MANAGING LARGE MULTIDIMENSIONAL DATASETS INSIDE A DATABASE SYSTEM

BY

KAUSHIK CHAKRABARTI

B.Tech, Indian Institute of Technology, Kharagpur, 1996
M.S., University of Illinois, Urbana-Champaign, 1999

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2001

Urbana, Illinois

Abstract

Many modern database applications deal with large amounts of multidimensional data. Examples include multimedia content-based retrieval (high dimensional multimedia feature data), time-series similarity retrieval, data mining/OLAP and spatial/spatio-temporal applications. To be able to handle multidimensional data efficiently, we need access methods (AMs) to selectively access some data items in a large collection associatively.

Traditional database AMs like B+-tree and hashing are not suitable for multidimensional data as they can handle only one dimensional data. Using multiple B+-trees (one per dimension) or space linearization followed by B+-tree indexing are not efficient solutions. We need multidimensional index structures: those that can index data based on multiple dimensions simultaneously. Most multidimensional index structures proposed so far do not scale beyond 10-15 dimensional spaces and are hence not suitable for high dimensional spaces that arise in modern database applications like multimedia retrieval (e.g., 64-d color histograms), data mining/OLAP (e.g., 52-d bank data in clustering) and time series/scientific/medical applications (e.g., 20-d Space Shuttle data, 64-d Electrocardiogram data). A simple sequential scan through the entire dataset to answer the query is often faster than using a multidimensional index structure.

To address the above need, we design and implement the hybrid tree, a multidimensional index structure that scales to high dimensional spaces. The hybrid tree combines the positive aspects of the two types of multidimensional index structures, namely data partitioning (e.g., R-tree and derivatives) and space partitioning (e.g., kdB-tree and derivatives), to achieve search performance more scalable to high dimensionalities than either of the above techniques. Our experiments show that the hybrid tree scales well to high dimensionalities for real-life datasets.

To achieve further scalability, we develop the local dimensionality reduction (LDR) technique to reduce the dimensionality of high dimensional data. The reduced space can be indexed more effectively using a multidimensional index structure. LDR exploits local, as opposed to global, correlations in the data and hence can reduce dimensionality with significantly lower loss of distance information compared to global dimensionality reduction techniques. This implies fewer false positives and hence significantly better search performance.

Another challenge in multidimensional indexing is handling time-series data which constitutes a major portion of all financial, medical and scientific information. We develop a new dimensionality reduction technique, called Adaptive Piecewise Constant Approximation (APCA), for time series data. APCA takes the idea of LDR one step further; it adapts locally to each time series object in the database and chooses the best reduced-representation for that object. We show how the APCA representation can be indexed using a multidimensional index structure. Our experiments show that APCA outperforms the other techniques by one to two orders of magnitude in terms of search performance.

Before multidimensional index structures can be supported as AMs in ”commercial-strength” database systems, efficient techniques to provide transactional access to data via the index structure must be developed. We develop concurrency control techniques for multidimensional index structures. Our solution, based on granular locking, offers a high degree of concurrency and has a low lock overhead.

An alternate technique to handle huge data volumes and fast search time requirements in multidimensional datasets is approximate query answering. This is especially true for decision support/OLAP applications where queries are usually exploratory in nature; fast approximate answers are often preferred to exact answers that take hours to compute. We develop a wavelet-based approximate query answering tool for DSS data. Our technique constructs compact synopses (comprising of wavelet coefficients) of the relevant database tables and subsequently answers any SQL query by working exclusively on the compact synopses. Our approach provides more accurate answers and faster response times compared to other approximate query answering techniques, namely random sampling and histograms, especially for high dimensional data.

Despite the increasing application need, commercial database management systems (DBMSs) lag far behind in their support for multidimensional data. One of the main reasons is the lack of scalable and effective techniques to manage large amounts of multidimensional data residing inside the DBMS. We believe that the techniques developed in this thesis address that problem. We hope that our solutions will encourage commercial database vendors to provide better support for multidimensional data in the future.

To my parents

Acknowledgments

First and foremost, I would like to thank my research advisor, Professor Sharad Mehrotra, for his exceptional guidance during the course of this research. He has been a constant source of motivation and was always available for technical discussions and professional advice.

I thank my academic advisor, Professor Geneva Belford, for her help and guidance. I thank Professors Geneva Belford, Marianne Winslett and Klara Nahrstedt for serving on my Qual committee. I thank Professors Josep Torrellas, Marianne Winslett, Kevin Chang and Leonard Pitt for serving on my Prelim and Final Defense committees.

I thank the members of our research group with whom I worked closely. Among them, I must particularly mention Michael Ortega and Kriengkrai Porkaew for being great colleagues and friends.

I collaborated with several other research groups, both at Illinois and Irvine. I worked with Yong Rui and Professor Thomas Huang of the Image Formation and Processing group at Illinois on multimedia content-based retrieval and multimedia feature indexing. I worked with Eamonn Keogh and Professor Michael Pazzani of the Machine Learning Group at Irvine on time series indexing. I had a great time working with them and I thank them for that.

I did the work on approximate query answering when I was visiting Bell Labs in the summer of 1999. I am grateful to Dr. Rajeev Rastogi, Dr. Minos Garofalakis and Dr. Kyuseok Shim for being great mentors and colleagues. Working with them was both educational and fun.

I thank Professors David Eppstein and Padhraic Smyth for the useful discussions on the Local Dimensionality Reduction technique.

For my research, I have often obtained datasets and code from other researchers. Stefan Berchtold gave us the FOURIER dataset. We obtained the hbπ-tree code from Georgio Evangelidis of North Eastern University. We obtained the R-tree code from Toni Guttman's web site at UCSC. Our lock manager implementation was derived from the lock manager code of MiniRel system provided to us by Mike Franklin. Vishy Poosala gave us his MaxDiff histogram computation code for our work on approximate query processing. I am grateful to all of them for their help with my research.

My research was primarily supported by NSF under NSF CAREER award IIS-9734300 and by Army Research Laboratory under Cooperative Agreement No. DAAL01-96-2-0003. It was supported in part by NSF/DARPA/NASA Digital Library Initiative Program under Cooperative Agreement No. 94-11318 and in part by NASA under Grant No. B9U415912.

I am thankful to Barb Cicone and Rhonda McElroy in the Academic Office of the Computer Science Department at Illinois for helping me with the official academic procedures at Illinois, especially during my stay at Irvine.

I thank all my friends, both at Illinois and Irvine (particularly, Arup, Avra, Deepa, Eamonn, Hemal, Kaushal, Michael, Nid, Prasun, Sathish, Saurabh, Sebastian and the two Vishals) for their friendship.

I owe a special debt of gratitude to my parents and family. I would not have been able to get this far without their constant support and encouragement. I thank my fiancee, Ratula Bhattacharya, for her support and love.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Many modern database applications deal with large amounts of multidimensional data. These applications include:

- *Multimedia Content-based Retrieval*: Such systems represent the visual content of multimedia objects (e.g., images) using features extracted from those objects. For example, for images, features include color histograms, texture vectors and shape descriptors. The extracted features are highly multidimensional in nature, i.e., we can view them as points in a high dimensional space (e.g., 64-dimensional space for color histograms, 16-d space for texture vectors [111]). The system allows the user to submit one or more query examples and request for the objects in the database that are visually most similar to those examples. The similarity of a database object to a query is defined as an aggregation of their similarities with respect to the individual features. The individual feature similarity measures as well as the aggregation function are chosen so as to capture the human perception of "similarity" and are dynamically fine-tuned by the system at query time to accurately reflect the subjective perception of the specific user [125, 118]. Multimedia similarity retrieval have numerous applications including e-commerce (e.g., find all shirts in the shopping catalog similar to chosen shirt), medical diagnosis/research (e.g., find all tumors with similar shape as the specified one [82]) and computer aided design (CAD).

- *Spatial/Spatio-temporal databases*: Spatial databases represent the positions of objects by their (x, y) (2-dimensional) or (x, y, z) (3-dimensional) co-ordinates and store them along with other object attributes [127]. Spatio temporal databases have an additional temporal dimension defining 3 (i.e., (x, y, t)) or 4 (i.e., (x, y, z, t)) dimensional spaces. Typical queries in such systems involve retrieving objects based on their positions and/or time [127]. For example, one might be interested in all vehicles within a mile of the location of an accident between 4-4:15pm when the accident happened.

- *Time Series/Scientific/Medical Databases*: Time series data account for a major fraction of all financial, medical and scientific data. Similarity search in time series data is useful not only as an end-user tool for exploratory data analysis but also as a component inside data mining algorithms like clustering, classification and association rule mining. Time series databases convert time series segments to multidimensional points using some transformation (e.g., Discrete Fourier Transform (DFT) [5, 46], Discrete Wavelet Transform (DWT) [29, 79], Singular Value Decomposition (SVD) [79, 76, 81]). Similarity search is then performed on the transformed data. Example applications include a doctor searching for a particular pattern (that implies a heart irregularity) in the ECG database for diagnosis, a stock analyst searching for a particular pattern in the stock database for prediction etc. [78]. Multidimensional data is common in scientific and medical databases as well. For example, the Sloan Digital Sky Survey (SDSS) astronomy database will be storing 200 million objects (galaxies, stars and quasars) with mostly numeric attributes (e.g., position, color, shape etc.) defining a 100-dimensional space [140]. Astronomers would then run spatial proximity queries, similarity queries, multidimensional range queries etc. on the high dimensional, multi-terabyte database. In the medical area, multidimensional features are extracted from medical data (e.g., tumor images in [82], ECG data in [78]) which can then be used for similarity retrieval (e.g., find similar tumors in [82], find patterns in ECG data [78]) for the purpose of diagnosis and/or forecasting.

- *Data Mining/OLAP*: In a database, each data record contains values for several attributes which together define a multidimensional space. For example, in the Census Population Survey database, each person record contains information on age, income, educational attainment, full/part-time work etc. of the person [20]. An OLAP query may involve finding the average income of all people between 35 and 45 years of age with educational attainment $\geq$ Bachelors degree (a 2-dimensional query). A data visualization application may be interested in visualizing all people as points in the age-income space. The visualization application would also like to perform zoom in/out operations and visual query constructions on the displayed space. A data mining application may run an algorithm to find correlations between age and income in the dataset.

Although several emerging application domains deal with large amounts of multidimensional data, commercial database management systems (DBMSs) lag far behind in their support for multidimensional data and are *not* able to support such applications efficiently. One of the main problems is inadequate support for multidimensional access methods. Access methods (AMs) provide an efficient way to selectively access some data items in a large collection associatively. For example, a similarity query in multimedia retrieval needs to access color feature vectors that are “close to” the color feature vector of the query image. Scanning the entire vector database to determine the close ones is usually too slow, especially when the database is large and resides on disk. Figure 1.1 shows the time taken by linear scan to retrieve the color histograms close to a given color histogram over a 64-dimensional color histogram database. The time increases linearly with the size of the database and takes 9 minutes for a 1 million item database. We need an access method that allows the application to access those vectors close to the query vector without having to see all the other vectors in the database. Traditional database AMs like B+-tree and hashing allow such accesses

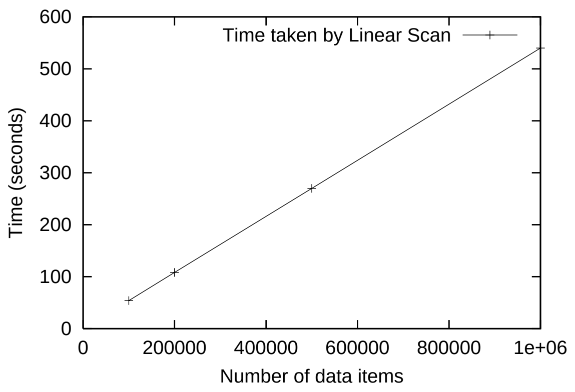


Figure 1.1: Time taken to answer a color similarity query on a 64-dimensional color histogram database using linear scan. The experiment was conducted on a Sun Ultra Enterprise 450 machine with 1 GB of physical memory and several GB of secondary storage, running Solaris 2.6.

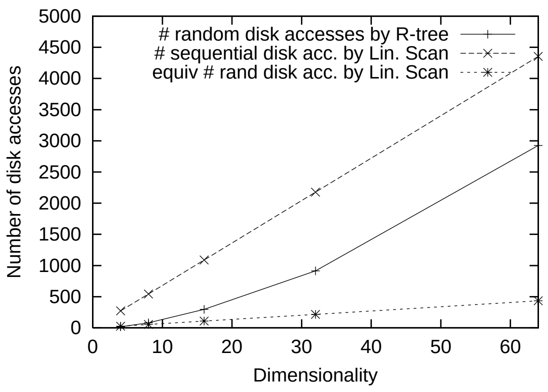


Figure 1.2: Cost of answering a range query on a 70K color histogram database using R-tree and linear scan (range query selectivity=0.2%). The cost of linear scan in terms of random disk accesses is computed as $\frac{\text{\# sequential disk accesses}}{10}$.

for one dimensional data (i.e., linearly orderable data). These AMs cannot be directly used to access multidimensional data as there is no linear order among points in a multidimensional space [88, 126]. There does exist some indirect ways to use 1-d AMs to index multidimensional data. For example, one can index each dimension of the multidimensional space using a B+-tree as shown in Figure 2.1. As discussed in Chapter 2, this technique turns out to be extremely costly, especially at high dimensionalities. Another indirect mechanism is to map mapping the multidimensional keys to one dimensional keys using a space filling curve like the Z-order; the resulting 1-d key space can then be indexed using a B+-tree. Once again, as shown in Figure 2.2, this technique is usually very expensive. We need index structures that can index data based on multiple dimensions simultaneously: they are known as multidimensional index structures.

Work on multidimensional index structures dates back to early 1980s. The first multidimensional index structures to be proposed were the spatial index structures (e.g., R-tree [59], kDB-tree [120], grid file [105]). Although the above index structures work well at the low dimensional spaces (2-5 dimensions) which they are designed for, they are not suitable for high dimensional spaces that arise in modern database applications like multimedia retrieval (e.g., 64-d color histograms), data mining/OLAP (e.g., 52-d bank data in clustering [2]) and time series/scientific/medical applications (e.g., 20-d feature vectors extracted from Space Shuttle data [79], 100-d SDSS data [140], 64-dimensional ECG data [78]). A simple sequential scan through the entire dataset to answer the query is often faster than accessing the data using a spatial access method [15, 16]. Figure 1.2 shows the cost of answering a range query on a 70,000-item color histogram database using R-tree and linear scan techniques for various dimensionalities. As the dimensionality increases, linear scan significantly outperforms R-tree in terms of random disk accesses. [1] We need indexing mechanisms

[1]We have ignored the CPU cost of the query in this example; the CPU cost of the linear scan technique is usually higher compared to R-tree [23]. Even when CPU cost is considered, linear scan is faster than the R-tree at high dimensionalities, so Figure 1.2 represents the trend accurately.

that can scale to high dimensionalities as discussed in the next section.

1.2 Challenges

The main challenges in managing large, complex multidimensional datasets inside a DBMS include:

High Dimensional Index Structures : We need multidimensional index structures that, unlike the spatial index structures discussed above, scale to high dimensionalities. Although several index structures have been proposed for high dimensional spaces, none of them scale beyond 10-15 dimensions. Most of them are variants of the R-tree (we refer to them as data partitioning index structures, cf. Section 3.2) and hence suffer from the same problems as the R-tree, viz., low node fanout and high degree of overlap between the bounding regions. The other class of multidimensional index structures, namely space partitioning index structures (cf. Section 3.2) do not suffer from the above limitations. However, they have their own share of problems (e.g., no guaranteed utilization (kDB-tree), storage of redundant information (hB-tree)) as discussed in Chapter 3. We need to develop index structures that overcome the above limitations of existing multidimensional index structures. Like B+-trees, the developed index structures should be paginated (so that we do not need the entire structure to fit in memory), height-balanced and have high node fanout. The index structure should support range and k-nearest neighbor (k-NN) searches based on arbitrary distance functions so that we can use the index to answer similarity queries (e.g., the color query above [43]) based on the similarity measure that best captures the perception of the user.[2].

Dimensionality Reduction Techniques : While a scalable index structure would be a big step towards enabling DBMSs to efficiently support queries over high dimensional data, further improvements are possible. High dimensional data often have highly correlated distributions [140, 24]. In order to exploit such correlations, a dimensionality reduction technique (e.g., Principal Component Analysis (PCA) [48]) is used in conjunction with a high dimensional index structure. The idea is to first reduce the dimensionality of the data and then index the reduced space using a multidimensional index structure [43]. If PCA is able to condense *most* of the distance information in the first few dimensions (the first few principal components (PCs)), the index, being built on a lower dimensional space, will be able to evaluate queries more efficiently than the index on the original high dimensional space. A good "condensation" is possible only when the data set is globally correlated, i.e., most of the variation in the data can be captured by a few (arbitrarily oriented) dimensions. In practice, datasets are often not globally correlated. In such cases, the above technique, referred to as global dimensionality reduction (GDR), causes significant loss of distance information resulting in a large number of false positives and hence a high query cost. A key observation is that even

[2]Here we consider answering single feature similarity queries using the F-index (e.g., the color query using color index). Multifeature queries (e.g., find similar images with respect to both color and texture) are typically evaluated by retrieving the similar items with respect to each individual feature (i.e., individual color and texture matches) using the corresponding F-indices and then merging them using a merging algorithm as described in [42, 111].

when a global correlation does not exist, there may exist subsets of data that are locally correlated. GDR can not exploit such local correlations. A technique that can discover such local correlations in data and exploit those correlations for building the reduced-space index can significantly enhance the scalability of a multidimensional index structure.

Time Series Indexing Techniques : Similarity search in time series databases poses several new indexing challenges. It is a difficult problem because of the typically high dimensionality of the raw data. For example, the raw ECG data in [78] has dimensionality between 256 and 1024. The most promising solution involves performing dimensionality reduction on the data, then indexing the reduced data with a multidimensional index structure. All dimensionality reduction techniques proposed so far for time-series data (e.g., DFT, DWT, SVD) are global techniques; they choose a common representation for all the items in the database that minimizes the global reconstruction error. A technique that adapts locally to each time-series item and chooses the best reduced-representation for that item (i.e., the one with the lowest reconstruction error for that item) can reduce the dimensionality of time-series data with significantly lower loss of information. If such a representation can be indexed using a multidimensional index structure, it, due to its high fidelity to the original signal, would support much faster similarity search compared to previous dimensionality reduction techniques.

Integration of Multidimensional Index Structures to DBMSs : While there exists several research challenges in designing scalable index structures, one of the most important practical challenges is that of integration of multidimensional index structures as access methods (AMs) in a DBMS. Building a database server with native support for all possible kinds of complex data and index structures that covers all application domains is not feasible. The solution is to build an extensible database server that allows the application developer to define her own data types and operations on those data types as well as her own indexing mechanisms on the stored data which the database query optimizer can exploit to access the data efficiently. Commercial ORDBMSs already support user-defined data types and operations and have recently started providing extensibility options for users to incorporate their own index structures [18, 135]. However, the interfaces exposed by commercial systems for index structure integration are too low level and places too much burden (e.g., writing code to pack records into pages, maintain links between pages, handle concurrency control etc.) on the AM implementor. The *Generalized Search Tree* (GiST) [63] provides a more elegant solution to the above problem by providing a higher level interface and abstracting out the primitive page-level operations from the AM implementor. The AM implementor just needs to register a few extension methods with GiST. GiST implements the standard index operations, search, insertion and deletion, with the help of those methods provided by the AM implementor, who thus controls the behavior of the search operation and organization of keys within the tree, thereby customizing GiST to her desired AM. Although GiST considerably reduces the effort of integrating multidimensional index structures as AMs in DBMSs, before it can be supported in a "commercial strength" DBMS, efficient techniques to support concurrent access to data via the GiST must be developed. Developing concurrency control (CC) techniques for GiST is particularly beneficial since it would need writing the CC code only once and would allow concurrent access to the database via any multidimensional index structure implemented in the DBMS using GiST, thus avoiding the need to write the code for each index structure separately.

Approximate Query Answering for Decision Support Applications : Another problem in multidimensional data management is dealing with huge data volumes and stringent response time requirements in decision support/OLAP systems. Multidimensional index structures are not always the best option for accessing OLAP data as OLAP queries may involve selections with unrestricted dimensions [128]. For example, the query “get the average income of all people between 35 and 45 years of age with educational attainment $\geq$ Bachelors degree” mentioned above is unrestricted along the income and full/part-time work dimensions and is only left-restricted along the educational attainment dimension. Assuming that the index is built on all dimensions, such queries may cause accessing large portions of the index structure leading to high cost [128]. [3] Furthermore, OLAP queries may involve joins with usually just 1 or 2 join dimensions. There is no efficient way to handle such queries using indexes built on all dimensions. Alternate techniques need to developed for managing multidimensional data for OLAP applications. Approximate query answering has recently emerged as a viable solution to this problem. Approximate answers are often acceptable in DSS applications as such applications are usually exploratory in nature. For example, during a drill-down query sequence in ad-hoc data mining, the main purpose of the initial queries in the sequence is to determine the truly interesting queries and regions in the database. Computing the exact answers for such queries would unnecessarily waste time and system resources. Providing fast and accurate approximate answers, on the other hand, would enable the users to focus on their explorations quickly and effectively. The general approach to approximate query answering is to first construct compact synopses of interesting relations in the database (using a data reduction technique like random sampling, histograms, wavelets etc.) and then answering the queries by using just the synopses (which usually fit in memory). Approximate query answering techniques proposed so far either suffer from high error rates (e.g., random sampling techniques for joins and non-aggregate queries, histogram techniques at high dimensions) or are severely limited in their query processing scope (e.g., wavelet-based techniques). We need to develop approximate query answering techniques that are accurate, efficient and general in their query processing scope.

1.3 Contributions and Structure of Thesis

In this thesis, we analyze the problems posed by the above challenges and design, implement and evaluate techniques to efficiently manage large, complex multidimensional datasets inside a database system. The developed techniques include:

[3] Techniques to overcome the above problem has been proposed in the literature [107, 128]. Multidimensional index structures have been used to index OLAP data in [39, 122].

- **Index Structure for High Dimensional Spaces:** We design an index structure, namely the *hybrid tree*, that scales to 50-100 dimensional spaces. Such dimensionalities are common for multimedia features (e.g., 64-d color histograms) and time-series/scientific/medical applications (e.g., 20-d feature vectors extracted from Space Shuttle data [79], 100-d space in SDSS [140], 64-d ECG data [78]). All previously proposed multidimensional index structures are either purely data partitioning (DP) (e.g., R-tree and its variants) or space partitioning (SP) (e.g., kDB-tree and its variants) in nature (see Section 3.2). We explore a *"hybrid"* technique that combines the positive aspects of the two types of index structures into a single data structure to achieve search performance more scalable than either of the above techniques. The hybrid tree is disk-based, height balanced and have high node fanout (independent of data dimensionality). It supports range and k-NN searches based on arbitrary distance functions. Our experiments on real-life datasets show the hybrid tree scales well to high dimensionalities and significantly outperforms both DP-based and SP-based index structures as well as sequential scan (which is a competitive technique for high dimensional data) at all dimensionalities.

- **Local Dimensionality Reduction for High Dimensional Indexing**: To improve the scalability of the hybrid tree even further, we propose a new dimensionality reduction technique called Local Dimensionality Reduction (LDR). LDR discovers local correlations in the data and performs dimensionality reduction on the locally correlated clusters individually. We develop an index structure (based on the hybrid tree [4]) that exploits the correlated clusters to efficiently support point, range and k-nearest neighbor queries over high dimensional datasets. Our technique guarantees that for any query, the reduced-space index returns the same answers as it would have if the query was executed in the original space (referred to as *"exact searching"*). LDR marks a significant improvement over the GDR technique which works well only when the data is globally correlated and cannot exploit local correlations in data. Our experiments on synthetic as well as real-life datasets show that our technique (1) reduces the dimensionality of the data with significantly lower loss in distance information compared to GDR (smaller number of false positives) and (2) significantly outperforms the GDR, original space indexing and linear scan techniques in terms of the query cost for both synthetic and real-life datasets.

- **Locally Adaptive Dimensionality Reduction for Time Series Data**: We introduce a new dimensionality reduction technique which we call Adaptive Piecewise Constant Approximation (APCA). While previous techniques (e.g., SVD, DFT and DWT) choose a common representation for all the items in the database that minimizes the global reconstruction error, APCA approximates each time series by a set of constant value segments of varying lengths such that their individual reconstruction errors are minimal. We show how APCA can be indexed using a multidimensional index structure. Since our distance measure in the APCA space lower bounds the true distance (i.e., the Euclidean distance in the original space), we guarantee exact searching, i.e., return the same answers as the original space

[4]Any multidimensional index structure can be used in conjunction with the LDR technique. We used the hybrid tree in our experiments for LDR due to its scalability of high dimensions.

index. Our experiments show the APCA outperforms DWT, DFT and SVD by one to two orders of magnitude in terms of search performance.

- **Concurrency Control Techniques to enable integration of multidimensional index structures as AMs in a DBMS**: We develop efficient techniques to provide transactional access to data via multidimensional index structures. Concurrent accesses to data via index structures introduce the problem of protecting ranges specified in the retrieval from phantom insertions and deletions (the *phantom problem*). We propose a dynamic granular locking approach to phantom protection in GiSTs. The granular locking technique offers a high degree of concurrency and has a low lock overhead. Our experiments show that the granular locking technique (1) scales well under various system loads and (2) similar to the B-tree case, provides a significantly more efficient implementation compared to predicate locking for multidimensional AMs as well. Since a wide variety of multidimensional index structures can be implemented using GiST, the developed algorithms provide a general solution to concurrency control in multidimensional AMs. The other advantage of developing the solution in the context of GiST is, as mentioned earlier, that the CC code would have to written only once. To the best of our knowledge, our proposal is the first such solution for multidimensional index structures based on granular locking.

- **Wavelet-based Approximate Query Processing Tool for DSS applications**: We develop a wavelet-based approximate query answering tool for high-dimensional DSS applications. We construct a compact and approximate synopsis of interesting tables based on multiresolution wavelet decomposition. We propose a novel wavelet decomposition algorithm that can build these synopses in an I/O-efficient manner. We develop novel query processing algorithms that can answer any SQL query by working just on the compact synopsis. This guarantees extremely fast response times since our approximate query execution engine can do the bulk of its processing over compact sets of wavelet coefficients, essentially postponing the expansion into relational tuples till the very end of the query. Unlike previous techniques, we can handle all types of queries: aggregate queries, GROUP BY queries as well as queries that return relations as answers (i.e., select-project-join queries). Our experiments on synthetic as well as real-life data sets show that our techniques (1) provide approximate answers of better quality than either sampling or histograms, (2) offer query execution-time speedups of more than two orders of magnitude, and (3) guarantee extremely fast synopsis construction times that scale linearly with the size of the data.

In addition to developing and implementing the above techniques individually, we have the integrated some of these techniques into the MARS database system. MARS (*Multimedia Analysis and Retrieval System*) is a new object-relational DBMS that supports flexible and customizable similarity-based search and ranked retrieval over arbitrary, application-defined data-types. [5] MARS consists of 100,000 lines of C++

[5]MARS also provides built-in support for query refinement (via relevance feedback) in order to improve the quality of search results.

code and is fully operational at this moment. MARS allows applications to create new data-types and define the meaning of similarity for those data-types. For example, an image content-based retrieval application can create a new image datatype and define a function that, given two images, returns the similarity between them. Subsequently, the application can run image similarity queries on MARS; MARS would compute the results based on the application-specified similarity functions. MARS allows queries to have exact search conditions in addition to similarity search conditions; the results are filtered based on the exact conditions and ranked based on the similarity conditions. Applications that deal with multidimensional data including multimedia content-based retrieval, spatial/spatio-temporal retrieval, time-series retrieval and data mining applications, can easily be supported on MARS. To support similarity search in such applications efficiently, the MARS index manager (MARS/IM) supports multidimensional AMs in addition to one-dimensional AMs like B+-trees. The multidimensional AMs supported by MARS/IM include the R-tree and the hybrid tree (they are supported only as secondary AMs). An application can create a multidimensional index of either type on one or more attributes of a relation. Similarity queries on a relation can then be answered by running a range query or a k-NN query on the index [6], the distance function will be chosen based on the application-specified similarity function. MARS allows the distance functions to be dynamically modified by the application at query time in order to adapt to the user's subjective perception of similarity. As expected, using the index significantly speeds up similarity queries in MARS, often by several orders of magnitude. We plan to integrate some of the other techniques developed in this thesis into MARS as well.

The rest of the thesis is organized as follows. Chapter 2 provides a background on multidimensional data management techniques. Chapters 3 to 7 form the core of this thesis. Chapter 3 introduces the hybrid tree, an index structure that scales to high dimensional feature spaces. Chapter 4 describes the local dimensionality reduction (LDR) technique. Chapter 5 proposes the locally adaptive dimensionality reduction technique, namely adaptive piecewise constant approximation APCA, for indexing time series data. In Chapter 6, we present concurrency control techniques in order to enable integration of multidimensional index structures as AMs in a DBMS. Chapter 7 describes our wavelet-based approximate query answering tool for DSS data. Finally, in Chapter 8, we summarize the contributions of this thesis and outline some directions for future research.

[6]The decision as to whether to use an index or not is made by the query optimizer – so the system may not always use the index to answer a similarity query.

Chapter 2

Background on Multidimensional Data Management Techniques

The purpose of this chapter is to provide a background on multidimensional data management techniques. We start with an overview of database access methods, mainly the B+-tree. In Section 2.2, we describe two ways B+-trees can be used to index multidimensional data and point out their limitations, thereby motivating the need for multidimensional index structures. Section 2.3 presents the R-tree, the most popular multidimensional index structure, and serves as a background for Chapter 3. Section 2.4 discusses the limitations of R-tree and motivates the need for index structures that scale to high dimensionalities. In Section 2.5, we present dimensionality reduction techniques and discuss their limitations. Section 2.6 provides an overview of existing dimensionality reduction techniques for time series indexing, highlighting their weaknesses. Section 2.7 presents concurrency control techniques for B-trees and explains why they cannot be applied for concurrency control in multidimensional access methods. Finally, in Section 2.8, we provide a background on approximate query answering techniques.

2.1 Access methods

Access methods (AMs) provide an efficient way to selectively access some data items in a large collection associatively. Consider a directory of all people in UIUC being stored as a relation in a database. Consider a query for all people with last name “Smith*” (“*” denotes wildcard) on the above relation. If there are no way to access the relation associatively by last name, the entire relation would have to be scanned to answer the query and every item in the relation would have to be examined. This technique is usually too slow, especially for large relations. Now let assume that there exists an ordering AM on last name. The AM can either keep the directory (i.e., the relation) itself sorted by last name (primary index) or maintain a separate sorted list of last names with pointers into the full records in the directory (secondary index). The AM can answer the above range query in time linear in the number of names in the range, after an initial search logarithmic in the size of the directory (e.g., binary search). The B+-tree is a robust ordering AM that is ubiquitous in database systems. It is a paginated search tree (i.e. each node corresponds to a disk page)

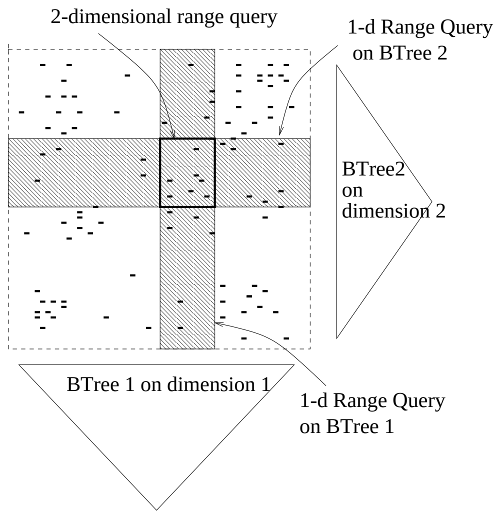

2-dimensional range query
1-d Range Query on BTree 2
BTree2 on dimension 2
BTree 1 on dimension 1
1-d Range Query on BTree 1

Figure 2.1: Using multiple B-trees to access multidimensional data.

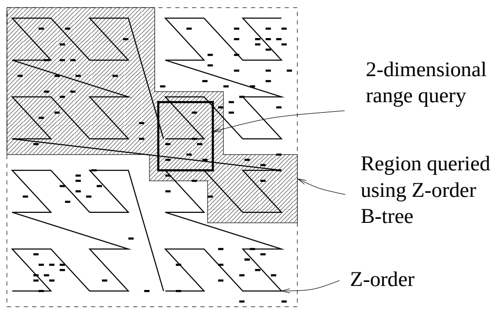

2-dimensional range query
Region queried using Z-order B-tree
Z-order

Figure 2.2: Using Z-order to linearize multidimensional space and indexing linearized space using a B-tree.

with high fanout nodes that is used for one dimensional (i.e., linearly orderable) key spaces (e.g., integers, floats, strings). B+-trees grow bottom up by splitting overfull nodes, followed by posting of index terms higher in the tree. Searchers touch only $log_{fanout}(filesize)$ pages (approximately), which is a factor of 8 improvement over binary search when fanout is 256, a typical value. Storage utilization with node splitting is about 69%.

2.2 Inadequacy of B-trees

The B-tree, being an ordering AM, cannot index data based on multiple dimensions simultaneously as there is no linear order between, say, two 2-d points $(2, 5)$ and $(4, 2)$. There are two ways a B-tree can be used for indexing multidimensional data:

- Using multiple B-trees, one per dimension: This approach is shown for a 2-d space in Figure 2.1. This approach is inefficient as at most one of the indexes can be a clustering (i.e. primary) index. If neither BTree1 or BTree2 are primary indices, a 2-d range query (that requests all points contained in the range) would need to execute a 1-d range query on each BTree (shaded regions) and then take the intersection of the results, thus accessing much more data than is necessary for the 2-d range query. If one of them is a primary index (say, BTree1), only one 1-d range queries needs to executed which still would access more data than necessary. This problem becomes more severe at high dimensionalities. Insertion and deletions also create problems as all the indices need to be updated.
- Linearizing multidimensional space and then using a B-tree: This approach is shown for a 2-d space in Figure 2.2. Linearization is achieved by Z-order but other ordering techniques (e.g., Hilbert curve) can be used instead [72]. This solution is also inefficient as a 2-d range query (same as the one in Figure 2.1) needs to access irrelevant regions of the data space (shaded region) just because those regions

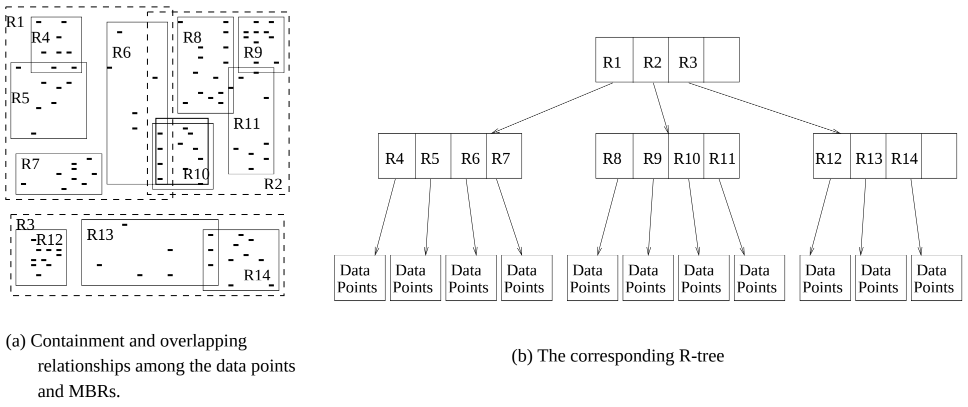

R1
R4
R5
R6
R7
R8
R9
R10
R11
R2
R3
R12
R13
R14
R1 R2 R3
R4 R5 R6 R7
R8 R9 R10 R11
R12 R13 R14
Data Points
(a) Containment and overlapping relationships among the data points and MBRs.
(b) The corresponding R-tree

Figure 2.3: A 2-dimensional R-tree.

happens to lie within the upper and lower bounds of the query according to superimposed Z-order. This would also create problems in terms of access method concurrency control as the searches would acquire locks on much larger regions than is necessary, leading to low concurrency and high lock overhead (see Chapter 6 for details). Once again, the problems are greatly exacerbated by increasing dimensionality.

2.3 R-trees

From the above discussion, it is clear can we need multidimensional index structures to efficiently access multidimensional datasets associatively. One of the earliest multidimensional index structures to be proposed is the R-tree [59]. It is also one of most popular ones and several variants of the R-tree have been proposed in the last few years (e.g., R+-tree, R*-tree, VAMSplit R-tree). To the best of our knowledge, it is the only true multidimensional index structure supported by a commercial DBMS [141]. An example R-tree (for the same point distribution as Figures 2.1 and 2.2 is shown in Figure 2.3. It recursively clusters the multidimensional data using minimum bounding rectangles (MBR), forming a hierarchical tree structure (e.g., a 3 level tree in Figure 2.3). Like the B-tree, it is height balanced and paginated (i.e., the nodes correspond to disk pages). The leaf nodes contain either the actual tuples (if it is a clustering/primary index) or just the multidimensional keys along with a pointer to the actual tuple (if it is a secondary index). Non-leaf nodes contain entries of the form $\langle MBR, child\_pointer\rangle$ where $child\_pointer$ is the address of a lower level node in the R-tree and MBR is the smallest rectangle the spatially contains all the items in the lower node's entries. The R-tree guarantees a node utilization bound i.e. every node contain between m and M entries except the root (m and M can be different for leaf and non-leaf nodes). Although Figure 2.3 shows point data, R-tree can store data with spatial extents (e.g., polygons).

A range search in the R-tree (to find all points contained in a rectangular box) proceeds top-down from the root by determining the overlapping entries in the node and recursively searching the corresponding

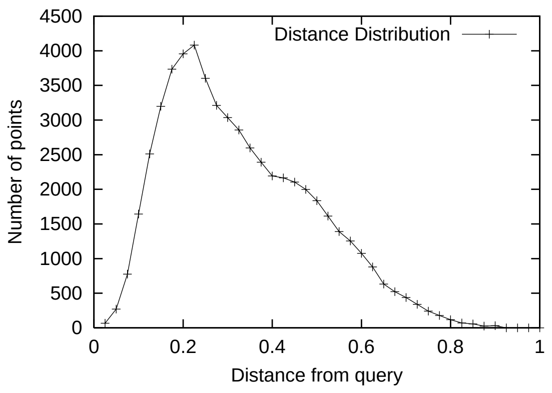


Figure 2.4: Average distances of points from a randomly chosen query point (64-d COLHIST data, L1 distance).

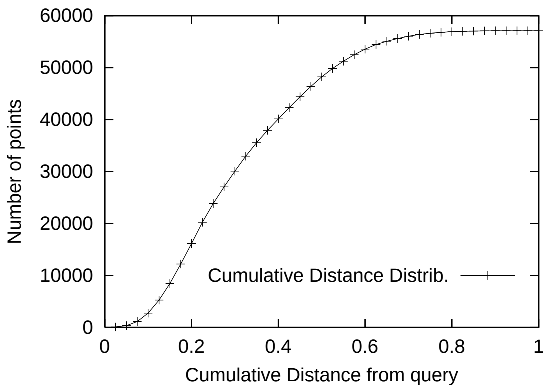


Figure 2.5: Cumulative version of the distribution in Figure 2.4.

child subtrees. For example, the 2-d range query shown in Figure 2.3 (same as the ones in Figures 2.1 and 2.2) would first examine the root node and determine that $R1$ and $R2$ are the only overlapping entries. [1] Then, it would explore the entries inside $R1$ and determine that it only overlaps with $R6$. Subsequently, it checks the leaf node corresponding to $R6$ and adds all the qualifying points to the result. Then, it would explore the entries inside $R2$ and determine that $R10$ is the only overlapping one: so it accesses the leaf node corresponding to $R10$ and adds all the qualifying points to the result. The R-tree can also support the k-nearest neighbor (k-NN) query efficiently, i.e., find the k nearest neighbors to a given point in terms of Euclidean distance [121, 66]. K-NN queries with respect to other distance functions has been studied recently [82, 131, 21].

Inserting a new point in the R-tree involves selecting a leaf node L to place the point (by starting from the root and recursively selecting the node that requires the least enlargement to accommodate the point), and placing the point in L. If the boundary of L changes or L is split (because it became overfull) due to the insertion, the changes are recursively propagated up the tree. The node splitting algorithm of the R-tree bipartitions the objects in the node such that the sum of the areas of the two MBRs after the split is minimized. Several optimizations of the R-tree bipartitioning algorithm have been proposed in the literature (e.g., new criteria like minimizing the overlap between the MBRs, minimizing the perimeter of the MBRs etc.) [11, 50]. The deletion takes place by locating the leaves L that may contain the point, looking for the point in those leaves and deleting it if found. If the leaf becomes underfull, it is deleted and the changes are propagated up the tree. We refer the interested readers to [59] for further details.

2.4 Dimensionality Curse

The R-tree and its variants (e.g., R+-tree, R*-tree) work well at 2-5 dimensional spaces. Beyond 5 dimensions, the performance of these index structures deteriorate rapidly. A simple sequential scan through the entire dataset (examining each item to determine whether it qualifies as an answer) turns out to be faster than using the R-tree (see Figure 1.2). This phenomenon, commonly known as the "dimensionality curse", occurs due to many reasons. First, let us consider the R-tree-specific reasons:

- *High Overlap*: There is a high degree is overlap between the index nodes of the R-tree at high dimensions [15]. Overlap increases the average number of paths a search needs to follow, thereby increasing query cost.
- *Low Fanout*: The fanout of the nodes decreases linearly with the increase in dimensionality. If the disk page size is 4KB, the fanout of an R-tree non-leaf node at 64-d drops to about 7. Lower the fanout, deeper the tree, higher the cost.

The above problems occur in all bounding region based index structures (also referred to as *data partitioning*(DP) index structures) no matter what the shape of the region is. Examples include SS-tree (that uses minimum bounding spheres [149]), SR-tree (that uses both rectangles and spheres [77]), X-tree (that uses rectangles [15]), M-tree (where region shape is determined by the L_p metric used, i.e., diamonds if L_1 is used, spheres if L_2 is used etc. [33]). *Space partitioning* (SP) index structures overcome the above problems by always splitting nodes along one dimension [2] (in contrast to DP-based structures that use all the dimensions to split) and representing the partitioning inside an index node using a kd-tree [90]. This eliminates overlap and makes the node fanout independent of dimensionality. However, existing SP-based techniques have other problems (e.g., no guaranteed node utilization in kdB-trees, redundant information in hB-trees) which will be discussed in detail in Chapter 3.

Another reason for dimensionality curse is the increasing sparsity of high dimensional spaces [146]. If the space is sparse, the nearest neighbors to a query point P would be far away from P (as well as from each other) requiring the k-NN algorithm to explore a larger region in space to return the answers and hence accessing more nodes of the index structure (i.e. more disk pages). This effect is most severe in uniformly distributed datasets [16, 146]. In a uniformly distributed dataset, above a certain dimensionality, all points are more or less equidistant (equally far) from each other, raising the question of "meaningfulness" of nearest neighbor queries in high dimensional spaces [16]. This is *not* the case with most real-life datasets and hence nearest neighbor queries *are* meaningful for such datasets. An example distance distribution for a real-life 64-d color histogram dataset (based on L1 distance) is shown in Figures 2.4 and 2.5. The figures show that all points are certainly not equidistant from the query point; a few points are close of the query point while most points are far from it. However, even real-life feature spaces do exhibit the some sparsity effect and sequential scan would start to outperform an index scan above a certain "cut-off" dimensionality. Our goal

[1]We are using the Ri's to denote both the node and the corresponding MBR. For example, $R1$ denotes the the node containing entries $(R4, R5, R6, R7)$ in Figure 2.3(b) and the corresponding MBR shown by the dashed rectangle in Figure 2.3(a).

[2]hB-tree, although a SP-based technique , uses multiple dimensions to split [90].

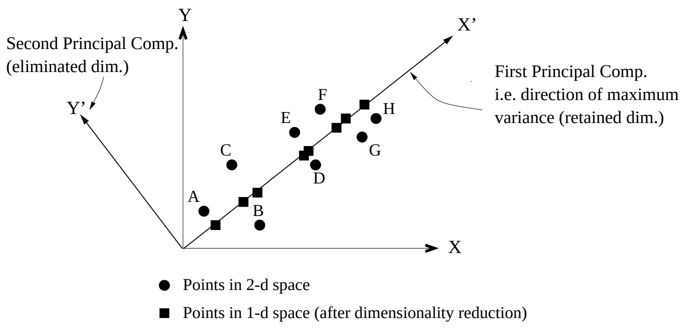


Figure 2.6: Principal Component Analysis on 2-d data.

is to design indexing techniques that have high cut-off dimensionality (above 100) so that it is useful for indexing feature spaces that arise in most real life applications (2-100 dimensional spaces).

2.5 Dimensionality Reduction Techniques

A common technique to overcoming dimensionality curse is to use a dimensionality reduction technique in conjunction with a multidimensional index structure. The most commonly used dimensionality reduction technique is Principal Component Analysis (PCA) [38, 48]. PCA examines the variance structure in the data and determines the directions (which are linear combinations of the original dimensions) along which the data exhibits high variance. The first direction (called the first principal component (PC)) accounts for as much of the variability in the data as possible, and each succeeding PC accounts for as much of the remaining variability as possible. Figure 2.6 shows a set of points and the two PCs (X' and Y'). Since the first few PCs account for most of the variation in the data, the rest can be eliminated without significant loss of information. For example, in Figure 2.6, the second principal component Y' can be eliminated, thus reducing the dimensionality from 2 to 1. The 1-d images of the 2-d points are obtained by projecting them on the first principal component X' (shown by squares in Figure 2.6). The position of any point along an eliminated component is assumed to be the mean value of all points along that component. It can be shown that PCA is the optimal way to map points in a D-dimensional space to points in a d-dimensional space ($d \leq D$), i.e., it minimizes the mean square error, where the error is the distance between each D-d point and its d-d image [48].

Algebraically, the principal components are computed as follows. Let A be the $N \times D$ data matrix whose each row corresponds to a point in the original D-dimensional space (N is the number of points in the dataset). The first principal component is the eigenvector corresponding to the largest eigenvalue of the variance-covariance matrix of A, the second component correspond to the eigenvector with the second largest eigenvalue and so on. The mapping (to reduced dimensionality) corresponds to the well known Singular Value Decomposition (SVD) of data matrix A and can be done in $O(ND^2)$ time.

The reduced dimensional points can be indexed more effectively using a multidimensional index struc-

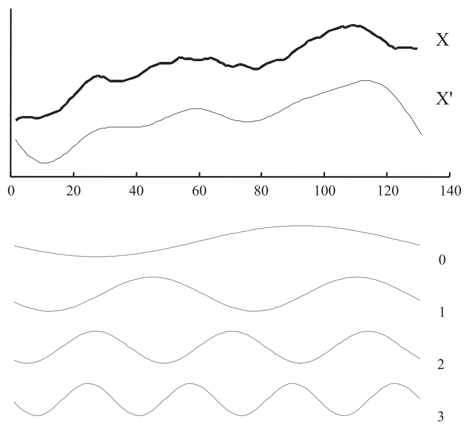


Figure 2.7: The first 4 Fourier bases can be combined in a linear combination to produce X', an approximation of the original sequence X. Each basis wave requires two numbers to represent it (phase and magnitude), so reduced dimensionality $N = 2 \times 4 = 8$ in this case.

ture. It can be shown that distances in the reduced space satisfies the lower bounding lemma [43]:

$$\mathcal{D}(Image(A), Image(B)) \leq \mathcal{D}(A, B) \tag{2.1}$$

where A and B are two points in the original space, $Image(A)$ and $Image(B)$ are their images in the reduced space and $\mathcal{D}$ is any L_p metric. The above property guarantees that executing the query on the reduced space index cannot result in any false dismissals and hence (by appropriate post-processing) can produce the exact same results as original space querying (see Chapter 4 for details).

One of the main limitation of PCA is that it works well only when the dataset is globally correlated, i.e., most of the variation in the data can be captured by a few dimensions. In practice, datasets are often not globally correlated. In such cases, reducing the dimensionality using PCA causes significant loss of distance information and hence degrades the query performance. Our goal in this thesis is to develop a dimensionality reduction technique that works well under all circumstances, even when the dataset is not globally correlated.

2.6 Time Series Indexing

Time series data is usually high dimensional in nature. For example, the ECG data in [78] has dimensionality between 256 and 1024. As discussed in Section 2.5, the most common technique to handle high dimensionality is to first reduce the dimensionality of the data and then index the reduced-dimensional data using a multidimensional index structure. Although PCA is the most common dimensionality reduction technique for other types of high dimensional data, other techniques like Discrete Fourier Transform (DFT), Discrete Wavelet Transform (DWT) and Piecewise Aggregate Approximation (PAA) are more common for time series data. We discuss these techniques in further detail in this section:

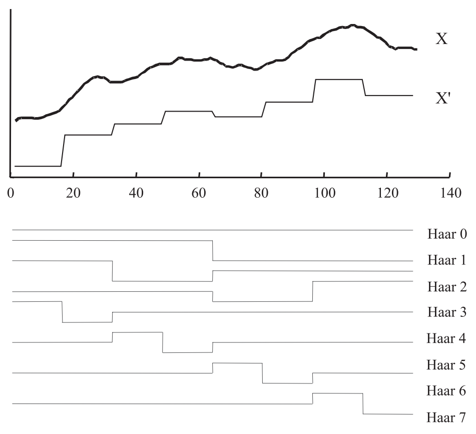


Figure 2.8: The first 8 Haar wavelet bases can be combined in a linear combination to produce X', an approximation of the original sequence X. There is one number per basis wavelet (the magnitude), so reduced dimensionality $N = 8$.

- *Discrete Fourier Transform (DFT):* The first technique suggested for dimensionality reduction of time series is DFT [5]. The basic idea of DFT is that any signal can be represented by the superposition of a finite number of sine (and/or cosine) waves, where each wave is represented by a single complex number known as a Fourier coefficient. A time series represented in this way is said to be in the frequency domain. There are many advantages to representing a time series in the frequency domain; the most important of which is data reduction. A signal of length n can be decomposed into n sine/cosine waves that can be recombined into the original signal. However, most of the later coefficients (the higher frequency ones) have very low amplitude and thus contribute little to the reconstructed signal; they can be discarded without much loss of information thereby producing data reduction.

  To perform dimensionality reduction of a time series X of length n into a reduced feature space of dimensionality N, the DFT of X is computed. The vector containing the first $\frac{N}{2}$ coefficients (lowest frequency ones) forms the reduced N-d representation of X. The reason the truncation takes place at $\frac{N}{2}$ and not at N is that each coefficient is a complex number, and therefore we need one dimension each for the imaginary and real parts of the coefficients. Figure 2.7 shows a signal X and its approximation X' computed from the retained $\frac{N}{2}$ (which is 4 in this case) coefficients.

  The key observation is that the Euclidean distance between two signals in the time domain is preserved in the frequency domain. This result is an implication of a well-known result called Parseval's law [46]. If some coefficients are discarded, then the estimate of the distance between two signals is guaranteed to be an underestimate, thus obeying the lower bounding requirement in Equation 2.1. Hence we can use DFT for indexing series data without compromising the exactness of the results. The original work demonstrated a speedup of 3 to 100 of such an index over sequential scanning [5, 46].

- *Discrete Wavelet Transform (DWT):* Wavelets are mathematical functions that represent data in terms

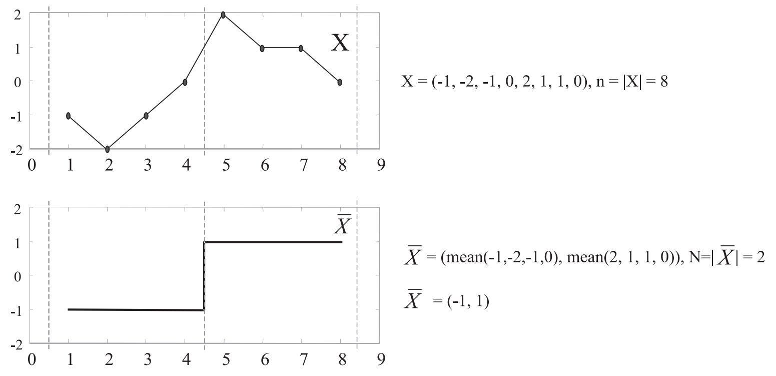


Figure 2.9: An illustration of the PAA technique. A time series consisting of eight (n) points is projected into two (N) dimensions.

of the sum and difference of a prototype function, called the basis function. In this sense they are similar to DFT. They differ in several important respects, however. One important difference is that wavelets are localized in time, i.e., each wavelet coefficient of a time series object contributes to the reconstruction of small portions of the object. This is in contrast to DFT where each Fourier coefficient contributes to the reconstruction of each and every datapoint of the time series. This property of DWT is useful for multiresolution analysis of the data. The first few coefficients contain an overall, coarse approximation of the data; addition coefficients can be imagined as ”zooming-in” to areas of high detail. Figure 2.8 illustrates this idea for Haar Wavelets.

To perform of a time series X of length n into a reduced feature space of dimensionality N, we compute the wavelet coefficients and retain the first N coefficients. Chan and Fu developed a distance measure defined on wavelet coefficients (Haar wavelets) which provably satisfies the lower bounding requirement in Equation 2.1 [29]; hence DWT can be used for indexing.

- *Piecewise Aggregate Approximation (PAA):* PAA reduces the dimensionality of a time series X of length (dimensionality) n to N ($1 \leq N \leq n$) by dividing X into N equal-length segments and recording the mean value of the datapoints falling within the segment [79, 153]. Figure 2.9 illustrates PAA. The distances in the PAA space lower bounds the distances in the original space, so PAA can be used for indexing [79]. It can be shown that PAA is identical to the wavelet technique proposed in [29] except that PAA is faster to compute and can support more general distance measures [153, 79].

All the above dimensionality reduction techniques choose a common representation for *all* the time series objects in the database; the first $\frac{N}{2}$ fourier coefficients in DFT, the first N wavelet coefficients in DWT and the low resolution version of the object in PAA where all parts of the object are represented at equal resolution. A technique that adapts the reduced-representation locally to each time series and chooses the best one for that item (i.e., the one with the lowest reconstruction error) can reduce dimensionality with significantly lower loss of information. Our goal in this thesis is to develop such a representation that can also be indexed using a multidimensional index structure (and support exact searching).

2.7 Access Method Integration: Concurrency Control in B-trees

Concurrent access to data via a general index structure introduces two independent concurrency control problems:

- *Preserving consistency of the data structure* in presence of concurrent insertions, deletions and updates
- *Protecting search regions from phantoms*

In this thesis, we address the problem of phantom protection in multidimensional AMs (in the context of GiSTs). Although this problem has received little attention in the context of multidimensional AMs, it has been addressed effectively for B-trees. We discuss the solution for B+-trees in this section.

The phantom problem is defined as follows: Transaction T1 reads a set of data items satisfying some `<search condition>`. Transaction T2 then creates data items that satisfies T1's `<search condition>` and commits. If T1 then repeats its scan with the same `<search condition>`, it gets a set of data items (known as "phantoms") different from the first read. Phantoms must be prevented to guarantee serializable execution. Note that object level locking [55] does not prevent phantoms since even if all objects currently in the database that satisfy the search predicate are locked, concurrent insertions[3] into the search range cannot be prevented.

One solution is for transactions to acquire locks on predicates (instead of objects). For example, a range scan that accesses all employees in the database whose salary is between 10K and 20K will acquire a shared mode (S) lock on the predicate: $10K \leq emp.salary \leq 20K$. A lock request $< t, p, m >$ by transaction t on predicate p with mode m conflicts with another request $< t', p', m' >$ iff *all* of the following are true: (1) t and t' are different (2) m and m' conflict and (3)$p \wedge p'$ is satisfiable (i.e. there may exist an object that satisfies both predicates). A transaction wishing to insert a new employee record whose salary is 11K will acquire an exclusive mode (X) lock on the predicate $emp.sal = 11K$ which conflicts with the predicate $10K \leq emp.salary \leq 20K$ associated with the range scan and will not be permitted. On the other hand, a transaction wishing to insert a new employee whose salary is 30K is permitted to execute concurrently with the scan.

While the predicate locking solves the problem of phantoms, unfortunately, testing for predicate satisfiability may be expensive. Even if predicates are simple and their satisfiability can be checked in constant time, the complexity of acquiring a predicate lock is linear in the number of concurrent transactions which is an order of magnitude costlier compared to acquiring object locks that can be set in constant time [55]. This problem is overcome using *granular locking* which is an engineering approach towards implementing predicate locks. The idea is to divide the predicate space into a set of resource granules that may include or overlap with other resource granules. Transactions acquire locks on granules instead of on predicates. The locking protocol guarantees that if two transactions request conflicting mode locks on predicates p and p'

[3]These insertions may be a result of insertion of new objects, updates to existing objects or rolling-back deletions made by other concurrent transactions.

such that $p \wedge p'$ is satisfiable, then the two transactions will request conflicting locks on at least one granule in common. Granular locks can be set and released as efficiently as object locks.

An example of a granular locking approach is the *multi-granularity locking protocol* (MGL) [89]. Besides preventing phantoms, MGL also has an added benefit that transactions can acquire locks on granules at different levels of coarseness based on their requirements– a lock on a node of the granule graph in mode M *implicitly* locks all the descendants of that node in mode M. To achieve this, MGL exploits additional lock modes called *intention* mode locks which represent the intention to set locks at finer granularity[89]. An intention mode lock on a node prevents other transactions from setting coarse granularity (i.e S or X) locks on that node. (see the lock compatibility matrix shown in Table 6.1). Transactions acquire locks from the root to the leaf of the granule graph and release locks in the reverse order. A transaction can acquire an S or IS mode lock at a granule g if it has at least *one* parent of g locked in either IS or higher mode. A transaction can acquire an X, SIX or IX mode lock at g if it has *all* parents of g locked in IX or SIX mode.

Application of MGL to the key space associated with a B-tree is referred to as *key range locking*. In key range locking, the entire key space is partitioned into certain key ranges which are supported as lockable granules. For example, if the domain of the key is the set of integers, the range may be divided into 4 distinct key ranges $(-\infty, 10], (10, 35], (35, 50], (50, \infty)$. A scan acquires locks to completely cover its query range. For example, a scan that accesses the keys between 5 to 30 will acquire S locks on the ranges $(-\infty, 10]$ and $(10, 35]$. Similarly, a transaction that inserts, deletes, or updates an object that lies in a given range, acquires an IX lock which denotes its intention to change an object in that range. For example, a transaction wishing to insert an object whose key value is 11 will acquire an IX lock on the range $(10, 35]$ which conflicts with the S lock held by the scan (see Table 6.1) and will therefore not be permitted concurrently with the scan thereby preventing phantoms.

