.3 Structure and Construction of the Hierarchical Bitmap Array Index 29
4.3.4 Bin Boundaries Merging in Parent Nodes 30
4.3.5 Double Range Encoding of Bitmap Indices in Internal Nodes 32
4.3.6 Locality of the Hierarchical Index 33
4.3.7 Appending and Modifying Data 34
4.4 Querying Dimensions And Attributes 34
4.4.1 Attribute based Matches 35
4.4.2 Dimension based Matches 36
4.4.3 Partial and Complete Matches 37
4.4.4 Estimating Cardinality of Results; Membership Queries 39
4.5 Experimental Evaluation 39
4.5.1 Fastbit Integration 40
4.5.2 Bitmap Indexing Methods 40
4.5.3 Range Queries 40
4.5.4 Parameterization 42
4.6 Conclusion 42

5 Compressed Inverted Index on GPU 43
5.1 Introduction 44
5.2 Related Work 46
5.2.1 Integer Lists Compressions 47
5.2.1.1 Integer Lists Compressions on GPUs 50
5.2.1.2 Integer Lists Compressions on FPGAs 50
5.2.2 Database Acceleration using GPUs 50
5.2.3 Exact Similarity Search on CPUs and GPUs 51
5.2.3.1 Similarity Search Using Heterogeneous Architectures . . . 53
5.2.4 Approximate Similarity Search 54
5.2.4.1 Approximate Similarity Search on GPUs 55
5.2.4.2 Locality Sensitive Hashing 55
5.2.4.3 Locality Sensitive Hashing on GPU 56

5.2.5 GPGPU and CUDA 56
5.3 GENIE 57
5.3.1 Problem Statement 58
5.3.2 Using LSH For Approximate Nearest Neighbors 61
5.3.2.1 Re-Hashing 63
5.3.3 Examples of LSH Function Families 63
5.3.4 Preprocessing Data Using Other Sources 64
5.4 Compressed Match Counting in GENIE 65
5.4.1 CUDA Kernel Models For Decompression 65
5.4.1.1 Delta Encoding 65
5.4.1.2 Varint Encoding 66
5.4.1.3 Bitpacking Encoding 67
5.4.1.4 Composite Codec 72
5.4.1.5 Parallel Models for Decoding 72
5.4.2 Further Reducing Time by Suppressing Multi-Load 74
5.4.3 Efficient Inverted Lists Balancing 74
5.4.4 Integration Into GENIE 75
5.5 Experimental Evaluation 76
5.5.1 Description of Datasets 77
5.5.2 Environment And Settings 78
5.5.3 Analysis of Inverted Tables 78
5.5.4 Analysis of Individual Compression Schemes 79
5.5.5 Efficiency of Inverted List Encoding 80
5.6 Conclusion 83
5.6.1 Future Work 84
5.6.2 Acknowledgement 85

6 Conclusion **87**
6.1 Future Work 88

References **89**

Publications of the Author **109**

List of Figures

2.1 An example of a range query on a two-dimensional array. 6
2.2 Basic quadtrees for indexing point data in two dimensional space. 7
2.3 Hierarchical taxonomy of point access structures developed in [174]. 8
2.4 Quadtrees for indexing line segments and rectangular objects. 9
2.5 Planar decomposition of two dimensional space into recursive patterns. 10
2.6 Space filling curves on a [4^4] tiling – two dimensional array. 11
2.7 Hierarchical triangular mesh. 14
2.8 Bitmap index for attribute `a` of the array `A` from Figure 2.1: empty bitmask `EBM`, equality encoded index `E`, range encoded index `R` and interval encoded index I. 14
2.9 An example of World Aligned Hybrid (WAH) bitmap compression. 15

3.1 Array regridding scheme used during incremental generation of Latitude-Longitude-Time index. 19
3.2 Trajectory and a corresponding mask – sequence of hyperrectangles on latitude, longitude and time dimensions. 22

4.1 Example of merging $|B| = 8$ bin boundaries to $|R| = 4$ bin boundaries for 4 child nodes 31
4.2 Processing of a query in a single node of the hierarchical index. 38
4.3 Query execution time and disk space required to store the indices for different array sizes. 41
4.4 Query execution time for 2D, 3D and 4D queries of various hit ratios. 41

5.1 Integer lists encoding and decoding pipeline (with linear delta encoding and Bitpacking32) in inverted index setting. 47
5.2 Generic nature of GENIE, showing the relationship with different data types. . 58
5.3 Example of (d_1, d_2, p_1, p_2)-sensitive hash function. 62
5.4 Example of locality sensitive hashes based on p-stable distribution in 2 dimensional space. 64

5.5 Visualization of Parallel Varint decoding 68
5.6 Visualization of parallel Bitpacking32 decoding 70
5.7 High level parallel execution model of a query set. 73
5.8 Distribution of inverted list lengths across the datasets. 79
5.9 Query time analysis of GENIE with various integrated (encoding and matching) kernels. 82
5.10 Memory usage analysis of GENIE with various integrated (encoding and matching) kernels. 83

List of Tables

3.1 Timing results of a query - Select QuikSCAT Data Along Trajectory 23

5.1 Sample dataset and queries represented by 5 dimensional objects. 59
5.2 Example of inverted index stored in GENIE. 60
5.3 Summary of inverted index for each dataset. 78
5.4 Codec comparison of compression ratio (bits per integer), encoding and decoding times (s). 80
5.5 Query time and memory usage measurements of GENIE with various integrated (encoding and matching) kernels. 81

List of Algorithms

4.1 Iterative equi-depth binning approximation 32
4.2 Evaluation of partial and complete match bitmaps for a single node. 35

5.1 Parallel Varint decoding implementation 69
5.2 Parallel Bitpacking32 decoding implementation 71

Chapter 1

Introduction

Multidimensional arrays are a common data structure for effective storage, access, management, querying, disseminating, analysis, and visualization of scientific datasets [213].

Arrays are also often referred to as *raster data*, *gridded data*, or *datacubes*. Multidimensional array data are being used in many domains, including computational fluid dynamics, oceanography [197], spatiotemporal climate analysis [119], hurricane forecast [184], medical [19], biomedical [178], astronomical [208, 193, 206], and satellite [84, 85, 165, 55] data processing.

Multidimensional arrays are also widely used as an access structure for general spatiotemporal data. Many other problems can be transformed into lower dimensional structures using locality preserving transformations, such as locality sensitive hashing, most often for the purpose of location based selection queries or similarity search [174].

Efficient processing of multidimensional data is difficult due to the arbitrary size, dimensionality and cardinality of the arrays. Large spectrum of array processing tasks consists of data selection and querying (“subseting”) [84], aggregating data, searching based on template patterns [15] or aggregate constraints [215], contrast set discovery [235], spatial pattern mining [83], and general computation [196].

Most of these array processing tasks are computationally very intensive and thus require some form of index structure in order to increase the access efficiency. Furthermore, the size of array data in scientific application often reaches the order of terabytes, effectively enforcing usage of distributed storage and parallel processing.

Multidimensional array storage and processing is implemented in distributed open-source array-based data management and analytics systems, examples of which are RasDaMan [18], SciDB [196], and MonetDB [90]. Array databases provide extensive data processing and frameworks with storage and computational functionality built natively on top of arrays, compared to their more general distributed counterparts such as MapReduce [55, 47] and SciHadoop [32, 217].

Inverted indexing is a commonly used technique in a variety of selection query or search applications. Examples of large data intensive applications come from astronomy and astrophysics [87], finance [43] neuroscience, engineering, multimedia and others [236].

In a domain, where each document consists of a list of terms, inverted index maps the individual terms to documents using lists of document identifiers. Similarly, a column index of a relational database may map individual column values to a list of corresponding rows.

Multidimensional data points can be also seen as column values (where each column corresponds to a single dimension), and thereby indexed using inverted indexes. To evaluate a query, inverted lists are usually intersected to obtain a list of points satisfying all the constraints. Our interest is in a more generalized approach, where each query evaluates similarity based on the number of matched dimensions.

We use a distributed open source array database SciDB as the platform for our work on multidimensional array indexing and searching.

For the purpose of inverted indexing research on more generic multidimensional data, we use a publicly available, GPU-accelerated generic similarity search framework *GENIE* (Generic inverted index on GPU). Additionally, we extend GENIE with multiple inverted index encoding schemes in order to speed up execution on large datasets.

1.1 Contributions

We have proposed, designed, implemented and tested two methods of indexing multidimensional array data for selection queries, and an extension of data parallel inverted index as part of generic framework for similarity search on the GPU. Following is a list of individual contributions:

- We have designed and implemented a multidimensional array inverted index based on grid transformation. The index allows for efficient execution of various spatiotemporal selection queries.

- We have demonstrated the efficiency of the multidimensional array index on a complete dataset of large-scale satellite sensor data (QuikSCAT). Given a trajectory query into a satellite sensor data, we perform accurate data retrieval of relevant regions.

- The work was implemented (including visualizations) and integrated as an extension of the distributed open-source array database SciDB.

- We have proposed and implemented a multidimensional array hierarchical indexing scheme that overcomes the dimensionality-induced inefficiencies.

- The indexing scheme is based on a novel n-dimensional sparse trees for dimension partitioning, with bound number of individual, adaptively binned indices for attribute partitioning.

◦ We have improved query performance of generic high dimensional data similarity search in *GENIE* (Generic inverted index on GPU) by incorporating compressed inverted index, query compiler, and data parallel decoding on GPU.

◦ Multiple decoding schemes were designed, implemented, and evaluated for fully data parallel decoding and query evaluation. We use heuristics for encoding selection based on the properties of the dataset, and properties of the inverted lists.

◦ This data parallel decoding and query evaluation have sped up total query processing time in GENIE 3-4 times on real world datasets.

◦ All the components were integrated into the publicly available framework GENIE in a robust and modular architecture, with configurable query compiler and index management components.

◦ The extensions of GENIE were designed for multi-GPU multi-node distributed deployment with initial implementations of the distributed functionality publicly available.

1.2 Structure of The Document

This Chapter 1 (*Introduction*) goes over the general problem and objectives of this thesis, describes the motivation behind our effort, our goals, and main contributions.

Chapter 2 (*Background and State-of-the-Art*) provides an in-depth survey on several topics closely related to the subsequent chapters. These topics are categorized and structured according to general research areas and their influence on the research work.

Following are three chapters, each focused on one original research topic. Each of the following chapters states the motivation, contributions, and options for future work in the respective area.

Chapter 3 (*Inverted Regridding Indexing*) describes our work on inverted grid indices with application in effective execution of various spatiotemporal selection queries on satellite data. This work has been published in [A.1]. This work was part of a project *Array-Based Database Technology for Large-Scale Satellite Data and Their Analysis* and had been carried out in its entirety at the Computational Intelligence Laboratory (CIL), School of Computer Science and Engineering (SCSE), Nanyang Technological University (NTU), Singapore.

Chapter 4 (*Hierarchical Bitmap Index for Range and Membership Queries on Multidimensional Arrays*) proposes a novel method for multidimensional array indexing called *ArrayBit* that overcomes the dimensionality-induced inefficiencies. This work has been published in [A.4]. This work had been carried out in its entirety at the Computational Intelligence Laboratory (CIL), School of Computer Science and Engineering (SCSE), Nanyang Technological University (NTU), Singapore.

Chapter 5 (*Similarity Search Using Compressed Inverted Lists on Graphic Processing Units*) describes an approach on indexing large high-dimensional datasets for similarity search queries using compressed inverted list compression on GPUs. Part of this work has been published in the original GENIE paper [A.2], in extended technical report [A.5]. The remaining part is primarily focused on inverted index compression and data parallel decoding and matching is described in this chapter. This research work, including GENIE, had been partially carried out at the SeSaMe Centre, under Interactive Digital Media Institute (IDMI), National University of Singapore (NUS), Singapore.

The last Chapter 6 (*Conclusion*) summarizes the results of our research, suggests possible topics of further research, and concludes the report.

Chapter **2**

Background and State-of-the-Art

2.1 Mutlidimensional Array Model

An array A consists of *cells* with *dimensions* indexed by $d_1, \ldots, d_n$. Each cell is a tuple of several *attributes* $a_1, \ldots, a_m$. We assume the structure of the attributes is the same for all cells in the array. The array is denoted as $A\langle a_1, \ldots, a_m\rangle[d_1, \ldots, d_n]$. For example, satellite data may have latitude, longitude, altitude, and time as dimensions, and precipitation, temperature, wind speed, etc. as attributes.

Due to the large size of scientific data, it is often necessary to split the data into subarrays called *chunks*.

There are two commonly used strategies. Regularly gridded chunking, where all chunks are of equal shape and do not overlap. This array data model is known in SciDB as MAC (Multidimensional Array Clustering) [196]. This array model works well for coarse dimension-based queries, but requires either additional indexes or filtering for fine dimension-bases and for any attribute-based queries. This array data model is the foundation (the lowest level) of our hierarchical bitmap array index. The second strategy is irregularly gridded chunking, which is one of the chunking option in RasDaMan [18].

2.1.1 Multidimensional Selection Queries

Focusing on multidimensional arrays, we first describe selection queries, where our objective is to design a family of indexing schemes for multidimensional array data that allows for fast selection queries and general computational operations on the results.

First, we define the multidimensional query with dimension and attribute constraints. Second, we describe our array regridding based approach resembling inverted indexing of array ranges.

The selection query parameters are a set of dimension values or ranges; dimension result range constraints; attribute values or ranges; attribute template patterns, including sparse and don't care cells, with distance measure definition and corresponding value; and aggregate conditions.

Computational operation run on top of selection results include result aggregations, exploratory tasks and general purpose array operations.

We form a *query* on arrays based on *constraints*. A dimension and attribute constraint is a constraint on a dimension and attribute in one of the following formats. A one-sided range query: $y \leq 45$; two-sided range query: $23.4 \leq y < 73.2$, equality query: $y = 89$; membership query: $y \in \{2, 4, 6, 8, 10\}$, where y is either dimension or attribute of the array. Figure 2.1 shows a query that has a two-sided constraint on an attribute a and a one-sided constraint on dimension d_2 on a 2-dimensional array and the (shaded) query outcome. Note that equality query is a special case of membership query, and that all queries can be rewritten to a set of range queries. *Mixed queries* are queries that pose constrains on at least one dimension and one attribute.

An example query on array SatelliteArray ⟨snowfall, rainfall, temperature⟩ [latitude, longitude, altitude, time] may look like this:
`SELECT * FROM` SatelliteArray `WHERE` $50.68 \leq latitude \leq 50.88$ `AND` $14.37 \leq longitude \leq 14.57$ `AND` $30.0 \leq snowfall$.

The result would then be a possibly empty subarray of the same format as SatelliteArray.

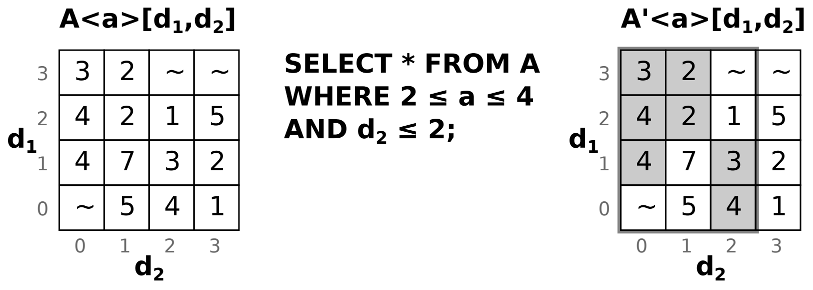


Figure 2.1: An example of a range query on a two-dimensional array.

2.2 Multidimensional Spatial Indexing

Traditional spatial indexing works with multidimensional spatial points or spatial regions are a more common setup in many past applications, due to their convenient representation of location (arbitrary precision) and their sparse nature. Majority of the indexing methods use trees that recursively split the domain space. Indexing methods described in this section are not directly applied in indexing arrays, but are a prerequisite.

Based on the data and application, index structures can be divided into *point access methods* for indexing multidimensional points only, and *spatial access method* for indexing non-zero objects, e.g. regions, polygons, etc. We will first describe point access methods.

2.2.1 Point Access Methods

The most frequently used one-dimensional indexing method is the *B-tree*. Its variant, *B^+tree* is used in majority of database management systems. The *multidimensional B-tree* [180, 75] is a hierarchy of interconnected B-trees to each dimension.

Multidimensional range tree [23] is an embedding of d binary range trees (i.e. binary tree of binary range trees for 2 dimensions). The leaves to binary range search trees represent data points sorted by a double linked list.

Quadtrees [58, 171, 172] are simple two dimensional 4-ary trees, which adaptively decompose the domain space into a grid. There are two major categories of quadtrees: *point quadtrees* and *trie-based* quadtrees. The former uses data points as internal nodes, thereby splitting the domain space according to the location of data points; whereas the latter uses data independent internal node boundaries, i.e. equal size grid in the simplest form. See Figure 2.2 for an example. Mutlidimensional extension of quadtrees are known as d-dimensional 2^d-ary trees. Three dimensional tree is called octree.

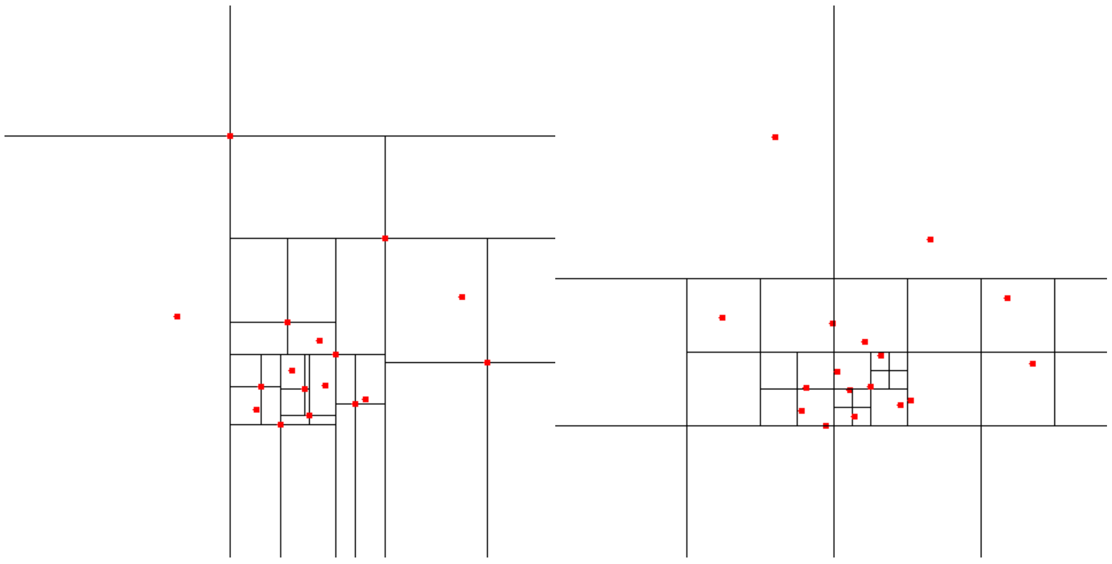

(a) Point quadtree (b) PR quadtree

Figure 2.2: Basic quadtrees for indexing point data in two dimensional space. Note that in PR quadtree, the space delimiters are at fixed positions, while in point quadtree, the delimiters are aligned with data points.

KD-trees [22] are de facto a modification of quadtrees. Since quadtrees have exponential fanout based on the dimensionality – 2^d, they tend to have many empty cells. K-d trees are binary trees that at each level split the space along a single dimension. There are many variations of k-d trees, most notable *point k-d trees* are adaptive k-d tree, which store data only in leaves, thereby having the opportunity to freely move dimension boundaries, e.g. by median [59]; or bucket k-d trees [135]. Similarly to quadtrees, *trie-based* k-d trees exist as well. The most common unbalanced trie-based k-d tree is PR k-d trie [150].

There are improvements over the PR k-d trie in terms of bucketing, moving dimension boundaries (instead of always placing them in a fixed position), and many more. Interesting modification of k-d trie is the *balanced box decomposition tree* [12], which uses two types of of internal nodes: split (tradition k-d tree node) and shrink (hypercube bounding box for child node, and a remainder for the second child).

R-trees [77], R*-trees [20] are another elementary data structure. R-trees are based on hierarchical structure of minimum bounding boxes (regions). R-trees and their variations can be used either as a point access method or a spatial access method, indexing hyperrectangles or polygons. There are plenty many indexing methods that use the idea of *buckets* – storing a list of values within a single node or leaf of the tree. These methods are used especially in relation to a slower data storage medium. Their description is out the scope of this work.

For an overview of point access method and their hierarchical organization, see Figure 2.3.

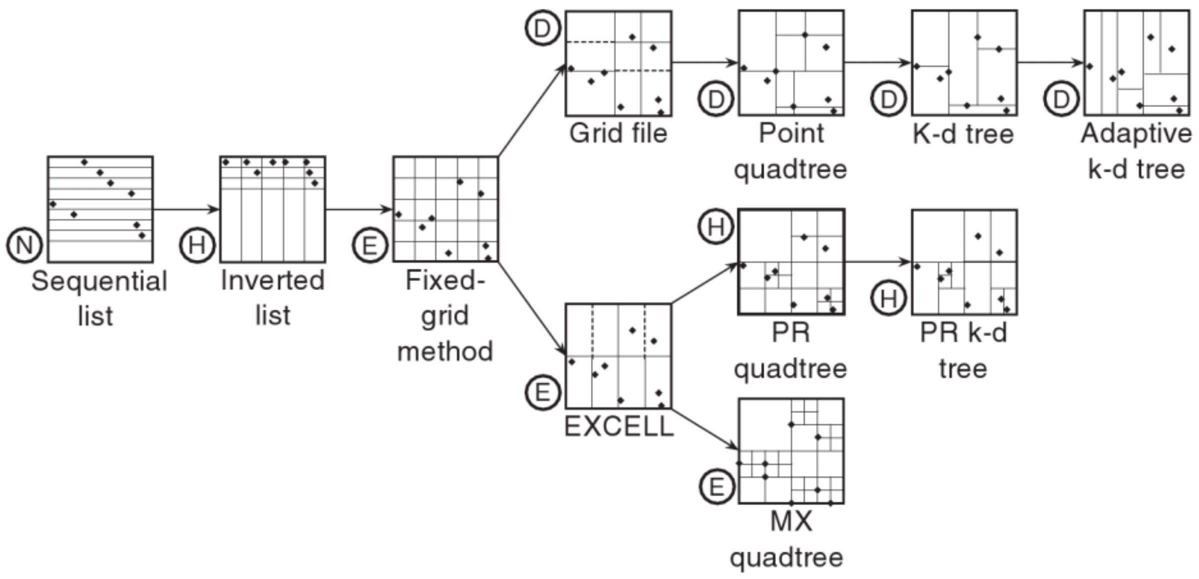


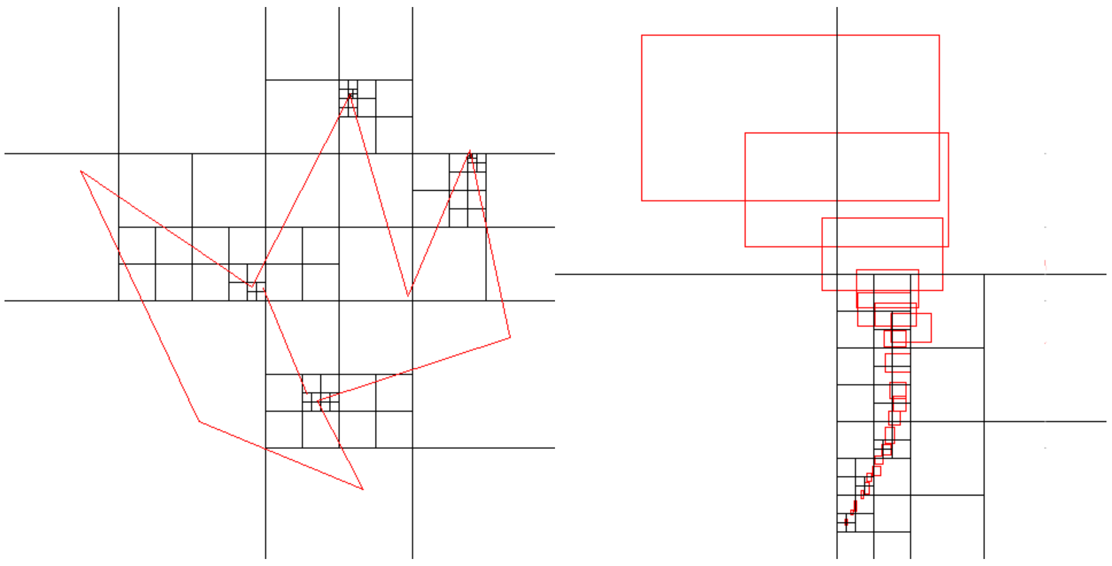
Figure 2.3: Hierarchical taxonomy of point access structures developed in [174]. Letters at each node represent: D – organize data based on its values; E – organize the embedding space from which data is drawn; N – organize neither; H – hybrid of two form D, E or N. The depth in the taxonomy tree also represents how flexible and adaptable the data structure is. Figure taken from [174].

2.2.2 Spatial Access Methods

Now we will describe some basic spatial access methods. These methods are used for indexing more complex spatial objects than point, such as lines, rectangles, polygons, objects with boundaries aligned with hyperplanes of the domain space or objects with arbitrary boundaries. There are also two main methods of object representation: *interior-based* and boundary-based representation, as identified in [174]. Based on the complexity and alignment of indexed objects, we may be able to decompose the object into cells whose boundaries are parallel to the coordinate axes of our domain space. In case this does not hold, in many cases there are representation of such objects based on decompositions into cells whose boundaries are parallel to the coordinate axes.

Basic boundary representations are used for lines and curves. Compared to R-trees, which use coordinate aligned bounding hyperrectangles, a *strip tree* [16] consists of a hierarchy of arbitrarily aligned hyperrectangles enclosing given lines and curves. The strip tree is a binary tree built top-down by partitioning the bounding boxes at each level in the points where the curve touches the bounding box. *Arc tree* [73] builds a complete binary tree by splitting the curve into line length segments, then enclosing these segments in ellipses.

Examples of quadtree based methods for lines and regions indexing consist of *MX quadtrees* [88] which approximate the line segments by cells the segments intersect; and a family of PM quadtrees [144]. See Figure 2.4 for examples of MX and PM quadtrees.



(a) PM_1 quadtree (b) MX quadtree

Figure 2.4: Quadtrees for indexing line segments and rectangular objects. PM quadtrees (and MX quadtrees) are used for indexing line segments. The condition to split internal nodes is the presence of multiple segments, unless they are incident at the same vertex. MX-CIF quadtrees are used for indexing of relatively small rectangular objects. These rectangles are always associated with a single node of the tree.

For full overview of both points access methods and spatial access methods, refer to complete surveys [60, 174, 175].

2.2.3 Tilings of the Domain Space

A common modeling approach is a decomposition of the domain space into a regular structure of unit-size cells. Ideally, such structure should be infinitely repetitive and recursively decomposable. Most of the previously described methods used some form of partitioning of the domain space. The time complexity, space requirements and accuracy of

Large number of polygonal tilings is described in [21]. See Figure 2.5 for examples. The tiling are distinguished based on their ability to decompose recursively infinitely – *unlimited*, versus *limited*tilings that don't have this property. If a tiling has the same shape on all the levels, it's called *similar*.

In our work, we focus on the $[2d^{2d}]$ tiling, i.e. a regularly gridded space, which is both unlimited and similar.

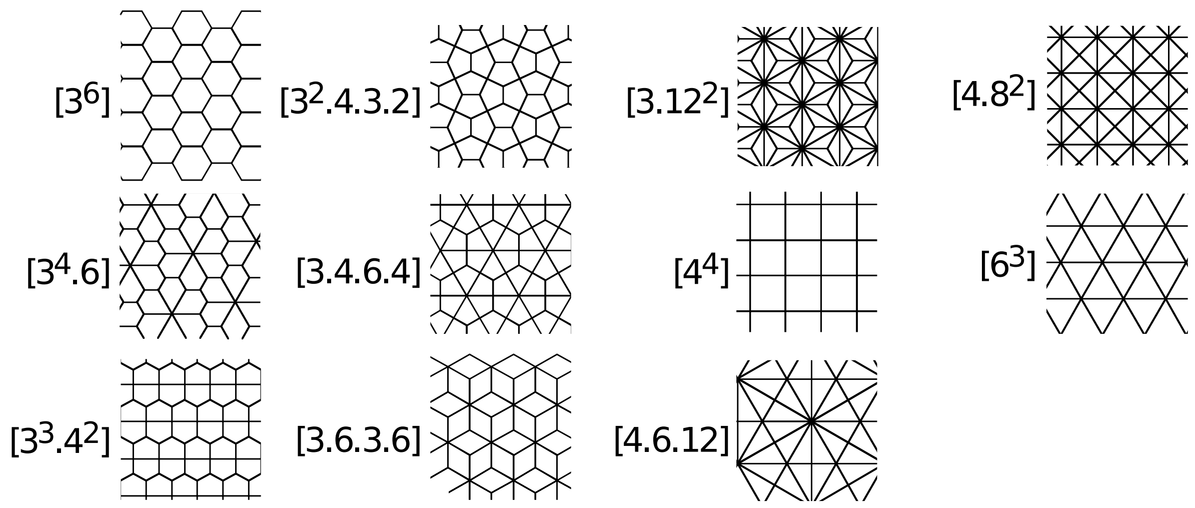


Figure 2.5: Planar decomposition of two dimensional space into recursive patterns. The array model is $[4^4]$ in this figure. There are altogether 11 types of different adjacency structure. The notation $[3^4.6]$ means the first 4 vertices are incident with 3 edges and the fifth vertex is incident with 6 edges. Figure modified from [176].

2.2.4 Space-Filling Curves

Space-filling curves [168] are a tool to linearize / map addressed from d-dimensional space into a linear space. There are many differed space-filling curves, but the most notable include C-style row-major order, row-prime order, Z-order (Morton) [150], Peano-Hilbert (also knows as Hilbert) [82], Cantor-diagonal order, spiral order, Gray order, double Gray order, or most recently U-order [122].

An important property of space filling curves is the computation time needed to transform the higher dimensional space into an integer – single dimensional space on the curve.

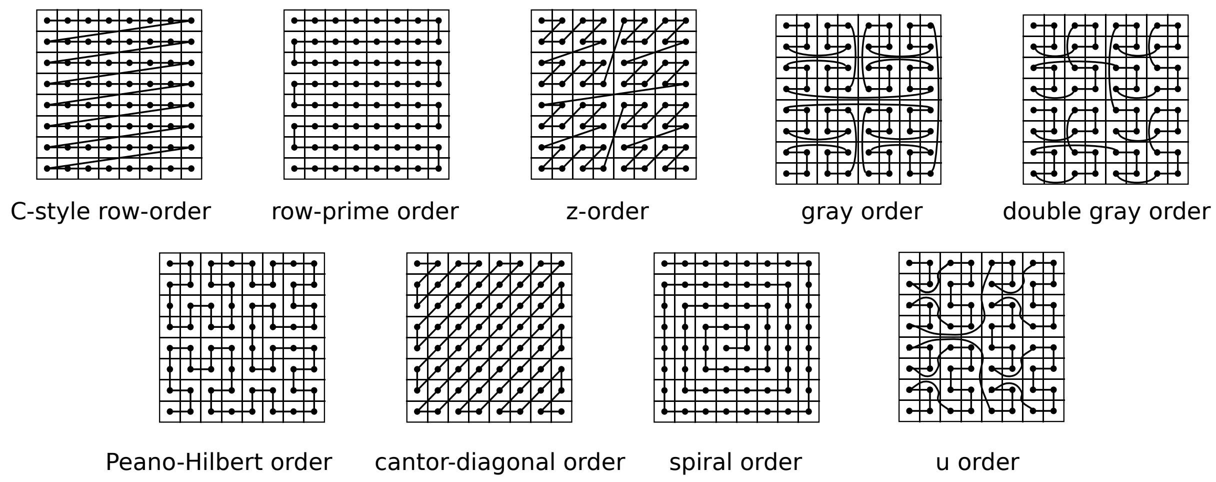


Figure 2.6: Space filling curves on a [4^4] tiling – two dimensional array. Figure modified from [176].

All the transformations are relatively simple and fast to compute, except for the Peano-Hilbert curve.

Another important property is a *stability* through different levels of the curves. A stable order preserves the relative ordering of the individual locations when the resolution is doubled (for each dimension). For example Z-order , U-order, Gray and double Gray orders are stable, while row-major, row-prime, Cantor-diagonal, spiral and Peano-Hilbert are not stable. See Figure 2.6 for example of space filling curves.

2.3 Spatial Indexing of Arrays

Array data is fundamentally different from standard spatial data. The data is sparse, but forms continuously dense regions with values that do not vary as much – they have high value locality.

Since our focus is on regularly gridded space, we'll assume the cells are of uniform size hyperrectangles, where each cell has a fixed set of attributes. See Section 2.1 for formal definition of the array model.

There are two general ways to model indexing.

The second general approach to model the index is to index individual cells. This approach never forms hyperrectangles (as the objects are treated as individual cells. Z-order curve is the most commonly used method for linearization. Basic B-trees based method built on top of the Z-order curve is for example N-tree [216], or on top of U-order curves [122].

Family of binning bitmap indexing algorithms on top of linearized arrays is another example of this model. See Section 2.4 for more details.

The first model treats sets of hyperrectangles (subarrays) of certain value ranges in the array as objects, thereby we may assume our objects are interior represented and can be decomposed into cells whose boundaries are parallel to the coordinate axes of our domain space (or can be transformed into such representation). It is then plausible to use standard spatial access methods for indexing hyperrectangles over integral domain (dimensions of this array). An static method was used in Searchlight [100] – a SciDB based system for range queries with aggregation constraints, using constraints programming on top of array synopsis.

There are several approximate indexing structures for regularly gridded space (arrays). A common technique is a P-tree [94], which is built on top of recursive space filling curves to describe the object of interest. Next, convex hull (both free and aligned with the coordinate axes) has been used in many solutions [181]. Many other combination of geometric object can be used, such as an intersection of minimum bounding boxes with minimum bounding spheres used in SR-trees [101].