The above discussion suggests that the set of key ranges supported as granules are static. In practice, an approach in which the key ranges that dynamically evolve as new key values are inserted and/or deleted from the database are preferred. Dynamic key range schemes are more adaptive to the changes in the key space over time and provides a higher degree of concurrency. However, since the granules may dynamically change, the locking protocols are significantly more complex. Further details about granular locking and key range locking can be found in [55]. In Chapter 6, we discuss in detail why the solution for B-trees cannot be applied for phantom protection in multidimensional AMs. We need new techniques for concurrency control in multidimensional AMs.

2.8 Approximate Query Answering Techniques

Approximate query processing has recently emerged as a viable, cost-effective solution for dealing with the huge data volumes and stringent response time requirements of today's Decision Support Systems (DSS) [1, 51, 53, 61, 64, 70, 115, 144, 145]. The general approach is to first construct compact synopses of the interesting relations in the database (using a data reduction technique) and then answering the user queries

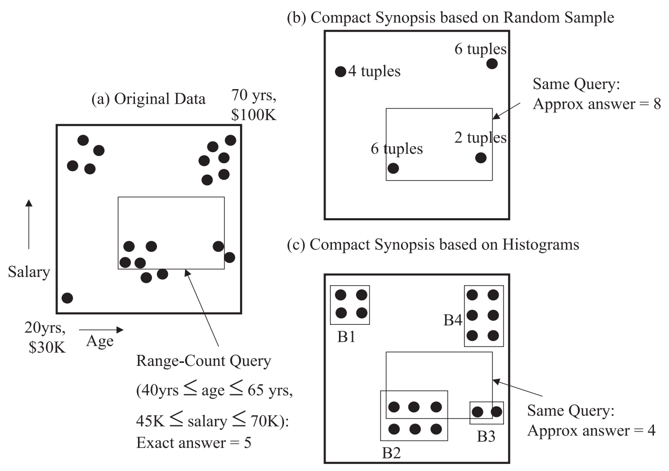


Figure 2.10: Data reduction techniques for approximate query answering.

by using just the synopsis. Data reduction techniques used for constructing the synopses include sampling, histograms and wavelets.

- *Sampling-based techniques* use random samples as synopses for large datasets. Figure 2.10(a) shows an example 2-dimensional DSS dataset where the location of each point represent the age and salary of a single individual. Figure 2.10(b) shows the synopsis of that dataset using random samples. Each sample point stores the location of the point and the number of tuples it represents. The ratio of the number of sample points to the size of the original dataset is the compression ratio (assuming that storing the count information has negligible cost compared to the point location which is true for high dimensional data). For example, in Figure 2.10, the compression ratio is 5. Consider the range count query shown in Figure 2.10, i.e., we want to know the number of individuals between 40 and 64 years making between 45K and 70K. The exact answer is 5 as shown in Figure 2.10(a). Figure 2.10(b) shows the computation of the approximate answer using the random sample synopsis. Since the synopsis is usually much smaller than the original data and usually resides in memory, the approximate answer can be computed much faster compared to the exact answer. Sample synopsis can be either precomputed (as shown in the example above) and maintain incrementally [1, 51] or can be obtained progressively at run-time by accessing the base data using appropriate access methods [61, 64]. Random samples typically provide accurate estimates for aggregate quantities (e.g., count, sum and average). Random samples can provide probabilistic guarantees on the quality of estimated aggregate [60]. Sampling techniques have several disadvantages, especially for non-aggregate queries and when join operations are involved, which is discussed in detail in Chapter 7.

- *Histogram-based techniques* use multidimensional histograms as synopses for large datasets. Figure 2.10(c) shows the synopsis of the same dataset using multidimensional histograms. Each histogram bucket stores a rectangle that specifies the coverage of the bucket, the number of points represented by that bucket and the number of unique positions along each dimension to capture the distribution of the points. For example, for bucket B4 in Figure 2.10(c), the count is 6 and the number of unique positions are 2 and 3 along the age and salary dimensions respectively. Approximate answers for range-aggregate queries are obtained by determining the overlap of the range with the buckets and then computing the aggregates based on the distribution of the points in the overlapping regions. Figure 2.10(c) shows the computation of the the approximate answer for the range-count query using the histogram synopsis. Once again, since the histogram synopsis is usually memory-resident, the approximate answer can be computed much faster compared to the exact answer. While histograms have been studied mostly in the context of selectivity estimation [52, 68, 69, 99, 116, 117], recently it has been proposed as an approximate query answering tool [70, 115]. Histogram techniques have several limitations, especially for high dimensional data, which is discussed in detail in Chapter 7.

- *Wavelet-based techniques* use wavelet coefficients as synopses. Recent work shows the wavelet-based synopses can produce surprisingly accurate results with very few retained coefficients, even at high dimensions [144, 145]. However, the work on wavelet-based approximate querying has so far been extremely limited in their query processing scope as discussed in Chapter 7.

Chapter 3

High Dimensional Index Structures: The Hybrid Tree

This chapter describes the hybrid tree, an index structure for high dimensional feature spaces.

3.1 Introduction

Feature based similarity search has emerged as an important search paradigm in database systems. The technique used is to map the data items as points into a high dimensional feature space. The feature space is indexed using a multidimensional data structure. Similarity search then corresponds to a range or k-NN search on that data structure. To support efficient similarity search in a database system, robust techniques to index high dimensional feature spaces needs to be developed. Traditional multidimensional data structures (e.g., R-trees [59], kDB-trees [120], grid files [105]), which were designed for indexing spatial data, are not suitable for multimedia feature indexing due to (1) inability to scale to high dimensionality and (2) lack of support for queries based on arbitrary distance measures. Recently, there has been significant research effort in developing indexing mechanisms suitable for multimedia feature spaces. One of the techniques is *dimensionality reduction* (DR). Existing DR techniques have several limitations: (1) they work well only when the data is strongly correlated (2) they usually do not support similarity queries based on arbitrary distance functions [13] and (3) they are not suitable for dynamic database environments. We address some of these limitations in Chapter 4. Since DR techniques are typically used in conjunction with multidimensional index structures (to index the reduced space) and the reduced spaces are still expected to be high dimensional in nature, a robust solution to feature indexing requires multidimensional data structures that scale to high dimensionalities and supports arbitrary distance measures.

This chapter introduces the hybrid tree for this purpose. What distinguishes the hybrid tree from other multidimensional data structures is that it is *neither a pure DP-based nor a pure SP-based technique*. Experience has shown that neither of these techniques are suitable for high dimensionalities but for different reasons. Simple sequential scan performs better beyond 10-15 dimensions [16]. BR-based techniques tend to have low fanout and a high degree of overlap between bounding regions (BRs) at high dimensions. On

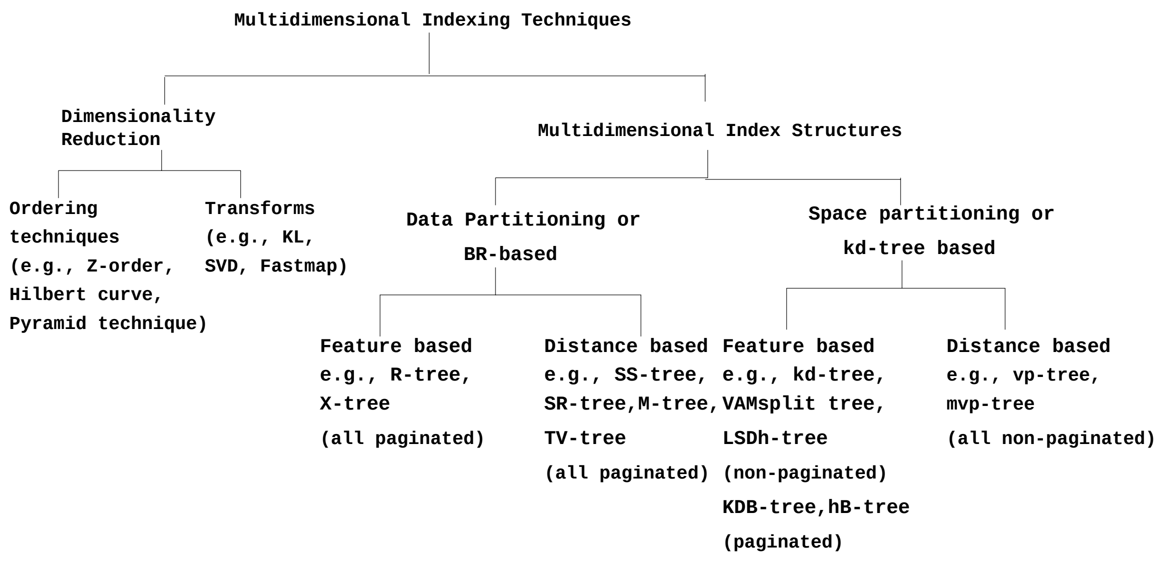

Multidimensional Indexing Techniques
Dimensionality Reduction
Multidimensional Index Structures
Ordering techniques (e.g., Z-order, Hilbert curve, Pyramid technique)
Transforms (e.g., KL, SVD, Fastmap)
Data Partitioning or BR-based
Space partitioning or kd-tree based
Feature based e.g., R-tree, X-tree (all paginated)
Distance based e.g., SS-tree, SR-tree, M-tree, TV-tree (all paginated)
Feature based e.g., kd-tree, VAMsplit tree, LSDh-tree (non-paginated) KDB-tree, hB-tree (paginated)
Distance based e.g., vp-tree, mvp-tree (all non-paginated)

Figure 3.1: Classification of Multidimensional Indexing Techniques

the other hand, SP-based techniques have fanout independent of dimensionality and no overlap between subspaces. But SP-based techniques suffer from problems like no guaranteed utilization (e.g., kDB-trees) or require storage of redundant information (e.g., hB-trees). The main contribution of this chapter is the *"hybrid"* approach to multidimensional indexing: a technique that combines positive aspects of the two types of index structures a single data structure to achieve search performance more scalable to high dimensionalities than either of the two techniques. On one hand, like SP-based index structures, the hybrid tree performs node splitting based on a single dimension and represents space partitioning using kd-trees. This makes the fanout independent of dimensionality and enables fast intranode search. On the other hand, space partitions, like the BRs in DP-based techniques, are allowed to overlap whenever clean splits necessitate downward cascading splits, thus retaining the guaranteed utilization property. The tree construction algorithms in the hybrid tree are geared towards providing optimal search performance. As desired, the hybrid tree allows search based on arbitrary distance functions. The distance function can be specified by the user at query time. Our experiments on "real" high dimensional large size feature databases show that the hybrid tree scales well to high dimensionality and large database sizes. It significantly outperforms both purely DP-based and SP-based index mechanisms as well as linear scan at all dimensionalities for large sized databases.

The rest of the chapter is organized as follows. Recently, many multidimensional data structures have been developed for the purpose of high dimensional feature indexing. In Section 3.2, we develop a classification of these data structures that allows us to compare them to the hybrid tree. Section 3.3 introduces the hybrid tree and is the main contribution of this chapter. In Section 3.4, we present the performance results. Section 3.5 offers the final concluding remarks.

Index Structure	Number of dimensions used to split	Number of (k-1)-d hyperplanes used to split	Number of kd-tree nodes used to represent the split	Fanout	Degree of Overlap	Node Utilization Guarantee	Storage Redundancy
KDB-tree	1	1	1	High (Independent of k)	None	No	None
hB-tree	d ($1 \leq d \leq k$)	d	d	High (Independent of k)	None	Yes	Yes
R-tree	k	2k	-	Low for large k ($\propto \frac{1}{k}$)	High	Yes	None
Hybrid tree	1	1 or 2	1	High (Independent of k)	Low	Yes	None

Table 3.1: Splitting strategies for various index structures. k is the total number of dimensions.

3.2 Classification of Multidimensional Index Structures

The increasing need of applications to be able to store multidimensional objects (e.g., features) in a database and index them based on their content has trigerred a lot of research on multidimensional index structures. In this section, we develop a classification of multidimensional indexing techniques which allows us to compare the hybrid tree with the previous research in this area. The classification is summarized in Figure 3.1. Since we have already discussed dimensionality reduction techniques, we restrict the discussion in this section to multidimensional index structures.

Existing multidimensional techniques can be classified in two different ways. One way to classify them is into **Data Partitioning (DP)-based and Space Partitioning (SP)-based** index structures. A DP-based index structure consists of bounding regions (BRs) arranged in a (spatial) containment hierarchy. At the data level, the nearby data items are clustered within BRs. At the higher levels, nearby BRs are recursively clustered within bigger BRs, thus forming a hierarchical directory structure. The BRs may overlap with each other. The BRs can be bounding boxes (e.g., R-tree[59], X-tree[15]) or bounding spheres/diamonds (e.g., SS-tree[149], M-tree[33], TV-tree[86]). On the other hand, a SP-based index structure consists of space recursively partitioned into mutually disjoint subspaces. The hierarchy of partitions form the tree structure (e.g., kDB-tree[120], hB-tree[90] and LSDh-tree[65]). We compare these two types of index structures with the hybrid tree as a solution to high dimensional feature indexing in Section 3.3.6.

An alternative way of classification is into **Feature-based and Distance based** techniques. In feature based techniques, the data/space partitioning is based on the values of the vectors along each independent dimension and is independent of the distance function used to compute the distance among objects in the database or between query objects and database objects. Examples of DP-based techniques that are feature based include R-tree and X-tree. Examples of SP-based techniques that are feature based include kDB-tree, hB-tree, LSDh-tree. On the other hand, distance based techniques partition data/space based on the distance of objects from one or more selected pivot point(s), where the distance is computed using a given distance function. Examples of DP-based techniques that are distance based include SS-tree, M-tree and TV-tree. Examples of SP-based techniques that are distance based include vp-tree [31] and mvp-tree [19]. A comparison between the two classes can be found in [22].

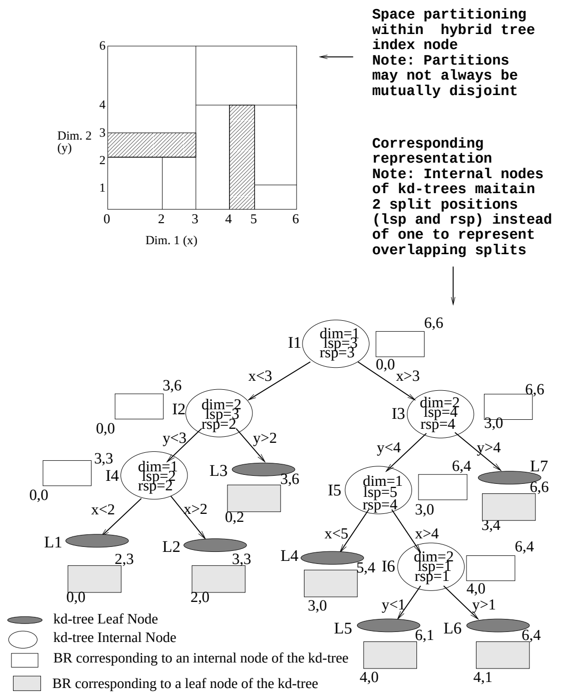

Space partitioning within hybrid tree index node
Note: Partitions may not always be mutually disjoint
Corresponding representation
Note: Internal nodes of kd-trees maitain 2 split positions (lsp and rsp) instead of one to represent overlapping splits
Dim. 2 (y)
Dim. 1 (x)
I1 dim=1 lsp=3 rsp=3
I2 dim=2 lsp=3 rsp=2
I3 dim=2 lsp=4 rsp=4
I4 dim=1 lsp=2 rsp=2
I5 dim=1 lsp=5 rsp=4
I6 dim=2 lsp=1 rsp=1
x<3
x>3
y<3
y>2
y<4
y>4
x<2
x>2
x<5
x>4
y<1
y>1
L1
L2
L3
L4
L5
L6
L7
kd-tree Leaf Node
kd-tree Internal Node
BR corresponding to an internal node of the kd-tree
BR corresponding to a leaf node of the kd-tree

Figure 3.2: Mapping between each node and the corresponding BR. The shaded area represents overlap between BRs

3.3 The Hybrid Tree

In this section, we introduce the hybrid tree. We discuss how the hybrid tree partitions the space into subspaces and how the space partitioning is represented in the hybrid tree. We discuss the node splitting algorithms and show how they optimize expected search performance. We describe the tree operations and conclude with a discussion on where the hybrid tree fits into the classification developed in Section 2.

3.3.1 Space Partitioning in the Hybrid Tree

First, we describe the “space partitioning strategy” in the hybrid tree i.e. how to partition the space into two subspaces when a node splits. The first issue is the number of dimensions used to partition the node. The hybrid tree always splits a node using a *single* dimension. 1-d split is the *only* way to guarantee that the fanout is totally independent of dimensionality. This is in sharp contrast with DP-based techniques which are at the other extreme: they use all the k dimensions to split, leading to a linear decrease in fanout with increase in dimensionality. Some index structures follow intermediate policies [90]. The only disk-based index structure that follows a 1-d split policy is the kDB-tree [120]. Single dimension splits in the kDB-tree necessitate costly cascading splits and causes creation of empty nodes. Due to the above reasons, kDB-tree shows poor performance even in 4 dimensional feature spaces [56]. kDB-trees cause cascading splits since it requires the node splits to be necessarily *clean* i.e. the split *must* divide the indexed space into two mutually disjoint partitions. We relax the above constraint in the hybrid tree: the indexed subspaces need *not* be mutually disjoint. The overlap is allowed only when trying to achieve an overlap-free would cause downward cascading splits and hence a possible violation of utilization constraints. The splitting strategies of the various index structures is summarized in the Table 3.1.

It is clear from the above discussion that the hybrid tree is more similar to SP-based data structures than DP-based index structures. But the above "relaxation" necessitates several changes in terms of representation and algorithms for tree operations as compared to the pure SP-based index structures. The first change is in the representation. As in other SP-based techniques, the space partitioning within each index node in a hybrid tree is represented using a kd-tree. Since regular kd-trees can represent only overlap free splits, we need to modify the kd-tree in order to represent possibly overlapping splits. Each internal node of the regular kd-tree represents a split by storing the split dimension and the split position. We add a second split position field to the kd-tree internal node. The first split position represents the right (higher side) boundary of the left (lower side) partition (denoted by lsp or left side partition) while the second split position represents the left boundary of the right partition (denoted by rsp or right side partition). While $lsp = rsp$ means non-overlapping partitions, $lsp > rsp$ indicate overlapping partitions. The second change is in the algorithms for regular tree operations, namely, search, insertion and deletion. The tree operations in SP-based index structures are based on the assumption that the partitions are mutually disjoint. This is not true for the hybrid tree. We solve the problem by treating the indexed subspaces as BRs in a DP-based data structure (which can overlap). In other words, we define a *mapping* the kd-tree based representation to an "array of BRs" representation. This allows us to directly apply the search, insertion and deletion algorithms used in DP-based data structures to the hybrid tree. The mapping is defined recursively as follows: *Given any index node N of the hybrid tree and the BR R_N corresponding to it, we define the BRs corresponding to each child of N.* The BR of the root node of the hybrid tree is the entire data space. Given that, the above "mapping" can compute the BR of any hybrid tree node.

Let N be an index node of the hybrid tree. Let K_N be the kd-tree that represents the space partitioning within N and R_N be the BR of N. We define a BR associated with each node (both internal as well as leaf nodes) of K_N. This defines the BRs of the children of N since the leaf nodes of K_N are the children of N. For example, the leaf nodes $L1$ to $L7$ are the children of the hybrid tree node N shown in the Figure 3.2. The BR associated with the root of K_N is R_N. Now given an internal node I of K_N and the corresponding BR R_I, the BRs of the two children of I are defined as follows. Let $I = \langle dim, lsp, rsp \rangle$, where dim, lsp and rsp are the split dimension, left split position and right split position respectively. The BR of the left child of I is defined as $R_I \cap (dim \leq lsp)$ where, in the expression $(dim \leq lsp)$, dim denotes the variable that represents the value along dimension dim (for simplicity) and $\cap$ represents geometric intersection. Similarly, the BR of the right child of I is defined as $R_I \cap (dim \geq rsp)$. For example, $(0, 0, 6, 6)$ is the BR for the hybrid tree node shown in Figure 3.2 (BR is denoted as $x_{lo}, y_{lo}, x_{hi}, y_{hi}$). The BR of I1 (the root) is $(0, 0, 6, 6)$. The BRs of I2 and I3 are $(0, 0, 6, 6) \cap (x \leq 3) = (0, 0, 3, 6)$ and $(0, 0, 6, 6) \cap (x \geq 3) = (3, 0, 6, 6)$ respectively. Similarly, the BR of L3, which, being a leaf of K_N, is a child of N, is obtained by $BR(I2) \cap (y \geq 2)$ i.e. $(0, 0, 3, 6) \cap (y \geq 2) = (0, 2, 3, 6)$. The children of internal nodes with $lsp > rsp$ have overlapping BRs

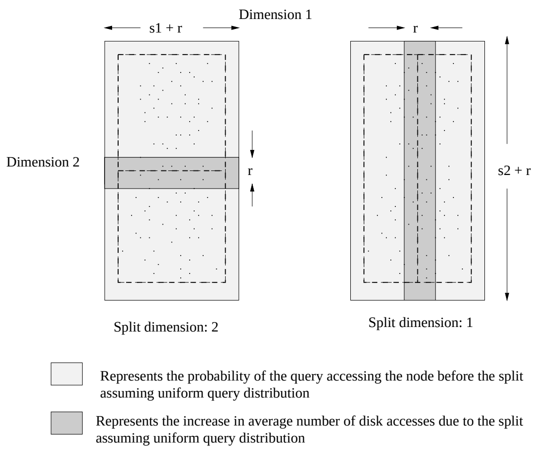

Dimension 1
s1 + r
r
Dimension 2
r
s2 + r
Split dimension: 2
Split dimension: 1
Represents the probability of the query accessing the node before the split assuming uniform query distribution
Represents the increase in average number of disk accesses due to the split assuming uniform query distribution

Figure 3.3: Choice of split dimension for data nodes. The first split is the optimal choice in terms for search performance.

(e.g., BRs of I4 and L3 (children of I2) overlap). Figure 3.2 shows all the BRs – the shaded rectangles are the BRs of the children of the node while the white ones correspond to the internal nodes of K_N.

Note that the above mapping is "logical". The search/insert/delete algorithm does not actually compute the "array of BRs" during tree traversal: rather it navigates the node using the kd-tree and computes the BR only when necessary (cf. Section 3.3.4). The kd-tree based navigation allows faster intranode search compared to array-based navigation. While searching for a correct lower level node using a kd-tree usually requires order log n comparisons (for a balanced kd-tree), searching in a array requires linear number of comparisons. Also, in a kd-tree representation, BRs share boundaries. In an array representation, the boundaries are checked redundantly while in a kd-tree, a boundary is checked only once [90].

3.3.2 Data Node Splitting

The choice of a split of a node consists of two parts: the choice of the split dimension and the split position(s). In this section, we discuss the choice of splits for data nodes in the hybrid tree.

Choice of split dimension: When a data node splits, it is replaced by two nodes. Assuming that the rest of the tree has not changed, the expected number of disk accesses per query (EDA) would increase due to the split. The hybrid tree chooses as the split dimension the one that minimizes the increase in EDA due to the split, thereby optimizing the expected search performance for future queries.

Let N be the data node being split. Let R be the k-dimensional BR associated with N. Let s_i be the extent of R along the ith dimension, $i = [1, k]$. Consider a bounding box range query Q with each side of length r. We assume that the feature space is normalized (extent is from 0 to 1 along each dimension) and the queries are uniformly distributed in the data space. Let $P_{overlap(Q,R)}$ denote the probability that Q overlaps with R. To determine $P_{overlap(Q,R)}$, we move the center point of the query to each point of the data space marking the positions where the query rectangle intersects the BR. The resulting set of marked positions is called the Minkowski Sum which is the original BR having all sides extended by query side length r [12]. Therefore, $P_{overlap(Q,R)} = (s_1 + r)(s_2 + r)...(s_k + r)$. This is the probability that Q needs to access node N (1 disk access) (It is the volume of lightly shaded region in Figure 3.3).

Now let us consider the splitting of N and let j be the splitting dimension. Let $N1$ and $N2$ be the nodes after the split and $R1$ and $R2$ be the corresponding BRs. $R1$ and $R2$ have the same extent as R along all dimensions except j i.e. s_i, $i = [1, k], i \neq j$. Let αs_j and βs_j be the extents of $R1$ and $R2$ along the jth dimension. Since the split is overlap-free, $\beta = 1 - \alpha$. The probabilities $P_{overlap(Q,R1)}$ and $P_{overlap(Q,R2)}$ are $(s_1+r)...(\alpha s_j+r)...(s_k+r)$ and $(s_1+r)...((1-\alpha)s_j+r)...(s_k+r)$ respectively. Since $R = R1 \cup R2$ (where $\cup$ is the geometric union) and Q is uniformly distributed, $P_{overlap(Q,R)} = P_{overlap(Q,R1\cup R2)} = P_{overlap(Q,R1)\cup overlap(Q,R2)}$. Thus, the probability $P_{overlap(Q,R1)\cap overlap(Q,R2)}$ that both $N1$ and $N2$ are accessed is equal to $P_{overlap(Q,R1)} + P_{overlap(Q,R2)} - P_{overlap(Q,R)}$. ($P_{overlap(Q,R1)\cap overlap(Q,R2)}$ is equal to the volume of the dark shaded region in Figure 3.3). If Q does not overlap with R, there is no increase in number of disk accesses due to the split. If it does, $P_{overlap(Q,R1)\cap overlap(Q,R2)}$ is the probability that the disk accesses increases by 1 due to the split. Thus, the conditional probability that Q overlaps with both $R1$ and $R2$ given Q overlaps with R, i.e. $\frac{P_{overlap(Q,R1)\cap overlap(Q,R2)}}{P_{overlap(Q,R)}}$ represents the increase in EDA due to the split. The increase in EDA if j is chosen as the split dimension evaluates out to be $\frac{r}{s_j+r}$. Note that $\frac{r}{s_j+r}$ is minimum if j is chosen such that $s_j = max_{i=1}^{k} s_i$, independent of the value of r. The hybrid tree always chooses the dimension along with the BR has the largest extent as the split dimension for splitting data nodes so as to minimize the increase in EDA due to the split.

An example of the choice of split dimension is shown in Figure 3.3. Note that the optimality of the above choice is independent of the distribution of data. It is also independent of the choice of split position. Previous proposals regarding choice of splitting dimensions include arbitrary/round-robin [65] and maximum variance dimension [150]. The maximum variance dimension is chosen to make the choice insensitive to "outliers" [150]. Since the number of disk accesses to be made depends on the size of the subspaces indexed by data nodes and is independent of the actual distribution of data items within the subspace, presence or absence of "outliers" is inconsequential to the query performance. We performed experiments to compare our choice of maximum extent dimension as the splitting dimension with the maximum variance choice and is discussed is Section 5.

Choice of split position: The most common choice of the split position for data node splitting is the median [120, 90, 150]. The median choice, in general, distributes the data items equally among the two nodes (assuming unique median). The hybrid tree, however, chooses the split position as close to the middle as possible. [1] This tends to produce more cubic BRs and hence ones with smaller surface areas. The smaller the surface area, the lower the probability that a range query overlaps with that BR, the lower the number of expected number of disk accesses [14]. Our experiments validate the above observation.

[1]To find the position, we first check whether it is possible to split in the middle without violating utilization constraint. If yes, it is chosen. Otherwise the split position is shifted from the middle position in the proper direction just enough to satisfy the utilization requirement.

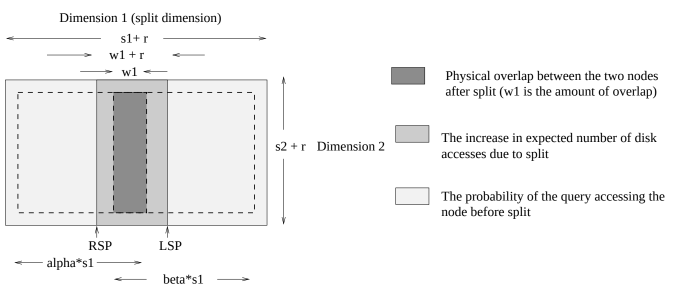

Dimension 1 (split dimension)
s1+ r
w1 + r
w1
s2 + r Dimension 2
RSP
LSP
alpha*s1
beta*s1
Physical overlap between the two nodes after split (w1 is the amount of overlap)
The increase in expected number of disk accesses due to split
The probability of the query accessing the node before split

Figure 3.4: Index node splitting (with overlap). s_j, w_j and split positions (LSP and RSP) only along dimension 1 are shown.

3.3.3 Index Node Splitting

In this section, we discuss the choice of split dimension and split position for index nodes.

Choice of the split dimension: Like data node splitting, the choice of split dimension for index nodes splitting is also based on minimization of the increase in EDA. However, unlike data node splitting where the choice is independent of the query size, the choice of the split dimension for index nodes depends on the probability distribution of the query size as discussed below.

The main difference here compared to data node splitting is splits are not always overlap free. Let w_j ($w_j \leq s_j$) be the amount of overlap between $R1$ and $R2$ along the jth dimension (how w_j is computed is discussed in the following paragraph on choice of split position). So $\alpha s_j + \beta s_j = s_j + w_j$. An example of an index node split is shown in Figure 3.4. The probabilities $P_{overlap(Q,R1)}$ and $P_{overlap(Q,R2)}$ are $(s_1 + r)...(\alpha s_j + r)...(s_k + r)$ and $(s_1 + r)...(\beta s_j + r)...(s_k + r)$ respectively. Proceeding in the same way as before, the increase in EDA if j is chosen as the split dimension evaluates out to be$\frac{w_j+r}{s_j+r}$. The choice of j that minimizes the above quantity optimizes search performance. But the choice depends on r and can differ for different values of r. For a given probability distribution of r, the hybrid tree chooses the dimension that minimizes the increase in EDA averaged over all queries. Let $P(r)$ be probability distribution of r. The increase in EDA averaged over all queries is equal to$\int_R^{R+\Delta R} P(r).\frac{w_j+r}{s_j+r}dr$ where r can vary from R to $R+\Delta R$. The dimension that minimizes the above quantity is chosen as the split dimension. For example, for uniform distribution, where $P(r) = \frac{1}{\Delta R}$, the above integral evaluates to be $\left(1 - (\frac{s_j-w_j}{\Delta R})log(1 + \frac{\Delta R}{s_j+R})\right)$. In this case, the hybrid tree chooses that j for which $(s_j-w_j)log(1+\frac{\Delta R}{s_j+R})$ is maximum. In our experiments, we use all queries of the same size, say R. In this case, the dimension j that minimizes $\frac{w_j+R}{s_j+R}$ should be chosen as the split dimension which is indeed the case since $lim_{\Delta R \to 0}\left(1 - (\frac{s_j-w_j}{\Delta R})log(1 + \frac{\Delta R}{s_j+R})\right) = \frac{w_j+R}{s_j+R}$.

Choice of split position: Given the split dimension, the split positions are chosen such that the overlap is minimized without violating the utilization requirement. The problem of determining the best split positions along a given dimension is a 1-d version of the R-tree bipartitioning problem. In the latter, the problem is to equally divide the rectangles into two groups to reduce the total area covered by the bounding boxes. while in the former, the problem is to divide the line segments (indexed subspaces of the children projected along

Property of index structure	BR-based index structures	kd-tree based index structures	Hybrid Tree
Representation of space partitioning	Array of bounding boxes	kd-tree	kd-tree (modified to represent overlapping partitions)
Indexed subspaces	May mutually overlap	Strictly disjoint	May mutually overlap
Node splitting	Using all dimensions	Using 1 or more dimensions	Using 1 dimension
Dead space † elimination	Yes	No	Yes (with live space encoding)

Table 3.2: Comparison of the hybrid tree with the BR-based and kd-tree based index structures. † Dead space refers to portions of feature space containing no data items (cf. Section 4.2).

the split dimension) into two groups in a way to minimize the the overlap along the split dimension without violating the utilization constraint. We sort the line segments based on both their left (leftmost to rightmost) and right (rightmost to leftmost) boundaries. Then we choose new segments alternately from the left and right sorted lists and place them in left and right partitions respectively till the utilization is achieved. The remaining line segments are put in the partition that needs least elongation without caring about utilization. The above bipartitioning algorithm is similar to the R-tree quadratic algorithm but runs in $O(nlogn)$ time instead of $O(n^2)$ (where n is the number of children nodes) since 1-d intervals can be sorted based on their values (left and right boundaries) along the split dimension.

Before the split dimension is actually chosen, the best split positions are determined for all the dimensions. Then the w_j's and s_j's are calculated for each dimension and the one with the lowest $\int_R^{R+\Delta R} P(r) . \frac{w_j+r}{s_j+r} dr$ is selected. After the selection of the split dimension, the split positions for the selected dimension determined during the pre-selection phase are used as split positions.

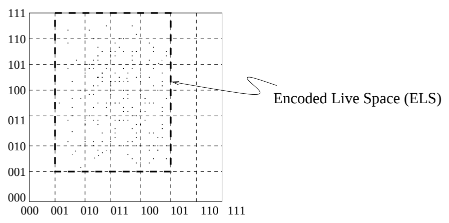
Implicit Dimensionality Reduction:

We conclude the subsection on index node splitting with the following observation. The hybrid tree *implicitly* eliminates "non-discriminating" dimensions i.e. those dimensions along which the feature vectors are not much different from each other. In other words, these dimensions are never used for node splitting. This is true for data node splitting due to the "maximum extent" choice. To ensure that these dimensions are indeed eliminated, we must guarantee that an eliminated dimension is never chosen for splitting the index node. Let N be an index node. Let $\mathcal{D}_N$ be the set of dimensions used for partitioning space within N. We can provide the above guarantee if the the split dimension d_N of N satisfies $d_N \in \mathcal{D}_N$, The reason is that a dimension not used to split any data node cannot be in $\mathcal{D}_N$. Suppose we restrict our choice of the split dimension of N to $\mathcal{D}_N$ instead of all dimensions. We show that even then we would make the EDA-optimal choice.

Lemma 1 (Implicit Dimensionality Reduction) *It is possible to make the EDA-optimal choice even when restricting the choice of the split dimension of node N to $\mathcal{D}_N$.*

Proof:

The EDA-optimal choice of the split dimension of N is the one with the lowest $\frac{r+w_j}{r+s_j}$ ratio. We need to show that the above ratio for any dimension $j \in \mathcal{D}_N$ is less than or equal to the ratio for every dimension



Live space encoding using 3 bit precision (ELSPRECISION=3)
Encoded Live Space BR = (001, 001, 101, 111)
Bit required: 2*number_of_dimensions*ELSPRECISION=12 bits

Figure 3.5: Encoded Live Space (ELS) Optimization

$i \notin \mathcal{D}_N$. For any dimension $j \in \mathcal{D}_N$, $w_j \leq s_j$. So for any $j \in \mathcal{D}_N$ and for any value of r, $\frac{r+w_j}{r+s_j} \leq 1$. For any dimension $i \notin \mathcal{D}_N$, $w_i = s_j$, hence $\frac{r+w_j}{r+s_j} = 1$ for all r (worst case). Hence the proof. ■

The hybrid tree achieves implicit dimension elimination through the above choice. This effect is not seen in most paginated multidimensional data structures. For example, DP-based techniques, all dimensions are used for indexing - so nothing is eliminated. SP-based techniques which choose the split dimension arbitrarily/round robin fashion cannot provide the above guarantee.

3.3.4 Dead Space Elimination

The hybrid tree, like other SP techniques, indexes dead space i.e. space the contains no data objects. DP-techniques, on other other hand, does not. Dead space indexing cause unnecessary disk accesses. This effect increases at higher dimensionality. Storage of the live space BRs would reduce the hybrid tree into a DP-based technique, making the fanout of the node sensitive to dimensionality. Instead, we encode the live space BR *relative* to the entire BR (defined by kd-tree partitioning) using a few bits as suggested in [65]. The live space encoding is explained in Figure 3.5. More the number of bits used, the higher the precision of the representation, lower the number of unnecessary disk accesses. We observed that using as few as 4 bits per dimension eliminates most dead space. For 8K page, 4 bit precision and 64-d space, the overhead is less than 1% of the database size and can be stored in memory. The overhead is even less for lower dimensionality. During search (say range search), the overlap check is performed in 2 steps: first, the BR defined by kd-tree is checked and if they overlap, the live space BR is decoded and checked, thus saving any unnecessary decoding/checking costs. We performed experiments to demonstrate the effect of ELS optimization in the hybrid tree as discussed in Section 5.

3.3.5 Tree Operations

The hybrid tree, like other disk based index structures (e.g., B-tree, R-tree) is completely dynamic i.e. insertions, deletions and updates can occur interspersed with search queries without requiring any reorganization. The tree operations in the hybrid tree are similar to the R-trees i.e. indexed subspaces are treated as BRs but the kd-tree based organization is exploited to achieve faster intranode search. In addition to point and bounding-box queries (i.e. feature-based queries), the hybrid tree supports distance-based queries: both range and nearest neighbor queries. Unlike several index structures (e.g., distance-based index structures like SS-tree, M-tree), the hybrid tree, being a feature-based technique, can support queries with arbitrary distance measures. This is important advantage since the distance function can vary from query to query for the same feature or even between several iterations of the same query in a relevance feedback environment [71, 124].

The insertion and deletion operations in the hybrid tree is also similar to that in R-trees. The insertion algorithm recursively picks the child node in which the new object should be inserted. The best candidate is the node that needs the minimum enlargement to accommodate the new object. Ties are broken based on the size of the BR. The deletion operation is based on the eliminate-and-reinsert policy as in [59].

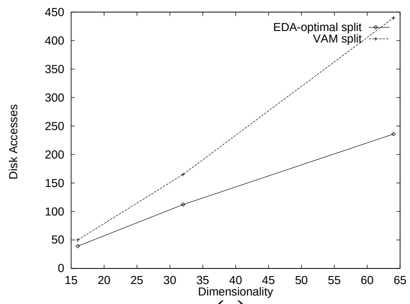
3.3.6 Summary

It is clear from the above discussion that the hybrid tree resembles both DP and SP techniques in some aspects and differs from them in others: rather it is a "hybrid" of the two approaches. The comparison of the hybrid tree with the two techniques is shown in Table 3.2. Now we summarize the reasons why hybrid tree is more suitable for high dimensional indexing either DP or SP techniques. It is more suitable than than pure DP techniques since (1) its fanout is independent of dimensionality while DP-techniques have low fanout at high dimensionalities (2) enables faster intranode search by organizing the space partitioning as a kd-tree instead of an array and (3) eliminates overlap from the lowest level (since data node splits are always mutually non-overlapping) and reduces overlap at higher levels by using EDA-optimal 1-d splits instead of k-d splits as in DP techniques. The hybrid tree performs better than other SP-based techniques using 1-d splits (e.g., KDB-trees) since unlike the latter, it provides (1) guaranteed storage utilization (2) avoids costly cascading splits and (3) chooses EDA-optimal split dimensions instead of arbitrarily. It performs better than SP-based techniques using multiple dimensional splits (e.g., hB-trees) since (1) 1-d splits usually provide better search performance compared to multiple dimensional ones since the latter tends to produce subspaces with larger surface area and hence more disk accesses [14] and (2) it does not require storage of redundant information (e.g., posting full paths).

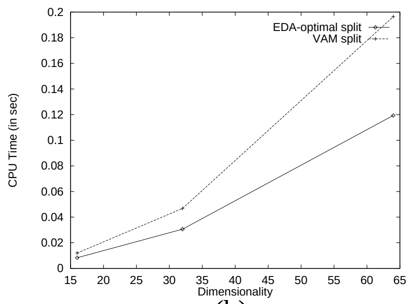
3.4 Experimental Evaluation

We performed extensive experimentation to (1) evaluate the various design decisions made in the hybrid tree and (2) compare the hybrid tree with other competitive techniques. We conducted our experiments over the following two "real world" datasets:

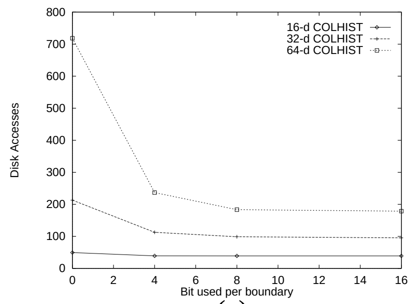
(1) The **FOURIER** dataset contains 1.2 million 16-d vectors produced by fourier transformation of polygons. We construct 8-d, 12-d and 16-d vectors by taking the first 8, 12 and 16 fourier coefficients respectively.



(a)



(b)



(c)

Figure 3.6: (a) and (b) shows the effect of EDA Optimization on query performance. (c) shows the effect of ELS Optimization on query performance. Both experiments were performed on 64-d COLHIST data.

(2) The **COLHIST** dataset comprises of color histograms extracted from about 70,000 color images obtained from the Corel Database. We generate 16, 32 and 64 dimensional vectors by extracting 4x4, 8x4 and 8x8 color histograms [110] from the images.

The queries are randomly distributed in the data space with appropriately chosen ranges to get constant selectivity. In all experiments discussed below, the selectivity is maintained constant at 0.07 % for FOURIER and 0.2 % for COLHIST. All the experiments were conducted on a Sun Ultra Enterprise 3000 with 512MB of physical memory and several GB of secondary storage. In all our experiments, we use a page size of 4096 bytes.

We performed experiments to evaluate (1) the impact of EDA-optimal node splitting algorithms and (2) the effect of live space optimization in the hybrid tree. Both the experiments were performed on the 64-d COLHIST data. The performance is measured by (1) the average number of disk accesses required to execute a query and (2) the average CPU time required to execute a query. Figure 3.6(a) and (b) show the performance of the hybrid tree constructed using EDA-optimal node splitting algorithms compared to the hybrid tree constructed using the VAM-split node splitting algorithm [150]. The EDA-optimal split algorithms consistently outperforms the VAMSplit algorithm. The performance gap increases with the increase in dimensionality. Figure 3.6(c) shows the effect of live space optimization. Using 4-bit ELS improves the performance significantly compared to no ELS but using more bits does not improve it much further.

We conducted experiments to compare the performance of the hybrid tree with the following competitive techniques: (1) SR-tree [77] (2) hB-tree [90] (3) Sequential Scan. We chose SR-tree since it is one of the most competitive BR-based data structures proposed for high dimensional indexing. Similarly, hB-tree is among the best known SP-based techniques for high dimensionalities. We normalize the I/O cost and the CPU cost of each of the 3 indexing techniques against the cost of linear scan. We define the normalized costs as follows:

- *The Normalized I/O cost*: the ratio of the average number of disk accesses required to execute a query using the indexing technique to the number of disk accesses to execute a linear scan. The

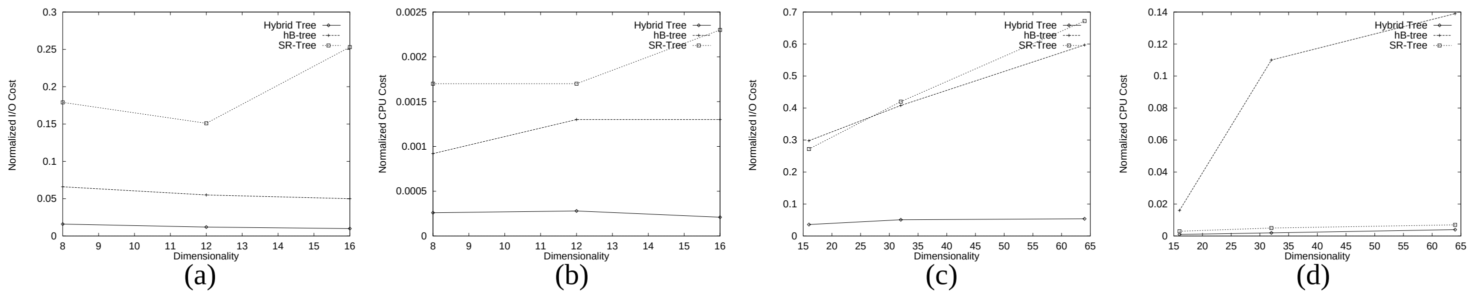


Figure 3.7: Scalability to dimensionality. (a) and (b) shows the query performance (I/O and CPU costs) for medium dimensional data (FOURIER dataset(400K points)). (c) and (d) shows the same for high dimensional data (COLHIST dataset(70K points))

latter is computed by $\frac{DatabaseSize}{PageSize}$ i.e. $\frac{NumberOfObjects*Dimensionality*sizeof(float)}{PageSize}$. Note that since sequential disk accesses are about 10 times faster compared to random accesses, the normalized I/O cost of linear scan is 0.1 instead of 1.0. Hence, for any index mechanism, a normalized I/O cost of more than 0.1 indicate worse I/O performance compared to linear scan.

- *The Normalized CPU cost*: the ratio of average CPU time required to execute a query using the index mechanism to the average CPU time required to perform a linear scan. The normalized CPU cost of linear scan is 1.0.

Using normalized costs instead of direct costs (1) allows us to compare each of the techniques against linear scan as the latter is widely recognized as a competitive search technique in high dimensional feature spaces [16] while still comparing them to each other and (2) makes the measurements independent of the experimental settings (e.g., H/W platform, pagesize).

Figures 3.7 shows the scalability of the various techniques to medium dimensional and high dimensional feature spaces respectively. The hybrid tree performs significantly better than any other technique including linear scan. The hB-tree performs better compared to SR-tree since SP-based techniques are more suited for high dimensional indexing than BR-techniques as argued in [146]. The fast intranode search in the hybrid tree due to its kd-tree based organization account for the faster CPU times.

Figures 3.8(a) and (b) compares the different techniques in terms of their scalability to very large databases. The hybrid tree significantly outperforms all other techniques by more than an order of magnitude for all database sizes. The hybrid tree shows a decreasing normalized cost with increase in database size indicating sublinear growth of the actual cost with database size. Figures 3.8(c) and (d) compares the query performance of various techniques[2] for distance-based queries. As suggested in [110], we use the L1 metric. Again, the hybrid tree outperforms the other techniques.

From the experiments, we can conclude that the hybrid tree scales well to high dimensional feature

[2]hB-tree is not used since it does not support distance-based search.

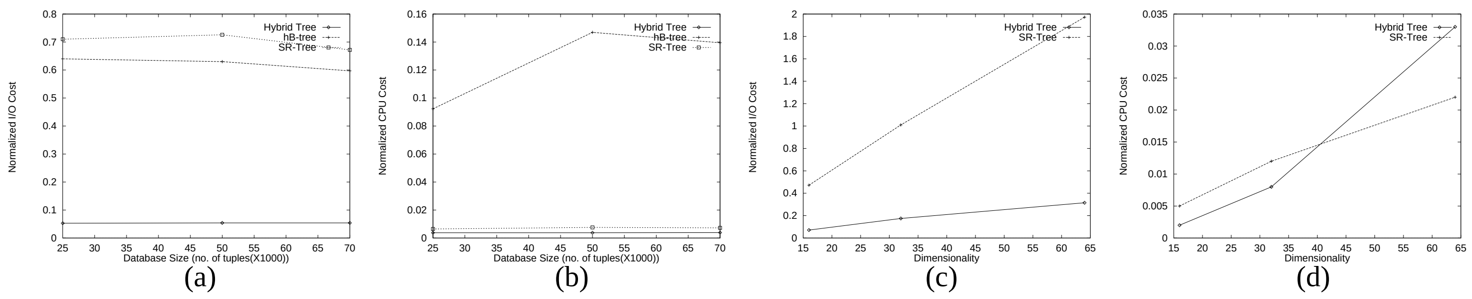


Figure 3.8: (a) and (b) compares the scalability of the various techniques with database size of high dimensional data. (c) and (d) compares the query performance of the various techniques for distance-based queries (Manhattan Distance). Both experiments were performed on 64-d COLHIST data.

spaces, large database sizes and efficiently supports arbitrary distance measures.

3.5 Conclusion

Feature based similarity search is emerging as an important search paradigm in database systems. Efficient support of similarity search requires robust feature indexing techniques. In this chapter, we introduce the hybrid tree - a multidimensional data structure for indexing high dimensional feature spaces. The hybrid tree combines positive aspects of bounding region based and space partitioning based data structures into a single data structure to achieve better scalability. It supports queries based on arbitrary distance functions. Our experiments show that the hybrid tree is scalable to high dimensional feature spaces and provides efficient support of distance based retrieval. The hybrid tree is a fully operational software and is currently being deployed for feature indexing in MARS [111].

In the next chapter, we introduce the Local Dimensionality Reduction (LDR) technique in order to enhance the scalability of the hybrid tree even further. LDR reduces the dimensionality of data by exploiting local correlations in data. We describe how the reduced data can be indexed using the hybrid tree (or any other multidimensional index structure). We show that LDR used in conjunction with the hybrid tree provides a very scalable solution to the problem of high dimensional indexing.

Chapter 4

Local Dimensionality Reduction for High Dimensional Indexing

We present the local dimensionality reduction (LDR) technique in this chapter. We show that LDR used in conjunction with the hybrid tree proposed in the previous chapter provides a very scalable solution to the problem of high dimensional indexing.

4.1 Introduction

While designing *high dimensional index trees* like the hybrid tree is a big step towards providing efficient access over high dimensional feature spaces (HDFS), it must be used in conjunction with a *dimensionality reduction* technique in order to exploit the correlations in data and hence achieve further scalability. This approach is commonly used in both multimedia retrieval ([43, 103, 76, 142]) and data mining ([47, 8, 49]) applications. The idea is to first reduce the dimensionality of the data and then index the reduced space using a multidimensional index structure [43]. Most of the information in the dataset is condensed to a few dimensions (the first few principal components (PCs)) by using principal component analysis (PCA). The PCs can be arbitrarily oriented with respect to the original axes [48]. The remaining dimensions (i.e. the later components) are eliminated and the index is built on the reduced space. To answer queries, the query is first mapped to the reduced space and then executed on the index structure. Since the distance in the reduced-dimensional space lower bounds the distance in the original space, the query processing algorithm can guarantee no false dismissals [43]. The answer set returned can have false positives (i.e. false admissions) which are eliminated before it is returned to the user. We refer to this technique as *global dimensionality reduction* (GDR) i.e. dimensionality reduction over the *entire* dataset taken together.

GDR works well when the dataset is *globally correlated* i.e. most of the variation in the data can be captured by a few orthonormal dimensions (the first few PCs). Such a case is illustrated in Figure 4.1(a) where a single dimension (the first PC) captures the variation of data in the 2-d space. In such cases, it is possible to eliminate most of the dimensions (the later PCs) with little or no loss of distance information. However, in practice, the dataset may not be globally correlated (see Figure 4.1(b)). In such cases, reducing

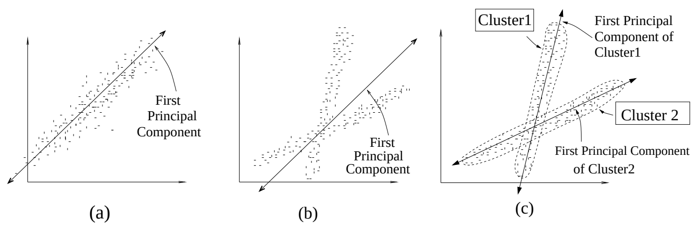


Figure 4.1: Global and Local Dimensionality Reduction Techniques (a) GDR(from 2-d to 1-d) on globally correlated data (b) GDR (from 2-d to 1-d) on globally non-correlated (but locally correlated) data (c) LDR (from 2-d to 1-d) on the same data as in (b)

the data dimensionality using GDR will cause a significant loss of distance information. Loss in distance information is manifested by a large number of false positives and is measured by precision [76] (cf. Section 4.5). More the loss, larger the number of false positives, lower the precision. False positives increase the cost of the query by (1) causing the query to make unnecessary accesses to nodes of the index structure and (2) adding to the post-processing cost of the query, that of checking the objects returned by the index and eliminating the false positives. The cost increases with the increase in the number of false positives. Note that false positives do not affect the quality the answers as they are not returned to the user.

Even when a global correlation does not exist, there may exist subsets of data that are *locally correlated* (e.g., the data in Figure 4.1(b) is not globally correlated but is locally correlated as shown in Figure 4.1(c)). Obviously, the correlation structure (the PCs) differ from one subset to another as otherwise they would be globally correlated. We refer to these subsets as *correlated clusters* or simply *clusters*.[1] In such cases, GDR would not be able to obtain a single reduced space of desired dimensionality for the entire dataset without significant loss of query accuracy. If we perform dimensionality reduction on each cluster *individually* (assuming we can find the clusters) rather than on the entire dataset, we can obtain a set of different reduced spaces of desired dimensionality (as shown in Figure 4.1(c)) which together cover the entire dataset[2] but achieves it with minimal loss of query precision and hence significantly lower query cost. We refer to this approach as local dimensionality reduction (LDR).

Contributions: In this chapter, we propose LDR as an approach to high dimensional indexing. Our contributions can be summarized as follows:

- We develop an algorithm to discover correlated clusters in the dataset. Like any clustering problem, the problem, in general, is NP-Hard. Hence, our algorithm is heuristic-based. Our algorithm performs dimensionality reduction of each cluster individually to obtain the reduced space (referred to as subspace) for each cluster. The data items that do not belong to any cluster are outputted as outliers. The algorithm allows the user to control the amount of information loss incurred by dimensionality reduction and hence the query precision/cost.

[1]Note that correlated clusters (formally defined in Section 4.3) differ from the usual definition of clusters i.e. a set of spatially close points. To avoid confusion, we refer to the latter as *spatial clusters* in this chapter.

[2]The set of reduced spaces may not necessarily cover the entire dataset as there may be outliers. We account for outliers in our algorithm.

- We present a technique to index the subspaces individually. We present query processing algorithms for point, range and k-nearest neighbor (k-NN) queries that execute on the index structure. Unlike many previous techniques [76, 142], our algorithms guarantee correctness of the result i.e. returns exactly the same answers as if the query executed on the original space. In other words, the answer set returned to the user has no false positives or false negatives.
- We perform extensive experiments on synthetic as well as real-life datasets to evaluate the effectiveness of LDR as an indexing technique and compare it with other techniques, namely, GDR, index structure on the original HDFS (referred to as the original space indexing (OSI) technique) and linear scan. Our experiments show that (1) LDR can reduce dimensionality with significantly lower loss in query precision as compared to GDR technique. For the same reduced dimensionality, LDR outperforms GDR by almost an order of magnitude in terms of precision. and (2) LDR performs significantly better than other techniques, namely GDR, original space indexing and sequential scan, in terms of query cost for both synthetic and real-life datasets.

Roadmap: The rest of the chapter is organized as follows. In Section 4.2, we provide an overview of related work. In Section 4.3, we present the algorithm to discover the correlated clusters in the data. Section 4.4 discusses techniques to index the subspaces and support similarity queries on top of the index structure. In Section 4.5, we present the performance results. Section 4.6 offers the final concluding remarks.

4.2 Related Work

Previous work on high dimensional indexing techniques includes development of high dimensional index structures (e.g., X-tree[15], SR-tree [77], TV-tree [86], Hybrid-tree [23]) and global dimensionality reduction techniques [48, 43, 47, 76]. The techniques proposed in this chapter build on the above work. Our work is also related to the clustering algorithms that have been developed recently for database mining (e.g., BIRCH, CLARANS, CURE algorithms) [154, 102, 58]. The algorithms most related to this chapter are those that discover patterns in low dimensional subspaces [2, 3]. In [2], Agarwal et. al. present an algorithm, called CLIQUE, to discover“dense” regions in all subspaces of the original data space. The algorithm works from lower to higher dimensionality subspaces: it starts by discovering 1-d dense units and iteratively discovers all dense units in each k-d subspace by building from the dense units in (k-1)-d subspaces. In [3], Aggarwal et. al. present an algorithm, called PROCLUS, that clusters the data based on their correlation i.e. partitions the data into disjoint groups of correlated points. The authors use the hill climbing technique, popular in spatial cluster analysis, to determine the projected clusters. Neither CLIQUE, nor PROCLUS can be used as an LDR technique since they cannot discover clusters when the principal components are arbitrarily oriented. They can discover only those clusters that are correlated along one or more of the original dimensions. The above techniques are meant for discovering interesting patterns in the data; since correlation along arbitrarily oriented components is usually not that interesting to the user, they do not attempt to discover such correlation. On the contrary, the goal of LDR is efficient indexing; it must be able to discover

Symbols	Definitions
N	Number of objects in the database
M	Maximum number of clusters desired
K	Actual number of clusters found ($K \leq M$)
D	Dimensionality of the original feature space
S_i	The ith cluster
C_i	Centroid of S_i
n_i	Size of S_i (number of objects)
$\mathcal{A}_i$	Set of points in S_i
Φ_i	The principal components of S_i
$\Phi_i^{(j)}$	The jth principal component of S_i
d_i	Subspace dimensionality of S_i
ϵ	Neighborhood range
$MaxReconDist$	Maximum Reconstruction distance
$FracOutliers$	Permissible fraction of outliers
$MinSize$	Minimum Size of a cluster
$MaxDim$	Maximum subspace dimensionality of a cluster
$\mathcal{O}$	Set of outliers

Table 4.1: Summary of symbols and definitions

such correlation in order to minimize the loss of information and make indexing efficient. Also, since the motivation of their work is pattern discovery and not indexing, they do not address the indexing and query processing issues which we have addressed in this thesis. To the best of our knowledge, this is the first work that proposes to exploit the local correlations in data for the purpose of indexing.

4.3 Identifying Correlated Clusters

In this section, we formally define the notion of correlated clusters and present an algorithm to discover such clusters in the data.

4.3.1 Definitions

In developing the algorithm to identify the correlated clusters, we will need the following definitions.

Definition 1 (Cluster and Subspace) Given a set $\mathcal{A}$ of N points in a D-dimensional feature space, we define a *cluster* S as a set $\mathcal{A}_S$ ($\mathcal{A}_S \subseteq \mathcal{A}$) of locally correlated points. Each cluster S is defined by $S = \langle \Phi_S, d_S, C_S, \mathcal{A}_S \rangle$ where:

- Φ_S are the principal components of the cluster, $\Phi_S^{(i)}$ denoting the ith principal component.
- d_S is the reduced dimensionality i.e. the number of dimensions retained. Obviously, the retained dimensions correspond to the first d_S principal components $\Phi_S^{(i)}, 1 \leq i \leq d_S$ while the eliminated dimensions correspond to the next $(D - d_S)$ components. Hence we use the terms (principal) components and dimensions interchangeably in the context of the transformed space.

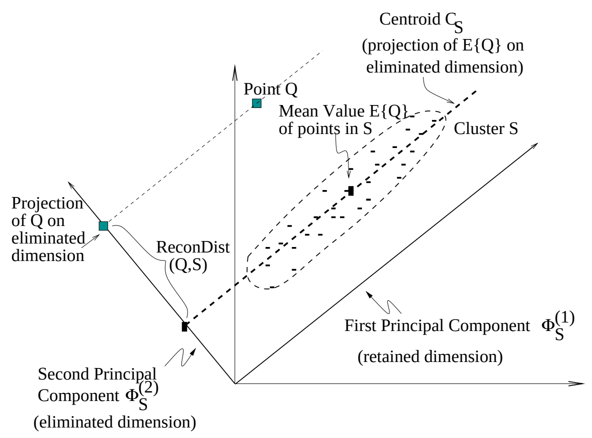

Centroid C_S
(projection of E{Q} on eliminated dimension)
Point Q
Mean Value E{Q} of points in S
Cluster S
Projection of Q on eliminated dimension
ReconDist (Q,S)
First Principal Component Φ_S^(1)
(retained dimension)
Second Principal Component Φ_S^(2)
(eliminated dimension)

Figure 4.2: Centroid and Reconstruction Distance.

- $C_S = [C_S^{(d_S+1)} \cdots C_S^{(D)}]$ is the centroid, that stores, for each eliminated dimension $\Phi_i, (d_S+1) \leq i \leq D$, a single constant which is "representative" of the position of every point in the cluster along this unrepresented dimension (as we are not storing their unique positions along these dimensions).
- $\mathcal{A}_S$ is the set of points in the cluster

The reduced dimensionality space defined by $\Phi_S^{(i)}, 1 \leq i \leq d_S$ is called the *subspace* of S. d_S is called the subspace dimensionality of S.

■

Definition 2 (Reconstruction Vector) Given a cluster $S = \langle \Phi_S, d_S, C_S, \mathcal{A}_S \rangle$, we define the *reconstruction vector* $\overline{ReconVect}(Q, S)$ of a point Q from S as follows:

$$\overline{ReconVect}(Q,S) = \bar{\Sigma}_{i=(d_S+1)}^{D}(Q \bullet \Phi_S^{(i)} - C_S^{(i)})\Phi_S^{(i)} \tag{4.1}$$

where $\bar{\Sigma}$ denotes vector addition and $\bullet$ denotes scalar product (i.e. $Q \bullet \Phi_S^{(i)}$ is the projection of Q on $\Phi_S^{(i)}$ as shown in Figure 4.2). $(Q \bullet \Phi_S^{(i)} - C_S^{(i)})$ is the (scalar) distance of Q from the centroid along each eliminated dimension and $\overline{ReconVector}(Q, S)$ is the vector of these distances.

■

Definition 3 (Reconstruction Distance) Given a cluster $S = \langle \Phi_S, d_S, C_S, \mathcal{A}_S \rangle$, we now define the *reconstruction distance* (scalar) $ReconDist(Q, S, \mathcal{D})$ of a point Q from S. $\mathcal{D}$ is the distance function used to define the similarity between points in the HDFS. Let $\mathcal{D}$ be an L_p metric i.e. $\mathcal{D}(P, P') = \parallel P - P' \parallel_p =$

$[\Sigma_{i=1}^{d}(|P[i] - P'[i]|)^{p}]^{1/p}$. We define $ReconDist(Q, S, \mathcal{D})$ [3] as follows:

$$\begin{aligned} ReconDist(Q, S, \mathcal{D}) &= ReconDist(Q, S, L_p) && (4.2)\\ &= \| \overline{ReconVect}(Q, S) \|_p && (4.3)\\ &= [\Sigma_{i=d_S+1}^{D}(|Q \bullet \Phi_S^{(i)} - C_S^{(i)}|)^{p}]^{1/p} && (4.4) \end{aligned}$$

■

Note that for any point Q mapped to the d_S-dimensional subspace of S, $\overline{ReconVect}(Q, S)$ (and $ReconDist(Q, S)$) represent the error in the representation i.e. the vector (and scalar) distance between the exact D-dimensional representation of Q and its approximate representation in the d_S-dimensional subspace of S. Higher the error, more the amount of distance information lost.

4.3.2 Constraints on Correlated Clusters

Our objective in defining clusters is to identify low dimensional subspaces, one for each cluster, that can be indexed separately. We desire each subspace to have as low dimensionality as possible without losing too much distance information. In order to achieve the desired goal, each cluster must satisfy the following constraints:

1. **Reconstruction Distance Bound:** In order to restrict the maximum representation error of any point in the low dimensional subspace, we enforce the reconstruction distance of any point $P \in \mathcal{A}_S$ to satisfy the following condition: $ReconDist(P, S) \leq MaxReconDist$ where $MaxReconDist$ is a parameter specified by the user. This condition restricts the amount of information lost within each cluster and hence guarantees a high precision which in turn implies lower query cost.

2. **Dimensionality Bound:** For efficient indexing, we want the subspace dimensionality to be as low as possible while still maintaining high query precision. A cluster must not retain any more dimensions that necessary. In other words, it must retain the minimum number of dimensions required to accommodate the points in the dataset. Note than a cluster S can accommodate a point P only if $ReconDist(P, S) \leq MaxReconDist$. To ensure that the subspace dimensionality d_S is below the critical dimensionality of the multidimensional index structure (i.e. the dimensionality above which a sequential scan is better), we enforce the following condition: $d_S \leq MaxDim$ where $MaxDim$ is specified by the user.

3. **Choice of Centroid:** For each cluster S, we use PCA to determine the subspace i.e. Φ_S is the set of eigenvectors of the covariance matrix of $\mathcal{A}_S$ sorted based on their eigenvalues. [48] shows that for a given choice of reduced dimensionality d_S, the representation error is minimized by choosing the

[3]Assuming that $\mathcal{D}$ is a fixed L_p metric, we usually omit the $\mathcal{D}$ in $ReconDist(Q, S, \mathcal{D})$ for simplicity of notation.

first d_S components among Φ_S and choosing C_S to be the mean value of the points (i.e. the centroid) projected on the eliminated dimensions. To minimize the information loss, we choose $C_S^{(i)} = E\{P \bullet \Phi_S^{(i)}\} = E\{P\} \bullet \Phi_S^{(i)}$ (see Figure 4.2).

4. **Size Bound:** Finally, we desire each cluster to have a minimum cardinality (number of points) : $n_S \geq MinSize$ where $MinSize$ is user-specified. The clusters that are too small are considered to be outliers.

The goal of the LDR algorithm described below is to discover the set $\mathcal{S} = S_1, S_2, ..., S_K$ of K clusters (where $K \leq M$, M being the maximum number of clusters desired) that exists in the data and that satisfy the above constraints. The remaining points, that do not belong to any of the clusters, are placed in the outlier set $\mathcal{O}$.

4.3.3 The Clustering Algorithm

Since the LDR algorithm needs to perform *local* correlation analysis (i.e. PCA on subsets of points in the dataset rather than the whole dataset), we need to first identify the right subsets to perform the analysis on. This poses a cyclic problem: how do we identify the right subsets without doing the correlation analysis and how do we do the analysis without knowing the subsets. We break the cycle by using *spatial clusters* as an initial guess of the right subsets. Then we perform PCA on each spatial cluster individually. Finally, we 'recluster' the points based on the correlation information (i.e. principal components) to obtain the correlated clusters. The clustering algorithm is shown in Table 4.2. It takes a set of points $\mathcal{A}$ and a set of clusters $\mathcal{S}$ as input. When it is invoked for the first time, $\mathcal{A}$ is the entire dataset and each cluster in $\mathcal{S}$ is marked 'empty'. At the end, each identified cluster is marked 'complete' indicating a completely constructed cluster (no further change); the remaining clusters remain marked 'empty'. The points that do not belong to any of the clusters are placed to the outlier set $\mathcal{O}$. The details of each step is described below:

- **Construct Spatial Clusters**(Steps FC1 and FC2): The algorithm starts by constructing M spatial clusters where M is the maximum number of clusters desired. We use a simple single-pass partitioning-based spatial clustering algorithm to determine the spatial clusters [102]. We first choose a set of $\mathcal{C} \subset \mathcal{A}$ of *well-scattered* points as the centroids such that points that belong to the same spatial cluster are not chosen to serve as centroids to different clusters. Such a set $\mathcal{C}$ is called a *piercing* set [3]. We achieve this by ensuring that each point $P \in \mathcal{C}$ in the set is sufficiently far from any already chosen point $P' \in \mathcal{C}$ i.e. $Dist(P, P') > threshold$ for a user-defined threshold. [4] This technique, proposed by Gonzalez [54], is guaranteed to return a piercing if no outliers are present. To avoid scanning though the whole database to choose the centroids, we first construct a random sample of the dataset and choose the centroids from the sample [3, 58]. We choose the sample to be large

[4]For subsequent invocations of FindClusters procedure during the iterative algorithm (Step 2 in Table 4.3), there may exist already completed clusters (does not exist during the initial invocation). Hence P must also be sufficiently far from all complete clusters formed so far i.e. $ReconDist(P, S) > threshold$ for each complete cluster S.

```
Clustering Algorithm
Input: Set of Points $\mathcal{A}$, Set of clusters $\mathcal{S}$ (each cluster is either empty or complete)
Output: Some empty clusters are completed, the remaining points form the set of outliers $\mathcal{O}$
FindClusters($\mathcal{A}, \mathcal{S}, \mathcal{O}$)
FC1: For each empty cluster, select a random point $P \in \mathcal{A}$ such that $P$ is sufficiently far from all completed and
     valid clusters. If found, make $P$ the centroid $C_i$ and mark $S_i$ valid.

FC2: For each point $P \in \mathcal{A}$, add $P$ to the closest valid cluster $S_i$ (i.e. $i = argmin(Distance(P, C_i))$) if $P$ lies in
     the $\epsilon$-neighborhood of $C_i$ i.e. $Distance(P, C_i) \leq \epsilon$.

FC3: For each valid cluster $S_i$, compute the principal components $\Phi_i$ using PCA. Remove all points from $\mathcal{A}_i$.

FC4: For each point $P \in \mathcal{A}$, find the valid cluster $S_i$ that, among all the valid clusters requires the minimum
     subspace dimensionality $LD(P)$ to satisfy $ReconDist(P, S_i) \leq MaxReconDist$ (break ties arbitrarily). If
     $LD(P) \leq MaxDim$, increment $V_i[j]$ for $j = 0$ to $(LD(P) - 1)$ and $n_i$.

FC5: For each valid cluster $S_i$, compute the subspace dimensionality $d_i$ as: $d_i = \{j | F_i[j] \leq$
     $FracOutliers$ and $F_i[j-1] > FracOutliers\}$ where $F_i[j] = \frac{V_i[j]}{n_i}$.

FC6: For each point $P \in \mathcal{A}$, add $P$ to the first valid cluster $S_i$ such that $ReconDist(P, S_i) \leq MaxReconDist$. If
     no such $S_i$ exists, add P to $\mathcal{O}$.

FC7: If a valid cluster $S_i$ violates the size constraint i.e. $(|\mathcal{A}_i| < MinSize)$, mark it empty. Remove each point $P \in$
     $\mathcal{A}_i$ from $S_i$ and add it to the first succeeding cluster $S_j$ that satisfies $ReconDist(P, S_j) \leq MaxReconDist$
     or to $\mathcal{O}$ if there is no such cluster. Mark the other valid clusters complete. For each complete cluster $S_i$, map
     each point $P \in \mathcal{A}_i$ to the subspace and store it along with $ReconDist(P, S, \mathcal{D})$.
```

Table 4.2: Clustering Algorithm

enough (using Chernoff bounds [98]) such that the probability of missing clusters due to sampling is low i.e. there is at least one point from each cluster present in the sample with a high probability [58]. Once the centroids are chosen, we group each point $P \in \mathcal{A}$ with the closest centroid $C_{closest}$ if $Distance(P, C_{closest}) \leq \epsilon$ and update the centroid to reflect the mean position of its group. If $Distance(P, C_{closest}) > \epsilon$, we ignore P. The restriction of the neighborhood range to ϵ makes the correlation analysis *localized*. Smaller the value of ϵ, the more localized the analysis. At the same time, ϵ has to be large enough so that we get a sufficiently large number of points in the cluster which is necessary for the correlation analysis to be robust.

- **Compute PCs**(Step FC3): Once we have the spatial clusters, we perform PCA on each spatial cluster S_i individually to obtain the principal components $\Phi_S^{(i)}, i = [1, D]$. We do not eliminate any components yet. We compute the mean value M_i of the points in S_i so that we can compute $ReconDist(P, S_i)$ in Steps FC4 and FC5 for any choice of subspace dimensionality d_i. Finally, we remove the points from the spatial clusters so that they can be reclustered as described in Step FC6.

- **Determine Subspace Dimensionality**(Steps FC4 and FC5): For each cluster S_i, we must retain no more dimensions than necessary to accommodate the points in the dataset (except the outliers). To determine the number of dimensions d_i to be retained for each cluster S_i, we first determine, for

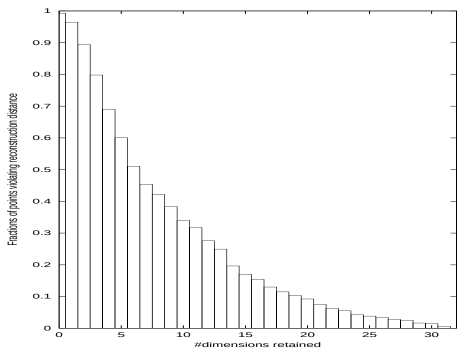


Figure 4.3: Determining subspace dimensionality (MaxDim=32).

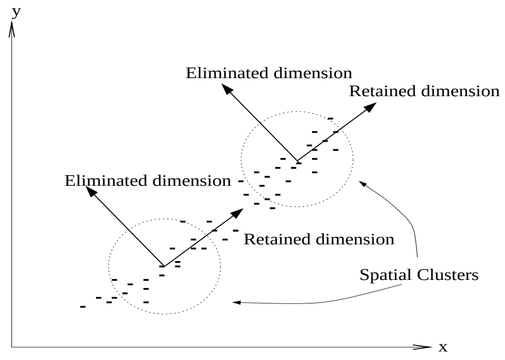


Figure 4.4: Splitting of correlated clusters due to initial spatial clustering.

each point $P \in \mathcal{A}$, the best cluster, if one exists, for placing P. Let $LD(P, S_i)$ denote the the least dimensionality needed for the cluster S_i to represent P with $ReconDist(P, S_i) \leq MaxReconDist$. Formally,

$$LD(P, S_i) = \{d | ReconDist(P, S_i) \leq MaxReconDist \text{ if } d_i \geq d$$
$$\text{and } ReconDist(P, S_i) > MaxReconDist \text{ otherwise } \} \quad (4.5)$$

In other words, the first $LD(P, S_i)$ PCs are just enough to satisfy the above constraint. Note that such a $LD(P, S_i)$ always exists for a non-negative $MaxReconDist$. Let $LD(P) = min$ { $LD(P, S_i) | S_i$ is a valid cluster }. If $LD(P) \leq MaxDim$, there exists a cluster that can accommodate P without violating the dimensionality bound. Let $LD(P, S_i) = LD(P)$ (if there are multiple such clusters S_i, break ties arbitrarily). We say S_i is the "best" cluster for placing P since S_i is the cluster that, among all the valid clusters, needs to retain the minimum number of dimensions to accommodate P. P would satisfy the $ReconDist(P, S_i) \leq MaxReconDist$ bound if the subspace dimensionality d_i of S_i is such that $LD(P, S_i) \leq d_i \leq MaxDim$ and would violate it if $0 \leq d_i < LD(P, S_i)$. For each cluster S_i, we maintain this information as a count array $V_i[j], j = [0, MaxDim]$ where $V_i[j]$ is the number of points that, among the points chosen to be placed in S_i, would violate the $ReconDist(P, S_i) \leq MaxReconDist$ constraint if the subspace dimensionality d_i is j: so in this case (for point P), we must increment $V_i[j]$ for $j = 0$ to $(LD(P, S_i) - 1)$ and the total count n_i of points chosen to be placed in S_i. ($V_i[j]$ and n_i is initialized to 0 before FC4 begins). On the other hand, if $LD(P) > MaxDim$, there exists no cluster in which P can be placed without violating the dimensionality bound; so we do nothing.

At the end of the pass over the dataset, for each cluster S_i, we have computed $V_i[j], j = [0, MaxDim]$ and n_i. We use this to compute $F_i[j], j = [0, MaxDim]$ where $F_i[j]$ is the fraction of points that, among those chosen to be placed in S_i (during FC4), would violate the $ReconDist(P, S_i) \leq MaxReconDist$ constraint if the subspace dimensionality d_i is j i.e. $F_i[j] = \frac{V_i[j]}{n_i}$. An example of F_i

from one of the experiments conducted on the real life dataset (cf. Section 4.5.3) is shown in Figure 4.3. We choose d_i to be as low as possible without too many points violating the reconstruction distance bound i.e. not more than $FracOutliers$ fraction of points in S_i where $FracOutliers$ is specified by the user. In other words, d_i is the minimum number of dimensions that must be retained so that the fraction of points that violate the $ReconDist(P, S_i) \leq MaxReconDist$ constraint is no more that $FracOutliers$ i.e. $d_i = \{j | F_i[j] \leq FracOutliers$ and $F_i[j-1] > FracOutliers\}$. In Figure 4.3, d_i is 21 for $FracOutliers = 0.1$, 16 for $FracOutliers = 0.2$ and 14 for $FracOutliers = 0.3$. We now have all the subspaces formed. In the next step, we assign the points to the clusters.

- **Recluster Points**(Step FC6): In the reclustering step, we reassign each point $P \in \mathcal{A}$ to a cluster S that covers P i.e. $ReconDist(P, S) \leq MaxReconDist$. If there exists no such cluster, P is added to the outlier set $\mathcal{O}$. If there exists just one cluster that covers P, P is assigned to that cluster. Now we consider the interesting case of multiple clusters covering P. In this case, there is a possibility that some of these clusters are actually parts of the same correlated cluster but has been split due to the initial spatial clustering. This is illustrated in Figure 4.4. Since points in a correlated cluster can be spatially distant from each other (e.g., form an elongated cluster in Figure 4.4) and spatial clustering only clusters spatially close points, it may end up putting correlated points in different spatial clusters, thus breaking up a single correlated cluster into two or more clusters. Although such ‘splitting’ does not affect the indexing cost of our technique for range queries and k-NN queries, it increases the cost of point search and deletion as multiple clusters may need to searched in contrast to just one when there is no ‘splitting’. (cf. Section 4.4.2). Hence, we must detect these ‘broken’ clusters and merge them back together. We achieve this by maintaining the clusters in some fixed order (e.g., order in which they were created). For each point $P \in \mathcal{P}$, we check each cluster sequentially in that order and assign it to the first cluster that covers P. If two (or more) clusters are part of the same correlated cluster, most points will be covered by all of them but will *always* be assigned to only one them, whichever appears first in the order. This effectively merges the clusters into one since only the first one will remain while the others will end up being almost empty and will be discarded due to the violation of size bound in FC7. Note that the $FracOutliers$ bound in Step FC5 still holds i.e. besides the points for which $LD(P) > MaxDim$, no more that $FracOutliers$ fraction of points can become outliers.

- **Map Points**(Step FC7): In the final step of the algorithm, we eliminate clusters that violate the size constraint. We remove each point from these clusters and add it to the first succeeding valid cluster S_j that satisfies the $ReconDist(P, S_j) \leq MaxReconDist$ bound or to $\mathcal{O}$ otherwise. For the remaining clusters S_i, we map each point $P \in \mathcal{A}_i$ to the subspace by projecting P to $\Phi_i^{(j)}, 1 \leq j \leq d_i$ and refer it as the (d_i-d) image $Image(P, S_i)$ of P:

$$Image(P, S_i)[j] = P \bullet \Phi_i^{(j)} \text{ for } 1 \leq j \leq d_i \tag{4.6}$$

We refer to P as the (D-d) original $Original(Image(P, S_i), S_i)$ of its image $Image(P, S_i)$. We store the image of each point along with the reconstruction distance $ReconDist(P, S_i)$.

Since FindClusters chooses the initial centroids from a random sample, there is a risk of missing out some clusters. One way to reduce this risk is to choose a large number of initial centroids but at the cost of slowing down the clustering algorithm. We reduce the risk of missing clusters by trying to discover more clusters, if there exists, among the points returned as outliers by the initial invocation of FindClusters. We iterate the above process as long as new clusters are still being discovered as shown below:

Iterative Clustering	
(1)	FindClusters($\mathcal{A}$, $\mathcal{S}$, $\mathcal{O}$); /* initial invocation */
(2)	Let $\mathcal{O}'$ be an empty set. Invoke FindClusters($\mathcal{O}$, $\mathcal{S}$, $\mathcal{O}'$). Make $\mathcal{O}'$ the new outlier set i.e. $\mathcal{O} \leftarrow \mathcal{O}'$. If new clusters found, go to (2). Else return.

Table 4.3: Iterative Clustering Algorithm

The above iterative clustering algorithm is somewhat similar to the hill climbing technique, commonly used in spatial clustering algorithms (especially in partitioning-based clustering algorithms like k-means, k-medoids and CLARANS [102]). In this technique, the "bad quality" clusters (the ones that violate the size bound) are discarded (Step FC7) and is replaced, if possible, by better quality clusters. However, unlike the hill climbing approach where all the points are reassigned to the clusters, we do not reassign the points already assigned to the 'complete' clusters. Alternatively, we can follow the hill climbing approach but it is computationally more expensive and requires more scans of the database [102].

Cost Analysis: We conclude this section with a analysis of the cost of the clustering algorithm. Let us first analyze the cost of the first invocation of the FindClusters procedure (where $\mathcal{A}$ is the whole dataset). The centroid selection step (FC1) has a small cost since we are using a random sample and $|sample| \ll |\mathcal{A}|$. Step FC2 requires one pass through the dataset $\mathcal{A}$ and has a time complexity of $O(NKD)$. Step FC3 has a complexity of $O(n_i D^2)$ for each cluster S_i and hence an overall complexity of $O(ND^2)$ (since $\Sigma_i n_i \leq N$). This step also has a memory requirement of $O(n_i D)$ for each cluster and hence a maximum of $O(max_i(n_i)D)$ which is smaller than the memory requirement of $O(ND)$ of GDR. This is an advantage of LDR over GDR: while the latter requires the whole dataset to fit in memory, the former requires only the points in the cluster to fit in memory. In either case, if the memory is too small, we can perform SVD on a sample rather than the whole data [76]. Step FC4 requires another pass through the database and has a time complexity of $O(ND^2K)$ (assuming $MaxDim$ is a constant). Step FC5 is a simple step with a complexity of $O(KD)$. Step FC6 requires a final pass through the database and has a time complexity of $O(ND^2K)$. Also, the first invocation of FindClusters accounts for most of the cost of the algorithm since the later invocations have much smaller sets as input and hence much smaller cost. Thus, the algorithm requires three passes through the dataset (FC2,FC4 and FC6) and a time complexity of $O(ND^2K)$.

4.4 Indexing Correlated Clusters

Having developed the technique to find the correlated clusters, we now shift our attention to how to use them for indexing. Our objective is to develop a data structure that exploits the correlated clusters to efficiently support range and k-NN queries over HDFSs. The developed data structure must also be able to handle insertions and deletions.

4.4.1 Data Structure

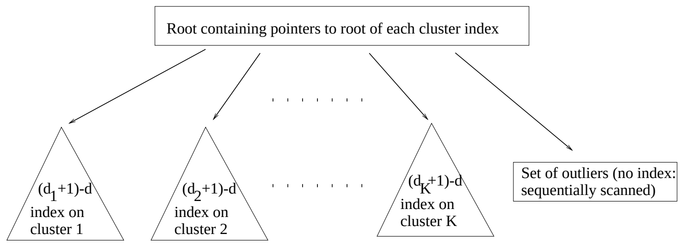
The data structure, referred to as the global index structure (GI) (i.e. index on entire dataset), consists of separate multidimensional indices for each cluster, connected to a single root node. The global index structure is shown in Figure 4.5. We explain the various components in details below:

- *The Root Node R* of GI contains the following information for each cluster S_i: (1) a pointer to the root node R_i (i.e. the address of disk block containing R_i) of the cluster index I_i (the multidimensional index on S_i), (2) the principal components Φ_i (3) the subspace dimensionality d_i and (4) the centroid C_i. It also contains an access pointer O to the outlier cluster $\mathcal{O}$. If there is an index on $\mathcal{O}$ (discussed later), O points to the root node of that index; otherwise, it points to the start of the set of blocks on which the outlier set resides on disk. R may occupy one or more disk blocks depending on the number of clusters K and original dimensionality D.
- *The Cluster Indices:* We maintain a multidimensional index I_i for each cluster S_i in which we store the reduced dimensional representation of the points in S_i. However, instead of building the index I_i on the d_i-d subspace of S_i defined by $\Phi_i^{(j)}, 1 \leq j \leq d_i$, we build I_i on the (d_i+1)-d space, the first d_i dimensions of which are defined by $\Phi_i^{(j)}, 1 \leq j \leq d_i$ as above while the (d_i+1)th dimension is defined by the reconstruction distance $ReconDist(P, S_i, \mathcal{D})$. Including reconstruction distance as a dimension helps to improve query precision (as explained later). We redefine the image $NewImage(P, S_i)$ of a point $P \in \mathcal{A}_i$ as a (d_i+1)-d point (rather than a d_i-d point), incorporating the reconstruction distance as the (d_i+1)th dimension:

$$NewImage(P,S_i)[j] = Image(P,S_i)[j] = P \bullet \Phi_i^{(j)} \text{ for } 1 \leq j \leq d_i \quad (4.7)$$

$$= ReconDist(P,S_i,\mathcal{D}) \text{ for } j = d_i + 1 \quad (4.8)$$

  The $(d_i{+}1)$-d cluster index I_i is constructed by inserting the $(d_i{+}1)$-d images (i.e. $NewImage(P, S_i)$) of each point $P \in \mathcal{A}_i$ into the multidimensional index structure using the insertion algorithm of the index structure. Any disk-based multidimensional index structure (e.g., R-tree [59], X-tree [15], SR-tree [77], Hybrid Tree [23]) can be used for this purpose. We used the hybrid tree in our experiments since it is a space partitioning index structure (i.e. has “dimensionality-independent” fanout), is more scalable to high dimensionalities in terms of query cost and can support arbitrary distance metrics [23].



Root containing pointers to root of each cluster index
. . . .
(d_1+1)-d index on cluster 1
(d_2+1)-d index on cluster 2
. . . .
(d_K+1)-d index on cluster K
Set of outliers (no index: sequentially scanned)

Figure 4.5: The global index structure

- *The Outlier Index:* For the outlier set $\mathcal{O}$, we may or may not build an index depending on whether the original dimensionality D is below or above the critical dimensionality. In this chapter, we assume that D is above the critical dimensionality of the index structure and hence choose not to index the outlier set (i.e. use sequential scan for it).

Like other database index trees (e.g., B-tree, R-tree), the global index (GI) shown in Figure 4.5 is disk-based. But it may not be perfectly height balanced i.e. all paths from R to leaf may not be of exactly equal length. The reason is that the sizes and the dimensionalities may differ from one cluster to another causing the cluster indices to have different heights. We found that GI is *almost* height balanced (i.e. the difference in the lengths of *any* two paths from R to leaf is never more than 1 or 2) due to the size bound on the clusters. Also, its height cannot exceed the height of the original space index by more than 1.

Lemma 2 (Height and balance of GI) *GI is almost height balanced and the height cannot exceed cannot exceed the height of the original space index by more than 1*

Proof: Let h_{GI} denote the the height of GI. Let h_{orig} denote the height of the original space index i.e. index on the entire dataset in the D-d original space. We assume that the multidimensional index structure used as the original space index is same as the one used to index the clusters (e.g., hybrid tree in both cases). Then, $h_{GI} \leq 1 + h_{orig}$. Since I_i is built on a subset of points of the entire set (i.e. $n_i \leq N$) and fewer dimensions (i.e. $d_i \leq D$), its height h_{I_i} cannot be greater h_{orig}. Since $h_{GI} = 1 + max_i h_{I_i}$ and $h_{I_i} \leq h_{orig}$ for all i, $h_{GI} \leq 1 + h_{orig}$. The bound is a conservative one as the h_{GI} is usually smaller than h_{orig} due to the reduced size of the index.

We now show that GI is almost height-balanced. There are two factors that affect the height of a cluster index I_i: the number of points n_i and the subspace dimensionality d_i. Lower the value of n_i, lower the height. Also, lower the value of d_i, lower the height. Let I_{short} be the shortest index. Note $n_{short} \geq MinSize$. Let C_{short} and F_{short} denote the average number of entries in a leaf and index node of I_{short} respectively Then, as explained in [55], the minimum possible height of I_{short} is $(1 + \lceil log_{F_{short}}(\lceil \frac{MinSize}{C_{short}} \rceil) \rceil)$ Similarly, the maximum possible height of tallest index I_{tall} is $(1 + \lceil log_{F_{tall}}(\lceil \frac{N}{C_{tall}} \rceil) \rceil)$ since $n_{tall} \leq N$. For space partitioning index structures (which is preferred for high dimensional indexing due to its "dimensionality-independent" fanout), $F_{short} \sim F_{tall}$ (say, F) [23]. C_{short} and C_{tall} depend on the respective subspace dimensionalities i.e. $\frac{C_{short}}{C_{tall}} \frac{d_{tall}}{d_{short}}$. The maximum difference l_{max} in the lengths of *any* two paths from R to leaf is $l_{max} \sim log_F(\frac{N*C_{short}}{MinSize*C_{tall}})$ i.e. $l_{max} \sim log_F(\frac{N*d_{tall}}{MinSize*d_{short}})$. Usually, the subspace dimensionalities are close i.e. $d_{tall} \sim d_{short}$. For space-partitioning indexes, F is typically around 50-100 [23]. Under the above assumptions, $l_{max} \leq 1$ if $MinSize \geq \frac{N}{50}$ and $l_{max} \leq 2$ if $MinSize \leq \frac{N}{2500}$. In other words, with a proper size bound, l_{max} is usually 1 or at most 2, implying that GI is almost height balanced. ■

To guarantee the correctness of our query algorithms (i.e. to ensure no false dismissals), we need to show that the cluster index distances *lower bounds* the actual distances in the original D-d space [43]. In other words, for any two D-d points P and Q, $\mathcal{D}(NewImage$(P,S_i), $NewImage$(Q,S_i)) must always lower bound $\mathcal{D}(P, Q)$.

Lemma 3 (Lower Bounding Lemma) $\mathcal{D}(NewImage(P, S_i), NewImage(Q, S_i))$ *always lower bounds* $\mathcal{D}(P, Q)$.

Proof: Let P_i denote $Image(P, S_i)$ and Q_i denote $Image(Q, S_i)$. Let $P' = \bar{\Sigma}_{j=1}^{D}(P \bullet \Phi_i^{(j)})$ and $Q' = \bar{\Sigma}_{j=1}^{D}(Q \bullet \Phi_i^{(j)})$. Then, $\mathcal{D}(P', Q') = \mathcal{D}(P, Q)$ since Φ_i is orthonormal. Now,

$$P' = P_i + \overline{ReconVect}(P, S_i) + \bar{\Sigma}_{j=d_i+1}^{D} C_i^{(j)} \Phi_i^{(j)} \quad (4.9)$$

$$Q' = Q_i + \overline{ReconVect}(Q, S_i) + \bar{\Sigma}_{j=d_i+1}^{D} C_i^{(j)} \Phi_i^{(j)} \quad (4.10)$$

The vector distance $\overline{Dist}(P', Q')$ between P' and Q' is

$$\overline{Dist}(P', Q') = \overline{Dist}(P_i, Q_i) + (\overline{ReconVect}(P, S_i) - \overline{ReconVect}(Q, S_i)) \quad (4.11)$$

$$\Rightarrow \mathcal{D}(P', Q') = [\mathcal{D}(P_i, Q_i)^p + \| \overline{ReconVect}(P, S_i) \|_p - \overline{ReconVect}(Q, S_i))^p]^{1/p} \quad (4.12)$$

Since L_p functions obey triangle inequality,

$$\| \overline{ReconVect}(P, S_i) - \overline{ReconVect}(Q, S_i) \|_p \geq |(ReconDist(P, S_i, \mathcal{D}) - ReconDist(Q, S_i, \mathcal{D}))| \quad (4.13)$$

$$\Rightarrow \mathcal{D}(P', Q') \geq [\mathcal{D}(P_i, Q_i)^p + |(ReconDist(P, S_i, \mathcal{D}) - ReconDist(Q, S_i, \mathcal{D}))|^p]^{1/p} \quad (4.14)$$

Now,

$$\mathcal{D}(NewImage(P, S_i), NewImage(Q, S_i)) = [\mathcal{D}(P_i, Q_i)^p + |(ReconDist(P, S_i, \mathcal{D}) - ReconDist(Q, S_i, \mathcal{D}))|^p]^{1/p} \quad (4.15)$$

Since $\mathcal{D}(P', Q') = \mathcal{D}(P, Q)$ and from Equations 4.14 and 4.15,

$$\mathcal{D}(Q,P) \geq \mathcal{D}(NewImage(P,S_i), NewImage(Q,S_i)) \tag{4.16}$$

■

Note that instead of incorporating reconstruction distance as the $(d_i + 1)$th dimension, we could have simply constructed GI with each cluster index I_i defined on the corresponding d_i-d subspace $\Phi_i^{(j)}, 1 \leq j \leq d_i$. Since the lower bounding lemma holds for the d_i-d subspaces (as shown in [43]), the query processing algorithms described below would have been correct. The reason we use $(d_i + 1)$-d subspace is that the distances in the $(d_i + 1)$-d subspace upper bounds the distances in the d_i-d subspace and hence provides a tighter lower bound to distances in the original D-d space:

$$\mathcal{D}(NewImage(P,S_i), NewImage(Q,S_i)) = [\mathcal{D}(Image(P,S_i), Image(Q,S_i))^p + |(ReconDist(P,S_i,\mathcal{D}) - ReconDist(Q,S_i,\mathcal{D}))|^p]^{1/p} \tag{4.17}$$

$$\Rightarrow \mathcal{D}(NewImage(P,S_i), NewImage(Q,S_i)) \geq \mathcal{D}(Image(P,S_i), Image(Q,S_i)) \tag{4.18}$$

Furthermore, the difference between the two (i.e. $\mathcal{D}(NewImage(P,S_i), NewImage(Q,S_i))$ and $\mathcal{D}(Image(P,S_i), Image(Q,S_i))$) is usually significant when computing the distance of the query from a point in the cluster: Say, P is a point in S_i and Q is the query point. Due to the reconstruction distance bound, $ReconDist(P,S_i,\mathcal{D})$ is *always* a small number ($\leq MaxReconDist$). On the other hand, $ReconDist(Q,S_i,\mathcal{D})$ can have any arbitrary value and is usually much larger than $ReconDist(P,S_i,\mathcal{D})$), thus making the difference quite significant. This makes the distance computations in the $(d_i + 1)$-d more optimistic than that in the d_i-d index and hence a better estimate of the distances in the original D-d space. For example, for a range query, the range condition ($\mathcal{D}(NewImage(P,S_i), NewImage(Q,S_i)) \leq \rho$) is more optimistic (i.e. satisfies fewer objects) than the range condition ($\mathcal{D}(Image(P,S_i), Image(Q,S_i)) \leq \rho$), leading to fewer false positives. The same is true for k-NN queries. Fewer false positives imply lower query cost. At the same time, adding a new dimension also increases the cost of the query. Our experiments show that decrease in the query cost from fewer false positives offsets the increase of the cost of the adding a dimension, reducing the overall cost of the query significantly (cf. Section 4.5, Figure 4.12).

4.4.2 Query Processing over the Global Index

In this section, we discuss how to execute similarity queries efficiently using the index structure described above (cf. Figure 4.5). We describe the query processing algorithm for point, range and k-NN queries. For correctness, the query processing algorithm must guarantee that it always returns exactly the same answer as

```
RangeSearch(Query $\mathcal{Q} = \langle Q, \rho, \mathcal{D} \rangle$)

1   for (i=1; i $\leq K$; i++)
2       $Q_i \leftarrow$ NewImage(Q, $S_i$);
3       $\mathcal{Q}_i \leftarrow \langle Q_i, \rho, \mathcal{D} \rangle$;
4       RangeSearchOnClusterIndex($\mathcal{Q}_i, R_i, S_i, result$);
5   for each $O \in \mathcal{O}$
6       if $\mathcal{D}(Q, O) \leq \rho$ $result \leftarrow result \cup O$;
```

```
RangeSearchOnClusterIndex(Query $\mathcal{Q}$, Node T, Cluster S, Set result)

1   if (T is a non-leaf node)
2     foreach child $N$ of $T$
3         if $MINDIST(Q, N, \mathcal{D}) \leq \rho$ RangeSearchOnClusterIndex(Q, N, $S, result$);
4   else /* T is a leaf node */
5     for each object $O$ in $T$
6         if $\mathcal{D}(Q, O) \leq \rho$
7           if $\mathcal{D}(Original(Q, S), Original(O, S)) \leq \rho$ $result \leftarrow result \cup O$;
```

Table 4.4: Range Query.

the query on the original space [43]. Often dimensionality reduction techniques do not satisfy the correctness criteria [76, 142]. We show that all our query processing algorithms satisfy the above criteria.

Point Search

To find an object O, we first find the cluster that contains O. It is the first cluster S (in the order mentioned in Step FC6) for which the reconstruction distance bound is satisfied. If such a cluster S exists, we compute $NewImage(O, S)$ and find it in the corresponding index by invoking the point search algorithm of the index structure. The point search returns the object if it exists in the cluster, otherwise it returns null. If no such cluster S exists, O must be, if at all, in $\mathcal{O}$. So we sequentially search through $\mathcal{O}$ and return it if it exists in $\mathcal{O}$.

Range Queries

A range query $\mathcal{Q} = \langle Q, \rho, \mathcal{D} \rangle$ retrieves all objects O in the database that satisfies the range condition $\mathcal{D}(Q, O) \leq \rho$. The algorithm for range queries is shown in Table 4.4. For each cluster S_i, we map the query anchor Q to its $(d_i + 1)$-d image Q_i (using the principal components Φ_i and subspace dimensionality d_i stored in the root node R of GI) and execute a range query (with the same range ρ) on the corresponding cluster index I_i by invoking the procedure RangeSearchOnClusterIndex on the root node R_i of I_i. RangeSearchOnClusterIndex is the standard R-tree-style recursive range search procedure that starts from the root node and explores the tree in a depth-first fashion. It examines the current node T: if T is a non-leaf node, it recursively searches each child node N of T that satisfies the condition $MINDIST(Q,N,\mathcal{D}) \leq \rho$ (where $MINDIST(Q,N,\mathcal{D})$ denotes the minimum distance of the (d_i+1)-d image of query point to the (d_i+1)-d bounding rectangle of N based on distance function $\mathcal{D}$); if T is a leaf node, it retrieves each data item O stored in T (which is the $NewImage$ of the original D-d object) that satisfies the range condition $\mathcal{D}(Q,O) \leq \rho$ in the (d_i+1)-d space, accesses the full D-dimensional tuple on disk to determine whether it is a false positive and adds it to the result set if it is not a false positive (i.e. it also satisfies the range condition $\mathcal{D}(Q,O) \leq \rho$ in the original D-d space). After all the cluster indices are searched, we add all the qualifying points from among the outliers to the result by performing a sequential scan on $\mathcal{O}$. Since the distance in the index space lower bounds the distance in the original space (cf. Lemma 3), the above algorithm cannot have any false dismissals. The algorithm cannot have any false positives either as they are filtered out before adding to the result set. The above algorithm thus returns exactly the same answer as the query on the original space.

In the above discussion, we assumed that we store the reduced representation of the points (i.e. the 'NewImage's) in the leaf pages of the cluster indices. Another option was to store the original D-d point in the leaf pages (although the index is built on the reduced space). With the former option, the index will have much fewer leaf nodes than the latter due to the smaller representation. On the other hand, in the latter case, the false positives can be eliminated at the leaf page level while the former would require an additional page access into the relation (where the full tuple is stored) to eliminate false positives. Since the index is usually a secondary index, we assume that for each match, we need to access the full tuple anyway (to retrieve the additional attributes). In that case, the extra cost of the former option is that of additional page accesses for *only* the false positives (see Section 4.5.1 for the details on the cost computations). Our experiments show that our technique usually operates in a high precision zone ($> 90\%$) i.e. has very few false positives. The experiments also show that the smaller size of the indices in the former approach saves enough query cost to compensate the few extra I/Os due to false positives. Hence we store just the $NewImage$s in the leaf pages of the index structure.

k Nearest Neighbor Queries

A k-NN query $\mathcal{Q} = \langle Q, k, \mathcal{D}\rangle$ retrieves a set $\mathcal{R}$ of k objects such that for any two objects $O \in \mathcal{R}, O' \notin \mathcal{R}$, $\mathcal{D}(Q,O) \leq \mathcal{D}(Q,O')$. The algorithm for k-NN queries is shown in Table 4.5. Like the basic k-NN algorithm, the algorithm uses a priority queue $queue$ to navigate the nodes/objects in the database in increasing order of their distances from Q. Note that we use a single queue to navigate the entire global index i.e. we explore the nodes/objects of all the cluster indices in an intermixed fashion and do not require separate queues to navigate the different clusters. Each entry in $queue$ is either a node or an object and stores 3 fields: the id of the node/object T it corresponds to, the cluster S it belongs to and its distance $dist$ from the query anchor Q. The items (i.e. nodes/objects) are prioritized based on $dist$ i.e. the smallest item appears at the top of the queue (min-priority queue). For nodes, the distance is defined by $MINDIST$ while for objects, it is the the point-to-point distance. Initially, for each cluster, we map the query anchor Q

```
k-NNSearch(Query $\mathcal{Q} = Q, k, \mathcal{D}\rangle$)

1   for (i=1; i $\leq K$; i++)
2       $Q_{S_i} \leftarrow$ NewImage(Q, $S_i$);
3       $queue$.push($S_i, R_i, MINDIST(Q_i, R_i, \mathcal{D})$);
4   Add to $temp$ the $k$ closest neighbors of $Q$ among $\mathcal{O}$ (lin. scan)
5   while (not $queue$.IsEmpty())
6       top=queue.Top();
7       for each object O in $temp$ such that $O.dist \leq top.dist$
8           $temp \leftarrow temp - O$;
9           $result$ = $result \cup$ O;
10          retrieved++;
11          if (retrieved = k) return $result$;
12      queue.Pop();
13      if $top.T$ is an object
14        $top.dist = \mathcal{D}(Q, Original(top.T, top.S))$;
15        $temp$ = $temp \cup top.T$;
16      else if $top.T$ is a leaf node
17        for each object $O$ in $top.T$
18            $queue$.push(top.S, O, $\mathcal{D}(Q_{top.S}, O)$);
19      else /* $top.T$ is an index node */
20        for each child $N$ of $top.T$
21            $queue$.push(top.S, $N$, $MINDIST(Q_{top.S}, N, \mathcal{D})$);
```

Table 4.5: k-NN Query.

to its $(d_i + 1)$-d image Q_i using the information stored in the root node R of GI (Line 2). Then, for each cluster index I_i, we compute the distance $MINDIST(Q_i, R_i, \mathcal{D})$ of Q_i from the root node R_i of I_i and push R_i into $queue$ along with the distance and the id of the cluster S_i to which it belongs (Line 3). We also fill the set $temp$ with the k closest neighbors of Q among the outliers by sequentially scanning through $\mathcal{O}$ (Line 4).

After these initialization steps, we start navigating the index by popping the item from the top of $queue$ at each step (Line 11). If the popped item is an object, we compute the distance of the original D-d object (by accessing the full tuple on disk) from Q and append it to $temp$ (Lines 12-14). If it a node, we compute the distance of each of its children to the appropriate query image $Q_{top.S}$ (where $top.S$ denotes the cluster which top belongs to) and push them into the queue (Lines 15-20). Note that the image for each cluster is computed just once (in Step 2) and is reused here. We move an object O from $temp$ to $result$ only when we are sure that it is among the k nearest neighbors of Q i.e. there exists no object $O' \notin result$ such that $\mathcal{D}(O', Q) < \mathcal{D}(O, Q)$ and $|result| < k$. The second condition is ensured by the exit condition in Line 11. The condition $O.dist \leq top.dist$ in Line 7 ensures that there exists no *unexplored* object O' such that $\mathcal{D}(O', Q) < \mathcal{D}(O, Q)$. The proof is simple: $O.dist \leq top.dist$ implies $O.dist \leq \mathcal{D}(NewImage(O', S), NewImage(Q, S))$ for any unexplored object O' in a cluster S (by the property of min-priority queue) which in turn implies $\mathcal{D}(O, Q) \leq \mathcal{D}(O', Q)$ (since $D(NewImage(O', S), NewImage(Q, S))$ lower bounds $\mathcal{D}(O', Q)$, see Lemma 3). By inserting the objects in $temp$ (i.e. already explored items) into $result$ in increasing order of their distances in the original D-d space (by keeping $temp$ sorted), we also ensure there exists no *explored* object O' such that $\mathcal{D}(O', Q) < \mathcal{D}(O, Q)$. This shows that the algorithm returns the correct answer i.e. the exact set of objects as the query in the original D-d space. It is also easy to show that the algorithm is I/O optimal.

Lemma 4 (Optimality of k-NN algorithm) *The k-NN algorithm is optimal i.e. it does not explore any object outside the range of kth nearest neighbor.*

Proof: Let $\alpha = max_{O \in \mathcal{A}} \mathcal{D}(Q, O)$ where $\mathcal{A}$ is the set of final answers (the k nearest neighbors). The algorithm is optimal if it does not explore any indexed object O (in any cluster) (13-15) such that $\mathcal{D}(NewImage(O, S),\ NewImage(Q, S)) > \alpha$. Let us assume that it does explore such an object O. When O is explored, $|result| < k$ because otherwise the algorithm would have terminated before reaching this point. We will show that when O is explored, $|result|$ is at least k and hence prove the lemma (by contradiction). Each $O' \in \mathcal{A}$ has been explored before O since $\mathcal{D}(NewImage(O', S), NewImage(Q, S)) \leq \alpha < \mathcal{D}(NewImage(O, S), NewImage(Q, S))$ (by property of min-priority queue). Now $top.dist = \mathcal{D}(NewImage(O, S)\ , NewImage(Q, S))$ when O is explored i.e. $top.dist > \alpha$. Since each $O' \in \mathcal{A}$ satisfies the condition $\mathcal{D}(Q, O) \leq \alpha$, it satisfies the condition $\mathcal{D}(Q, O) < top.dist$ and is hence added to $result$ (Line 7). So $|result|$ is at least k. ■

4.4.3 Modifications

We assume that the data is static in order to build the index. However, we must support subsequent insertions/deletions of the objects to/from the index efficiently. To insert an object O, we find the first cluster S (in the order mentioned earlier) for which the reconstruction distance bound is satisfied i.e. $ReconDist(O, S, \mathcal{D}) \leq ReconError$. If such a cluster exists, we compute $NewImage(O, S)$ and insert it into the corresponding index using the insertion algorithm of the index structure. Otherwise, we append O to $\mathcal{O}$.

The deletion algorithm is also simple. To delete an object O, we first find O by invoking the point search algorithm (cf. Section 4.4.2). If it is found in a cluster, we delete it using the deletion algorithm of the index structure; else if it is found in $\mathcal{O}$, we delete it from $\mathcal{O}$; else, we return not found.

If the database is dynamic (i.e. frequent insertions and deletions), the principal components need to be updated from time to time. One option is to repeat the entire clustering algorithm and construct the index structure from scratch. This can be done more efficiently using techniques proposed by Ravi Kanth et. al. [76]. The idea is to use aggregate data, obtained from the cluster indices, to recompute the principal components for each cluster and then incorporate the new components back into the cluster indices. [76] shows that this technique improves the recomputation time significantly without degrading the quality of the index structure. We can use their approach to handle dynamic databases. On the other hand, if the database is more or less static (i.e. insertions and deletions are rare) as is often the case [43, 41], such recomputations are not necessary.

4.5 Experiments

In this section, we present the results of an extensive empirical study we have conducted to (1) evaluate the effectiveness of LDR as a high dimensional indexing technique and (2) compare it with other techniques, namely, GDR, original space indexing (OSI) and linear scan. We conducted our experiments on both synthetic and real-life datasets. The major findings of our study can be summarized as follows:

- **High Precision:** LDR provides up to an order of magnitude improvement in precision over the GDR technique at the same reduced dimensionality. This indicates that LDR can achieve the same reduction as GDR with significantly lower loss of distance information.
- **Low Query Cost:** LDR consistently outperforms other indexing techniques, namely GDR, original space indexing and sequential scan, in terms of query cost (combined I/O and CPU costs) for both synthetic and real-life datasets.

Thus, our experimental results validate the thesis of this chapter that LDR is an effective indexing technique for high dimensional datasets. All experiments reported in this section were conducted on a Sun Ultra Enterprise 450 machine with 1 GB of physical memory and several GB of secondary storage, running Solaris 2.5.

4.5.1 Experimental Methodology

We conduct the following two sets of experiments to evaluate the LDR technique and compare it with other indexing techniques.

Precision Experiments Due to dimensionality reduction, both GDR and LDR, cause loss of distance information. More the number of dimensions eliminated, more the amount of information lost. We measure this loss by *precision* defined as $Precision = \frac{|R_{original}|}{|R_{reduced}|}$ where $R_{reduced}$ and $R_{original}$ are the sets of answers returned by the range query on the reduced dimensional space and the original HDFS respectively [76]. We repeat that since our algorithms guarantee that the user always gets back the correct set $R_{original}$ of answers (as if the query executed in the original HDFS), precision does *not* measure the quality of the answers returned to the user but just the information loss incurred by the DR technique and hence the query cost. For a DR technique, if we fix the reduced dimensionality, the higher the precision, the lower the cost of the query, the more efficient the technique. We compare the GDR and LDR techniques based on precision at fixed reduced dimensionalities.

Cost Experiments We conducted experiments to measure the query cost (I/O and CPU costs) for each of the following four indexing techniques. We describe how we compute the I/O and CPU costs of the techniques below.

- *Linear Scan:* In this technique, we perform a simple linear scan on the original high dimensional dataset. The I/O cost in terms of sequential disk accesses is $\frac{N*(D*sizeof(float)+sizeof(id))}{PageSize}$. Since

Param.	Description	Default Value
n	Total number of points	100000
D	Original dimensionality	64
k	Number of clusters	5
d	Avg. subspace dimensionality	10
z_{dim}	Skew in subspace dim. across clusters	0.5
z_{size}	Skew in size across clusters	0.5
c	Number of spatial cluster per cluster	10
r	Extent (from centroid) along subspace dim	0.5
p	Max displacement along non-subspace dim	0.1
o	Fraction outliers	0.05

Table 4.6: Input parameters to Synthetic Data Generator

$sizeof(id) \ll (D * sizeof(float))$, we will ignore the $sizeof(id)$ henceforth. Assuming sequential I/O is 10 times faster than random I/O, the cost in terms of the random accesses is $\frac{N*sizeof(float)*D}{10*PageSize}$. The CPU cost is the cost of computing the distance of the query from each point in the database.

- *Original Space Indexing (OSI):* In this technique, we build the index on the original HDFS itself using a multidimensional index structure. We use the hybrid tree as the index structure. The I/O cost (in terms of random disk accesses) of the query is the number of nodes of the index structure accessed. The CPU cost is the CPU time (excluding I/O wait) required to navigate the index and return the answers.
- *GDR:* In this technique, we peform PCA on the original dataset, retain the first few principal components (depending on the desired reduced dimensionality) and index the reduced dimensional space using the hybrid tree index structure. In this case, the I/O cost has 2 components: index page accesses (discussed in OSI) and accessing the full tuples in the relation for false positive elimination (post processing cost). The post processing cost can be one I/O per false positives in the worst case. However, as observed in [55], this assumption is overly pessimistic (and is confirmed by our experiments). We, therefore, assume the postprocessing I/O cost to be $\frac{num\_false\_positives}{2}$. The total I/O cost (in number of random disk accesses) is $index\_page\_access\_cost + \frac{num\_false\_positives}{2}$. The CPU cost is the sum of the index CPU cost and the post processing CPU cost i.e. cost of computing the distance of the query from each of the false positives.
- *LDR:* In this technique, we index each cluster using the hybrid tree multidimensional index structure and used a linear scan for the outlier set. For LDR, the I/O cost of a query has 3 components: index page accesses for each cluster index, linear scan on the outlier set and accessing the full tuples in the relation (post processing cost). The total index page access cost is the total number of nodes accessed of all the cluster indices combined. The number of sequential disk accesses for the outlier scan is $\frac{|\mathcal{O}|*D*sizeof(float)}{PageSize}$. The cost of outlier scan in terms of random accesses is $\frac{|\mathcal{O}|*sizeof(float)*D}{10*PageSize}$. The postprocessing I/O cost is $\frac{num\_false\_positives}{2}$ (as discussed above). The total I/O cost (in number of random disk accesses) is $index\_page\_access\_cost + \frac{|\mathcal{O}|*sizeof(float)*D}{10*PageSize} + \frac{num\_false\_positives}{2}$. Simi-

larly, the CPU cost is the sum of the index CPU cost, outlier scan CPU cost (i.e. cost of computing the distance of the query from each of the outliers) and the post processing cost (i.e. cost of computing the distance of the query from each of the false positives).

We chose the hybrid tree as the index structure for our experiments since it is a space partitioning index structure ("dimensionality-independent" fanout) and has been shown to scale to high dimensionalities [23]. [5] We use a page size of 4KB for all our experiments.

4.5.2 Experimental Results - Synthetic Data Sets

Synthetic Data Sets and Queries In order to generate the synthetic data, we use a method similar to that discussed in [154] but appropriately modified so that we can generate the different clusters in subspaces of different orientations and dimensionalities. The generator generates k clusters with a total of $n.(1-o)$ points distributed among them using a Zipfian distribution with value z_{size}. The subspace dimensionality of each cluster also follows a Zipfian distribution with value z_{dim}, the average subspace dimensionality being d. Each cluster is generated as follows. For a cluster with size n_i and subspace dimensionality d_i (computed using the Zipfian distributions described above), we randomly choose d_i dimensions among the D dimensions as the subspace dimensions and generate n_i points in that d_i-d plane. Along each of the remaining $(D-d_i)$ non-subspace dimensions, we assign a randomly chosen coordinate to all the n_i points in the cluster. Let f_j be the randomly chosen coordinate along the jth non-subspace dimension. In the subspace, the points are spatially clustered into several regions (c regions on average) with each region having a randomly chosen centroid and an extent of r from the centroid along each of the d_i dimensions. After all the points in the cluster are generated, each point is displaced by a distance of at most p in either direction along each non-subspace dimension i.e. the point is randomly placed somewhere between (f_j-p) and (f_j+p) along the jth non-subspace dimension. The amount of displacement (i.e. value of p) determines the degree of correlation (since r is fixed). Lower the value, more the correlation. To make the subspaces arbitrarily oriented, we generate a random orthonormal rotation matrix (generated using MATLAB) and rotate the cluster by multiplying the data matrix with the rotation matrix. After all the clusters are generated, we randomly generate $N.o$ points (with random values along all D dimensions) as the outliers. The default values of the various parameters is shown in Table 4.6.

We generated 100 range queries by selecting their query anchors randomly from the dataset and choosing a range value such that the average query selectivity is about 2%. We tested with only range queries since the k-NN algorithm, being optimal, is identical to the range query with the range equal to the distance of the kth nearest neighbor from the query (Lemma 3). We use L_2 distance (Euclidean) as the distance metric. All our measurements are averaged over the 100 queries.

[5] The performance gap between our technique and the other techniques was even greater with SR-tree [77] as the index structure due to higher dimensionality curse [23]. We do not report those results here but can be found in the full version of the LDR paper [25].

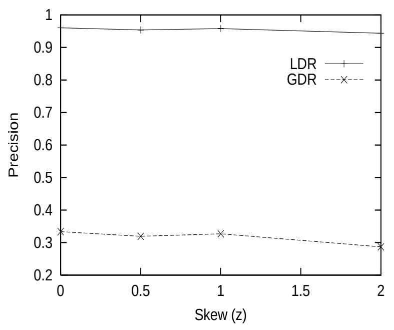


Figure 4.6: Sensitivity of precision to skew.

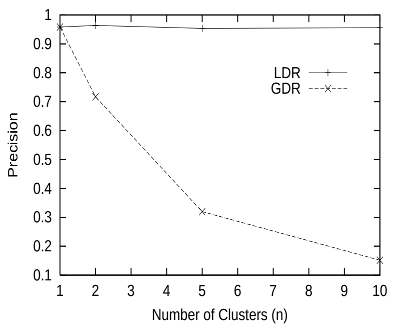


Figure 4.7: Sensitivity of precision to number of clusters.

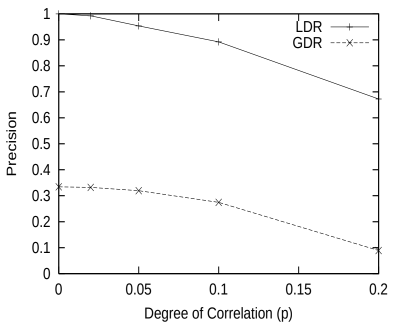


Figure 4.8: Sensitivity of precision to degree of correlation.

Precision Experiments In our first set of experiments, we carry out a sensitivity analysis of the GDR and LDR techniques to parameters like skew in the size of the clusters (z_{size}), number of clusters (k) and degree of correlation (p). In each experiment, we vary the parameter of interest while the remaining parameters are fixed at their default values. We fix the reduced dimensionality of the GDR technique to 15. We fix the average subspace dimensionality of the clusters (i.e. $\Sigma_{i=1}^{K} \frac{n_i d_i}{K}$) also to 15 by choosing $FracOutliers$ and $MaxReconDist$ appropriately ($FracOutliers = 0.1$ and $MaxReconDist = 0.5$). Figure 4.6 compares the precision of the LDR technique with that of GDR for various value of z_{size}. LDR achieves about 3 times higher precision compared to GDR i.e. the latter has more than three times the number of false positives as the former. The precision of neither technique changes significantly with the skew. Figure 4.7 compares the precision of the two techniques for various values of k. As expected, for one cluster, the two techniques are identical. As k increases, the precision of GDR deteriorates while that of LDR is independent of the number of clusters. For $k = 10$, LDR is almost an order of magnitude better compared to GDR in terms of precision. Figure 4.8 compares the two techniques for various values of p. As the degree of correlation decreases (i.e. the value of p increases), the precision of both techniques drop but LDR outperforms GDR for all values p. Figure 4.9 shows the variation of the precision with the reduced dimensionality. For the GDR technique, we vary the reduced dimensionality from 15 to 60. For the LDR technique, we vary the $FracOutliers$ from 0.2 to 0.01 (0.2, 0.15, 0.1, 0.05, 0.02, 0.01) causing the average subspace dimensionality to vary from 7 to 42 (7, 10, 12, 14, 23 and 42) ($MaxDim$ was 64). The precision of both techniques increase with the increase in reduced dimensionality. Once again, LDR consistently outperforms GDR at all dimensionalities. The above experiments show that LDR is a more effective dimensionality reduction technique as it can achieve the same reduction as GDR with significantly lower loss of information (i.e. high precision) and hence significantly lower cost as confirmed in the cost experiments described next.