2.3.1 Array Indexing of Arrays

Spatial indexing with arrays is a specific case of spatial access methods. It is possible to use arrays as an implicit representation of regularly gridded space. Retrieval of objects (connected components) can be done using depth first search. It is also possible to locate object by having and index – one representative location for each object.

Effective methods of indexing advanced spatial features, such as points, regions and objects are still an open problem. [1]

This approach has also been explored in a work by the author using array regridding. See Chapter 3 for details.

2.3.2 Tree Indexing of Arrays

Since the arrays may be sparse and objects contained in them can be larger than one cell (or alternatively, there may be large areas of the same values), it is not space efficient to use arrays directly for indexing.

We can use previously described spatial access methods from Sections 2.2.1 and 2.2.2. Both options are viable: using point access methods and spatial access methods on points, respective regions or more complex objects; or using linearization of the data and subsequently one-dimensional index structures on non-empty cells of the arrays. Some of the data structures, such as MX quadtrees allows searching for adjacent cells without separate tree queries [173].

This approach has been explored in a work by the author using hierarchical binning bitmap indexing. See Chapter 4 for details.

[1]This is a part of a research project called *Array-based Database Technology for Large-Scale Satellite Data and Their Analysis*. The author worked on this project at Nanyang Technologicla University in Singapore.

2.3.3 Tree Indexing of Array Blocks

We may relax the regularly gridded domain space and allow for disjoint rectangular unions of cells, called *blocks*, such that all the cells in a block either belong to the same object (or equivalently have the same value). The process of aggregation of cells into blocks itself is not trivial. Let's assume we have some fixed dimension ordering. We then determine the locations and sizes of the blocks based on predefined criteria and block dimension restrictions, construct a spatial access structure on top of these blocks and associate objects, resp. values with each of the block.

A simple method is a one-dimensional aggregation similar to run-length encoding. we create blocks over same-valued cells along one dimension. [2]

An efficient method to find a set of maximal blocks is *medial axis transformation* [107], also known as squarecode or rectangular coding, originally designed for encoding arbitrary planar shapes, but later adapted to regularly gridded space. Medial axis transformation is based on the distance of the cells from the boundary of the block. Subsequently, only the maximal squares (on the skeleton of the object) are stored.

Another option is to use irregularly gridded array. Where the grid boundaries are aligned with object boundaries. This is a form of compressed array representation. Additional d linear data structures called *linear scales* are needed to access the offsets of the grid boundaries. This approach is commonly used for example in SciDB.

Region quadtrees and region octrees are an adaptation of standard quadtrees and octrees for representing objects in arrays. However they lack in space efficiency due to the restriction of n-dimensional trees in terms of possible sizes n-dimensional trees can encode. Combination of quadtrees and medial axis transform, called *quadtree medial axis transform* [170], overcomes this problem by storing the maximal blocks and the quadtree itself represents skeletons of the objects.

Other variations include *Atrees* [28], which divide the domain space into a different amount of blocks along different dimensions; *bintrees* [177], which are an adaptation of PR k-d trees; *X-Y trees*, which generalize k-d trees into splitting the space into two or more parts along a dimension at the same time.

2.3.4 Indexing of Triangular Blocks on a Sphere

In many Earth science and astronomical domains, array (regularly gridded square cells) representation is not ideal. Instead, a spherical representation is more convenient.

Hierarchical Triangular Mesh (HTM) [200] is a tree data structure that indexes a sphere by recursively subdividing the surface of the sphere into finer triangles. It is based on sphere quadtrees [57] HTM allows to linearize a spherical model (two planar or three spatial dimensions) into single dimensional data. This single dimension represent a pre-order linearization of a tree. Models consisting of additional dimensions then form an array, where one dimensions is the linearized HTM. See Figure 2.7 for a visualization of HTM.

[2]This block aggregation method is implemented in current version mesh query module in Fastbit for aggregating bitmaps.

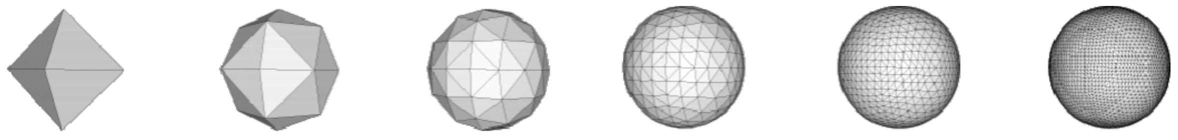


Figure 2.7: Hierarchical triangular mesh.

2.4 Bitmap Indexing

Bitmap indices, originally introduced in [37], were shown to be very effective for read-only or append-only data, and are used in many relational databases and for scientific data management [68, 189, 190, 42].

Bitmaps can either be created for a single attribute value, called *low-level bitmaps*, or for multiple values, called *high-level bitmaps*, where the bitmap is set to 1 for the cell of the arrays whose indexed value is in the value range of such bitmap.

The structure of high-level bitmaps is determined by a *binning* strategy. For high cardinality attributes, binning is the essential minimum to keep the size of the index reasonable [223, 222]. Binning effectively reduces the overall number of bitmaps required to index the data, but increases the number of cells that have to be later verified. This is called a *candidate check*. Two most common binning strategies are *equi-width* binning, which divides the attribute domain into equal intervals, and *equi-depth* binning, which divides the attribute domain into intervals covering equal (or near equal) number of cells. Equi-width binning is highly prone to excessive candidate checks, especially on skewed data.

d_1	d_2	a	EBM
0	0	3	0
0	1	2	0
0	2	~	1
0	3	~	1
1	0	4	0
1	1	2	0
1	2	1	0
1	3	5	0
2	0	4	0
2	1	7	0
2	2	3	0
2	3	2	0
3	0	~	1
3	1	5	0
3	2	4	0
3	3	1	0

$E_{[1]}$	$E_{[2]}$	$E_{[3]}$	$E_{[4]}$	$E_{[5]}$	$E_{[6]}$	$E_{[7]}$
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	1	0	0	0	0	0
1	0	0	0	0	0	0
0	0	0	0	1	0	0
0	0	0	1	0	0	0
0	0	0	0	0	0	1
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	1	0	0
0	0	0	1	0	0	0
1	0	0	0	0	0	0

$R_{[1,1]}$	$R_{[1,2]}$	$R_{[1,3]}$	$R_{[1,4]}$	$R_{[1,5]}$	$R_{[1,6]}$	$R_{[1,7]}$
0	0	1	1	1	1	1
0	1	1	1	1	1	1
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	1	1	1	1
0	1	1	1	1	1	1
1	1	1	1	1	1	1
0	0	0	0	1	1	1
0	0	0	1	1	1	1
0	0	0	0	0	0	1
0	0	1	1	1	1	1
0	1	1	1	1	1	1
0	0	0	0	0	0	0
0	0	0	0	1	1	1
0	0	0	1	1	1	1
1	1	1	1	1	1	1

$I_{[1,4]}$	$I_{[2,5]}$	$I_{[3,6]}$	$I_{[4,7]}$
1	1	1	0
1	1	0	0
0	0	0	0
0	0	0	0
1	1	1	1
1	1	0	0
1	0	0	0
0	1	1	1
1	1	1	1
0	0	0	1
1	1	1	0
1	1	0	0
0	0	0	0
0	1	1	1
1	1	1	1
1	0	0	0

Figure 2.8: Bitmap index for attribute `a` of the array `A` from Figure 2.1: empty bitmask `EBM`, equality encoded index `E`, range encoded index `R` and interval encoded index I.

Another crucial aspect of bitmap indexing is *encoding* [37]. which determines how a set of bins, B, of attribute domain is encoded in each bitmap and consecutively into a bitmap index. The simplest encoding, called *equality encoding*, encodes each bin with one bitmap for a total of $|B|$ bitmaps. Processing of equality queries reads a single bitmap, but processing of range queries has to read at most half of all the bitmaps. *Range encoding* uses $B-1$ bitmaps, each bitmap R_i encodes a range of bins $[B_1, B_i]$. The processing of range encoded bitmap index for range queries reads at most two bitmaps. *Interval encoding* [36] uses $\frac{|B|}{2}$ bitmaps, each bitmap I_i is based on range encoded bitmaps $R_i \oplus R_{i+\frac{|B|}{2}}$. Interval encoding uses at most two bitmaps to process range queries. Compared to range encoding, it uses only half the space. Figure 2.8 shows an example of equality, range and interval bitmaps for the array in Figure 2.1.

Bitmap indices, based on the number of bins, may take up to $|B| \cdot C$, where C is the cardinality of the indexed attribute, leading to a situation, where even a very small number of bins exceeds the size of the raw data. Binary run-length compression algorithms are usually applied on bitmap indices to reduce the overall size. However, another requirement is posed to these compression algorithms, such that it must be possible to run bit-wise operations effectively on the compressed bitmaps. There are two main representative compression algorithms, namely Byte-aligned Bitmap Code – BBC [7] and Word-Aligned Hybrid (WAH) compression [220], which is depicted in Figure 2.9. Many other algorithms that improve on BBC or WAH are described in [40]. Research in bitmap index compression is still ongoing, with the currently state-or-the-art algorithm called *Super Byte-aligned Hybrid* (SBH) [103]. All the current research on compressed bitmap indexing is focus on single-dimensional data.

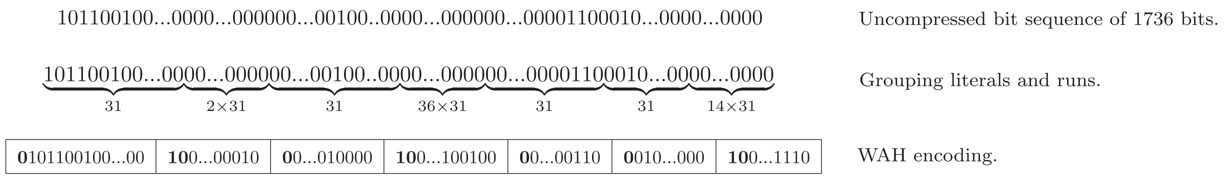


Figure 2.9: An example of World Aligned Hybrid (WAH) bitmap compression. WAH is a form of runtime length encoding compression, where only runs of zeros are run length encoded, while runs with ones are treated as literals. [103]

In order to facilitate effectively high cardinality attributes with space efficient indices and fast querying, two composite methods were introduced. The first method is *multi-component*, where the attribute value is decomposed into multiple components, which are then indexed independently. An example of multi-component index is a bit-sliced index [149], where each component corresponds to a bit of the value. Second composite method is called *multi-level* indexing [190], where the binning of the attribute becomes progressively more precise with increasing levels.

Thorough performance analysis of bitmap indexing, especially multi-level and multi-component both uncompressed and compressed is presented in [221]. An open-source bitmap indexing framework Fastbit [219] implements most of currently existing indexing schemes, mainly two-level indices.

2.5 Bucket Indexing Methods

Many spatial indexing data structures aggregate individual data points into buckets in order to save the overall size of the index and to reduce the number of potential disc access requests.

Simple extension of k-d trees called *k-d-B trees* [166] is a hybrid between k-d trees and B-trees. In k-d-B trees, it is possible for one internal node to consist of a set of adjacent regions, and for leaves to consist of multiple points. There are many improvements on k-d-B trees, including LST tree, hB-tree, k-d-b-trie, but their description is out of the scope of this work.

Grid indexing methods are much closer to array data model than trees. Gird methods split the domain space into non-regularly gridded space. *Grid file* [145] is the basic method of griding the domain space along all axes at the same time. Grid file requires the index to maintain another data structures called linear scales. *EXCELL* [201] is a modification of grid file that allows to split the domain space along a single dimension only and splits the domain space in predefined manner.

Many more bucket indexing methods are described in [174].

2.6 Similarity Hashing

Similarity hashing is a method for hashing similar items into the same buckets (as opposed to random hashing). This property is widely used in similarity searching systems. Its feasibility for multidimensional (as opposed to multi-attribute) indexing has not been deeply explored to date.

There are many ways to measure similarity between 2 data items. One of the major types of distances are *feature-based*,which rely most often on n-dimension vector or a set of values.

For such n-dimensional vectors, randomized fingerprinting schemes using features hashing were introduced. The most famous is the similarity hash (*simhash*) [38]. Min-wise independed hashing *minhash* first introduced in [29] deals with dimension reduction using multiple independent hashing functions. Minhash estimates the Jaccard similarity. A single hash function probes the Jaccard similarity on a single pair from the sets. Similar techniques are *Sdhash* [167] and *Sketching* [132, Chapter 19].

Locality Sensitive Hashing originally introduced in [91] uses these similarity hashing techniques to find the approximate nearest neighbors.

2.7 Spatiotemporal Indexing and Databases

Extending spatial data with a time dimension yield so called *spatiotemporal* data. The temporal dimension is usually treated in a special manner, especially due to the semantics. In many cases, however, the indexing can be generalized into n+1 dimensions. In arrays, the most common modeling scenario is a dedicated linear scale index for the temporal dimension.

Following is a list of common temporal and spatial predicates and operators

- *Temporal predicates:* overlap, precede, contain, equal, meet, intersect,...
- *Temporal operators:* intersect, following, preceding,...
- *Spatial predicates:* equal, disjoint, overlap, meet, contain, adjacent and common border,...
- *Spatial operators:* intersect, area, perimeter, distance,...

For a broader overview of spatiotemporal operators and indexing, see [204]. For overview of spatiotemporal idexing methods for both past, present and current events, see a thorough overview in [140].

Spatio-temporal databases, sometimes also known as moving objects databases or STDBMS, are popular database and computational platforms for spatiotemporal data of various kinds. The most known representative of spatiotemporal databases is Secondo [76]. Many spatial and spatiotemporal indexing algorithms have found their application in these databases.

2.8 Approximate Array Pattern Matching

Approximate pattern matching problems are a specific case of general searching problems with a user-defined template pattern (or a set of template patterns) and a pre-defined similarity measure and error bound.

The goal is to find a set of locations in the original array, where the distance to the pattern is less or equal to a given threshold.

Standard approximate pattern matching algorithms are not applicable in large multi-dimensional arrays due to the necessarily distributed nature of such large data and due to the high complexities of such algorithms. Although more advanced pattern searching algorithms use filtering to minimize the search space, it is still not applicable for large scale array data. Good overview on multidimensional approximate pattern searching algorithms is available in [15].

Many pattern matching solutions are limited to two dimensions, as such problems relate well to computer graphics and image processing. Some extensions allow for rotations as well [2].

Chapter 3

Inverted Regridding Indexing

In this section, we describe our work on inverted indices that allow effective execution of various spatiotemporal selection queries on satellite data. This work is more application oriented.

Given a trajectory from a user and satellite sensor data, we perform accurate data retrieval in the form of valid regions from the SciDB database containing complete ten-year QuikSCAT ocean surface wind fields satellite data. Subsequently, we regridded the regions and further processed the result, including visualization. This is a practical work aimed at large data processing and simple indexing.

This section is partially taken from [A.1]

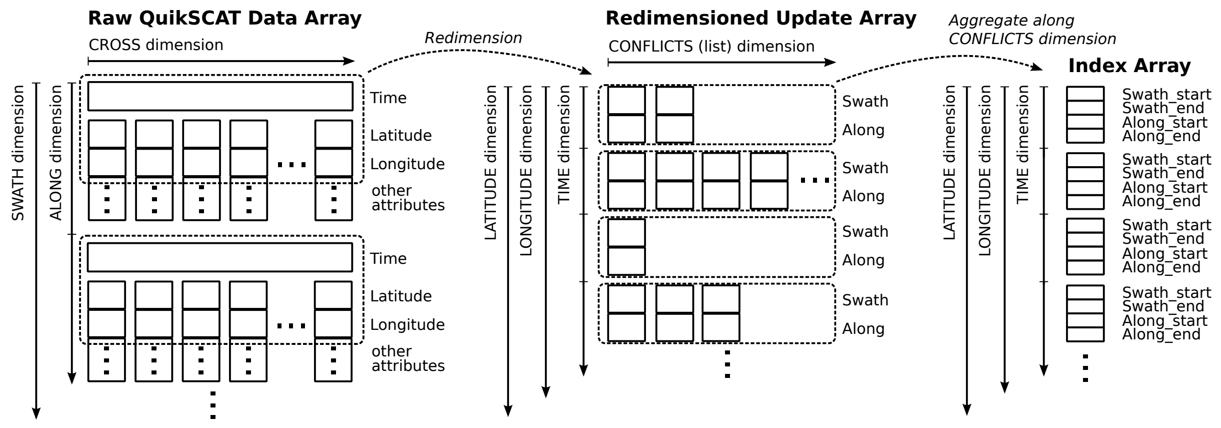


Figure 3.1: Array regridding scheme used during incremental generation of Latitude-Longitude-Time index.

3.1 Indexing Spatiotemporal Data

3.1.1 Regridding Index

Latitude-Longitude-Time index is a form of regridding index. The processing is based on a redimension operator of SciDB, where attributes of one array become dimensions of another (new) array. The new array's dimensions are usually rescaled compared to the original attributes' range. Conflicting cells (i.e. cells that are the target of multiple source array cells) can be treated by concatenating the values into a list followed by computing an aggregate function along the list of conflicting cells. Note the aggregation can be done without explicitely storing the list of conflicting cells.

The regridding process is depicted in Figure 3.1. The dimensions of the original raw data array *swath* and *along*, are turned into attributes, the *cross* dimension is discarded, and the attributes *latitude*, *longitude* and *time* are discretized and turned into dimensions by the redimension operation. Since there are many cells that ended up assigned the same *latitude*, *longitude* and *time*, most of them with different values of *swath* and *along*, we need to run an aggregate along the auxiliary dimension. This yields another array without conflicts, where the attributes *swath_start*, *swath_end*, *along_start* and *along_end* define a range on the *swath* and *along* dimensions in the original raw data array.

There is an array with dimensions representing latitude and longitude from *QuikSCAT_D3* array and time from *QuikSCAT_D2* (see array schemes for QuikSCAT data). The granularity of the target dimensions is determined by the use-cases or by individual levels in the hierarchical indices (see Section 3.1.3). The four attributes, `swath_start`, `swath_end`, `along_start`, `along_end`, of the index define a range in the data array, i.e. the values are pointers into the original data.

Every index cell covers a range in the data swath, possibly with overlaps. It is possible that multiple source cells end up indexed into a single target cell. As mentioned previously, an aggregate function is used. This aggregate function returns the range union of the swaths and along dimensions. For example, if there is an incoming data point from `swath=1863`, `along=341`, while currently the values for that index cell are `swath_start=1862`, `along_start=1860`, `swath_end=1862`, `along_end=2894`, then the index cell's pointers get updates to `swath_start=1862`, `along_start=1860`, `swath_end=1863`, `along_end=341`.

Index generation is fully incremental. However, to maximize parallelism, it is better to process swaths that are further apart. Therefore, the chunks accessed in the modified index arrays are more or less random, compared to consequent chunks, where there may be a lot of data-write dependencies due to a substantial amount of data being written to the same physical chunk.

The array schema of Latitude-Longitude-Time index for QuikSCAT data is as follows:

```
/* Index array with pointers to projected data */
LatLongTime_Index
<   swath_begin: uint32,
    swath_end:   uint32,
    row_begin:   uint16,
    row_end:     uint16
> [
    lat=0:720,    ?, 0,
    long=0:1440,  ?, 0,
    time=0:96432, ?, 0
];
```

3.1.2 Cartesian Index

An extension of Latitude-Longitude-Time index into Cartesian coordinates is *Cartesian Index*.

The main idea of this index is to allow for faster and more convenient spatial queries. Since SciDB uses Cartesian coordinates, it is more effective to keep index data in Cartesian coordinates as well. Distance based queries, predicates and operators can be effectively truncated based on the dimensions only. This effectively eliminates the need to read chunks that may be possibly rendered unneeded when further processing a query.

An example of Cartesian index with support for data projections is as follows:

```
/* Index array with pointers to projected data */
Cartesian_Index
<
    swath_start:       int32     not null,
    swath_end:         int32     not null,
    row_start:         int16     not null,
    row_end:           int16     not null,
    /* earliest and lates time points covered by this cell */
    time_start:        datetime  not null,
    time_end:          datetime  not null,
    /* direction of time flow within the cell (along swath path) */
    time_angle:        float     not null,
    /* 3d angle -- normal to the plane of projection */
    polar_angle:       float     not null,
    azimuthal_angle:   float     not null,
    /* projection pointer */
    projection_ptr:    int64     not null,
> [
    x    = 0:1024,  ?,  0,
    y    = 0:1024,  ?,  0,
    z    = 0:1024,  ?,  0,
    time = 0:87600, ?,  0
];
```

3.1.3 Hierarchical Structure of Indices

Our implementation of Latitude-Longitude-Time index consists of multiple levels of the indexing arrays. Each indexing array has different granularity, i.e., its cells cover different latitude, longitude and/or time. Due to uneven distribution of spatiotemporal data from satellites, some array locations may contain denser or sparser data. For example, the data density of scans around the equator is different from the density close to the poles. Note that for swath data the biased distribution of the data does not occur when using Cartesian indices.

With hierarchical indices, queries can be executed subsequently on more detailed (lower level) indices where necessary. When generating indices on multiple levels, we do not need to use online heuristics to determine where to use more levels. To do so, it would require a complicated re-indexing and possible re-reading of the raw data. Instead, since we are working with Earth observing satellite data, we can determine the boundaries for individual levels statically prior to indexing.

3.2 Use Case Scenario: Select QuikSCAT data given tropical cyclone trajectory

We demonstrate the spatiotemporal query capabilities on an example of data selection along a moving object:

Select QuikSCAT data along a given cyclone trajectory, with lat-lon radius of 1.0 degree from the cyclone eye and time span of [-3,+21] hours.

Cyclone trajectory is loaded into SciDB as a list of points in the following format: `<lat, lon, time >[i]`. This is done using CSV preprocessing and directly loaded from CSV into SciDB. Points of the trajectory are interpolated so that hyperrectangles on the interpolated points completely cover the neighborhood. Note that *all the processing is done within SciDB*, which also demonstrates the computational capabilities and the ease of integration of our solution.

3.3 Retrieving Data Regions – Pointers into the original data

Data is selected from a Latitude-Longitude-Time index array given the mask (as a list of hyperrectangles) using *cross_between* operator. Figure 3.2 shows an example of a trajectory mask, which is used as an argument for the index query.

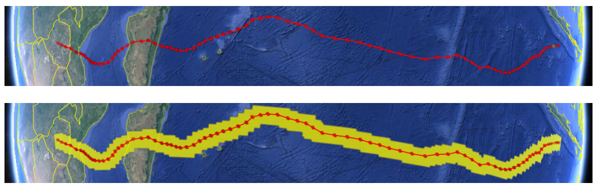


Figure 3.2: Trajectory and a corresponding mask – sequence of hyperrectangles on latitude, longitude and time dimensions.

Table 3.1: Timing results of a query - Select QuikSCAT Data Along Trajectory

Cyclone Data points	12	35	Isabel 98	Isabel 98	Isabel 98	Isabel 98
Radius [deg]	1.0	1.0	1.0	2.0	2.0	8.0
Span [hours]	24	24	24	24	96	168
Index query time [sec]	0.91	1.45	4.15	4.73	5.46	8.97

3.4 Time Complexity

Individual operations on the index arrays that do not require data transfer between nodes, i.e., lookup, cross between, filtering are all very fast.

Theoretical time complexities of such array operations used are at most linear (in size of the array) with expected linear parallel speedup. This assumes asymptotically lower initial overhead and coordination overhead, sufficient network speed for data retrieval (if needed) and fine enough chunk granularity (i.e. not all data in a single big chunk).

Note that assuming the chunking allocation scheme (i.e. to which node each chunk is placed) respects the neighborhood by allocating nearby chunks to different physical nodes, we estimate that the physical locations of chunks retrieved after the operations on the index arrays will be more or less random.

For a random allocation of the chunks to physical machine of both index and data arrays, the speedup is linear in expectancy – with high probability, the speedup $S \geq c \cdot n$ for some desired constant $0 < c < 1$.

Table 3.1 shows the timing results of the data retrieval query. Note that the time scales mainly with the number of data points of the trajectory. This is due to the fact that each data point is more likely to hit a brand new chunk in both the index arrays and the raw data arrays.

Increasing the radius (space) 2 times, i.e., the total area is increased 4 time, has minimal effect on the time. Based on the level of the index, the probability of hitting additional chunks of the index that need to be retrieved increases with lover level indices (denser mesh). Same goes for span (time). If we had a prior knowledge of our queries, we could adapt the chunking scheme of the index array to accommodate more spatial or more temporal data in a single chunk; thus, resulting in increasing performance for spatial, respective temporal intensive queries. However, in our data retrieval query example we kept the ratio balanced.

We run SciDB on a single physical server – Intel Xeon E5-2640 v3 2.6GHz, 20M Cache, 8.00GT/s QPI, 8 cores; 8x16GB RAM, 2133 MT/s, 4x1TB 7.2K RPM SATA 6Gbps. There are 4 virtual machines, each with 2 cores, 1 hard drive, 16GB RAM, running Ubuntu 14.04.1 server (3.16.0-30 kernel). Timing was measured as an average of 3 runs with cold start on the virtual machines. Note that the main server was kept running, which might have incurred some bias. However, the variation between individual measurements was negligible.

3.5 Conclusion

We described our work on the utilization of the distributed array-based SciDB database management system to support range query and data retrieval based on inverted indexing on arrays (regridding).

We demonstrated the regridding on Latitude-Longitude-Time index, which is the most straightforward and easy to understand and required the least preprocessing during data loading and index generating, assuming the original data contain latitude and longitude as attributes, which is the case of QuikSCAT data used during the demonstration.

Chapter 4

Hierarchical Bitmap Index for Range and Membership Queries on Multidimensional Arrays

In this chapter, we propose a new method for multidimensional array indexing that overcomes the dimensionality-induced inefficiencies. The hierarchical indexing method is based on n-dimensional sparse trees for dimension partitioning, with bound number of individual, adaptively binned indices for attribute partitioning. This indexing performs well on ranges involving both dimensions and attributes, as it prunes the search space early, avoids reading entire index data, and does at most a single index traversal. Moreover, the indexing is easily extensible to membership queries.

The indexing method was implemented on top of a state of the art bitmap indexing library Fastbit. We show that the hierarchical bitmap index outperforms conventional bitmap indexing built on auxiliary attribute for each dimension. Furthermore, the adaptive binning significantly reduces the amount of bins and therefore memory requirements.

This section is based on [A.4].

4.1 Related Work and Previous Results

Querying multidimensional array data requires effective index. Majority of the current systems rely are either built on top of relational databases or require a linearization of the array data, i.e., mapping the data into one dimension, enabling many one-dimensional access methods to be used. Others, such as array databases [196, 18], work natively with multidimensional arrays. Note that there are many schemes for linearization of array data, called space-filling curves (see Section 2.2.4, and schemes for linearization of spherical surface data [200], which have been used for partial linearization over spherical spatial dimensions only.

Traditional indexing methods like B-trees and hashing are not effectively applicable to indexing multidimensional arrays, and are being replaced by multidimensional indexing methods, such as R-trees [77], R*-trees [20], KD-trees, n-dimensional trees (quadtrees, octrees, etc.) [171, 172]. These methods are not very effective for high dimension arrays with attribute constraints and are relatively space demanding. A good overview of spatial indexing algorithms is in [174], though majority of the focus is on traditional spatial data instead of multidimensional arrays.

Approximate pattern matching problems are a specific case of general searching problems with a user-defined template pattern (or a set of template patterns) and a pre-defined similarity measure and error bound. Standard approximate pattern matching algorithms are not applicable in large multidimensional arrays due to the necessarily distributed nature of such large data and due to the high complexities of such algorithms. Although more advanced pattern searching algorithms use filtering to minimize the search space, it is still not applicable for large scale array data. Good overview on multidimensional approximate pattern searching algorithms is available in [15].

Many pattern matching solutions are limited to two dimensions, as such problems relate well to computer graphics and image processing. Some extensions allow for rotations as well [2].

A popular and very effective method of indexing arbitrary data is bitmap indexing, which is an index consisting of a set of bitmaps (bitvectors) with associated metadata. Bitmap indices leverage hardware support for fast bit-wise operations (AND, OR, NOT, XOR), and are very space-efficient, especially for low-cardinality attributes [219], although this was partially overcome by sophisticated multi-level and multi-component indices. Bitmap indices are used in majority of commercial relational databases [68, 189, 190, 42].

Compressed, hierarchical, multi-component bitmap indexing has also been used for relational data [221] or in a simple form for sparse array data [143].

Other works utilize bitmap indexing for spatial applications, but do not model the data as multidimensional arrays [124, 191, 195].

Many generalized searching problems on arrays have occurred recently as a result of a boom in scientific computing based on data representation as multidimensional arrays. The most prominent problem is constraint programming with aggregate-based constraints called Searchlight [100], using simple synopsis indexes. Indexing for aggregate-based problems has been investigated in a context of approximate aggregation problem in [215] and subgroup discovery [214].

More general problems closely related to array searching problems involved data uncertainty (both positional and values) [156], similarity search [207], similarity join [231] iterative processing [193], data exploration [100] and other.

The multidimensional distributed array database SciDB has been used as a framework for multiple applications on top of spatiotemporal data. From previously mentioned solutions, Searchlight [100], AscotDB [208] iterative processing [193] is built on top of SciDB as well.

Recently, a selective filtering (attribute-based search) improvement was introduced for SciDB [104].

4.2 Problem Statement

In our work, we focus on selection, searching, and exploratory tasks, which make up a foundation for majority of other data processing tasks and scientific applications. Our objective is to design a family of indexing schemes for multidimensional array data that allows for fast selection queries and general computational operations.

The selection query parameters are a set of dimension values or ranges; dimension result range constraints; attribute values or ranges; attribute template patterns, including sparse and don't care cells, with distance measure definition and corresponding value; and aggregate conditions.

Computational operation run on top of selection results include result aggregations, exploratory tasks and general purpose array operations.

An example of an array query is:
`From SATELLITEARRAY, find all hyperrectangular regions` R`, such that` $R[lat] \in [-50, 80]$ `and` $R[long] \in [20, 150]$ `and sizeof(R[lat])` ≥ 5 `and sizeof(`$R[long]$`)` ≥ 5 `and` $1420070400 \leq R[time] \leq 1451606400$`, where avg RAINFALL in` $R \geq 100$ `and WIND_DIRECTION in` R `matches TORNADOPATTERN with HAMMINGDISTANCE` ≤ 0.5`, output` 100 `region` $\in R$ `with max avg WINDSPEED;`

Since the query execution leads to a multidimensional array optimization task, which based on the region constraints may lead to optimization problem over integral domain – equivalent to NP-Hard Integer Programming, we intend to employ randomization and approximation techniques in the query execution.

We allow probability relaxation on the correctness of each result region, both in terms of false positives and false negatives. Furthermore, we introduce accuracy relaxation in terms of constraints satisfaction of selected regions.

4.3 Description of Hierarchical Bitmap Array Index

We now briefly discuss a common way of indexing multidimensional arrays using additional bitmap indexes for each dimension. Then we describe the structure of our hierarchical bitmap array index.

Arrays $A\langle a_1, \ldots, a_m\rangle[d_1, \ldots, d_n]$ are usually stored in a linearized representation, most commonly C-style row-major array representation. Creating one index $I_{d_i=k}(d_1, \ldots, d_n)$ for each dimension d, which is set to 1 for cells of array A where d is equal to a value k. This allows filtering out results based on dimensions using binary AND.

Note that the dimensions index $I_{d_i=k}(d_1, \ldots, d_n)$ does not necessarily have to use equality encoding, but based on the expected queries, we may choose a better combination of binning, encoding and compression. This approach is used in [215, 235] with equi-depth binning or in [214] with v-optimized binning based on v-optimal histograms [95] and C-style row-major linearization in [198].

Unfortunately, dimension bitmap index is not effectively compressible. Consider an example of row-major ordering on 5x5 array. Then the row dimension index for $column = 1$

is `01000 01000 01000 01000 01000`, which cannot be effectively compressed using either BCC or WAH, since the compression context of both is a single bit. This can be partially mitigated by stretching dimensions to multiples of bytes or words, and extending the run-length compression to use byte or word in its compression context, instead of single bits. Another option is to use either Z-order or Hilbert space filling curves to further increase locality of the dimensions. Neither, however, solves the problem entirely.

4.3.1 Partitioning of Arrays

Non-partitioned data require much finer binning and the domain of the dimension is higher than its partitioned counterpart, thus higher amount of bins is required. By partitioning the array $A\langle a_1, \ldots, a_m\rangle[d_1, \ldots, d_n]$ into a set of regularly gridded chunks C in the *Multidimensional Array Clustering* fashion described in Section 2.1, such that:

$$C_i[o_1, o_2, \ldots o_n, e_1, e_2, \ldots, e_n] = \\ A\langle a_1, \ldots, a_m\rangle[o_1 \leq d_1 < e_1, \ldots, o_n \leq d_n < e_n]$$

All chunks in our data model are of the same shape, i.e., for all chunks C_i, C_j of array A, it holds that

$$C_i[e_k] - C_i[o_k] = C_j[e_k] - C_j[o_k]$$

for all dimensions k, and chunks are not overlapping and completely cover the whole array A. In the chunk notation, o_k stands for offset and e_k stands for end of the chunk along that dimension (exclusive boundary).

By chunking the array, we limit the domain of both attributes and dimensions in a given partition. In our adaptive binning indices, we use the fact that the domain of the attribute varies based on the location.

The first problem arising from the equal size chunking model is that within a single chunk, we are still required to use either indexing or at least aggregate information on the attributes, such as *min* and *max* for precise queries or *histograms* for probabilistic queries, or data exploration. We choose to use bitmap indexing on both attributes and dimensions within the chunk. Note that the dimension indices are the same for all chunks in the array, since for each chunk, we can simply subtract its offset from the dimensions query constraints.

The second problem lies in the overall structure of the chunks. There is no direct, high level index of the attributes for the chunks. It is necessary to scan through the synopsis of all the individual chunks, or generate a hierarchical synopsis. The latter has been utilized in [100] in a form of a graph generated over merging sub-arrays.

We propose a unified solution that solves both the problem with dimension attributes and with synopsis of array chunks. Our solution is in a form of hierarchical bitmap index on top of a n-dimensional tree (such as octree for 3 dimensions) with variable binning for each node in the tree.