Cost Experiments We compare the 4 techniques, namely LDR, GDR, OSI and Linear Scan, in terms of query cost for the synthetic dataset. Figure 4.10 compares the I/O cost of the 4 techniques. Both the LDR and GDR techniques have U-shaped cost curves: when the reduced dimensionality is too low, there is a high degree of information loss leading to a large number of false positives and hence a high post-processing cost;

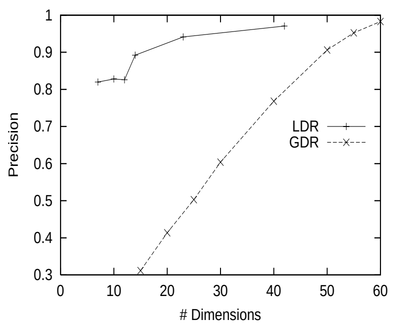


Figure 4.9: Sensitivity of precision to reduced dimensionality.

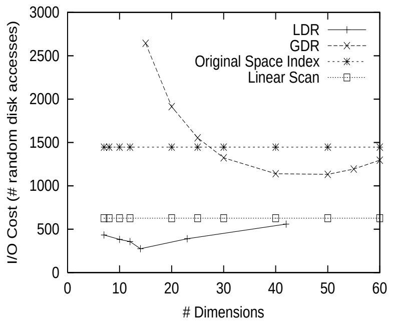


Figure 4.10: Comparison of LDR, GDR, Original Space Indexing and Linear Scan in terms of I/O cost. For linear scan, the cost is computed as: $\frac{num\_sequential\_disk\_accesses}{10}$.

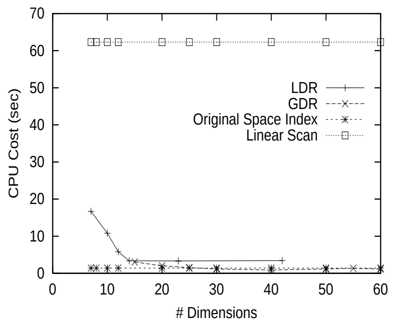


Figure 4.11: Comparison of LDR, GDR, Original Space Indexing and Linear Scan in terms of CPU cost.

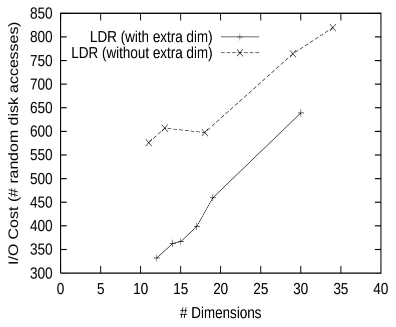


Figure 4.12: Effect of adding the extra dimension.

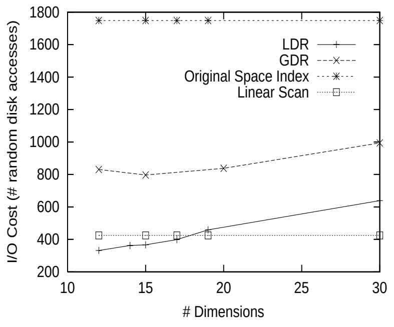


Figure 4.13: Comparison of LDR, GDR, Original Space Indexing and Linear Scan in terms of I/O cost. For linear scan, the cost is computed as: $\frac{num\_sequential\_disk\_accesses}{10}$.

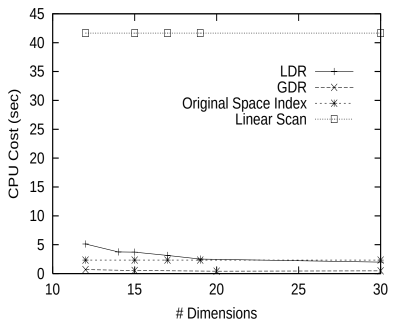


Figure 4.14: Comparison of LDR, GDR, Original Space Indexing and Linear Scan in terms of CPU cost.

when it is too high, the index page access cost becomes too high due to dimensionality curse. The optimum points lies somewhere in the middle: it is at dimensionality 14 (about 250 random disk accesses) for LDR and at 40 (about 1200 random disk accesses) for GDR. The I/O cost of OSI and Linear Scan is obviously independent of the reduced dimensionality. LDR significantly outperforms all the other 3 techniques in terms of I/O cost. The only technique that comes close to LDR in terms of I/O cost is the linear scan (but LDR is 2.5 times better as the latter performs 6274 sequential accesses $\sim$ 627 random accesses). However, linear scan loses out mainly due to its high CPU cost shown in Figure 4.11. While LDR, GDR and OSI techniques have similar CPU cost (at their respective optimum points), the CPU cost linear scan is almost two orders of magnitude higher that the rest. LDR has slightly higher CPU cost compared to GDR and OSI since it uses linear scan for the outlier set: however, the savings in the I/O cost over GDR and OSI (by a factor of 5-6) far offsets the slightly higher CPU cost.

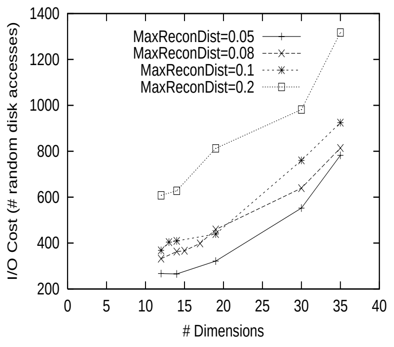


Figure 4.15: Sensitivity of I/O cost of LDR technique to MaxReconDist.

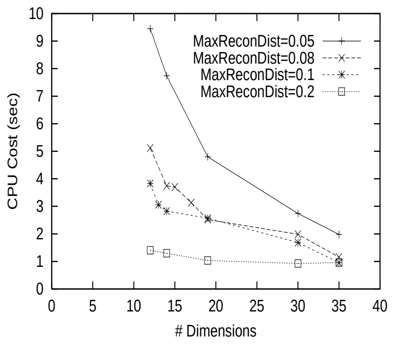


Figure 4.16: Sensitivity of CPU cost of LDR technique to MaxReconDist.

4.5.3 Experimental Results - Real-Life Data Sets

Description of Dataset Our real-life data set (COLHIST dataset [23]) comprises of 8×8 color histograms (64-d data) extracted from about 70,000 color images obtained from the Corel Database (http://corel.digitalriver.com/) and is available online at the UCI KDD Archive web site (http://kdd.ics.uci.edu/databases/CorelFeatures). We generated 100 range queries by selecting their query anchors randomly from the dataset and choosing a range value such that the average query selectivity is about 0.5%. All our measurements are averaged over the 100 queries.

Cost Experiments First, we evaluate the impact of adding $ReconDist$ as an additional dimension of each cluster in the LDR technique. Figure 4.12 shows that the additional dimension reduces the cost of the query significantly. We performed the above experiment on the synthetic dataset as well and observed a similar result. Figures 4.15 and 4.16 shows the sensitivity of the LDR technique to the $MaxReconDist$ parameter in terms of I/O and CPU costs respectively. The I/O cost improves with decrease in $MaxReconDist$ due to decrease in the information loss (i.e. fewer false positives) and hence decrease in post processing cost. However, with the decrease in $MaxReconDist$, the number of outliers increase as fewer points satisfy the reconstruction distance bound which causes the CPU cost to increase (the cost of scanning the outlier set) as shown in the Figure 4.16. The choice of $MaxReconDist$ must consider the combined I/O and CPU cost; for example, $MaxReconDist = 0.08$ represents a good choice for this real-life dataset.

Figure 4.13 compares the 4 techniques, namely LDR, GDR, OSI and Linear Scan, in terms of I/O cost. LDR outperforms all other techniques significantly. Again, the only technique that come close to LDR in I/O cost (i.e. number of random disk accesses) is the linear scan. However, again, linear scan turns out to significantly worse compared to LDR in terms of the overall cost due to its high CPU cost as shown in Figure 4.14.

4.6 Conclusion

With numerous emerging applications requiring efficient access to high dimensional datasets, there is a need for scalable techniques to indexing high dimensional data. In this chapter, we proposed local dimensionality reduction (LDR) as an approach to indexing high dimensional spaces. We developed an algorithm to discover the locally correlated clusters in the dataset and perform dimensionality reduction on each of them individually. We presented an index structure that exploits the correlated clusters to efficiently support similarity queries over high dimensional datasets. We have shown that our query processing algorithms are correct and optimal. We conducted an extensive experimental study with synthetic as well as real-life datasets to evaluate the effectiveness of our technique and compare it to GDR, original space indexing and linear scan techniques. Our results demonstrate that our technique (1) reduces the dimensionality of the data with significantly lower loss in distance information compared to GDR, outperforming GDR by almost an order of magnitude in terms of query precision (for the same reduced dimensionality) and (2) significantly outperforms all the other 3 techniques (namely, GDR, original space indexing and linear scan) in terms of the query cost for both synthetic and real-life datasets.

In the next chapter, we present a new dimensionality reduction technique, called Adaptive Piecewise Constant Approximation (APCA), for time series data. APCA goes a step further compared to LDR; while LDR chooses a reduced-representation that is local to each cluster, APCA adapts locally to each data item in the database and chooses the best reduced-representation for that item. We show how APCA can be indexed using a multidimensional index structure. Such an index enables extremely fast similarity searching in time series data.

Chapter 5

Indexing Time Series Data

In this, we present a new locally adaptive dimensionality reduction technique for indexing large time series databases.

5.1 Introduction

Time series account for a large proportion of the data stored in financial, medical and scientific databases. Recently there has been much interest in the problem of similarity search (query-by-content) in time series databases. Similarity search is useful in its own right as a tool for exploratory data analysis, and it is also an important element of many data mining applications such as clustering [36], classification [80, 101] and mining of association rules [35].

The similarity between two time series is typically measured with Euclidean distance, which can be calculated very efficiently. However the volume of data typically encountered exasperates the problem. Multi-gigabyte datasets are very common. As typical example, consider the MACHCO project. This database contains more than a terabyte of data and is updated at the rate of several gigabytes a day [148].

The most promising similarity search methods are techniques that perform dimensionality reduction on the data, then use a multidimensional index structure to index the data in the transformed space. The technique was introduced in [5] and extended in [119, 32]. The original work by Agrawal et. al. utilizes the Discrete Fourier Transform (DFT) to perform the dimensionality reduction, but other techniques have been suggested, including Singular Value Decomposition (SVD) [79, 76, 81], the Discrete Wavelet Transform (DWT) [29, 151, 75] and Piecewise Aggregate Approximation (PAA) [79, 153].

For a given index structure, the efficiency of indexing depends only on the fidelity of the approximation in the reduced dimensionality space. However, in choosing a dimensionality reduction technique, we cannot simply choose an arbitrary compression algorithm. What is required is a technique that produces an indexable representation. For example, many time series can be efficiently compressed by delta encoding, but this representation does not lend itself to indexing. In contrast SVD, DFT, DWT and PAA all lend themselves naturally to indexing, with each eigenwave, fourier coefficient, wavelet coefficient or aggregate segment mapping onto one dimension of an index tree.

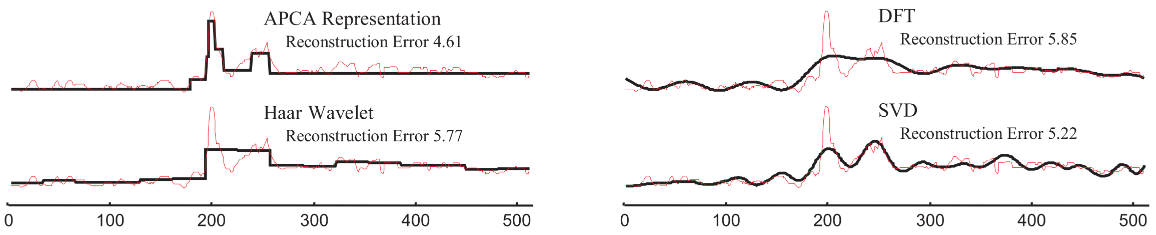


Figure 5.1: A visual comparison of the time series representation proposed in this work (APCA), and the 3 other representations advocated in the literature. For fair comparison, all representations have the same compression ratio. The reconstruction error is the Euclidean distance between the original time series and its approximation.

The main contribution of this chapter is to propose a simple, but highly effective compression technique, Adaptive Piecewise Constant Approximation (APCA), and show that it can be indexed using a multidimensional index structure. This representation was considered by other researchers, but they suggested it ”does not allow for indexing due to its irregularity” [153]. We will show that indexing APCA is possible, and, using APCA is up to one to two orders of magnitude more efficient than alternative techniques on real world datasets. We will show that our distance measure in the APCA space lower bounds the true distance (i.e., Euclidean distance in the original space); hence our APCA index always returns exact results. We will define the APCA representation in detail in Section 5.3, however an intuitive understanding can be gleaned from Figure 5.1.

The rest of the chapter is organized as follows. In Section 5.2 we provide background on and review related work in time series similarity search. In Section 5.3 we introduce the APCA representation, an algorithm to compute it efficiently and two distance measures defined on it. In Section 5.4 we demonstrate how to index the APCA representation. Section 5.5 contains a comprehensive experimental comparison of APCA with all the competing techniques. In section 5.8 we discuss several advantages APCA has over the competing techniques, in addition to being faster. Section 5.9 offers the conclusions.

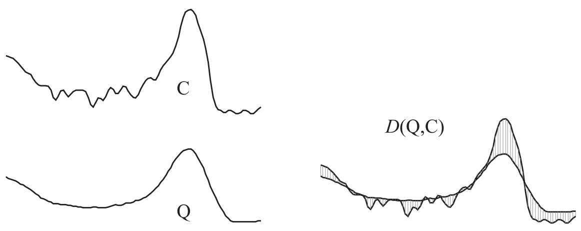


Figure 5.2: The intuition behind the Euclidean distance. The Euclidean distance can be visualized as the square root of the sum of the squared lengths of the gray lines.

5.2 Background and Related Work

Given two time series Q= $\{q_1, \ldots, q_n\}$ and C= $\{c_1, \ldots, c_n\}$, the Euclidean distance D(Q,C) between Q and C is defined as:

$$D(Q,C) = \sqrt{\sum_{i=1}^{n} (q_i - c_i)^2} \tag{5.1}$$

Figure 5.2 shows the intuition behind the Euclidean distance.

There are essentially two ways the data might be organized [46]:

- **Whole Matching**: Here it assumed that all sequences to be compared are the same length n.
- **Subsequence Matching**: Here we have a query sequence Q (of length n), and a longer sequence C (of length m). The task is to find the subsequence in C of length n, beginning at c_i, which best matches Q, and report its offset within C.

Whole matching requires comparing the query sequence to each candidate sequence by evaluating the distance function and keeping track of the sequence with the lowest distance. Subsequence matching requires that the query Q be placed at every possible offset within the longer sequence C. Note it is possible to convert subsequence matching to whole matching by sliding a "window" of length n across C, and making copies of the $(m - n)$ windows. Figure 5.3 illustrates the idea. Although this causes storage redundancy it simplifies the notation and algorithms so we will adopt this policy for the rest of this chapter.

There are two important kinds of queries that we would like to support in time series database, range queries (e.g., return all sequences within an epsilon of the query sequence) and nearest neighbor (e.g., return the K closest sequences to the query sequence). The brute force approach to answering these queries, sequential scanning, requires comparing every time series c_i to Q. Clearly this approach is unrealistic for large datasets.

Any indexing scheme that does not examine the entire dataset could potentially suffer from two problems, false alarms and false dismissals. False alarms occur when objects that appear to be close in the index are actually distant. Because false alarms can be removed in a post-processing stage (by confirming distance estimates on the original data), they can be tolerated so long as they are relatively infrequent. A

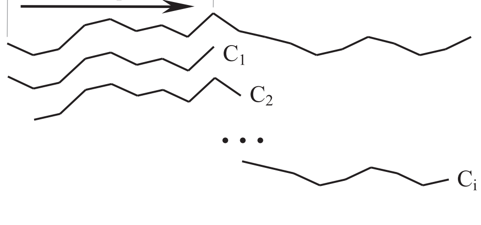


Figure 5.3: The subsequence matching problem can be converted into the whole matching problem by sliding a "window" of length n across the long sequence and making copies of the data falling within the windows.

false dismissal is when qualifying objects are missed because they appear distant in index space. Similarity-searching techniques that guarantee no false dismissals are referred to as *exact* while techniques that do not have this guarantee are called *approximate*. Although approximate techniques can sometimes be useful for exploring large databases, we do not consider them in this thesis. We devote the rest of this section in reviewing exact techniques for similarity search in time series data.

A time series $C = \{c_1, \ldots, c_n\}$ with n datapoints can be considered as a point in n-dimensional space. This immediately suggests that time series could be indexed by multidimensional index structure such as the R-tree and its many variants [59]. Since realistic queries typically contain 20 to 1,000 datapoints (i.e., n varies from 20 to 1000) and most multidimensional index structures have poor performance at dimensionalities greater than 8-12 [23], we need to first perform dimensionality reduction in order to exploit multidimensional index structures to index time series data. In [46], the authors introduced GEneric Multimedia INdexIng method (GEMINI) which can exploit any dimensionality reduction method to allow efficient indexing. The technique was originally introduced for time series, but has been successfully extend to many other types of data [81].

An important result in [46] is that the authors proved that in order to guarantee no false dismissals, the distance measure in the index space must satisfy the following condition:

$$D_{index\_space}(A, B) \leq D_{true}(A, B) \tag{5.2}$$

This theorem is known as the lower bounding lemma or the contractive property. Given the lower bounding lemma, and the ready availability of off-the-shelf multidimensional index structures, GEMINI requires just the following three steps:

- Establish a distance metric D_{true} from a domain expert (in this case Euclidean distance).
- Produce a dimensionality reduction technique that reduces the dimensionality of the data from n to N, where N can be efficiently handled by your favorite index structure.
- Produce a distance measure D_{index_space} defined on the N dimensional representation of the data, and prove that it obeys Equation 5.2

Table 5.1 contains an outline of the GEMINI indexing algorithm. All sequences in the dataset **C** are transformed by some dimensionality reduction technique and then indexed by the index structure of choice. The indexing tree represents the transformed sequences as points in N dimensional space. Each point contains a pointer to the corresponding original sequence on disk.

```
Algorithm BuildIndex(C,$n$)                    // C is the dataset, $n$ is the size of the window
for each object $C_i \in$ C
  $C_i \leftarrow C_i - Mean(C_i)$;                      // Optional: remove the mean of Ci
  $\bar{C_i} \leftarrow SomeTransformation(C_i)$;        // $\bar{C_i}$ is any dimensionality reduced representation
  Insert $\bar{C_i}$ into Spatial Access Method with a pointer to Ci on disk from leaf page;
```

Table 5.1: An outline of the GEMINI index building algorithm.

Note that each sequence has its mean subtracted before indexing. This has the effect of shifting the sequence in the y-axis such that its mean is zero, removing information about its offset. This step is included because for most applications the offset is irrelevant when computing similarity.

Table 5.2 below contains an outline of the GEMINI range query algorithm.

Algorithm RangeQuery(Q,ϵ)
Project the query Q into the same feature space as the index.
Find all candidate objects in the index within ϵ of the query.
Retrieve from disk the actual sequences pointed to by the candidates.
Compute the actual distances, and discard false alarms.

Table 5.2: The GEMINI range query algorithm.

The range query algorithm is called as a subroutine in the K Nearest Neighbor algorithm outlined in Table 5.3. There are several optimizations to this basic K Nearest Neighbor algorithm that we utilize in this chapter [131]. We will discuss them in more detail in Section 5.4.

Algorithm K_NearestNeighbor(Q,K)
Project the query Q into the same feature space as the index.
Find the K nearest candidate objects in the index.
Retrieve from disk the actual sequences pointed to by the candidates.
Compute the actual distances and record the maximum, call it ϵ_{max}.
Issue the range query, RangeQuery(Q,ϵ_{max}).
Compute the actual distances, and choose the nearest K.

Table 5.3: The GEMINI nearest neighbor algorithm.

The efficiency of the GEMINI query algorithms depends only on the quality of the transformation used to build the index. The tighter the bound in Equation 5.2 the better, as tighter bounds imply fewer false alarms hence lower query cost [24]. Time series are usually good candidates for dimensionality reduction because they tend to contain highly correlated features. For brevity, we will not describe the three main dimensionality reduction techniques, SVD, DFT and DWT, in detail. Instead we refer the interested reader to the relevant papers or to [79] which contains a survey of all the techniques. We will briefly revisit related work in Section 5.8 when the reader has developed more intuition about our approach.

5.3 Adaptive Resolution Representation

In recent work Keogh et. al. [79] and Yi and Faloutsos [153] independently suggested approximating a time series by dividing it into equal-length segments and recording the mean value of the datapoints that fall within the segment. The authors use different names for this representation, for clarity we will refer to it as Piecewise Aggregate Approximation (PAA). This simple technique is surprisingly competitive with the more sophisticated transforms.

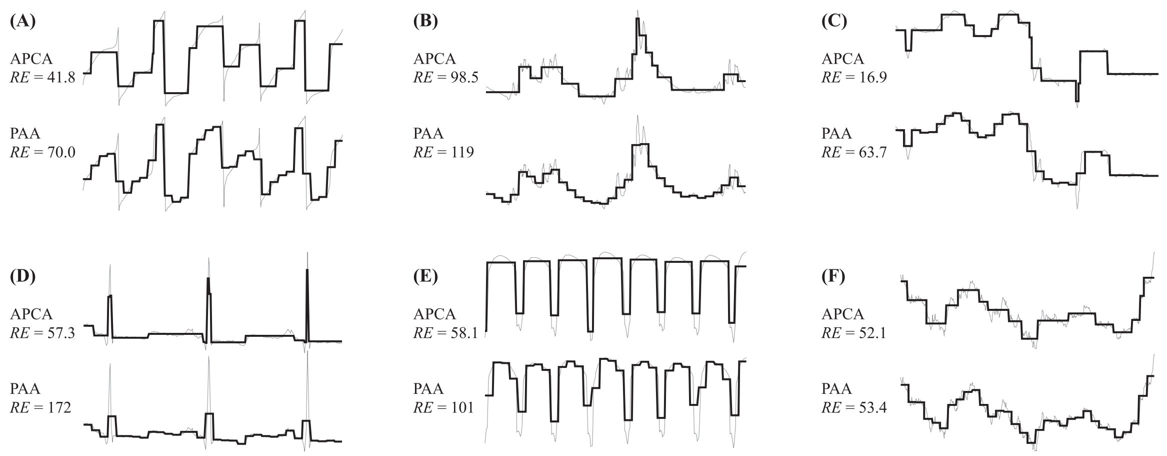


Figure 5.4: A comparison of the reconstruction errors of the equal-size segment approach (PAA) and the variable length segment approach (APCA), on a collection of miscellaneous datasets. A) INTERBALL Plasma processes. B) Darwin sea level pressures. C) Space Shuttle telemetry. D) Electrocardiogram. E) Manufacturing. F) Exchange rate.

The fact that each segment in PAA is the same length facilitates indexing of this representation. Suppose however we relaxed this requirement and allowed the segments to have arbitrary lengths, does this improve the quality of the approximation? Before we consider this question, we must remember that the approach that allows arbitrary length segments, which we call Adaptive Piecewise Constant Approximation (APCA), requires two numbers per segment. The first number records the mean value of all the datapoints in segment, the second number records the length. So a fair comparison is N PAA segments to M APCA segments, were $M = \lfloor \frac{N}{2} \rfloor$.

It is difficult to make any intuitive guess about the relative performance of the two techniques. On one hand PAA has the advantage of having twice as many approximating segments. On the other hand APCA has the advantage of being able to place a single segment in an area of low activity and many segments in areas of high activity. In addition one has to consider the structure of the data in question. It is possible to construct artificial datasets where one approach has an arbitrarily large reconstruction error, while the other approach has reconstruction error of zero.

Figure 5.4 illustrates a fair comparison between the two techniques on several real datasets. Note that for the task of indexing, subjective feelings about which technique “looks better” are irrelevant. All that matters is the quality of the approximation, which is given by the reconstruction error (because lower reconstruction errors result in tighter bounds on $D_{index\_space}(A, B) \leq D_{true}(A, B)$).

On five of the six time series APCA outperforms PAA significantly. Only on the Exchange Rate data are they essentially the same. In fact, we repeated similar experiments for more than 40 different time series datasets, over a range of sequence lengths and compression ratios and we found that APCA is always at least as good as PAA, and usually much better. This comparison motivates our approach. If the APCA representation can be indexed, its high fidelity to the original signal should allow very efficient pruning of the index space (i.e., few false alarms, hence low query cost). We will show how APCA can be indexed

Symbols	Definitions
S	The number of objects in the database
n	The length of a time series (also called query length, original dimensionality)
$C = \{c_1, \ldots, c_n\}$	A time series in a database, stored a vector of length n
$Q = \{q_1, \ldots, q_n\}$	The user specified query, represented as a vector of length n
N	The dimensionality of an index structure, with $N \ll n$
M	The number of segments in a APCA representation, with M = $\lfloor \frac{N}{2} \rfloor$
$C = \{\langle cv_1, cr_1 \rangle, \ldots, \langle cv_M, cr_M \rangle\}$	An adaptive piecewise constant approximation of C, with c_i the value of the ith segment and cr_i the right endpoint of the ith segment
$Q' = \{\langle qv_1, qr_1 \rangle, \ldots, \langle qv_M, qr_M \rangle\}$	Also an adaptive piecewise constant approximation, but obtained using a special algorithm as describe in Equation 5.4
D	The Euclidean distance, defined for Q and C
D_{LB}	An approximation of the Euclidean distance, defined for Q' and C

Table 5.4: The notation used in this chapter.

in the next section (Section 5.4). In the rest of this section, we define the APCA representation formally, describe the algorithm to obtain the APCA representation of a time series and discuss the distance measures for APCA.

5.3.1 The APCA representation

Given a time series $C = \{c_1, \ldots, c_n\}$, we need to be able to produce an APCA representation, which we will represent as:

$$C = \{\langle cv_1, cr_1 \rangle, \ldots, \langle cv_M, cr_M \rangle\}, \quad cr_0 = 0 \tag{5.3}$$

where cv_i is the mean value of datapoints in the ith segment (i.e., cv_i = mean($c_{cr_{i-1}+1}, \ldots, c_{cr_i}$)) and cr_i the right endpoint of the ith segment. We do not represent the length of the segments but record the locations of their right endpoints instead for indexing reasons as will be discussed in Section 5.4. The length of the ith segment can be calculated as $(cr_i - cri - 1)$. Figure 5.5 illustrates this notation.

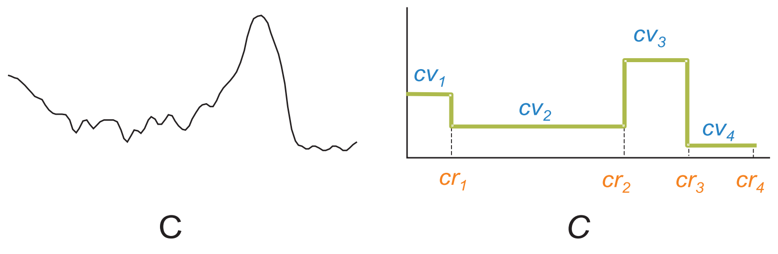


Figure 5.5: A time series C and its APCA representation C, with M = 4

5.3.2 Obtaining the APCA representation

As mentioned before, the performance of the index structure built on the APCA representation defined in Equation 5.3 depends on how closely the APCA representation approximates the original signal. Closer the approximation, fewer the number of false alarms, better the performance of the index. We say that an M-segment APCA representation C of a time series C is *optimal* (in terms of the quality of approximation) iff C has the least reconstruction error among all possible M-segment APCA representations of C. Finding the optimal piecewise polynomial representation of a time series requires a $O(Mn^2)$ dynamic programming algorithm [15, 35]. This is too slow for high dimensional data. In this chapter, we propose a new algorithm to produce almost optimal APCA representations in $O(nlog(n))$ time. The algorithm works by first converting the problem into a wavelet compression problem, for which there are well known optimal solutions, then converting the solution back to the ACPA representation and (possibly) making minor modifications. The algorithm leverages off the fact that the Haar wavelet transformation of a time series signal can be calculated in $O(n)$, and that an optimal reconstruction of the signal for any level of compression can be obtained by sorting the normalized coefficients in order of decreasing magnitude, then truncating off the smaller coefficients [136]. Note that such a reconstruction is equivalent to an APCA representation. There are, however, two issues we must address before utilizing this approach.

1. The DWT is defined only for time series with a length that is an integer power of two while n may not necessarily be a power of two. This problem can be solved easily by padding those time series with zeros, then truncating the corresponding segment after performing the DWT.

2. There is no direct mapping between the number of Haar coefficients retained and the number of segments in the APCA representation resulting from the reconstruction. For example a single coefficient Haar approximation could produce a 1, 2 or 3-segment APCA representation. Our solution is to keep the largest M coefficients, and if this results in an APCA representation with more than M segments, adjacent pairs of segments are merged until exactly M segments remain. The segment pairs targeted for merging are those that can be fused into a single segment with the minimum increase in reconstruction error.

Table 5.5 contains the outline of the algorithm, and Figure 5.6 illustrates the working of the algorithm on real world data.

We experimentally compared this algorithm with several of the heuristic, merging algorithms [45, 114, 133] and found it is faster (at least 5 times faster for any length time series) and slightly superior in terms of reconstruction error.

5.3.3 Lower Bounding Distance measure for the APCA representation

Suppose we have a time series C, which we convert to the APCA representation C, and a query time series Q. Clearly, no distance measure defined between Q and C can be exactly equivalent to the Euclidean distance

```
Algorithm Compute_APCA(C,M)
if length(C) is not a power of two, pad it with zeros to make it so.
Perform the Haar Discrete Wavelet Transform on C.
Sort coefficients in order of decreasing magnitude, truncate after M.
Reconstruct approximation (APCA representation) of C from retained coeffs.
If C was padded with zeros, truncate it to the original length.
while the number of segments is greater than M
  Merge the pair of segments that can be merged with the least rise in error.
```

Table 5.5: An algorithm to produce the APCA.

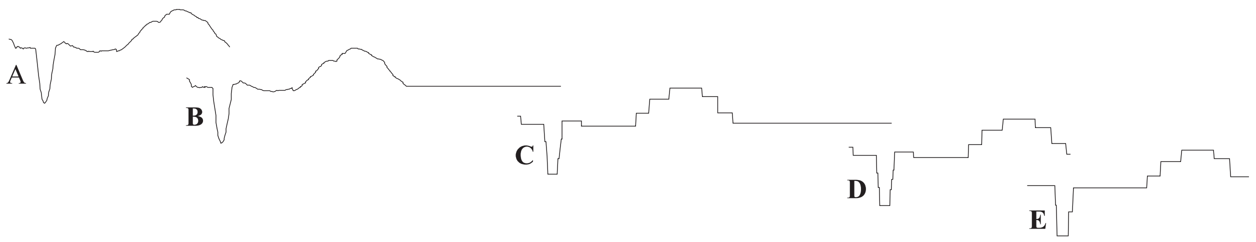


Figure 5.6: A visualization of the algorithm used to produce the APCA representation. The original time series (A) is padded with zeros up to the next power of two (B). The optimal Haar compression for M coefficients is obtained (C), it consists of slightly more than M segments. The sequence is truncated back to the original length (D) and segments are merged until exactly M remain (E).

D(Q,C) (defined in Equation 5.1) because C generally contains less information than C. We need to define a distance measures D_{LB}(Q,C) between Q and C that lower bounds the Euclidean distance D(Q,C) so that we can utilize the GEMINI framework. To define D_{LB}(Q,C), we must first introduce a special version of the APCA. Normally the algorithm mentioned in Section 5.3.2 is used to obtain the APCA representation of any time series. However we can also obtain the APCA representation of the query time series Q by "projecting" the endpoints of C onto Q, and finding the mean value of the sections of Q that fall within the projected intervals. A time series Q converted into the APCA representation this way is denoted as Q'. The idea can be visualized in Figure 5.7(a). Q' is defined as:

$$Q' = \{\langle qv_1, qr_1\rangle, \dots, \langle qv_M, qr_M\rangle\} \quad \text{where } qr_i = cr_i \text{ and } qv_i = mean(q_{cr_{i-1}+1}, \dots, q_{cr_i}) \tag{5.4}$$

DLB(Q',C) is defined as (see Figure 5.7(b)):

$$D_{LB}(Q', C) = \sqrt{\sum_{i=1}^{M}(cr_i - cr_{i-1})(qv_i - cv_i)^2} \tag{5.5}$$

Lemma 5 (Lower Bounding Lemma) *D_{LB}(Q',C) lower bounds the Euclidean Distance D(Q,C).*

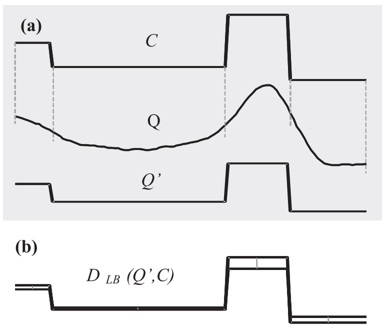


Figure 5.7: A visualization of the lower bounding distance measure $D_{LB}(Q', C)$ defined on the APCA representation. (a) Q' is obtained by projecting the endpoints of C onto Q and calculating the mean values of the sections falling within the projected lines. (b) $D_{LB}(Q', C)$ can be visualized as the square root of the sum of the product of squared length of the gray lines with the length of the segments they join.

Proof: We present a proof for the case where there is a single segment in the APCA representation. The more general proof for the M segment case can be obtained by applying the proof to each of the M segments.

Let $W = \{w_1, w_2, \ldots, w_p\}$ be a vector of p real numbers. Let $\bar{W}$ denote the arithmetic mean of W, i.e., $\bar{W} = \frac{\sum w_i}{p}$. We define a vector ΔW of real numbers where $\Delta w_i = \bar{W} - w_i$. It is easy to see that $\sum \Delta w_i = 0$. The definition of Δw_i allows us to substitute w_i by $\bar{W} - \Delta w_i$, a fact which we will utilize in the proof below.

Let Q and C be the query and data time series respectively, with $|\text{Q}| = |\text{C}| = n$. Let Q' and C be the corresponding APCA vectors as defined in Equations 5.4 and 5.3 respectively.

We want to prove

$$\sqrt{\sum_{i=1}^{n}(q_i - c_i)^2} \geq \sqrt{\sum_{i=1}^{M}(cr_i - cr_{i-1})(qv_i - cv_i)^2} \tag{5.6}$$

We start the proof with the assumption that the above is true. Since we are considering just the single segment case, we can remove summation over M segments and rewrite the inequality as:

$$\sqrt{\sum_{i=1}^{n}(q_i - c_i)^2} \geq \sqrt{(cr_i - cr_{i-1})(qv_i - cv_i)^2} \tag{5.7}$$

Since $(cr_i - cr_{i-1}) = n$,

$$\sqrt{\sum_{i=1}^{n}(q_i - c_i)^2} \geq \sqrt{n(qv_i - cv_i)^2} \tag{5.8}$$

Squaring both sides,

$$\sum_{i=1}^{n}(q_i - c_i)^2 \geq n(qv_i - cv_i)^2 \tag{5.9}$$

Since $qv_i = \bar{Q}$ and $cv_i = \bar{C}$,

$$\sum_{i=1}^{n}(q_i - c_i)^2 \geq n(\bar{Q} - \bar{C})^2 \tag{5.10}$$

Substituting q_i by $\bar{Q} - \Delta q_i$ and c_i by $\bar{C} - \Delta c_i$,

$$\sum_{i=1}^{n}\big((\bar{Q} - \Delta q_i) - (\bar{C} - \Delta c_i)\big)^2 \geq n(\bar{Q} - \bar{C})^2 \tag{5.11}$$

Rearranging terms,

$$\sum_{i=1}^{n}\big((\bar{Q} - \bar{C}) - (\Delta q_i - \Delta c_i)\big)^2 \quad \geq \quad n(\bar{Q} - \bar{C})^2 \tag{5.12}$$

$$\sum_{i=1}^{n}\big((\bar{Q} - \bar{C})^2 - 2(\bar{Q} - \bar{C})(\Delta q_i - \Delta c_i) + (\Delta q_i - \Delta c_i)^2\big) \quad \geq \quad n(\bar{Q} - \bar{C})^2 \tag{5.13}$$

$$\sum_{i=1}^{n}(\bar{Q} - \bar{C})^2 - \sum_{i=1}^{n}2(\bar{Q} - \bar{C})(\Delta q_i - \Delta c_i) + \sum_{i=1}^{n}(\Delta q_i - \Delta c_i)^2 \quad \geq \quad n(\bar{Q} - \bar{C})^2 \tag{5.14}$$

$$n(\bar{Q} - \bar{C})^2 - 2(\bar{Q} - \bar{C})\sum_{i=1}^{n}(\Delta q_i - \Delta c_i) + \sum_{i=1}^{n}(\Delta q_i - \Delta c_i)^2 \quad \geq \quad n(\bar{Q} - \bar{C})^2 \tag{5.15}$$

$$n(\bar{Q} - \bar{C})^2 - 2(\bar{Q} - \bar{C})(\sum_{i=1}^{n}\Delta q_i - \sum_{i=1}^{n}\Delta c_i) + \sum_{i=1}^{n}(\Delta q_i - \Delta c_i)^2 \quad \geq \quad n(\bar{Q} - \bar{C})^2 \tag{5.16}$$

Since $\sum \Delta w_i = 0$,

$$n(\bar{Q} - \bar{C})^2 - 2(\bar{Q} - \bar{C})(0 - 0) + \sum_{i=1}^{n}(\Delta q_i - \Delta c_i)^2 \quad \geq \quad n(\bar{Q} - \bar{C})^2 \tag{5.17}$$

$$n(\bar{Q} - \bar{C})^2 + \sum_{i=1}^{n}(\Delta q_i - \Delta c_i)^2 \quad \geq \quad n(\bar{Q} - \bar{C})^2 \tag{5.18}$$

$$\sum_{i=1}^{n}(\Delta q_i - \Delta c_i)^2 \quad \geq \quad 0 \tag{5.19}$$

The sum of squares must be nonnegative, so our assumption was true. Hence the proof. ■

```
Algorithm ExactKNNSearch(Q,K)
Variable *queue*: MinPriorityQueue;
Variable *temp*: List;

1.  queue.push(root_node_of_index, 0);
2.  while not queue.IsEmpty() do
3.      top = queue.Top();
4.      for each time series C in temp such that D(Q,C) $\leq$ top.dist
5.          Remove C from temp;
6.          Add C to $result$;
7.          if $|result|$ = K return $result$;
8.      queue.Pop();
9.      if top is an APCA point $C$
10.         Retrieve full time series C from database;
11.         temp.insert(C, D(Q,C));
12.     else if top is a leaf node
13.         for each data item $C$ in top
14.             queue.push($C$, $D_{LB}(Q', C)$);
15.     else // top is a non-leaf node
16.         for each child node U in top
17.             queue.push(U, MINDIST(Q,R)) // R is MBR associated with U
```

Table 5.6: K-NN algorithm to compute the exact K nearest neighbors of a query time series Q using a multidimensional index structure

5.4 Indexing the APCA representation

The APCA representation proposed in Section 5.3.1 defines a N-dimensional feature space ($N = 2M$). In other words, the proposed representation maps each time series C = $\{c_1, \ldots, c_n\}$ to a point $C = \{cv_1, cr_1, \ldots, cv_M, cr_M\}$ in a N-dimensional space. We refer to the N-dimensional space as the APCA space and the points in the APCA space as APCA points. In this section, we discuss how we can index the APCA points using a multidimensional index structure (e.g., R-tree) and use the index to answer range and K nearest neighbors (K-NN) queries efficiently. We will concentrate on K-NN queries in this section; range queries will be discussed briefly at the end of the section.

A K-NN query (Q, K) with query time series Q and desired number of neighbors K retrieves a set $\mathcal{C}$ of K time series such that for any two time series C $\in \mathcal{C}$, E $\notin \mathcal{C}$, D(Q, C) $\leq$ D(Q, E). The algorithm for answering K-NN queries using a multidimensional index structure is shown in Table 5.6. [1] The above

[1]In this chapter, we restrict our discussion to only feature-based index structures i.e. multidimensional index structures that recursively cluster points using minimum bounding rectangles (MBRs). Examples of such index structures are R-tree, X-tree and Hybrid Tree . Note that the MBR-based clustering can be logical i.e. the index structure need not store the MBRs physically as long as they can be derived from the physically stored information. For example, space partitioning index structures like the hB-tree and the Hybrid Tree store the partitioning information inside the index nodes as kd-trees [90, 23]. Since the MBRs can be derived from the kd-trees, the techniques discussed here are applicable to such index structures [23].

algorithm is an optimization on the GEMINI K-NN algorithm described in Table 5.3 and was proposed in [131]. Like the basic K-NN algorithm [121, 66], the algorithm uses a priority queue queue to navigate nodes/objects in the index in the increasing order of their distances from Q in the indexed (i.e., APCA) space. The distance of an object (i.e., APCA point) C from Q is defined by $D_{LB}(Q', C)$ (cf. Section 5.3.3) while the distance of a node U from Q is defined by the minimum distance MINDIST(Q,R) of the minimum bounding rectangle (MBR) R associated with U from Q (definition of MINDIST will be discussed later). Initially, we push the root node of the index into the *queue* (Line 1). Subsequently, the algorithm navigates the index by popping out the item from the top of queue at each step (Line 8). If the popped item is an APCA point C, we retrieve the original time series C from the database, compute its exact distance D(Q,C) from the query and insert it into a temporary list *temp* (Lines 9-11). If the popped item is a node of the index structure, we compute the distance of each of its children from Q and push them into *queue* (Lines 12-17). We move a time series C from *temp* to *result* only when we are sure that it is among the K nearest neighbors of Q, i.e., there exists no object E $\notin result$ such that D(Q,E) $<$ D(Q,C) and $|result| < K$. The second condition is ensured by the exit condition in Line 7. The first condition can be guaranteed as follows. Let $\mathcal{I}$ be the set of APCA points retrieved so far using the index (i.e., $\mathcal{I} = temp \cup result$). If we can guarantee that $\forall C \in \mathcal{I}, \forall E \notin \mathcal{I}, D_{LB}(Q', C) \leq$ D(Q,E), then the condition "D(Q,C) $\leq$ top.dist" in Line 4 would ensure that there exists no unexplored time series E such that D(Q, E) $<$ D(Q,C). By inserting the time series in $temp$ (i.e., already explored objects) into result in increasing order of their distances D(Q,C) (by keeping temp sorted by D(Q,C)), we can ensure that there exists no explored object E such that D(Q, E) $<$ D(Q,C). In other words, if $\forall C \in \mathcal{I}, \forall E \notin \mathcal{I}, D_{LB}(Q', C) \leq$ D(Q,E), the above algorithm would return the correct answer.

Before we can use the above algorithm, we need to describe how to compute MINDIST(Q,R) such that the correctness requirement is satisfied, i.e., $\forall C \in \mathcal{I}, \forall E \notin \mathcal{I}, D_{LB}(Q', C) \leq$ D(Q,E). We now discuss how the MBRs are computed and how to compute MINDIST(Q,R) based on the MBRs. We start by revisiting the traditional definition of an MBR [17]. Let us assume we have built an index of the APCA points by simply inserting the APCA points $C = \{cv_1, cr_1, \ldots, cv_M, cr_M\}$ into a MBR-based multidimensional index structure (using the insert function of the index structure). Let U be a leaf node of the above index. Let $R = (L, H)$ be the MBR associated with U where $L = \{l_1, l_2, \ldots, l_N\}$ and $H = \{h_1, h_2, \ldots, h_N\}$ are the lower and higher endpoints of the major diagonal of R. By definition, R is the smallest rectangle that spatially contains each APCA point $C = \{cv_1, cr_1, \ldots, cv_M, cr_M\}$ stored in U. Formally, $R = (L, H)$ is defined as:

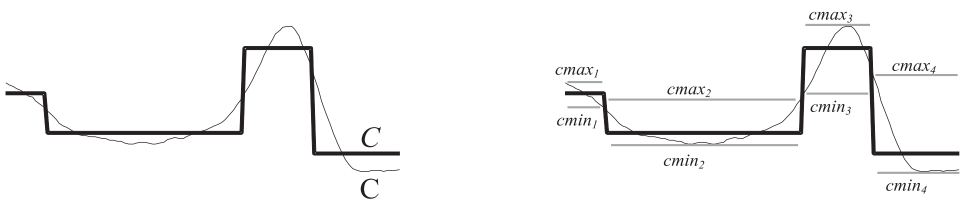


Figure 5.8: Definition of $cmax_i$ and $cmin_i$ for computing MBRs

Definition 4 (Old definition of MBR)

$$\begin{aligned} l_i &= min_{C \text{ in } U} cv_{(i+1)/2} \quad \text{if i is odd} && (5.20) \\ &= min_{C \text{ in } U} cr_{i/2} \quad \text{if i is even} && (5.21) \\ h_i &= max_{C \text{ in } U} cv_{(i+1)/2} \quad \text{if i is odd} && (5.22) \\ &= max_{C \text{ in } U} cr_{i/2} \quad \text{if i is even} && (5.23) \end{aligned}$$

■

The MBR associated with a non-leaf node would be the smallest rectangle that spatially contains the MBRs associated with its immediate children [59].

However, if we build the index as above (i.e., the MBRs are computed as in Definition 4), it is not possible to define a MINDIST(Q,R) that satisfies the correctness criteria. To overcome the problem, we define the MBRs are follows. Let us consider the MBR R of a leaf node U. For any APCA point $C = \{cv_1, cr_1, \ldots, cv_M, cr_M\}$ stored in node U, let $cmax_i$ and $cmin_i$ denote the maximum and minimum values of the corresponding time series C among the datapoints in the ith segment, i.e.,

$$\begin{aligned} cmax_i &= max_{t=cr_{i-1}+1}^{cr_i} c_t && (5.24) \\ cmin_i &= min_{t=cr_{i-1}+1}^{cr_i} c_t && (5.25) \\ & && (5.26) \end{aligned}$$

The $cmax_i$ and $cmin_i$ of a simple time series with 4 segments is shown in Figure 5.8. We define the MBR $R = (L, H)$ associated with U as follows:

Definition 5 (New definition of MBR)

$$\begin{aligned} l_i &= min_{C \text{ in } U} cmin_{(i+1)/2} \quad \text{if i is odd} && (5.27) \\ &= min_{C \text{ in } U} cr_{i/2} \quad \text{if i is even} && (5.28) \\ h_i &= max_{C \text{ in } U} cmax_{(i+1)/2} \quad \text{if i is odd} && (5.29) \\ &= max_{C \text{ in } U} cr_{i/2} \quad \text{if i is even} && (5.30) \end{aligned}$$

■

As before, the MBR associated with a non-leaf node is defined as the smallest rectangle that spatially contains the MBRs associated with its immediate children.

How do we build the index such that the MBRs satisfy Definition 5. We insert rectangles instead of the APCA points. In order to insert an APCA point $C = \{cv_1, cr_1, \ldots, cv_M, cr_M\}$, we insert a rectangle $\bar{C} = (\{cmin_1, cr_1, \ldots, cmin_M, cr_M\}, \{cmax_1, cr_1, \ldots, cmax_M, cr_M\})$ (i.e., $\{cmin_1, cr_1, \ldots, cmin_M, cr_M\}$ and $\{cmax_1, cr_1, \ldots, cmax_M, cr_M\}$) are the lower and higher endpoints of the major diagonal of $\bar{C}$) into the multidimensional index structure (using the insert function of the index structure). Since the insertion algorithm ensures that the MBR R of a leaf node U spatially contains all the $\bar{C}$'s stored in U, R satisfies definition 5. The same is true for MBRs associated with non-leaf nodes. Since we use one of the existing multidimensional index structures for this purpose, the storage organization of the nodes follows that of the index structure (e.g., $\langle MBR, child\_ptr \rangle$ array if R-tree is used, kd-tree if hybrid tree is used). For the leaf nodes, we need to store the cv_i's of each data point (in addition to the $cmax_i$'s, $cmin_i$'s and cr_i's) since they are needed to compute D_{LB} (Line 14 of the K-NN algorithm in Table 5.6). The index can be optimized (in terms of leaf node fanout) by not storing the $cmax_i$'s and $cmin_i$'s of the data points at the leaf nodes, i.e., just storing the cv_i's and cr_i's (a total of $2M$ numbers) per data point in addition to the tuple identifier. The reason is that the $cmax_i$'s and $cmin_i$'s are not required for computing D_{LB}, and hence are not used by the K-NN algorithm. They are needed just to compute the MBRs properly (according to definition 5) at the time of insertion. The only time they are needed later (after the time of insertion) is during the recomputation of the MBR of the leaf node containing the data point after a node split. The insert function of the index structure can be easily modified to fetch the $cmax_i$'s and $cmin_i$'s of the necessary data points from the database (using the tuple identifiers) on such occasions. The small extra cost of such fetches during node splits is worth the improvement in search performance due to higher leaf node fanout. We have applied this optimization in the index structure for our experiments but we believe the APCA index would work well even without this optimization.

Once we have built the index as above (i.e., the MBRs satisfy Definition 5), we define the minimum distance MINDIST(Q,R) of the MBR R associated with a node U of the index structure from the query time series Q. For correctness, $\forall C \in \mathcal{I}, \forall E \notin \mathcal{I}, D_{LB}(Q', C) \leq$ D(Q,E) (where $\mathcal{I}$ denotes the set of APCA points retrieved using the index at any stage of the algorithm). We show that the above correctness criteria is satisfied if MINDIST(Q,R) lower bounds the Euclidean distance D(Q,C) of Q from any time series C placed under U in the index. [2]

Lemma 6 *If MINDIST(Q,R) $\leq$ D(Q,C) for any time series C placed under U, the algorithm in Table 5.6 is correct, i.e., $\forall C \in \mathcal{I}, \forall E \notin \mathcal{I}, D_{LB}(Q', C) \leq$ D(Q,E) where $\mathcal{I}$ denotes the set of APCA points retrieved using the index at any stage of the algorithm.*

[2]Note that MINDIST (Q,R) does not have to lower bound the lower bounding distance $D_{LB}(Q, C)$ for any APCA point C under U; it just has to lower bound the Euclidean distance D(Q,C) for any time-series C under U.

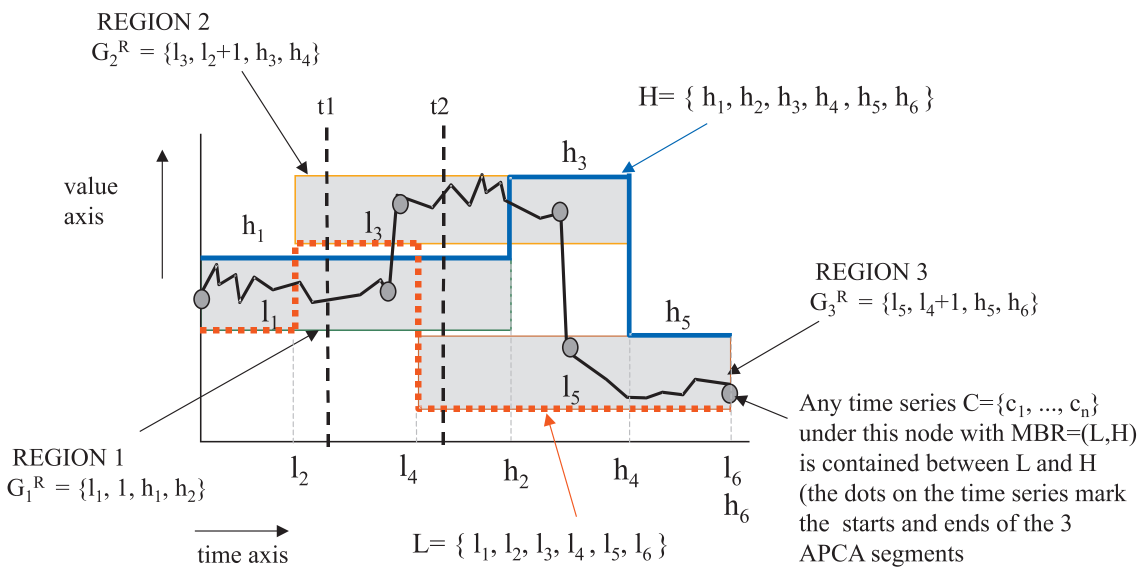

REGION 2
$G_2^R = \{l_3, l_2{+}1, h_3, h_4\}$
t1
t2
$H = \{h_1, h_2, h_3, h_4, h_5, h_6\}$
value axis
h_1
h_3
l_3
l_1
h_5
REGION 3
$G_3^R = \{l_5, l_4{+}1, h_5, h_6\}$
l_5
Any time series $C=\{c_1, ..., c_n\}$ under this node with MBR=(L,H) is contained between L and H (the dots on the time series mark the starts and ends of the 3 APCA segments
REGION 1
$G_1^R = \{l_1, 1, h_1, h_2\}$
l_2
l_4
h_2
h_4
l_6
h_6
time axis
$L = \{l_1, l_2, l_3, l_4, l_5, l_6\}$

Figure 5.9: The M Regions associated with a 2M-dimensional MBR. The boundary of a region G is denoted by G = G[1], G[2], G[3], G[4]

Proof: According to the K-NN algorithm, any item $E \notin \mathcal{I}$ must satisfy one of the following conditions:

1. E has been inserted into the queue but has not been popped yet, i.e., $\forall C \in \mathcal{I}$, $D_{LB}(Q', C) \leq D_{LB}(Q', E)$

2. E has not yet been inserted into the queue, i.e., there exists a parent node U of E whose MBR R satisfies the following condition: $\forall C \in \mathcal{I}$, $D_{LB}(Q', C) \leq$ MINDIST(Q,R).

Since $D_{LB}(Q', E) \leq$ D(Q,E) (Lemma 5), (1) implies $\forall C \in \mathcal{I}$, $D_{LB}(Q', C) \leq$ D(Q,E). If MINDIST(Q,R) $\leq$ D(Q,E) for any time series E under U, (2) implies that $\forall C \in \mathcal{I}$, $D_{LB}(Q', C) \leq$ D(Q,E). Since either (1) or (2) must be true for any item $E \notin \mathcal{I}$, $\forall C \in \mathcal{I}$, $D_{LB}(Q', C) \leq$ D(Q,E). ■

A trivial definition MINDIST(Q,R) that lower bounds D(Q,C) for any time series C under U is MINDIST(Q,R) = 0 for all Q and R. However, this definition is too conservative and would cause the K-NN algorithm to visit all nodes of the index structure before returning any answer (thus defeating the purpose of indexing). The larger the MINDIST, the more the number of nodes the K-NN algorithm can prune, the better the performance. We provide such a definition of MINDIST below[3].

Let us consider a node U with MBR $R = (L, H)$. We can view the MBR as two APCA representations $L = \{l_1, l_2, \ldots, l_N\}$ and $H = \{h_1, h_2, \ldots, h_N\}$. The view of a 6-dimensional MBR $(\{l_1, l_2, \ldots, l_6\}, \{h_1, h_2, \ldots, h_6\})$ as two APCA representations $\{l_1, l_2, \ldots, l_6\}$ and $\{h_1, h_2, \ldots, h_6\}$ is shown in Figure 5.9. Any time series $C = \{c_1, c_2, \ldots, c_n\}$ under the node U is "contained" within the two bounding time series L and H (as shown in Figure 5.9). In order to formalize this notion of containment, we define a set of M regions associated with R. The ith region G_i^R ($i = 1, \ldots, M$) associated with R is defined as the 2-dimensional

[3]Index structures can allow external applications to plug in domain-specific MINDIST functions and point-to-point distance functions and retrieve nearest neighbors based on those functions (e.g., Consistent function in GiST).

rectangular region in the value-time space that fully contains the ith segment of all time series stored under U. The boundary of a region G, being a 2-d rectangle, is defined by 4 numbers: the low bounds $G[1]$ and $G[2]$ and the high bounds $G[3]$ and $G[4]$ along the value and time axes respectively.

By definition,

$$\begin{aligned} G_i^R[1] &= min_{C \text{ under } U}(cmin_i) && (5.31)\\ G_i^R[2] &= min_{C \text{ under } U}(cr_{i-1}+1) && (5.32)\\ G_i^R[3] &= max_{C \text{ under } U}(cmax_i) && (5.33)\\ G_i^R[4] &= max_{C \text{ under } U}(cr_i) && (5.34) \end{aligned}$$

Based the definition of MBR in Definition 5, G_i^R can be defined in terms of the MBR R as follows:

Definition 6 (Definition of regions associated with MBR)

$$\begin{aligned} G_i^R[1] &= l_{2i-1} && (5.35)\\ G_i^R[2] &= l_{2i-2}+1 && (5.36)\\ G_i^R[3] &= h_{2i-1} && (5.37)\\ G_i^R[4] &= h_{2i} && (5.38) \end{aligned}$$

■

Figure 5.9 shows the 3 regions associated with the 6-dimensional MBR $(\{l_1, l_2, \ldots, l_6\}, \{h_1, h_2, \ldots, h_6\})$. At time instance t ($t = 1, \ldots, n$), we say a region G_i^R is active iff $G_i^R[2] \leq t \leq G_i^R[4]$. For example, in Figure 5.9, only regions 1 and 2 are active at time instant t_1 while regions 1, 2 and 3 are active at time instant t_2. The value c_t of a time series C under U at time instant t must lie within one of the regions active at t, i.e., $\vee_{G_i^R \text{ is active}} G_i^R[1] \leq c_t \leq G_i^R[3]$.

Lemma 7 *The value c_t of a time series C under U at time instant t must lie within one of the regions active at t.*

Proof: Let us consider a region G_i^R that is not active at time instant t, i.e., either $G_i^R[2] > t$ or $G_i^R[4] < t$. First, let us consider the case $G_i^R[2] > t$. By definition, $G_i^R[2] \leq cr_{i-1} + 1$ for any C under U . Since $G_i^R[2] > t$, $t < cr_{i-1} + 1$, i.e., c_t is not in segment i.

Now let us consider the case $G_i^R[4] < t$. By definition, $G_i^R[4] \geq cr_i$ for any C under U. Since $G_i^R[4] < t$, $t > cr_i$, i.e., c_t is not in segment i.

Hence, if region G_i^R is not active at t, c_t cannot lie in segment i, i.e., c_t can lie in segment i only if G_i^R is active. By definition of regions, c_t must lie within one of the regions active at t, i.e., $\vee_{G_i^R \text{ is active}} G_i^R[1] \leq c_t \leq G_i^R[3]$. ■

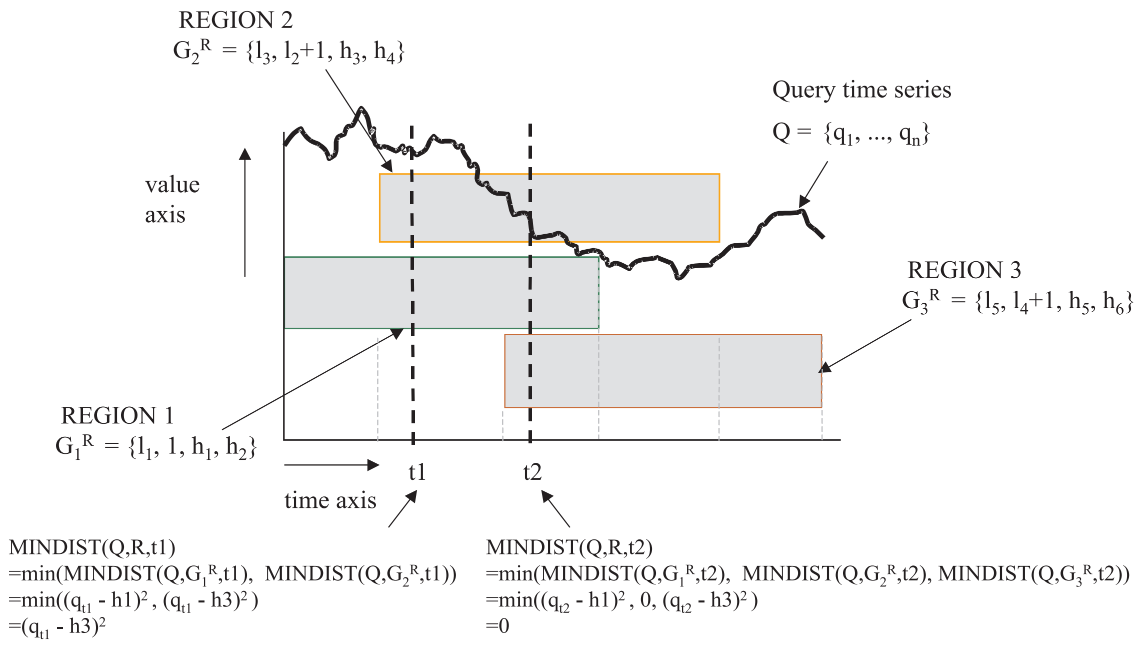

REGION 2
G_2^R = {l_3, l_2+1, h_3, h_4}
Query time series
Q = {q_1, ..., q_n}
value axis
REGION 3
G_3^R = {l_5, l_4+1, h_5, h_6}
REGION 1
G_1^R = {l_1, 1, h_1, h_2}
t1
t2
time axis
MINDIST(Q,R,t1)
=min(MINDIST(Q,G_1^R,t1), MINDIST(Q,G_2^R,t1))
=min((q_{t1} - h1)^2, (q_{t1} - h3)^2)
=(q_{t1} - h3)^2
MINDIST(Q,R,t2)
=min(MINDIST(Q,G_1^R,t2), MINDIST(Q,G_2^R,t2), MINDIST(Q,G_3^R,t2))
=min((q_{t2} - h1)^2, 0, (q_{t2} - h3)^2)
=0

Figure 5.10: Computation of MINDIST

Given a query time series $Q = \{q_1, q_2, \ldots, q_n\}$, the minimum distance MINDIST(Q,R,t) of Q from R at time instant t (cf. Figure 5.10) is given by MINDIST(Q,G,t) where

$$\begin{aligned} MINDIST(Q,G,t) &= (G[1]-q_t)^2 \text{ if } q_t < G[1] && (5.39)\\ &= (q_t - G[3])^2 \text{ if } G[3] < q_t && (5.40)\\ &= 0 \text{ otherwise}. && (5.41)\\ & && (5.42) \end{aligned}$$

MINDIST(Q,R) is defined as follows:

$$MINDIST(Q,R) = \sqrt{\sum_{t=1}^{n} MINDIST(Q,R,t)} \qquad (5.43)$$

Lemma 8 *MINDIST(Q,R) lower bounds D(Q,C) for any time series C under U.*

Proof: We will first show MINDIST(Q,R,t) lower bounds $D(Q,C,t) = (q_t - c_t)^2$ for any time series C under U. We know that c_t must lie in one of the active regions (Lemma 7). Without loss of generality, let us assume that c_t lies in an active region G, i.e., $G[1] \leq c_t \leq G[3]$. Hence $MINDIST(Q,G,t) \leq D(Q,C,t)$. Also, $MINDIST(Q,R,t) \leq MINDIST(Q,G,t)$ (by definition of $MINDIST(Q,R,t)$). Hence $MINDIST(Q,R,t)$ lower bounds $D(Q,C,t)$. Since $MINDIST(Q,R) = \sqrt{\sum_{t=1}^{n} MINDIST(Q,R,t)}$ and $D(Q,C) = \sqrt{\sum_{t=1}^{n} MINDIST(Q,C,t)}$, $MINDIST(Q,R,t) \leq D(Q,C,t)$ implies $MINDIST(Q,R) \leq$

```
Algorithm ExactRangeSearch(Q, $\epsilon$, T)

1.  if T is a non-leaf node
2.    for each child U of T
3.      if $MINDIST(Q,R) \leq \epsilon$ ExactRangeSearch(Q, $\epsilon$, U); // R is MBR of U
4.  else // T is a leaf node
5.    for each APCA point C in T
6.      if $D_{LB}(Q',C) \leq \epsilon$
7.        Retrieve full time series C from database;
8.        if D(Q,C) $\leq epsilon$ Add C to result;
```

Table 5.7: Range search algorithm to retrieve all the time series within a range of ϵ from query time series Q. The function is invoked as ExactRangeSearch(Q, ϵ, root_node_of_index).

$D(Q,C)$. ■

Note that, in general, lower the number of active regions at any instant of time, higher the MINDIST, better the performance of the K-NN algorithm. Also, narrower the regions along the value dimension, higher the MINDIST. The above two principles justify our choice of the dimensions of the APCA space. The odd dimensions help clustering APCA points with similar cvi's, thus keeping the regions narrow along the value dimension. The even dimensions help clustering APCA points that are approximately aligned at the segment end points, thus ensuring only one region (minimum possible) is active for most instants of time.

Although we have focussed on K-NN search in this section, the definitions of D_{LB} and MINDIST proposed in this chapter are also needed for answering range queries using a multidimensional index structure. The range search algorithm is shown in Table 5.7. It is a straightforward R-tree-style recursive search algorithm combined with the GEMINI range query algorithm shown in Table 5.2. Since both MINDIST(Q,R) and DLB(Q',C) lower bound D(Q,C), the above algorithm is correct [46].

5.5 Experimental Evaluation

In this section we will experimentally demonstrate the superiority of APCA in terms of query response time. We will also demonstrate that the APCA index can be built in reasonable time. For completeness we experimentally compare *all* the state of the art indexing techniques with our proposed method. We have taken great care to create high quality implementations of all competing techniques. For example we utilized the symmetric properties of the DFT as suggested in [119]. Additionally when taking the DFT of a real signal, the first imaginary coefficient is zero, and because all objects in our database have had their mean value subtracted, the first real coefficient is also zero. We do not include these constants in the index, making room instead for two additional coefficients that carry information. All other approaches are similarly optimized.

5.5.1 Experiment methodology

We performed all tests over a range of reduced dimensionalities (N) and query lengths (i.e., original dimensionalities, n). Because we wanted to include the DWT in our experiments, we are limited to query lengths that are an integer power of two. We consider a length of 1024 to be the longest query likely to be encountered (by analogy, one might query a text database with a word, a phrase or a complete sentence, but the would be little utility in a paragraph-length text query. A time series query of length 1024 corresponds approximately with sentence length text query). We tested on two datasets, one chosen because it is very heterogeneous and one chosen because it is very homogeneous.

- **Homogeneous Data:Electrocardiogram**: This dataset is taken from the MIT Research Resource for Complex Physiologic Signals [32]. It is a *"relatively clean and uncomplicated"* electrocardiogram. The total size of the data is 100,000 objects.

- **Heterogeneous Data: Mixed Bag**: This dataset we created by combining 7 datasets with widely varying properties of shape, structure, noise etc. The only preprocessing performed was to insure that each time series had a mean of zero and a standard deviation of one (otherwise many queries become pathologically easy). The 7 datasets are, Space Shuttle STS-57 [80], Arrhythmia [97], Random Walk [79, 153], INTERBALL Plasma processes (Figure 5.4) [134], Astrophysical data (Figure 5.1) [147], Pseudo Periodic Synthetic Time Series [10], Exchange rate (Figure 5.4) [147]. The total size of the data is 100,000 objects.

To perform realistic testing we need queries that do not have exact matches in the database but have similar properties of shape, structure, spectral signature, variance etc. To achieve this we used cross validation. We removed 10% of the dataset, and build the index with the remaining 90%. The queries are then randomly taken from the withheld subsection. For each result reported for a particular dimensionality and query length, we averaged the results of 50 experiments.

For simplicity we only show results for nearest neighbor queries, however we obtained similar results for range queries.

5.5.2 Experimental results: Computing the dimensionality reduced representation

We begin our experiments by measuring the time taken to compute the reduced dimensionality representation for each of the suggested approaches. We did this for query lengths from 32 to 1024 and database sizes of 40 to 640 kilobytes. [4] The relatively small databases were necessary to include SVD in the experiments. We used a Pentium PC 400 with 256 megs of ram. Experimental runs requiring more than 1,000 seconds were abandoned as indicated by the black-topped histogram bars in Figure 5.11.

[4]We wish to reemphasize that a small database is only used in this experiment. All other experiments use 100,000-item datasets.

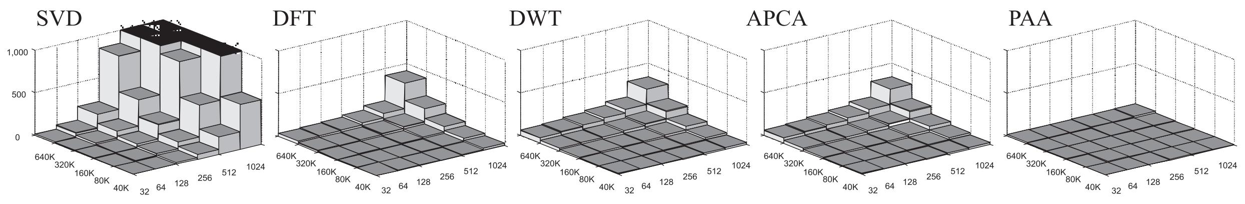


Figure 5.11: The time taken (in seconds) to build an index using various transformations over a range of query lengths and database sizes. The black topped histogram bars indicate that an experimental run was abandoned at 1,000 seconds.

We can see that SVD, being $O(Sn^2)$, is simply intractable for even moderately sized databases with long queries. We extrapolated from these experiments that it would take several months of CPU time to include SVD in all the experiments in this chapter. For this reason we shall exclude SVD from the rest of the experiments (in Section 5.8 we will discuss more reasons why SVD is not a practical approach). The results for DWT and APCA are virtually indistinguishable, which is to be expected given that the algorithm used to create the APCA spends most of its time in a subroutine call to the DWT. The main conclusion of this experiment is that APCA is tractable for indexing.

5.6 Experimental results: Pruning power

In comparing the four competing techniques there exists a danger of implementation bias. That is, consciously or unconsciously implementing the code such that some approach is favored. As an example of the potential for implementation bias in this work consider the following. At query time DFT must do a Fourier transform of the query. We could use the nave algorithm which is $O(n^2)$ or the faster radix-2 algorithm (padding the query with zeros for $n \neq 2^{integer}$) which is $O(nlogn)$. If we implemented the simple algorithm it would make the other indexing methods appear to perform better relative to DFT. While we do present detailed experimental evaluation of an implemented system in the next section, we also present experiments in this section which are free of the possibility of implementation basis. We achieve this by comparing the pruning power of the various approaches.

To compare the pruning power of the four techniques under consideration we measure P, the fraction of the database that must be examined before we can guarantee that we have found the nearest match to a 1-NN query.

$$P = \frac{\text{Number of objects that must be examined}}{\text{Number of objects in database}} \tag{5.44}$$

To calculate P we do the following. Random queries are generated (as described above). Objects in the database are examined in order of increasing (feature space) distance from the query until the distance in feature space of the next unexamined object is greater than minimum actual distance of the best match so far. The number of objects examined at this point is the absolute minimum in order to guarantee no false

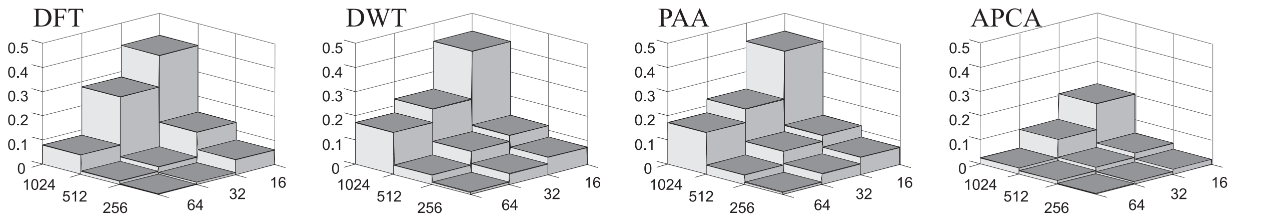


Figure 5.12: The fraction P, of the Mixed Bag database that must be examined by the four dimensionality reduction techniques being compared, over a range of query lengths (256-1024) and dimensionalities (16-64).

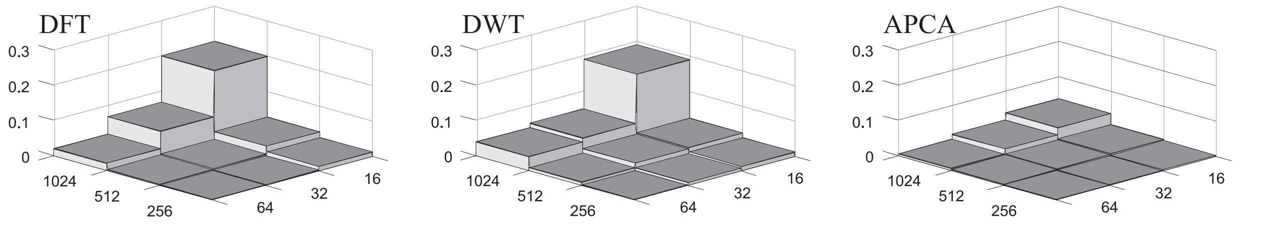


Figure 5.13: The fraction P, of the Electrocardiogram database that must be examined by the three dimensionality reduction techniques being compared over a range of query lengths (256-1024) and dimensionalities (16-64).

dismissals.

Note the value of P for any transformation depends only on the data and is completely independent of any implementation choices, including spatial access method, page size, computer language or hardware. A similar idea for evaluating indexing schemes appears in [62].

Figure 5.12 shows the value of P over a range of query lengths and dimensionalities for the experiments that were conducted the Mixed Bag dataset.

Note that the results for PAA and DWT are identical. This because the pruning power of DWT and PAA are identical when $N = 2^{integer}$ [79]. Having empirically shown this fact which was proved in [79, 153], we have excluded PAA from future experiments for clarity.

We repeated the experiment for the Electrocardiogram data, the results are shown in Figure 5.13.

In both Figure 5.12 and Figure 5.13 we can see that APCA outperforms DFT and DWT significantly, generally by an order of magnitude. These experiments indicate that the APCA technique has fewer false alarms, hence lower query cost as confirmed by the experiments below.

5.7 Experimental results: Implemented system

Although the pruning power experiments are powerful predictors of the (relative) performance of indexing systems using the various dimensionality reduction schemes, we include a comparison of implemented systems for completeness. We implemented four indexing techniques: linear scan, DFT-index, DWT-index and APCA-index. We compare the four techniques in terms of the I/O and CPU costs incurred to retrieve the exact nearest neighbor of a query time series. All the experiments reported in this subsection were conducted on a Sun Ultra Enterprise 450 machine with 1 GB of physical memory and several GB of secondary storage, running Solaris 2.6.

Cost Measurements: We measured the I/O and CPU costs of the four techniques as follows:

1. *Linear Scan (LS)*: In this technique, we perform a simple linear scan on the original n-dimensional dataset and determine the exact nearest neighbor of the query. The I/O cost in terms of sequential disk accesses is $\frac{(S*(n*sizeof(float)+sizeof(id)))}{PageSize}$. Since $sizeof(id) \ll (n * sizeof(float))$, we will ignore the $sizeof(id)$ henceforth. Assuming sequential I/O is about 10 times faster than random I/O, the cost in terms of random accesses is $\frac{(S*sizeof(float)*n)}{(PageSize*10)}$. The CPU cost is the cost of computing the distance D(Q,C) of the query Q from each time series C $= \{c_1, \ldots, c_n\}$ in the database.

2. *DFT-index (DFT)*: In this technique, we reduce the dimensionality of the data from n to N using DFT and build an index on the reduced space using a multidimensional index structure. We use the hybrid tree as the index structure. The I/O cost of a query has two components: (1) the cost of accessing the nodes of the index structure and (2) the cost of accessing the pages to retrieve the full time series from the database for each indexed item retrieved (cf. Table 5.6). For the second component, we assume that a full time series access costs one random disk access. The total I/O cost (in terms of random disk accesses) is the number of index nodes accessed plus the number of indexed items retrieved by the K-NN algorithm before the algorithm stopped (i.e. before the distance of the next unexamined object in the indexed space is greater than the minimum of the actual distances of items retrieved so far). The CPU cost also has two components: (1) the CPU time (excluding the I/O wait) taken by the K-NN algorithm to navigate the index and retrieve the indexed items and (2) the CPU time to compute the exact distance D(Q,C) of the query Q from the original time series C of each indexed item C retrieved (Line 11 in Table 5.6). The total CPU cost is the sum of the two costs.

3. *DWT-index (DWT)*: In this technique, we reduce the dimensionality of the data from n to N using DWT and build the index on the reduced space using the hybrid tree index structure. The I/O and CPU costs are computed in the same way as in DFT.

4. *APCA-index (APCA)*: In this technique, we reduce the dimensionality of the data from n to N using APCA and build the index on the reduced space using the hybrid tree index structure. The I/O and CPU costs are computed in the same way as in DFT and DWT.

We chose the hybrid tree as the index structure for our experiments since it is a space partitioning index structure ("dimensionality-independent" fanout) and has been shown to scale to high dimensionalities [23, 79, 118]. Since we had access to the source code of the index structure (`http://www-db.ics.uci.edu`), we implemented the optimization discussed in Section 5.4 (i.e., to increase leaf node fanout) for our experiments. We used a page size of 4KB for all our experiments.

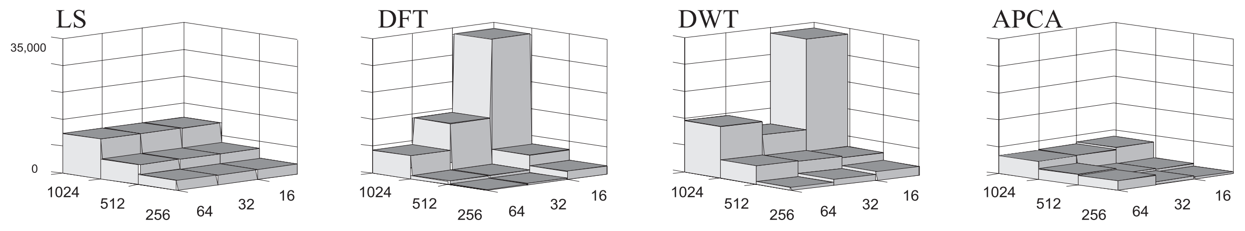


Figure 5.14: Comparison of LS, DFT, DWT and APCA techniques in terms of I/O cost (number of random disk accesses). For LS, the cost is computed as $\frac{number\_sequential\_disk_{a}ccesses}{10}$.

Dataset: We used the Electrocardiogram (ECG) database for these experiments. We created 3 datasets from the ECG database by choosing 3 different values of query length n (256, 512 and 1024). For each dataset, we reduced the dimensionality to $N = 16$, $N = 32$ and $N = 64$ using each of the 3 dimensionality reduction techniques (DFT, DWT and APCA) and built the hybrid tree indices on the reduced spaces (resulting a total of 9 indices for each technique). As mentioned before, the queries were chosen randomly from the withheld section of the dataset. All our measurements are averaged over 50 queries.

Figure 5.14 compares the LS, DFT, DWT and APCA techniques in terms of I/O cost (measured by the number of random disk accesses) for the 3 datasets ($n =$ 256, 512 and 1024) and 3 different dimensionalities of the index ($N =$ 16, 32 and 64). The APCA technique significantly outperforms the other 3 techniques in terms of I/O cost. The LS technique suffers due to the large database size (e.g., 100,000 sequential disk accesses for n = 1024 which is equivalent to 10,000 random disk accesses). Although LS is not considerably worse than APCA in terms of I/O cost, it is significantly worse in terms of the overall cost due to its high CPU cost component (see Figure 5.15). The DFT and DWT suffer mainly due to low pruning power (cf. Figure 5.13). Since DFT and DWT retrieve a large number of indexed items before it can guaranteed that the exact nearest neighbor is among the retrieved items, the second component of the I/O cost (that of retrieving full time series from the database) tends to be high. The DFT and DWT costs are the highest for large n and small N (e.g., $n = 1024$, $N = 16$) as the pruning power is the lowest for those values (cf. Figure 5.13). The DWT technique shows a U-shaped curve for $n = 1024$: when the reduced dimensionality is low ($N = 16$), the second component of the I/O cost is high due to low pruning power, while when N is high (N = 64), the first component of the I/O cost (index node accesses) becomes large due to dimensionality curse. We did not observe such U-shaped behavior in the other techniques as their costs were either dominated entirely by the first component (e.g., $n = 256$ and $n = 512$ cases of APCA) or by the second component (all of DFT and $n = 1024$ case of APCA).

Figure 5.15 compares the LS, DFT, DWT and APCA techniques in terms of CPU cost (measured in seconds) for the 3 datasets ($n =$ 256, 512 and 1024) and 3 different dimensionalities of the index ($N =$ 16, 32 and 64). Once again, the APCA technique significantly outperforms the other 3 techniques in terms of CPU cost. The LS technique is the worst in terms of CPU cost as it computes the exact (n-dimensional) distance D(Q,C) of the query Q from every time series C in the database. The DFT and DWT techniques suffer again due to their low pruning power (cf. Figure 5.13), causing the second component of the CPU

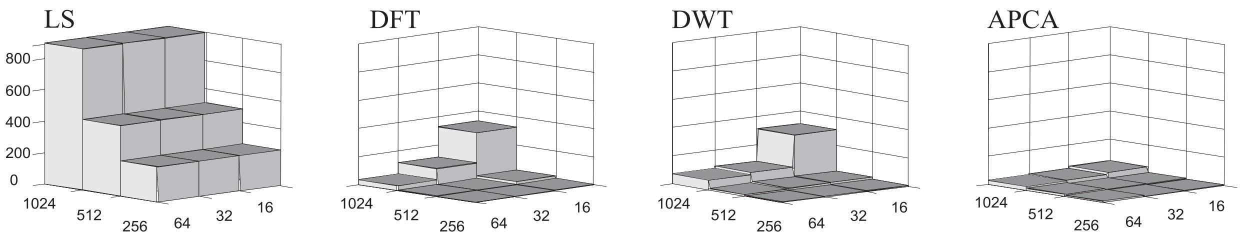


Figure 5.15: Comparison of LS, DFT, DWT and APCA techniques in terms of CPU cost (seconds).

cost (i.e. the time to compute the exact distances D(Q,C) of the original time series of the retrieved APCA points from the query) to become high.

5.8 Discussion

Now that the reader is more familiar with the contribution of this chapter we will briefly revisit related work. We believe that this thesis is the first to suggest locally adaptive indexing time series indexing. A locally adaptive representation for 2-dimensional shapes was suggested in [26] but no indexing technique was proposed. Also in the context of images, it was noted by [152] that the use of the first N Fourier coefficients does not guarantee the optimal pruning power. They introduced a technique where they adaptively choose which coefficients to keep after looking at the data. However, the choice of coefficients was based upon a global view of the data. Later work [151] in the context of time series noted that the policy of using the first N wavelet coefficients [29, 151] is not generally optimal, but *"keeping the largest coefficients needs additional indexing space and (more complex) indexing structures"*. Singular value decomposition is also a data adaptive technique used for time series [79, 81, 76], but it is globally, not locally, adaptive. Recent work [24] has suggested first clustering a multi-dimensional space and then doing SVD on local clusters, making it a semi-local approach. It is not clear however that this approach can be made work for time series. Finally a representation similar to APCA was introduced in [45] (under the name "piecewise flat approximation") but no indexing technique was suggested.

5.8.1 Other factors in choosing a representation to support indexing.

Although we have experimentally demonstrated that the APCA representation is superior to other approaches in terms of query response time, there are other factors which one may wish to consider when choosing a representation to support indexing. We will briefly consider some of these issues here.

One important issue is the length of queries allowed. For example the wavelet approach only allows queries with lengths that are an integer power of two [79]. This problem could be addressed by having the system pad in zeros up to the next power of two, then filter out the additional false hits. However this will severely degrade performance. The APCA approach, in contrast, allows arbitrary length queries.

Another important point to consider are the set of distance measures supported by a representation. It has been argued that for many applications, distance measures other than Euclidean distance are required. For example in [153], the authors noted that the PAA representation can support queries where the distance measure is an arbitrary Lp norm (i.e., $p = 1, 2, \ldots, \infty$). We refer the interested reader to that paper for a discussion of the utility of these distance metrics, but note that the APCA representation can easily handle such queries by trivial generalizations of Equation 5.5 to Equation 5.45.

$$D_{LB}(Q', C) = \sqrt[p]{\sum_{i=1}^{M} cl_i (q_i - c_i)^p} \tag{5.45}$$

Note that as with the approach of [153] we can reuse the same index for any Lp norm.

Almost all time series databases are dynamic. For example, NASA updates its archive of Space Shuttle telemetry data after each mission. Some databases are updated continuously, for example financial datasets are updated (at least) at the end of each business day. It is therefore important that any indexing technique be able to support dynamic inserts. Our proposed approach (along with DWT, DFT and PAA) has this property. However dynamic insertion is the Achilles heel of SVD, a single insertion requires recomputing the entire index. Faster methods do exist for incremental updates, but they introduce the possibility of false dismissals [30].

5.9 Conclusions

The main contribution of this chapter is to show that a simple, novel dimensionality reduction technique, namely APCA, can outperform more sophisticated transforms by one to two orders of magnitude. In contrast to popular belief [153, 45], we have shown that the APCA representation can be indexed using a multidimensional index structure. We have also shown that our approach can support arbitrary Lp norms, using the same index.

So far in this thesis, we have focussed on developing index structures and dimensionality reduction techniques to handle high dimensional data. In the next chapter, we address the challenge of integration of multidimensional index structures as access methods in a DBMS. One of the main issues there is providing transactional access to data via multidimensional index structures. We develop efficient concurrency control techniques for multidimensional access methods in the next chapter.

Chapter 6

Integration of Multidimensional Index Structures into DBMSs

In this chapter, we develop efficient concurrency control techniques for multidimensional access methods. This is one of the key challenges in integrating multidimensional index structures as access methods in a DBMS.

6.1 Introduction

Modern database applications like computer-aided design (CAD), geographical information systems (GIS), multimedia retrieval systems etc., require database systems to allow the application developer (1) define their own data types and operations on those data types, and (2) define their own indexing mechanisms on the stored data which the database query optimizer can exploit to access the data efficiently. While object relational DBMS (ORDBMSs) have addressed the first problem effectively [139], the ability to allow application developers to easily define their own access methods (AMs) still remains an elusive goal.

The *Generalized Search Tree* (GiST) [63] addresses the above problem. GiST is an index structure that is extensible "both" in the data types it can index and in the queries it can support. It is like a "template" – the application developer can implement her own AM using GiST by simply registering a few extension methods with the DBMS. GiST solves two problems:

- Over the last few years, several multidimensional data structures have been developed for specific application domains. Implementing these data structures from scratch every time requires a significant coding effort. GiST can be adapted to work like these data structures, a much easier task than implementing the tree package from scratch.
- Since GiST is extensible, if it is supported in a DBMS, the DBMS can allow application developers to define their own AM, a task that was not possible before.

Although GiST considerably reduces the effort of integrating new AMs in DBMSs, before it can be supported in a "commercial strength" DBMS, efficient techniques to support concurrent access to data via the GiST must be developed. Developing concurrency control (CC) techniques for GiST have several im-

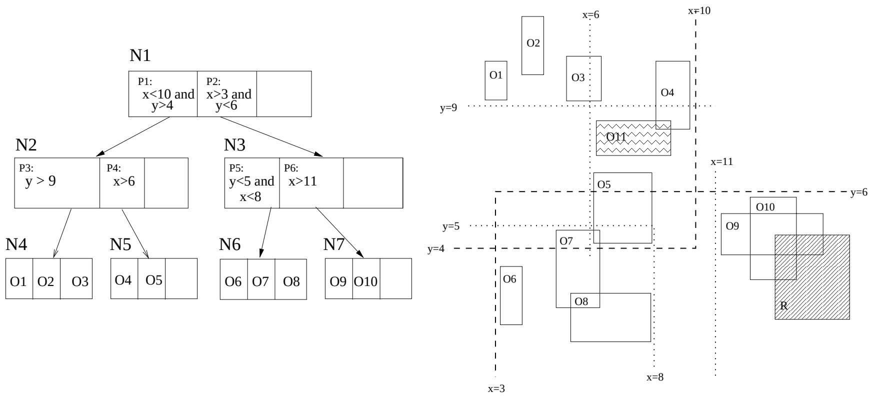


Figure 6.1: A GiST for a key set comprising of rectangles in 2 dimensional space. $O11$ is a new object being inserted in node $N5$. R is a search region. Predicates $P1$ through $P6$ are the BPs of the nodes $N2$ through $N7$ respectively.

portant benefits. (1) Since a wide variety of index structures can be implemented using GiST, developing CC techniques in the context of GiST would solve the CC problem for multidimensional index structures in general. (2) Experience with B-trees has shown that the implementation of CC protocols requires writing complex code and accounts for a major fraction of the effort for the AM implementation [55]. Developing the protocols for GiST is particularly beneficial since it would need writing the code *only once* and would allow concurrent access to the database via *any* index structure implemented in the DBMS using GiST, thus avoiding the need to write the code for each index structure separately.

Concurrent access to data via a general index structure introduces two independent concurrency control problems:

- *Preserving consistency of the data structure* in presence of concurrent insertions, deletions and updates.
- *Protecting search regions from phantoms*

This chapter addresses the problem of phantom protection in GiSTs. In our previous research, we had studied a granular locking (GL) solution for phantom protection in R-trees [28]. We refer to it as the *GL/R-tree* protocol. Due to fundamental differences between R-tree and GiST in the notion of a search key, the approach developed for R-trees is not a feasible solution for GiST. Specifically, the GL/R-tree protocol needs several modifications for making it applicable to GiSTs and the modified algorithms, when applied to GiSTs, impose a significant overhead, both in terms of disk I/O as well as computational cost, on the tree operations. To overcome this problem, we develop a new granular locking approach for phantom protection in GiSTs in this chapter. We refer to it as the *GL/GiST* protocol. The GL/GiST protocol differs from the GL/R-tree protocol in its strategy to partition the predicate space and hence defines a new set of lockable resource granules. Based on the set of granules defined, lock protocols are developed for the various operations on GiSTs. For an R-tree implemented using GiST, GL/GiST protocol provides similar performance as the GL/R-tree protocol. On the other hand, for index structures where the search keys do not satisfy the *"containment hierarchy"* constraint, the GL/GiST protocol performs significantly better than the GL/R-tree protocol. Examples of such index structures include distance-based (centroid-radius based) index structures (e.g., M-tree, SS-tree). In summary, GL/GiST provides a *general solution* to concurrency control in multidimensional AMs rather than a specific solution for a particular index structure (e.g., GL/R-tree), without any compromise in performance.

The problem of phantom protection in GiSTs has previously been addressed in [83] where the authors develop a solution based on predicate locking (PL). As discussed in [55], although predicate locking offers potentially higher concurrency, typically granular locking is *preferred* since the lock overhead of predicate locking is much higher compared to that of granular locking. The reason is while granular locks can be set and cleared as efficiently as object locks ($\sim$ 200 RISC instructions), setting of a predicate lock requires checking for predicate satisfiability against the predicates of all concurrently executing transactions. For this reason, all existing commercial DBMSs implement granular locking in preference to the predicate based approach. Our experiments on various "real" multidimensional data sets show that (1) GL/GiST scales well under various system loads and (2) Similar to the B-tree case, GL provides a significantly more efficient implementation compared to PL for multidimensional AMs as well.

The rest of the chapter is developed as follows. Section 6.2 reviews the preliminaries. Section 6.3 describes the space partitioning strategy for GiSTs and discusses the difficulty in applying the R-tree approach to GiSTs. Section 6.4 presents the dynamic granular locking approach to phantom protection in GiSTs. The experimental results are presented in Section 6.5. Finally, Section 6.6 offers the conclusions.

6.2 Preliminaries

In this section, we first review the basic GiST structure. Next we describe the phantom problem, its solutions for B-trees and why they cannot be applied to multidimensional data structures. Finally, we state the desiderata of a granular locking solution to the phantom problem in multidimensional index structures followed by the terminology used in presenting the algorithms.

Generalized Search Trees GiST is a height balanced multiway tree. Each tree node contains a number of node entries, $E = \langle p, ptr \rangle$, where $E.p$ is a predicate that describes the subtree pointed by $E.ptr$. If N is the node pointed by $E.ptr$, $E.p$ is defined to be the bounding predicate (BP) of N, denoted by $BP(N)$. The BP of the root node is the entire key space S. Figure 6.1 shows a GiST for a key space comprising of 2-d rectangles.

A key in GiST can be any arbitrary predicate. The application developer can implement her own AM by specifying the key structure via a key class. The design of the key class involves providing a set of six extension methods which are used to implement the standard *search*, *insert* and *delete* operations over the AM. A more detailed description can be found in [63].

Lock Mode	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	
IX	✓	✓			
S	✓		✓		
SIX	✓				
X					

LOCK MODE	PURPOSE
S	Shared Access
X	Exclusive Access
IX	Intention to set shared or exclusive locks at finer granularity
IS	Intention to set shared locks at finer granularity
SIX	A course granularity shared lock with intention to set finer-granularity exclusive locks (union of S and IX)

Table 6.1: Lock mode compatibility matrix for granular locks. The purpose of the various lock modes are shown alongside.

Serializability Concepts and the Phantom Problem Transactions, locking and serializability concepts are well documented in the literature [112, 113, 55]. The phantom problem is defined as follows (from the ANSI/ISO SQL-92 specifications [93, 7]): Transaction T1 reads a set of data items satisfying some `<search condition>`. Transaction T2 then creates data items that satisfy T1's `<search condition>` and commits. If T1 then repeats its scan with the same `<search condition>`, it gets a set of data items (known as "phantoms") different from the first read. Phantoms must be prevented to guarantee serializable execution. Object level locking *does not* prevent phantoms since even if all objects currently in the database that satisfy the search predicate are locked, concurrent insertions into the search range cannot be prevented. These insertions may be a result of insertion of new objects, updates to existing objects or rolling-back deletions made by other concurrent transactions.

Approaches to Phantom Protection There are two general strategies to solve the phantom problem, namely *predicate locking* and its engineering approximation, *granular locking*. In predicate locking, transactions acquire locks on predicates rather than individual objects. Although predicate locking is a complete solution to the phantom problem, the cost of setting and clearing predicate locks can be high since (1) the predicates can be complex and hence checking for predicate satisfiability can be costly and (2) even if predicate satisfiability can be checked in constant time, the complexity of acquiring a predicate lock is proportional in the number of concurrent transactions which is an order of magnitude costlier compared to acquiring object locks that can be set and released in constant time [55]. In contrast, in granular locking, the predicate space is divided into a set of lockable resource granules. Transactions acquire locks on granules instead of on predicates. The locking protocol guarantees that if two transactions request conflicting mode locks on predicates p and p' such that $p \wedge p'$ is satisfiable, then the two transactions will request conflicting locks on at least one granule in common. Granular locks can be set and released as efficiently as object locks. For this reasons, all existing commercial DBMSs use granular locking in preference to predicate locking. A more detailed comparison between the two approaches can be found in [55].

An example of the granular locking approach is the *multi-granularity locking protocol* (MGL) [89]. MGL exploits additional lock modes called *intention* mode locks which represent the intention to set locks at finer granularity (see Table 6.1). Application of MGL to the key space associated with a B-tree is referred to as *key range locking*(KRL) [89, 95]. KRL cannot be applied for phantom protection in multidimensional data structures since it relies on the total order over the underlying objects based on their key values which does not exist for multidimensional data. Imposing an artificial total order (say a Z-order [108]) over multidimensional data to adapt KRL would result in a scheme with low concurrency and high lock overhead since protecting a multidimensional region query from phantom insertions and deletions will require accessing and locking objects which may not be in the region specified by the query (since an object will be accessed as long as it is within the upper and the lower bounds in the region according to the superimposed total order). It would severely limit the usefulness of the multidimensional AM, essentially reducing it to a 1-d AM with the dimension being the total order.

Desiderata of the Solution Since KRL cannot be used in multidimensional index structures, new techniques need to be devised to prevent phantoms in such data structures. The principal challenges in developing a solution based on granular locking are:

- *Defining a set of lockable resource granules* [1] over the multidimensional key space such that they (1) dynamically adapt to key distribution (2) fully cover the entire embedded space and (3) are fine enough to afford high concurrency. The importance of these factors in the choice of granules has been discussed in [55]. The lock granules (i.e. key ranges) in KRL satisfy these 3 criteria.
- *Easy mapping of a given predicate onto a set of granules* that needs to be locked to scan the predicate. Subsequently, the granular locks can be set or cleared as efficiently as object locks using a standard lock manager (LM).
- *Ensuring low lock overhead* for each operation.
- *Handling overlap among granules* effectively. This problem does not arise in KRL since the key ranges are always mutually disjoint. In multidimensional key space partitioning, the set of granules defined may be, in GiST terminology, "mutually consistent". For example, there may be spatial overlap among R-tree granules. This complicates the locking protocol since a lock on a granule may not provide an "exclusive coverage" on the entire space covered by the granule. For correctness, the granular locking protocols must guarantee that any two conflicting operations will request conflicting locks on at least one granule in common. This implies that at least one of the conflicting operations must acquire locks on all granules that *overlap* with its predicate while the other must acquire conflicting locks on enough granules to fully *cover* its predicate [28]. This leads to two alternative strategies:
    - *Overlap-for-Search and Cover-for-Insert Strategy (OSCI)* in which the searchers acquire shared mode locks on all granules consistent with its search predicate whereas the inserters, deleters and updators acquire IX locks on a minimal set of granules sufficient to fully cover the object being inserted, deleted or updated.
    - *Cover-for-Search and Overlap-for-Insert Strategy (CSOI)* in which the searchers acquire shared mode locks on a minimal set of granules sufficient to fully cover its search predicate whereas the

[1] In this chapter, we use the term "granules" to mean lock units – resources that are locked to insure isolation and not in the sense of granules in "granule graph" of MGL [55]. This is discussed in further detail in Section 4.1.

inserters, deleters and updators acquire IX locks on all granules consistent with the object being inserted, deleted or updated.

While the former strategy favors the insert and delete operations by requiring them to do minimal tree traversal and disfavors the search operation by requiring them to traverse all consistent paths, the latter strategy does exactly the reverse. Intermediate strategies are also possible. For GL/GiST, we choose the OSCI strategy in preference to the rest. The OSCI strategy effectively does not impose *any* additional overhead on any operation as far as tree traversal is concerned since searchers in GiST anyway follow all consistent paths. The CSOI strategy may be better for index structures where inserters follow all overlapping paths and searchers follow only enough paths to cover its predicate. The R+-tree is an example of such an index structure [132]. We assume that the OSCI strategy is followed for all discussions in the rest of the chapter.

Terminology In developing the algorithms, we assume, as in [89], that a transaction may request the following types of operations on GiST: Search, Insert, Delete, ReadSingle, UpdateSingle and UpdateScan. In presenting the solution to the phantom problem, we describe the lock requirements of each of these and present the algorithms used to acquire the necessary locks. The lock protocols assumes the presence of a standard LM which supports all the MGL locks modes (as shown in Table 6.1) as well as conditional and unconditional lock options [96]. Furthermore, locks can be held for different durations, namely, instant, short and commit durations [96]. While describing the lock requirements of various operations for phantom protection, we assume the presence of some protocol for preserving the physical consistency of the tree structure in presence of concurrent operations. The lock protocol presented in this chapter guarantees phantom protection independent of the specific algorithm used to preserve tree consistency. In our implementation, we have combined the GL/GiST protocol with the latching protocol proposed in [83]. We do not describe the combined algorithms in this chapter due to space limitations but can be found in the longer version of this paper [27].

6.3 Why the R-tree protocol cannot be applied to GiSTs?

The most obvious solution to the phantom problem in GiSTs is to treat GiSTs as extensible R-trees and apply the GL/R-tree protocol we developed in [28] to GiSTs. In this section, we argue that GL/R-tree protocol is not a feasible solution for GiSTs. We first briefly review the approach developed for phantom protection in R-trees [28]. We do this for two main reasons: (1) it builds the context for the solution developed for GiSTs and (2) it enables us to illustrate why GL/R-tree cannot be applied to GiSTs. Subsequently, we define the resource granules in GiST. We conclude the section by discussing why GL/R-tree is inapplicable to GiSTs.

6.3.1 The R-tree granular locking protocol

In GL/R-tree, we define the following two types of lockable granules:

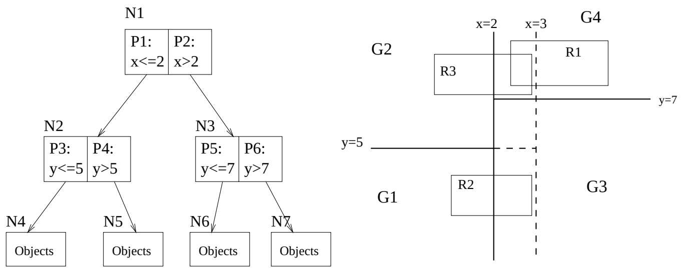


Figure 6.2: Insertion causes growth of tree granules that are outside the insertion path.

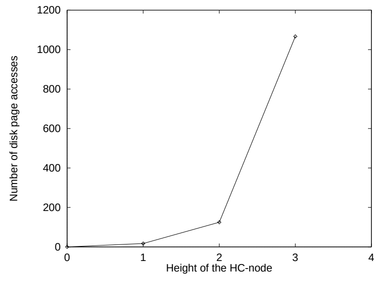


Figure 6.3: Increase of I/O overhead with the height of the HC-node

(1) A *leaf granule* associated with each leaf level index node L of the R-tree. We denote it by $TG(L)$ i.e. the tree granule associated with the leaf node L. The *bounding rectangle (BR)* associated with L defines the lock coverage of $TG(L)$.

(2) An *external granule* associated with each non-leaf node N of the R-tree. We denote it by $ext(N)$ i.e. the external granule associated with the non-leaf node N. The lock coverage of $ext(N)$ is defined to be the space covered by the BR of N which is not covered by the BRs of any of its children.

The search operation acquires locks on all leaf granules and external granules overlapping with the search predicate (referred to as SP/R-tree).

To prevent insertion of objects into search ranges of uncommitted searchers, we follow the OSCI policy. Although the plain OSCI policy guarantees phantom protection when the operations do not change the granules, phantoms may arise when the granule boundaries dynamically change due to insertions and deletions. To prevent phantoms, inserters in GL/R-tree follows the following protocol (referred to as IP/R-tree):

Let g be the granule corresponding to the leaf node in which the insertion takes place (referred to as the *target* granule) and O be the object being inserted. IP/R-tree handles the following 2 cases separately:

- *Case 1 - Insertion does not cause g to grow*: In this case, the inserter acquires (1) a commit duration IX lock on g and (2) a commit duration X lock on O.
- *Case 2 - Insertion causes g to grow (to say, g')*: In this case, it acquires (1) a commit duration IX lock on g (2) a commit duration X lock on O and (3) short duration IX locks on *all* granules into which it grew i.e. all granules overlapping with $(g' - g)$. (3) ensures that there exists no old searchers which could lose their lock coverage due to the growth of g. Note that acquiring the extra locks of (3) may cause the inserter to perform additional disk accesses.

A detailed discussion of the lock requirements for other tree operations and the protocols followed to acquire the locks can be found in [28].

6.3.2 Space partitioning strategy for GiSTs

The first task in developing a granular locking solution to the phantom problem is to develop a strategy to partition the key space. Note that the BPs in GiST, unlike the BRs in R-tree, *cannot* be used to define the granules since the BPs, unlike the BRs, are *not* arranged in a *"containment hierarchy"* i.e. given a node T, for any node N *under* (i.e. reachable from) T, $BP(N) \rightarrow BP(T)$ is *not* necessarily true. So, for a search with predicate P, there might exist a leaf (or external) granule that is consistent with the search predicate P *under* a non-leaf node N whose BP is not consistent with P. For example, in Figure 6.1, the search predicate R is not consistent with $BP(N2)$ (i.e. $P1$) but is consistent with $TG(N5)$ (i.e. $P4$) where $N5$ lies *under* $N2$ in the tree. This means that to follow the OSCI policy (i.e. get locks on all consistent granules), the searcher cannot "prune" its search below $N2$ as it would normally do. This is impractical since the searcher would have to access extra nodes (and possibly extra disk accesses) for the purpose of getting locks.

It is clear from the above discussion that we must define granules such that their lock coverage satisfy the "containment hierarchy" constraint even if the BPs do not. For that purpose, we define a *granule predicate* associated with every index node of a GiST.

Definition 1(Granule Predicate): Let N be an index node and P be the parent of N. The *granule predicate* of N, denoted by $GP(N)$, is defined as:

$$\begin{aligned} GP(N) &= BP(N) \text{ if } N \text{ is the root} && (6.1) \\ &= BP(N) \wedge GP(P) \text{ otherwise} && (6.2) \end{aligned}$$

Note that GPs, unlike BPs, are guaranteed to satisfy the "containment hierarchy" property.

Using GPs, we define the following two types of granules:

(1) A *leaf granule* $TG(L)$ associated with each leaf node L whose coverage is defined by GP(L). For example, in Figure 6.1, there are 4 leaf granules: TG(N4), TG(N5), TG(N6) and TG(N7) with lock coverage s lock coverage s $P1 \wedge P3$, $P1 \wedge P4$, $P2 \wedge P5$ and $P2 \wedge P6$ respectively

(2) An *external granule* $ext(N)$ associated with each non-leaf node N whose coverage defined as $(GP(N) \wedge \neg (\bigvee_{i=1}^{n} GP(Q_i)))$. where $Q_1, Q_2, ...Q_n$ are the children of N. For example, in Figure 6.1, there are 3 external granules: ext(N1), ext(N2) and ext(N3) will have lock coverages $S \wedge \neg(P1 \vee P2)$, $P1 \wedge \neg((P1 \wedge P3) \vee (P1 \wedge P4))$ and $P2 \wedge \neg((P2 \wedge P5) \vee (P2 \wedge P6))$ respectively.

Apart from the fact that the granules obey "containment hierarchy", the above definition has another motivation. In GiST, for any index node N, $BP(N)$ holds for each object in the subtree rooted at N. For example, in Figure 6.1, $P1$ holds for objects $O1, O2, O3, O4$ and $O5$ while both $P1$ and $P3$ holds for objects $O1$, $O2$ and $O3$. This implies that if an insertion does not change the BP of any node, it is guaranteed to be covered by the BP of each node in the path from the root to the leaf in which the object is being inserted. For example, in Figure 6.1, the object $O11$ (being inserted in node $N5$) is covered by both $P1$ and $P4$. So the leaf granule $TG(N5)$ should have lock coverage of $P1 \wedge P4$ since that is what the inserter needs for covering the object. This is exactly the definition of GP.

Having defined the new set of granules, we next try to apply GL/R-tree on GiST.

6.3.3 Problems in Applying GL/R-tree to GiSTs

Let us consider the GiST shown in Figure 6.2. There are 4 leaf granules $G1$, $G2$, $G3$ and $G4$ corresponding to nodes $N4, N5, N6$ and $N7$ with GPs $P1 \wedge P3$, $P1 \wedge P4$, $P2 \wedge P5$ and $P2 \wedge P6$ respectively. For simplicity, the partitioning of the space has been so chosen that all the external granules are empty.

Let t_s be a transaction searching region $R1$. Let t_{ins} be a new transaction that arrives to insert $R2$ into N4. After the insertion, t_{ins} updates $P1$ from $x \leq 2$ to $x \leq 3$. This causes t_s to lose it lock coverage. GL/R-tree prevents this by requiring t_{ins} to acquire locks on all granules which the target granule $G1$ has grown into. This is not sufficient for GiSTs since, unlike in R-trees, the target granule is *not* the only granule that can grow due to an insertion. For example, in Figure 6.2, both $G1$ and $G2$ grow due to the insertion. Assuming that only the target granule can grow can lead to phantoms. Under that assumption, t_{ins} would request a short duration IX lock on only $G3$ since that is the only granule into which $G1$ has grown, get the lock and commit. Now if t_{ins}^{new} arrives to insert $R3$ into $N2$, it would get the IX lock on $G2$ and proceed with insertion. Now if t_s repeats its scan, it would find $R3$ has arrived from nowhere. Growing of multiple leaf granules can happen in GiSTs because the lock coverage of the leaf granules, due to the definition of GP, depend of the BPs of the parents. So if an inserter modifies a node, the lock coverage of any granule under that node can possibly change. This is not possible in GL/R-tree since the lock coverage of a granule is independent of the BRs of its parent nodes.

To prevent phantoms, if the insertion changes any granule, it must acquire the following locks:
Let HC-node (Highest Changed Node) denote the the highest node in the insertion path path from root to leaf in which insertion takes place) whose BP (hence GP) changes due to the insertion. In Figure 6.2, $N2$ is the HC-node for the insertion of $R2$. Let G' be the new GP of HC-node after the insertion (e.g., $x \leq 3$ is the new GP of $N2$). Since *any* granule that grows due to the insertion is fully covered by G', short duration IX locks on *all* granules consistent with G' would ensure that no searcher loses its lock. In Figure 6.2, since all the 4 leaf granules are consistent with the predicate $x \leq 3$, t_{ins} would need to acquire short duration IX locks on $G2$, $G3$ and $G4$ in addition to the commit duration lock on $G1$ and X lock on $R2$. This would prevent t_{ins}(by the conflicting lock on $G4$) till t_s commits, thus preventing the phantom.

The above solution involves additional disk accesses to acquire those extra locks. In our experiments, we found that the number of disk accesses involved is significant and increases *exponentially* with the level of the HC-node. as shown in Figure 6.3. In general, the HC-node can be at any level of the GiST: all levels are equally likely. For the above experiment, performed on a 5-level GiST with fanout of about 100 and containing 400,000 2-d point objects, an insertion that causes a BP-change (about 6% of all insertions caused BP change) may need upto 1000 additional disk accesses to get all the locks (when the HC-node is at height 3 i.e. 3 levels above the leaf). This indicates that GL/R-tree can impose significant I/O cost for index structures where BPs do not obey "containment hierarchy" (e.g., distance-based index structures like M-tree).

Besides high cost, GL/R-tree has some other limitations for GiSTs: (1) It requires checking consistency with external granules during search, an extra task not performed by the regular GiST algorithm. This check can be computationally expensive in GiSTs. (2) It cannot allow an insertion or deletion to take place at an arbitrary level of the tree, a situation that can arise in GiSTs.

6.4 Phantom Protection in GiSTs

In this section, we present a dynamic granular locking approach to phantom protection in GiST. In the following subsections, we define the set of lockable resource granules for GiSTs and present lock protocols for various operations on GiSTs.

6.4.1 Resource granules in GiSTs

In GL/GiST, we define two types of granules:

(1) **Leaf granules:** This is the same as the previous GP-based definition of leaf granules. A leaf granule $TG(L)$ is associated with each leaf node L whose lock coverage is defined by GP(L).

(2) **Non-leaf granules:** This is a new set of granules. A *non-leaf granule* $TG(N)$ is associated with each non-leaf node N whose lock coverage, like leaf granules, is defined by $GP(N)$. In Figure 6.1, there are 3 non-leaf granules associated with the 3 non-leaf nodes $N1$, $N2$ and $N3$ with GPs S (entire key space), $P1$ and $P2$ respectively.

For both types of granules, the page ids of the index nodes are the resource ids used to lock the granules.

Thus, GL/GiST defines a different set of lock granules compared to those in the GL/R-tree protocol developed in [28]. External granules are no longer used as lockable granules. Non-leaf granules are used instead. There are several reasons for this choice: (1) it allows us to develop protocols that imposes absolutely no overhead (in terms of extra node accesses) on any tree operation (2) it causes almost no loss in concurrency since all commit duration locks held on non-leaf granules are *shared* mode locks (3) it has no extra computational cost since checking for consistency with non-leaf granules, unlike that with external granules, does not involve any extra checking other than what is performed anyway during the regular GiST search algorithm and (4) it allows the protocols to work even when insertions/deletions take place at arbitrary levels of the tree.

It is important to note that although non-leaf granules are introduced as lockable units, the GiST/GL protocol is completely different from and should not be confused with MGL. First, in MGL, the granules are hierarchically arranged to form a "granule graph" over which it follows the DAG protocol. In a granule graph, each node represents or "covers" a "logical" predicate. Since they are "logical", operations cannot dynamically change the predicate covered by any node in the graph. On the other hand, in GL/GiST, each node in a GiST represents a "physical" predicate: the GP of the node. Since GP is "physical" (i.e. defined based on the structure of the tree), operations (like insertions, deletions and updates) can dynamically change their lock coverages which complicates the protocol. Second, in MGL, a lock on a coarse (higher level) granule grants a certain lock coverage on the finer (lower level) granules under it. In GiST/GL, that is not

Algorithm Search(R, q, t)	
Input:	GiST rooted at R, predicate q, transaction t
Output:	All tuples that satisfy q
S1:	If R is root, request an S mode unconditional commit duration lock on R.
S2:	If R is non-leaf, check each entry E on R to determine whether Consistent(E,q). For each entry that is consistent, request an S mode unconditional commit duration lock on the node N referenced by $E.ptr$ and Search is invoked on the subtree rooted at N.
S3:	If R is a leaf, check each entry E on R to determine whether Consistent(E,q). If E is Consistent, it is a qualifying entry that can be returned to the calling process.

Table 6.2: Concurrent Search Algorithm

the case: the higher level (non-leaf) granules are introduced in order to *cover* the entire embedded space and a lock on *does not* grant coverage on any granule under it. In summary, DAG locking and GL/GiST are *fundamentally different* protocols and serve different purposes. We believe that the idea of defining lock granules associated with non-leaf nodes is novel and, to the best of our knowledge, has been discussed before only in the context of bulk insertions in B-trees as an open problem in [55].

6.4.2 Search

In this section, we describe the lock protocol followed by the search operation in GiST. According to the OSCI policy, a searcher with search predicate Q acquires commit duration S mode locks on all granules consistent with Q. The concurrent search algorithm is described is Table 6.2.

We refer to the above lock protocol as SP/GiST (Search Protocol for GiST). SP/GiST is a straightforward protocol and does not require any modification to the basic tree-navigation algorithm of GiST. This gives rise to a possible discrepancy. Like the regular GiST search algorithm, SP/GiST uses the BPs to do the "Consistency(E,q)" check during tree navigation. But the granules in GiST are defined in terms of the GPs. To show that SP/GiST is correct, we need to show that it guarantees that a searcher acquires locks on all the necessary granules i.e. for any index node T, *if* $GP(T) \wedge Q$ is satisfiable, *then* the searcher acquires an S lock on $TG(T)$.

To prove it, let us assume that $P_0, P_1, ..., P_m$ are the nodes in the path from the root to T where P_0 is the root and P_m is T. Since a searcher acquires a shared lock on $TG(T)$ iff it is consistent with with the BPs of all $P_i, i = [1, m]$, we need to prove that if $GP(T) \wedge Q$ is satisfiable, Q is consistent with the BP of $P_i, \forall i = [1, m]$. In other words, we need to prove that

$$GP(T) \wedge Q \text{ is satisfiable} \Rightarrow \bigwedge_{i=0}^{m} Consistent(BP(P_i), Q) \tag{6.3}$$

Using the definition of $GP(T)$,

$$GP(T) \wedge Q \text{ is satisfiable} \Leftrightarrow \left(\bigwedge_{i=1}^{m} BP(P_i)\right) \wedge Q\text{is satisfiable} \tag{6.4}$$

Since $\wedge$ is idempotent,

$$\left(\bigwedge_{i=1}^{m} BP(P_i)\right) \wedge Q \text{ is satisfiable} \Leftrightarrow \bigwedge_{i=1}^{m} (BP(P_i) \wedge Q)\text{is satisfiable} \tag{6.5}$$

Since $p \wedge q$ is satisfiable $\Rightarrow Consistent(p, q)$, so $\forall i, i = [1, m]$

$$(BP(P_i) \wedge Q) \text{ is satisfiable} \Rightarrow Consistent(BP(P_i), Q) \tag{6.6}$$

Since ($A \Rightarrow B \wedge C \Rightarrow D$) $\Rightarrow$ ($A \wedge C \Rightarrow C \wedge D$),

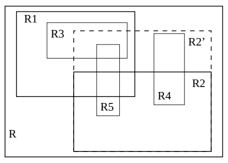
$$\bigwedge_{i=1}^{m} (BP(P_i) \wedge Q) \text{ is satisfiable} \Rightarrow \bigwedge_{i=0}^{m} Consistent(BP(P_i), Q) \tag{6.7}$$

Equations (4) and (7) together implies (3). ■

6.4.3 Insertion

The locking protocol for an insert operation must guarantee:

- *Full Coverage of the object being inserted till the time of transaction commit/rollback*: We say an object O being inserted (deleted) is fully covered by a set of granules $\mathcal{G}$ *iff* $O \Rightarrow \bigcup_{g \in \mathcal{G}} g$. An insertion (as well as a deletion or an update) operation must acquire commit duration IX locks on $\mathcal{G}$ such that $\mathcal{G}$ *fully covers* O. Full coverage guarantees that an insertion is permitted *only if* O does not conflict with the predicate of any uncommitted searcher *assuming* that each searcher hold commit duration locks on all consistent granules.
- *Prevent Phantoms due to Loss of Lock Coverage:* Since insertions (as well deletions and updates) can dynamically modify one or more granules which in turn can affect the lock coverage of transactions holding locks on other granules, full coverage is *not* sufficient to prevent phantoms. For example, the insertion of an object O into a leaf node L of a GiST may cause the granule $TG(L)$ to grow into the search range of an old uncommitted searcher, resulting in the searcher losing its lock. This loss of lock coverage may cause future insertions, in spite of satisfying the full coverage condition, giving rise to phantoms as illustrated in Figure 6.4. The insertion lock protocol must prevent such phantoms from arising.



1. t1 arrives to scan R3; acquires S lock on R1.
2. t2 arrives to insert R4; acquires IX locks on R2 and ext(R) and X lock on R4
3. R2 grows to R2'
4. t2 commits; releases all locks
5. t3 arrives to insert R5; acquires IX lock on R2' and X lock on R5
6. t1 repeats its scan; R4 has appeared from nowhere

Figure 6.4: Loss of lock coverage can cause phantoms.

To ensure full coverage and prevention of phantoms due to loss of lock coverage, the following protocol, referred to as IP/GiST (Insert Protocol for GiST), is used.