4.3.2 Structure of the Array Chunk Index

The index is done separately for each attribute of the array A. Let's fix an attribute α. All the following functions refer to this attribute.

Each chunk $C(o_1, o_2, \ldots, o_n)$ of array $A\langle a_1, \ldots, a_m\rangle[d_1, \ldots, d_n]$ is associated with exactly one leaf $N_\ell(o_1, o_2, \ldots, o_n)$. Independently, each leaf uses an equi-depth binning index with a total of at most `BINS` bins, where bin boundaries $bins(N_\ell$ of the index are based on an exact chunk values histogram. Note that this assumes uniform distribution of queries. If we had any prior knowledge of the queries based on the attribute, we would instead opt for weighted histogram to construct the binning. The leaf's dimension boundaries correspond to its associated chunk's boundaries, clipped by the global shape of the array A.

Accounting for empty values (missing cells in A) is done using a special bitmask, known as *empty bitmask*, for a total of $\texttt{BINS} + 1$ indices. Only leaves with at least $E \cdot \texttt{BINS}$ non-empty cells are indexed, where the constant E is dependent on the data structure used for the leaf representation, i.e., do not use bitmap indexing if listing the values is more space efficient.

Encoding of the leaf indices is left as a parameter to the user, as the bitmap indexing performance heavily depends on the cardinality of the array attribute, desired number of bins, and query types. For generality, we assume high cardinality attributes, such as integers and doubles and small number of bins such as $\texttt{BINS} \leq 16$.

Except for very narrow dimension range queries, a dimension query will either cover the whole span of a leaf node, or result in a one-sided dimension range query once the query processing reaches a single chunk. Thus, the ideal encodings for chunks are *range* and *interval* encodings [36]. Our default encoding is interval encoding since it uses half the memory range encoding does. Encoding of inner nodes is more complicated and we describe it in Section 4.3.5.

4.3.3 Structure and Construction of the Hierarchical Bitmap Array Index

To deal with the higher level index, we create a special composite index on tree similar to n-dimensional tree. Each internal node of the index has at most F children, where F is called a *fanout*. Note that, unlike in quadtrees, octrees or n-dimensional trees, F is not necessarily 2^n, where n is the number of dimensions. Our bitmap indices are based on the fanout and we want to utilize binary operations as much as possible. For this reason, the fanout F should be a multiple of the processor word size W, or as close to it as possible.

The overall internal node fanout F can be expressed in terms of a fanout F_{d_k} for a single dimension k as

$$F = \prod_{k=1}^{n} F_{d_k} \leq \left(\max_{1 \leq k \leq n} F_{d_k}\right)^n$$

Assuming that the dimension fanout F_{d_k} is the same for all dimensions, we can get

$$F_{d_k} = \left\lfloor F^{\frac{1}{n}} \right\rfloor$$

As we will see in Section 4.4.2, in order to facilitate efficient dimension range queries, the size of F cannot be too large, since the size of precomputed dimension clipping bitmaps depends on F.

The index tree construction works in a bottom-up fashion, where the leaf nodes are indexed at first. This allows both data appending and modification (see Section 4.3.7). Each internal node is constructed from at most F direct children and with at most `BINS` attribute bins, with one additional index for empty bitmask. Each child node N_i of internal node N provides its attribute's $min(N_i)$ and $max(N_i)$ values. These values are used for the construction of the bitmap index of N.

Let $B = (min(N_1), max(N_1)), \ldots, (min(N_F), max(N_F))$ be the set of all intervals ranging from the minimum to the maximum value of the indexed attribute α among all the child nodes N_i. The set B is the set of bins – the individual interval boundaries are delimiters, where the attribute's α value a is in the attribute domain of different child nodes. Formally, let $nodesin(a) \subset N_i$ be a function of a value $a \in \alpha$ of attribute α, which returns a subset of child nodes.

$$N_i \in nodesin(a) \iff min(N_i) \le a \le max(N_i)$$

The set $nodesin(a)$ is used to construct the binning for index of this internal node. We describe the encoding of this bitmap index in Section 4.3.5.

The index bins are aligned with the bins from B. This guarantees that no two indices for different bins will be identical, i.e., represent the same set of children. It also directly implies that adding more boundaries to B would be pointless.

4.3.4 Bin Boundaries Merging in Parent Nodes

The number of bins from all F child nodes is higher than `BINS` for majority of the internal nodes N, therefore it is necessary to reduce the size of the set of bins, B. There are several strategies to choose $B \subset D$ such that $|B| = \texttt{BINS}$. An example of such binning reduction is in Figure 4.1.

The first strategy is to use an equi-width distribution of the bins. This is the ideal choice assuming the attribute part of the query is uniformly distributed or when there is no prior knowledge about the attribute query and assuming the data distribution is not skewed.

The second strategy is to use equi-depth binning. This is ideal if the attribute distribution of the child nodes is skewed. It is possible to maintain the weights of the bins for leaf nodes, since those have direct access to the data. However, internal nodes can only make estimates about the weight of merged bins. In each internal node and leaf, we store the weight estimate $w(b)$, where $b \in B$. The weighted square error of a bin b is

$$wse(b) = \left| w(b) - \frac{w(D)}{\texttt{BINS}} \right|^2$$

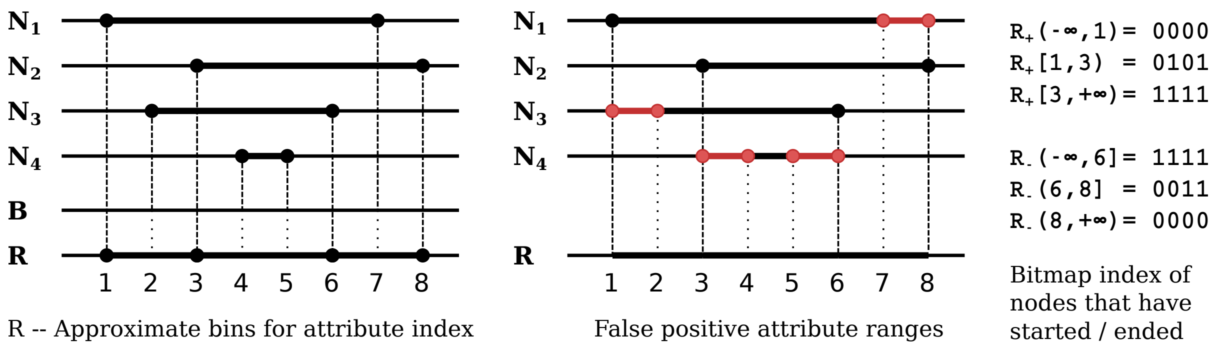


Figure 4.1: Example of merging $|B| = 8$ bin boundaries to $|R| = 4$ bin boundaries for 4 child nodes. False positive ranges are marked in red. Two sided range encoded bitmaps are generated for R.

and the weighted sum square error is

$$wsse(B) = \sum_{b \in B} wse(b)$$

.

To estimate the weight of merged bin $r \in R \subset B$, we assume uniform distribution of values over the intervals of bins $b \in B$. Then the estimated weight of r is

$$w(r) = \sum_{b \in B} w(b) \cdot \texttt{sizeof}(b \cap r)$$

where $\texttt{sizeof}(b \cap r)$ is the size of the intersection of r and b.

We cannot use the trivial algorithm for equi-depth binning, because we can only iterate by bins of variable weight, instead of iterating by single data points. This is why we need to approximate the equi-depth using a simple iterative algorithm. Details on selecting $R \subset B$ approximately equi-depth bins are shown in Algorithm 4.1. We first start with equi-width binning (line 1). Then, we generate sets of all possible bin splits and merges (lines 2-3), setup two priority queues and evaluate all possible splits and merges in terms of weighted sum square error (lines 4-11). After that, we perform one valid split and one merge on the binning as long as this leads to an improvement of the overall binning (lines 14-18). This preserves the total number of bins.

In case a node has either a low cardinality attribute throughout all the child nodes, we create bins mapped to single values of the attribute and their corresponding bitmaps.

Note that v-optimal binning does not work in our case, since we don't have the individual data values available during construction of the internal nodes, although we could approximate this using uniformly or normally distributed estimates within the bins of child nodes, or by propagating at least basic data synopsis.

```
Input: set of bins $B$, set of weights $w(b)$, $b \in B$, number of output bins BINS
Result: approx equi-depth bins $R \subset B$, $|R| = BINS$
$R \leftarrow$ eq-width bins from $B$, $|B| = BINS$;
$B_S \leftarrow$ all possible split bins of $R$;
$B_M \leftarrow$ all possible merged bins of $R$;
$Q_{SPLIT} \leftarrow$ priority_queue();
$Q_{MERGE} \leftarrow$ priority_queue();
for $s \in B_S$ do // bins to split
    add $(s, \Delta wse(s))$ to $Q_{SPLIT}$;
end
for $(m, m') \in BM$ do // bins to merge
    add $((m, m'), \Delta wse((m, m'))$ to $Q_{MERGE}$;
end
// split that decreases wsse the most
$(s, \Delta wse(s) \leftarrow min(Q_{SPLIT})$;
// merge that increases wsse the least
$((m, m'), \Delta wse((m, m'))) \leftarrow min(Q_{MERGE})$;
while $\Delta wse((m, m')) > \Delta wse(b)$ do
    split $b$;
    merge $(b, b')$;
    update $R, B_S, B_M, Q_{MERGE}, Q_{SPLIT}$;
end
```

Algorithm 4.1: Iterative equi-depth binning approximation

4.3.5 Double Range Encoding of Bitmap Indices in Internal Nodes

Unlike in bitmap indexing in leaves where one encodes positions of individual values, we encode sets of child nodes $nodesin(a)$ for attribute values a in the internal nodes. Our binning B has the property that for all attribute values $a_b, a''_b \in b \in B$ it holds that $nodesin(a_b) = nodesin(a'_b)$. Note that this does not hold for intervals $r \in R$ (See Figure 4.1 for an example).

We will now describe an effective bitmap encoding of $nodesin(a)$, $a \in r \in R$. Let's have two adjacent intervals $r \in R$ and $r' \in R$, such that $r_h = r'_\ell$ Note that since $R \subset B$, we have $nodesin(r) \neq nodesin(r')$. If $nodesin(r') \supset nodesin(r)$, then r' corresponds to a bin, where nodes are added, and we add r' to a set R_+. Else, if $nodesin(r') \subset nodesin(r)$, then nodes are removed in set $nodesin(r')$, and we add r' to set R_-. Otherwise, some nodes are added and some are removed and we add r' to both R_+ and R_-. In our example in Figure 4.1, $R_+ = \{[1, 3), [3, 6)\}$ and $R_- = \{(3, 6], (6, 8]\}$.

There is no guarantee that $|R_+| = |R_-|$. If we wanted, we could run Algorithm 4.1 separately on boundaries B_+ and B_- (likewise defined) and with $\frac{\texttt{BINS}}{2}$ bins, but then we'd lose the equi-width approximation.

Now, we encode $|R_+| + 1$ bitmaps using range encoding, so that the index for bin $r_+ \in R_+$ corresponds to children, whose attribute range minimum $min(N_i)$ is $\leq$ to the upper boundary of interval r_+. In our example, bitmap corresponding to $r = [1, 3) \in R_+$ is `0101`, indicating that N_1 and N_3 have started in or before this interval. Similarly, we encode $|R_-| + 1$ bitmaps for values r_- using inverse range encoding, i.e., children, whose attribute range maximum $max(N_i)$ is $>$ to r_- are encoded by 0 in the bitmap, representing children that have already ended before or in the interval r_-.

These two bitmaps easily allow evaluation of partial and complete matches (see Section 4.4.1) using only two bitmap reads and one logical operation for both partial and complete query.

4.3.6 Locality of the Hierarchical Index

In order to preserve locality of the data during queries, we store the whole index in a locality preserving linearization of an n-dimensional tree. For each query, blocks of the index are loaded sequentially and sparsely, based on the parameters in the query. Thus, only one traversal, possibly incomplete, of the index data is needed. The index data consist of bin boundaries, weight estimates and bitmap indices.

We use space filling curves, namely the Z-order curve to linearize the multidimensional array index. We choose not to use recursive multi-level Z-order curves, as this would force the query processing to be based on pre-order traversal of the index tree. We also choose not to use row major ordering, since it has poor locality and it would slow down retrieving locations child nodes and partitions. Hilbert curve has perfect locality, but it does not preserve dimensions ordering. This means we would need to precompute bitmaps for dimension constraints for each block of Hilbert curve separately. Z-order curve allows for fast child and parent node index computations, preserves dimensionality between different level and has a good locality.

The order $\mathcal{Z}_\ell$ of the Z-order curve of level ℓ is determined by the maximal fanout $F_{max} = \max_{1 \leq k \leq n} F_{d_k}$, where F_{d_k} is a fanout of dimension k.

$$\mathcal{Z}_\ell = \ell \cdot \lceil \log_2 F_{max} \rceil$$

Assuming F_{d_k} is the same for all dimensions, the order of Z-order curve is then

$$\mathcal{Z}_\ell = \ell \cdot \left\lceil \log_2 \left\lfloor F^{\frac{1}{n}} \right\rfloor \right\rceil$$

and such a Z-order curve has length of $(\mathcal{Z}_\ell)^n$.

Several of the higher levels are stored in a dense vector, as specified by a user parameter. These vectors are expected to be densely filled. The remaining levels are stored as non-overlapping intervals on a Z-order dimension (1D) in continuous blocks, indexed by a binary search tree. This is a compromise between sparse single node map and full vector used for higher levels. Note that the blocks may not be sequential in memory, but at most a single transition is guaranteed, i.e., no blocks are read twice during the processing of a single query.

4.3.7 Appending and Modifying Data

Scientific data is often considered either fixed or append only, our indexing approach allows for both appending and data modification, although the latter is not convenient.

To append data along any dimension, we apply the same bottom-up procedure to update the index. It is necessary to update the dimension bounds of internal nodes (that were possibly previously clipped by the global shape of the array) and bitmap indices (to include the new child nodes). Note that we do not have to update the weight estimates and bin boundaries (except min and max) in order to assure index correctness. However, in order to assure the equi-depth optimal binning, we need to run the bin merge algorithm again on affected nodes.

4.4 Querying Dimensions And Attributes

In this work, we focus on selection queries over dimensions and attributes of an array. Such query consists of a set of dimension constraints and attribute constraints. Let's specify a query q over an array $\mathcal{A}\langle a_1, \ldots, a_m\rangle[d_1, \ldots, d_n]$ as a set of ranges over dimensions q_D and attributes q_A.

$$q = q_A \cup q_D = \{(a, a_\ell, a_h)\} \cup \{(d_j, j_\ell, j_h), \ldots\}$$

where (a, a_ℓ, a_h) is a triple specifying attribute constraint: attribute, its lower bound and its (exclusive) upper bound; same goes for dimensions. In this work, we focus on a single attribute query. Therefore, we simplify q_A to (a_ℓ, a_h). It is possible for a query to not specify constraints for some dimensions, in which case we fill all q with remaining dimensions, to a complete query. Dimensions, that were not specified, are filled with $(d_j, min(d_j), max(d_j))$ triples. One-sided range constraints are also extended in similar manner.

The core of the query algorithm is a breadth-first descent through the index tree. At each level, the search space is pruned according to both dimension and attribute values.

Let N be the currently searched node, N_i be its child nodes, where $0 \leq i < F$; multi-dimensional range D_N be the set of dimension boundaries in the format $[D_N[d]_\ell, D_N[d]_h]$, where d is dimension, ℓ designates lower bound, h upper bound, associated with node N.

Throughout the query processing, we maintain a queue of partially matched nodes P and a set of completely matched nodes C. We start at a root node N_r, setting $P = \{N_r\}$, assuming that both: node N's boundaries and query dimensions are not disjoint: $D_N \cap Q_D \neq \emptyset$ and $(min(N), max(N)) \cap Q_A \neq \emptyset$, otherwise node $N \notin P$ and $N \notin C$.

Let p, p', $p*$ and c, c', $c*$ be zero bitmaps of size F; the bitmaps p indicates partial attribute matches among the children of node N, p' indicated partial dimensions matches, $p*$ indicates partial matches, similarly the vectors c, c', $c*$ indicate complete matches. We will now set these vectors according to the query Q for the first node in queue P. The partial and complete matches bitmap computation is also described in Algorithm 4.2 and in Figure 4.2.

```
Input: query $q = \{(a_\ell, a_h), (d_1, d_\ell, d_h), \ldots\}$ with DIMS
dimension constraints; node $N$; node children $N_1, \ldots, N_F$; boundaries
$[D_N[d]_\ell, D_N[n]_h]$ for $N$ and all $N_i$ and dimensions $d$;
Result: partial matches $p*$; complete matches $c*$;
1  $\mathcal{P}_{N,\mathcal{S}}$, $\mathcal{C}_{N,\mathcal{S}} \leftarrow$ load index for node $N$;
2  $\mathcal{P}'_{\mathcal{S},d}$, $\mathcal{C}'_{\mathcal{S},d}$; // precomputed;
3  $p \leftarrow \{0\}^F, p' \leftarrow \{0\}^F, p*$;
4  $c \leftarrow \{1\}^F, c' \leftarrow \{1\}^F, c*$;
5  if $a_h < min(N)$ or $a_\ell > max(N)$ then
6      return $p* \leftarrow \{0\}^F$, $c* \leftarrow \{0\}^F$
7  $c = c \,\&\, \mathcal{C}_{N,\mathcal{S}}(a_\ell, a_h)$;
8  $p = p \mid \mathcal{P}_{N,\mathcal{S}}(a_\ell, a_h) \,\&\, {\sim}c$;
9  for dimensions $d$, $1 \le d \le$ DIMS do
10     if $d_h < D_{N_i}[d]_\ell$ or $a_\ell > D_{N_i}[d]_h$ then
11         return $p* \leftarrow \{0\}^F$, $c* \leftarrow \{0\}^F$
12     if $d_\ell > D_N[d]_\ell$ then
13         $p' = p' \mid \mathcal{P}'_{\mathcal{S},d}(d_\ell)$;
14     if $d_h < D_N[d]_h$ then
15         $p' = p' \mid \mathcal{P}'_{\mathcal{S},d}(d_h)$;
16     $c' = c' \,\&\, \mathcal{C}'_{\mathcal{S},d}(d_\ell, d_h)$;
17 end
18 $p' \leftarrow p' \,\&\, c'$;
19 $c' \leftarrow c' \,\&\, {\sim}p'$;
20 $c* \leftarrow c \,\&\, c'$;
21 $p* \leftarrow (p \mid c) \,\&\, (p' \mid c') \,\&\, {\sim}c*$;
22 return $p*$, $c*$
```

Algorithm 4.2: Evaluation of partial and complete match bitmaps for a single node.

4.4.1 Attribute based Matches

In this subsection, we explain how attribute bitmask is set. This subsection further describes lines 5–8 in Algorithm 4.2.

If $a_h < min(N)$, or $a_\ell > max(N)$, there are neither partial nor complete attribute matches and we terminate processing the current node.

Let $\mathcal{P}_{N,\mathcal{S}}(a_\ell, a_h)$ be a *partial attribute match* bitmasks specific to node N of for an array of shape $\mathcal{S}$, with bits set to one corresponding to children N_i so that the intersection $[a_\ell, a_h] \cap [min(N_i), max(N_i)] \neq \emptyset$.

$$
\begin{aligned}
\mathcal{P}_{N,\mathcal{S}}(a_\ell, a_h)[i] = 1 &\iff \mathcal{P}_{B|N,\mathcal{S}}(a_h)[i] \wedge \neg\mathcal{P}_{E|N,\mathcal{S}}(a_\ell)[i] \\
\mathcal{P}_{B|N,\mathcal{S}}(a)[i] = 1 &\iff min(N_i) \le a \\
\mathcal{P}_{E|N,\mathcal{S}}(a)[i] = 1 &\iff max(N_i) \ge a
\end{aligned}
$$

The second expression describes bitmap set to 1 for children that have started before or at value a, the third one describes children that have ended at or after a. The first expression then combines both.

To evaluate $\mathcal{P}_{N,\mathcal{S}}(a_\ell, a_h)$, we first use binary search on R_+ and R_- to find two bins $L \in R_+$ and $H \in R_-$ such that $a_\ell \in L$ and $a_h \in H$. These bins L and H mark the attribute boundary bins. Then, $\mathcal{P}_{B|N,\mathcal{S}}(a_h)$ is identical to $R_+[H]$ and $\neg\mathcal{P}_{E|N,\mathcal{S}}(a)$ is identical to $R_-[L]$, where R_+ and R_- are the bitmap indices described in Section 4.3.3, each queried for a single bin. Then we add $\mathcal{P}_{N,\mathcal{S}}(a_\ell, a_h)$ to p using bitwise OR.

Now, we process complete candidates in a similar fashion. Let $\mathcal{C}_{N,\mathcal{S}}(a_\ell, a_h)$ be a *complete attribute match* bitmask specific to node N for array of shape $\mathcal{S}$, so that the intersection $[a_\ell, a_h] \cap [min(N_i), max(N_i)] = [a_\ell, a_h]$.

$$\mathcal{C}_{N,\mathcal{S}}(a_\ell, a_h)[i] = 1 \iff \mathcal{P}_{B|N,\mathcal{S}}(a_\ell)[i] \land \neg\mathcal{P}_{E|N,\mathcal{S}}(a_h)[i]$$

This expression is very similar to $\mathcal{P}_{N,\mathcal{S}}(a_\ell, a_h)$, describing children that have started at or before a_ℓ and have not ended at or before a_h. To evaluate $\mathcal{C}_{N,\mathcal{S}}(a_\ell, a_h)$, we query $R_+[L]$ and $R_-[H]$. Then, we add the result to c using bitwise OR and remove those from p, i.e., $p = p \land \neg c$.

Note that both partial and complete attribute candidates use a total of 4 index queries. An example of attribute query is displayed in the bottom row in Figure 4.2.

4.4.2 Dimension based Matches

Next, we explain how the dimension masks are set. This subsection further describes lines 9–17 in Algorithm 4.2.

If for any dimension d it holds that $d_h < D_{N_i}[d]_\ell$ or $a_\ell > D_{N_i}[d]_h$, there are neither partial nor complete dimension matches and we terminate processing the current node.

Unlike attribute query, the evaluation of dimension query is the same for all nodes N, so all the bitmaps for processing dimensions queries are *precomputed*.

Let $\mathcal{P}'_{\mathcal{S},d}(d_\ell, d_h)$ be a *partial dimension match*, where d is a dimension in the query constraint (d, d_ℓ, d_h), for an array of shape S, indicating child nodes N_i such that the intersection $[D_{N_i}[d]_\ell, D_{N_i}[d]_h,] \cap [d_\ell, d_h] \neq \emptyset$.

Let's fix a dimension d for which we evaluate partial matc0hes $\mathcal{P}'_{\mathcal{S},d}(d_\ell, d_h)$:

$$\begin{aligned}
\mathcal{P}'_{\mathcal{S},d}(d_\ell)[i] = 1 &\iff d_\ell \in D_{N_i}[d] \land d_\ell \neq D_{N_i}[d]_\ell \\
\mathcal{P}'_{\mathcal{S},d}(d_h)[i] = 1 &\iff d_h \in D_{N_i}[d] \land d_h \neq D_{N_i}[d]_h \\
\mathcal{P}'_{\mathcal{S},d}(d_\ell, d_h)[i] = 1 &\iff \mathcal{P}'_{\mathcal{S},d}(d_\ell)[i] \lor \mathcal{P}'_{\mathcal{S},d}(d_h)[i] \\
\mathcal{P}'_{\mathcal{S}}(d_\ell, d_h)[i] = &\bigcup_{1 \leq d \leq \texttt{DIMS}} \mathcal{P}'_{\mathcal{S},d}[i]
\end{aligned}$$

The first expression describes which children N_i have dimension d range such that the query limit d_ℓ falls inside the range, but it is not equal to the lower limit of that range. The second expression is similar, but for d_h. Third and forth expression combine the partial matches over both query limits and all dimensions. Note that this results in excessive partial candidates since all child nodes that intersect the query constraints along at least one dimension qualify as partial candidates.

Partial dimension matches are evaluated using one precomputed bitmap index corresponding to

$$\mathcal{P}'_{\mathcal{S},d}(b)[i] = 1 \iff b = D_{N_i}[d]$$

where b is a bucket corresponding to the chunking of the array $\mathcal{A}$. There are a total of F_d such buckets along dimension d, resulting in a total of $F_d \cdot d$ bitmaps of size F. We query these bitmaps for all dimensions and combine them using OR into p'

There is a special case of false negative dimension result. If d_ℓ or d_h is equal to the d'th dimension range border of a child node N_i, and at the same time the other end of d_ℓ or d_h causes the dimension to be fully covered in N_i, i.e. $d_\ell = D_{N_i}[d]_\ell$ and $d_h \geq D_{N_i}[d]_h$ or $d_h = D_{N_i}[d]_h$ and $d_\ell \leq D_{N_i}[d]_\ell$, the query is evaluated as partial match for N_i and dimension d, while in fact dimension d contributes to complete matches. A check for this scenario requires comparing the dimension ranges of child nodes to the query range, and was ignored on purpose, as it complicates and slows down the query process.

For complete candidates, we will slightly modify the definition of $\mathcal{C}$ used for attributes. Let $\mathcal{C}'_{\mathcal{S},d}(d_\ell, d_h)$ be a *complete dimension match* for array of shape S, indicating which child nodes N_i are *partially or fully* covered by interval $[d_\ell, d_h]$. Despite the semantics indicating partially matches should not be included, we later trim the complete dimension match bitmap accordingly.

$$\mathcal{C}'_{\mathcal{S},d}(d_\ell, d_h)[i] = 1 \iff [d_\ell, d_h] \cap D_{N_i}[n] \neq \emptyset$$
$$\mathcal{C}'_{\mathcal{S}}(d_\ell, d_h)[i] = \bigcap_{1 \leq n \leq \texttt{DIMS}} \mathcal{C}'_{\mathcal{S},d}[i]$$

Complete dimension matches are evaluated using two precomputed bitmap indices corresponding to

$$\mathcal{C}'_{B|\mathcal{S},d}(b)[i] = 1 \iff b \leq D_{N_i}[d]$$
$$\mathcal{C}'_{E|\mathcal{S},d}(b)[i] = 1 \iff b \geq D_{N_i}[d]$$

similarly to bitmaps used for partial matches. There is a total of $2 \cdot F_d \cdot d$ bitmaps of size F for complete matches. We query these bitmaps for all dimensions and combine them using AND into c'.

We now combine the partial dimension matches bitmap c' with p', such that $p' = p' \wedge c'$. Then, we clip the complete dimension bitmap by the partial bitmap as $c' = c' \wedge \neg p'$. During the evaluation of dimension matches, we used a total of $3 \cdot d$ index queries. An example of dimension query is displayed in the top row in Figure 4.2.

4.4.3 Partial and Complete Matches

Now that we have both attribute and dimension, and both partial and complete candidates, we may proceed to merging the candidates and generating a bitmap representing the set of result node children $\mathcal{C}^*_{N,\mathcal{S}}$ and a bitmap representing the set of potential node children $\mathcal{P}^*_{N,\mathcal{S}}$ that will be recursively explored. This subsection further describes lines 18–22 in Algorithm 4.2.

The $\mathcal{C}^*_{N,\mathcal{S}}$ bitmap is easier to obtain, as it is the intersection of both complete bitmaps without partial candidates bitmaps.

$$\mathcal{C}^*_{N,\mathcal{S}} = \mathcal{C}_{N,\mathcal{S}} \wedge \mathcal{C}'_{\mathcal{S}}$$

We obtain the set of partial candidates $\mathcal{P}^*_{N,\mathcal{S}}$ by joining the dimension-based partial candidates with the attribute-based candidates and clipping both by complete candidates

$$\mathcal{P}^*_{N,\mathcal{S}} = (\mathcal{P}_{N,\mathcal{S}} \vee \mathcal{C}_{N,\mathcal{S}}) \wedge (\mathcal{P}'_{\mathcal{S}} \vee \mathcal{C}'_{\mathcal{S}}) \wedge \neg\mathcal{C}^*_{N,\mathcal{S}}$$

We then iterate through the results, adding child nodes from $\mathcal{C}^*_{N,\mathcal{S}}$ to the result set C and the partial candidates $\mathcal{P}^*_{N,\mathcal{S}}$ into the queue P to be processed subsequently. This process is done on top of Z-order indices, as it is trivial to generate Z-order indices corresponding to nodes in the lower levels. The Z-order ordering of the inner nodes and breadth-first traversal also ensures single traversal through the index.

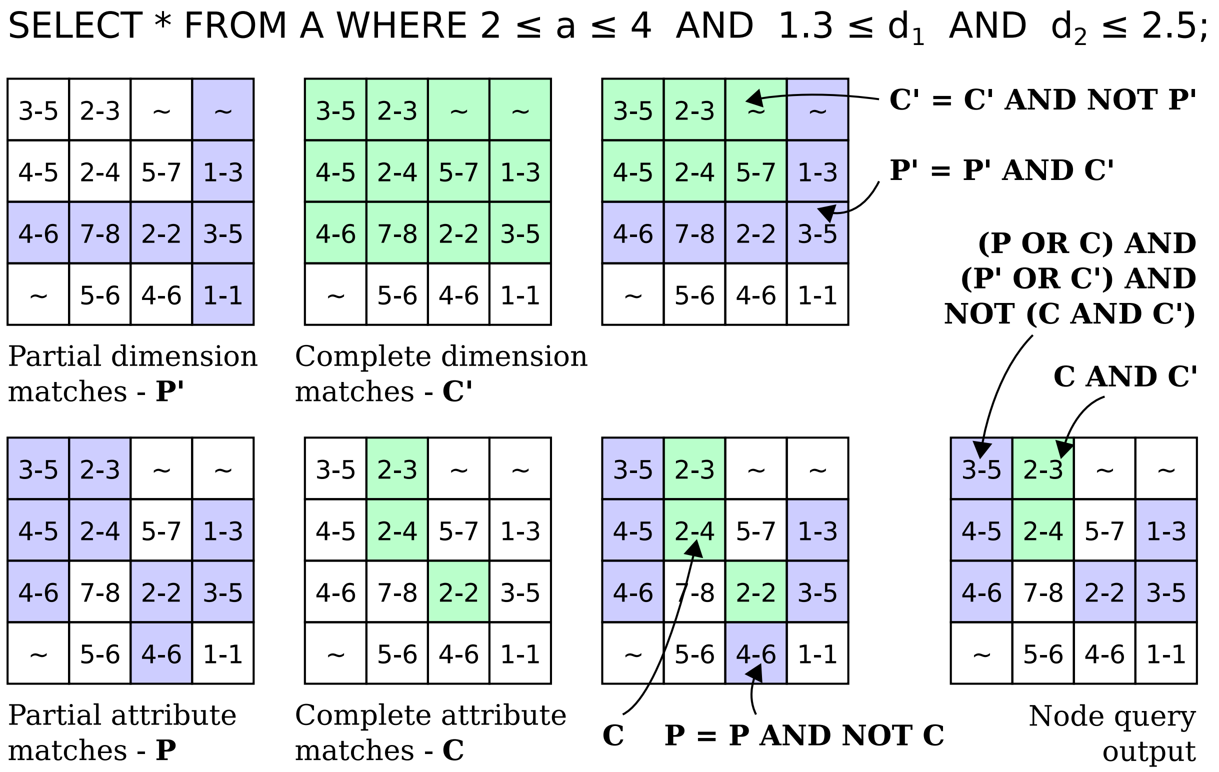

SELECT * FROM A WHERE 2 ≤ a ≤ 4 AND 1.3 ≤ d1 AND d2 ≤ 2.5;
3-5 2-3 ~ ~
4-5 2-4 5-7 1-3
4-6 7-8 2-2 3-5
~ 5-6 4-6 1-1
Partial dimension matches - P'
Complete dimension matches - C'
C' = C' AND NOT P'
P' = P' AND C'
(P OR C) AND (P' OR C') AND NOT (C AND C')
C AND C'
Partial attribute matches - P
Complete attribute matches - C
C
P = P AND NOT C
Node query output

Figure 4.2: Processing of a query in a single node of the hierarchical index. Top row represents dimension constraints, bottom row represents attribute constraints. Bottom right is the final product. Blue nodes represent partial matches and green node represent complete matches.

Running the algorithm for multiple queries or multiple attribute constraints in a single query can be implemented using iteration through the constraints in the worst case.

4.4.4 Estimating Cardinality of Results; Membership Queries

It is fairly straightforward to output estimates on minimal and maximal number of matching cells by iterating some bounded number of levels of the index. The minimal number outputs the size of nodes in C, while the maximum outputs the size of nodes in $C \cup P$. Using the $w(b)$ estimate, we may also provide estimates on aggregates over the attribute, based on bin-wise linear approximation.

There is a simple modification of the algorithm for membership queries. (See Section 2.1 for details about membership queries). On top of two sided range indices $\mathcal{P}_{N,\mathcal{S}}$ and $\mathcal{C}_{N,\mathcal{S}}$ for attribute queries, we keep equality indices and iterate through the attribute constraint. For dimension membership queries, we precompute an index for all dimension values (within a single chunk), as opposed to buckets corresponding to child nodes, that are used in $\mathcal{P}'_{\mathcal{S},d}$ and $\mathcal{C}'_{\mathcal{S},d}$.

4.5 Experimental Evaluation

We have tested our implementation against several other solutions, of which none is specifically tailored to mixed attribute and dimensions range queries, but those are the only readily available solutions involving bitmap indices and being capable of executing range queries.

We measured the time and space efficiencies for each individual query, i.e. total query execution time, and space requirements for the index. Timing was measured as an average of 3 runs with data preloaded into memory. For Fastbit queries, we use their internal wall time measuring systems, meaning certain pre and post processing steps are not included in the time measurements, such as query string parsing. Space requirements were measured based on the disk space required to store the bitmap index together with all relevant metadata.

The experiments were run on a single physical machine – Intel(R) Xeon(R) CPU E5-1650 v2 @ 3.50GHz, 16 GB RAM, 1TB 7.2K RPM SATA 6Gbps; running Ubuntu 14.04.1 (3.19.0-32 kernel).

We use a synthetic dataset to test our queries on – *randomly generated multidimensional sum gaussian distribution* SumGauss. Its only attribute a_G is a sum of G randomly initialized Gaussian distribution in D dimensions:

$$a_G(\vec{d}) = \sum_{i=1}^{G} \left(\frac{1}{\sqrt{(2\pi)^D |\Sigma_i|}} \exp\left(-\frac{(d-\mu_i)^T \Sigma_i^{-1} (d-\mu_i)}{2} \right) \right)$$

where μ_i and Σ_i are randomly generated distribution mean vector and a bounded symmetric positive definite covariance matrix for dimension i. For sparse arrays, a threshold for the Gaussian functions is used. Attribute is treated as empty if the value is below this threshold. Only partitions with at least one non empty value are generated.

4.5.1 Fastbit Integration

Fastbit [219] is an open source library that implements bitmap indexing. It's not a complete database management system, rather a data processing tool, as its main purpose is to facilitate selection queries and estimates. Fastbit's key technological features are WAH bitmap compression multi-component and multi-level indices with many different combinations of encoding and binning schemes.

We use Fastbit's partitions to setup the lowest level of our indices (leaves), and base our binning indices on Fastbit's single-level binning index. This approach requires preprocessing of the data into evenly shaped partitions, generating empty bitmasks and shape metadata. Once a table is preprocessed into even partitions, it is indexed as described in Section 4.3. The index generation processes one partition at a time, and once processed, the partition is never accessed again during the index generation.

4.5.2 Bitmap Indexing Methods

BoxClip represents a naive algorithm using 32 equi-depth binned indices, interval encoding and WAH compression. The result bitmask from the attribute query is transformed to a set of "line" hyperrectangles (size of the hyperrectangle in all but one dimensions is 1), which are filtered from the dimension query, then merged into a set of result hyperrectangles. All the steps except filtering are built on top for Fastbit's mesh query. The filtering is implemented using recursive sweeping line algorithm.

DimsAtts uses indexed `uint` auxiliary attributes made from dimensions (see Section 4.3). The dimension query is preprocessed into attributes, then run as a multi constraint query in Fastbit. The configuration is the same as in BoxClip, using 32 binned indices, range encoding and WAH compression on all attributes.

ArrayBit represents our hierarchical multidimensional index. We use 16 equi-depth binned indices, range encoding and WAH compression to index the partitions, and 16 approximately equi-depth binned indices (described in Section 4.3.4) with two sided range encoding and no compression for the hierarchical index. Note that compared to BoxClip and DimsAtts, we only use half of the bins in the partition index. It is sufficient in our algorithm, because the bin boundaries are adapted to the actual data in each partition, and because we need to store the bin boundaries within the partitions.

4.5.3 Range Queries

In our work, we focus on mixed attribute and dimension queries. Regardless of the dataset, we categorize the queries based on the overall ratio of the size of the query result to the size of the total array size.

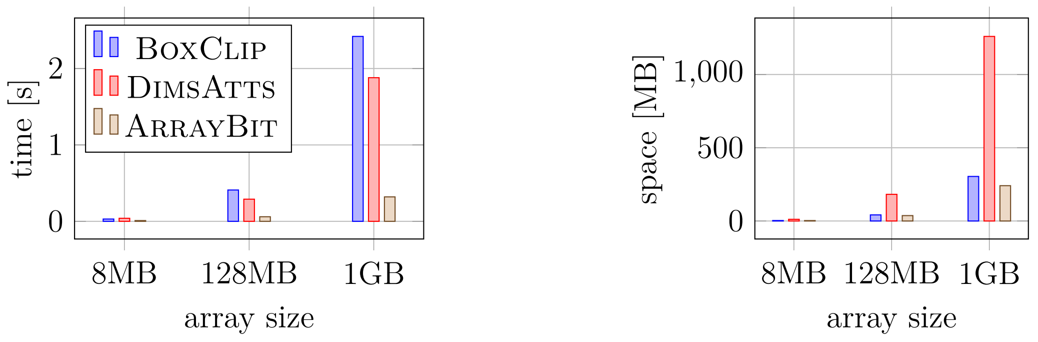


Figure 4.3: Query execution time and disk space required to store the indices for different array sizes.

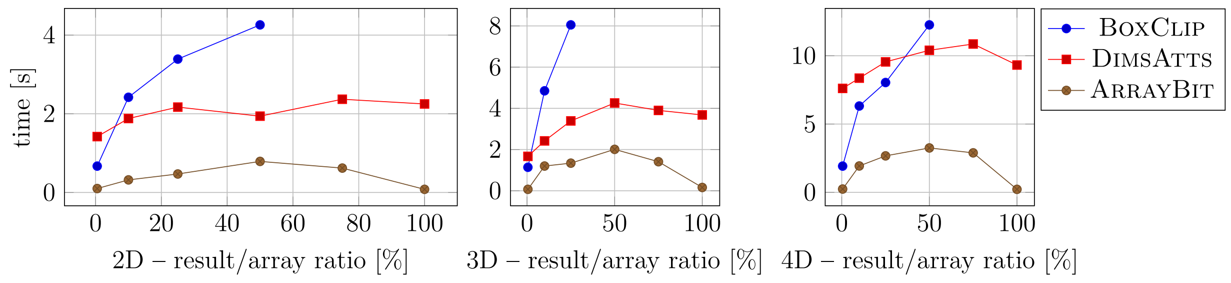


Figure 4.4: Query execution time for 2D, 3D and 4D queries of various hit ratios. Queries contained an attribute constraint and all dimension constraints, each constraint with approximately the same domain reduction.

Figure 4.3 shows the time required to return all results. The index file is preloaded into memory prior to the test for all the systems used. We used 2D array for this experiment. and a query with ≈ 10% hit ratio. Both BoxClip and DimsAtts run slower than ArrayBit. In case of BoxClip, the reason is that all the attribute query results had to be processed, while for DimsAtts the reason is that the attribute made from second dimension didn't effectively compress. In terms of space requirements, all of the algorithms save attribute index. ArrayBit uses less bins in the leaves, but stores bin boundaries for all leaves and internal nodes, plus bitmaps for internal nodes, effectively taking up the same space as BoxClip. On the other hand, DimsAtts stores indices for all dimension attributes. Row major ordering is used in this measurement.

Figure 4.4 demonstrates the dependency of the query processing time on a hit ratio of the query, i.e., the ratio of selected cells vs total cells in the array. BoxClip algorithm does not prune the search space based on the dimensions, resulting in number of hits dependent on the attribute only. Filtering these is is time intensive. DimsAtts depends linearly on the total number of dimensions. This is because there is an additional attribute for each dimension. There is also a small dependency on the hit ratio, where the increase is due to the results retrieval. ArrayBit achieves very good results for low or high hit rate queries. This is due to a large number of complete matches, and due to fast pruning of search space. For medium hit rate queries, the algorithm has relatively high number of candidate nodes to explore, but still manages to prune the search space faster.

4.5.4 Parameterization

We also experimented with different setups of our hierarchical index. The major objectives remain the same: query execution time and space requirements of the index.

First, the *partition size* determines the ratio of partition index vs hierarchical index. We set this in equilibrium with *number of index bins*, which increases the precision of the binning and results in higher probability of pruning the search space earlier.

Another important parameter is a *fanout* of nodes. If we use a smaller fanout (the smallest possible is 2^D), we may not fill a single memory word with the index, significantly impair bit parallelism, furthermore the index size will be larger due to much deeper indexing tree. If the fanout is too high, we will not prune infeasible candidates fast enough. We got optimal results with a fanout close to a multiple of the word size, such as $8^2 = 64$ for 2D arrays, $4^3 = 64$ for 3D, $4^4 = 256$ for 4D, $3^5 = 243$ for 5D, etc.

4.6 Conclusion

Most of the work on bitmap indexing to date focus on improving the space efficiency and speed, while a few applied the bitmap indices to multidimensional data. However, the linear form of bitmap indices was never adapted to support multidimensional array data.

We have proposed a bitmap indexing method that is designed for multidimensional arrays and focuses on overcoming the dimensionality issue. The hierarchical nature of the proposed method allows for continuous results and estimates to be output as intermediate results. Our approach effectively prunes the search space, uses data adaptive, approximate equi-depth binning. Furthermore, the index supports partitioned array data and allows distributed storage.

Our experimental results show that the proposed bitmap indexing method outperforms standard linearized approaches for mixed attribute and dimension range query processing.

There is a possible caveat that more complex multi-level and multi-component indices exist. None of these indices overcome the problem of dimensionality, rather due to their effectiveness delay the threshold where the drawbacks became noticeable (in terms of number of dimensions and size of the array).

Future work includes adapting the tree structure based on dimensions, such as adaptive mesh refinement widely used in physical simulations [24]. Another interesting possibility is multi-attribute index in a single hierarchical structure. Last, we want to use better approximation algorithms to determine feasible regions from finer attribute bins.

Chapter 5

Similarity Search Using Compressed Inverted Lists on Graphic Processing Units

Similarity search using GPU-accelerated inverted index relies on the availability of large amount of data in the GPU memory. Query execution requires frequent transfer of large index data from CPU to GPU in order to resolve many top-k similarity queries in parallel. This transfer often creates a bottleneck for such query execution.

In this chapter, we have designed, implemented, and evaluated multiple decoding schemes for fully data parallel decoding and query evaluation, and a compressed representation of inverted index for similarity search on the GPU. This improves the overall query evaluation time by reducing the cost of data transfer. It comes with at a small cost to computational time. The reduced transfer time often out-weights the minor computational increase needed to decode the inverted index, effectively reducing the overall query execution time. In practical scenarios, the decoding cost is completely overshadowed by data transfer in this data-bound problem. The data parallel implementations have sped up query time of the system 3-4 times on most real world datasets. Compression of inverted index also increases the overall amount of data to be simultaneously loaded for database applications in GPU memory.

All the components were integrated into the publicly available generic similarity search framework *GENIE* (Generic inverted index on GPU) in a robust and modular architecture, with configurable query compiler and index management components. We modify the standard inverted index models abstracting query compilation and index structures. These extensions of GENIE were designed for multi-GPU multi-node distributed deployment with initial implementations of the distributed functionality publicly available.

Furthermore, we use heuristics for encoding algorithm selection based on the properties of the datasets and corresponding inverted list. We demonstrate this efficiency of our system and compression on several real life datasets.

The original work on GENIE has been published in [A.2] (with additional technical report [A.5]), which includes data preprocessing using locality sensitive hashing or shotgun & assembly, the match counting algorithm for τ-*ANN* search, original match counting GPU implementation, and comparison to other locality sensitive hashing schemes on GPU and to other *ANN* search implementation on the GPU. This chapter mainly focuses on contributions beyond the scope of the original paper.

This research work, including GENIE, had been partially carried out at the SeSaMe Centre, under Interactive Digital Media Institute (IDMI), National University of Singapore (NUS), Singapore, from October 2016 to August 2017. Afterwards, the work has been carried out with main affiliation at the Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague.

The main contributions of the chapter are:

- We have improved query performance of generic high dimensional data similarity search by implementing a compressed inverted index, query compiler, incorporating data parallel decoding of the inverted index on GPU.
- Multiple decoding schemes were designed, implemented, and evaluated for fully data parallel decoding and query evaluation. We use heuristics for encoding selection based on the properties of the dataset being indexed, and properties of the inverted lists.
- The data parallel implementations have sped up query execution time 3-4 times on most real world datasets.
- All the components were integrated into the publicly available framework *GENIE* (Generic inverted index on GPU)[1] in a robust and modular architecture, with configurable query compiler and inverted index management components.
- The extensions of GENIE were designed for multi-GPU multi-node distributed deployment with initial implementations of the distributed functionality publicly available[2].

5.1 Introduction

Inverted indexing is a commonly used technique in a variety of indexing applications. Examples of large data intensive applications come from astronomy and astrophysics [87], finance [43] neuroscience, engineering, multimedia and others [236].

In a domain, where each document consists of a list of terms, inverted index maps the individual terms to documents using lists of document identifiers – commonly known as *docId*. Similarly, a column index of a relational database may map individual column values to a list of corresponding rows. Alternatively, a multidimensional data points can

[1]The source code of released versions of *GENIE* (Generic inverted index on GPU) is available at `https://github.com/SeSaMe-NUS/genie`, with optional compilation of the inverted index compression functionality available with the CMake option `GENIE_COMPR:BOOL`.

[2]The optional compilation of the distributed capability of GENIE is available with the CMake option `GENIE_DISTRIBUTED:BOOL`.

be also seen as column values (where each column corresponds to a single dimension), and thereby indexed using inverted indexes. To evaluate a query, inverted lists are usually intersected to obtain a list of docIds satisfying all the constraints, i.e., matching exactly the query terms in all columns. Our interest develops from the latter case by extending the queries to *top-k* similarity search under a specific similarity model called *match count* [A.2], which evaluates the similarity of searched objects by number of matching dimensions. This prevents the use of common methods of list intersection, since such approach relies on a full match in all the dimensions, In our approximate nearest neighbor problem, we are most often accepting partial matches.

The problem of match counting itself calls for processing whole inverted lists, as opposed to a list intersection, where we are able to discard invalid results relatively early. Cases where we can discard a potential result in match counting are rare (few last processed lists can all contain the same docsIds, effectively bumping such docID into top-k results).

Evaluating multiple queries, where each query consists of inverted lists counting is a highly parallel operation. Graphical Processing Unit (GPU) is a popular parallel architecture, previously chosen for both inverted indexing [153, 49] and compression of integer lists [56]. One of the most commonly available general consumer (about USD 700) graphic card to date is GeForce GTX 1080 Ti[3], based on Pascal GPU architecture, with 11 GB GDDRX5 RAM having effective bandwidth of 484.4 GB/s, with 3584 CUDA cores. One of newest data center class GPUs is the A100 80GB SXM[4], based on Ampere architecture, with 6912 CUDA cores, 80 GB HBM2e memory having effective bandwidth of 2TB/s, and a PCIe Gen4 (64GB/s) and NVLink (600GB/s) interconnect.

Eight of such A100 GPUs interconnected with NVSwitch are packaged for example in the new NVidia DGX A100[5] system. On top of the GPUs such system consists of two 64-core AMD EPYC 7742 CPUs, 1TB DDR4 system memory, 15TB high-speed NVMe SSD, and eight Mellanox 200Gbps HDR adapters.

Even though the parallel processing power of these GPUs is unmatched by any CPU to date, the memory size is a limiting factor in terms of usefulness for any in-memory index. For large databases, we must rely on either partially loading the data into GPUs, where the GPU memory transfer rate is a bottleneck, or on prediction of dataset partitions containing the correct top-k results, which in turn necessarily introduces a potential error. Therefore by efficiently compressing our dataset, we can maximize the utilization of the limited GPU memory.

We build our system as an extension of GENIE (Generic Inverted Index on the GPU) [A.2], which supports match count similarity model, and a simple query execution on single GPU. To increase effective size of in-GPU-memory inverted index, we use compression of the inverted index. Our goal is to maximize the amount of index that fits onto GPUs at a single time and to minimize the index transfer between CPU and GPU (for indexes

[3]`https://www.techpowerup.com/gpu-specs/geforce-gtx-1080-ti.c2877`

[4]`https://www.techpowerup.com/gpu-specs/a100-sxm4-80-gb.c3746`

[5]`https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-dgx-a100-datasheet.pdf`

exceeding the portion of GPU memory allocated for index data). The system is built on top of CUDA framework.

Compression for inverted indexing for similarity search on GPU is different from conventional inverted lists compression mainly in terms of the objectives. In our work, we focus on achieving high compression ratio of the index and efficient data parallel decompression in order to keep the decompression time during query execution as low as possible. The added decompression time on GPU during query execution is compensated for with the decrease in index transfer time from main memory to GPU memory. Furthermore, we deal with short and long inverted lists by balancing the sizes of inverted lists to maximize parallelism of the GPU. Short lists often do not achieve good compression if compressed standalone, while long lists add a layer of complexity in the CUDA framework, therefore decreasing performance. For short inverted lists, we use different encoding to prevent such inefficiencies. Additionally, the CPU side query compiler adds a constant size information to compiled queries for GPU. The information is then used to initialize and aid the decompression process.

In Section 5.2, we go through related work on topics of integer lists compression (on CPU, using SIMD, and on GPU), acceleration of database operations using GPUs, similarity search, approximate similarity search, similarity search engines on heterogeneous architectures, and locality sensitive hashing. Most of the topics are elaborated from the point of view of different architectures, such as CPU, SIMD instruction sets, GPUs, even some on FPGAs and ASICs. The section further introduces basic concepts of GP-GPU and CUDA. In Section 5.3, we introduce GENIE (GENeric Inverted indEx), a GPU framework for approximate similarity search that is used as a foundation for this work. In Section 5.4, we describe our extension and adaptation of GENIE's query processor and data preprocessing for the purposed of compressed match counting. In Section 5.5 we evaluate the work with an analysis of individual compression schemes, data transfer time to decompression time trade-offs, inverted lists balancing, and inverted lists partitioning. We conclude in Section 5.6 with a summary of results and elaborating on future work directions.

5.2 Related Work

This section provides a broad overview of work and research on several topic related to approximate similarity search on GPU using compressed inverted index. An overview of integer lists compression is in Section 5.2.1 and GPU-accelerated integer list compression in Section 5.2.1.1. Since GENIE is seen as GPU-accelerated similarity search database, some closely related database operations are reviewed in Section 5.2.2. Next a GPU-accelerated exact similarity search is summarized in Section 5.2.3 and approximate similarity search in Section 5.2.4. The most common technique used for approximate similarity search is locality sensitive hashing, which is further summarized in Sections 5.2.4.2 and 5.2.4.3. Finally, a brief overview of GPGPU and CUDA is available in Section 5.2.5.

The closest related work to GENIE is *GPU-LSH* [153] (for similarity hashing) and GPU k-selection [1] (for top-k results selection). Other GPU accelerated approximate exact nearest neighbor search systems exist, but mostly relying on brute force similarity evaluation. See Section 5.2.4.1 for more details. However none of the other tools compare to GENIE in terms of a complete *k-ANN* functionality. The closest related work to data parallel decoding is *ParaPFor* [8], *Para-EF* from *Griffin-GPU* [123] and two decompression schemes *GPU-BP* and *GPU-VByte* introduced in [131].

5.2.1 Integer Lists Compressions

Integer lists compression is very often used for compression of inverted lists (posting lists) in search engines. The purpose of the compression is to minimize the storage overhead of the cold-stored inverted lists.

Most inverted lists compression schemes also store additional inverted index metadata, such as frequencies, positional information, document length, or compression metadata, such as value ranges. Note that docIds are often sorted and therefore majority of the compression schemes benefit from some type of delta encoding. Most metadata are not sorted and often are saved in raw form.

The initial effort on integer lists compression has focused on efficient CPU only algorithms. *VByte* [44] is a byte-aligned compression with variable representation size. This means it first tries to store an integer value using one byte. If that fails, the algorithms tries to use two bytes, and so on. The most significant bit serves as a control bit. This compression has been extremely popular in industry, where it was used by Google (using a 4 integer packed version called *Varint-GB*) [46] or Amazon (using a vectorized, fixed size 8-byte packing called *Varint-G8IU*). This concept has later been subsequently altered to separate control bits and the data into two different arrays [113].

Another class of compression algorithms relies on a *patched frame of reference* (*PFor*) concept to encode blocks of integers, where integers deviating form the frame are encoded separately as exceptions. The most common examples are *PForDelta* [237], *NewPFD* [224] which stores the exception offsets and overflows of the the exception values in two separate arrays, and *OptPFD* [224], which finds the exceptions threshold as an optimization function of the estimated decompression speed. The frame of reference has been vectorized in *SIMD-FastPFOR* [110].

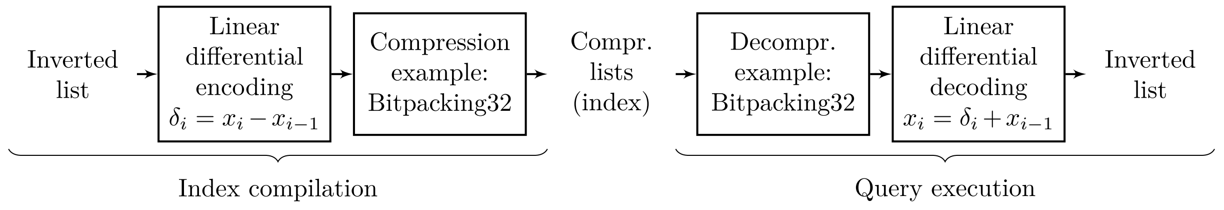

Inverted list
Linear differential encoding $\delta_i = x_i - x_{i-1}$
Compression example: Bitpacking32
Compr. lists (index)
Decompr. example: Bitpacking32
Linear differential decoding $x_i = \delta_i + x_{i-1}$
Inverted list
Index compilation
Query execution

Figure 5.1: Integer lists encoding and decoding pipeline (with linear delta encoding and Bitpacking32) in inverted index setting.

Quasi-succinct indexes [210] represent another approach for encoding inverted indices. This approach does not rely on delta-encoded block-based indexes, but is based on Elias-Fano codes and bitpacking for dense sections on the lists.

Similar family of integer compression is called *bit-packing* (or binary packing). Bit-packing often utilizes hardware vectorization due to the regular structure of the codewords. The initial 32-bit codes were presented in [4] and consisted of *Simple-9*, *Relative-10*, and *Carryover-12*. Each code represents a selection of sub-codes to pack a variable amount of integers in 9, 10, resp 12 different ways. Additional coding families were subsequently introduced, namely *Slide* [5], and *interpolative codes* [138, 41]. Majority of previous 32-bit codes were subsequently extended to 64-bit versions [6]. Note that all these codes predate common adoption of SIMD architectures. Significant amount of work has also been focused on index list caching [229]

Later, majority of the codes were extended from 64-bit to 128-bit SIMD implementations. One of the first SIMD implementations was Elias gamma code called *k-gamma* [182]. Two schemes were introduced using SSE2, namely bit-packing *SIMD-BP128* and frame of reference *SIMD-FastPFOR* [110]. Variable byte schemes were vectorized as well, including a successor of VByte called*varint-G8IU* [194] using SSE3 (uses one control byte for 8 bytes of data). *QMX* [205] combines bitpacking with RLE and also separates control and data streams. Masked-VByte [162] used the original VByte format (i.e. no grouping across multiple codewords), and Stream-VByte [113] also separates the control bits and data into the streams. *Roaring bitmaps* [35, 112] use either RLE or bit-packing for dense sequences, and array container for sparse sequences. Roaring bitmaps have been used for example in Elasticsearch.

Reordering of docIDs Due to inverted index compression relying heavily on the average size of deltas, several approaches were developed to reorder (remap) docIDs of the dataset in order to minimize the delta sizes. Document reordering has been processed using recursive similarity graph decomposition [27], by solving Traveling Salesman Problem on similarity graph [185], or by reducing the dimensionality of the dataset with Singular Value Decomposition [26]. In a newer work, a set of heuristics is used to maximize the runs of deltas of 1s using intersection-based docID assignment(IBDA) followed by run-length encoding [11]. Inverted index compression techniques for optimally ordered documents are further elaborated in [224]. All of these reordering approaches are based on computing the similarity between documents, and do no utilize other techniques such as similarity hashing.

Partitioning Methods Improvements to the integer lists codes often incorporate some form of partitioning of each inverted list into smaller sub-lists or blocks. *VSEncoding* [187] using dynamic programming to find an optimal block length, given the integer logarithm of every integer in the array, minimizing the total size of the index. *Partitioned Elias-Fano method* (PEF) [151] uses dynamic programming for selecting chunk endpoint to minimize the overall index size, creating a two level index (to store the chunk endpoints).

Compression scheme MILC [212] achieves similar query performance similar to that on uncompressed lists while keeping relatively low space overhead. MILC is built with SIMD and uses input partitioning, offset-based encoding (not delta) and a fixed codeword size within a block. A recursive partitioning data structure *Slicing* improves this by further partitioning sparse sequences [157]. One of the latest contributions to the VByte family introduced a variable length partitioning algorithm, where optimal VByte coding is used for each partition in order to maximize compression [160].

Clustered Index Compression A grouping of inverted list into clusters has been proposed in Clustered Elias-Fano coding [159], or in our previous work [A.6] for more generic context. The main idea is to merge lists with similar values. Then for each cluster a reference list is created, and all lists in the cluster are encoded with reference to that list. This clustering refers to whole list being clustered together, as opposed to clusters of similar integer values withing a single list, which is a property many encoding schemes exploit.

Asymmetric Numerical Systems (ANS) Some list compression algorithms with high frequency of repeated deltas rely on relatively new entropy coding – *Asymmetric Numerical Systems* [52]. The ANS entropy coding becomes less efficient with large alphabet sizes. Therefore a solution utilizing alphabet reduction stage has been used in [136], where ANS was used in conjunction with VByte, Simple and Packed encodings. These techniques were further adapted to include a two-dimensional context selection for blocks of inverted list (such as max and median value in a block), and further improvements [137].

Dictionary Index Compression Another approach is to partially encode the integer list using a dictionary. A *DIctionary of INTegers* [158] is based on 16-bit codewords for the most frequent patterns. The algorithms further encodes runs and exceptions as separate codewords. The decoding phase, which simply performs a lookup in the dictionary, is very efficient. This approach has only been implemented on CPU, however thanks to its 16-bit fixed codeword size could be very efficient on parallel architectures.

A GPU accelerated co-processor has been used as extension in MonetDB to compress data columns [56]. Multiple compression schemes were used, including run-length encoding, Null Suppression with Variable Length, Bitmap, Dictionary and their combinations. These combinations are carefully selected per each column by a planner that uses heuristics based on data properties in such column. The compression indirectly accelerates the query times due to decreasing data load times and due to only having to perform partial decompression for some compression plans.

Finally, several good comprehensive overview and experimentation works exist on inverted index compression. The latest survey [161] has wide scope of various techniques, includes additional historical techniques mostly on the CPU. Another good overview of CPU only techniques from the point of view of efficient decoding is available in [110].

5.2.1.1 Integer Lists Compressions on GPUs

Most of the GPU research focused on similarity search queries does not focus on index encoding. In a few cases, GPU is used for index decoding, which was previously indexed and encoded using CPU/SIMD.

A parallel version *ParaPFor* [8] of PFor has been implemented for GPUs, in which a linked list of exception is placed in the original segment. In the same work, a linear regression has been applied to list compression, which was then directly used for list intersection.

Information retrieval system *Griffin-GPU* [123] uses a parallel decompression scheme *Para-EF* (based on Elias-Fano code) to decompress inverted lists, and a parallel merge-based intersection algorithm.

Two schemes for index decompression were introduced in [131], GPU-BP (based on bit-packing) and GPU-VByte (based on variable byte). These GPU implementations achieve significant speedup on very long lists, and a minimal speedup on shorter lists, at the cost of a small size increment.

5.2.1.2 Integer Lists Compressions on FPGAs

Not a lot of work has been done solely on integer list compression on FPGA. A compression scheme using VByte with Huffman coding was accompanied by an FPGA based decompression [225]. BitWeawing [118], a bit-packed storage of database columns on FPGA, can be seen as a form of lossless compression Another work focuses on bitpacking compression [96], or a multi level bitpacking with prefix suppression [139].

5.2.2 Database Acceleration using GPUs

Significant focus in recent years closely related to similarity search has been on GPU acceleration of several database operations. This section describes GPU acceleration efforts on database operations; some of which were integrated as co-processors into existing DBMS's.

Another application of GPU accelerated database operations has been demonstrated on sorting large wide-key datasets [69]. Sorting is a common operation in databases both as part of query evaluation or a proxy to I/O operations. The work itself doesn't integrate the sort into any database, but demonstrates the capabilities on a database like data.

More applications integrated into DBSM have been implemented, such as predicates (comparisons, semi-linear query), boolean combinations, and aggregate operations (count, k-th largest number, sum, avg, etc.) [70]. The work was done on GPUs without integer arithmetic cores. This is no longer a limitation for modern GPUs, which have many integer arithmetic cores. Regardless, all of these operations are still relevant in modern DBSM.

An in-memory relational query co-processing system called GDB was a first comprehensive DBMS GPU accelerated solution [78]. It consisted of data parallel primitives (map, scatter, gather, scan, split, filter, and sort), data access methods (table scan, B+ trees, hash indexes), and relational operators (select, project, order, group and aggregate, and several joins). This process depends on computing a prefix scan on the individual join operation to allocate the device memory for the resulting output. This is a commonly used technique in modern GPU accelerated applications.

Similarity Join The problems of similarity join and self similarity-join (finds all objects within a dataset that are within a search distance) are an extension to *k-NN* similarity search. *Super-EGO* is a CPU only framework for similarity join [99]. The authors also provide a CPU parallel implementation, and an in depth analysis of selectivity factor of many algorithms. Original GPU implementation to relational joins (a predecessor to similarity join) were elaborated in [79] and consist of lock-free and atomic-operations-free three stage process. The first original approach to similarity join on GPU called *LSS* uses space filling curves [121]. *GPU-Join* is an implementation of self-join using a grid-based index, combined with index dimensionality reduction, reordering the data by the variance in each dimension, and distance calculation reduction [71]. *COSS* is subsequent work on self-join, which overcomes the curse of dimensionality by ignoring the coordinates values of the data points and instead uses distances to reference points, with several reference point placement strategies [51]. Many similarity join frameworks also elaborate the cases where their implementation is more efficient than a simple brute force search, which comes with a quadratic complexity based on the dataset size.

MapReduce Based Similarity Join A popular framework MapReduce [48] allows users to easily run large scale parallel operations on a cluster. These operations are specified in parallel friendly phases – map, shuffle, and reduce. Despite Google's departure from the framework and transition to more generic GCP Dataflow, the original implementation in Apache Hadoop and concepts still remain very popular. MapReduce is used to perform *k-NN* joins with Voronoi diagrams to partition the datasets [125]. The pipeline consists of two MapReduce jobs, where the first job finds the nearest pivots for both datasets, and the second phase find the nearest neighbors in each partition and then performs the reduce step to obtain the kNN join. Subsequent work on *k-NN* join has used only a single MapReduce phase, with supplementary histograms over values of the join attribute [179]. These histograms are then used to facilitate early termination and load balancing.

List Intersections A common operation for database purposes and information retrieval system are lists intersections. For uncompressed sorted lists intersection, a linear regression and hash segmentation techniques are used in [8]. Additional SIMD-enabled intersection algorithms, namely *V1*, *v3*, and *SIMD-Galloping* were introduced in [111].

Additional related GPU accelerated work includes key-value stores [230], or online transaction processing [80].

5.2.3 Exact Similarity Search on CPUs and GPUs

This section focuses on a more specialized operation from the area of *similarity search*, which is often represented by the exact *k-NN* (k nearest neighbors) search problem.

Substantial amount of work has been invested into GPU accelerating *k-NN* and related database operations.

Brute Force Multiple GPU accelerated implementations of exact *k-NN* algorithms exists. Relatively simple but efficient implementations use brute force approach. It usually consists of a computation of distance matrix to all the query points, and subsequent selection of nearest neighbors. These algorithms mostly vary in methods of pruning the search space and sorting algorithm used. In [106] radix sort is used, in [64] insert sort is used instead, and in [105] quicksort is used. A set of truncated sort algorithms is used in [192]. Slightly different sorting approach is due to [120], which uses merging by thread blocks to select the nearest neighbors. More advanced brute force GPU implementations [116] uses truncated merge sort, or a combination of selection sort, quicksort and heaps-based algorithms [17].

Brute Force With Partitioning Brute force implementations usually achieve high performance in cases where GPU resources can be fully saturated (i.e. device memory to hold the data space, and shared memory to hold the query space). However, these approaches usually fall short in case of small queries, and do not support larger data spaces. In larger data spaces, some form partitioning is used, which is processed in sequence, such as in *GPU-FS-kNN* [10]. A modern and efficient implementation [209] uses symmetrical data space partitioning and buffering of nearest neighbor results to save device memory.

Brute Force With Pruning Strategies Another class of GPU *k-NN* implementations rely on usage of reference points to prune the overall distance evaluations. Example of this is *GPU-SME-kNN* [74], which uses iterative distance evaluation to subsets of reference points. Distances of queries to these reference points are then merged to provide candidate results sets. Several implementations also provide multi-GPU implementation, where additional synchronization is used to accumulate candidate results [102, 133]. A different heterogeneous (GPU and CPU) implementation utilizes threshold pruning compression with a heap based partial sorting, where a GPU is used for distance computation and a CPU for sorting [134]. The pruning is used to discard entries from the distance matrix that cannot make candidates for the nearest neighbors.

Space Filling Curves All GPU accelerated algorithms naturally rely on highly parallel operations, such as similarity hashing, brute force similarity evaluations and sorts. Notable example of competitive multi-node CPU only implementation called *FLANN* [141] uses space filling structures (randomized kd-trees) and a newly introduced algorithm based on priority search k-means tree. Spatial indexing using R-trees has also been implemented on the GPU [127], and using buffer kd-trees for exact nearest neighbor search [66].

The GPU-accelerated brute force *k-NN* approach was also used for calibration of *eNN10* classifier using Artificial Bee Colony (ABC) algorithm [188].

Exact similarity search is very often used in its simplest form in document retrieval and full-text search systems, such as Apache Lucene [25], Apache Solr [183], and Elasticsearch [67].

Another system uses WAND (Weak AND) ranking algorithm of [31] with two parallel strategies, one for single queries and one for batch of queries [61]. For the latter case, three threshold sharing policies are used to retrieve top-k results. In another work [62], the same authors adapted the system for the MaxScore ranking.

5.2.3.1 Similarity Search Using Heterogeneous Architectures

Most GPU accelerated system only use CPU for general IO, scheduling and other orchestration. Other systems try to leverage the advantages of both CPU and GPU to carry the bulk of the workload, splitting the queries and/or processing layers to the more appropriate of the processing units. Some systems go even further by using either FPGAs of ASICs instead of more generic GPUs.

FPGA One of the early FPGA solutions was used for text search and document relevance evaluation, without any preprocessed index [72]. Later, a suite called Pinaka [226] including inverted index search engine, decoder, matcher, and ranker was the first comprehensive FPGA accelerated web search engine.