Let O be the object being inserted and g be the target granule. We consider the following two cases:

- *Case 1 - Insertion does not cause g to grow*: In this case, the inserter acquires (1) a commit duration IX lock on g and (2) a commit duration X lock on O.
- *Case 2 - Insertion causes g to grow*: Let LU-node (Lowest Unchanged Node) denote the lowest node in the insertion path whose GP does not change due to the insertion. For example, in Figure 6.2, $N1$ is the LU-node for the insertion operation of $R2$. The insertion acquires (1) a commit duration IX lock on g (2) a commit duration X lock on O and (3) a *short* duration IX lock on TG(LU-node).[2] For example, in Figure 6.2, t_{ins} would need to acquire a short duration IX lock on $TG(N1)$ in addition to the IX lock on $TG(N4)$ and X lock on $R2$.

The concurrent insert algorithm is described in Table 6.3.

IP/GiST is a simple and efficient protocol since it, unlike the IP/R-tree, does not impose *any* I/O or computational overhead on the insertion operation. As a result, IP/GiST is more efficient that IP/R-tree even on R-trees. Second, unlike IP/R-tree, IP/GiST works even if the target granule is a non-leaf granule i.e. when insertion takes place at a higher level of the tree.

Now we show that IP/GiST satisfy the above requirements of correctness. First, we prove full coverage. In Case 1, g fully covers O, so commit duration IX lock on g ensures full coverage. In Case 2, at the start of the operation, g does not fully cover O but TG(LU-node) does. So full coverage is provided by the sequence of 2 locks: (1) the short duration IX lock on TG(LU-node) from the beginning of the operation till the end of the operation [3] (2) the commit duration IX lock on g from the end of the operation till the end of the transaction (since g has already grown to accommodate O).

Next we show prevention of phantoms due to loss of lock coverage. In Case 1, there can be no loss of lock coverage of any searcher. In Case 2, the short duration IX lock on TG(LU-node) guarantees that no searcher can lose it lock coverage. Let us first consider a searcher t_s already executing when the inserter t_{ins} arrives to insert O. Let Q be the search predicate of t_s. Let h be a granule that grows to h' due to the insertion of O. t_s can lose its lock *iff* $h \wedge Q$ is not satisfiable but $h' \wedge Q$ is satisfiable. From the definition of LU-node, $h' \Rightarrow$ TG(LU-node). $(h' \wedge Q)$ is satisfiable *and* ($h' \Rightarrow$ TG(LU-node)) imply (TG(LU-node)

[2]The short duration IX lock can be released immediately if the AdjustKeys operation is performed right away i.e. in a top-down fashion rather than bottom-up as is done in GiSTs. This would avoid holding the lock across I/O operations.

[3]Note that this the best we can do since, at this point of time, TG(LU-node) is the *smallest* granule in the insertion path that *fully covers* O.

```
Algorithm Insert(R, E, l, t)
Input:      GiST rooted at R, entry E=(p, ptr) (where p is a predicate such that p holds for all tuples
            reachable from ptr), level l, transaction t.
Output:     New GiST resulting from insert of E at level l
Variables:  root is global variable (const) pointing to the root node of the GiST. $L$ is a lock initialized
            to NULL.
I1:         If R is not at level l, check all entries $E_i = (p_i, ptr_i)$ in R and evaluate Penalty($E_i$,E) for
            each $i$. Let $m$ be $argmin_i\ (Penalty(E_i, E))$. If ((L == NULL) ∧ (Union(E.p, $E_m.p_m$)
            $\neq E_m.p_m$)), request a unconditional IX mode lock $L$ on R (for short duration). Insert is
            invoked on the subtree rooted at the node referenced by $E_m.ptr_m$.
I2:         Otherwise (level of insertion reached), request a commit duration unconditional IX lock
            on $R$ and a commit duration unconditional X lock on $E.ptr$. If there is room for E on R,
            install E on R. Otherwise invoke Split(root, R, E, t).
I3:         AdjustKeys(root, R, t).
I4:         If $L \neq NULL$, release $L$.
```

Table 6.3: Concurrent Insert Algorithm

$\wedge Q$) is satisfiable which in turn implies *Consistent(TG(LU-node), Q)*. This means that t_s can lose it's lock coverage *iff* it has an S lock on TG(LU-node) (since searcher acquires S locks on all consistent granules). Thus, the IX lock requirement on TG(LU-node) prevents any searcher from losing its lock coverage. The IX lock on TG(LU-node), being a short duration lock, would prevent any loss of lock by even those searchers that arrive during the operation. Any searcher that arrives after the completion of the insertion operation cannot lose its lock coverage due to the insertion.

6.4.4 Node Split

We now consider the special case where the insertion by a transaction t into an already full node causes the target granule g to split into granules g_1 and g_2. Insertions causing node splits follow the IP/GiST except that it needs to acquire some additional locks when it causes the splits.

If the insertion by t causes g to split, since the IX lock held by t on g is lost after the split, t needs to acquire IX locks on g_1 and g_2 to protect the inserted object. Since t acquires an IX lock on g before the insertion, no other transaction, besides t itself, can be holding an S lock on g. If t itself holds an S lock on g, it needs to inherit its S lock on g to g_1 and g_2. This is because g_1 and g_2 are the only additional granules that may become consistent with the search predicate of t due to the split.

Since before the split the inserter acquires an IX lock on g, other inserters and deleters may also be holding IX locks on g. When g splits, all transactions holding IX locks on g must acquire IX locks on g_1 and g_2 after the split. This is sufficient as all the insert and/or delete ranges (logical deletion) is guaranteed to be protected by the IX locks on g_1 and g_2 since all objects in g will be either in g_1 or g_2. It may not possible for t to change lock requests of other transactions using a standard lock manager. The problem can be avoided if the inserter acquires a instant duration SIX lock on g in case it causes g to split. After the split, the inserter

Operation	Lock Requirements	Other Actions
Insertion(no granule change /no node split)	Commit dur. IX on g; Commit dur. X on O	None
Insertion (granule change)	Short dur. IX on TG(LU-node); IX on g; X on O	None
Insert (node split)	If T is leaf : Instant dur. SIX on $TG(T)$ before split; IX on either $TG(T)$ or $TG(TT)$, whichever contains O after split If T is non-leaf : Instant dur. SIX on $TG(T)$;	Inherit S locks to $TG(TT)$ if itself holding S lock on $TG(T)$
Search	S on all consistent leaf and non-leaf granules	None
Delete (Logical)	IX on g; X on O	Mark O deleted; Remove O from page
Delete (Deferred)	If node is not empty: Short dur. IX on TG(HC-node); IX on g; X on O. If becomes empty: If T is leaf, Short dur. SIX on TG(T); If T is non-leaf , Short dur. IX on TG(T)	Eliminate node if empty
ReadSingle	S on O	None
UpdateSingle	If no indexed attribute changed: IX on g; X on O Otherwise: Delete O; Insert modified O	None
UpdateScan	S on all consistent granules; For every individual object updated, same requirement as UpdateSingle	None

Table 6.4: Lock requirements for various operations in the dynamic granular locking approach. g is the target granule for insertion/deletion, O is the object being inserted/deleted/updated.

acquires a commit duration IX lock on either g_1 or g_2, whichever contains O.

The splitting of the granule may propagate upwards causing the non-leaf nodes to split. As in the case of leaf node split, the transaction causing a non-leaf node N to split acquires a instant duration SIX lock on $TG(N)$ to prevent any other transaction losing its lock. If t itself was holding an S lock on $TG(N)$, it needs to inherit its S lock on the two granules formed after split.

The node split operation can be allowed to be carried out "asynchronously". This requires maintaining the information of an "outstanding split" in the node - the transaction can subsequently commit while a separate transaction executes the split operation later by checking the "outstanding split" flags. The lock requirements remain the same as in the "synchronous" case.

6.4.5 Deletion

Similar to insertion, to delete an object O, the deleter requires an IX lock on the region that covers O. However, unlike insertion, (in which the granule where the object is inserted grows and covers the inserted object), the granule g from which O is deleted may shrink due to the deletion and may not cover O. To protect the delete region, the deleter would need a *commit duration* IX lock on TG(LU-node) (here it is the LU-node of the deletion of operation) since TG(LU-node) is the smallest granule to fully cover O at the completion of the deletion operation. This would result in low concurrency since a large number of searchers may be unnecessarily prevented till the deleter commits. For this reason, we do not consider this approach any further. Instead, deletes are performed logically. We present the lock needs of the logical and physical deletions in the following subsections.

Logical Deletion

The logical deleter needs to acquire a commit duration IX lock on only the leaf granule g that contains the object and an X lock on O itself. The IX lock on g is sufficient to cover O since even if the GP of g changes due to other insertions and deletions (physical) since g would still cover O. Subsequently, it removes the object from the page and marks it as deleted. If the transaction aborts, the changes are undone, the delete mark is removed and the locks are released. On the other hand, if it commits, the physical deletion of O from the GiST is executed as a separate operation.

If the transaction requests deletion of an object O that does not exist, other transactions wishing to insert the same object should be prevented as long as the deleter is active. For this purpose, the deleter acquires S locks on all consistent granules just like a search operation with O as the search predicate.

Deferred (Physical) Deletion

The deferred delete operation removes the logically deleted object from the GiST and adjusts the BPs of the ancestors. To physically delete an object from a granule g, a short duration IX lock on g is acquired to prevent other searchers having S locks on g from losing their lock coverage. The IX lock is sufficient as inserters and other deleters holding locks on g would not lose the necessary lock coverage even after g shrinks due to the physical deletion. Deletion of an entry from the node may also result in the node becoming empty in which case it is eliminated from the GiST. Since a node is eliminated only when it becomes empty, no transaction can lose its IX lock due to elimination of g as g does not cover any object. So the IX lock on g is sufficient even if the deletion causes the elimination of the node.

In either case, since the change of g may propagate upwards causing BPs of the ancestor nodes to change, the non-leaf granules associated with the ancestors may shrink. Since only searchers hold locks on non-leaf granules (inserters request only instant-duration locks), only searchers can lose their lock coverage due to this shrinkage. Note that only the searchers whose predicates are consistent with the HC-node (i.e. the highest index node in the deletion path whose BP changes due to the deletion) can lose lock coverage, possibly giving rise to phantoms. The loss of lock coverage of the searchers can be prevented by acquiring a short duration IX lock on TG(HC-node). Note that for insertion, it was the TG(LU-node) on which the short duration IX lock had to be acquired. The difference comes from the fact that insertion causes granules to grow while deletion causes them to shrink.

6.4.6 Other Operations

The locks needs for the other operations are:

Parameters	Meaning
MPL	multiprogramming level
Transaction Size	the number of operations per transaction
Write Probability	the fraction of operations in a transaction that are writes (i.e. inserts)
Query Size	the average selectivity of a search operation
External Think Time	mean time between transactions
Restart Delay	mean time after which an aborted transaction is restarted

Table 6.5: Workload Parameters

- The *ReadSingle* operation just acquires an S lock on the object.
- The *UpdateSingle* operation, if none of the attributes indexed by GiST are changed, just needs an IX lock on the granule containing the object and an X lock on the object. Otherwise, it first executes a deletion operation of the object to be updated followed by the insertion of the updated object obeying the respective lock protocols.
- The *UpdateScan* operation acquires S locks on all consistent granules just like a Search operation. For every individual object O updated, it requires the same locks as an UpdateSingle operation on O.

The lock requirements for the various operations is shown in the Table 6.4.

6.5 Experimental Evaluation

We performed several experiments to (1) evaluate the performance of the GL/GiST protocol under various degrees of system loads and (2) compare it with other protocols in terms of concurrency and lock overhead. In this section, we discuss our implementation of the protocols followed by the performance results.

6.5.1 Implementation

Implementation of the Protocols We implemented the complete GL/GiST protocol as described in this chapter. To evaluate the performance of the GL/GiST protocol, we also implemented the pure predicate locking (referred to as the *PurePL* protocol) to serve as the baseline case. In PurePL, each search operation checks its predicate against the objects of the insert/delete/update operations of all currently executing transactions. If there is any conflict, it blocks on that transaction by requesting an S lock on that transaction ID, assuming that every transaction acquires an X lock on its own ID when it starts up. Otherwise it proceeds with the search. Similarly, each insert/delete/update operation checks its object against the predicates of the search operations of all currently executing transactions and in case of a conflict, blocks on the conflicting transaction.

Construction of GiST We conducted our experiments on two different GiSTs constructed over the following two datasets:

- The **2-d dataset:** is the 2-d point data set of the Sequoia 2000 benchmark [138]. It contains locations(easting and northing values) of 62,556 California places extracted from the US Geological Survey's Geographic Names Information System (GNIS)). The points are geographically distributed over a 1046km by 1317km area.
- The **3-d dataset:** is derived from the FOURIER dataset [23]. The FOURIER dataset data set comprises of 1.2 million vectors of fourier coefficients produced by fourier transformation of polygons. We constructed the 3-d dataset by taking the first 3 fourier coefficients of each vector.

We set aside some points (by random choice) from the above data files for insertion into the GiST during the run of transactions. The searches to be executed during the run are generated by randomly choosing the query anchor from the data file and generating a bounding box by choosing a proper side length needed to obtain desired search selectivity. The set-aside points and the queries are stored in two separate files which are used by the workload generator.

We created the GiSTs by bulkloading the remaining points. The two GiSTs are described below:

- *2-d GiST:* constructed on 56,655 2-d points with 2K page size (fanout 102, 821 nodes). Since the size of the data set is small, we use a comparatively small page size to make the GiST of significant size.
- *3-d GiST:* constructed on 480,471 3-d points with 8K page size (fanout 292, 2360 nodes)

In both cases, we configured the GiST to behave as an R-tree by specifying the extension methods appropriately.

Workload Generator and the Lock Manager The workload generator (WG) generates a workload based on the input parameters shown in Table 6.5. The WG assigns some search operations (from the bounding box query file) and some insertion operations (from the set-aside point file) to each transaction. Each transaction executes as a separate thread. We use the Pthread library (Solaris 2.6 implementation) for creating and managing the threads [104]. One thread only executes one transaction: it is created at the beginning of the transaction and is terminated when the latter commits. The WG maintains the MPL at the specified value by using an array of flags (MPL number of them): when a thread finishes, it sets a flag. The main WG thread constantly polls on this array and when it detects the setting of a flag, it starts a new thread and assigns the next transaction to it. The thread waits for some time (external think time) and starts executing the transaction: it executes one operation after another on the GiST following the lock protocols. If any lock request returns an error (due to a deadlock or a timeout), the transaction aborts. If it aborts, it is re-executed within the same thread after a certain restart delay (each transaction remembers its constituent operations till it commits for possible re-execution). Our implementation of the WG consists of 3 main C++ classes (TransactionManager, Transaction and Operation). The TransactionManager class also maintains the global statistics of the run (e.g., throughput, conflict-ratio, number of locks acquired, number of aborts etc.) which are used to measure the performance of the various protocols. Although the other 4 simulation

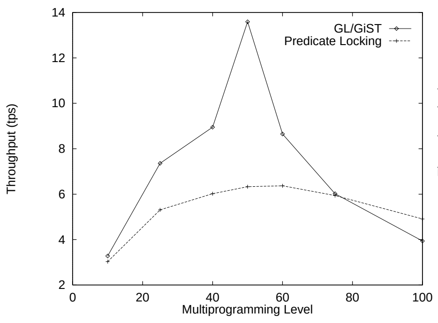


Figure 6.5: Throughput at various MPLs for 2-d data (write probability=0.2, transaction size=10, query selectivity=0.1%)

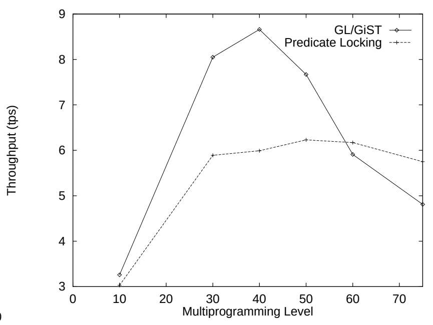


Figure 6.6: Throughput at various MPLs for 3-d data (write probability=0.2, transaction size=10, query selectivity=0.05%)

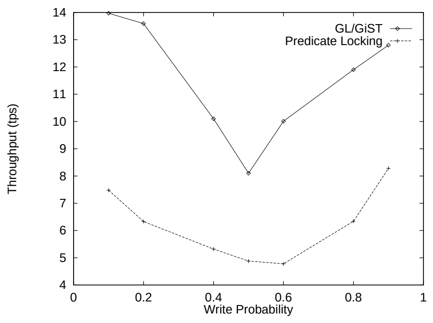


Figure 6.7: Throughput at various mixes of reads and writes (MPL=50, transaction size=10, query selectivity=0.1%)

parameters are varied, we fix the external think time to 3 seconds and the restart delay to 3 seconds for all our experiments. Also, for the two GiSTs, the buffer sizes are set such that about 75% of the pages fit in memory.

For the lock manager (LM) implementation, we reused most of the LM code of MiniRel system obtained from the University of Maryland. The LM code closely follows the description in [55].

All experiments were performed on a Sun Ultra Enterprise 3000 Server running Solaris 2.6 with two 167MHz CPU, 512MB of physical memory and several GB of secondary storage.

6.5.2 Experimental Results

Evaluation of the GL/GiST protocol We conducted experiments to evaluate the performance of the GL/GiST protocol under various system loads. Performance is measured using throughput i.e. the ratio of the total number of transactions that completed during the period when the transactions ran at full MPL (ignoring the starting phase and the dying phase when the MPLs are lower) to the total duration of the full-MPL phase [4]. Figures 6.5 shows the throughput of GL/GiST and PurePL protocols at various MPLs for the 2d dataset. Initially, the throughput increases with the MPL as the system resources were underutilized at low MPLs. For GL/GiST, the throughput reaches a peak (~ 14 tps) at an MPL of 50 while for PurePL, the peak ($\sim$ 6 tps) is reached at an MPL of 60. Beyond that point, the throughput starts decreasing as the system starts thrashing. Figures 6.6 shows the performance of the two protocols for the 3d dataset. Like the 2-d dataset, the GL/GiST achieves significantly higher throughput compared to PurePL.

We also varied the system load by tweaking the other parameters like write probability, transaction size and size of search [4]. These experiments were conducted on the 2-d dataset. Figure 6.7 shows the performance of the two protocols under various mixes of read(search) and write(insert) operations. GL/GiST significantly outperforms PurePL under all workloads. Figure 6.8 shows the throughputs at various transaction sizes. Again, GL/GiST mostly outperforms PurePL. At an MPL of 50, for transactions with 20 or more

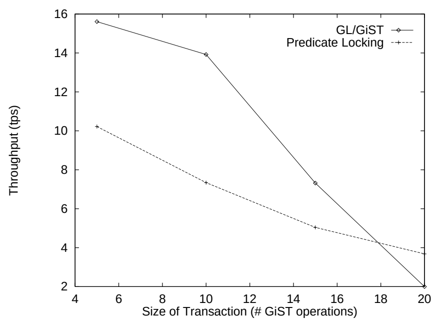


Figure 6.8: Throughput at various transaction sizes (MPL=50, write probability=0.1, query selectivity=0.1%)

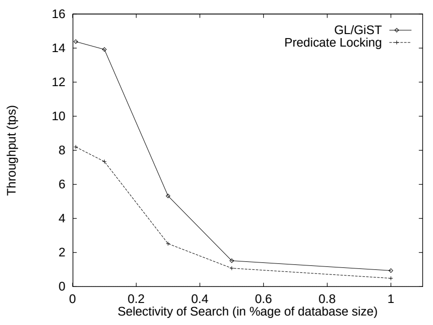


Figure 6.9: Throughput at various query sizes (MPL=50, transaction size=10, write probability=0.1)

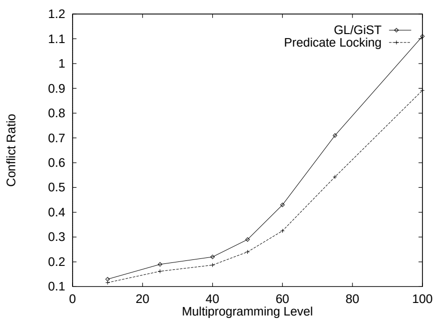


Figure 6.10: Conflict Ratio (transaction size=10, write probability=0.2, query selectivity=0.1%)

operations, since a large portion of the GiST is locked by some transaction or the other, GL/GiST starts thrashing due to high lock contention leading to decrease in throughput. Figure 6.9 shows the performance for various query sizes. Once again, GL/GiST performs better than PL for all workloads.

Comparison to other techniques In this section, we compare GL/GiST protocol with the predicate locking protocol presented in [83]. We refer to the above protocol as the PL/GiST protocol. In PL/GiST, a searcher attaches its search predicate Q to all the index nodes whose BPs are consistent with Q. Subsequently, the searcher acquires S locks on all objects consistent with Q. An inserter checks the object to be inserted against all the predicates attached to the node in which the insertion takes place. If it conflicts with any of them, the inserter also attaches its predicate to the node (to prevent starvation) and waits for the conflicting transactions to commit. If the insertion causes a BP of a node N to grow, the predicate attachments of the parent of N is checked with new BP of N and are replicated at N if necessary. The process is carried out top-down over the entire path where node BP adjustments take place. Similar predicate checking and replication is done between sibling nodes during split propagation. The details of the protocol can be found in [83]. A complete performance study would require a full fledged implementation of the PL/GiST protocol (including implementation of the Predicate Manager, augment GiST with data structures to be able to attach/detach predicates to tree nodes etc.). Due to the complexity of the this task, we only compare the two protocols in terms of the degrees of concurrency offered and their lock overheads. Again PurePL is used to serve as the baseline case. All the experiments were conducted on the 2-d dataset.

Figure 6.10 compares the concurrency offered by the GL/GiST and the PL protocols. Concurrency is measured using conflict ratio i.e. the average number of times some transaction blocked on a lock request per committed transaction [4]. Lower the conflict ratio, higher the concurrency. Both PL/GiST and PurePL protocols offer the maximum permissible concurrency since transactions are blocked only when they truly conflict. On the other hand, GL/GiST offers lower concurrency due to “false conflicts” i.e. a situation where although the predicates do not conflict with each other, they end up requesting conflicting locks on the same granule (e.g., in R-trees, a search predicate and an object being inserted do not overlap with each

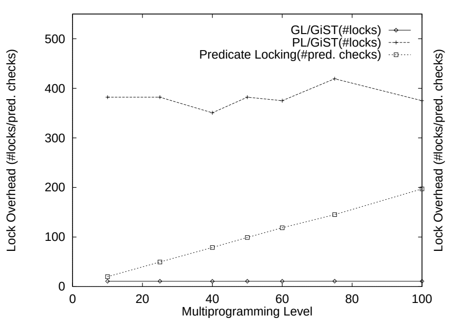


Figure 6.11: Lock Overhead of Search Operation (transaction size=10, write probability=0.2, query selectivity=0.1%)

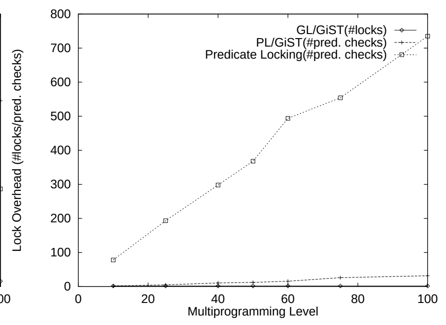


Figure 6.12: Lock Overhead of Insert Operation (transaction size=10, write probability=0.2, query selectivity=0.1%)

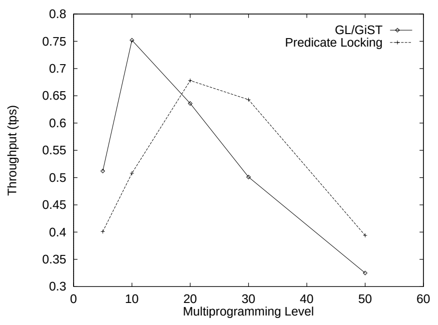


Figure 6.13: Throughput at various MPLs for 5-d data (write probability=0.1, transaction size=10, query selectivity=0.1%)

other but they overlap with the BR of the same leaf node). More the number of false conflicts, higher the loss of concurrency. Figure 6.10 shows that false conflicts do not cause a significant loss of concurrency in GL/GiST compared to PL. This is an outcome of the “fineness” of the chosen granules.

Figure 6.11 and 6.12 shows the lock overheads imposed by the GL/GiST, PL/GiST and PurePL protocols for the search and insert operations respectively. The lock overhead is measured by the average number of locks acquired or the average number of predicate checks performed, as the case may be, measured on the same scale. Although the two costs (i.e. acquiring a lock and performing a predicate check) are within the same order of magnitude (between 50-200 RISC instructions) for 2d data, the costs would differ for higher dimensional data (predicate checking becomes costlier while the cost of acquiring a lock remains the same). While the lock overhead of predicate locking increases linearly with MPL, that of GL is independent of MPL. The figures show that for both search and insert operations, GL/GiST imposes considerably lower lock overhead compared to PL protocols.

To study the performance of GL at higher dimensionalities, we also conducted experiments on 5-d data. The 5-d dataset is derived from the FOURIER dataset and is constructed by taking the first 5 fourier coefficients of each vector. We built the GiST on 480,471 points of the 5-d dataset with 8K page size(fanout 136, 5186 nodes). The buffer size was set to about 10% of the size of the GiST. Figure 6.13 shows the performance the two approaches at various MPLs for 5-d data. Like 2-d and 3-d datasets, granular locking outperforms predicate locking for 5-d data as well.

In summary, there is a tradeoff between GL and PL – while GL enjoys lower lock overhead, it has lower concurrency compared to PL. Our experiments confirm that similar to granule based protocols for 1-d datasets, the GL protocol performs significantly better than PL for multidimensional datasets as well.

6.6 Conclusions and Future Work

Numerous emerging applications (e.g., GIS, multimedia, CAD) need support of multidimensional AMs in DBMSs. The Generalized Search Tree (GiST) is an important step to meet that need. GiST, being an extensible index structure, when supported in a DBMS, will allow application developers to define their own AMs by supplying a set of extension methods. However, before GiSTs can be supported by any commercial strength DBMSs, efficient techniques to support concurrent access to data via the GiST must be developed. Concurrent access to data via a general index structure introduces two independent concurrency control problems. First, techniques must be developed to ensure the consistency of the data structure in presence of concurrent insertions, deletions and updates. Second, mechanisms to protect search regions from phantom insertions and deletions must be developed. Developing such mechanisms to guarantee transactional access to data via multidimensional data structures has been identified as one of the key challenges to transaction management in future database systems [55].

This chapter presents a dynamic granular locking approach to phantom protection in GiSTs. The chapter builds on our previous work on a dynamic granular locking strategy for R-trees [28]. Due to some fundamental differences between R-tree and GiST in the notion of a search key, the algorithms developed for R-trees do not provide a feasible solution for phantom protection in GiST. Motivated by the limitations of the previous approach in the context of GiSTs, we develop a new granular locking approach suited for concurrency control in GiSTs. The developed protocols provide a high degree of concurrency and have low lock overhead. Our experiments have shown that the granular locking technique (1) scales well under various system loads and (2) significantly outperforms predicate locking for low to medium dimensional datasets (2d, 3d and 5d). While most applications that involve dynamic datasets and require highly concurrent accesses to the data deal with low to medium dimensional spaces,[4] it is nevertheless interesting to explore approaches that provide good performance for high dimensional datasets as well. Although the granular locking proposed in this chapter provides almost as high concurrency as the predicate locking approach for low to medium dimensionalities (see Figure 6.10), the loss of concurrency increases with the increase in dimensionality. The reason is that at high dimensionalities, the data space gets increasingly sparse (a phenomenon commonly known as the "dimensionality curse" [12]), resulting in coarser leaf granules which causes more "false conflicts" and hence a higher loss in concurrency. While at low to medium dimensionalities the efficiency of granular locking far outweighs the loss of concurrency resulting in better performance compared to predicate locking, it may not be the case at high dimensionalities. This is evidenced by the fact that for 5-d data, though granular locking still outperforms predicate locking, the performance gap between them is less compared to the 2-d and 3-d datasets. A simple approach to improve the concurrency offered by granular locking is to define finer granules. The benefit of such an approach is not clear since while the finer granules will improve concurrency, it will also increase the lock overhead of each operation. A hybrid strategy between the granular and predicate locking techniques may be a more suitable solution for high dimensional datasets.

So far in this thesis, we have concentrated on multidimensional access methods as the primary weapon to deal with large volumes of highly multidimensional data. In the next chapter, we explore approximate

[4]For example, GIS and CAD systems deals with spatial data which is either 2-d or 3-d. Spatio-temporal applications (e.g., management of moving objects) deals with 3-d or 4-d data. Multimedia retrieval systems like QBIC index images using 3-d feature vectors [44].

query answering as a technique to deal with the large data volumes and stringent access time requirements in DSS/OLAP systems. We develop a wavelet-based approximate query answering tool for high dimensional DSS applications.

Chapter 7

Approximate Query Processing

In this chapter, we explore approximate query answering as a technique to deal with the large data volumes and stringent response time requirements in DSS/OLAP systems. We develop a wavelet-based approximate query answering tool for high dimensional DSS applications.

7.1 Introduction

Approximate query processing has recently emerged as a viable solution for dealing with the huge amounts of data, the high query complexities, and the increasingly stringent response-time requirements that characterize today's Decision Support Systems (DSS) applications. Typically, DSS users pose very complex queries to the underlying Database Management System (DBMS) that require complex operations over Gigabytes or Terabytes of disk-resident data and, thus, take a very long time to execute to completion and produce exact answers. Due to the *exploratory nature* of many DSS applications, there are a number of scenarios in which an exact answer may not be required, and a user may prefer a fast, approximate answer. For example, during a drill-down query sequence in ad-hoc data mining, initial queries in the sequence frequently have the sole purpose of determining the truly interesting queries and regions of the database [64]. Providing (reasonably accurate) approximate answers to these initial queries gives users the ability to focus their explorations quickly and effectively, without consuming inordinate amounts of valuable system resources. An approximate answer can also provide useful feedback on how well-posed a query is, allowing DSS users to make an informed decision on whether they would like to invest more time and resources to execute their query to completion. Moreover, approximate answers obtained from appropriate *synopses* of the data may be the only available option when the base data is remote and unavailable [6]. Finally, for DSS queries requesting a numerical answer (e.g., total revenues or annual percentage), it is often the case that the full precision of the exact answer is not needed and the first few digits of precision will suffice (e.g., the leading few digits of a total in the millions or the nearest percentile of a percentage) [1].

Prior Work. The strong incentive for approximate answers has spurred a flurry of research activity on approximate query processing techniques in recent years [1, 51, 53, 61, 64, 70, 115, 144, 145]. The majority of the proposed techniques, however, have been somewhat limited in their *query processing scope*, typically focusing on specific forms of *aggregate queries*. Besides the type of queries supported, another crucial aspect of an approximate query processing technique is the employed *data reduction mechanism*; that is, the method used to obtain synopses of the data on which the approximate query execution engine can then operate [9]. The methods explored in this context include *sampling* and, more recently, *histograms* and *wavelets*.

• *Sampling-based techniques* are based on the use of random samples as synopses for large data sets. Sample synopses can be either precomputed and incrementally maintained (e.g., [1, 51]) or they can be obtained progressively at run-time by accessing the base data using appropriate access methods (e.g., [61, 64]). Random samples of a data collection typically provide accurate estimates for aggregate quantities (e.g., `count`s or `average`s), as witnessed by the long history of successful applications of random sampling in population surveys [34, 130] and selectivity estimation [87]. An additional benefit of random samples is that they can provide probabilistic guarantees on the quality of the estimated aggregate [60]. Sampling, however, suffers from two inherent limitations that restrict its applicability as an approximate query processing tool. First, a `join` operator applied on two uniform random samples results in a *non-uniform* sample of the join result that typically contains *very few tuples*, even when the join selectivity is fairly high [1]. Thus, `join` operations typically lead to significant degradations in the quality of an approximate aggregate. ("Join synopses" [1] provide a solution, but only for *foreign-key joins that are known beforehand*; that is, they cannot support arbitrary join queries over any schema.) Second, for a *non-aggregate* query, execution over random samples of the data is guaranteed to always produce a small subset of the exact answer which is often *empty* when `join`s are involved [1, 70].

• *Histogram-based techniques* have been studied extensively in the context of query selectivity estimation [52, 68, 69, 99, 116, 117] and, more recently, as a tool for providing approximate query answers [70, 115]. The very recent work of Ioannidis and Poosala [70] is the first to address the issue of obtaining practical approximations to *non-aggregate* query answers, making two important contributions. First, it proposes a novel error metric for quantifying the quality of an approximate set-valued answer (in general, a multiset of tuples). Second, it demonstrates how standard relational operators (like `join` and `select`) can be processed directly over histogram synopses of the data. The experimental results given in [70] prove that certain classes of histograms can provide higher-quality approximate answers compared to random sampling, when considering simple queries over low-dimensional data (one or two dimensions). It is a well-known fact, however, that histogram-based approaches become problematic when dealing with the high-dimensional data sets that are typical of modern DSS applications. As the dimensionality of the data increases, both the *storage overhead* (i.e., number of buckets) and the *construction cost* of histograms that can achieve reasonable error rates increase in an explosive manner [85, 144]. The dimensionality problem is further exacerbated by `join` operations that can cause the dimensionality of intermediate query results (and the corresponding histograms) to explode.

• *Wavelet-based techniques* provide a mathematical tool for the hierarchical decomposition of functions, with a long history of successful applications in signal and image processing [74, 100, 137]. Recent studies have demonstrated the applicability of wavelets to selectivity estimation [91] and the approximation of range-sum queries over OLAP data cubes [144, 145]. The idea is to apply wavelet decomposition to the input data collection (attribute column(s) or OLAP cube) and retain the best few *wavelet coefficients* as a compact synopsis of the input data. The results of Vitter et al. [144, 145] have shown that wavelets are effective in handling aggregates over high-dimensional OLAP cubes, while avoiding the high construction costs and storage overheads of histograming techniques. Their wavelet decomposition requires only a logarithmically small number of passes over the data (regardless of the dimensionality) and their experiments prove that a few wavelet coefficients suffice to produce surprisingly accurate results for summation aggregates. Nevertheless, the focus of these earlier studies has always been on a very specific form of queries (i.e., range-sums) over a single OLAP table. Thus, the problem of whether wavelets can provide a solid foundation for general-purpose approximate query processing has hitherto been left unanswered.

Our Contributions. In this chapter, we significantly extend the scope of earlier work on approximate query answers, establishing the viability and effectiveness of wavelets as a generic approximate query processing tool for modern, high-dimensional DSS applications. More specifically, we propose a novel approach to general-purpose approximate query processing that consists of two basic steps. First, multi-dimensional Haar wavelets are used to efficiently construct compact synopses of general relational tables. Second, using novel query processing algorithms, standard SQL operators (both aggregate and non-aggregate) are evaluated *directly* over the wavelet-coefficient synopses of the data to obtain fast and accurate approximate query answers. The crucial observation here is that, as we demonstrate in this work, our approximate query execution engine can do all of its processing *entirely in the wavelet-coefficient domain*; that is, both the input(s) and the output of our query processing operators are compact collections of wavelet coefficients capturing the underlying relational data. This implies that, for any arbitrarily complex query, we can defer expanding the wavelet-coefficient synopses back into relational tuples till the very end of the query, thus allowing for extremely fast approximate query processing. [1] The contributions of our work are summarized as follows.

- **New, I/O-Efficient Wavelet Decomposition Algorithm for Relational Tables.** The methodology developed in this chapter is based on a different form of the multi-dimensional Haar transform than that employed by Vitter et al. [144, 145]. As a consequence, the decomposition algorithms proposed by Vitter and Wang [144] are not applicable. We address this problem by developing a novel, I/O-efficient algorithm for building the wavelet-coefficient synopsis of a relational table. The worst-case I/O complexity of our algorithm matches that of the best algorithms of Vitter and Wang, requiring

[1] Note that the `join` processing algorithm of Ioannidis and Poosala [70], on the other hand, requires each histogram to be partially expanded to generate the tuple-value distribution for the corresponding approximate relation. As our results demonstrate, this requirement can slow down join processing over histograms significantly, since the partially expanded histogram can give rise to large numbers of tuples, *especially* for high-dimensional data (cf. Figure 7.15).

only a logarithmically small number of passes over the data. Furthermore, there exist scenarios (e.g., when the table is stored in *chunks* [37, 129]) under which our decomposition algorithm can work in a *single pass* over the input table.

- **Novel Query Processing Algebra for Wavelet-Coefficient Data Synopses.** We propose a new algebra for approximate query processing that operates *directly over the wavelet-coefficient synopses of relations*, while guaranteeing the correct relational operator semantics. Our algebra operators include the conventional aggregate and non-aggregate SQL operators, like `select`, `project`, `join`, `count`, `sum`, and `average`. Based on the semantics of Haar wavelet coefficients, we develop novel query processing algorithms for these operators that work *entirely* in the wavelet-coefficient domain. This allows for extremely fast response times, since our approximate query execution engine can do the bulk of its processing over compact wavelet-coefficient synopses, essentially postponing the expansion into relational tuples until the end-result of the query. We also propose an efficient algorithm for this final *rendering* step, i.e., for expanding a set of multi-dimensional Haar coefficients into an approximate relation which is returned to the user as the final (approximate) answer of the query.

- **Extensive Experiments Validating our Approach.** We have conducted an extensive experimental study with synthetic as well as real-life data sets to determine the effectiveness of our wavelet-based approach compared to sampling and histograms. Our results demonstrate that (1) the quality of approximate answers obtained from our wavelet-based query processor is, in general, better than that obtained by either sampling or histograms for a wide range of `select`, `project`, `join`, and aggregate queries, (2) query execution-time speedups of more than two orders of magnitude are made possible by our approximate query processing algorithms; and (3) our wavelet decomposition algorithm is extremely fast and scales linearly with the size of the data.

Roadmap. The remainder of this chapter is organized as follows. After reviewing some necessary background material on the Haar wavelet decomposition, Section 7.2 presents our I/O-efficient wavelet decomposition algorithm for multi-attribute relational tables. In Section 7.3, we develop our query algebra and operator processing algorithms for wavelet-coefficient data synopses. Section 7.3 also proposes an efficient rendering algorithm for multi-dimensional Haar coefficients. In Section 7.4, we discuss the findings of an extensive experimental study of our wavelet-based approximate query processor using both synthetic and real-life data sets. Section 7.5 concludes the chapter.

7.2 Building Synopses of Relational Tables Using Multi-Dimensional Wavelets

7.2.1 Background: The Wavelet Decomposition

Wavelets are a useful mathematical tool to hierarchically decompose functions in a manner that is both efficient to compute and theoretically sound. Broadly speaking, the wavelet decomposition of a function consists of a coarse overall approximation together with detail coefficients that influence the function at various resolutions [137]. The wavelet decomposition has excellent energy compaction and de-correlation properties, which can be used to effectively generate compact representations that exploit the structure of data. Furthermore, wavelet transforms can generally be computed in linear time, thus allowing for very efficient algorithms.

The work in this chapter is based on the multi-dimensional *Haar wavelet* decomposition. Haar wavelets are conceptually simple, very fast to compute, and have been found to perform well in practice for a variety of applications ranging from image editing and querying [100, 137] to selectivity estimation and OLAP approximations [91, 144]. Recent work has also investigated methods for dynamically maintaining Haar-based data representations [92]. In this section, we discuss Haar wavelets in both one and multiple dimensions.

One-Dimensional Haar Wavelets. Suppose we are given a one-dimensional data vector A containing the following four values $A = [2, 2, 5, 7]$. The Haar wavelet transform of A is computed as follows. We first perform pairwise averaging of the values to get the following "lower-resolution" representation of the data vector: $[2, 6]$. In other words, the average of the first two values (that is, 2 and 2) is 2 and that of the next two values (that is, 5 and 7) is 6. Obviously, some information has been lost in this averaging process. To be able to restore the original four values of the data array, we need to store some *detail coefficients,* that capture the missing information. In Haar wavelets, these detail coefficients are simply the differences of the second element of the pairs being averaged from the average value.

In our example, for the first pair of averaged values, the detail coefficient is 0 since 2-2 =0, while for the second we need to store -1 since $6 - 7 = -1$. Note that it is possible to reconstruct the four values $[2, 2, 5, 7]$ of the original data array from the lower-resolution array containing the two averages $[2, 6]$ and the two detail coefficients $[0, -1]$. Recursively applying the above pairwise averaging and differencing process on the lower-resolution array containing the averages, we get the following full decomposition.

Resolution	Averages	Detail Coefficients
2	[2, 2, 5, 7]	–
1	[2, 6]	[0, -1]
0	[4]	[-2]

We define the *wavelet transform* (also known as the *wavelet decomposition*) of A to be the single coefficient representing the overall average of the data values (i.e. $[4]$) followed by the detail coefficients in the order of increasing resolution (i.e. $[-2]$ at the lowest resolution and $[0, -1]$ at the next higher resolution as there are only two resolutions of detail coefficients of A). The one-dimensional Haar wavelet transform of A is therefore given by $W_A = [4, -2, 0, -1]$. Each entry in W_A is called a *wavelet coefficient*. The main advantage of using W_A instead of the original data vector A is that for vectors containing similar values in neighboring positions (i.e. having locality), most of the detail coefficients would have very small values. Eliminating such small coefficients from the wavelet transform (i.e., treating them as zeros) introduces only small errors when reconstructing the original data, giving an effective form of lossy data compression.

Note that, intuitively, wavelet coefficients carry different weights with respect to their importance in rebuilding the original data values. For example, the overall average is obviously more important than any detail coefficient since it affects the reconstruction of all entries in the data array. In order to equalize the importance of all wavelet coefficients while determining which coefficients to retain and which to eliminate (i.e. during thresholding), we need to *normalize* the final entries of W_A appropriately. This is achieved by dividing each wavelet coefficient by $\sqrt{2^l}$, where l denotes the *level of resolution* at which the coefficient appears (with $l = 0$ corresponding to the "coarsest" resolution level). Thus, the normalized wavelet transform for our example data array becomes $W_A = [4, -2, 0, -1/\sqrt{2}]$. Note that the unnormalized transform is used for the reconstruction of the original vector; the normalized version is used only for thresholding (cf. Section 7.2.2).

Multi-Dimensional Haar Wavelets. There are two common methods to compute the Haar wavelet transform of a *multi-dimensional* array. Each of these transformations is a generalization of the one-dimensional decomposition process described above. To simplify the exposition to the basic ideas of multi-dimensional wavelets, we assume that the input array is of equal size along all dimensions.

The first method is known as *standard decomposition*. In this method, we first fix an ordering of the dimensions of the input array A (say, $1, 2, \ldots, d$) and then proceed to apply the complete one-dimensional wavelet transform for each one-dimensional "row" of array cells along dimension k, for all $k = 1, \ldots, d$. The standard Haar decomposition forms the basis of the recent results of Vitter et al. on OLAP data cube approximations [144, 145].

The work presented in this chapter is based on the second method of extending Haar wavelets to multiple dimensions, namely the *nonstandard decomposition*. Instead of performing one-dimensional wavelet transform on *all* one-dimensional rows along dimension 1 followed by transform on *all* rows along dimension 2 and so on as in standard decomposition, the nonstandard Haar decomposition alternates between the the one-dimensional rows along different dimensions i.e. at each step, it performs a one-dimensional wavelet transform on one row along dimension 1 followed by transform on one row along dimension 2 and so on till dimension k. The steps are repeated till each row along each dimension has been transformed. Note that the transform of A is performed "in place": i.e. the results of a transform is used as the input data for subsequent transforms. The above process is then repeated recursively on the quadrant containing averages across all dimensions. One way of to conceptualize (and implement [100]) the above process is sliding a

$2 \times 2 \times \cdots \times 2$ d-dimensional hyper-box across the data array, performing pairwise averaging and differencing of the cells in A falling inside the hyper-box, distributing the results to the appropriate locations of the wavelet transform array W_A (with the averages for each box going to the “lower-left” quadrant of W_A) and, finally, recursing the computation on the lower-left quadrant of W_A. This procedure is demonstrated pictorially for a 2-dimensional data array A in Figure 7.1(a). A is $2^m \times 2^m$ in size. The figure shows the pairwise averaging and differencing step for one positioning of the 2×2 box with its “root”(i.e., lower-left corner) located at the coordinates $[2i_1, 2i_2]$ of A followed by the distribution of the results in the wavelet transform array. The above step is repeated for every possible combination of i_j’s, $i_j \in \{0, \ldots, 2^{m-1} - 1\}$. A detailed description of the nonstandard Haar decomposition can be found in any standard reference on the subject (e.g., [74, 137]).

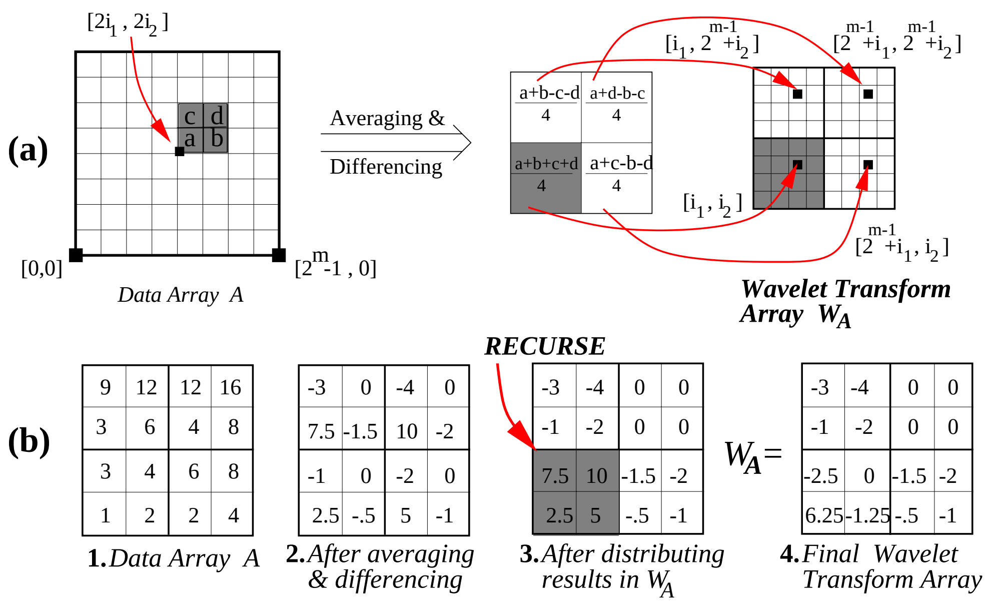


Figure 7.1: Non-standard decomposition in two dimensions. (a) Computing pairwise averages and differences and distributing them in the wavelet transform array. (b) Example decomposition of a 4×4 array.

Example 7.2.1: Consider the 4×4 array A shown in Figure 7.1(b.1). In the first level of recursion, the 2×2 sliding hyper-box is placed at the 4 possible “root” positions on A, namely $[0, 0]$, $[0, 2]$, $[2, 0]$ and $[2, 2]$, and pairwise averaging and differencing is performed on each of them individually. The result is shown in Figure 7.1(b.2). For example, the pairwise averaging and differencing on the hyper-box with root position $[2, 0]$ (containing values $A[2, 0] = 2$, $A[3, 0] = 4$, $A[2, 1] = 6$, and $A[3, 1] = 8$) produces the average coefficient $\frac{A[2,0]+A[3,0]+A[2,1]+A[3,1]}{4} = 5$ and detail coefficients $\frac{A[2,0]+A[2,1]-A[3,0]-A[3,1]}{4} = -1$, $\frac{A[2,0]+A[3,0]-A[2,1]-A[3,1]}{4} = -2$ and $\frac{A[2,0]+A[3,1]-A[3,0]-A[2,1]}{4} = 0$ (shown in the same positions ($A[2, 0]$, $A[3, 0]$, $A[2, 1]$ and $A[3, 1]$). Figure 7.1(b.3) shows the array after the results are distributed in the right positions in W_A. For the hyper-box with root position $[2, 0]$ (i.e. $i_1 = 1$, $i_2 = 0$ and $m = 2$ according to the notation in Figure 7.1(a)), the results $5, -1, -2$ and 0 are placed at positions $[i_1, i_2] = [1, 0]$, $[2^{m-1} + i_1, i_2] = [3, 0]$, $[i_1, 2^{m-1} + i_2] = [1, 2]$ and $[2^{m-1} + i_1, 2^{m-1} + i_2] = [3, 2]$ respectively. The process is then recursed on the lower-left quadrant of W_A (which contains the average values $2.5, 7.5, 5$ and 10 of the 4 boxes), resulting in the average coefficient 6.25 and detail coefficients -1.25, -2.5 and 0. That ends the recursion, producing the final wavelet transform array W_A shown in Figure 7.1(b.4). ▮

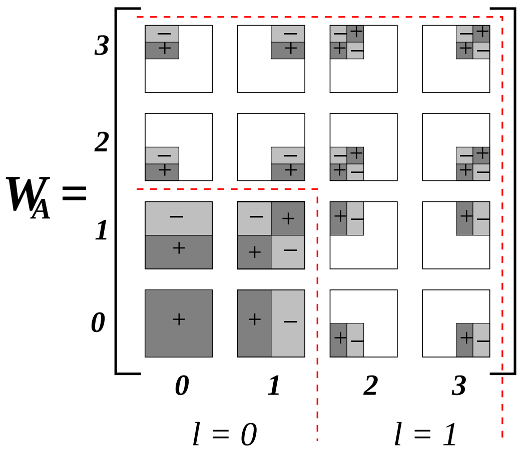
As noted in the wavelet literature, both methods for extending one-dimensional Haar wavelets to higher dimensionalities have been used in a wide variety of application domains and, to the best of our knowledge, neither has been shown to be uniformly superior. Our choice of the nonstandard method was mostly motivated by our earlier experience with nonstandard two-dimensional Haar wavelets in the context of effective image retrieval [100]. An advantage of using the nonstandard transform is that, as we explain later in the chapter, it allows for an efficient representation of the sign information for wavelet coefficients. This efficient representation stems directly from the construction process for a nonstandard Haar basis [137]. Using nonstandard Haar wavelets, however, also implies that the standard decomposition algorithms of Vitter and Wang [144] are no longer applicable. We address this problem by proposing a novel I/O-efficient algorithm for constructing the nonstandard wavelet decomposition of a relational table (Section 7.2.2). (We often omit the "nonstandard" qualification in the rest of the chapter.)

Multi-Dimensional Haar Coefficients: Semantics and Representation. Consider a wavelet coefficient W generated during the multi-dimensional Haar decomposition of a d-dimensional data array A. Mathematically, the coefficient is a multiplicative factor for an appropriate *Haar basis function* when the data in A is expressed using the d-dimensional Haar basis [137]. The d-dimensional Haar basis function corresponding to W is defined by (1) a *d-dimensional rectangular support region* in A that captures the region of A's cells that W contributes to during reconstruction; and (2) the *quadrant sign information* that defines the sign ($+$ or $-$) of W's contribution (i.e., $+W$ or $-W$) to any cell contained in a given quadrant of its support rectangle. The wavelet decomposition process guarantees that (1) W can contribute only to a rectangular regions of A's cells i.e. the support region is always a d-dimensional rectangle and (2) the signs of W's contribution to those cells can change only across quadrants of the support region i.e. we need to store at most one sign per quadrant. For example, the overall average coefficient $W_A[0, 0] = 6.25$ in Figure 7.1(b) contributes positively (i.e.,"$+6.25$") to the reconstruction of all the cells in A, so its support region in the whole array A and its sign is $+$ for all quadrants of the support region. On the other hand, the detail coefficient $W_A[1, 2] = -2$ contributes only the cells in the lower-right quadrant of A (i.e. $A[2, 0], A[3, 0], A[2, 1]$ and $A[3, 1]$) and the signs are $+$ for the lower left and lower right quadrants of the support region and $-$ for the other two quadrants (i.e. contributes $+(-2) = -2$ to A[2,0] and A[3,0] and $-(-2) = +2$ to A[2,1] and A[3,1]). The support regions and signs of all the sixteen coefficients in Figure 7.1(b.4) are shown in Figure 7.2(a). The support regions are superimposed on the entire array A: the white areas for each coefficient correspond to regions of A which it does not contribute to i.e. whose reconstruction is independent of the coefficient (e.g., $W_A[1,2]$ is white for all cells except $A[2,0], A[3,0], A[2,1]$ and $A[3,1]$). Figure 7.2(a) also depicts the two *levels of resolution* ($l = 0, 1$) for our example two-dimensional Haar coefficients; as in the one-dimensional case, these levels define the appropriate constants for normalizing coefficient values (see, e.g., [137]).

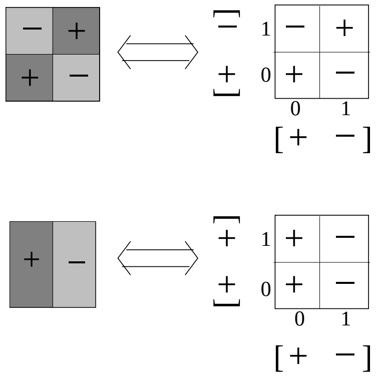
Example 7.2.2: Since the support region represents the cells in A which a wavelet coefficient contributes to, the value of a cell in A can be reconstructed by adding up the contributions (with the appropriate signs) of those coefficients who support regions include the cell. For example, the coefficients whose support regions include A[0,1] are $W_A[0,0](+)$, $W_A[0,1](+)$, $W_A[1,0](+)$, $W_A[1,1](+)$, $W_A[0,2](-)$, $W_A[2,0](+)$ and $W_A[2,2](-)$, so $A[0,1]$ can be reconstructed using the following formula:

$$A[0,1] = +W_A[0,0]+W_A[0,1]+W_A[1,0]+W_A[1,1]-W_A[0,2]+W_A[2,0]-W_A[2,2] = 2.5-(-1)+(-.5) = 3.$$

∎



(a)



(b)

Figure 7.2: (a) Support regions and signs for the sixteen nonstandard two-dimensional Haar basis functions. The coefficient magnitudes are multiplied by $+1$ (-1) where a sign of $+$ (resp., $-$) appears, and 0 in blank areas. (b) Representing quadrant sign information for coefficients using "per-dimension" sign vectors.

To simplify the discussion in this chapter, we abstract away the distinction between a coefficient and its corresponding basis function by representing a Haar wavelet coefficient with the triple $W = \langle R, S, v \rangle$, where:

1. $W.R$ is the *d-dimensional support hyper-rectangle of W* enclosing all the cells in the data array A which W contributes to (i.e., the support of the corresponding basis function). We represent this hyper-rectangle by its low and high boundary values (i.e., starting and ending array cells) along each dimension j, $1 \leq j \leq d$; these are denoted by $W.R.boundary[j].lo$ and $W.R.boundary[j].hi$, respectively. Thus, the coefficient W contributes to each data cell $A[i_1, \ldots, i_d]$ satisfying the condition

$W.R.boundary[j].lo \leq i_j \leq W.R.boundary[j].hi$ for all dimensions j, $1 \leq j \leq d$. For example, for the detail coefficient $W_A[1, 2]$ in Figure 7.1(b), $W.R.boundary[0].lo = 2$, $W.R.boundary[0].hi = 3$, $W.R.boundary[1].lo = 0$ and $W.R.boundary[1].hi = 1$. The space required to store the support hyper-rectangle of a coefficient is $2 \log N$ bits, where N denotes the total number of cells of A.

2. $W.S$ stores the *sign information for all d-dimensional quadrants of W.R*. Storing the quadrant sign information directly (i.e. a sign per quadrant) would mean a space requirement of $O(2^d)$ as there are 2^d quadrants in d-dimensional hyper-rectangle. Instead, we use a more space-efficient representation of the quadrant sign information (using only $2d$ bits) that exploits the regularity of the nonstandard Haar transform. The basic observation here is that a nonstandard d-dimensional Haar basis is formed by scaled and translated products of d one-dimensional Haar basis functions [137]. Thus, our idea is to store a 2-bit *sign vector* for each dimension j that captures the sign variation of the corresponding one-dimensional basis function. The two elements of the sign vector of coefficient W along dimension j are denoted by $W.S.sign[j].lo$ and $W.S.sign[j].hi$, and contain the signs that correspond to the lower and upper half of $W.R$'s extent along dimension j, respectively. Given the sign vectors along each dimension and treating a sign of $+$ ($-$) as being equivalent to $+1$ (resp., -1), the sign of any d-dimensional quadrant can be computed by taking the product of the d sign-vector entries that map to that quadrant; that is, following exactly the basis construction process. (Note that we will continue to make use of this "+1/-1" interpretation of signs throughout the chapter.) Figure 7.2(b) shows the sign-computation methodology for two example coefficient hyper-rectangles from Figure 7.2(a). For example, the upper example in Figure 7.2(b) shows a coefficient with sign vectors $W.S.sign[0].lo = +1$ and $W.S.sign[0].hi = -1$ along dimension 0 (x-axis) and $W.S.sign[1].lo = +1$ and $W.S.sign[1].hi = -1$ along dimension 1 (y-axis); the signs of the lower left, lower right, upper left and upper right quadrants of its support region are therefore $W.S.sign[0].lo * W.S.sign[1].lo = +1$, $W.S.sign[0].hi * W.S.sign[1].lo = -1$, $W.S.sign[0].lo * W.S.sign[1].hi = -1$, and $W.S.sign[0].hi * W.S.sign[1].hi = +1$ respectively.

3. $W.v$ is the *(scalar) magnitude of coefficient W*. This is exactly the quantity that W contributes (either positively or negatively, depending on $W.S$) to all data array cells enclosed in $W.R$. For example, the magnitude of $W_A[0, 0]$ in Figure 7.1(b) is 6.25 and that of $W_A[1, 2]$ is -2.

Thus, our view of a d-dimensional Haar wavelet coefficient is that of a d-dimensional hyper-rectangle with a magnitude and a sign that may change across quadrants. Note that, by the properties of the nonstandard Haar decomposition, given *any pair* of coefficients, their hyper-rectangles are either *completely disjoint* or one is *completely contained* in the other; that is, coefficient hyper-rectangles cannot *partially overlap*. As will be seen later, it is precisely these containment properties coupled with our sign-vector representation of quadrant signs that enable us to efficiently perform `join` operations directly over wavelet-coefficient synopses.

7.2.2 Building and Rendering Wavelet-Coefficient Synopses

Consider a relational table R with d attributes $X_1, X_2, \ldots X_d$. A straightforward way of obtaining a wavelet-based synopsis of R would be to take the traditional two-dimensional array view of a relational table (with attributes on the x-axis and tuples on the y-axis), apply a two-dimensional wavelet decomposition on R, and retain a few large coefficients. It is highly unlikely, however, that this solution will produce a high-quality compression of the underlying data. The reason is that wavelets (like most compression mechanisms) work by exploiting locality (i.e., clusters of constant or similar values), which is almost impossible when grouping together attributes that can have vastly different domains (e.g., consider an `age` attribute adjacent to a `salary` attribute). Similar problems occur in the vertical grouping as well, since even sorting by some attribute(s) cannot eliminate large "spikes" for others. We address these problems by taking a slightly different view of the d-attribute relational table R. We can represent the information in R as a d-dimensional array A_R, whose j^{th} dimension is indexed by the values of attribute X_j and whose cells contain the count of tuples in R having the corresponding combination of attribute values. A_R is essentially the *joint frequency distribution* (JFD) of all the attributes of R. Figure 7.3 shows an example relation with 2 attributes (Figure 7.3(a)) and the corresponding JFD array (Figure 7.3(b)). We obtain the wavelet synopsis of R by performing non-standard multi-dimensional wavelet decomposition (denoted by W_R) of A_R and then retaining only some of the coefficients (based on the desired size of the synopsis) using a thresholding scheme. In this section, we propose a novel, I/O-efficient algorithm for constructing W_R. Note that, even though our algorithm computes the decomposition of A_R, it in fact works off the "set-of-tuples" (ROLAP) representation of R. (As noted by Vitter and Wang [144], this is a requirement for computational efficiency since the JFD array A_R is typically very sparse, especially for the high-dimensional data sets that are typical of DSS applications.) We also briefly describe our thresholding scheme for controlling the size of a wavelet-coefficient synopsis. We have also developed a time- and space-efficient algorithm (termed `render`) for *rendering* (i.e., expanding) a synopsis into an approximate "set-of tuples" relation (which is used during query processing as the final step). We begin by summarizing the notational conventions used throughout the chapter.

Notation. Let $\mathcal{D} = \{D_1, D_2, \ldots, D_d\}$ denote the set of dimensions of A_R, where dimension D_j corresponds to the *value domain* of attribute X_j. Without loss of generality, we assume that each dimension D_j is indexed by the set of integers $\{0, 1, \cdots, |D_j| - 1\}$, where $|D_j|$ denotes the size of dimension D_j. [2] The d-dimensional JFD array A_R comprises $N = \prod_{i=1}^{d} |D_i|$ cells with cell $A_R[i_1, i_2, \ldots, i_d]$ containing the count of tuples in R having $X_j = i_j$ for each attribute $1 \leq j \leq d$. We define N_z to be the number of populated (i.e., non-zero) cells of A_R (typically, $N_z << N$). Table 7.1 outlines the notation used in this

[2] We assume that the attributes $\{X_1, \ldots, X_d\}$ are ordinal in nature i.e. their domain are ordered. This includes all numeric attributes (e.g., age, income) and some categorical attributes (e.g., education). Such domains can always be mapped to the set of integers mentioned above while preserving the order and hence the locality of the distribution. It is also possible to map unordered to domains to the set of integers; however, such mappings do not always preserve the locality of the data. For example, mapping countries to integers using alphabetic ordering does not preserve locality. There may be alternate mappings that are more locality preserving, e.g., assigning neighboring integers to neighboring countries. Such mapping techniques based on concept hierarchies are discussed in [40].

chapter with a brief description of its semantics. We provide detailed definitions of some of these parameters in the text. Additional notation will be introduced when necessary.

Symbol	**Semantics**
d	Number of attributes (i.e., dimensionality) of the input relational table
R, A_R	Relational table and corresponding d-dimensional joint frequency array
X_j , D_j	j^{th} attribute of relation R and corresponding domain of values ($1 \leq j \leq d$)
$\mathcal{D} = \{D_1, \ldots, D_d\}$	Set of all data dimensions of the array A_R
$A_R[i_1, i_2, \cdots, i_d]$	Count of tuples in R with $X_j = i_j$ ($i_j \in \{0, \ldots, \lvert D_j\rvert - 1\}$) , $\forall 1 \leq j \leq d$
$N = \prod_j \lvert D_j\rvert$	Size (i.e., number of cells) of A_R
N_z	Number of *non-zero* cells of A_R ($N_z << N$)
$W_R[i_1, i_2, \cdots, i_d]$	Coefficient located at coordinates $[i_1, i_2, \cdots, i_d]$ of the wavelet transform array W_R
$W.R.boundary[j].\{lo, hi\}$	Support hyper-rectangle boundaries along dimension D_j for coefficient W ($1 \leq j \leq d$)
$W.S.sign[j].\{lo, hi\}$	Sign vector information along dimension D_j for the wavelet coefficient W ($1 \leq j \leq d$)
$W.S.signchange[j]$	Sign-change value along dimension D_j for the wavelet coefficient W ($1 \leq j \leq d$)
$W.v$	Scalar magnitude of the wavelet coefficient W
l	Current level of resolution of the wavelet decomposition

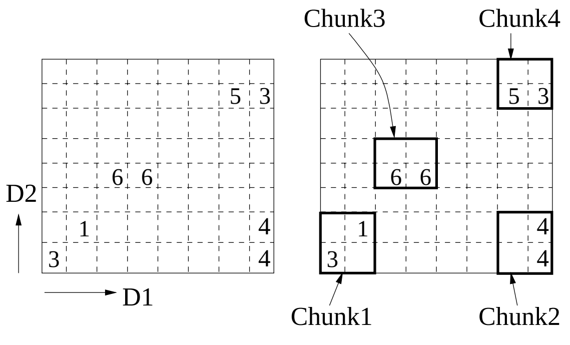
Table 7.1: Notation

Most of the notation pertaining to wavelet coefficients W has already been described in Section 7.2.1. The only exception is the *sign-change value vector* $W.S.signchange[j]$ that captures the value along dimension j (between $W.R.boundary[j].lo$ and $W.R.boundary[j].hi$) at which a transition in the value of the sign vector $W.S.sign[j]$ occurs, for each $1 \leq j \leq d$. That is, the sign $W.S.sign[j].lo$ ($W.S.sign[j].hi$) applies to the range $[W.R.boundary[j].lo, \ldots, W.S.signchange[j] - 1]$ (resp., $[W.S.signchange[j], \ldots, W.R.boundary[j].hi]$). As a convention, we set $W.S.signchange[j]$ equal to $W.R.boundary[j].lo$ when there is no "true" sign change along dimension j, i.e., $W.S.sign[j]$ contains $[+, +]$ or $[-, -]$. Note that, for base Haar coefficients with a true sign change along dimension j, $W.S.signchange[j]$ is simply the midpoint between $W.R.boundary[j].lo$ and $W.R.boundary[j].hi$ (Figure 7.2). This property, however, no longer holds when arbitrary selections and joins are executed over the wavelet coefficients. As a consequence, we need to store sign-change values explicitly in order to support general query processing operations in an efficient manner.

The COMPUTEWAVELET **Decomposition Algorithm.** We now present our I/O-efficient algorithm (called COMPUTEWAVELET) for constructing the wavelet decomposition of R. Our algorithm exploits the interaction of nonstandard wavelet decomposition and "chunk-based" organizations of relational tables [129, 37]. In chunk-based organizations, the JFD array A_R is split into d-dimensional *chunks* and tuples of R belonging to the same chunk are stored contiguously on disk. Figure 7.3 shows an example chunking of A_R (Figure 7.3(c)) and the corresponding organization of R (Figure 7.3(d)). If R is organized in chunks, COMPUTEWAVELET can perform the decomposition in a *single pass* over the tuples of R. Note that such data organizations have already been proposed in earlier work (e.g., the *chunked-file organization* of Deshpande et al. [37] and Orenstein's *z-order* linearization [73, 109]), where they have been shown to have significant

Dim D1 (Attr X1)	Dim D2 (Attr X2)	Count
0	0	3
7	0	4
1	1	1
7	1	4
2	2	6
3	2	6
6	6	5
7	6	3

(a)



Dim D1 (Attr X1)	Dim D2 (Attr X2)	Count
0	0	3
1	1	1
7	0	4
7	1	4
2	2	6
3	2	6
6	6	5
7	6	3

(d)

Figure 7.3: (a) An example relation R with 2 attributes (b) The corresponding JFD array A_R (c) One possible chunking of A_R: all cells inside a chunk are stored contiguously on disk. The chunk size is assumed to be 2 i.e. 2 cells (or tuples) fit in one chunk. (d) The corresponding chunked organization of R: all tuples belonging to the same chunk are stored contiguously.

performance benefits for DSS applications due to their excellent multi-dimensional clustering properties.

We present our I/O-efficient COMPUTEWAVELET algorithm below assuming that R's tuples are organized in d-dimensional chunks. If R is not chunked, then an extra preprocessing step is required to reorganize R on disk (e.g., to reorganize the relation shown in Figure 7.3(a) as that in Figure 7.3(d)). This preprocessing is no more expensive than a sorting step (e.g., in *z-order*) which requires a logarithmic number of passes over R. Thus, while the wavelet decomposition requires just a single pass when R is chunked, in the worst-case (i.e., when R is not "chunked"), the I/O complexity of COMPUTEWAVELET matches that of Vitter and Wang's I/O-efficient algorithm for standard Haar wavelet decomposition [144]. We also assume that each chunk can individually fit in memory. We show that the extra memory required by our wavelet decomposition algorithm (in addition to the memory needed to store the chunk itself) is at most $O(2^d \cdot \log(\max_j\{|D_j|\}))$. Finally, our implementation of COMPUTEWAVELET also employs several of the improvements suggested by Vitter and Wang [144], including a dynamic coefficient thresholding scheme to ensure that the density of the data remains approximately constant across successive averaging and differencing steps. We do not discuss the dynamic thresholding step below to keep the presentation of COMPUTEWAVELET simple.

Our I/O-efficient decomposition algorithm is based on the following observation:

The decomposition of a d-dimensional array A_R can be computed by independently computing the decomposition for each of the 2^d d-dimensional subarrays corresponding to A_R's quadrants and then performing pairwise averaging and differencing on the computed 2^d averages of A_R's quadrants.

Due to the above property, when a chunk is loaded from the disk for the first time, COMPUTEWAVELET can perform the entire computation required for decomposing the chunk right away (hence no chunk is read twice). Lower resolution coefficients are computed by first accumulating, in main memory, averages from the 2^d quadrants (generated from the previous level of resolution) followed by pairwise averaging and differencing, thus requiring no extra I/O. Due to the depth first nature of the algorithm, the pairwise averaging and differencing is performed *as soon as* all the 2^d averages are accumulated, making the algorithm memory efficient (as, at no point of computation, there can be more than one "active" subarray (whose averages are still being accumulated) for each level of resolution).

The outline of our I/O-efficient wavelet decomposition algorithm COMPUTEWAVELET is depicted in Figure 7.4. To simplify the presentation, the COMPUTEWAVELET pseudo-code assumes that all dimensions of the data array A_R are of equal size, i.e., $|D_1| = |D_2| = \ldots = |D_d| = 2^m$. We discuss handling of unequal sizes later in this section. Besides the input JFD array (A_R) and the logarithm of the dimension size (m), COMPUTEWAVELET takes two additional arguments: (a) the *root (i.e., "lower-left" endpoint) coordinates* of the d-dimensional subarray for which the wavelet transform is to be computed $(i_1, i_2, \ldots, i_d)$, and (b) the current *level of resolution* for the wavelet coefficients (l). Note that, for a given level of resolution l, the extent (along each dimension) of the d-dimensional array rooted at $(i_1, i_2, \ldots, i_d)$ being processed is exactly 2^{m-l}. The procedure computes the wavelet coefficients of the input subarray and returns the overall average to the caller (Step 14). It does so by: (1) performing wavelet decomposition recursively on each of the 2^d quadrants of the input array and collecting the averages returned in a $2 \times \cdots \times 2 = 2^d$ temporary hyper-box T (Steps 2–4), (2) performing pairwise averaging and differencing on T to produce the average and detail coefficients for the level-l decomposition of the input subarray (Step 5), and finally, (3) distributing these level-l wavelet coefficients to the appropriate locations of the wavelet transform array W_R (computing their support hyper-rectangles and dimension sign vectors at the same time) (Steps 6–12). The initial invocation of COMPUTEWAVELET is done with parameters $(A, 3, (0, 0), 0)$.

Example 7.2.3: Figures 7.5 illustrates the working on the COMPUTEWAVELET algorithm on the 8×8 data array A_R (corresponding to the relation shown in Figure 7.3). The recursive invocations of COMPUTEWAVELET form a depth-first invocation tree: the root corresponds to the initial invocation COMPUTEWAVELET (A, 3, (0,0), 0) with the entire A_R as the input subarray. The root then invokes COMPUTEWAVELET on its four quadrants: COMPUTEWAVELET (A, 3, (0,0), 1), COMPUTEWAVELET (A, 3, (0,4), 1), COMPUTEWAVELET (A, 3, (4,0), 1) and COMPUTEWAVELET (A, 3, (4,4), 1) with the lower left, upper left, lower right and upper right quadrants as the input subarrays respectively. COMPUTEWAVELET (A, 3, (0,0), 1) in turn invokes COMPUTEWAVELET on its four quadrants: COMPUTEWAVELET (A, 3, (0,0), 2), COMPUTEWAVELET (A, 3, (0,2), 2), COMPUTEWAVELET (A, 3, (2,0), 2) and COMPUTEWAVELET (A, 3, (2,2), 2). COMPUTEWAVELET (A, 3, (0,0), 2) then invokes COMPUTEWAVELET on its four quadrants: COMPUTEWAVELET (A, 3, (0,0), 3), COMPUTEWAVELET (A, 3, (0,1), 3), COMPUTEWAVELET (A, 3, (1,0), 3) and COMPUTEWAVELET (A, 3, (1,1), 3). Each of these 4 invocation satisfy the terminating condition in Line 1 of Figure 7.4: so they simply return the respective input 1-cell subarrays (3, 0, 0 and 1 respectively). The caller i.e. COMPUTEWAVELET (A, 3, (0,0), 2) collects those returned values (i.e. 3, 0, 0 and 1) in the "quadrant averages array" T, performs pairwise averaging and differencing, distributes the results in W_R and returns the average (i.e. 1) to its caller (i.e. the COMPUTEWAVELET (A, 3, (0,0), 1) invocation). The other three invocations by COMPUTEWAVELET (A, 3, (0,0), 1), namely, COMPUTEWAVELET (A, 3, (0,2), 2), COMPUTEWAVELET (A, 3, (2,0), 2) and COMPUTEWAVELET (A, 3, (2,2), 2) are processed in the same way. The caller i.e. COMPUTEWAVELET (A, 3, (0,0), 1) then collects those returned values (i.e. 1, 0, 0 and 3) in the "quadrant averages array" T, performs pairwise averaging and differencing, distributes the results in W_R and returns the average (i.e. 1) to its caller (i.e. the COMPUTEWAVELET (A, 3, (0,0), 0) invocation). The other three invocations by COMPUTEWAVELET (A, 3, (0,0), 0), namely, COMPUTEWAVELET (A, 3, (0,4), 1), COMPUTEWAVELET (A, 3, (4,0), 1) and COMPUTEWAVELET (A, 3, (4,4), 1) are processed in the same way. The caller i.e. COMPUTEWAVELET (A, 3, (0,0), 0) then collects those returned values (i.e. 1, 0, 0.5 and 0.5) in the "quadrant averages array" T, performs pairwise averaging and differencing, distributes the results in W_R and returns the average (i.e. 0.5). ∎

```
procedure COMPUTEWAVELET($A_R, m, (i_1, i_2, \ldots, i_d), l$)
begin
1.  if $l \geq m$ return $A_R[i_1, \ldots, i_d]$
2.  for $t_1$ := 0,1   ···   for $t_d$ := 0,1
3.      $T[t_1, \ldots, t_d] :=$ COMPUTEWAVELET($A_R, m, (i_1 + t_1 \cdot 2^{m-l-1}, i_2 + t_2 \cdot 2^{m-l-1}, \ldots, i_d + t_d \cdot 2^{m-l-1}), l+1$)
4.  end   ···   end
5.  perform pairwise averaging and differencing on the $2 \times \ldots \times 2 = 2^d$ hyper-box $T$
6.  for $t_1$ := 0,1   ···   for $t_d$ := 0,1
7.      $W_R[t_1 \cdot 2^l + \frac{i_1}{2^{m-l}}, \ldots, t_d \cdot 2^l + \frac{i_d}{2^{m-l}}].v := T[t_1, \ldots, t_d]$
8.      for $j := 1, \ldots, d$
9.          $W_R[t_1 \cdot 2^l + \frac{i_1}{2^{m-l}}, \ldots, t_d \cdot 2^l + \frac{i_d}{2^{m-l}}].R.boundary[j] := [i_j, i_j + 2^{m-l} - 1]$
10.         $W_R[t_1 \cdot 2^l + \frac{i_1}{2^{m-l}}, \ldots, t_d \cdot 2^l + \frac{i_d}{2^{m-l}}].S.sign[j] := (t_j == 0)\ ?\ [+,+]\ :\ [+,-]$
11.         $W_R[t_1 \cdot 2^l + \frac{i_1}{2^{m-l}}, \ldots, t_d \cdot 2^l + \frac{i_d}{2^{m-l}}].S.signchange[j] := (t_j == 0)\ ?\ i_j\ :\ i_j + 2^{m-l}$
12.     end
13. end   ···   end
14. return $T[0, \ldots, 0]$
end
```

Figure 7.4: COMPUTEWAVELET: An I/O-efficient wavelet decomposition algorithm.

Assuming we can to store the temporary quadrant averages arrays T in memory, COMPUTEWAVELET can load the d-dimensional chunks of A_R into memory one at a time and compute the wavelet coefficients *at all levels* for each chunk with no additional I/O's. This property guarantees that all computation is completed in a single pass over the chunks of A_R i.e. the time complexity of COMPUTEWAVELET is $O(N_z)$. [3] If A_R is not chunked, the complexity of $O(N_z.log(N_z))$ due to the preprocessing step as discussed before. The memory requirement of the algorithm is that of storing those temporary hyper-boxes (in addition to that needed to store the data chunk itself). Each such hyper-box consists of exactly 2^d entries and the

[3]For simplicity, the COMPUTEWAVELET algorithm shown in Figure 7.4 works on A_R and hence has a complexity of $O(N)$. Our implementation, as mentioned before, works on R itself and hence has a time complexity of $O(N_z)$.

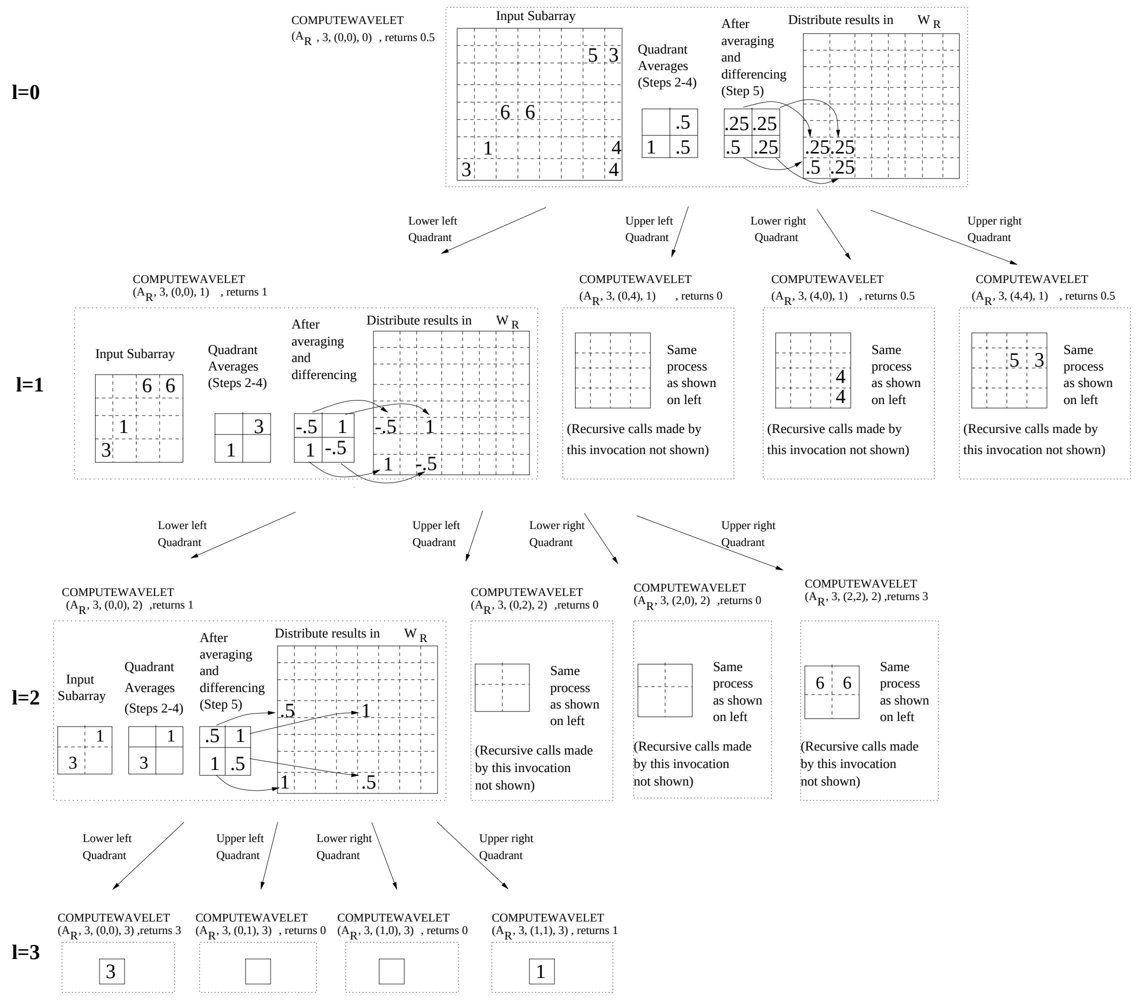


Figure 7.5: Execution of the ComputeWavelet algorithm of a 8×8 data array. Each invocation of the ComputeWavelet procedure is shown in a dotted box labeled with the procedure call with the right parameters.

number of distinct hyper-boxes that can be "active" at any given point in time during the operation of ComputeWavelet is bounded by the depth of the recursion, or equivalently, the number of distinct levels of coefficient resolution. Thus, the extra memory required by ComputeWavelet is at most $O(2^d \cdot m)$ (when $|D_1| = \ldots = |D_d| = 2^m$) or $O(2^d \cdot \log(\max_j\{|D_j|\}))$ (for the general case of unequal dimension extents).

We should note here that both the hyper-rectangle and the sign information for any coefficient generated during the execution of ComputeWavelet over a base relation R can easily be derived from the location of the coefficient in the wavelet transform array W_R, based on the regular recursive structure of the decomposition process. Thus, in order to conserve space, hyper-rectangle boundaries and sign vectors are not explicitly stored in the wavelet-coefficient synopses of base relations. (All that we need are the coefficients' coordinates in W_R.) As we will see later, however, this information does need to be stored explicitly for intermediate collections of wavelet coefficients generated during query processing,

Handling Unequal Dimension Extents If the sizes of the dimensions of A_R are not equal, then the recursive invocation of COMPUTEWAVELET for quadrant $[t_1, \ldots, t_d]$ (Step 3) takes place only if the inequality $i_j + t_j \cdot 2^{m-l-1} < |D_j|$ is satisfied, for each $j = 1, \ldots, d$. This means that, initially, quadrants along certain "smaller" dimensions are not considered by COMPUTEWAVELET; however, once quadrant sizes become smaller than the dimension size, computation of coefficients in quadrants for such smaller dimensions is initiated. Consequently, the pairwise averaging and differencing computation (Step 5) is performed only along those dimensions that are *"active"* in the current level of the wavelet decomposition. The support hyper-rectangles and dimension sign vectors for such active dimensions are computed as described in Steps 8–10, whereas for an "inactive" dimension j the hyper-rectangle boundaries are set at $boundary[j] := (0, |D_j|-1)$ (the entire dimension extent) and the sign vector is set at $sign[j] = [+, +]$.

As mentioned in Section 7.2.1, the coefficient values computed by COMPUTEWAVELET need to be properly *normalized* in order to ensure that the Haar basis functions are orthonormal and the coefficients are appropriately weighted according to their importance in reconstructing the original data. This is obviously crucial when thresholding coefficients based on a given (limited) amount of storage space. When all dimensions are of equal extents (i.e., $|D_1| = |D_2| = \ldots = |D_d| = 2^m$), we can normalize coefficient values by simply dividing each coefficient with $\sqrt{2^l}^d$, where l is the level of resolution for the coefficient. As for one-dimensional wavelets, this normalization ensures the orthonormality of the Haar basis [137]. The following lemma shows how to extend the normalization process for nonstandard Haar coefficients to the important case of unequal dimension extents. (The proof follows by a simple verification of the orthonormality property for the constructed coefficients.)

Lemma 7.2.4: Let W be any wavelet coefficient generated by pairwise averaging and differencing during the nonstandard d-dimensional Haar decomposition of $A = |D_1| \times \cdots \times |D_d|$. Also, let $W.R.length[j] := W.R.boundary[j].hi - W.R.boundary[j].lo + 1$ denote the extent of W along dimension j, for each $1 \leq j \leq d$. Then, dividing the value $W.v$ of each coefficient W by the factor $\prod_j \sqrt{\frac{|D_j|}{W.R.length[j]}}$ gives an *orthonormal basis*. ▮

Coefficient Thresholding. Given a limited amount of storage for maintaining the wavelet-coefficient synopsis of R, we can only retain a certain number C of the coefficients stored in W_R. (The remaining coefficients are implicitly set to 0.) Typically, we have $C << N_z$, which implies that the chosen C wavelet coefficients form a highly compressed approximate representation of the original relational data. The goal of coefficient thresholding is to determine the "best" subset of C coefficients to retain, so that the error in the approximation is minimized.

The thresholding scheme that we have employed for the purposes of this study is to retain the C largest wavelet coefficients in *absolute normalized value*. It is a well-known fact that (for any orthonormal wavelet basis) this thresholding method is in fact *provably optimal* with respect to minimizing the overall mean squared error (i.e., L^2 error norm) in the data compression [137]. Given that our goal in this work is to support effective and accurate *general* query processing over such wavelet-compressed relational tables, we felt that the L^2 error norm would provide a reasonable aggregate metric of the accuracy of the approximation over all the individual tuples of R. Our thresholding approach is also validated by earlier results, where it has been proven that minimizing the L^2 approximation error is in fact optimal (on the average) for estimating the sizes of join query results [69]. [4] For the remainder of the chapter, we use the symbol W_R to denote the set of wavelet coefficients *retained* from the decomposition of relation R (i.e., the *wavelet-coefficient synopsis* of R).

Rendering a Wavelet-Coefficient Synopsis. A crucial requirement for any lossy data-compression scheme is the ability to reconstruct an approximate version of the original data from a given compressed representation. In our context, this requirement translates to *rendering* a given set of wavelet coefficients $W_T = \{W_i = \langle R_i, S_i, v_i \rangle\}$ corresponding to a relational table T, to produce an "approximate version" of T that we denote by `render`(W_T). It is important to note that T can correspond to either a base relation or the result of an arbitrarily complex SQL query on base relations. As we show in Section 7.3, our approximate query execution engine does the bulk of its processing directly over the wavelet coefficient domain. This means that producing the final approximate query answer in "human-readable" form can always be done by placing a `render`() operator at the root of the query plan or as a post-processing step.

Abstractly, the approximate relation `render`(W_T) can be constructed by summing up the contributions of every coefficient W_i in W_T to the appropriate cells of the (approximate) MOLAP array A_T. Consider a cell in A_T with coordinates $(i_1, \ldots, i_d)$ that is contained in the W_i's support hyper-rectangle $W_i.R$. Then, the contribution of W_i to $A_T[i_1, \ldots, i_d]$ is exactly $W_i.v \cdot \prod_{1 \leq j \leq d} s_j$, where $s_j = W.S.sign[j].lo$ if $i_j < W.S.signchange[j]$; otherwise, $s_j = W.S.sign[j].hi$. Once the counts for all the cells in the approximate MOLAP array A_T have been computed, the non-zero cells can be used to generate the tuples in the approximate relation `render`(W_T). In Section 7.3.5, we present an efficient algorithm for rendering a set of wavelet coefficients W_T to an approximate MOLAP representation. (The tuple generation step is then trivial.)

7.3 Processing Relational Queries in the Wavelet-Coefficient Domain

In this section, we propose a novel query algebra for wavelet-coefficient synopses. The basic operators of our algebra correspond directly to conventional relational algebra and SQL operators, including the (non-aggregate) `select`, `project`, and `join`, as well as aggregate operators like `count`, `sum`, and `av-`

[4]Note that it is possible to optimize the COMPUTEWAVELET algorithm for this thresholding scheme (e.g, do not perform Steps 6-12 for coefficients with absolute normalized value less than the C best coefficients found so far). We do not incorporate those optimizations into COMPUTEWAVELET in order to keep it independent of the thresholding scheme. This will allow us to try out new thresholding approaches in the future without having to change COMPUTEWAVELET .

`erage`. There is, however, one crucial difference: our operators are defined *over the wavelet-coefficient domain*; that is, their input(s) and output are *sets of wavelet coefficients* (rather than relational tables). The motivation for defining a query algebra for wavelet coefficients comes directly from the need for efficient approximate query processing. To see this, consider an n-ary relational query Q over $R_1, \ldots, R_n$ and assume that each relation R_i has been reduced to a (truncated) set of wavelet coefficients W_{R_i}. A simplistic way of processing Q would be to render each synopsis W_{R_i} into the corresponding approximate relation (denoted `render`(W_{R_i})) and process the relational operators in Q over the resulting sets of tuples. This strategy, however, is clearly inefficient: the approximate relation `render`(W_{R_i}) may contain just as many tuples as the original R_i itself, which implies that query execution costs may also be just as high as those of the original query. Therefore, such a "render-then-process" strategy essentially defeats one of the main motivations behind approximate query processing.

On the other hand, the synopsis W_{R_i} is a highly-compressed representation of `render`(W_{R_i}) that is typically orders of magnitude smaller than R_i. Executing Q in the compressed wavelet-coefficient domain (essentially, postponing `render`-ing until the final query result) can offer tremendous speedups in query execution cost. We therefore define the operators `op` of our query processing algebra over wavelet-coefficient synopses, while guaranteeing the valid semantics depicted pictorially in the transition diagram of Figure 7.6. (These semantics can be translated to the equivalence `render`(`op`$(T_1, \ldots, T_k)) \equiv$ `op`(`render`$(T_1, \ldots, T_k)$), for each operator `op`.) Our algebra allows the fast execution of any relational query Q *entirely* over the wavelet-coefficient domain, while guaranteeing that the final (rendered) result is identical to that obtained by executing Q on the approximate input relations.

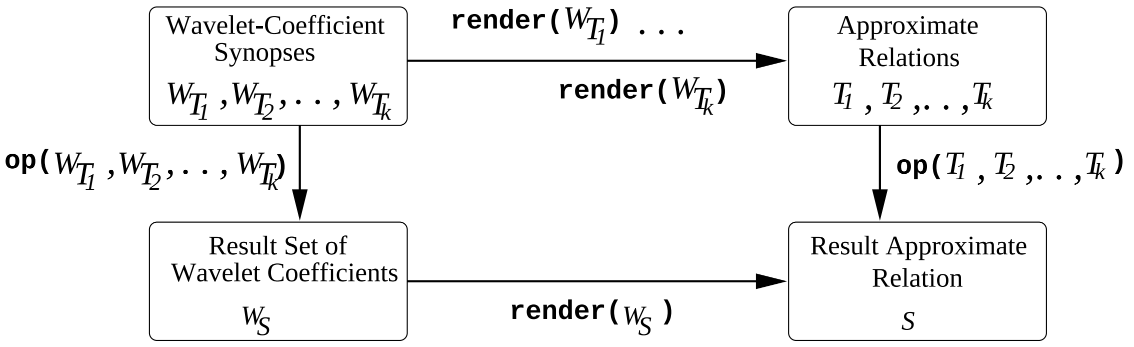

Wavelet-Coefficient Synopses
$W_{T_1}, W_{T_2}, \ldots, W_{T_k}$
render(W_{T_1}) ...
render(W_{T_k})
Approximate Relations
$T_1, T_2, \ldots, T_k$
op($W_{T_1}, W_{T_2}, \ldots, W_{T_k}$)
op($T_1, T_2, \ldots, T_k$)
Result Set of Wavelet Coefficients
W_S
render(W_S)
Result Approximate Relation
S

Figure 7.6: Valid semantics for processing query operators over the wavelet-coefficient domain.

In the following subsections, we describe our algorithms for processing the SQL operators in the wavelet-coefficient domain. Each operator takes as input one or more set(s) of multi-dimensional wavelet coefficients and appropriately combines and/or updates the components (i.e., hyper-rectangle, sign information, and magnitude) of these coefficients to produce a "valid" set of output coefficients (Figure 7.6). Note that, while the wavelet coefficients (generated by COMPUTEWAVELET) for base relational tables have a very regular structure, the same is not necessarily true for the set of coefficients output by an arbitrary `select` or `join` operator. Nevertheless, we loosely continue to refer to the intermediate results of our algebra operators as "wavelet coefficients" since they are characterized by the exact same components as base-relation coefficients (e.g., hyper-rectangle, sign-vectors) and maintain the exact same semantics with respect to the underlying intermediate relation (i.e., the rendering process remains unchanged).

7.3.1 Selection Operator (`select`)

Our selection operator has the general form $\texttt{select}_{pred}(W_T)$, where $pred$ represents a generic conjunctive predicate on a subset of the d attributes in T; that is, $pred = (l_{i_1} \leq X_{i_1} \leq h_{i_1}) \wedge \ldots \wedge (l_{i_k} \leq X_{i_k} \leq h_{i_k})$, where l_{i_j} and h_{i_j} denote the low and high boundaries of the selected range along each selection dimension D_{i_j}, $j = 1, 2, \cdots, k$, $k \leq d$. This is essentially a k-dimensional range selection, where the queried range is specified along k dimensions $\mathcal{D}' = \{D_{i_1}, D_{i_2}, \ldots, D_{i_k}\}$ and left unspecified along the remaining $(d-k)$ dimensions $(\mathcal{D} - \mathcal{D}')$. ($\mathcal{D} = \{D_1, D_2, \ldots, D_d\}$ denotes the set of all dimensions of T.) Thus, for each unspecified dimension D_j, the selection range spans the full index domain along the dimension; that is, $l_j = 0$ and $h_j = |D_j| - 1$, for each $D_j \in (\mathcal{D} - \mathcal{D}')$.

The `select` operator effectively filters out the portions of the wavelet coefficients in the synopsis W_T that do not overlap with the k-dimensional selection range, and thus do not contribute to cells in the selected hyper-rectangle. This process is illustrated pictorially in Figure 7.7. More formally, let $W \in W_T$ denote any wavelet coefficient in the input set of our `select` operator. Our approximate query execution engine processes the selection over W as follows. If W's support hyper-rectangle $W.R$ overlaps the k-dimensional selection hyper-rectangle; that is, if *for every* dimension $D_{i_j} \in \mathcal{D}'$, the following condition is satisfied:

$$l_{i_j} \leq W.R.boundary[i_j].lo \leq h_{i_j} \qquad \text{or} \qquad W.R.boundary[i_j].lo \leq l_{i_j} \leq W.R.boundary[i_j].hi,$$

then

1. For all dimensions $D_{i_j} \in \mathcal{D}'$ do
   - 1.1. Set $W.R.boundary[i_j].lo := \max\{l_{i_j}, W.R.boundary[i_j].lo\}$ and $W.R.boundary[i_j].hi := \min\{h_{i_j}, W.R.boundary[i_j].hi\}$.
   - 1.2. If $W.R.boundary[i_j].hi < W.S.signchange[i_j]$ then set $W.S.signchange[i_j] := W.R.boundary[i_j].lo$ and $W.S.sign[i_j] := [W.S.sign[i_j].lo, W.S.sign[i_j].lo]$.
   - 1.3. Else if $W.R.boundary[i_j].lo \geq W.S.signchange[i_j]$ then set $W.S.signchange[i_j] := W.R.boundary[i_j].lo$ and $W.S.sign[i_j] := [W.S.sign[i_j].hi, W.S.sign[i_j].hi]$.
2. Add the (updated) W to the set of output coefficients; that is, set $W_S := W_S \cup \{W\}$, where $S = \texttt{select}_{pred}(T)$.

Our `select` processing algorithm chooses (and appropriately updates) only the coefficients in W_T that overlap with the k-dimensional selection hyper-rectangle. For each such coefficient, our algorithm (a) updates the hyper-rectangle boundaries according to the specified selection range (Step 1.1), and (b) updates

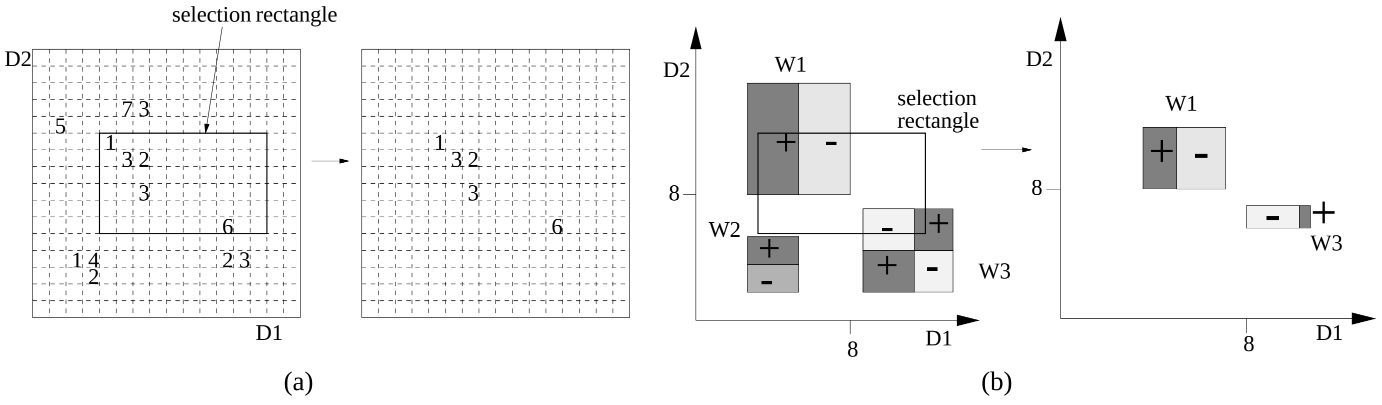


Figure 7.7: (a) Processing selection operation in the relation domain. (b) Processing selection operation in the wavelet-coefficient domain.

the sign information, if such an update is necessary (Steps 1.2-1.3). Briefly, the sign information along the queried dimension D_{i_j} needs to be updated only if the selection range along D_{i_j} is completely contained in either the low (1.2) or the high (1.3) sign-vector range of the coefficient along D_{i_j}. In both cases, the sign-vector of the coefficient is updated to contain only the single sign present in the selection range and the coefficient's sign-change is set to its leftmost boundary value (since there is no change of sign along D_{i_j} after the selection). The sign-vector and sign-change of the result coefficient remain untouched (i.e., identical to those of the input coefficient) if the selection range spans the original sign-change value.

Example 7.3.1: Figure 7.7(a) shows the semantics of the selection operation in the relation domain. A relation T with 2 dimensions ($|D_1| = 16, |D_2| = 16$) is shown in its JFD representation A_T. The `select` operator is a 2-dimensional selection hyper-rectangle with boundaries $[l_1, h_1] = [4, 13]$ and $[l_2, h_2] = [5, 10]$ along dimensions D_1 and D_2 respectively. The output of the operation consists of only those tuples that fall inside the selection hyperrectangle.

Figure 7.7(b) shows the semantics of the same selection operation in the wavelet domain. We illustrate the processing for one of the wavelet coefficients: the others are processed similarly. Consider the wavelet coefficient W_3 having hyper-rectangle ranges $W_3.R.boundary[1] = [9, 15]$ and $W_3.R.boundary[2] = [2, 7]$. The sign information for W_3 is $W_3.S.sign[1] = [+, -]$, $W_3.S.sign[2] = [+, -]$ (Figure 7.2(b)), $W_3.S.signchange[1] = 12$, and $W_3.S.signchange[2] = 4$. Since W_3's hyper-rectangle overlaps with the selection hyper-rectangle, it is processed by the `select` operator as follows. First, in Step 1.1, the hyper-rectangle boundaries of W_3 are updated to $W_3.R.boundary[1] := [9, 13]$ and $W_3.R.boundary[2] := [5, 7]$ (i.e., the region that overlaps with the select ranges along D_1 and D_2). Since $W_3.S.signchange[1] = 12$ which is between 9 and 13 (the new boundaries along D_1), the sign information along D_1 is not updated. Along dimension D_2, however, we have $W_3.S.signchange[2] = 4$ which is less than $W_3.R.boundary[2].lo = 5$, and so Step 1.3 updates the sign information along D_2 to $W_3.S.sign[2] := [-, -]$ and $W_3.S.signchange[2] :=$

5 (i.e., the low boundary along D_2). ∎

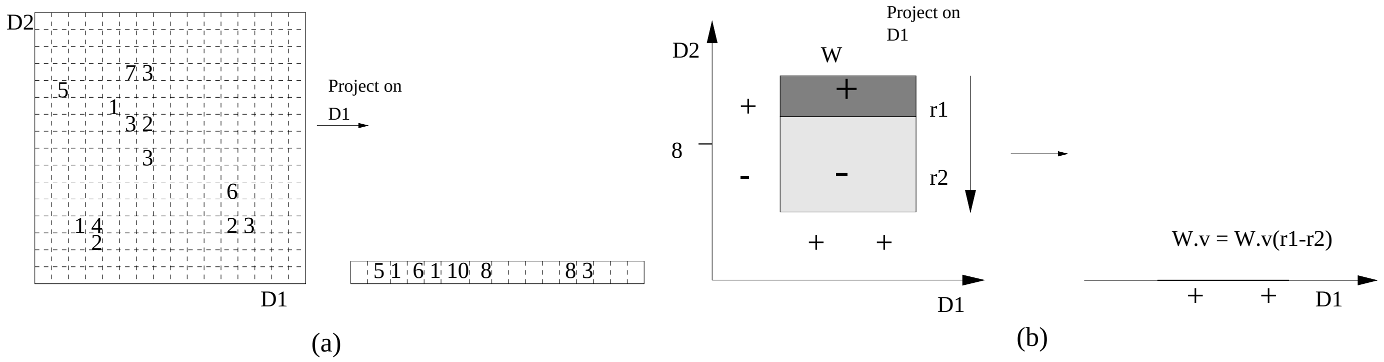


Figure 7.8: (a) Processing projection operation in the relation domain. (b) Processing projection operation in the wavelet-coefficient domain.

7.3.2 Projection Operator (`project`)

Our projection operator has the general form $\texttt{project}_{X_{i_1},\ldots,X_{i_k}}(W_T)$, where the k projection attributes $X_{i_1},\ldots,X_{i_k}$ form a subset of the d attributes of T. Letting $\mathcal{D}' = \{D_{i_1},\ldots,D_{i_k}\}$ denote the $k \leq d$ projection dimensions, we are interested in *projecting out* the $d-k$ dimensions in $(\mathcal{D} - \mathcal{D}')$. We give a general method for projecting out a single dimension $D_j \in \mathcal{D} - D'$. This method can then be applied repeatedly to project out all the dimensions in $(\mathcal{D} - \mathcal{D}')$, one dimension at a time.

Consider T's corresponding multi-dimensional array A_T. Projecting a dimension D_j out of A_T is equivalent to summing up the counts for all the array cells in each one-dimensional row of A_T along dimension D_j and then assigning this aggregated count to the single cell corresponding to that row in the remaining dimensions $(\mathcal{D} - \{D_j\})$. The above process is illustrated with an example 2-dimensional array A_T in Figure 7.8(a). Consider any d-dimensional wavelet coefficient W in the `project` operator's input set W_T. Remember that W contributes a value of $W.v$ to every cell in its support hyper-rectangle $W.R$. Furthermore, the sign of this contribution for every one-dimensional row along dimension D_j is determined as either $W.S.sign[j].hi$ (if the cell lies above $W.S.signchange[j]$) or $W.S.sign[j].lo$ (otherwise). Thus, we can work directly on the coefficient W to project out dimension D_j by simply adjusting the coefficient's magnitude with an appropriate multiplicative constant $W.v := W.v * p_j$, where:

$$\begin{aligned} p_j \;=\; & (W.R.boundary[j].hi - W.S.signchange[j] + 1) * W.S.sign[j].hi \;+ \\ & (W.S.signchange[j] - W.R.boundary[j].lo) * W.S.sign[j].lo. \end{aligned} \tag{7.1}$$

A two-dimensional example of projecting out a dimension in the wavelet-coefficient domain is depicted in

Figure 7.8(b). Multiplying $W.v$ with p_j (Equation (7.1)) effectively projects out dimension D_j from W by summing up W's contribution on each one-dimensional row along dimension D_j. Of course, besides adjusting $W.v$, we also need to discard dimension D_j from the hyper-rectangle and sign information for W, since it is now a $(d-1)$-dimensional coefficient (on dimensions $\mathcal{D}-\{D_j\}$). Note that if the coefficient's sign-change lies in the middle of its support range along dimension D_j (e.g., see Figure 7.2(a)), then its adjusted magnitude will be 0, which means that it can safely be discarded from the output set of the projection operation.

Repeating the above process for each wavelet coefficient $W \in W_T$ and each dimension $D_j \in \mathcal{D} - D'$ gives the set of output wavelet coefficients W_S, where $S = \texttt{project}_{\mathcal{D}'}(T)$. Equivalently, given a coefficient W, we can simply set $W.v := W.v * \prod_{D_j \in \mathcal{D}-D'} p_j$ (where p_j is as defined in Equation (7.1)) and discard dimensions $\mathcal{D} - D'$ from W's representation.

Example 7.3.2: Figure 7.8(a) shows the semantics of the projection operation in the relation domain. It shows the same 2-dimensional relation ($|D_1| = 16, |D_2| = 16$) from Example 7.3.1 and the result of its projection on dimension D_1.

Figure 7.8(b) shows the semantics of the projection operation in the wavelet domain. Consider the wavelet coefficient W whose hyper-rectangle and sign information along dimension D_2 are as follows: $W.R.boundary[2] = [4, 11]$, $W.S.sign[2] = [-, +]$, and $W.S.signchange[2] = 10$. Also, let the magnitude of W be $W.v = 2$. Then, projecting W on dimension D_1 causes $W.v$ to be updated to $W.v := 2 \cdot ((11 - 10 + 1) - (10 - 4)) = -8$. ∎

7.3.3 Join Operator (`join`)

Our join operator has the general form $\texttt{join}_{pred}(W_{T_1}, W_{T_2})$, where T_1 and T_2 are (approximate) relations of arity d_1 and d_2, respectively, and $pred$ is a conjunctive k-ary equi-join predicate of the form $(X_1^1 = X_1^2) \wedge \ldots \wedge (X_k^1 = X_k^2)$, where X_j^i (D_j^i) ($j = 1, \ldots, d_i$) denotes the j^{th} attribute (resp., dimension) of T_i ($i = 1, 2$). (Without loss of generality, we assume that the join attributes are the first $k \leq \min\{d_1, d_2\}$ attributes of each joining relation.) Note that the result of the join operation W_S is a set of $(d_1 + d_2 - k)$-dimensional wavelet coefficients; that is, the join operation returns coefficients of (possibly) different arity than any of its inputs.

To see how our join processing algorithm works, consider the multi-dimensional arrays A_{T_1} and A_{T_2} corresponding to the join operator's input arguments. Let $(i_1^1, \ldots, i_{d_1}^1)$ and $(i_1^2, \ldots, i_{d_2}^2)$ denote the coordinates of two cells belonging to A_{T_1} and A_{T_2}, respectively. If the indexes of the two cells match on the join dimensions, i.e., $i_1^1 = i_1^2, \ldots, i_k^1 = i_k^2$, then the cell in the join result array A_S with coordinates $(i_1^1, \ldots, i_{d_1}^1, i_{k+1}^2, \ldots, i_{d_2}^2)$ is populated with the *product* of the count values contained in the two joined cells. Figure 7.9(a) illustrates the above process with two example 2-dimensional arrays A_{T_1} (having dimensions D_1 and D_2, $|D_1| = |D_2| = 16$) and A_{T_2} (having dimensions D_1 and D_3, $|D_1| = |D_3| = 16$) and join dimension D_1. For example, the cells $(9, 6)$ in A_{T_1} (count value 2) and $(9, 2)$ in A_{T_2} (count value

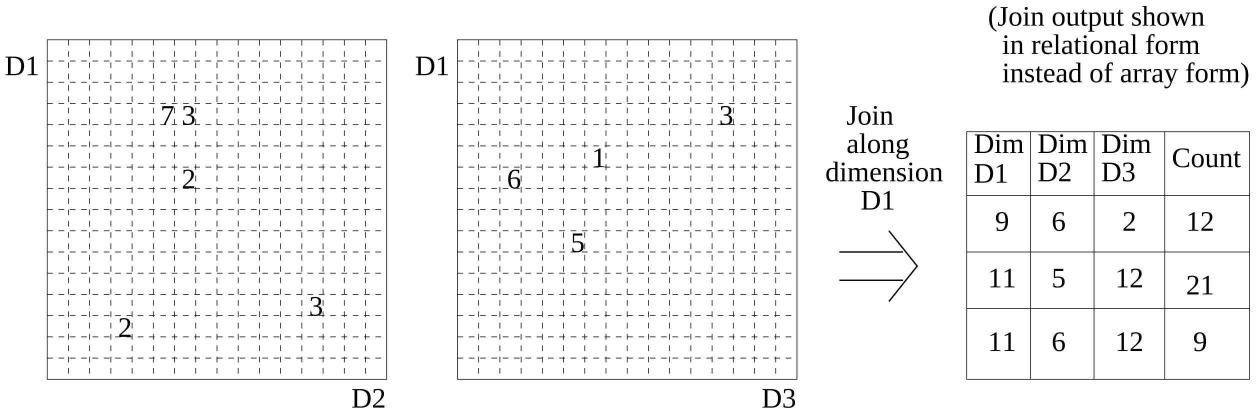
6) match on join dimension D_1 (both 9); hence the output is populated with the cell $(9,6,2)$ (count value $= 2 * 6 = 12$). Since the cell counts for A_{T_i} are derived by appropriately summing the contributions of the wavelet coefficients in W_{T_i} and, of course, a numeric product can always be distributed over summation, we can process the `join` operator entirely in the wavelet-coefficient domain by considering all pairs of coefficients from W_{T_1} and W_{T_2}. Briefly, for any two coefficients from W_{T_1} and W_{T_2} that overlap in the join dimensions and, therefore, contribute to joining data cells, we define an output coefficient with magnitude equal to the product of the two joining coefficients and a support hyper-rectangle with ranges that are (a) equal to the overlap of the two coefficients for the k (common) join dimensions, and (b) equal to the original coefficient ranges along any of the $d_1 + d_2 - 2k$ remaining dimensions. The sign information for an output coefficient along any of the k join dimensions is derived by appropriately multiplying the sign-vectors of the joining coefficients along that dimension, taking care to ensure that only signs along the overlapping portion are taken into account. (The sign information along non-join dimensions remains unchanged.) An example of this process in two dimensions ($d_1 = d_2 = 2$, $k = 1$) is depicted in Figure 7.9(b).

More formally, our approximate query execution strategy for joins can be described as follows. (To simplify the notation, we ignore the "1/2" superscripts and denote the join dimensions as $D_1, \ldots, D_k$, and the remaining $d_1 + d_2 - 2k$ dimensions as $D_{k+1}, \ldots, D_{d_1+d_2-k}$.) For each pair of wavelet coefficients $W_1 \in W_{T_1}$ and $W_2 \in W_{T_2}$, if the coefficients' support hyper-rectangles overlap in the k join dimensions; that is, if *for every* dimension D_i, $i = 1 \ldots, k$, the following condition is satisfied:

$$\begin{aligned} W_1.R.boundary.lo[i] \;\leq\; & W_2.R.boundary.lo[i] \;\leq\; W_1.R.boundary.hi[i] \qquad \text{or} \\ & W_2.R.boundary.lo[i] \;\leq\; W_1.R.boundary.lo[i] \;\leq\; W_2.R.boundary.hi[i], \end{aligned}$$

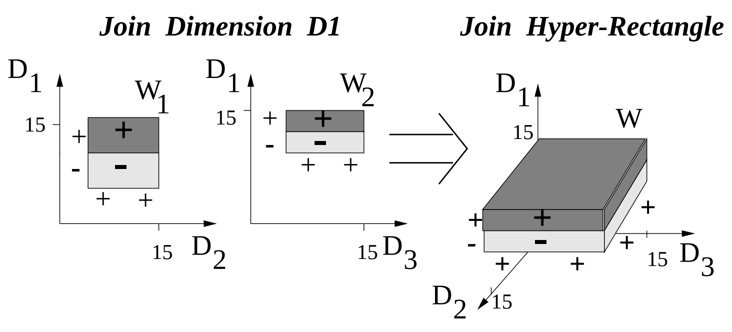
then the corresponding output coefficient $W \in W_S$ is defined in the following steps.

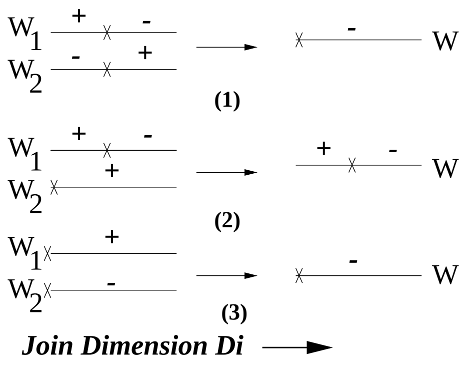
```
1. For all join dimensions $D_i$, $i = 1, \ldots, k$ do
   1.1. Set $W.R.boundary[i].lo := \max\{W_1.R.boundary[i].lo, W_2.R.boundary[i].lo\}$ and $W.R.boundary[i].hi := \min\{W_1.R.boundary[i].hi, W_2.R.boundary[i].hi\}$.
   1.2. For $j = 1, 2$        /* let $s_j$ be a temporary sign-vector variable */
      1.2.1. If $W.R.boundary[i].hi < W_j.S.signchange[i]$ then set $s_j := [W_j.S.sign[i].lo, W_j.S.sign[i].lo]$.
      1.2.2. Else if $W.R.boundary[i].lo \geq W_j.S.signchange[i]$ then set $s_j := [W_j.S.sign[i].hi, W_j.S.sign[i].hi]$.
      1.2.3. Else set $s_j := W_j.S.sign[i]$.
   1.3. Set $W.S.sign[i] := [s_1.lo * s_2.lo\,,\ s_1.hi * s_2.hi]$.
   1.4. If $W.S.sign[i].lo == W.S.sign[i].hi$ then set $W.S.signchange[i] := W.R.boundary[i].lo$.
   1.5 Else set $W.S.signchange[i] := \max_{j=1,2}\{W_j.S.signchange[i] : W_j.S.signchange[i] \in [W.R.boundary[i].lo, W.R.boundary[i].hi]\}$.
2. For each (non-join) dimension $D_i$, $i = k+1, \ldots, d_1$ do: Set $W.R.boundary[i] := W_1.R.boundary[i]$, $W.S.sign[i] := W_1.S.sign[i]$, and $W.S.signchange[i] := W_1.S.signchange[i]$.
3. For each (non-join) dimension $D_i$, $i = d_1+1, \ldots, d_1+d_2-k$ do: Set $W.R.boundary[i] := W_2.R.boundary[i - d_1 + k]$, $W.S.sign[i] := W_2.S.sign[i - d_1 + k]$, and $W.S.signchange[i] := W_2.S.signchange[i - d_1 + k]$.
```


D1
7 3
2
3
2
D2
D1
3
1
6
5
D3
Join along dimension D1
(Join output shown in relational form instead of array form)

Dim D1	Dim D2	Dim D3	Count
9	6	2	12
11	5	12	21
11	6	12	9

(a)





(b) (c)

Figure 7.9: (a) Processing `join` operation in the relation domain. (b) Processing `join` operation in the wavelet-coefficient domain. (c) Computing sign information for `join` output coefficients.

4. Set $W.v := W_1.v * W_2.v$ and $W_S := W_S \cup \{W\}$, where $S = \texttt{join}_{pred}(T_1, T_2)$.

Note that the bulk of our join processing algorithm concentrates on the correct settings for the output coefficient W along the k join dimensions (Step 1), since the problem becomes trivial for the $d_1 + d_2 - 2k$ remaining dimensions (Steps 2-3). Given a pair of joining input coefficients and a join dimension D_i, our algorithm starts out by setting the hyper-rectangle range of the output coefficient W along D_i equal to the overlap of the two input coefficients along D_i (Step 1.1). We then proceed to compute W's sign information along join dimension D_i (Steps 1.2-1.3) , which is slightly more involved. (Remember that T_1 and T_2 are (possibly) the results of earlier `select` and/or `join` operators, which means that their rectangle boundaries and signs along D_i can be arbitrary.) The basic idea is to determine, for each of the two input coefficients W_1 and W_2, where the boundaries of the join range lie with respect to the coefficient's sign-change value along dimension D_i. Given an input coefficient W_j ($j = 1, 2$), if the join range along D_i is completely contained in either the low (1.2.1) or the high (1.2.2) sign-vector range of W_j along D_i, then a temporary sign-vector

s_j is appropriately set (with the same sign in both entries). Otherwise, i.e., if the join range spans W_j's sign-change (1.2.3), then s_j is simply set to W_j's sign-vector along D_i. Thus, s_j captures the sign of coefficient W_j in the joining range, and multiplying s_1 and s_2 (element-wise) yields the sign-vector for the output coefficient W along dimension D_i (Step 1.3). If the resulting sign vector for W does not contain a true sign change (i.e., the low and high components of $W.S.sign[i]$ are the same), then W's sign-change value along dimension D_i is set equal to the low boundary of $W.R$ along D_i, according to our convention (Step 1.4). Otherwise, the sign-change value for the output coefficient W along D_i is set equal to the maximum of the input coefficients' sign-change values that are contained in the join range (i.e., $W.R$'s boundaries) along D_i (Step 1.5).

In Figure 7.9(c), we illustrate three common scenarios for the computation of W's sign information along the join dimension D_i. The left-hand side of the figure shows three possibilities for the sign information of the input coefficients W_1 and W_2 along the join range of dimension D_i (with crosses denoting sign changes). The right-hand side depicts the resulting sign information for the output coefficient W along the same range. The important thing to observe with respect to our sign-information computation in Steps 1.3–1.5 is that the join range along any join dimension D_i can contain *at most one* true sign change. By this, we mean that if the sign for input coefficient W_j actually changes in the join range along D_i, then this sign-change value is unique; that is, the two input coefficients cannot have true sign changes at distinct points of the join range. This follows from the *complete containment* property of the base coefficient ranges along dimension D_i (Section 7.2.1). (Note that our algorithm for `select` retains the value of a true sign change for a base coefficient if it is contained in the selection range, and sets it equal to the value of the left boundary otherwise.) This range containment along D_i ensures that if W_1 and W_2 both contain a true sign change in the join range (i.e., their overlap) along D_i, then that will occur *at exactly the same value* for both (as illustrated in Figure 7.9(c.1)). Thus, in Step 1.3, W_1's and W_2's sign vectors in the join range can be multiplied to derive W's sign-vector. If, on the other hand, one of W_1 and W_2 has a true sign change in the join range (as shown in Figure 7.9(c.2)), then the $\max$ operation of Step 1.5 will always set the sign change of W along D_i correctly to the true sign-change value (since the other sign change will either be at the left boundary or outside the join range). Finally, if neither W_1 nor W_2 have a true sign change in the join range, then the high and low components of W's sign vector will be identical and Step 1.4 will set W's sign-change value correctly.