ASICs Inverted index search system called *IIU* [81] uses a specialized hardware architecture to accelerate inverted list queries, including intersections and unions. IIU uses a standard pipeline of many inverted index search engines: load inverted lists for all query terms, decompress, perform set operations, and evaluate the relevance for top k results. The system uses inverted lists compression using a combination of delta encoding and per-block bitpacking, where blocks are dynamically partitioned. IIU uses a concept of cores to load balance the workload, which is controlled by schedulers. The system was evaluated using a hardware simulator at the register-transfer level (RTL).

Heterogeneous by Design An information retrieval system [49] is one of the first that uses heterogeneous computing with CPU and GPU scheduling to minimize response time based on the queries. The system uses inverted lists compressed facilitating Rice and PForDelta encoding. A GPU query processing method was proposed as an improvement over [8] using pruned list caching to fit more lists in the GPU memory [211]. This system only performs batching on queries to be executed on GPUs when there are enough caches queries, otherwise it reverts to CPU. Griffin [123] is a heterogeneous CPU and GPU implementation of information retrieval system (performing lists intersections). The system uses a dynamic intra-query scheduling algorithm that breaks a query into sub-operations and schedules them to the GPU or to the CPU based on their runtime characteristics.

Some systems solve a similar problem of data series similarity search. SING [155] is a specialized CPU and GPU accelerated system, where the index structure consists of summaries based on Piecewise Aggregate Approximation (uses mean value for approximation, stored in GPU memory). MESSI [154] is a tree based indexing structure (stored on main memory) on multi-socket and multi-core architectures.

5.2.4 Approximate Similarity Search

In many scenarios, most often due to data size, it is not feasible to retrieve k exact nearest neighbors – *k-NN*. Hence more generic problem is defined as approximate nearest neighbor – *ANN*; or k approximate nearest neighbors – *k-ANN*. The approximation constraint is usually expressed by a constant $c > 1$, which denotes a maximal multiplicative error from the exact nearest neighbors. Similarly a $\tau > 0$ constraint is used as the additive error from the exact nearest neighbors.

Many algorithms for *k-ANN* build a similarity graph that serves the purpose of an index based on a similarity measure, where each data point is represented by a vertex, and each vertex is connected with its true nearest neighbors. Examples of these are *KGraph*[6], *HNSW* (Hierarchical Navigable Small Word Graph) [129, 130], *NN-Descent* [50], and the NGG family of algorithms [92, 93].

Other class of *k-ANN* algorithms relies on space partitioning using trees, such as kd-trees [34] or M-trees [227], adaptive configuration of hierarchical k-means trees in *FLANN* [142], random projections tree in *Annoy*[7], or use of multiple random projection trees with a voting scheme in *MRTP* [89].

MIH (Multi-Index-Hashing) [146] is a technique that aims to reduce the number of empty buckets on binary codes by splitting them into shorter disjoint binary codes. This hashing is very effective for Hamming distances. The general technique is also applicable for exact nearest neighbor search [147] or for cosine similarity [54].

Majority of the *k-ANN* algorithms rely on some form of locality sensitive hashing (LSH), which are described in more details in Section 5.2.4.2.

Different approach to LSH makes use of the data distribution in order to create a similarity preserving index. The most common example of this is product quantization [97, 65, 14], which has been improved throughout the last decade. Product quantization decomposed the space into cartesian product of lower dimensional spaces, then performs principal component analysis (PCA), followed by quantization step.

Another class of data dependent algorithms is using deep learning in what is referred to as *deep hashing* or *semantic hashing* [169, 163, 228]. An overview and evaluation of supervised deep hashing vs LSH is available in [33].

A good overview and a benchmark suite called *ANN-Benchmark* with recent experimental evaluation of *k-ANN* algorithms was presented in [13]. Many CPU and SIMD implementations are readily available as Docker images, with automated testing dataset downloads. GPU support is very limited.

One of the latest overviews with experimental evaluation is available in [117]. The overview focuses on both locality sensitive hashing (data independent method) and what

[6]`https://github.com/aaalgo/kgraph`
[7]`https://github.com/spotify/annoy`

they refer as Learning to Hash, which is a set of data dependent hashing methods for *ANN*. These data dependent methods are further split into pair-wise similarity preserving, multi-wise similarity preserving, implicitly similarity preserving, and quantization methods.

A brief overview of approximate similarity search, current algorithms and main applications is alsoavailable in [53].

5.2.4.1 Approximate Similarity Search on GPUs

Very popular approach (used in GPU accelerated implementations) to k-*ANN* is based on locality sensitive hashing (LSH), which can be created fully in parallel and results in a bucket or list data structure that can be searched in parallel as well. Furthermore, LSH and similarity hashing in general allows to partially overcome the high dimensionality problem. Running times of LSH based algorithms are often competitive with existing brute-force implementations on problem sizes suitable for brute-force algorithms, and additionally allow to efficiently process much larger problems.

Initial work on k-*ANN* has been done using LSH and two level bucket hashing, using universal and cuckoo hashes to reduce the total number of buckets searched [152]. Subsequent improvement of the work has extended the two level hashing with RP-trees [153].

Another algorithm adapted to multimedia applications uses hypercurves – a concurrent index based on space filling curves and has a hybrid CPU-GPU implementation with a scheduler [203].

Different approach based on product quantization and vector quantization has introduced a two level Product Quantization Tree (PQT) and reranking step to approximate exact distances between query and database points [218].

FAISS is a library using product quantization capable of multi-GPU sharding of the dataset [98]. This has been later improved with adaptive termination in [115].

5.2.4.2 Locality Sensitive Hashing

Locality sensitive hashing (LSH) originally introduced in [91]. LSH is family of hashing functions for dimension reduction with probabilistic mapping of input objects into a much smaller number of buckets. The property is that similar input objects in the domain of these functions have a higher probability of colliding in the buckets than non-similar objects. The hashing algorithms can be used to find the exact nearest neighbor in Euclidean space under certain conditions [45].

Two popular representatives of LSH are MinHash [30] and SimHash [39]. The theoretical and practical differences between these two have been elaborated in [186], where MinHash has been established to be more suitable for resemblance similarity (which is defined for binary vectors) as well as cosine similarity (defined for real values).

The original LSH-based k-*ANN* algorithm [152] first processes all different k-*ANN* queries in parallel, but iterates over all L hashing functions. Then for each hashing function it computes in parallel all the keys into buckets, and unique keys for the cuckoo hashing table (using prefix sum), which are then inserted into multiple hash sub-tables. During querying, when computing LSH and preforming hashing, the parallelism is limited to the size of the query, i.e. the number of query points.

Subsequent work [153] introduced several improvements on top of the previous algorithm. RP tree is used to cluster the data points. This is similar to kd-trees with splitting along random directions, and therefore relies on reasonable spatial properties of the data. Then LSH table is constructed for all the clusters from previous step, with two extra projections used to preserve the neighborhood relationships. The parallelism is further improved by utilizing additional parallel sort of distances to query points.

FALCONN (FAst Lookups of Cosine and Other Nearest Neighbors) [3] is a popular LSH library for cosine similarity (also called angular distance), including a multiprobe version (one that examines multiple nearby buckets for a reduction in total hash space).

More recent examples of LSH with small signature space (linear space in size of the dataset) include *LSB-forests* (Locality-Sensitive B-tree Forest) [202], *C2LSH* (Collision Counting LSH) [63], SRS [199], LazyLSH [232].

QALSH (Query-Aware LSH) uses information from a query (anchor) to derive bucket distribution in LSH [86] and improve error bounds of the search. The LSH has also been used with generalized weighted euclidean distance in [108], when weights are supplied with each query. Another method uses Longest Circular Co-Substrings (LCCS) over hashed objects as part of the search framework *LCCS-LSH* [109]

5.2.4.3 Locality Sensitive Hashing on GPU

A GPU implementation [126] further improved the efficiency by utilizing *multi-probe LSH* [128]. During querying, for each query point, the algorithm first explores buckets with the highest probability of containing nearest neighbors. The extraction of query points is done using deterministic skip-lists on the GPU.

Recent and comprehensive overview of all GPU data-parallel hashing techniques is available in [114], including open-addressing (probing), perfect hashing (collision-free), grid-based spatial hashing, locality sensitive hashing, separate chaining. The overview contains comprehensive introduction the hashing problem on GPU in general and is not restricted to locality sensitive hashing.

5.2.5 GPGPU and CUDA

A modern GPU has a special purpose architecture designed for high throughput data parallel computations. Originally used for graphic processing, now often used for general purpose (GPGPU) computing.

Each GPU consists of *streaming processors* (SP), which are part of larger *streaming multiprocessors* (SM). Each SM operates a *Single Instruction Multiple Threads* processing model (SIMT), i.e., streams, providing instruction level parallelism. Different SMs can execute code independently of each other. A kernel is the largest GPU executable logical unit, configured and dispatched at runtime by the CPU. Each kernel is divided into thread blocks, which are executed independently of each other, and are scheduled onto SMs. Due to SMs limited resources, only a limited amount of thread blocks can be active at a time. Each thread block is further scheduled within the SM in smaller units called warps. Warps, also known as cooperative thread arrays, are then the smallest units of SIMT execution. This model can be seen as a two level parallelism (at the thread block in SM level, and SIMT in warp level) and allows to use scheduling to overcome data access inefficiencies or lack of compute resources, and improve instruction level parallelism.

The memory model is likewise hierarchical. The main *global* memory of the GPU is a fast GDDR with bandwidth in the 100 GB/sec to 1 TB/sec range. The global memory of size up to 24GB in contemporary GPUs is relatively small compared to the size of main memory often available in HPC machines. Therefore one of the main bottlenecks is between the main memory and GPU memory, which is often restricted to relatively slow one-way transfer speeds of 12-16 GB/sec on PCIe 3.0, and 24-32 GB/sec on PCIe 4.0. Additional L2 cache is used for the whole global memory, and L1 cache (also used for shared memory) that is used per SM. Then each SM has a limited amount of registers which can be used by individual threads. Memory caching and access patterns in general are extremely important, since memory stalls result in either a long wait or force re-scheduling of warps.

Additional technologies by NVidia for HPC purposes allow for external memory access, such as GPUDirect Storage for accessing NVMe drives, GPUDirect RDMA for network access to GPU memory; or NVlink and NVSwitch for multi-GPU memory interconnect.

When designing algorithms for GPU, several characteristics have to be taken into account: GPUs in general prefer much higher ratio of arithmetic operations to memory operations; high data parallelism is required to fully utilize the compute resources; and high locality is required to minimize the number of memory transactions.

We use the CUDA Toolkit and parallel programming libraries for NVidia GPUs. Further details about the GPGPU compute and memory models, design practices, and a good starting point are available for example in the official CUDA Toolkit documentation [148].

5.3 GENIE

GENIE (Generic Inverted Index on the GPU) [A.2, A.5] is a GPU-based highly parallel inverted index framework implemented to efficiently support a *match count* model (see Section 5.3.1) between objects and queries. In GENIE's matching layer, we encode dimensions and all possible values for these dimensions into a keyword. From these keywords, we construct an inverted index, which is dynamically loaded onto the GPU for approximate nearest neighbor evaluation. Each query is converted to a set of keywords, then the inverted lists corresponding to these keywords are scanned, and match counts are evaluated between each query and all objects from the dataset, with dynamic count pruning in order to prevent calculation of all distances.

GENIE is designed to support similarity search on various data types. Many problems can be converted to similarity search using locality sensitive hashing (see Section 5.3.2) or shotgun and assembly (see Section 5.3.4). The internal representation of GENIE is agnostic of the data source, however the data properties have to be taken into account in order to determine the matching configuration and desired precision (see Section 5.3.1). Figure 5.2 shows a high level relationship between data types and GENIE.

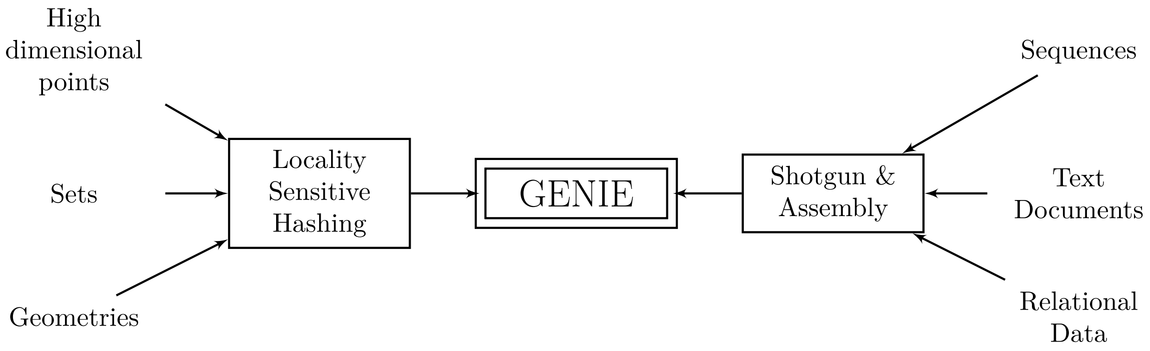


Figure 5.2: Generic nature of GENIE, showing the relationship with different data types.

Note that in all the cases when data is preprocessed (using both LSH and S&A) all the attributes are already discretized, but in the case of using raw data, continuous valued attributes must be discretized manually according to a preprocessing policy.

GENIE was originally designed to be used for approximate nearest neighbor search for lazy machine learning as part of a *GENIE & LAMP* (LAzy Mining Paradigm) project[8], with applications such as path prediction [234] or time series prediction [233].

5.3.1 Problem Statement

A *dataset* D is a set of N objects $O_1, \dots, O_N$, where each object $O_i = (v_1, \dots, v_d)$ is an d-dimensional vector from a *universe* U.

A *point query* $Q \in U$ is a d-dimensional point $Q = (v_1, \dots, v_d)$. A point query is often represented in the dataset domain, as opposed in the hash set.

A *range query* R is an d-dimensional vector of ranges

$$R = (r_1, r_2, \dots, r_d) = \left(\left[v_1^L, v_1^H\right], \left[v_2^L, v_2^H\right] \dots, \left[v_d^L, v_1^H d\right]\right)$$

where the dimensionality of the range query is the same as dimensionality of the dataset[9]. The range query is often used to query a range of inverted lists in locality preserving inverted index space (such as for LSH). A point query can be seen a degenerate version of a range query.

Definition 5.3.1 (Match Count) *Given a range query $R = (r_1, r_2, \dots, r_d)$ and an object $O = (v_1, \dots, v_d)$, a* range match count *is a function* $\texttt{rmc} : (r_j, O) \to [0, 1]$ *indicating whether*

[8]https://www.comp.nus.edu.sg/~atung/gl/

[9]The dimensionality of the query can be lower than the dimensionality of the dataset, GENIE supports this concept, however we do not consider it in this work.

Sample Data

O_1	10	6	52	62	0	O_{11}	54	1	0	16	8
O_2	16	50	12	0	1	O_{12}	20	13	53	50	0
O_3	10	11	58	69	4	O_{13}	28	86	16	1	4
O_4	25	75	19	5	8	O_{14}	13	102	22	1	0
O_5	10	8	56	64	0	O_{15}	18	3	20	40	18
O_6	13	75	13	0	4	O_{16}	97	18	9	9	0
O_7	17	28	64	37	0	O_{17}	35	30	2	3	27
O_8	8	78	21	1	2	O_{18}	36	27	3	0	1
O_9	3	15	2	0	11	O_{19}	11	8	2	3	6
O_{10}	3	14	1	0	0	O_{20}	42	21	4	5	4

Sample Point Query

Q_1	18	28	3	1	6

Sample Range Query

R_1	16-20	26-30	1-5	0-3	4-8

Table 5.1: Sample dataset and queries represented by 5 dimensional objects.

the dimension value v_i of object O is contained in the range r_j of query R – in dimension j. The range query match count `MC` $: (R, O) \to \mathbb{N}$ *is then the sum of range match counts* `rmc` *over all the dimensions / ranges from the query:*

$$\texttt{MC}: (R, O) = \sum_{r_i \in R} \texttt{rmc}(r_i, O)$$

Note that the maximum match count `MC` therefore cannot exceed d – the dimensionality of the dataset and the query space. Furthermore for queries identical to an object, the match count will always be equal to d between such query and object.

The objective of the *k-ANN* search is to return at least k objects O_i from the dataset D with the highest match count. In a case where multiple objects score the same, resulting in more than k objects as candidates for the result set, the objects with the lower qualifying match count may be pruned at random.

Table 5.2 shows an example of a inverted index and a range query execution on that index. The index is generated from the sample data presented in Table 5.1. *Dimensional Metadata* store the lowest and highest values in all the dimensions. The lowest value is then used as offset into the *Inverted Lists Metadata*, where all the values start from 0 for all dimensions. The *Inverted Lists Data* is a concatenation of all inverted lists into one array.

Dimensional Metadata

Dimension	0	1	2	3	4
Lower value bounds	3	1	0	0	0
Upper value bounds	97	102	64	69	27

Inverted Lists Metadata

Dim.	Value	Length	Offset	Dim.	Value	Length	Offset	Dim.	Value	Length	Offset	Dim.	Value	Length	Offset	Dim.	Value	Length	Offset
0	0	2	0	1	0	1	20	2	0	1	40	3	0	5	60	4	0	7	80
0	5	1	2	1	2	1	21	2	1	1	41	3	1	3	65	4	1	2	87
0	7	3	3	1	5	1	22	2	2	3	42	3	3	2	68	4	2	1	89
0	8	1	6	1	7	2	23	2	3	1	45	3	5	2	70	4	4	4	90
0	10	2	7	1	10	1	25	2	4	1	46	3	9	1	72	4	6	1	94
0	13	1	9	1	12	1	26	2	9	1	47	3	16	1	73	4	8	2	95
0	14	1	10	1	13	1	27	2	12	1	48	3	37	1	74	4	11	1	97
0	15	1	11	1	14	1	28	2	13	1	49	3	40	1	75	4	18	1	98
0	17	1	12	1	17	1	29	2	16	1	50	3	50	1	76	4	27	1	99
0	22	1	13	1	20	1	30	2	19	1	51	3	62	1	77				
0	25	1	14	1	26	1	31	2	20	1	52	3	64	1	78				
0	32	1	15	1	27	1	32	2	21	1	53	3	69	1	79				
0	33	1	16	1	29	1	33	2	22	1	54								
0	39	1	17	1	49	1	34	2	52	1	55								
0	51	1	18	1	74	2	35	2	53	1	56								
0	94	1	19	1	77	1	37	2	56	1	57								
				1	85	1	38	2	58	1	58								
				1	101	1	39	2	64	1	59								

Inverted Lists Data

0x00	8	9	7	0	2	4	18	5	13	1	6	14	11	3	12	16
0x10	17	19	10	15	10	14	0	4	18	2	11	9	8	15	19	17
0x20	6	16	1	3	5	7	12	13	10	9	8	16	18	17	19	15
0x30	1	5	12	3	14	7	13	0	11	4	2	6	1	5	8	9
0x40	17	7	12	13	16	18	3	19	15	10	6	14	11	0	4	2
0x50	0	4	6	9	11	13	15	1	17	7	2	5	12	19	18	3
0x60	10	8	14	16												

Match Counts

Object	16	17	18	1	5	6	8	9	12	19	10	11	13	14	2	3	7
Count	3	3	3	2	2	2	2	2	2	2	1	1	1	1	1	1	1

Table 5.2: Example of inverted index stored in GENIE.

The highlighted cells in Figure 5.2 show a lookup process for the following query: $R_1 = ([16, 20], [26, 30], [1, 5], [0, 3], [4, 8])$, which in this case is a modification of point query Q_1 with a range of 2 in all dimensions. For example in the range query R_1 has a value range of $[16, 20]$ in dimension 0, so when querying the Inverted List Metadata, we use a modified range of $[13, 17]$ instead, because the lowest indexed value in dimension 0 for the whole dataset is 3. The resulting counts are shown at the bottom of the table, with top-3 results highlighted. Note that for example in dimension 3, the queried range is $[0, 3]$, which maps to three lists. These 3 list all belong to the same query, therefore can be greedily merged by the query compiler for execution into one virtual inverted list of length 10.

5.3.2 Using LSH For Approximate Nearest Neighbors

High dimensional data is often transformed using LSH (see Section 5.2.4.2). This preprocessing step is optional in GENIE, as the data locality can be partially exploited using ranged queries in GENIE (if dimensionality is low enough and the dataset space is discretized). After preprocessing data using LSH, we can build an inverted index such that each bucket from LSH corresponds to an attribute in GENIE.

The dataset D is transformed into a hash space using a set of locality sensitive hashing functions, and the result hashes correspond to the preprocessed data points that are loaded into GENIE for *ANN* search. The hash space preserves similarity properties to original real world data, with the intention of reducing a potentially high dimensionality and high range of values of the original dataset.

For LSH processing, the requirement is that each object from the dataset is processed with a family $\mathcal{H}$ of (d_1, d_2, p_1, p_2)-sensitive similarity hash functions [91], where similarity of two objects p and q results in high probability of their hash function $h(\cdot)$ yielding the same hash.

Definition 5.3.2 ((d_1, d_2, p_1, p_2)-sensitive similarity hash functions) *Let $d_1 < d_2$ be two distances. A function family $\mathcal{H}$ is (d_1, d_2, p_1, p_2)-sensitive if for all functions $h \in \mathcal{H}$:*
- ◦ *If $dist(p, q) \leq d_1$, then $Pr[h(p) = h(q)] \geq p1$*
- ◦ *If $dist(p, q) \geq d_2$, then $Pr[h(p) = h(q)] \leq p2$*

The locality sensitive hash function is then defined by having a collision probability equal to the similarity measure between the two data points p and q:

$$Pr[h(p) = h(q)] = sim(p, q)$$

The similarity function $sim(p, q) \in [0, 1]$, where 1 means the objects p and q are identical.

Note that each similarity measure may require a differed LSH function family in order to comply with the locality sensitivity property.

The universe of all the object hashes would be ℓ-dimensional integer space, with each dimension corresponding to a single hash function, and the range of values corresponding

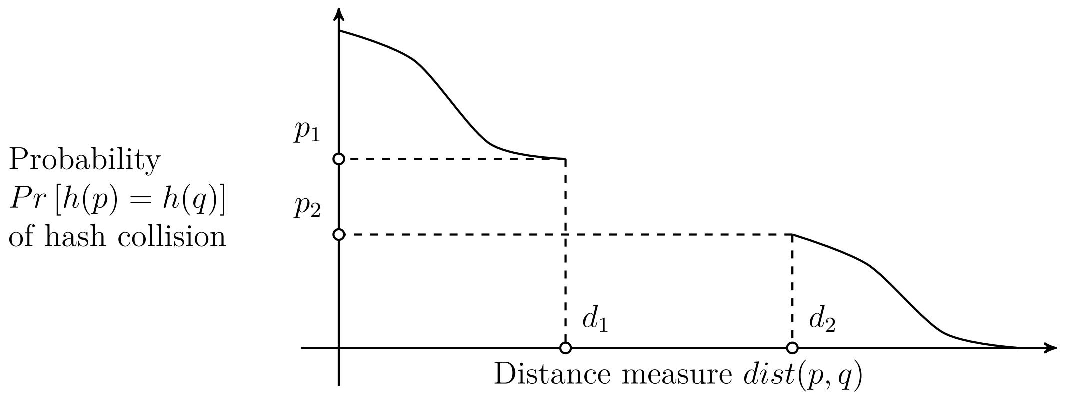


Figure 5.3: Example of (d_1, d_2, p_1, p_2)-sensitive hash function.

to the range of values of that hash function. This space is often very large and the resulting index may therefore be larger than the dataset itself, hence several reduction techniques can be used. See Section 5.3.2.1 for details.

A common definition of *ANN* search relies on an approximation ratio c, resulting it what is known and the *c-ANN* search problem.

Definition 5.3.3 (c-Approximate Nearest Neighbor) *Let $dist(\cdot,\cdot)$ be a distance function between two points from the universe (e.g. Euclidean distance). When searching a dataset of points $O_1, O_2, \ldots, O_N$ for a query Q under, given an approximation ratio $c > 1$, the* c-Approximate Nearest Neighbor *c-ANN returns an object O_R, which is with high probability similar to the true nearest neighbor O_T:*

$$dist(O_R, Q) \leq c \cdot dist(O_T, Q)$$

In many sources, the *c-ANN* is instead referred to as *ϵ-ANN*, which simply uses an additive approximation ratio such that $c = 1 + \epsilon$.

This definition of *c-ANN* can be altered to work with a similarity measure $sim(\cdot,\cdot)$ as opposed to distance function. For this reason, we need an approximation ratio $c' < 1$, so that the similarity constraint is $sim(O_R, Q) \geq c' \cdot dist(O_T, Q)$.

A match count model introduced in GENIE behaves differently from inverted list intersection used in other *ANN* search systems. However, the match count model can still be effectively used for *ANN* search.

Definition 5.3.4 (τ-Approximate Nearest Neighbor) *When searching a dataset of points $O_1, O_2, \ldots, O_N$ for a query Q under a similarity measure $sim(\cdot,\cdot)$, the* Tolerance Approximate Nearest Neighbor *τ-ANN returns an object O_R, which is with high probability similar to the true nearest neighbor O_T:*

$$|sim(O_R, Q) - sim(O_T, Q)| \leq \tau$$

Both definitions are also extended for *k-c-ANN* and *k-τ-ANN* search, where the conditions are applied to all the returned vs true nearest neighbors.

5.3.2.1 Re-Hashing

The range of values of LSH hashes can be further reduced in GENIE by using a second layer of hashing functions, called *re-hashing* in GENIE. Re-hashing is done using a random projection function, specifically in GENIE the hash function used is Murmurhash3 [9]. See the technical report [A.5] for more details on LSH and re-hashing in GENIE, theoretical analysis and case studies.

The hashed objects $O_i \in D$ will then be represented in the dataset as $O_i^{LSH} = [r_1(h_1(O_i)), r_2(h_2(O_i)), \ldots, r_\ell(h_\ell(O_i))]$. The queries need to go through the same process, using the same hash functions (and additional random projection functions), therefore $Q^{LSH} = [r_1(h_1(Q)), r_2(h_2(Q)), \ldots, r_\ell(h_\ell(Q))]$

5.3.3 Examples of LSH Function Families

The original idea of LSH function families [91] applies almost exclusively on points the Hamming space. It is possible to efficiently hash points in Euclidean space directly, without the need to embedding them into lower spaces first, which results in increased query time and error [45].

Probably the best known LSH function family is *MinHash* [30], which approximates the Jaccard similarity $J(P,Q) = \frac{|P \cap Q|}{|P \cup Q|}$ of two sets P and Q. MinHash works by computing k independent random hash functions $h_1, h_2, \ldots, h_k$ and keeping track of minimum values of these hashes over all points from the dataset. For LSH purposes, the hash values are split into b bands, and estimation of similar candidates is selected as across all bands of the hashes.

A common LSH method called *SimHash* [38] is used for cosine similarity $cos(\vec{p}, \vec{q}) = \frac{\vec{p} \cdot \vec{q}^T}{\|\vec{p}\| \|\vec{q}\|}$ of two vectors $\vec{p}$ and $\vec{q}$. SimHash works by choosing a set of hash functions, where each hash function was a random hyperplane $\vec{r_i}$ and $h_i(\vec{u}) = \mathtt{sign}(\vec{r} \cdot \vec{p})$. These hashes are then combined into a bitwise-average.

Commonly used LSH function families used in machines learning tasks, for example for Support Vector Machines, include the *Random Binning Features* (RBF) [164], which partitions the space into a grid on random cell size g, with a random shift vector $\vec{u} = [u_1, u_2, \ldots, u_d]$. The hash function of object $O = [o_1, o_2, \ldots, o_d]$ is then defined as:

$$h(O) = \left[\left\lfloor \frac{o_1 - u_1}{g} \right\rfloor, \left\lfloor \frac{o_2 - u_2}{g} \right\rfloor, \ldots, \left\lfloor \frac{o_d - u_d}{g} \right\rfloor \right]$$

The expected collision probability of RBF is then $Pr\left[h(p) = h(q)\right] = k(p,q)$, where $k(p,q) = \exp(\frac{-\|p-q\|}{\sigma})$ is the Laplacian kernel.

Another very popular LSH function family used is based on p-stable distribution [45], which work for any ℓ_p norms space. For a vector $\vec{a} = [a_1, a_2, \ldots, a_d]$ whose each entry is chosen independently from a p-stable distribution, and b is a real number chosen uniformly from a range $[0, r]$.

We can see the hash function has that we split a real line into segments of size r and assign hash values to vectors based on which segment they project onto. The hash function of object $O = \vec{o}$ (objects are d-dimensional vectors) is formally defined as:

$$h(\vec{o}) = \left\lfloor \frac{\vec{a}^T \cdot \vec{o} + b}{r} \right\rfloor$$

Examples of p-stable distributions include the Cauchy distribution for l_1 space and the Gaussian (normal) distribution for the l_2 space.

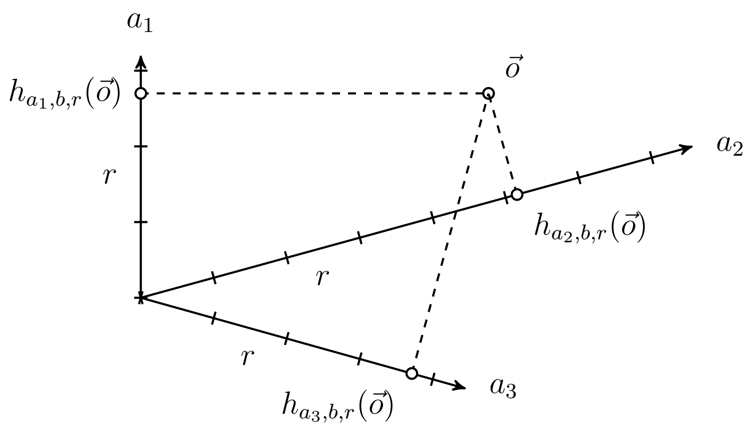


Figure 5.4: Example of locality sensitive hashes based on p-stable distribution in 2 dimensional space. Using 3 hash functions with vectors a_1, a_2, and a_3 chosen randomly from a p-stable distribution.

Other approaches to similarity hashing are: *Sdhash*, that (as opposed to previously described techniques) selects statistically improbable features, that are unique for each object [167]. *Sketching* is another method based on hashing object n-grams multiple times to make a sketch of the object. Sketching is described in [132, Chapter 19].

5.3.4 Preprocessing Data Using Other Sources

It is also possible to use GENIE for similarity search without LSH. This can be achieved by splitting each object into smaller units, then constructing the inverted index based on containment of that unit in the objects (or documents). The match count of GENIE is then used as an estimate of the similarity measure.

For searching strings (i.e. sequences), we split the string into a set of ordered n-grams (pairs of n-gram and the number of occurrences of that n-gram). The set of ordered n-grams then serves as the key domain of the inverted list. The match count of GENIE then returns the sum of minimum number of occurrences of each n-gram in the dataset and in the query.

For searching documents and object documents, the keyword domain of the inverted index is based on the words from the documents. The match count then represented the number of common words between the two documents.

When searching relational data, GENIE can be simply used as is, with the attributes (columns) representing dimensions, attribute values representing the indexed values (with potential discretization of real valued attributes), and the inverted index contains primary key as the docIDs. Note that the match count model is not weighted in the current design.

5.4 Compressed Match Counting in GENIE

In this section we present GENIE's query processor with modification to data preprocessing, and an extension to matching GPU kernels capable of match counting on compressed inverted index. Our objective is to extend GENIE with efficient inverted index compression schemes, inverted list balancing and partitioning. We also elaborate on ideal mapping of query execution to the parallel model of GPUs in order to maximize query throughput, response times and to minimize the total size of GPU-resident inverted lists cache.

By reducing the space on GPU memory required for query execution, we are able to speed up the execution time by reducing the cost of data transfer, run more queries at the same time by having more data present at a time, and decrease the threshold when large datasets have to be incrementally loaded.

We first start by analyzing the properties of raw inverted lists and tables created by applying LSH on real world and synthetic datasets. Based on these observations, we find strategies for reducing the overall size of indices, while identifying ways to improve parallelism when executing queries on the GPU.

The compressed inverted index supports sharding, which is useful for the purposes of multi-loading large datasets and for purposes of distributed GENIE execution, where queries to one dataset are distributed as well.

5.4.1 CUDA Kernel Models For Decompression

In this section, we describe decoding kernels, which are integrated together with the matching kernel in GENIE. The codec selection for inverted index is user configurable.

5.4.1.1 Delta Encoding

Generally docIDs in inverted lists are sorted, and therefore can be represented as *deltas* (sometimes also called gaps, or differences) between two consecutive docIDs. Delta encoding is used to generate smaller number for subsequent encoding steps.

We assume that all docIDs are integers smaller than 2^{32}. This is the same assumption that is taken across all of the related work, as it is very unlikely that an indexing system will ever hold more than this amount of documents in a single shard.

Delta encoding is parallel scan type of operation, so the time complexity is $\mathbb{O}(\log(n))$. A single step (constant time) delta encoding can be implemented by only encoding the delta from the first element of the parallel block. Full sequential delta encoding requires a logarithmic amount of steps. Therefore a compromise can be made between these two extremes, which is referred to as vectorized delta, or vectorized differential encoding. This encoding can achieve significant performance gain in CPU SIMD implementations [110]. However, in our experiments on the GPU, full sequential encoding has always shown better compression with minimal computational overhead.

5.4.1.2 Varint Encoding

Varint is a popular encoding scheme used for integer lists, and has been partially vectorized using SIMD instruction sets. The premise of the algorithm is relatively simple. Each byte is split into 1 and 7 bits, the highest one bit serves as metadata indicating whether a new integer starts at the byte location. The following bytes with a zero leading bit are then joined into the decoded integer.

Our representation of encoded data requires each thread to access two adjacent 32-bit words from the encoded array. The key logic is that each thread extracts only integers that *begin* in its primary word. For this purpose, each thread requires to have access to the next word in order to decode the last integer if there is any overflow.

The first improvement of the decoding process is to externalize the control bits from the data bits into a separate array. This idea has been first used for SIMD processing in [113]. For a list of N encoded 32-bit words, there will be at most $\frac{N}{8}$ control bits, so for maximum length of 1024 encoded integers[10], there are at most 128 32-bit words in the control array (in a case where all integers are of length 4 bytes).

One inefficiency in the current algorithm still remains. Because the representation of control bits spans variable amount of bits, each thread needs to do excessive binary operations, after which it ends up with anywhere between 0 to 4 integers to decode. We use an idea originally proposed in Varint-GB [46], where exactly 2 control bits are used per integer. These two bits are serialized in external array, just like in the previous case. Decoding starts by loading the whole control array, followed by scatter operation to distribute the 2 control bits to each thread. Now each thread knows how many integers to decode, ranging from 1 to 4 integers. Note that in this change the size of the control array depends only on the number of integers to be encoded, unlike in the original variant where the control size depends on the values as well. This also enables the algorithm to use 8-bit codewords and therefore simplify the decoding steps. The decoder continues by loading the encoded array, followed by a shuffle operation to fetch the next word with overflow encoded data.

[10]Theoretically, when using Varint, it is possible to achieve an encoding of larger size than the original data. If that is the case, the inverted index encoding component of GENIE will discard the encoded list and use raw index format instead. Note that for long lists, such as 1024 integers, it is not possible to reach this edge scenario in practice, because the delta encoding will result in deltas of expected size smaller than the necessary 29 bits to cause negative compression.

Then the decodes performs only one parallel scan to decode deltas. Each thread ends up with 1 to 4 integers.

The parallel model we use has one thread block decode and match count a single inverted list, i.e. a query component. For a maximum list size of 1024, the decoder uses up to 1024 threads, depending on the encoded array size. The access to global memory for both encoded data and control array is fully coalesced. We use warp level shuffle instructions and small amount of shared memory to scatter the data across the warp as necessary for each thread.

The decoding process of Varint is visualized in Figure 5.5 and the detailed algorithm is described in Algorithm 5.1.

5.4.1.3 Bitpacking Encoding

The premise of *Bitpacking*[6] is to take a fixed number of integers and encode them all using the same amount of bits b. The amount of bits b is determined by the largest integer. We use the standard block size of 32 integers, this is also known as *Bitpacking32*. For each block, one additional byte is used to encode b.

Similarly to Varint, we also separate the control bytes into a different array. For a list of N encoded 32-bit words, there will be exactly one control byte, so for maximum length of 1024 encoded words, there will be exactly 32 bytes of the control array.

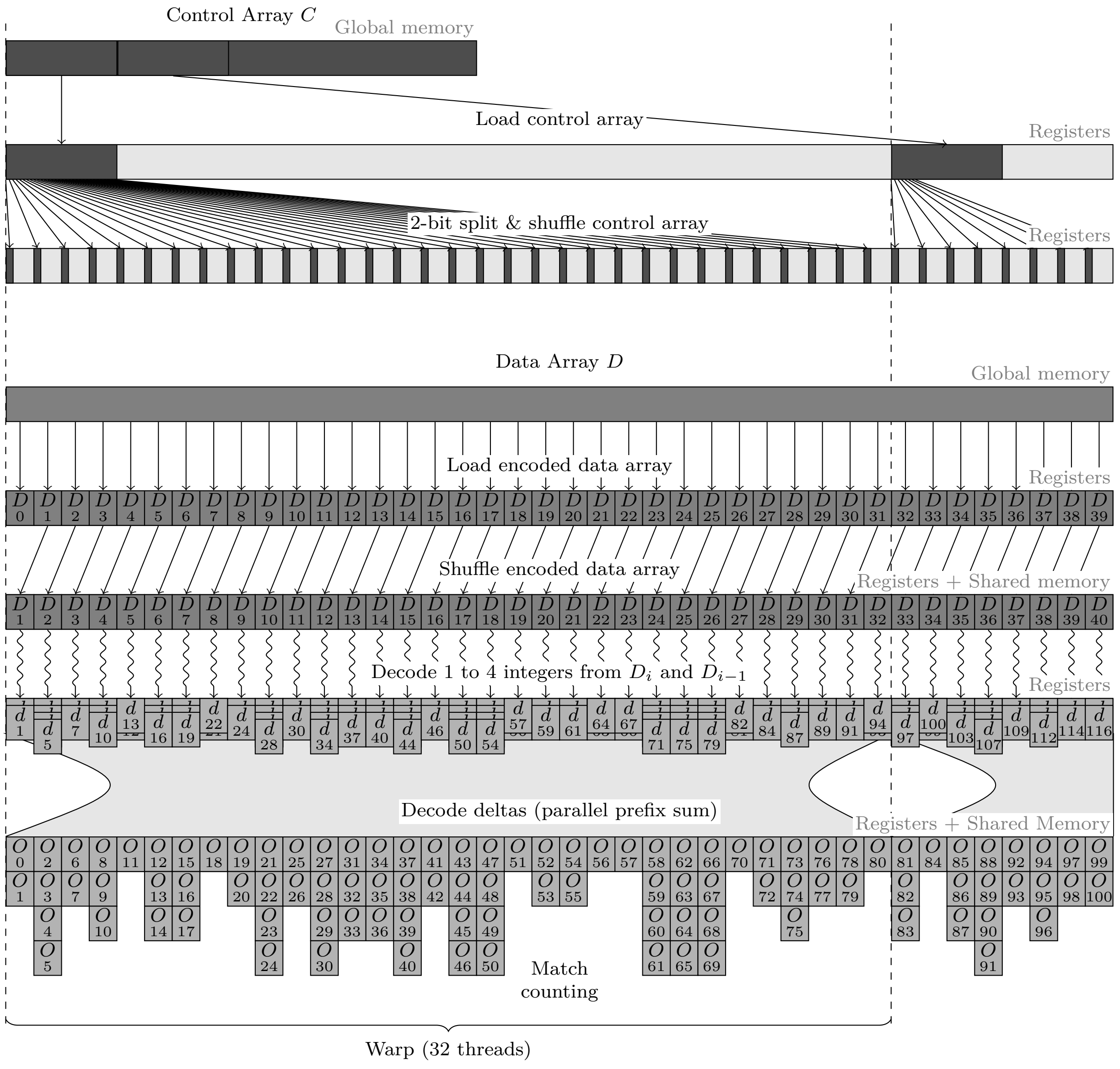
The decoding process of Bitpacking starts with the decoder reading the control array for 4 bitpacking blocks per warp, that is 128 integers per warp. We try to maintain the same parallelism for match counting, i.e. for one thread to do counting of at most 4 integers. Because we only use Bitpacking for full blocks of 32 integers in GENIE, each thread will decode *exactly* 4 integers.

Note that we are not using variable block length bitpacking such as in VSEncoding [187] or QMX [205] since this would complicate the fully data parallel model for decoding and impair its efficiency. Variable sized blocks often break coalesced access to global memory, and require additional data shuffle.

The parallel model we use has one thread block decode and match count single inverted list, i.e. a query component. Each thread then decodes exactly one integer. First, 32 threads are used to compute the offsets for the 32 bitpacking blocks, then these offsets are used by all 1024 threads to find a location of the encoded bits. Each warp then decodes exactly 4 bitpacking blocks.

In order to prevent random access to the encoded data array in global memory, the whole encoded array is first loaded, then shuffle instructions and shared memory are used to distribute the encoded array to the appropriate threads It is often that case that one thread has to fetch two words from this encoded data array. Similarly to our implementation of Varint, this is done via shuffle operations. The two encoded words are then used by each thread to decode exactly 4 integers. Delta decoding is the same using parallel prefix sum across the thread block.

The decoding process of Bitpacking32 is visualized in Figure 5.6 and the detailed algorithm is described in Algorithm 5.2.



Control Array C
Global memory
Load control array
Registers
2-bit split & shuffle control array
Registers
Data Array D
Global memory
Load encoded data array
Registers
Shuffle encoded data array
Registers + Shared memory
Decode 1 to 4 integers from D_i and D_{i-1}
Registers
Decode deltas (parallel prefix sum)
Registers + Shared Memory
Match counting
Warp (32 threads)

Figure 5.5: Visualization of Parallel Varint decoding, using separate control array with 2-bit representation of lengths. The coding expects 8 bit codewords, hence the range of 1 to 4 decoded integers per thread.

```
Data: Numbers of 2-bit control blocks $N$;
Offsets into control array $C'$;
Offsets into data array $D'$;
Control array $C$;
Encoded data array $D$
Result: Decoded objects $O_i$ (stored in registers), Match count

foreach block $b$ in parallel ;      // each thread block processes one inv. list
do
    $n \leftarrow N[b]$ ;                          // load number of control blocks
    $C \leftarrow C + C'[b]$ ;                     // get control array for inv. list
    $D \leftarrow D + D'[b]$ ;                     // get data array for inv. list
    foreach thread $t \in [0, n)$ in parallel do
        $c \leftarrow C[t]$ ;                      // load dense control blocks
        $c \leftarrow$ (Shuffle($c$, $t/16$) $\gg 2 * t \,\%\, 32$) & 0x03 ;   // extract encoded length from control array
        $\ell \leftarrow D[t]$ ;                   // load lower encoded data
        $h \leftarrow$ ShuffleBlk($h$, $t+1$) ;    // fetch higher encoded data
        $d_0, d_1, d_2, d_3 \leftarrow$ VarintDecode($\ell$, $h$, $c$) ;   // decode 1 to 4 integers
        $O_1, O_2, O_3, O_4 \leftarrow$ ParallelPrefixSum($d_0, d_1, d_2, d_3$) ;   // decode delta
        foreach $o$ in $[0, c)$ do
            MatchCount($b$, $o_i$);
        end
    end
end
```

Algorithm 5.1: Parallel Varint decoding implementation, using separate control array with 2-bit representation of lengths. The algorithm is fully data parallel with the encoded list length. The `Shuffle(x, t)` function fetches register `x` from a thread `t` within a warp. Similarly `ShuffleBlk(x, src)` does shuffle within the whole block partially using shared memory for shuffles across warps. `VarintDecode`(ℓ,h,c) decodes c (1 to 4) integers from low and high words. `ParallelPrefixSum`(d_0, d_1, d_2, d_3) computes parallel prefix sum across a thread block, with a variable amount of local values – up to 4.

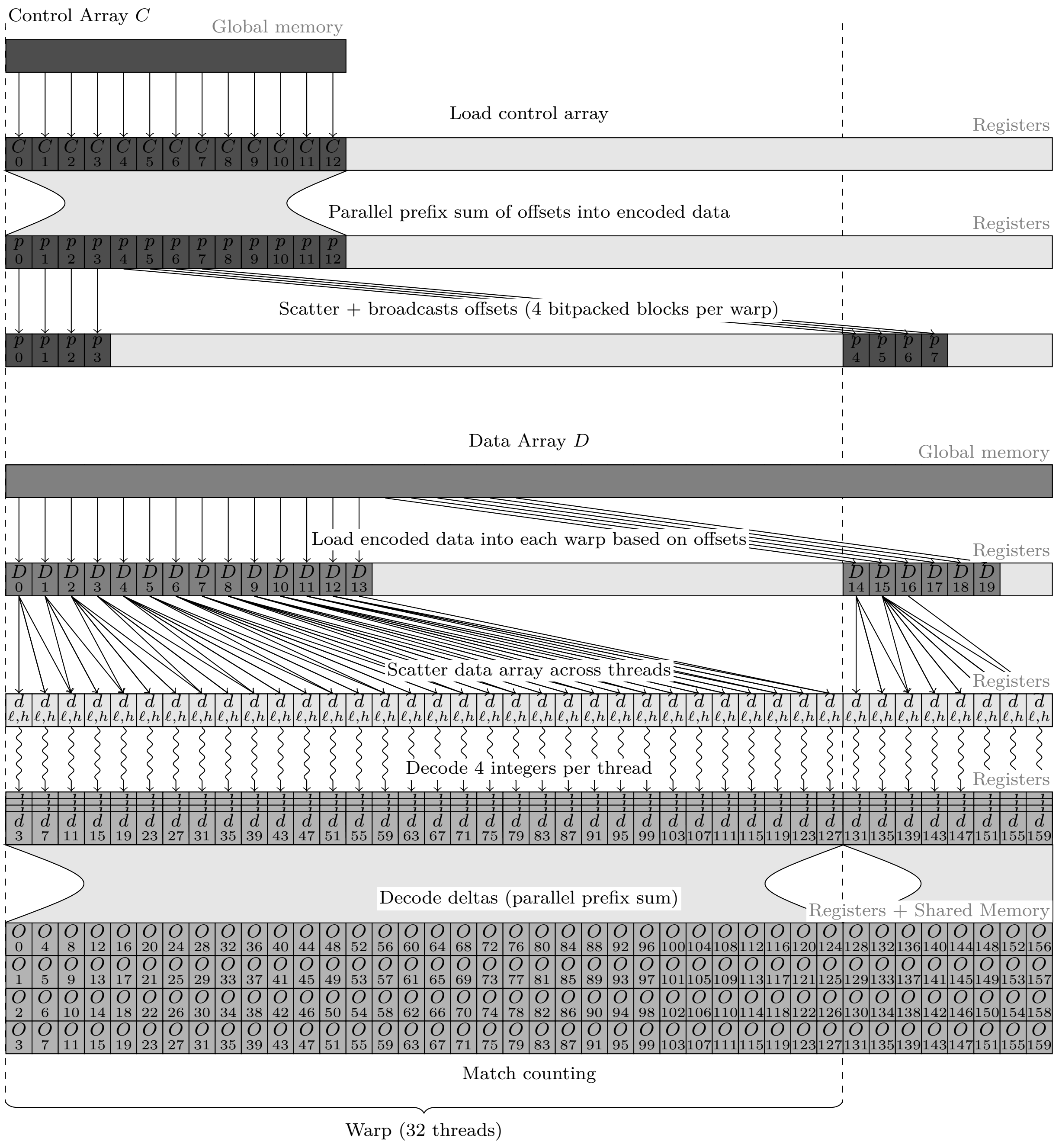

Control Array C
Global memory
Load control array
Registers
Parallel prefix sum of offsets into encoded data
Registers
Scatter + broadcasts offsets (4 bitpacked blocks per warp)
Data Array D
Global memory
Load encoded data into each warp based on offsets
Registers
Scatter data array across threads
Registers
Decode 4 integers per thread
Registers
Decode deltas (parallel prefix sum)
Registers + Shared Memory
Match counting
Warp (32 threads)

Figure 5.6: Visualization of parallel Bitpacking32 decoding, using separate control array and block of size 32. The decoding as depicted decoded 4 blocks per warp, therefore uses a block size of 256 threads with each inverted list of maximum size of 1024.

Data: Lengths array N (numbers of integers in inverted lists);
Control array C and offsets C' (each control array is of length $N/32$ bytes);
Encoded data array D;
Encoded data array offsets D' (start of each inverted list)

Result: Decoded array (stored in registers d), Match count

```
foreach block $b$ in parallel ;                         // load from global memory
do
    $n \leftarrow N[b]$ ;    // load number of bitpacking blocks in this inv. list
    $C \leftarrow C + C'[b]$ ;             // get control array for this inv. list
    $D \leftarrow D + D'[b]$ ;                // get data array for this inv. list
    foreach thread $t \in [0, n)$ in parallel do
        $c \leftarrow C[t]$ ;                                // load block sizes
        $p \leftarrow$ParallelPrefixSum($c$) ;          // cumulative block sizes
    end
    foreach thread $t \in [0, n \cdot 32)$ in parallel do
        foreach $i \in [0, 4)$ do
            $p_i \leftarrow$ ShuffleBlk ($p$,$(t/32) \cdot 4 + i$) ;       // broadcast block sizes
            $d_i \leftarrow D[4 \cdot t + i]$ ;                      // load encoded data array
            $d_{i,\ell} \leftarrow$ (ShuffleBlk($d_i$, $\frac{tb}{32}$) $\ll (tb \mod 32)$) ;   // extract low bits
            $d_{i,h} \leftarrow$ (ShuffleBlk($d_i$, $\frac{(t+1)b-1}{32}$) $\gg 32 - (tb \mod 32)$) ; // high bits
            $d_i \leftarrow (d_{i,\ell} \,||\, d_{i,h})\ \&\ (1 \gg b) - 1$ ;         // compose decoded integer
            $o_i \leftarrow$ ParallelPrefixSum($d_i$) ;                        // decode delta
            MatchCount($b$, $o_i$)
        end
    end
end
```

Algorithm 5.2: Parallel Bitpacking32 decoding implementation, using separate control array with byte representation of bit widths b used for each block. The algorithm is fully data parallel with the decoded list length. The `ShuffleBlk(x, src)` function fetches register `x` from a thread `src` within the whole block, partially using shared memory for shuffles across warps. `ParallelPrefixSum(`x`)` computes parallel prefix sum across a thread block.

An extension of Bitpacking algorithms known as *Patched Frame of Reference* (PFor) uses separate exception encoding to store all integers (deltas) with binary representation longer than b_t bits. The threshold b_t is often determined by a desired percentage of exceptions vs total numbers. A parallel GPU implementation *ParaPFor* [8] is an example implementation of such algorithm. In our experiments, PFor schemes have shown only a small increase of compression ratio, but at a cost of significant decoding time overhead.

5.4.1.4 Composite Codec

As we see in the analysis of inverted tables (see Section 5.5.3), the proportions of short lists (less than 32 docIDs) and long lists (max length of 1024 docIDs) is substantial. For short lists, a fixed size block encoding Bitpacking32 does not make sense, and likewise, for lists of maximal length Varint encoding is not as efficient. Similarly, list length is rarely a multiple of a block length of 32 numbers, therefore a significant loss of compression efficiency is due to ragged block overflow.

We use a *composite* encoding, where Bitpacking32 is used for parts of inverted lists that fit into full size bitpacking blocks, and Varint is used for smaller inverted lists and bitpacking block overflows. In practice, this is implemented by two different kernels being executed dynamically for most queries. Dynamic execution allows us to preserve data in GPU memory through the decoding and counting process. Once deltas are decoded, a parallel scan computes the raw docIDs, which are then match counted. This scan is performed as a parallel prefix sum after both codecs finish.

In some rare cases, especially for short inverted list, it is possible for an encoded inverted list to have a negative compression, i.e. is longer than the original. In this case, we simply store the inverted list in raw form.

This approach gives us the composite codec, which uses different kernels for different parts of the inverted index. The query compiler is responsible for composing the query execution correctly.

5.4.1.5 Parallel Models for Decoding

The parallel model used in GENIE for decoding and query evaluation consists of mapping one inverted list (i.e. one query component) evaluation to one thread block. The decoding and matching model remains the same, with each thread block having to decode and match their respective compressed inverted list.

All the encoding schemes have a locality property, which means that each thread is able to decode integers from the list given only small portion of the encoded data. The process of determining the location in the encoded array is codec dependent. The Varint codec is data parallel in *encoded* data size, while the Bitpacking32 and Raw codecs are data parallel in *decoded* data size.

The query execution starts with the scheduler and query compiler. The scheduler makes sure the encoded data and control array are loaded in GPU memory. It tries to optimize the

GPU resources usage before initiating the GPU execution[11]. The decoding process itself often involves several scan and shuffle operations, resulting in compute intensive part of the algorithm. Data transfer phase (encoded data, control array, queries) always precedes the computation phase, followed by data store phase (saving the top-k match results). The query scheduler tries to minimize the necessary amount of data transfer between main and GPU memory, i.e. by executing multiple queries on the same inverted lists[12]. This three-stage model of all kernels allows for efficient scheduling of these kernels such that data and compute phases overlap.

Figure 5.7 shows a high level parallel execution diagram of one query set. The load, execute, and store phases are staggered in order to maximize resource usage of the GPU. Data transfers (mainly Host to Device) constitutes the main bottleneck in parallel execution. Number of streams is not fixed and is dependent on the overall number of queries active in the system. Duration of operations in the diagram is not to scale.

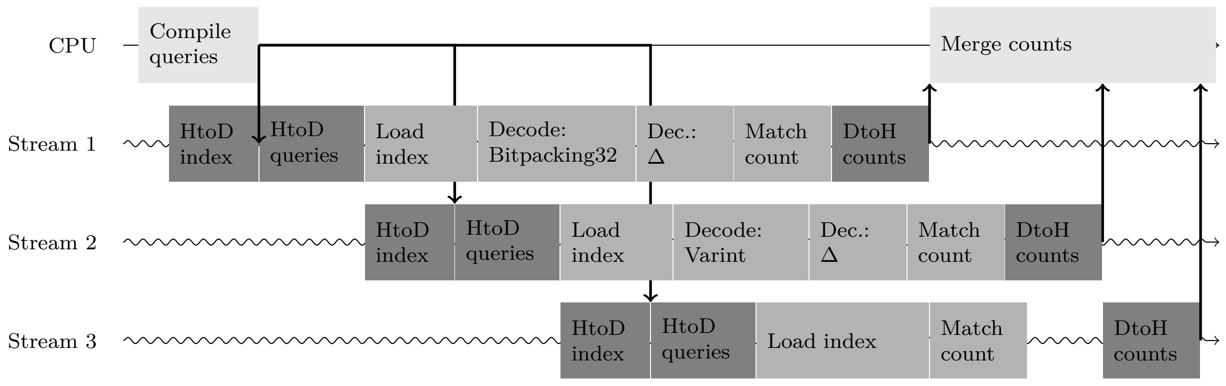


Figure 5.7: High level parallel execution model of a query set.

The query compiler is a component that retrieves the lengths and locations of all inverted lists (from the inverted lists index, see Section 5.3.1), and initial delta encoding values.

For scenarios where most of the lists are not of maximal efficient length (1024 in the current configuration), the parallel model (of processing one query component / inverted list using one thread block) will become inefficient. This is especially notable in queries with high ratio of short lists (length < 32), due to lack of warp saturation. For such cases, we can use a different parallel model called *query merging*, where multiple queries are executed by a single thread block (i.e. the thread block is packed with queries), and all the parallel operations are segmented. Each thread then submits its decoded docIDs to the counting queue corresponding to the correct query. In order to achieve this, we need

[11]The match counting part involves a priority queue and a hash table, which consist of a small amount of atomic instructions, therefore benefiting from staggering the match counting phase in thread blocks across non-concurrent time spans.

[12]If the dataset is small enough, such that the whole inverted index first in the GPU memory (with reserve space for compiled queries and counting data structures), there is no need to transfer anything but the compiled queries to the GPU memory.

to pass extra data to the kernel, namely thresholds into the control array and data array, plus an array of query identifiers for these thresholds. This feature has not been integrated into GENIE, therefore we omit detailed description of the algorithm and parallel model.

5.4.2 Further Reducing Time by Suppressing Multi-Load

In a case with a large dataset, we are forced to split the inverted index into parts that fit into GPU memory at a single time, then repeatedly load only one part of the dataset and execute all the queries. This process is called *multi-load*. When splitting a large dataset, the split is done by the CPU at an inverted lists level (meaning lists themselves are not split). The CPU maintains and subsequently merges partial top-k results.

Compressing the dataset proportionally reduces the multi-load overhead, and potentially fits the whole matching process into a single iteration. With average compressed size of 8-10 bits per docID, we can expect 3-4 times the index size threshold before multi-load is needed.

5.4.3 Efficient Inverted Lists Balancing

There are several factors that affect the performance of the decoding kernels. One of the factors is maximal occupancy of streaming multiprocessors in CUDA [148]. Higher occupancy allows the device to better hide the latency of memory accesses. The overall occupancy depends on number of threads (T) per thread block, register count (R) per thread, and shared memory (M) size per block. The occupancy can be expressed as a ratio of warps (W) allocated per streaming multiprocessor (SM) vs maximal number of warps allocatable. A simplified occupancy formula looks like this:

$$\texttt{Occupancy} = \min\left(\left\lfloor \frac{W_{perSM}}{\left\lceil \frac{\mathbf{T}}{T_{perW}} \right\rceil} \right\rfloor, \left\lfloor \frac{R_{perSM}}{\lceil \mathbf{R} \cdot T_{perW} \rceil} \right\rfloor, \left\lfloor \frac{M_{perSM}}{\mathbf{M}} \right\rfloor \left\lceil \frac{\mathbf{T}}{T_{perW}} \right\rceil \right) / \, W_{perSM}$$

Higher occupancy allows the device to better hide the latency of memory accesses and instruction executions, which in turn improves instruction level parallelism. Higher occupancy with more granular block size also allows more blocks (and therefore more query components) to be scheduled at the same time. If a thread block has shorter inverted list to process, it can partially free up compute resources to be used by another scheduled block. It is always important to profile how long a kernel takes for different block sizes, because other factors can be present, which affect the overall performance.

A common bottleneck in GPU database and indexing application is the limited integer operations throughput in the ALU. This is a hardware design decision implied by the consumer requirements[13].

[13]Modern GPU applications do not require 64-bit integer operations and always prefer floating point operations. Current compute heavy applications commonly target HPC, scientific, technical computing,

In our application, efficient implementation of parallel prefix sum is able to process up to 4 32-bit integers per thread. More than that requires increased usage of shared memory or increased number of registers, neither of which is desired due to decrease in occupancy. Efficient implementation of the decoding and matching within a single thread block thus limits the array size to 4 times the number of threads. Having much larger inverted lists (than what can be efficiently processed by a single thread block) would require scanning across thread block boundaries using global memory. This incurs significant overhead and would fracture the three-stage model of the kernels.

The cost for splitting inverted lists into smaller sub-lists is marginal, as all we have to save is the inverted list index and initial delta value for each list. This index never leaves main memory, since it is used only during query compilation. Disregarding the marginal increase in index space. Compression efficiency of the inverted lists is not affected by splitting the lists at block boundaries for Bitpacking32 encoding, nor by splitting the lists anywhere for Varint encoding.

Via theoretical evaluation and profiling, considering all the aforementioned aspects, we have settled on thread block size of 1024 threads for Varint, where each thread processes 1 to 4 integers, for a maximal inverted list length of 1024 docIDs. In the case of Bitpacking32, we use thread block size of 256, with each thread processing exactly 4 integers.

Adjacent Lists Merging in Range Queries In some cases, where the range of values of the indexed dimensions is too large, therefore the inverted lists are short, most of the operations during data loading and decoding are not utilizing resources efficiently. For example a short inverted list of length 5 still uses a full thread-block, with one full warp being active (where a single warp has the capacity to efficiently process 128 docIDs). Since GENIE supports range queries, which take several consecutive lists, it is be beneficial to process these lists in a single thread block, to a maximum of 1024. Some logic needs to be changed when executing merged lists. Specifically, the query compiler has to use a merging scheme (greedy works fine), and pass multiple initial deltas and offsets to the decoding function. During delta decoding, a segmented parallel prefix sum is then performed instead of normal parallel prefix sum. Note that with lists merging, decompression speed within decoding kernel does not change significantly. On the other hand, potentially lower amount of required thread blocks improves overall performance. Lists merging does not affect the inverted index and lists size in any way, it is an optimization of the query execution process.

5.4.4 Integration Into GENIE

We now briefly describe the integration process of query scheduler, query compiler, lists balancing, and compression aware matching into GENIE.

machine learning, and deep learning all depend on 32-bit floating point operations. It is possible to improve performance of integer heavy compute in some cases if the whole computation can be moved to the 23-bit mantissa of floating point types. The throughput of integer units can be as low as 1/6 of the floating point units. For more details, see the relevant section Maximize Instruction Throughput in CUDA docs: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#arithmetic-instructions`.

Large majority of GENIE code has been refactored to generalize all the major components, and allow for easy extensions by adding more query schedulers, compilers and encoding and matching kernels. Inverted tables and compressed inverted tables have been formalized in terms of binary format and interface, including functionality for serialization onto disk. Compilation of the inverted index compression functionality is available with the CMake option `GENIE_COMPR:BOOL`, which exposes configurable query compiler and inverted index management components.

A configuration interface is provided for the end users and as a part of the library. Based on the data and query configuration, a query evaluation models is dynamically selected by the query scheduler. For example, loading dataset encoded with composite codec, then running a single value query (not range), the query scheduler will select the appropriate simple point query compiler and composite matching kernels to execute the query.

Multiple matching kernels are available, including composite combinations of Bitpacking32, Varint and Raw encoding. Matching kernels are integrated with list decoding and delta decoding. This makes the matching process execute only one integrated kernel on the GPU, which at the end returns the match counts. The main reason for the integrated kernel is to save storing the decoded integers to global memory.

Large effort has been made on distributed version of GENIE, which allows to run the system on multiple GPU and multiple machines. A centralized scheduler has to distribute query sets to the relevant nodes and GPU, while minimizing the total query execution time. Common techniques for distributing the inverted tables includes dimension slicing or equi-size sharding. Compilation of distributed functionality is available with the CMake option `GENIE_DISTRIBUTED:BOOL`, which exposes additional interface and a runtime built on top of OpenMPI.

Lastly, an effort has also been made on clustering of the inverted tables, in order to balance the load in distributed deployment, and to potentially prune the search space without sacrificing the result accuracy. Any dataset clustering has to be done externally as part of the preprocessing step.

The GENIE code base has been refactored and upgraded to modern C++ standards and modern CUDA practices. The functionality has additionally been exposed via templated static and shared libraries, and robust CLI.

5.5 Experimental Evaluation

We have previously compared GENIE with other available nearest neighbor search tools, namely *GPU-LSH* [153], *C2LSH* [63], GPU k-selection [1]. However, none of the other tools compare to GENIE in terms of a complete *k-ANN* functionality. The comparison is available in the technical report [A.5].

For our experimental evaluation, we focus on the aspects of integer list decoding and query processing in GENIE. For comparison to the previous state, we use a configuration of GENIE that runs a *Copy* codec, which is a no op version of integrated decoding and matching kernel [14].

We start by providing an insight into the properties of inverted index for real world datasets. This is a crucial component of determining the configuration of inverted tables and query processing.

Our evaluation consists of general analysis of codec performances in terms of compression ratio and decoding speed, including the complete query evaluation process. Encoding speed is not measured, since we choose to not focus on encoding on GPU, mainly because GPU resources are fully utilized in the querying process in GENIE, rather than in data preparation. The inverted index construction is considered to be offline.

Furthermore, we evaluate GENIE performance with various codecs on real world datasets, whilst providing a white-box insight into individual components of the query process.

5.5.1 Description of Datasets

We use the same datasets that were used in the original GENIE work. These are real words datasets, each with preprocessing for a different similarity measure. For details and references to the LSH functions used, please see Section 5.3.3.

Adult[15] dataset represents relational data, where the dataset has a total of 1 million rows with 14 attributes. Numerical attributes are discretized into 1024 intervals before being indexed in GENIE. Total size of the dataset is 5.8 GB. Note that the discretization of the attributes is independent of GENIE's range queries.

OCR (Optical character recognition) is a high dimensional dataset with 1156 dimensions preprocessed using RBS for dimension reduction and Murmurhash3 for re-hashing into integer interval [0,8192).

SIFT[16] (Scale-invariant feature transform) dataset contains 4.5 million 128-dimensional features. The dataset is hashed using E^2LSH implementation[17] of p-stable distribution LSH into 67 buckets.

Tweets is a collection of 6.8 million tweets extracted[18] from Twitter over a three month period containing a set of keywords. The total size is 0.46 GB. The dataset is processed by removing stop words and indexed in a classical fashion where words are the keys and documents are docIDs in the inverted lists.

[14]Several optimizations have been done to GENIE that make the baseline querying process significantly faster, therefore we choose to compare against no op codec rather than previous version of GENIE.

[15]`http://archive.ics.uci.edu/ml/datasets/Adult`

[16]`http://corpus-texmex.irisa.fr/`

[17]`https://www.mit.edu/~andoni/LSH/manual.pdf`

[18]`https://developer.twitter.com/en/docs`

Dataset	Dimensions	Average list length	Inverted lists	Index size [MiB]	Compressed index size [MiB]
Adult	14	1019.00	644190	417.35	73.58
OCR	1156	432.08	11230620	3085.18	810.29
SIFT	128	1001.24	3451704	2197.26	596.94
Tweets	17	19.56	28225278	351.08	155.09

Table 5.3: Summary of inverted index for each dataset. Index was compressed using `gn-bp32-var-d1` composite encoding, and lists were partitioned to max length of 1024.

5.5.2 Environment And Settings

The experiments were conducted on machines with NVIDIA TITAN X[19] (12 GB GDDR5, 3072 CUDA cores 1 GHz) via PCI Express 3.0. The CPU used is Intel Core i7-3820 with 64 GB RAM. OS used is CentOS 6.5 server.

GENIE was compiled with CUDA Toolkit 9 for GPU architecture 6.1, Thrust library was included in the CUDA toolkit, Boost 1.63, and C++11 with all optimization configuration as provided by default in the CMake build system. Distributed functionality was disabled for the purpose of these experiments.

5.5.3 Analysis of Inverted Tables

In order to be able to establish efficient compression and matching models for the GPU, we first need to analyze the real world and synthetic datasets in order to configure the preprocessing step. The following is the set of attributes we try to tune when loading for datasets into GENIE:

- Preprocessed datasets size and dimension – more dimensions require scanning more inverted lists; larger size may saturate index memory.
- Distribution of lengths of inverted lists – longer lists allow for better partitioning and more efficient parallel execution.
- Distribution of delta values, i.e differences between subsequent values[20] in inverted lists – smaller deltas (or locally smaller deltas) allow more efficient inverted list compression.

The statistical data about datasets is then used with the expected compression ratio of codecs to select the appropriate coding scheme. Note that this preprocessing step is not automated. We use our extensions of GENIE for performance evaluation, which allows us to generate the statistical analysis before any data ingestion takes place. A summary of inverted index properties for all the datasets is listed in Table 5.3.

Figure 5.8 shows the distribution of inverted lists for the datasets used in our experiments (see Section 4.5). We configure GENIE such that majority of the inverted lists has

[19]`https://www.nvidia.com/en-us/geforce/graphics-cards/geforce-gtx-titan-x/specifications/`

[20]The delta values are not necessarily computed on adjacent list items, as parallel implementations of codecs often try to avoid fully serial dependencies. This is known as vectorized deltas.

been split into smaller lists of (maximal) length 1024. This is also known as partitioning *by cardinality*. For some of the datasets, there are many very short lists of length shorter than 32.

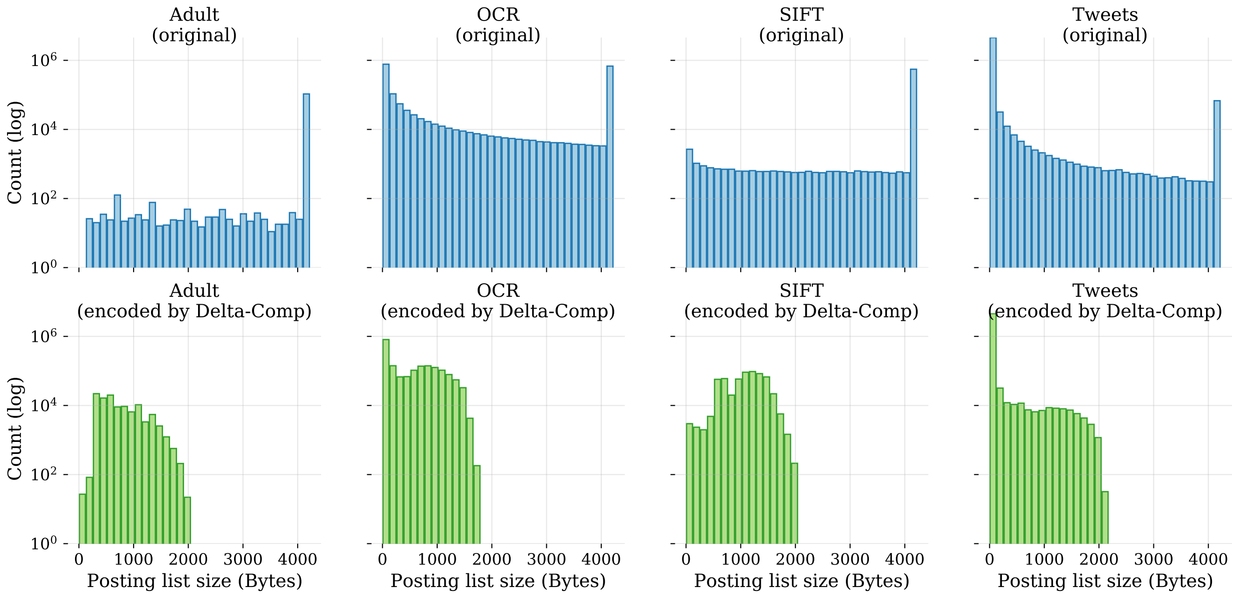


Figure 5.8: Distribution of inverted list lengths across the datasets. All datasets were loaded with a maximum inverted list length of 1024. Top row shows raw inverted list histogram. Bottom row shows histograms of encoded list length using the Composite codec.

5.5.4 Analysis of Individual Compression Schemes

To establish a baseline codecs performance, we used the *SIMDCompressionAndIntersection* compression library from [111] on our datasets. Table 5.4 shows the compression ratio (in number of bits per integer), encoding and decoding speeds. The measurements were done on per-list basis, then the results are summed over the whole dataset. All the reference algorithms involved are either CPU or SIMD implementations (denoted with *s4*), some involving sequential delta (denoted as *d1*) or vectorized delta (denoted as *d2*, *dm*, or *d4*).

Codecs starting with *gn-* are our GPU implementations according to the description in Section 5.4.1. Note that no encoder is available on the GPU, and we instead use relevant SIMD implementations during the preprocessing step.

		sift			tweets			adult			ocr	
Codec	*ratio*	*enc[s]*	*dec[s]*	*ratio*	*enc[s]*	*dec[s]*	*ratio*	*enc[s]*	*dec[s]*	*ratio*	*enc[s]*	*dec[s]*
copy	32.00	242.88	301.81	32.00	385.57	853.45	32.00	45.76	47.68	32.00	387.29	580.84
for	10.62	651.16	574.35	22.81	610.87	1059.36	5.89	121.98	111.31	19.97	1161.20	1084.82
s4-for	10.62	493.04	479.76	24.57	555.64	1010.80	5.89	106.76	96.30	20.09	850.94	914.38
bp32	8.21	578.48	466.96	16.82	648.76	1167.73	5.04	99.02	87.92	7.73	965.87	949.49
ibp32	8.21	549.55	417.41	16.82	1026.63	1056.61	5.04	94.94	70.21	7.73	962.38	832.09
s4-bp128-d1	8.53	377.33	266.39	17.61	1014.85	1003.26	5.21	63.17	46.83	8.19	742.35	626.13
s4-bp128-d2	8.96	380.08	268.91	17.92	1021.15	1001.59	5.69	63.78	46.89	8.59	729.28	621.70
s4-bp128-d4	9.47	296.48	271.61	18.27	1013.68	991.89	6.19	49.21	47.53	9.07	603.60	611.14
s4-bp128-dm	9.31	317.88	268.58	18.13	1007.00	990.99	6.08	53.83	47.59	8.93	638.76	614.65
fastpfor	7.73	1982.28	601.25	16.45	14242.62	14196.72	4.57	361.71	102.93	7.29	4834.00	3107.70
s4-fastpfor-d1	7.73	1804.38	436.29	16.45	14873.80	14984.82	4.57	326.46	75.71	7.29	4720.20	2966.72
s4-fastpfor-d2	8.39	1665.01	405.57	17.02	14656.57	14425.05	5.19	304.09	68.01	7.89	4343.72	2833.55
s4-fastpfor-d4	9.09	1541.85	364.40	17.58	15516.32	14950.96	5.92	282.17	60.23	8.54	4301.09	2769.21
s4-fastpfor-dm	8.78	1550.61	386.66	17.27	14388.22	14315.94	5.67	251.73	62.76	8.27	4213.22	2807.73
varint	9.87	1254.32	1173.10	14.63	823.80	1100.20	8.55	104.60	100.90	9.53	1462.58	1389.29
varintg8iu	10.32	2787.40	312.19	19.82	1295.29	963.36	9.39	407.89	48.66	10.33	3668.48	595.43
varintgb	11.04	825.52	461.15	17.96	945.76	948.13	10.31	66.47	52.99	11.09	1022.04	707.29
vbyte	9.87	1238.12	1115.18	14.63	805.71	1088.90	8.55	102.49	103.69	9.53	1461.65	1386.09
maskedvbyte	9.87	1202.23	431.31	16.96	808.23	1001.43	8.55	104.55	58.53	9.57	1423.07	759.70
streamvbyte	11.05	1295.09	264.36	20.29	998.66	951.70	10.31	157.98	48.44	11.13	1628.66	591.46
gn-copy	32.00	-	4.22	32.00	-	28.91	32.00	-	8.34	32.00	-	19.54
gn-var-d1	9.90	-	2.85	12.00	-	17.01	8.58	-	5.97	9.55	-	9.79
gn-bp32	21.43	-	11.84	58.05	-	58.38	22.20	-	16.41	21.82	-	51.58
gn-bp32-d1	8.66	-	13.79	46.66	-	72.22	5.61	-	18.97	8.74	-	64.74
gn-bp32-var-d1	8.69	-	18.42	14.13	-	86.61	5.64	-	22.30	8.40	-	82.52

Table 5.4: Codec comparison of compression ratio (bits per integer), encoding and decoding times (s). The measurements are conducted on per-inverted-list basis, not on any linearization of the datasets, therefore the time is impacted by the total number of inverted lists and the average size and compression ratio of individual lists. SIMDCompressionAndIntersection library is used for reference CPU/SIMD codecs. Codecs starting with *gn-* are our GPU implementations. Highlighted cells indicate competitive results.

5.5.5 Efficiency of Inverted List Encoding

To further gain insight into which part of the query process takes the majority of the time, we include a form of white-box benchmarking in GENIE, with insight into different phases and components of the matching process.

In our experiments, we run 20 queries, which are randomly selected and removed from the dataset, and $k = 20$. All measurements are averaged over 10 runs.

Each dataset is preprocessed to exactly one inverted table. This means that even if the query would have to read a small fraction of the index arrays, the whole indexed arrays are loaded to GPU. This is a common practice, since it allows continuous execution of queries once the dataset is loaded. Furthermore, selectively loading only parts of the dataset would require advanced logic on the CPU that is out of the scope of GENIE.

Our implementation of the integrated decoding and matching kernels is using pinned memory to achieve faster and more reliable transfer times of the inverted table to GPU

Adult

	Time (ms)						Memory (MiB)			
Codec	Preproc.	Transfer	Alloc.	Matching	*Decode*	Total	Table	Queries	Matching	Total
Raw	13.09	43.24	0.87	108.83	*21.77*	166.03	417.35	3.98	29.15	450.48
BP32	4.23	32.50	1.01	102.64	*15.58*	140.39	289.66	3.98	29.15	322.79
D1-BP32	3.52	8.37	0.86	129.88	*42.82*	142.63	73.18	3.98	29.15	106.31
D1-Varint	3.80	12.70	0.90	136.55	*49.49*	153.96	111.99	3.98	29.15	145.13
D1-Composite	3.63	8.49	0.91	145.24	*58.18*	158.27	73.59	3.98	29.15	106.72

OCR

	Time (ms)						Memory (MiB)			
Codec	Preproc.	Transfer	Alloc.	Matching	*Decode*	Total	Table	Queries	Matching	Total
Raw	3.38	330.80	1.02	17.74	*3.55*	353.17	3085.18	1.13	16.63	3102.95
BP32	2.30	223.77	1.07	15.97	*1.78*	243.33	2103.94	1.13	16.63	2121.70
D1-BP32	2.31	93.12	0.87	23.56	*9.37*	120.08	843.36	1.13	16.63	861.12
D1-Varint	2.01	94.18	0.80	25.95	*11.76*	123.16	921.51	1.13	16.63	939.27
D1-Composite	2.30	91.18	0.85	29.18	*14.99*	123.75	810.29	1.13	16.63	828.05

SIFT

	Time (ms)						Memory (MiB)			
Codec	Preproc.	Transfer	Alloc.	Matching	*Decode*	Total	Table	Queries	Matching	Total
Raw	15.96	242.43	1.01	49.23	*9.85*	308.63	2197.27	2.93	24.16	2224.36
BP32	3.51	152.18	0.84	46.03	*6.65*	202.79	1472.09	2.93	22.27	1497.30
D1-BP32	3.38	61.28	0.86	66.96	*27.58*	132.72	594.89	2.93	22.27	620.09
D1-Varint	4.31	74.00	0.89	71.51	*32.13*	150.95	680.19	2.93	22.27	705.40
D1-Composite	3.75	63.34	0.85	82.29	*42.91*	150.48	596.95	2.93	22.27	622.15

Tweets

	Time (ms)						Memory (MiB)			
Codec	Preproc.	Transfer	Alloc.	Matching	*Decode*	Total	Table	Queries	Matching	Total
Raw	0.66	36.82	1.03	10.45	*2.09*	49.16	351.09	0.51	32.48	384.08
BP32	0.47	65.62	0.88	9.59	*1.23*	76.77	636.94	0.51	32.48	669.94
D1-BP32	0.47	53.66	1.20	12.58	*4.22*	68.14	512.02	0.51	32.48	545.02
D1-Varint	0.48	14.10	0.85	13.57	*5.22*	29.23	131.73	0.51	32.48	164.73
D1-Composite	0.46	16.86	0.82	14.61	*6.26*	32.95	155.10	0.51	32.48	188.09

Table 5.5: Query time and memory usage measurements of GENIE with various integrated (encoding and matching) kernels.

memory. We have not experimented with unified memory[21], which based on the configuration of memory page size could potentially reduce the overall memory transfer to GPU.

Time Efficiency In this section we evaluate the running time of a query set in GENIE based on the codec and matching kernel used. The time measurement for each query set is split into phases to demonstrate the trade-off between data transfer time at the cost of additional compute for decoding the inverted lists. This total running time consists of several phases:

[21]`http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#um-unified-memory-programming-hd`

- *Preprocessing*: Consists mainly of query compilation – a CPU phase that provides parameters for each query; namely bounds of encoded data arrays and control arrays, and query identifiers for match counting.
- *Transfer*: Dataset and query load phase loads all required data from the main memory to the GPU global memory.
- *Allocation*: Allocation and initialization of all the matching components, i.e. zipper / threshold array, bitmap counter, and hash table for counting.
- *Matching*: Inverted list decoding and match counting phase consists of the main.

The timing results are listed in Table 5.5 and visualized in Figure 5.9. We can see that depending on the dataset, the majority of the time is spent on either data transfer to the GPU or the integrated decoding and matching kernel.

We use an optimized version of the decoding and matching kernel. This version does not allow for measuring the time phases of decoding and matching, because sending an event for measurement would require thread block synchronization, however in the optimized version of the kernel, match counting starts as soon as partial data is available for some threads. The table lists a decoding time among other measured times. This listed decoding time estimate is just an approximation to determine how much of the matching time is spend on decoding, as opposed to match counting.

Overall, the decoding process adds around 25% increase to the computing time, but the data size is in general 3 to 4 times smaller, resulting in significantly faster overall execution on most datasets. In case of the Adult dataset, we see that the matching time dominates the overall time even when no inverted list encoding is used. This fact remains the same when encoding is used. We can also observe that the Composite codec has the highest overhead compared to the other simpler codecs.

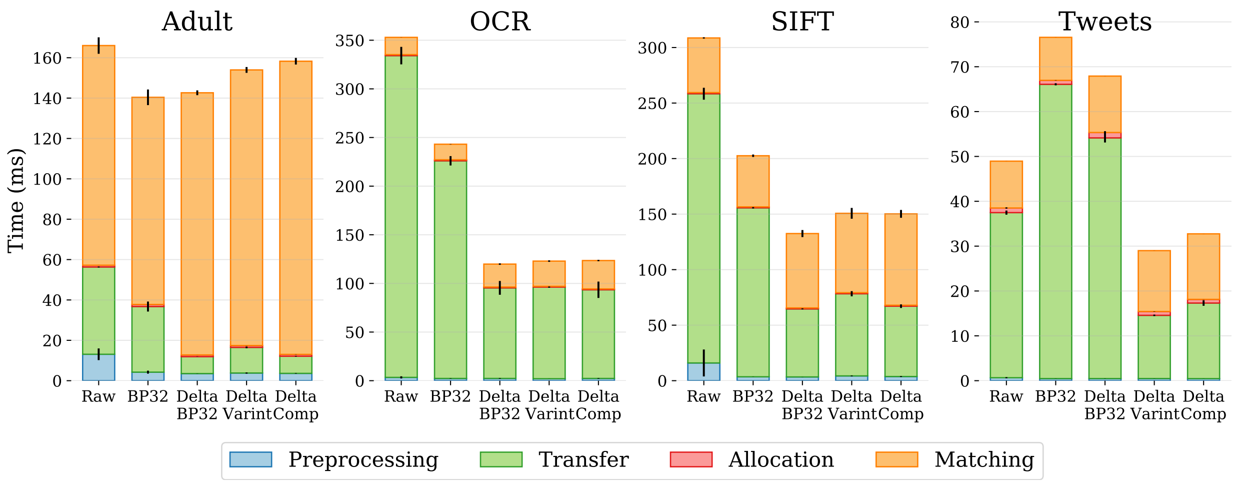


Figure 5.9: Query time analysis of GENIE with various integrated (encoding and matching) kernels.

Memory Usage In this section we examine the memory requirements of GENIE based on the codec and matching kernel used. Only memory actually allocated on the GPU is of concern for us, since GPU memory is the limiting factor. We divide the memory usage into several categories;

- *Table size*: is the size of the index – inverted table on the GPU; the table size depends on the dataset size, codec used and load balancing configuration.
- *Compiled queries size*: query set compiled for the execution on GPU; depends on the number and type of queries.
- *Matching size*: this is a total size of all the required matching components, i.e. zipper / threshold array, bitmap counter, and hash table for counting.

The memory results are listed in Table 5.5 and visualized in Figure 5.10. We see that the total memory usage is dominated by the inverted tables in all the cases.

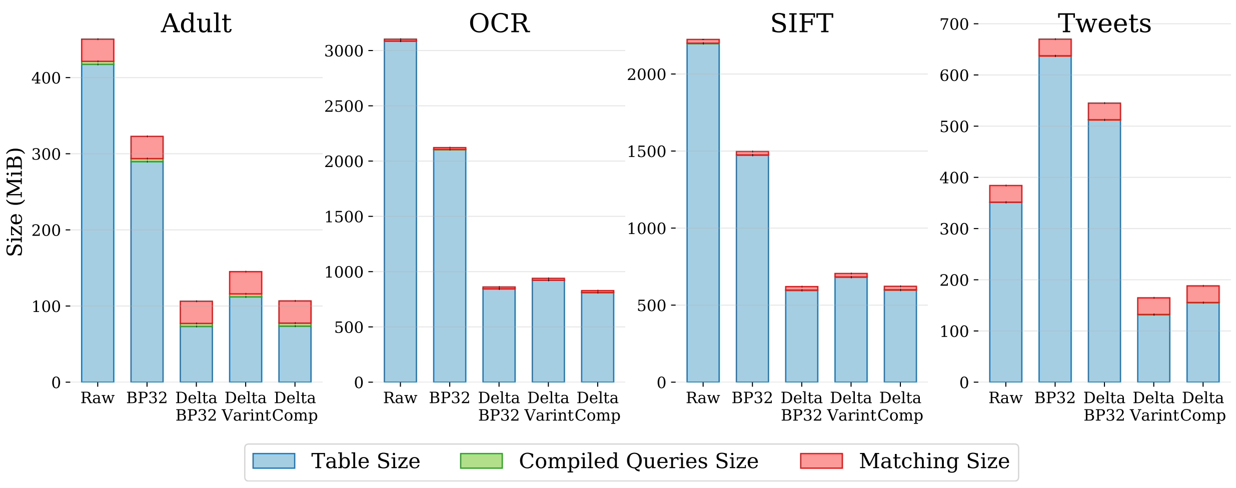


Figure 5.10: Memory usage analysis of GENIE with various integrated (encoding and matching) kernels.

Both timing and memory results indicate the efficiency of using the Composite codec for processing the inverted index. The codec causes a small overhead during data processing (up to 25% increase in compute time), but provides significantly reduced data transfer to GPU (up to 3-4 times faster transfer). Additionally, this further decreases the need of multi-loading large datasets in order to evaluate queries.

5.6 Conclusion

In this work we extended GENIE's inverted index representation and matching model with compressed inverted index. The codecs rely on data parallel implementation of delta decoding, variable byte integer decoding, bitpacking decoding, and a composite decoding, which is a dynamic combination of different codecs. The module adapts to the inverted lists to achieve good compression ratio without sacrificing GENIE's compute efficiency.

We have demonstrated that using encoded inverted index accelerates the matching process 3-4 times for majority of the real world datasets. It does so predominantly by reducing the transfer times, without significantly increasing the compute load. This further allows to execute more query sets on different datasets, since the bottleneck in inverted index matching is IO bound.

The work has been fully integrated into GENIE, with extensive configuration options for further research purposes. A distributed deployment option is available, which includes all the encoding support and additional tools for processing workloads in the distributed environment.

5.6.1 Future Work

This section lists candidates for interesting research topics related the the latest development of GENIE. Some of the listed topics have already been partially explored or investigated.

Merging Short and Similar Inverted Lists for Encoding Non-greedy merging of short lists can be used to efficiently encode a set of inverted lists in order to reduce memory footprint even further. A possible utilization of dynamic programming approach during the index construction could be done on the GPU to achieve splitting / merging of inverted lists by other measures than cardinality. While encoding short lists together, it is possible to save a set of bitmaps (or a recursive bitmap) to determine which list they come from, then masking the results during counting process. The use of bitvectors for logical operations and as parameter to intrinsic functions is very efficient on GPUs.

Similarly to encoding multiple shorts lists together, an idea has been explored to use wavelet trees for match counting directly on the compressed format, and therefore allowing for early pruning of the search space.

Clustering of Dataset Clustering of datasets before converting them to inverted index could improve the processing, such that we don't have to always load and match all of the inverted arrays. The query evaluation can instead focus on identifying parts of the inverted index that has the nearest neighbors with high probability. Additionally, clustering of the inverted tables can further help to load balance the workload in distributed deployment. This problem is partially related to reordering docIDs for inverted indexing, where clustering is one of the methods used (see Section 5.2.1).

The clustering should approach the optimal query and dataset partitioning for the purpose of maximizing the data transfer by efficiently scheduling queries, such that both PCI bus and GPU compute is saturated. This would require additional dataset and query partitioning logic as well as result merging logic.

Distributed and Multi-GPU GENIE In the current state of distributed GENIE, sharding has to be done as a data preprocessing step. Dynamic sharding of large datasets can be used in distributed GENIE. Static sharding exists in [98], but is very simple. Furthermore, we can use clustering as a basis for sharding well.

Additional work is needed on the distributed version of GENIE, in order to be able to bring it up to large scale tests, such as those on new Nvidia DGX for example.

Other It may be possible to speed up match counting using bloom filters. This reduces the accuracy and recall of the system in general, but that can be compensated with increasing the amount of LSH functions.

In the current state, each query is evaluated independently. Query compiler could be improved to be query-set aware, so it can reuse results from multiple queries - each block would then have to know how many queries it is doing count for.

Experiments with CUDA unified memory can be done to see if gradual transfer of memory pages from CPU to GPU would improve the overall performance.

A notable progress with GENIE as a database system could be made by generating the inverted index on GPU as well, including encoding. This would require improvements to the scheduler in order to accommodate GPU resources for both data preprocessing and querying.

5.6.2 Acknowledgement

This research work, including GENIE, had been partially carried out at the SeSaMe Centre, under Interactive Digital Media Institute (IDMI), National University of Singapore (NUS), Singapore. It was supported by the Singapore NRF under its Indirect Research Cost (IRC) Funding Initiative and administered by the Interactive Digital Media R&D Programme Office (IDMPO).

Chapter 6

Conclusion

In this thesis, we described our progress in several subproblems that were encountered during our research towards efficient indexing structures for similarity searching in multidimensional arrays and more generic multidimensional data similarity search on the GPU.

First, we proposed, designed and implemented a multidimensional array inverted index based on grid transformation. The index allows for efficient execution of various spatiotemporal selection queries. Subsequently, we demonstrated the efficiency of our multidimensional array index on large-scale satellite data (QuickSCAT). Given a trajectory query into a satellite sensor dataset, we perform accurate data retrieval of relevant regions. The work was implemented (including visualizations) and integrated as an extension of an open-source distributed multidimensional array database SciDB.

Next, we proposed and implemented a hierarchical multidimensional array indexing scheme *ArrayBit* that overcomes the high dimensionality-induced inefficiencies of standard spatial indexing techniques, especially on dense multidimensional arrays. ArrayBit is based on a novel n-dimensional sparse trees for dimension partitioning, with bound number of individual adaptively binned indices for attribute partitioning. This indexing performs well on range queries involving both dimensions and attributes, as it prunes the search space early, avoids reading entire index data, and does at most a single index traversal. Moreover, the indexing is easily extensible from range queries to more general membership queries.

Lastly, we extend a generic high dimensional data similarity search framework *GENIE* (Generic inverted index on GPU) by incorporating compressed inverted index, query compiler, and data parallel decoding of inverted lists on GPU. Multiple decoding schemes were implemented, and evaluated for fully data parallel decoding and query evaluation. We use heuristics for encoding selection based on the properties of the datasets being indexed, and properties of the inverted index. This data parallel decoding and query evaluation have sped up total query processing time in GENIE 3-4 times on real world datasets. All the components were integrated into the publicly available framework GENIE in a robust and modular architecture, with configurable query compiler and index management components. The extensions of GENIE were designed for multi-GPU multi-node distributed deployment with initial implementation of the distributed functionality publicly available.

6.1 Future Work

Future work related to multidimensional array indexing using ArrayBit should be focused on adapting the tree structure based on dimensions, such as adaptive mesh refinement widely used in physical simulations [24].

Another related goal is to add support for further parameters in selection queries. Apart from dimension and attribute constraints, we would like to incorporate aggregate constraint and multidimensional template pattern searching capability to our array indexing scheme.

Majority of the future work opportunities arise from our work on generic inverted index on the GPU using GENIE.

Merging of independent inverted lists can be used to efficiently encode a set of short inverted lists in order to reduce memory footprint even further and increase GPU utilization. While encoding short lists together, it is possible to save a set of bitmaps (or a recursive bitmap) to determine which list they come from, then masking the results during counting process. The use of binary vectors for logical operations and as parameter to intrinsic functions is very efficient on GPUs.

As part of the GENIE project, preliminary work has been done on clustering of datasets before converting them to inverted index. This would allow us to avoid loading and matching all of the inverted inverted index, but instead focus on parts of the inverted index that contain the nearest neighbors with high probability. Clustering of the inverted tables can further help to load balance the workload in distributed deployment.

Large opportunity in GENIE is the potential for multi-GPU multi-node distributed extension of the framework. Significant amount of work has already been done towards this goal, and the current architecture fully supports distributed deployment. Additional work needs to be done to make this extension fully functional, such as dynamic sharding of large datasets and a workload balancing query compiler and scheduler. With these components, GENIE should be able to deploy and run tests on HPC machines, such as the new Nvidia DGX.

For additional elaboration on future work on GENIE, please see Section 5.6.1.

References

[1] T. Alabi, J. D. Blanchard, B. Gordon, and R. Steinbach. Fast k-selection algorithms for graphics processing units. *J. Exp. Algorithmics*, 17(1):4.1, July 2012.

[2] A. Amir, O. Kapah, and D. Tsur. Faster two-dimensional pattern matching with rotations. *Theoretical Computer Science*, 368(3):196–204, 2006.

[3] A. Andoni, P. Indyk, T. Laarhoven, I. Razenshteyn, and L. Schmidt. Practical and optimal LSH for angular distance. Sept. 2015.

[4] V. N. Anh and A. Moffat. Inverted index compression using Word-Aligned binary codes. *Inf. Retr. Boston.*, 8(1):151–166, Jan. 2005.

[5] V. N. Anh and A. Moffat. Improved word-aligned binary compression for text indexing. *IEEE Trans. Knowl. Data Eng.*, 18(6):857–861, June 2006.

[6] V. N. Anh and A. Moffat. Index compression using 64-bit words. *Softw. Pract. Exp.*, 2010.

[7] G. Antoshenkov. Byte-aligned bitmap compression. In *Data Compression Conference, 1995. DCC'95. Proceedings*, page 476. IEEE, 1995.

[8] N. Ao, F. Zhang, D. Wu, D. S. Stones, G. Wang, X. Liu, J. Liu, and S. Lin. Efficient parallel lists intersection and index compression algorithms using graphics processing units. *Proceedings VLDB Endowment*, 4(8):470–481, May 2011.

[9] A. Appleby. Murmurhash3. `https://github.com/aappleby/smhasher/wiki/MurmurHash3`, 2016.

[10] A. S. Arefin, C. Riveros, R. Berretta, and P. Moscato. GPU-FS-kNN: a software tool for fast and scalable kNN computation using GPUs. *PLoS One*, 7(8):e44000, Aug. 2012.

[11] D. Arroyuelo, S. González, M. Oyarzún, and V. Sepulveda. Document identifier reassignment and run-length-compressed inverted indexes for improved search performance, 2013.

[12] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *Journal of the ACM (JACM)*, 45(6):891–923, 1998.

[13] M. Aumüller, E. Bernhardsson, and A. Faithfull. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Inf. Syst.*, 87:101374, Jan. 2020.

[14] A. Babenko and V. Lempitsky. The inverted Multi-Index. *IEEE Trans. Pattern Anal. Mach. Intell.*, 37(6):1247–1260, June 2015.

[15] R. Baeza-Yates and G. Navarro. New models and algorithms for multidimensional approximate pattern matching. *J. Discret. Algorithms*, 1(1):21–49, 2000.

[16] D. H. Ballard. Strip trees: a hierarchical representation for curves. *Communications of the ACM*, 24(5):310–321, 1981.

[17] R. J. Barrientos, F. Millaguir, J. L. Sánchez, and E. Arias. GPU-based exhaustive algorithms processing kNN queries. *J. Supercomput.*, 73(10):4611–4634, Oct. 2017.

[18] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann. The multidimensional database system rasdaman. In *Acm Sigmod Record*, volume 27, pages 575–577. ACM, 1998.

[19] P. Baumann, P. Furtado, R. Ritsch, and N. Widmann. Geo/environmental and medical data management in the RasDaMan system. *VLDB*, 1997.

[20] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. *The R*-tree: an efficient and robust access method for points and rectangles*, volume 19. ACM, 1990.

[21] S. B. Bell, B. Diaz, F. Holroyd, and M. Jackson. Spatially referenced methods of processing raster and vector data. *Image and vision computing*, 1(4):211–220, 1983.

[22] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[23] J. L. Bentley and H. A. Maurer. Efficient worst-case data structures for range searching. *Acta Informatica*, 13(2):155–168, 1980.

[24] M. J. Berger and P. Colella. Local adaptive mesh refinement for shock hydrodynamics. *Journal of computational Physics*, 82(1):64–84, 1989.

[25] A. Białecki, R. Muir, G. Ingersoll, and L. Imagination. Apache lucene 4. In *SIGIR 2012 workshop on open source information retrieval*, page 17, 2012.

[26] R. Blanco and Á. Barreiro. Document identifier reassignment through dimensionality reduction. In *Advances in Information Retrieval*, pages 375–387. Springer Berlin Heidelberg, 2005.

[27] D. Blandford and G. Blelloch. Index compression through document reordering. In *Proceedings DCC 2002. Data Compression Conference*, pages 342–351, Apr. 2002.

[28] P. Bogdanovich and H. Samet. The atree: a data structure to support very large scientific databases. In *Integrated Spatial Databases*, pages 235–248. Springer, 1999.

[29] A. Z. Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences 1997. Proceedings*, pages 21–29. IEEE, 1997.

[30] A. Z. Broder. On the resemblance and containment of documents. In *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171)*, pages 21–29, 1997.

[31] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. Efficient query evaluation using a two-level retrieval process. In *Proceedings of the twelfth international conference on Information and knowledge management*, CIKM '03, pages 426–434, New York, NY, USA, Nov. 2003. Association for Computing Machinery.

[32] J. B. Buck, N. Watkins, J. LeFevre, K. Ioannidou, C. Maltzahn, N. Polyzotis, and S. Brandt. Scihadoop: array-based query processing in hadoop. In *Proceedings of 2011 ICHPC*, page 66. ACM, 2011.

[33] D. Cai, X. Gu, and C. Wang. A revisit on deep hashings for large-scale content based image retrieval. Nov. 2017.

[34] K. Chakrabarti and S. Mehrotra. The hybrid tree: an index structure for high dimensional feature spaces. In *Proceedings 15th International Conference on Data Engineering (Cat. No.99CB36337)*, pages 440–447, Mar. 1999.

[35] S. Chambi, D. Lemire, O. Kaser, and R. Godin. Better bitmap performance with roaring bitmaps, 2016.

[36] C. Chan and Y. Ioannidis. An efficient bitmap encoding scheme for selection queries. *ACM SIGMOD Record*, 1999.

[37] C.-Y. Chan and Y. E. Ioannidis. Bitmap index design and evaluation. In *ACM SIGMOD Record*, volume 27, pages 355–366. ACM, 1998.

[38] M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 380–388. ACM, 2002.

[39] M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, STOC '02, pages 380–388, New York, NY, USA, May 2002. Association for Computing Machinery.

[40] Z. Chen, Y. Wen, J. Cao, W. Zheng, J. Chang, Y. Wu, G. Ma, M. Hakmaoui, and G. Peng. A survey of bitmap index compression algorithms for big data. *Tsinghua science and technology*, 20(1):100–115, 2015.

[41] C.-S. Cheng, J. J.-J. Shann, and C.-P. Chung. Unique-order interpolative coding for fast querying and space-efficient indexing in information retrieval systems. *Inf. Process. Manag.*, 42(2):407–428, Mar. 2006.

[42] J. Chou, M. Howison, B. Austin, K. Wu, J. Qiang, E. Bethel, A. Shoshani, O. Rübel, R. D. Ryne, et al. Parallel index and query for large scale data analysis. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 30. ACM, 2011.

[43] P. Christen and R. Gayler. Towards scalable real-time entity resolution using a similarity-aware inverted index approach. In *Proceedings of the 7th Australasian Data Mining Conference-Volume 87*, pages 51–60, 2008.

[44] D. Cutting and J. Pedersen. Optimization for dynamic inverted index maintenance. In *Proceedings of the 13th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '90, pages 405–411, New York, NY, USA, Dec. 1989. Association for Computing Machinery.

[45] M. Datar, 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, pages 253–262, New York, NY, USA, June 2004. Association for Computing Machinery.

[46] J. Dean. Challenges in building large-scale information retrieval systems: invited talk. In *Proceedings of the Second ACM International Conference on Web Search and Data Mining*, WSDM '09, page 1, New York, NY, USA, Feb. 2009. Association for Computing Machinery.

[47] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[48] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[49] S. Ding, J. He, H. Yan, and T. Suel. Using graphics processors for high performance IR query processing. In *Proceedings of the 18th international conference on World wide web*, WWW '09, pages 421–430, New York, NY, USA, Apr. 2009. Association for Computing Machinery.

[50] W. Dong, C. Moses, and K. Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th international conference on World wide web*, WWW '11, pages 577–586, New York, NY, USA, Mar. 2011. Association for Computing Machinery.

[51] B. Donnelly and M. Gowanlock. A coordinate-oblivious index for high-dimensional distance similarity searches on the GPU. In *Proceedings of the 34th ACM International Conference on Supercomputing*, number Article 8 in ICS '20, pages 1–12, New York, NY, USA, June 2020. Association for Computing Machinery.

[52] J. Duda. Asymmetric numeral systems: entropy coding combining speed of huffman coding with compression rate of arithmetic coding. Nov. 2013.

[53] K. Echihabi, K. Zoumpatianos, and T. Palpanas. High-Dimensional similarity search for scalable data science. 2021.

[54] S. Eghbali and L. Tahvildari. Fast cosine similarity search in binary space with angular Multi-Index hashing, 2019.

[55] A. Eldawy, M. F. Mokbel, S. Alharthi, A. Alzaidy, K. Tarek, and S. Ghani. Shahed: A mapreduce-based system for querying and visualizing spatio-temporal satellite data. In *2015 IEEE 31st International Conference on Data Engineering*, pages 1585–1596. IEEE, 2015.

[56] W. Fang, B. He, and Q. Luo. Database compression on graphics processors. *Proceedings VLDB Endowment*, 3(1-2):670–680, Sept. 2010.

[57] G. Fekete. Rendering and managing spherical data with sphere quadtrees. In *Proceedings of the 1st conference on Visualization'90*, pages 176–186. IEEE Computer Society Press, 1990.

[58] R. A. Finkel and J. L. Bentley. Quad trees a data structure for retrieval on composite keys. *Acta informatica*, 4(1):1–9, 1974.

[59] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3(3):209–226, Sept. 1977.

[60] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys (CSUR)*, 30(2):170–231, 1998.

[61] R. Gaioso, V. Gil-Costa, H. Guardia, and H. Senger. Performance evaluation of single vs. batch of queries on GPUs, 2020.

[62] R. Gaioso, H. C. Guardia, V. Gil-Costa, and H. Senger. Parallel strategies for the execution of top-k queries with MaxScore on GPUs. In *2019 31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 104–111. ieeexplore.ieee.org, Oct. 2019.

[63] J. Gan, 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, pages 541–552, New York, NY, USA, May 2012. Association for Computing Machinery.

[64] V. Garcia, É. Debreuve, F. Nielsen, and M. Barlaud. K-nearest neighbor search: Fast GPU-based implementations and application to high-dimensional feature matching. In *2010 IEEE International Conference on Image Processing*, pages 3757–3760, Sept. 2010.

[65] T. Ge, K. He, Q. Ke, and J. Sun. Optimized product quantization for approximate nearest neighbor search, 2013.

[66] F. Gieseke, J. Heinermann, C. Oancea, and C. Igel. Buffer k-d trees: Processing massive nearest neighbor queries on GPUs. In E. P. Xing and T. Jebara, editors, *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pages 172–180, Bejing, China, 2014. PMLR.

[67] C. Gormley and Z. Tong. *Elasticsearch: the definitive guide: a distributed real-time search and analytics engine.* " O'Reilly Media, Inc.", 2015.

[68] L. Gosink, J. Shalf, K. Stockinger, K. Wu, and W. Bethel. Hdf5-fastquery: Accelerating complex queries on hdf datasets using fast bitmap indices. In *Scientific and Statistical Database Management, 2006. 18th International Conference on*, pages 149–158. IEEE, 2006.

[69] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUTeraSort: high performance graphics co-processor sorting for large database management. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pages 325–336, New York, NY, USA, June 2006. Association for Computing Machinery.

[70] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast computation of database operations using graphics processors. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, pages 206–es, New York, NY, USA, July 2005. Association for Computing Machinery.

[71] M. Gowanlock and B. Karsin. GPU-Accelerated similarity Self-Join for Multi-Dimensional data. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*, number Article 6 in DaMoN'19, pages 1–9, New York, NY, USA, July 2019. Association for Computing Machinery.

[72] Gunther, Gunther, Milne, and Narasimhan. Assessing document relevance with runtime reconfigurable machines, 1996.

[73] O. Günther and E. Wong. The arc tree: an approximation scheme to represent arbitrary curved shapes. *Computer Vision, Graphics, and Image Processing*, 51(3):313–337, 1990.

[74] P. D. Gutiérrez, M. Lastra, J. Bacardit, J. M. Benítez, and F. Herrera. GPU-SME-kNN: Scalable and memory efficient kNN and lazy learning using GPUs. *Inf. Sci.*, 373:165–182, Dec. 2016.

[75] H. Güting and H.-P. Kriegel. Multidimensional b-tree: An efficient dynamic file structure for exact match queries. In *GI-10. Jahrestagung*, pages 375–388. Springer, 1980.

[76] R. H. Güting and M. Schneider. *Moving objects databases.* Elsevier, 2005.

[77] A. Guttman. *R-trees: a dynamic index structure for spatial searching*, volume 14. ACM, 1984.

[78] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational query coprocessing on graphics processors. *ACM Trans. Database Syst.*, 34(4):1–39, Dec. 2009.

[79] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 511–524, New York, NY, USA, June 2008. Association for Computing Machinery.

[80] B. He and J. X. Yu. High-Throughput transaction executions on graphics processors. Mar. 2011.

[81] J. Heo, J. Won, Y. Lee, S. Bharuka, J. Jang, T. J. Ham, and J. W. Lee. IIU: Specialized architecture for inverted index search. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pages 1233–1245, New York, NY, USA, Mar. 2020. Association for Computing Machinery.

[82] D. Hilbert. Ueber die stetige abbildung einer line auf ein flächenstück. *Mathematische Annalen*, 38(3):459–460, 1891.

[83] S.-S. Ho and A. Talukder. Automated cyclone discovery and tracking using knowledge sharing in multiple heterogeneous satellite data. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 928–936. ACM, 2008.

[84] S.-S. Ho, W. Tang, W. T. Liu, and M. Schneider. A framework for moving sensor data query and retrieval of dynamic atmospheric events. In *Scientific and Statistical Database Management*, pages 96–113. Springer, 2010.

[85] R. N. Hoffman and S. M. Leidner. An introduction to the near-real-time quikscat data. *Weather and Forecasting*, 20(4):476–493, 2005.

[86] Q. Huang, J. Feng, Y. Zhang, Q. Fang, and W. Ng. Query-aware locality-sensitive hashing for approximate nearest neighbor search. *Proceedings VLDB Endowment*, 9(1):1–12, Sept. 2015.

[87] P. Huijse, P. A. Estevez, P. Protopapas, J. C. Principe, and P. Zegers. Computational intelligence challenges and applications on large-scale astronomical time series databases. *IEEE Computational Intelligence Magazine*, 9(3):27–39, 2014.

[88] G. M. Hunter and K. Steiglitz. Operations on images using quad trees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, (2):145–153, 1979.

[89] V. Hyvonen, T. Pitkanen, S. Tasoulis, E. Jaasaari, R. Tuomainen, L. Wang, J. Corander, and T. Roos. Fast nearest neighbor search through sparse random projections and voting, 2016.

[90] S. Idreos, F. Groffen, N. Nes, S. Manegold, S. Mullender, M. Kersten, et al. Monetdb: Two decades of research in column-oriented database architectures. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 35(1):40–45, 2012.

[91] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613. ACM, 1998.

[92] M. Iwasaki. Pruned bi-directed k-nearest neighbor graph for proximity search. In *Similarity Search and Applications*, pages 20–33. Springer International Publishing, 2016.

[93] M. Iwasaki and D. Miyazaki. Optimization of indexing based on k-nearest neighbor graph for proximity search in high-dimensional data. Oct. 2018.

[94] H. V. Jagadish. Spatial search with polyhedra. In *Data Engineering, 1990. Proceedings. Sixth International Conference on*, pages 311–319. IEEE, 1990.

[95] H. V. Jagadish, N. Koudas, S. Muthukrishnan, V. Poosala, K. C. Sevcik, and T. Suel. Optimal histograms with quality guarantees. In *VLDB*, volume 98, pages 24–27, 1998.

[96] N. Jahan Lisa, T. D. A. Nguyen, D. Habich, A. Kumar, and W. Lehner. High-Throughput BitPacking compression. In *2019 22nd Euromicro Conference on Digital System Design (DSD)*, pages 643–646, Aug. 2019.

[97] H. Jégou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *IEEE Trans. Pattern Anal. Mach. Intell.*, 33(1):117–128, Jan. 2011.

[98] J. Johnson, M. Douze, and H. Jégou. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, pages 1–1, 2019.

[99] D. V. Kalashnikov. Super-EGO: fast multi-dimensional similarity join. *VLDB J.*, 22(4):561–585, Aug. 2013.

[100] A. Kalinin, U. Cetintemel, and S. Zdonik. Searchlight: enabling integrated search and exploration over large multidimensional data. *Proc. of the VLDB Endowment*, 8(10):1094–1105, 2015.

[101] N. Katayama and S. Satoh. The sr-tree: An index structure for high-dimensional nearest neighbor queries. In *ACM SIGMOD Record*, volume 26, pages 369–380. ACM, 1997.

[102] K. Kato and T. Hosino. Multi-GPU algorithm for k-nearest neighbor problem. *Concurr. Comput.*, 24(1):45–53, Jan. 2012.

[103] S. Kim, J. Lee, S. R. Satti, and B. Moon. Sbh: Super byte-aligned hybrid bitmap compression. *Information Systems*, 2016.

[104] S. Kim, S. G. Sohn, T. Kim, J. Yu, B. Kim, and B. Moon. Selective scan for filter operator of scidb. In *Proceedings of the 28th International Conference on Scientific and Statistical Database Management*, SSDBM '16, pages 28:1–28:4, New York, NY, USA, 2016. ACM.

[105] I. Komarov, A. Dashti, and R. M. D'Souza. Fast k-NNG construction with GPU-based quick multi-select. *PLoS One*, 9(5):e92409, May 2014.

[106] Q. Kuang and L. Zhao. A practical GPU based kNN algorithm. In *Proceedings. The 2009 International Symposium on Computer Science and Computational Technology (ISCSCI 2009)*, page 151, 2009.

[107] D.-T. Lee. Medial axis transformation of a planar shape. *IEEE Transactions on pattern analysis and machine intelligence*, (4):363–369, 1982.

[108] Y. Lei, Q. Huang, M. Kankanhalli, and A. Tung. Sublinear time nearest neighbor search over generalized weighted space. In K. Chaudhuri and R. Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 3773–3781. PMLR, 2019.

[109] Y. Lei, Q. Huang, M. Kankanhalli, and A. K. H. Tung. Locality-Sensitive hashing scheme based on longest circular Co-Substring. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, pages 2589–2599, New York, NY, USA, June 2020. Association for Computing Machinery.

[110] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Softw. Pract. Exp.*, 45(1):1–29, Jan. 2015.

[111] D. Lemire, L. Boytsov, and N. Kurz. SIMD compression and the intersection of sorted integers. *Softw. Pract. Exp.*, 46(6):723–749, June 2016.

[112] D. Lemire, O. Kaser, N. Kurz, L. Deri, C. O'Hara, F. Saint-Jacques, and G. Ssi-Yan-Kai. Roaring bitmaps: Implementation of an optimized software library. *Softw. Pract. Exp.*, 48(4):867–895, Apr. 2018.

[113] D. Lemire, N. Kurz, and C. Rupp. Stream VByte: Faster byte-oriented integer compression. *Inf. Process. Lett.*, 130:1–6, Feb. 2018.

[114] B. Lessley and H. Childs. Data-Parallel hashing techniques for GPU architectures. *IEEE Trans. Parallel Distrib. Syst.*, 31(1):237–250, Jan. 2020.

[115] C. Li, M. Zhang, D. G. Andersen, and Y. He. Improving approximate nearest neighbor search through learned adaptive early termination, 2020.

[116] S. Li and N. Amenta. Brute-Force k-nearest neighbors search on the GPU. In *Similarity Search and Applications*, pages 259–270. Springer International Publishing, 2015.

[117] W. Li, Y. Zhang, Y. Sun, W. Wang, M. Li, W. Zhang, and X. Lin. Approximate nearest neighbor search on high dimensional data — experiments, analyses, and improvement. *IEEE Trans. Knowl. Data Eng.*, 32(8):1475–1488, Aug. 2020.

[118] Y. Li and J. M. Patel. BitWeaving: fast scans for main memory data processing. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 289–300, New York, NY, USA, June 2013. Association for Computing Machinery.

[119] Z. Li, F. Hu, J. L. Schnase, D. Q. Duffy, T. Lee, M. K. Bowen, and C. Yang. A spatiotemporal indexing approach for efficient processing of big array-based climate data with mapreduce. *International Journal of Geographical Information Science*, pages 1–19, 2016.

[120] S. Liang, C. Wang, Y. Liu, and L. Jian. CUKNN: A parallel implementation of k-nearest neighbor on CUDA-enabled GPU. In *2009 IEEE Youth Conference on Information, Computing and Telecommunication*, pages 415–418, Sept. 2009.

[121] M. D. Lieberman, J. Sankaranarayanan, and H. Samet. A fast similarity join algorithm using graphics processing units. In *2008 IEEE 24th International Conference on Data Engineering*, pages 1111–1120, Apr. 2008.

[122] X. Liu and G. F. Schrack. A new ordering strategy applied to spatial data processing. *International Journal of Geographical Information Science*, 12(1):3–22, 1998.

[123] Y. Liu, J. Wang, and S. Swanson. Griffin: uniting CPU and GPU in information retrieval systems for intra-query parallelism. *SIGPLAN Not.*, 53(1):327–337, Feb. 2018.

[124] T. L. Lopes Siqueira, R. R. Ciferri, V. C. Times, and C. D. de Aguiar Ciferri. A spatial bitmap-based index for geographical data warehouses. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1336–1342. ACM, 2009.

[125] W. Lu, Y. Shen, S. Chen, and B. C. Ooi. Efficient processing of k nearest neighbor joins using MapReduce. June 2012.

[126] N. Lukač and B. Žalik. Fast approximate k-nearest neighbours search using GPGPU. In Y. Cai and S. See, editors, *GPU Computing and Applications*, pages 221–234. Springer Singapore, Singapore, 2015.

[127] L. Luo, M. D. F. Wong, and L. Leong. Parallel implementation of r-trees on the GPU. In *17th Asia and South Pacific Design Automation Conference*, pages 353–358, Jan. 2012.

[128] Q. Lv, 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, pages 950–961. VLDB Endowment, Sept. 2007.

[129] Y. Malkov, A. Ponomarenko, A. Logvinov, and V. Krylov. Approximate nearest neighbor algorithm based on navigable small world graphs. *Inf. Syst.*, 45:61–68, Sept. 2014.

[130] Y. A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs, 2020.

[131] A. Mallia, M. Siedlaczek, T. Suel, and M. Zahran. GPU-Accelerated decoding of integer lists. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, CIKM '19, pages 2193–2196, New York, NY, USA, Nov. 2019. Association for Computing Machinery.

[132] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval.* Cambridge University Press, New York, NY, USA, 2008.

[133] J. Masek, R. Burget, J. Karasek, V. Uher, and M. K. Dutta. Multi-GPU implementation of k-nearest neighbor algorithm. In *2015 38th International Conference on Telecommunications and Signal Processing (TSP)*, pages 764–767, July 2015.

[134] T. Matsumoto and M. L. Yiu. Accelerating exact similarity search on CPU-GPU systems. In *2015 IEEE International Conference on Data Mining*, pages 320–329, Nov. 2015.

[135] T. Matsuyama, M. Nagao, et al. A file organization for geographic information systems based on spatial proximity. *Computer Vision, Graphics, and Image Processing*, 26(3):303–318, 1984.

[136] A. Moffat and M. Petri. ANS-Based index compression. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, pages 677–686. Association for Computing Machinery, New York, NY, USA, Nov. 2017.

[137] A. Moffat and M. Petri. Index compression using Byte-Aligned ANS coding and Two-Dimensional contexts. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*, WSDM '18, pages 405–413, New York, NY, USA, Feb. 2018. Association for Computing Machinery.

[138] A. Moffat and L. Stuiver. Binary interpolative coding for effective index compression. *Inf. Retr. Boston.*, 3(1):25–47, July 2000.

[139] M. Mohsen, N. May, C. Färber, and D. Broneske. FPGA-Accelerated compression of integer vectors. In *Proceedings of the 16th International Workshop on Data Management on New Hardware*, number Article 9 in DaMoN '20, pages 1–10, New York, NY, USA, June 2020. Association for Computing Machinery.

[140] M. F. Mokbel, T. M. Ghanem, and W. G. Aref. Spatio-temporal access methods. *IEEE Data Eng. Bull.*, 26(2):40–49, 2003.

[141] M. Muja and D. G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *In VISAPP International Conference on Computer Vision Theory and Applications*, pages 331–340, 2009.

[142] M. Muja and D. G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. *VISAPP (1)*, 2(331-340):2, 2009.

[143] P. Nagarkar, K. Candan, and A. Bhat. Compressed spatial hierarchical bitmap (cSHB) indexes for efficiently processing spatial range query workloads. *Proceedings of the VLDB Endowment*, 2015.

[144] R. C. Nelson and H. Samet. A consistent hierarchical representation for vector data. In *ACM SIGGRAPH Computer Graphics*, volume 20, pages 197–206. ACM, 1986.

[145] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems (TODS)*, 9(1):38–71, 1984.

[146] M. Norouzi, A. Punjani, and D. J. Fleet. Fast search in hamming space with multi-index hashing, 2012.

[147] M. Norouzi, A. Punjani, and D. J. Fleet. Fast exact search in hamming space with Multi-Index hashing, 2014.

[148] Nvidia. Cuda toolkit documentation v11.4.0. `https://docs.nvidia.com/cuda/archive/11.4.0/`, 2021.

[149] P. O'Neil and D. Quass. Improved query performance with variant indexes. In *ACM Sigmod Record*, volume 26, pages 38–49. ACM, 1997.

[150] J. A. Orenstein and T. H. Merrett. A class of data structures for associative searching. In *Proceedings of the 3rd ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 181–190. ACM, 1984.

[151] G. Ottaviano and R. Venturini. Partitioned Elias-Fano indexes. In *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval*, SIGIR '14, pages 273–282, New York, NY, USA, July 2014. Association for Computing Machinery.

[152] J. Pan, C. Lauterbach, and D. Manocha. Efficient nearest-neighbor computation for GPU-based motion planning. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2243–2248, Oct. 2010.

[153] J. Pan and D. Manocha. Fast GPU-based locality sensitive hashing for k-nearest neighbor computation. In *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '11, pages 211–220, New York, NY, USA, Nov. 2011. Association for Computing Machinery.

[154] B. Peng, P. Fatourou, and T. Palpanas. MESSI: In-Memory data series indexing. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 337–348, Apr. 2020.

[155] B. Peng, P. Fatourou, and T. Palpanas. SING: Sequence indexing using GPUs. 2021.

[156] L. Peng and Y. Diao. Supporting data uncertainty in array databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 545–560. ACM, 2015.

[157] G. E. Pibiri. On slicing sorted integer sequences. July 2019.

[158] G. E. Pibiri, M. Petri, and A. Moffat. Fast Dictionary-Based compression for inverted indexes. In *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining*, WSDM '19, pages 6–14, New York, NY, USA, Jan. 2019. Association for Computing Machinery.

[159] G. E. Pibiri and R. Venturini. Clustered Elias-Fano indexes. *ACM Trans. Inf. Syst. Secur.*, 36(1):1–33, Apr. 2017.

[160] G. E. Pibiri and R. Venturini. On optimally partitioning Variable-Byte codes, 2020.

[161] G. E. Pibiri and R. Venturini. Techniques for inverted index compression. *ACM Comput. Surv.*, 53(6):1–36, Dec. 2020.

[162] J. Plaisance, N. Kurz, and D. Lemire. Vectorized VByte decoding. Feb. 2015.

[163] Z. Qiu, Y. Pan, T. Yao, and T. Mei. Deep semantic hashing with generative adversarial networks. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '17, pages 225–234, New York, NY, USA, Aug. 2017. Association for Computing Machinery.

[164] A. Rahimi, B. Recht, and Others. Random features for Large-Scale kernel machines. In *NIPS*, volume 3, page 5, 2007.

[165] R. P. Raunikar, W. M. Forney, and S. P. Benjamin. What is the economic value of satellite imagery. *US Geological Survey Fact Sheet*, 3003, 2013.

[166] J. T. Robinson. The kdb-tree: a search structure for large multidimensional dynamic indexes. In *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*, pages 10–18. ACM, 1981.

[167] V. Roussev. Data fingerprinting with similarity digests. In *IFIP International Conference on Digital Forensics*, pages 207–226. Springer, 2010.

[168] H. Sagan. *Space-filling curves.* Springer Science & Business Media, 2012.

[169] R. Salakhutdinov and G. Hinton. Semantic hashing. *Int. J. Approx. Reason.*, 50(7):969–978, July 2009.

[170] H. Samet. A quadtree medial axis transform. *Communications of the ACM*, 26(9):680–693, 1983.

[171] H. Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys (CSUR)*, 16(2):187–260, 1984.

[172] H. Samet. Applications of spatial data structures. 1990.

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

[174] H. Samet. *Foundations of multidimensional and metric data structures.* Morgan Kaufmann, 2006.

[175] H. Samet. Algorithms and theory of computation handbook. chapter Multidimensional Data Structures for Spatial Applications, pages 6–6. Chapman & Hall/CRC, 2010.

[176] H. Samet. Sorting in space: multidimensional, spatial, and metric data structures for computer graphics applications. In *ACM SIGGRAPH ASIA 2010 Courses*, page 3. ACM, 2010.

[177] H. Samet and M. Tamminen. Efficient component labeling of images of arbitrary dimension represented by linear bintrees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 10(4):579–586, 1988.

[178] S. Samsi, L. Brattain, V. Gadepally, and J. Kepner. D4m and large array databases for management and analysis of large biomedical imaging data. *New England Database Summit*, 2016.

[179] M. Saouk, C. Doulkeridis, A. Vlachou, and K. Norvag. Efficient processing of top-k joins in MapReduce, 2016.

[180] P. Scheuermann and M. Ouksel. Multidimensional b-trees for associative searching in database systems. *Information systems*, 7(2):123–137, 1982.

[181] M. Schiwietz and H.-P. Kriegel. Query processing of spatial objects: Complexity versus redundancy. In *International Symposium on Spatial Databases*, pages 377–396. Springer, 1993.

[182] B. Schlegel, R. Gemulla, and W. Lehner. Fast integer compression using SIMD instructions. In *Proceedings of the Sixth International Workshop on Data Management on New Hardware*, DaMoN '10, pages 34–40, New York, NY, USA, June 2010. Association for Computing Machinery.

[183] D. Shahi. *Apache Solr.* Springer, 2016.

[184] B.-W. Shen, R. Atlas, O. Reale, S.-J. Lin, J.-D. Chern, J. Chang, C. Henze, and J.-L. Li. Hurricane forecasts with a global mesoscale-resolving model: Preliminary results with hurricane katrina (2005). *Geophysical Research Letters*, 33(13), 2006.

[185] W.-Y. Shieh, T.-F. Chen, J. J.-J. Shann, and C.-P. Chung. Inverted file compression through document identifier reassignment, 2003.

[186] A. Shrivastava and P. Li. In Defense of Minhash over Simhash. In S. Kaski and J. Corander, editors, *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics*, volume 33 of *Proceedings of Machine Learning Research*, pages 886–894, Reykjavik, Iceland, 2014. PMLR.

[187] F. Silvestri and R. Venturini. VSEncoding: efficient coding and fast decoding of integer lists via dynamic programming. In *Proceedings of the 19th ACM international conference on Information and knowledge management*, CIKM '10, pages 1219–1228, New York, NY, USA, Oct. 2010. Association for Computing Machinery.

[188] A. Singh, K. Deep, and P. Grover. A novel approach to accelerate calibration process of a k-nearest neighbours classifier using GPU. *J. Parallel Distrib. Comput.*, 104:114–129, June 2017.

[189] R. R. Sinha, S. Mitra, and M. Winslett. Bitmap indexes for large scientific data sets: A case study. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, pages 10–pp. IEEE, 2006.

[190] R. R. Sinha and M. Winslett. Multi-resolution bitmap indexes for scientific data. *ACM Transactions on Database Systems (TODS)*, 32(3):16, 2007.

[191] T. L. L. Siqueira, C. D. de Aguiar 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, 2012.

[192] N. Sismanis, N. Pitsianis, and X. Sun. Parallel search of k-nearest neighbors with synchronous operations. In *2012 IEEE Conference on High Performance Extreme Computing*, pages 1–6, Sept. 2012.

[193] E. Soroush, M. Balazinska, S. Krughoff, and A. Connolly. Efficient iterative processing in the SciDB parallel array engine. In *Proceedings of the 27th International Conference on Scientific and Statistical Database Management*, page 39. ACM, 2015.

[194] A. A. Stepanov, A. R. Gangolli, D. E. Rose, R. J. Ernst, and P. S. Oberoi. SIMD-based decoding of posting lists. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, CIKM '11, pages 317–326, New York, NY, USA, Oct. 2011. Association for Computing Machinery.

[195] K. Stockinger and K. Wu. Bitmap indices for data warehouses. *Data Warehouses and OLAP: Concepts, Architectures and Solutions*, page 57, 2006.

[196] M. Stonebraker, P. Brown, D. Zhang, and J. Becla. SciDB: A database management system for applications with complex analytics. *Computing in Science and Engineering*, 15(3):54–62, 2013.

[197] M. Stonebraker, J. Duggan, L. Battle, and O. Papaemmanouil. SciDB DBMS Research at MIT. pages 1–10, 2013.

[198] Y. Su, Y. Wang, and G. Agrawal. In-situ bitmaps generation and efficient data analysis based on bitmaps. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 61–72. ACM, 2015.

[199] Y. Sun, 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. *Proceedings VLDB Endowment*, 2014.

[200] A. S. Szalay, J. Gray, G. Fekete, P. Z. Kunszt, P. Kukol, and A. Thakar. Indexing the sphere with the hierarchical triangular mesh. *arXiv preprint cs/0701164*, 2007.

[201] M. Tamminen and R. Sulonen. The excell method for efficient geometric access to data. In *Proceedings of the 19th Design Automation Conference*, pages 345–351. IEEE Press, 1982.

[202] Y. Tao, 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):1–46, July 2010.

[203] G. Teodoro, E. Valle, N. Mariano, R. Torres, W. Meira, and J. H. Saltz. Approximate similarity search for online multimedia services on distributed CPU–GPU platforms. *VLDB J.*, 23(3):427–448, June 2014.

[204] Y. Theoderidis, M. Vazirgiannis, and T. Sellis. Spatio-temporal indexing for large multimedia applications. In *Multimedia Computing and Systems, 1996., Proceedings of the Third IEEE International Conference on*, pages 441–448. IEEE, 1996.

[205] A. Trotman. Compression, SIMD, and postings lists, 2014.

[206] J. A. Tyson. Large synoptic survey telescope: overview. In *Astronomical Telescopes and Instrumentation*, pages 10–20. International Society for Optics and Photonics, 2002.

[207] M. S. Uysal, C. Beecks, J. Schmücking, and T. Seidl. Efficient similarity search in scientific databases with feature signatures. In *Proceedings of the 27th International Conference on Scientific and Statistical Database Management - SSDBM '15*, pages 1–12, New York, New York, USA, jun 2015. ACM Press.

[208] J. Vanderplas, E. Soroush, K. S. Krughoff, M. Balazinska, and A. Connolly. Squeezing a big orange into little boxes: The ascotdb system for parallel processing of data on a sphere. *IEEE Data Eng. Bull.*, 36(4):11–20, 2013.

[209] P. Velentzas, M. Vassilakopoulos, and A. Corral. A partitioning GPU-based algorithm for processing the k Nearest-Neighbor query. In *Proceedings of the 12th International Conference on Management of Digital EcoSystems*, MEDES '20, pages 2–9, New York, NY, USA, Nov. 2020. Association for Computing Machinery.

[210] S. Vigna. Quasi-succinct indices. In *Proceedings of the sixth ACM international conference on Web search and data mining*, WSDM '13, pages 83–92, New York, NY, USA, Feb. 2013. Association for Computing Machinery.

[211] D. Wang, W. Yu, R. J. Stones, J. Ren, G. Wang, X. Liu, and M. Ren. Efficient GPU-Based query processing with pruned list caching in search engines. In *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*, pages 215–224, Dec. 2017.

[212] J. Wang, C. Lin, R. He, M. Chae, Y. Papakonstantinou, and S. Swanson. MILC: inverted list compression in memory. *Proceedings VLDB Endowment*, 10(8):853–864, Apr. 2017.

[213] Y. Wang. *Data Management and Data Processing Support on Array-Based Scientific Data.* PhD thesis, The Ohio State University, 2015.

[214] Y. Wang, Y. Su, and G. Agrawal. A novel approach for approximate aggregations over arrays. In *Proceedings of the 27th International Conference on Scientific and Statistical Database Management*, page 4. ACM, 2015.

[215] Y. Wang, Y. Su, G. Agrawal, and T. Liu. Scisd: Novel subgroup discovery over scientific datasets using bitmap indices. *Proceedings of Ohio State CSE Technical Report*, 2015.

[216] M. White. N-trees: large ordered indexes for multi-dimensional space. *Application Mathematics Research Sta, Statistical Research Division, US Bureau of the Census*, 1981.

[217] T. White. *Hadoop: The definitive guide.* " O'Reilly Media, Inc.", 2012.

[218] P. Wieschollek, O. Wang, A. Sorkine-Hornung, and H. Lensch. Efficient large-scale approximate nearest neighbor search on the gpu. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2027–2035, 2016.

[219] K. Wu, S. Ahern, E. W. Bethel, J. Chen, H. Childs, E. Cormier-Michel, C. Geddes, J. Gu, H. Hagen, B. Hamann, et al. Fastbit: interactively searching massive data. In *Journal of Physics: Conference Series*, volume 180, page 012053. IOP Publishing, 2009.

[220] K. Wu, E. J. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *ACM Transactions on Database Systems (TODS)*, 31(1):1–38, 2006.

[221] K. Wu, A. Shoshani, and K. Stockinger. Analyses of multi-level and multi-component compressed bitmap indexes. *ACM Transactions on Database Systems (TODS)*, 35(1):2, 2010.

[222] K. Wu, K. Stockinger, and A. Shoshani. Breaking the curse of cardinality on bitmap indexes. In *International Conference on Scientific and Statistical Database Management*, pages 348–365. Springer, 2008.

[223] K.-L. Wu and P. S. Yu. Range-based bitmap indexing for high cardinality attributes with skew. In *COMPSAC'98. Proceedings. The Twenty-Second Annual International*, pages 61–66. IEEE, 1998.

[224] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *Proceedings of the 18th international conference on World wide web*, WWW '09, pages 401–410, New York, NY, USA, Apr. 2009. Association for Computing Machinery.

[225] J. Yan, N. Xu, Z. Xia, R. Luo, and F.-H. Hsu. A compression method for inverted index and its FPGA-based decompression solution. In *2010 International Conference on Field-Programmable Technology*, pages 261–264, Dec. 2010.

[226] J. Yan, Z.-X. Zhao, N.-Y. Xu, X. Jin, L.-T. Zhang, and F.-H. Hsu. Efficient query processing for web search engine with FPGAs. In *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, pages 97–100, Apr. 2012.

[227] P. Zezula, P. Savino, G. Amato, and F. Rabitti. Approximate similarity retrieval with m-trees. *VLDB J.*, 7(4):275–293, Dec. 1998.

[228] H. Zhang, L. Liu, Y. Long, and L. Shao. Unsupervised deep hashing with pseudo labels for scalable image retrieval. *IEEE Trans. Image Process.*, 27(4):1626–1638, Apr. 2018.

[229] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. In *Proceedings of the 17th international conference on World Wide Web*, WWW '08, pages 387–396, New York, NY, USA, Apr. 2008. Association for Computing Machinery.

[230] K. Zhang, J. Hu, B. He, and B. Hua. DIDO: Dynamic pipelines for In-Memory Key-Value stores on coupled CPU-GPU architectures. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 671–682, Apr. 2017.

[231] W. Zhao, F. Rusu, B. Dong, and K. Wu. Similarity join over array data. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 2007–2022, New York, NY, USA, 2016. ACM.

[232] Y. Zheng, Q. Guo, A. K. H. 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, pages 2023–2037, New York, NY, USA, June 2016. Association for Computing Machinery.

[233] J. Zhou and A. K. H. Tung. SMiLer: A Semi-Lazy time series prediction system for sensors. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1871–1886, New York, NY, USA, May 2015. Association for Computing Machinery.

[234] J. Zhou, A. K. H. Tung, W. Wu, and W. S. Ng. R2-D2: a system to support probabilistic path predictionin dynamic environments via "Semi-Lazy" learning. *Proceedings VLDB Endowment*, 6(12):1366–1369, Aug. 2013.

[235] G. Zhu, Y. Wang, and G. Agrawal. Scicsm: novel contrast set mining over scientific datasets using bitmap indices. In *Proceedings of the 27th International Conference on Scientific and Statistical Database Management*, page 38. ACM, 2015.

[236] K. Zoumpatianos and T. Palpanas. Data series management: Fulfilling the need for big sequence analytics. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 1677–1678, 2018.

[237] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-Scalar RAM-CPU cache compression, 2006.

Publications of the Author

Reviewed Relevant Publications of the Author

[A.1] Krčál, Luboš and Ho, Shen-Shyang *A SciDB-based Framework for Efficient Satellite Data Storage and Query based on Dynamic Atmospheric Event Trajectory.* Proceedings of the 4th International ACM SIGSPATIAL Workshop on Analytics for Big Geospatial Data, BigSpatial@SIGSPATIAL 2015, pp. 7–14, Bellevue, WA, USA, 2015.

[A.2] Zhou, Jingbo and Guo, Qi and Jagadish, H V and Krčál, Luboš and Liu, Siyuan and Luan, Wenhao and Tung, Anthony K H and Yang, Yueji and Zheng, Yuxin *A Generic Inverted Index Framework for Similarity Search on the GPU.* 2018 IEEE 34th International Conference on Data Engineering (ICDE), pp. 893–904 Paris, France, 2018.

Remaining Relevant Publications of the Author

[A.3] Ho, Shen-Shyang and Krčál, Luboš *Supporting Research using Satellite Data: A Framework for Spatiotemporal Queries in SciDB.* AGU Fall Meeting Abstracts, IN51B–1802 2015.

[A.4] Krčál, Luboš and Ho, Shen-Shyang and Holub, Jan *Hierarchical Bitmap Indexing for Range and Membership Queries on Multidimensional Arrays.* arXiv, cs.DB, 2021.

[A.5] Zhou, Jingbo and Guo, Qi and Jagadish, H V and Krčál, Luboš and Liu, Siyuan and Luan, Wenhao and Tung, Anthony K H and Yang, Yueji and Zheng, Yuxin *A Generic Inverted Index Framework for Similarity Search on the GPU – Technical Report.* arXiv, cs.DB, 1603.08390, 2018.

Remaining Publications of the Author

[A.6] Krčál, Luboš and Holub, Jan *Incremental Locality and Clustering-Based Compression.* Data Compression Conference, DCC 2015, pp. 203–212, Snowbird, UT, USA, April 7–9, 2015.

[A.7] Procházka, Petr and Cvacho, Ondřej and Krčál, Luboš and Holub, Jan *Backward Pattern Matching on Elastic Degenerate Strings.* 14th International Joint Conference on Biomedical Engineering Systems and Technologies (BIOSTEC 2021) – Volume 3: BIOINFORMATICS, pp. 50–59, 2021.