Example 7.3.3: Figure 7.9(a) shows the semantics of join operation in the relation domain as explained before. Figure 7.9(b) and (c) shows the semantics of the operation in the wavelet domain. Consider the wavelet coefficients W_1 and W_2. Let the boundaries and sign information of W_1 and W_2 along the join dimension D_1 be as follows: $W_1.R.boundary[1] = [4, 15]$, $W_2.R.boundary[1] = [8, 15]$, $W_1.S.sign[1] = [-, +]$, $W_2.S.sign[1] = [-, +]$, $W_1.S.signchange[1] = 8$, and $W_2.S.signchange[1] = 12$. In the following, we illustrate the computation of the hyper-rectangle and sign information for join dimension D_1 for the coefficient W that is output by our algorithm when W_1 and W_2 are "joined". Note that for the non-join dimensions D_2 and D_3, this information for W is identical to that of W_1 and W_2 (respectively), so we focus solely on the join dimension D_1.

First, in Step 1.1, $W.R.boundary[1]$ is set to $[8, 15]$, i.e., the overlap range between W_1 and W_2 along D_1. In Step 1.2.2, since $W.R.boundary[1].lo = 8$ is greater than or equal to $W_1.S.signchange[1] = 8$, we set $s_1 = [+, +]$. In Step 1.2.3, since $W_2.S.signchange[1] = 12$ lies in between $W.R$'s boundaries, we set $s_2 = [-, +]$. Thus, in Step 1.3, $W.S.sign[1]$ is set to the product of s_1 and s_2 which is $[-, +]$. Finally, in Step 1.5, $W.S.signchange[1]$ is set to the maximum of the sign change values for W_1 and W_2 along dimension D_1, or $W.S.signchange[1] := \max\{8, 12\} = 12$. ∎

7.3.4 Aggregate Operators

In this section, we show how conventional aggregation operators, like `count`, `sum`, and `average`, are realized by our approximate query execution engine in the wavelet-coefficient domain[5]. As before, the input to each aggregate operator is a set of wavelet coefficients W_T. If the aggregation is not qualified with a `GROUP-BY` clause, then the output of the operator is a simple scalar value for the aggregate. In the more general case, where a `GROUP-BY` clause over dimensions $\mathcal{D}' = \{D_1, \ldots, D_k\}$ has been specified, the output of the aggregate operator consists of a k-dimensional array spanning the dimensions in $\mathcal{D}'$, whose entries contain the computed aggregate value for each cell.

Note that, unlike our earlier query operators, we define our aggregate operators to provide output that is essentially a rendered data array, rather than a set of (un-rendered) wavelet coefficients. This is because there is no clean, general method to map the computed aggregate values (e.g., attribute sums or averages) onto the semantics and structure of wavelet coefficients. We believe, however, that exiting the coefficient domain after aggregation has no negative implications for the effectiveness of our query execution algorithms. The reason is that, for most DSS queries containing aggregation, the aggregate operator is the final operator at the root of the query execution plan, which means that its result would have to be rendered anyway.

While the earlier work of Vitter and Wang [144] has addressed the computation of aggregates over a wavelet-compressed relational table, their approach is significantly different from ours. Vitter and Wang focus on a very specific form of aggregate queries, namely *range-sum queries*, where the range(s) are specified over one or more *functional attribute* and the summation is done over a prespecified *measure attribute*. Their wavelet decomposition and aggregation algorithm are both geared towards this specific type of queries that essentially treats the relation's attributes in an "asymmetric" manner (by distinguishing the single measure attribute). Our approach, on the other hand, has a much broader query processing scope. As a result, all attributes are treated in a completely symmetric fashion, thus enabling us to perform a broad range of aggregate (and non-aggregate) operations over *any* attribute(s).

[5]Like most conventional data reduction and approximate querying techniques (e.g., sampling and histograms), wavelets are inherently limited to "trivial answers" when it comes to `min` or `max` aggregate functions (see, for example, [70]). In our case, this would amount to selecting the *non-zero* cell in the reconstructed array with minimum/maximum coordinate along the specified query range. We do not consider `min` or `max` aggregates further in this chapter.

Count Operator (count). Our count operator has the general form $\texttt{count}_{\mathcal{D}'}(W_T)$, where the k GROUP-BY dimensions $\mathcal{D}' = \{D_{i_1}, \ldots, D_{i_k}\}$ form a (possibly empty) subset of the d attributes of T. Counting is the most straightforward aggregate operation to implement in our framework, since each cell in our approximate multi-dimensional array already stores the count information for that cell. Thus, processing $\texttt{count}_{\mathcal{D}'}(W_T)$ is done by simply projecting each input coefficient onto the GROUP-BY dimensions $\mathcal{D}'$ and rendering the result into a multi-dimensional array of counts, as follows.

1. Let $W_S := \texttt{project}_{\mathcal{D}'}(W_T)$ (see Section 7.3.2).
2. Let $A_S := \texttt{render}(W_S)$ and output the cells in the $|\mathcal{D}'|$-dimensional array A_S with non-zero counts.

Sum Operator (sum). The general form of our summation operator is $\texttt{sum}_{\mathcal{D}'}(W_T, D_j)$, where $\mathcal{D}' = \{D_{i_1}, \ldots, D_{i_k}\}$ denotes the set of GROUP-BY dimensions and $D_j \notin \mathcal{D}'$ corresponds to the attribute of T whose values are summed. The sum operator is implemented in three steps. First, we project the input coefficients W_T on dimensions $\mathcal{D}' \cup \{D_j\}$. Second, for each coefficient W output by the first step and for each row of cells along the summation attribute D_j, we compute the sum of the product of the coefficient's magnitude $W.v$ and the index of the cell along D_j [6]. This sum (essentially, an *integral* along D_j) is then assigned to the coefficient's magnitude $W.v$ and the summing dimension D_j is discarded. Thus, at the end of this step, $W.v$ stores the contribution of W to the summation value for every $|\mathcal{D}'|$-dimensional cell in $W.R$. Third, the resulting set of wavelet coefficients is rendered to produce the output multi-dimensional array on dimensions $\mathcal{D}'$. More formally, our $\texttt{sum}_{\mathcal{D}'}(W_T, D_j)$ query processing algorithm comprises the following steps.

1. Let $W_S := \texttt{project}_{\mathcal{D}' \cup \{D_j\}}(W_T)$ (Section 7.3.2).
2. For each wavelet coefficient $W \in W_S$ do
   - 2.1. Set $W.v$ according to the following equation:

$$W.v := W.v \cdot \left(W.S.sign[j].lo \cdot \sum_{k=W.R.boundary[j].lo}^{W.S.signchange[j]-1} k \; + \; W.S.sign[j].hi \cdot \sum_{k=W.S.signchange[j]}^{W.R.boundary[j].hi} k \right).$$

     Note that, the summations of the index values along D_j in the above formula can be expressed in closed form using straightforward algebraic methods.
   - 2.2. Discard dimension D_j from the hyper-rectangle and sign information for W.
3. Let $A_S := \texttt{render}(W_S)$ and output the cells in the $|\mathcal{D}'|$-dimensional array A_S with non-zero values for the summation.

[6]To simplify the exposition, we assume that the (integer) cell index values along dimension D_j are identical to the domain values for the corresponding attribute X_j of T. If that is not the case, then a reverse mapping from the D_j index values to the corresponding values of X_j is needed to sum the attribute values along the boundaries of a coefficient.

Average Operator (average). The averaging operator $\texttt{average}_{\mathcal{D}'}(W_T, D_j)$ (where $\mathcal{D}'$ is the set of GROUP-BY dimensions and $D_j \notin \mathcal{D}'$ corresponds to the averaged attribute of T) is implemented by combining the computation of $\texttt{sum}_{\mathcal{D}'}(W_T, D_j)$ and $\texttt{count}_{\mathcal{D}'}(W_T)$. The idea is to compute the attribute sums and tuple counts for every cell over the data dimensions in the GROUP-BY attributes $\mathcal{D}'$, as described earlier in this section. We then render the resulting coefficients and output the average value (i.e., the ratio of sum over count) for every cell with non-zero sum and count.

7.3.5 Rendering a Set of Wavelet Coefficients

Since our approximate query execution engine does the bulk of its processing in the wavelet coefficient domain, an essential final step for every user query is to *render* an output set W_S of d-dimensional wavelet coefficients (over, say, $\mathcal{D} = \{D_1, \ldots, D_d\}$) to produce the approximate query answer in a "human-readable" form. (Note that rendering is required as a final step even for the aggregate processing algorithms described in the previous section.) The main challenge in the rendering step is how to *efficiently* expand the input set of d-dimensional wavelet coefficients W_S into the corresponding (approximate) d-dimensional array of counts A_S.

A naive approach to rendering W_S would be to simply consider each cell in the multi-dimensional array A_S and sum the contributions of every coefficient $W \in W_S$ to that cell in order to obtain the corresponding tuple count. However, the number of cells in A_S is potentially huge, which implies that such a naive rendering algorithm could be extremely inefficient and computationally expensive (typically, of order $O(N \cdot |W_S|)$, where $N = \prod_{i=1}^{d} |D_i|$ is the number of array cells). Instead of following this naive and expensive strategy, we propose a more efficient algorithm (termed render) for rendering an input set of multi-dimensional wavelet coefficients. (Note that render can be seen either as a (final) query processing operator or as a post-processing step for the query.) Our algorithm exploits the fact that the number of coefficients in W_S is typically *much smaller* than the number of array cells N. This implies that we can expect A_S to consist of large, contiguous multi-dimensional regions, where all the cells in each region contain exactly the same count. (In fact, because of the sparsity of the data, many of these regions will have counts of 0.) Furthermore, the total number of such "uniform-count" regions in A_S is typically considerably smaller that N. Thus, the basic idea of our efficient rendering algorithm is to partition the multi-dimensional array A_S, one dimension at a time, into such uniform-count data regions and output the (single) count value corresponding to each such region (the same for all enclosed cells).

Our render algorithm (depicted in Figure 7.10) recursively partitions the d-dimensional data array A_S, one dimension at a time and in the dimension order $D_1, \ldots, D_d$. Algorithm render takes two input arguments: (a) the index (i) of the next dimension D_i along which the array A_S is to be partitioned, and (b) the set of wavelet coefficients (COEFF) in the currently processed partition of A_S (generated by the earlier partitionings along dimensions $D_1, \ldots, D_{i-1}$). The initial invocation of render is done with $i = 1$ and COEFF $= W_S$.

```
procedure render(COEFF, $i$)
begin
1.  if ($i > d$) {
2.      $count$ := 0
3.      for each coefficient $W$ in COEFF
4.          $sign := \prod_{D_j \in \mathcal{D}} sign_j$
          /* $sign_j := W.S.sign[j].lo$ if $W.R.boundary[j].lo < W.S.signchange[j]$; else, $sign_j := W.S.sign[j].hi$ */
5.          $count := count + sign * W.v$
6.      output ($W.R.boundary, count$)        /* $W$ is any coefficient in COEFF */
7.      return
8.  }
9.  $Q := \emptyset$        /* elements $e$ in priority queue $Q$ are sorted in increasing order of $e.key$ */
10. for each coefficient $W$ in COEFF
11.     insert element $e$ into $Q$ where $e.key := W.R.boundary[i].lo - 1$ and $e.val := W$
12.     insert element $e$ into $Q$ where $e.key := W.R.boundary[i].hi$ and $e.val := W$
13.     if ($W.R.boundary[i].lo < W.S.signchange[i] \leq W.R.boundary[i].hi$)
14.         insert element $e$ into $Q$ where $e.key := W.S.signchange[i] - 1$ and $e.val := W$
15. $prev := -\infty$, TEMP1 $:= \emptyset$
16. while ($Q$ is not empty) do {
17.     TEMP2 $:= \emptyset$, $topkey := e.key$ for element $e$ at head of $Q$
18.     dequeue all elements $e$ with $e.key = topkey$ at the head of $Q$ and insert $e.val$ into TEMP1
19.     for each coefficient $W$ in TEMP1
20.         delete $W$ from TEMP1 if $W.R.boundary[i].hi < prev + 1$
21.         if $W.R.boundary[i]$ overlaps with the interval $[prev + 1, topkey]$ along dimension $D_i$
22.             $W' := W$
23.             $W'.R.boundary[i].lo := prev + 1$, $W'.R.boundary[i].hi := topkey$
24.             insert $W'$ into TEMP2
25.     render(TEMP2, $i + 1$)
26.     $prev := topkey$
27. }
end
```

Figure 7.10: render: An efficient algorithm for rendering multi-dimensional wavelet coefficients.

When partitioning A_S into uniform-count ranges along dimension D_i, the only points that should be considered are those where the cell counts along D_i could potentially change. These are precisely the points where a new coefficient W starts contributing ($W.R.boundary[i].lo$), stops contributing ($W.R.boundary[i].hi$), or the sign of its contribution changes ($W.S.signchange[i]$). Algorithm render identifies these points along dimension D_i for each coefficient in COEFF and stores them in sorted order in a priority queue Q (Steps 10–14). Note that, for any pair of consecutive partitioning points along D_i, the contribution of *each* coefficient in COEFF (and, therefore, their sum) is guaranteed to be *constant* for any row of cells along D_i between the two points. Thus, abstractly, our partitioning generates one-dimensional uniform-count ranges along D_i. Once the partitioning points along dimension D_i have been determined, they are used to partition the hyper-rectangles of the wavelet coefficients in COEFF along D_i (Steps 16–27). Algorithm render is then recursively invoked with the set of (partial) coefficients in each partition of D_i to further partition the coefficients along the remaining dimensions $D_{i+1}, \ldots, D_d$. Once the array has been partitioned along all

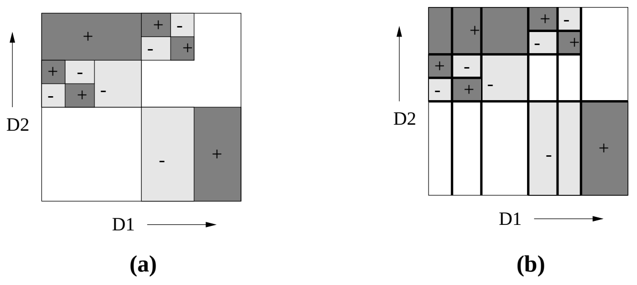


Figure 7.11: Partitioning a two-dimensional array by procedure `render`.

dimensions in $\mathcal{D}$ (i.e., `render` is invoked with parameter $i > d$), a coefficient W in the input set of coefficients COEFF is guaranteed to have a constant contribution to every cell in the corresponding d-dimensional partition. This essentially means that we have discovered a d-dimensional uniform-count partition in A_S, and we can output the partition boundaries and the corresponding tuple count (Steps 2–6).

Figure 7.11(b) depicts the partitioning of a two-dimensional data array generated by `render` for the input set consisting of the four wavelet coefficients shown in Figure 7.11(a). The time complexity of our `render` algorithm can be shown to be $O(|W_S| \cdot P)$, where P is the number of uniform-count partitions in A_S. As we have already observed, P is typically much smaller than the number of array cells N. Also, note that `render` requires only $O(|W_S| \cdot d)$ of memory, since it only needs to keep track of the coefficients in the partition currently being processed for each dimension.

7.4 Experimental Study

In this section, we present the results of an extensive empirical study that we have conducted using the novel query processing tools developed in this chapter. The objective of this study is twofold: (1) to establish the effectiveness of our wavelet-based approach to approximate query processing, and (2) to demonstrate the benefits of our methodology compared to earlier approaches based on sampling and histograms. Our experiments consider a wide range of queries executed on both synthetic and real-life data sets. The major findings of our study can be summarized as follows.

- **Improved Answer Quality.** The quality/accuracy of the approximate answers obtained from our wavelet-based query processor is, in general, better than that obtained by either sampling or histograms for a wide range of data sets and `select`, `project`, `join`, and aggregate queries.

- **Low Synopsis Construction Costs.** Our I/O-efficient wavelet decomposition algorithm is extremely fast and scales linearly with the size of the data (i.e., the number of cells in the MOLAP array). In contrast, histogram construction costs increase explosively with the dimensionality of the data.

- **Fast Query Execution.** Query execution-time speedups of more than two orders of magnitude are made possible by our approximate query processing algorithms. Furthermore, our query execution times are competitive with those obtained by the histogram-based methods of Ioannidis and Poosala [70], and sometimes significantly faster (e.g., for `join`s).

Thus, our experimental results validate the thesis of this chapter that wavelets are a viable, effective tool for general-purpose approximate query processing in DSS environments. All experiments reported in this section were performed on a Sun Ultra-2/200 machine with 512 MB of main memory, running Solaris 2.5.

7.4.1 Experimental Testbed and Methodology

Techniques. We consider three approximate query answering techniques in our study.

• *Sampling.* A random sample of the non-zero cells in the multi-dimensional array representation for each base relation is selected , and the counts for the cells are appropriately scaled. Thus, if the total count of all cells in the array is t and the sum of the counts of cells in the sample is s, then the count of every cell in the sample is multiplied by $\frac{t}{s}$. These scaled counts give the tuple counts for the corresponding approximate relation.

• *Histograms.* Each base relation is approximated by a multi-dimensional MaxDiff(V,A) histogram. Our choice of this histogram class is motivated by the recent work of Ioannidis and Poosala [70], where it is shown that MaxDiff(V,A) histograms result in higher-quality approximate query answers compared to other histogram classes (e.g., EquiDepth or EquiWidth). We process `select`s, `join`s, and aggregate operators on histograms as described in [70]. For instance, while `select`s are applied directly to the histogram for a relation, a `join` between two relations is done by first partially expanding their histograms to generate the tuple-value distribution of the each relation. An indexed nested-loop `join` is then performed on the resulting tuples.

• *Wavelets.* Wavelet-coefficient synopses are constructed on the base relations (using algorithm COMPUTE-WAVELET) and query processing is performed entirely in the wavelet-coefficient domain, as described in Section 7.3. In our `join` implementation, overlapping pairs of coefficients are determined using a simple nested-loop join. Furthermore, during the rendering step for non-aggregate queries, cells with negative counts are not included in the final answer to the query.

Since we assume d dimensions in the multi-dimensional array for a d-attribute relation, c random samples require $c * (d + 1)$ units of space; d units are needed to store the index of the cell and 1 unit is required to store the cell count. Storing c wavelet coefficients also requires the same amount of space, since we need d units to specify the position of the coefficient in the wavelet transform array and 1 unit to specify the value for the coefficient. (Note that the hyper-rectangle and sign information for a base coefficient can easily be derived from its location in the wavelet transform array.) On the other hand, each histogram bucket requires $3 * d + 1$ units of space; $2 * d$ units to specify the low and high boundaries for the bucket along each of the d dimensions, d units to specify the number of distinct values along each dimension, and 1 unit to specify the average frequency for the bucket [116]. Thus, for a given amount of space corresponding to c samples/wavelet coefficients, we store $b \approx \frac{c}{3}$ histogram buckets to ensure a fair comparison between the methods.

Queries. The workload used to evaluate the various approximation techniques consists of four main query types: (1) `SELECT` *Queries*: ranges are specified for (a subset of) the attributes in a relation and all tuples that satisfy the conjunctive range predicate are returned as part of the query result, (2) `SELECT-SUM` *Queries*: the total `sum` of a particular attribute's values is computed for all tuples that satisfy a conjunctive range predicate over (a subset of) the attributes, (3) `SELECT-JOIN` *Queries*: after performing selections on two input relations, an equi-join on a single join dimension is performed and the resulting tuples are output; and, (4) `SELECT-JOIN-SUM` *Queries*: the total `sum` of an attribute's values is computed over all the tuples resulting from a `SELECT-JOIN`.

For each of the above query types, we have conducted experiments with multiple different choices for (a) `select` ranges, and (b) `select`, `join`, and `sum` attributes. The results presented in the next section are indicative of the overall observed behavior of the schemes. Furthermore, the queries presented in this chapter are fairly representative of typical queries over our data sets.

Answer-Quality Metrics. In our experiments with aggregate queries (e.g., `SELECT-SUM` queries), we use the *absolute relative error* in the aggregate value as a measure of the accuracy of the approximate query answer. That is, if *actual_aggr* is the result of executing the aggregation query on the actual base relations, while *approx_aggr* is the result of running it on the corresponding synopses, then the accuracy of the approximate answer is given by $\frac{|actual\_aggr - approx\_aggr|}{actual\_aggr}$.

Deciding on an error metric for non-aggregate queries is slightly more involved. The problem here is that non-aggregate queries do not return a single value, but rather a set of tuples (with associated counts). Capturing the "distance" between such an answer and the actual query result requires that we take into account how these two (multi)sets of tuples differ in both (a) the tuple frequencies, and (b) the actual values in the tuples [70]. (Thus, simplistic solutions like "symmetric difference" are insufficient.) When deciding on an error metric for non-aggregate results, we considered both the *Match And Compare* (MAC) error of Ioannidis and Poosala [70] and the network-flow-based *Earth Mover's Distance* (EMD) error of Rubner et al. [123]. We eventually chose a variant of the EMD error metric, since it offers a number of advantages over MAC error (e.g., computational efficiency, natural handling of non-integral counts) and, furthermore, we found that MAC error can show unstable behavior under certain circumstances [67]. We briefly describe the MAC and EMD error metrics below and explain why we chose the EMD metric.

The EMD and MAC Set-Error Metrics One of the main observations of Ioannidis and Poosala [70] was that a correct error metric for capturing the distance between two set-valued query answers (i.e., multisets of tuples) should take into account how these two (multi)sets of tuples differ in both (a) the tuple frequencies, and (b) the actual values in the tuples. A naive option is to simply define the distance between two sets of elements S_1 and S_2 as $|(S_1 - S_2) \cup (S_2 - S_1)|$. However, as discussed in [70], this measure does not take into account the frequencies of occurrences of elements or their values. For example, by the above measure, the two sets $\{5\}$ and $\{5, 5, 5\}$ would be considered to be at a distance of 0 from each other, while the set $\{5\}$ would be at the same distance from both $\{5.1\}$ and $\{100\}$.

In [70], the authors define the notion of *Match And Compare* (MAC) distance to measure the error between two multisets S_1 and S_2. Let $dist(e_1, e_2)$ denote the distance between elements $e_1 \in S_1$ and $e_2 \in S_2$ (in this chapter, we use the euclidean distance between elements). The MAC error involves matching pairs of elements from S_1 and S_2 such that each element appears in at least one matching pair, and the sum of the distances between the matching pairs is minimum. The sum of the matching pair distances, each weighted by the maximum number of matches an element in the pair is involved in, yields the MAC error. Though the MAC error has a number of nice properties and takes both frequency and value of elements in the sets into account, in some cases, it may be unstable [67]. Also, the MAC error, as defined in [70], could become computationally expensive, since multiple copies of a cell need to be treated separately, thus making set sizes potentially large.

Due to the stability and computational problems of the MAC error, in our experiments, we use the *Earth Mover's Distance* EMD error instead, which we have found to solve the above-mentioned problems. The EMD error metric was proposed by Rubner et al. [123] for computing the dissimilarity between two distributions of points and was applied to computing distances between images in a database. The main idea is to formulate the distance between two (multi)sets as a bipartite network flow problem, where the objective function incorporates the distance in the values of matched elements and the flow captures the distribution of element counts. More formally, the EMD error involves solving the bipartite network flow problem which can be formalized as the following linear programming problem. Let S_1 and S_2 be two sets of elements and let c_i denote the count of element e_i. Without loss of generality, let the sum of the counts of elements in S_1 be greater than or equal to the sum of counts of elements in S_2. Consider an assignment of non-negative flows $f(e_i, e_j)$ such that the following sum is minimized:

$$\sum_{e_i \in S_1} \sum_{e_j \in S_2} f(e_i, e_j) * dist(e_i, e_j) \tag{7.2}$$

subject to the following constraints:

$$\sum_{e_i \in S_1} f(e_i, e_j) = c_j \tag{7.3}$$

$$\sum_{e_j \in S_2} f(e_i, e_j) \leq c_i \tag{7.4}$$

The EMD error, that we employ in this chapter[7] is as follows:

$$EMD(S_1, S_2) = \sum_{e_i \in S_1} \sum_{e_j \in S_2} f(e_i, e_j) * dist(e_i, e_j) * (\frac{\sum_{e_i \in S_1} c_i}{\sum_{e_j \in S_2} c_j})$$

Thus, intuitively, the flows $f(e_i, e_j)$ distribute the counts of elements in S_1 across elements in S_2 in a manner that the sum of the distances over the flows is minimum. Note that since S_2 has a smaller count than S_1, we require that the inflow into each element e_j of S_2 is equal to c_j (Constraint 7.3). Also, the outflow out of each element e_i in S_1 cannot exceed c_i (Constraint 7.4). Also, observe that since the count of S_1 could be much larger than that of S_2, we scale the sum in Equation 7.2 by the ratio of the sum of counts of S_1 and S_2. This ensures that counts for elements in S_1 that are not covered as part of the flows get accounted for in the EMD error computation.

Thus, the EMD naturally extends the notion of distance between single elements to distance between sets of elements. Also, the EMD has the nice property that if the counts of S_1 and S_2 are equal, then the EMD is a true metric. There are efficient algorithms available to compute the flows $f(e_i, e_j)$ such that constraints (7.2), (7.3) and (7.4) are satisfied. Another added benefit of the EMD error is that it is naturally applicable to the cases when elements in the sets have non-integral counts. Since in a number of cases, the number of tuples computed by the approximation techniques can be fractions, this is an advantage. Hence we chose EMD as the error metric for non-aggregate queries.

7.4.2 Experimental Results – Synthetic Data Sets

The synthetic data sets we use in our experiments are similar to those employed in the study of Vitter and Wang [144]. More specifically, our synthetic data generator works by populating randomly-selected rectangular regions of cells in the multi-dimensional array. The input parameters to the generator along with their description and default values are as illustrated in Table 7.2. The generator assigns non-zero counts to cells in r rectangular regions each of whose volume is randomly chosen between v_{min} and v_{max} (the volume of a region is the number of cells contained in it). The regions themselves are uniformly distributed in the multi-dimensional array. The sum of the counts for all the cells in the array is specified by the parameter t. Portion $t \cdot (1 - n_c)$ of the count is partitioned across the r regions using a Zipfian distribution with value z. Within each region, each cell is assigned a count using a Zipfian distribution with value between z_{min} and z_{max}, and based on the L_1 distance of the cell from the center of the region. Thus, the closer a cell is to the center of its region, the larger is its count value. Finally, we introduce noise into the data set by randomly choosing cells such that these noise cells constitute a fraction n_v of the total number of non-zero cells. The noise count $t \cdot n_c$ is then uniformly distributed across these noise cells.

Note that with the default parameter settings described in Table 7.2, there are a total of a million cells of

[7]Rubner et al. [123] define the EMD error as the ratio $\frac{\sum_{e_i \in S_1} \sum_{e_j \in S_2} f(e_i,e_j)*dist(e_i,e_j)}{\sum_{e_j \in S_2} c_j}$.

which about 25000 have non-zero counts. Thus, the density of the multi-dimensional array is approximately 2.5%. Further, in the default case, the approximate representations of the relations occupy only 5% of the space occupied by the original relation – this is because we retain 1250 samples/coefficients out of 25000 non zero cells which translates to a compression ratio of 20. The same is true for histograms. Finally, we set the default selectivity of range queries on the multi-dimensional array to be 4% – the `SELECT` query range along each dimension was set to (512,720).

Parameter	**Description**	**Default Value**
d	Number of dimensions	2
s	Size of each dimension (equal for all dimensions)	1024
r	Number of regions	10
v_{min}, v_{max}	Minimum and maximum volume of each region	2500, 2500
z	Skew across regions	0.5
z_{min}, z_{max}	Minimum and maximum skew within each region	1.0, 1.0
n_v, n_c	Noise volume and noise count	0.05, 0.05
t	Total count	1000000
c	Number of coefficients/samples retained	1250
b	Number of histogram buckets	420
sel	Selectivity in terms of volume	4%

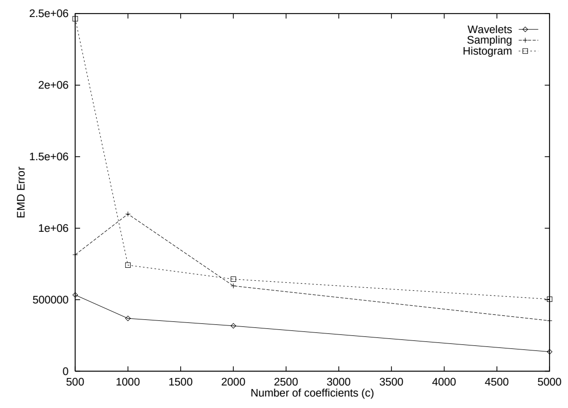
Table 7.2: Input Parameters to Synthetic Data Generator

Time to Compute the Wavelet Transform. In order to demonstrate the efficiency of our algorithm for computing the wavelet transform of a multi-dimensional array, in Table 7.3, we present the running times of COMPUTEWAVELET as the number of cells in the multi-dimensional array is increased from 250,000 to 16 million. The density of the multi-dimensional array is kept constant at 2.5% by appropriately scaling the number of cells with non-zero counts in the array. From the table, it follows that the computation time of our COMPUTEWAVELET algorithm scales linearly with the total number of cells in the array. We should note that the times depicted in Table 7.3 are actually dominated by CPU-computation costs – COMPUTEWAVELET required a single pass over the data in all cases.

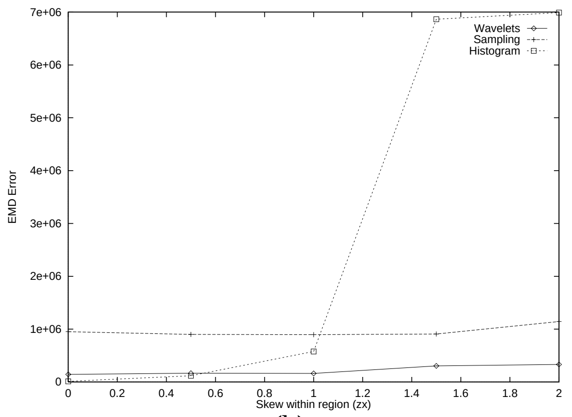
	Number of Cells in Multi-dimensional Array			
	250,000	1000,000	4000,000	16,000,000
Execution Time (in seconds)	6.3	26.3	109.9	445.4

Table 7.3: Wavelet Transform Computation Times

SELECT Queries. In our first set of experiments, we carry out a sensitivity analysis of the EMD error for `SELECT` queries to parameters like storage space, skew in cell counts within a region, cell density, and query selectivity. In each experiment, we vary the parameter of interest while the remaining parameters

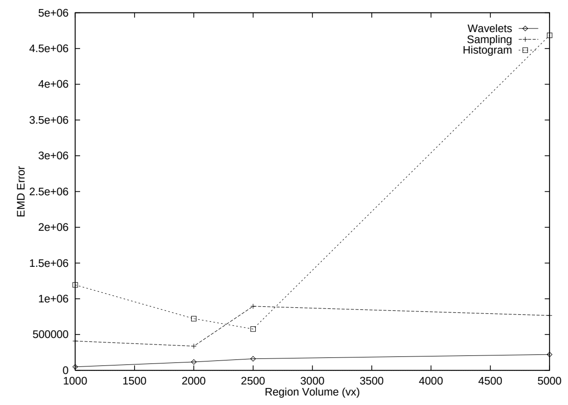


(a)

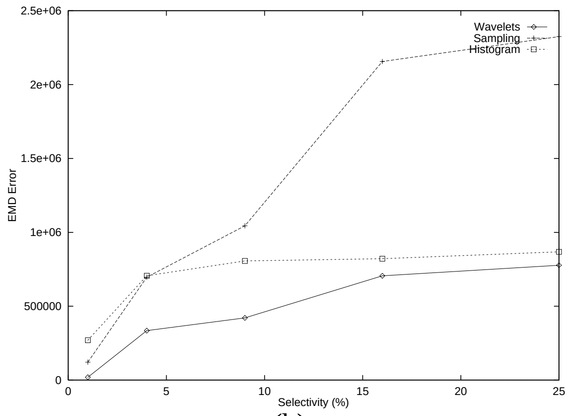


(b)

Figure 7.12: `SELECT` Queries: Sensitivity to (a) allocated space and (b) skew within regions.



(a)



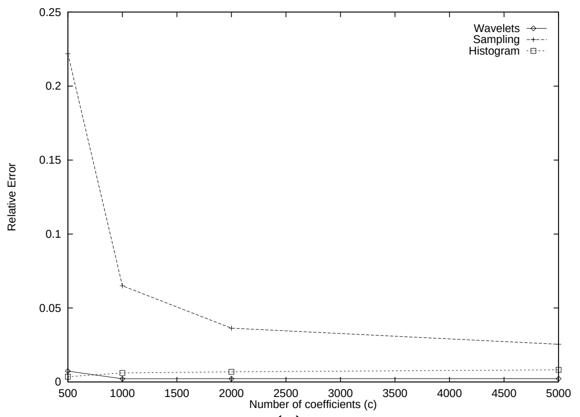
(b)

Figure 7.13: `SELECT` Queries: Sensitivity to (a) cell density and (b) query selectivity.

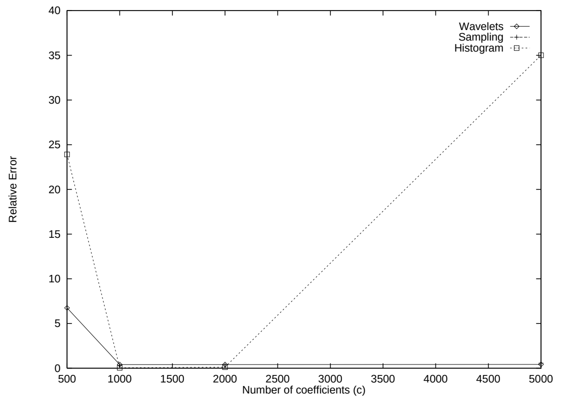
are fixed at their default values. Our results indicate that for a broad range of parameter settings, wavelets outperform both sampling and histograms – in some cases, by more than an order of magnitude.

• *Storage Space*. Figure 7.12(a) depicts the behavior of the EMD error for the three approximation methods as the space (i.e., number of retained coefficients) allocated to each is increased from 2% to 20% of the relation. For a given value of the number of wavelet coefficients c along the x-axis, histograms are allocated space for $\approx \frac{c}{3}$ buckets. As expected, the EMD error for all the cases reduces as the amount of space is increased. Note that for 500 coefficients, the EMD error for histograms is almost five times worse that the corresponding error for wavelets. This is because the few histogram buckets are unable to accurately capture the skew within each region (in our default parameter settings, the Zipfian parameter for the skew within a region is 1).

• *Skew Within Regions*. In Figure 7.12(b), we plot the EMD error as the Zipfian parameter z_{max} that controls the maximum skew within each region is increased from 0 to 2.0. Histograms perform the best for values



(a)



(b)

Figure 7.14: Effect of allocated space on (a) `SELECT-SUM`, and (b) `SELECT-JOIN-SUM` queries.

of z_{max} between 0 and 0.5 when the cell counts within each region are more or less uniformly distributed. However, once the maximum skew increases beyond 0.5, the histogram buckets can no longer capture the data distribution in each region accurately. As a consequence, we observe a spike in the EMD error for region skew corresponding to a value of $z_{max} = 1.5$. Incidentally, a similar behavior for MaxDiff histograms has been reported earlier in [70].

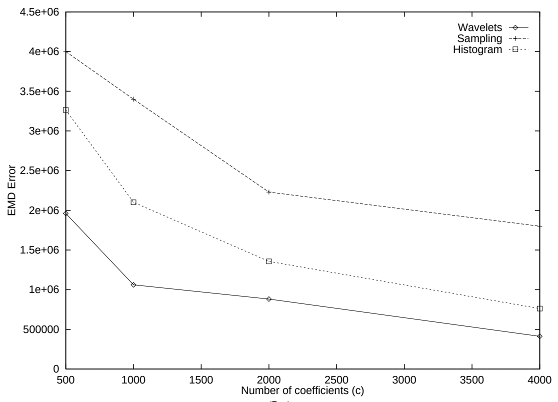
• *Cell Density*. In Figure 7.13(a), we plot the graphs for EMD error as v_{max}, the maximum volume of regions is varied between 1000 (1% density) and 5000 (5% density) (v_{min} is fixed at 1000). As the number of non-zero cells in the multi-dimensional array increases, the number of coefficients, samples and histogram buckets needed to approximate the underlying data also increases. As a consequence, in general, the EMD error is more when regions have larger volumes. Note the sudden jump in the EMD error for histograms when the volume becomes 5000. This is because the histogram buckets overestimate the total of the cell counts in the query region by almost 50%. In contrast, the error in the sum of the cell counts within the query range with wavelets is less than 0.1%.

• *Selectivity of Query*. Figure 7.13(b) illustrates the EMD errors for the techniques as the selectivity of range queries is increased from 2% to 25%. Since the number of tuples in both the accurate as well as the approximate answer increase, the EMD error increases as the selectivity of the query is increased (recall that the EMD error is the sum of the pairwise distances between elements in the two sets of answers weighted by the flows between them).

SELECT-SUM Queries. Figure 7.14(a) depicts the performance of the various techniques for `SELECT-SUM` queries as the allocated space is increased from 2% to 20% of the relation. Both wavelets and histograms exhibit excellent performance compared to random sampling; the relative errors are extremely low for both techniques – 0.2% and 0.6%, respectively. These results are quite different from the EMD error curves for the three schemes (see Figure 7.12(a)). We can thus conclude that although histograms are excel-

Technique	Number of Coefficients			
	500	1000	2000	5000
Wavelets	0.01	0.02	0.04	0.08
Histograms	9.8	1.48	0.43	1.26

(a)



(b)

Figure 7.15: (a) `SELECT-JOIN-SUM` query execution times. (b) `SELECT` query errors on real-life data.

lent at approximating aggregate frequencies, they are not as good as wavelets at capturing the distribution of values accurately. In [144], wavelets were shown to be superior to sampling for aggregation queries – however, the work in [144] did not consider histograms.

SELECT-JOIN and SELECT-JOIN-SUM Queries. For join queries, in Figure 7.14(b), we do not show the errors for sampling since in almost all cases, the final result contained zero tuples. Also, we only plot the relative error results for `SELECT-JOIN-SUM` queries, since the EMD error graphs for `SELECT-JOIN` queries were similar.

When the number of coefficients retained is 500, the relative error with wavelets is more than four times better than the error for histograms – this is because the few histogram buckets are not as accurate as wavelets in approximating the underlying data distribution. For histograms, the relative error decreases for 1000 and 2000 coefficients, but shows an abrupt increase when the number of coefficients is 5000. This is because at 5000 coefficients, when we visualized the histogram buckets, we found that a large bucket appeared in the query region (that was previously absent), in order to capture the underlying noise in the data set. Cells in this bucket contributed to the dramatic increase in the join result size, and subsequently, the relative error.

We must point out that although the performance of histograms is erratic for the query region in Figure 7.14(b), we have found histogram errors to be more stable on other query regions. Even for such regions, however, the errors observed for histograms were, in most cases, more than an order of magnitude worse than those for wavelets. Note that the relative error for wavelets is extremely low (less than 1%) even when the coefficients take up space that is about 4% of the relation.

Query Execution Times. In order to compare the query processing times for the various approaches, we measured the time (in seconds) for executing a `SELECT-JOIN-SUM` query using each approach. We do not consider the time for random sampling since the join results with samples did not generate any tuples, except for very large sample sizes. The running time of the join query on the original base relations (using an indexed nested-loop join) to produce an exact answer was 3.6 seconds. In practice, we expect that this time will be much higher since in our case, the entire relations fit in main memory. As is evident from Figure 7.15(a), our wavelet-based technique is more than two orders of magnitude faster compared to running the queries on the entire base relations.

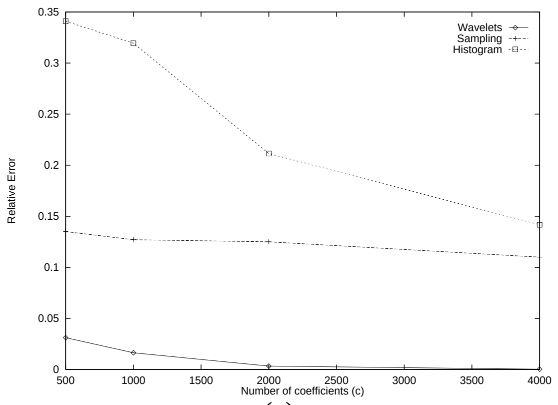
Also, note that the performance of histograms is much worse than that of wavelets. The explanation lies in the fact that the `join` processing algorithm of Ioannidis and Poosala [70] requires joining histograms to be partially expanded to generate the tuple-value distribution for the corresponding approximate relations. The problem with this approach is that the intermediate relations can become fairly large and may even contain more tuples than the original relations. For example, with 500 coefficients, the expanded histogram contains almost 5 times as many tuples as the base relations. The sizes of the approximate relations decrease as the number of buckets increase, and thus execution times for histograms drop for larger numbers of buckets. In contrast, in our wavelet approach, join processing is carried out exclusively in the compressed domain, that is, joins are performed directly on the wavelet coefficients without ever materializing intermediate relations. The tuples in the final query answer are generated at the very end as part of the rendering step and this is the primary reason for the superior performance of the wavelet approach.

7.4.3 Experimental Results – Real-life Data Sets

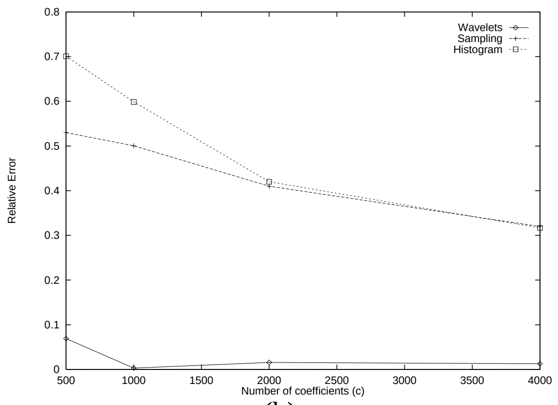
We obtained our real-life data set from the US Census Bureau (`www.census.gov`). We employed the Current Population Survey (CPS) data source and within it the Person Data Files of the March Questionnaire Supplement. We used the 1992 data file for the select and select sum queries, and the 1992 and 1994 data files for the join and join sum queries. For both files, we projected the data on the following four attributes whose domain values were previously coded: *age* (with value domain 0 to 17), *educational attainment* (with value domain 0 to 46), *income* (with value domain 0 to 41) and *hours per week* (with value domain 0 to 13). Along with each tuple in the projection, we stored a count which is the number of times it appears in the file. We rounded the maximum domain values off to the nearest power of 2 resulting in domain sizes of 32, 64, 64 and 16 for the four dimensions, and a total of 2 million cells in the array. The 1992 and the 1994 collections had 16271 and 16024 cells with non-zero counts, respectively, resulting in a density of $\approx$ 0.001. However, even though the density is very low, we did observe large dense regions within the arrays when we visualized the data – these dense regions spanned the entire domains of the *age* and *income* dimensions.

For all the queries, we used the following select range: $5 \leq age < 10$ and $10 \leq income < 15$ that we found to be representative of several select ranges that we considered (the remaining two dimensions were left unspecified). The selectivity of the query was 1056/16271$=$ 6%. For `sum` queries, the `sum` operation was performed on the *age* dimension. For `join` queries, the `join` was performed on the *age* dimension between the 1992 and 1994 data files.

SELECT Queries. In figures 7.15(b) and 7.16(a), we plot the EMD error and relative error for `SELECT` and `SELECT-SUM` queries, respectively, as the space allocated for the approximations is increased from



(a)



(b)

Figure 7.16: (a) `SELECT-SUM` and (b) `SELECT-JOIN-SUM` queries on real-life data.

3% to 25% of the relation. From the graphs, it follows that wavelets result in the least value for the EMD error, while sampling has the highest EMD error. For `SELECT-SUM` queries, wavelets exhibit more than an order of magnitude improvement in relative error compared to both histograms and sampling (the relative error for wavelets is between 0.5% and 3%). Thus, the results for the select queries indicate that wavelets are effective at accurately capturing both the value as well as the frequency distribution of the underlying real-life data set.

Note that unlike the EMD error and the synthetic data cases, the relative error for sampling is better than for histograms. We conjecture that one of the reasons for this is the higher dimensionality of the real-life data sets, where histograms are less effective.

JOIN Queries. We only plot the results of the `SELECT-JOIN-SUM` queries in Figure 7.16(b), since the EMD error graphs for `SELECT-JOIN` queries were similar. Over the entire range of coefficients, wavelets outperform sampling and histograms, in most cases by more than an order of magnitude. With the real-life data set, even after the `join`, the relative aggregate error using wavelets is very low and ranges between 1% to 6%. The relative error of all the techniques improve as the amount of allocated space is increased. Note that compared to the synthetic data sets, where the result of a `join` over samples contained zero tuples in most cases, for the real-life data sets, sampling performs quite well. This is because the size of the domain of the *age* attribute on which the join is performed is only 18, which is quite small. Consequently, the result of the `join` query over the samples is no longer empty.

In summary, our wavelet-based approach consistently outperforms the sampling and histograms approaches. Sampling suffers mainly for non-aggregate queries as it always produces a small subset of the exact answer. This problem is extreme when `join`s are involved as the results often contain zero tuples. Histograms perform poorly for non-uniform and high dimensional datasets as such data distributions cannot be accurately captured with a small number of rectangular regions containing uniformly distributed points. The wavelet approach do not suffer from the above problems. As mentioned before, wavelets are effective as long as the data distribution exhibits the locality property i.e. tuples corresponding to neighboring cells in the multidimensional representation have similar counts. They may not work well for “spiky” data distributions. Our experience shows that most datasets in real-life DSS applications do exhibit locality; hence the wavelet-based approach proposed in this chapter is an effective approximate query answering solution for such applications.

7.5 Conclusions

Approximate query processing is slowly emerging as an essential tool for numerous data-intensive applications requiring interactive response times. Most work in this area, however, has so far been limited in its scope and conventional approaches based on sampling or histograms appear to be inherently limited when it comes to complex approximate queries over high-dimensional data sets. In this chapter, we have proposed the use of multi-dimensional wavelets as an effective tool for general-purpose approximate query processing in modern, high-dimensional applications. Our approach is based on building wavelet-coefficient synopses of the data and using these synopses to provide approximate answers to queries. We have developed novel query processing algorithms that operate directly on the wavelet-coefficient synopses of relational data, allowing us to process arbitrarily complex queries *entirely* in the wavelet-coefficient domain. This guarantees extremely fast response times since our approximate query execution engine can do the bulk of its processing over compact sets of wavelet coefficients, essentially postponing the expansion into relational tuples until the end-result of the query. We have also proposed a novel I/O-efficient wavelet decomposition algorithm for building the synopses of relational data. Finally, we have conducted an extensive experimental study with synthetic as well as real-life data sets to determine the effectiveness of our wavelet-based approach compared to sampling and histograms. Our results demonstrate that our wavelet-based query processor (a) provides approximate answers of better quality than either sampling or histograms, (b) offers query execution-time speedups of more than two orders of magnitude, and (c) guarantee fast synopsis construction times that scale linearly to the size of the relation.

Chapter 8

Conclusion and Future Work

We conclude this dissertation with a summary of our contributions and directions for future work.

8.1 Summary

In this thesis, we identified some of the main challenges in managing large, complex multidimensional datasets inside a database system:

- **High Dimensional Index Structures**: High dimensional similarity search is common in many modern database applications like multimedia retrieval (e.g., 64-d color histograms), data mining/OLAP (e.g., 52-d bank data in clustering) and time series/scientific/medical applications (e.g., 20-d Space Shuttle data, 100-d astronomy data in SDSS, 64-dimensional ECG data). Sequential scanning and 1-dimensional index structures are not effective solutions; we need multidimensional index structures. Existing multidimensional index structures do not scale beyond 10-15 dimensions. We need multidimensional index structures that would scale to high dimensionalities (50-100 dimensions).

- **Dimensionality Reduction Techniques**: While a scalable index structure would be a big step towards enabling DBMSs to efficiently support queries over high dimensional data, we can achieve further scalability by first reducing the dimensionality of data and then building the index on the reduced data. Existing dimensionality reduction techniques work well only when the data set is globally correlated. In practice, datasets are often not globally correlated. We need dimensionality reduction techniques that would work well even when the data is not globally correlated.

- **Time Series Indexing Techniques**: Similarity search in time series databases is a difficult problem due to the typically high dimensionality of the raw data. The most promising solution involves performing dimensionality reduction on the data, then indexing the reduced data with a multidimensional index structure. Existing dimensionality reduction techniques choose a common representation for all the items in the database; this causes loss of fidelity of the reduced-representation to the original signal which in turn degrades the search performance. We need a dimensionality reduction technique where the reduced representation always closely approximates the original signal. The representation has to be indexable using a multidimensional index structure.

- **Integration of Multidimensional Index Structures to DBMSs**: One of the most important practical challenges in multidimensional data management is that of integration of multidimensional index structures as access methods in a DBMS. The Generalized Search Tree (GiST) provides an elegant solution to the above problem. However, before it can be supported in a “commercial strength” DBMS, efficient techniques to support transactional access to data via the GiST must be developed.

- **Approximate Query Answering for Decision Support Applications**: Approximate query answering has emerged as a viable approach for dealing with the huge data volumes and stringent response time requirements in decision support/OLAP systems. The general approach is to first construct compact synopses of interesting relations in the database and then answering the queries by using just the synopses (which usually fit in memory). Approximate query answering techniques proposed so far either suffer from high error rates or are severely limited in their query processing scope. We need to develop approximate query answering techniques that are accurate, efficient and general in their query processing scope.

This dissertation addresses the above challenges as follows:

- **Index Structure for High Dimensional Spaces:** We have designed an index structure, namely the *hybrid tree*, that scales to high dimensional feature spaces. The key idea is to combine the positive aspects of the two types of index structures, namely data partitioning and space partitioning index structures, into a single data structure to achieve scalable search performance. The details of the hybrid tree can be found in Chapter 3.

- **Local Dimensionality Reduction for High Dimensional Indexing**: We have developed the local dimensionality reduction (LDR) technique which reduces the dimensionality of data with significantly lower loss of information compared to global dimensionality reduction. The main idea here is to exploit local, as opposed to global, correlations in the data for dimensionality reduction. The details of LDR can be found in Chapter 4.

- **Locally Adaptive Dimensionality Reduction for Time Series Data**: We have introduced a new dimensionality reduction technique for time series called Adaptive Piecewise Constant Approximation (APCA). APCA adapts locally to each data item and thereby achieves high fidelity to the original signal. The details can be found in Chapter 5.

- **Concurrency Control in Generalized Search Trees**: In order to facilitate integration of multidimensional index structures as access methods in DBMSs, we have developed techniques to provide transactional access to data via multidimensional index structures. The details can be found in Chapter 6.

- **Wavelet-based Approximate Query Processing Tool**: We have developed a wavelet-based approximate query answering tool for high-dimensional DSS applications. We showed how we can process any SQL query entirely in the wavelet domain, thereby guaranteeing extremely fast response times. The details can be found in Chapter 7.

8.2 Software

The softwares developed during this dissertation include:

- **Hybrid Tree**: We have implemented the hybrid tree as described in Chapter 3 (about 6000 lines of C++ code) and distributed the software via our web site `http://www-db.ics.uci.edu/pages/software`. The software is being used by at least 3 companies and 8 universities for research and teaching purposes.

- **LDR**: We have also implemented LDR as described in Chapter 4 (about 2100 lines of C++ code) and distributed it via our web site `http://www-db.ics.uci.edu/pages/software`. The software is being used by at least 4 universities for research purposes.

- **Wavelet-based Approximate Query Answering Tool**: We implemented a wavelet-based approximate query answering engine as described in Chapter 7 (about 3200 lines of C++ code).

- **Integration of developed techniques to MARS**: We have integrated the hybrid tree into the MARS database system. An application in MARS can create a hybrid tree index of desired dimensionality on one or more attributes of a relation. The hybrid tree nodes can be striped across multiple disks (for parallel I/O). Similarity queries on a relation can then be answered by running an appropriate range query or k-NN query on the index. Besides the hybrid tree, the MARS index manager supports B+-tree, R-tree and text indices.

8.3 Future Directions

There are several interesting directions of future work based on the work described in this thesis. Some of these are extensions of our work, while some others are motivated by the general problems of managing multidimensional data.

- *Indexing and Mining Sequence Data*: In this thesis, we proposed dimensionality reduction and indexing techniques for time series data. Some of these techniques can be applied to sequence data as well, both one-dimensional and multidimensional sequences. Examples of one-dimensional sequences include gene/protein sequences and clickstream data generated by web sites; examples of multidimensional sequences include 2-dimensional shapes[1] . Developing new search and mining techniques for such types of data based on adaptive representations is an interesting direction of research.

- *High Dimensional Data Mining*: In Chapter 4, we introduced local dimensionality reduction (LDR) as a technique for high dimensional indexing. LDR has applications beyond indexing; it can be used to discover patterns in high dimensional data. Current efforts based on LDR include using LDR for selectivity estimation in high dimensional datasets [57] and text data mining [17].

- *Data Visualization and Visual Data Mining*: Data Visualization has been proven to be of high value for exploratory data analysis and database mining. The idea is to present the data in some visual form, allowing a human to get insight into the data, draw conclusions and interact directly with the data. Techniques proposed in this thesis can be applied to improve current data visualization techniques. For example, LDR can be used to handle the high dimensionality of the data to be visualized. Approximate query answering techniques like the wavelet-based technique proposed in Chapter 7 can be used to achieve tradeoff between the resolution of the display and speed.

- *Approximate Query Answering with Error Guarantees*: In Chapter 7, we developed an approximate query answering tool for OLAP data. Although our technique offers high accuracy and low response times, we do not provide guaranteed error bounds. Recent work suggests a trend toward approximate answering with error bounds [84, 106].

- *Location-dependent Querying from wireless/mobile devices*: One of most common types of multi-dimensional data is spatial or geographic data. With the advent of global positioning system (GPS) technology, all users/devices in the future will have knowledge of their locations. This information can be used to query location-sensitive information and/or obtain location-dependent service. Developing an infrastructure for such applications is an active area of research, with the potential of significant commercial impact [94, 143].

[1]We proposed a locally adaptive representation for 2-d shapes in [26].

References

[1] Swarup Acharya, Phillip B. Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. “Join Synopses for Approximate Query Answering”. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, pages 275–286, Philadelphia, Pennsylvania, May 1999.

[2] R. Agarwal, J. Gehrke, D. Gunopolos, and P. Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. *Proc. of SIGMOD*, 1998.

[3] C. Aggarwal, C. Procopiuc, J. Wolf, P. Yu, and J. Park. Fast algorithms for projected clustering. *Proc. of SIGMOD*, 1999.

[4] R. Agrawal, M. Carey, and M. Livny. Models for studying concurrency control performance: Alternatives and implications. In *SIGMOD*, May 1985.

[5] R. Agrawal, C. Faloutsos, and A. Swami. Efficient similarity search in sequence databases. In *Proceedings of FODO Conference*, October 1993.

[6] Laurent Amsaleg, Philippe Bonnet, Michael J. Franklin, Anthony Tomasic, and Tolga Urhan. “Improving Responsiveness for Wide-Area Data Access”. *IEEE Data Engineering Bulletin*, 20(3):3–11, September 1997. (Special Issue on Improving Query Responsiveness).

[7] ANSI. Ansi x3.135-1992, american national standard for information systems - database language - sql. November, 1992.

[8] D. Barbara, W. DuMouchel, C. Faloutsos, P. Haas, J. Hellerstein, Y. Ionnidis, H. Jagadish, T. Johnson, R. Ng, V. Poosala, K. Ross, and K. Sevcik. The new jersey data reduction report. *Data Engineering, 20(4)*, 1997.

[9] D. Barbarà, W. DuMouchel, C. Faloutsos, P.J. Haas, J.M. Hellerstein, Y. Ioannidis, H.V. Jagadish, T. Johnson, R. Ng, V. Poosala, K.A. Ross, and K.C. Sevcik. “The New Jersey Data Reduction Report”. *IEEE Data Engineering Bulletin*, 20(4):3–45, December 1997. (Special Issue on Data Reduction Techniques).

[10] S. Bay. The uci kdd archive. *http://kdd.ics.uci.edu*, 2000.

[11] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of ACM SIGMOD*, May 1990.

[12] S. Berchtold, C. Bohm, D. Keim, and H. P. Kriegel. A cost model for nearest neighbor search in high dimensional data spaces. *PODS*, 1997.

[13] S. Berchtold, C. Bohm, and H. P Kriegel. The pyramid technique: Towards breaking the curse of dimensionality. *Proc. of ACM SIGMOD*, 1998.

[14] S. Berchtold and D. A. Keim. Indexing high-dimensional spaces: Database support for next decade's application. *SIGMOD Tutorial*, 1998.

[15] S. Berchtold, D. A. Keim, and H. P Kriegel. The x-tree: An index structure for high-dimensional data. *Proc. of VLDB*, 1996.

[16] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is "nearest neighbor" meaningful? *Proc. of ICDT*, 1998.

[17] C. Blake. Text mining. *Information and Computer Science Technical Report, University of California*, 2001.

[18] R. Bliuhute, S. Saltenis, G. Slivinskas, and C. Jensen. Developing a datablade for a new index. *Proc. of ICDE*, 1999.

[19] T. Bozkaya and M. Ozsoyoglu. Distance-based indexing for high dimensional metric spaces. *Proc. of SIGMOD*, 1997.

[20] US Census Bureau. Current population survey, person data files. In *www.census.gov*, 1992.

[21] K. Chakrabarti, K.Porkaew, M. Ortega, and S. Mehrotra. Evaluating refined queries in top-k retrieval systems. *Submitted for publication. Available as Technical Report TR-MARS-00-04, University of California at Irvine, online at http://www-db.ics.uci.edu/pages/publications/*, July 2000.

[22] K. Chakrabarti and S. Mehrotra. The hybrid tree: An index structure for indexing high dimensional feature spaces. *Extended Version, Technical Report, MARS-TR-99-01, Available from http://luke.ics.uci.edu:8000/pages/publications*, 1998.

[23] K. Chakrabarti and S. Mehrotra. The hybrid tree: An index structure for high dimensional feature spaces. *Proceedings of the IEEE International Conference on Data Engineering*, March 1999.

[24] K. Chakrabarti and S. Mehrotra. Local dimensionality reduction: A new approach to indexing high dimensional spaces. *Proceedings of VLDB Conference*, 2000.

[25] K. Chakrabarti and S. Mehrotra. Local dimensionality reduction: A new approach to indexing high dimensional spaces. *Technical Report, TR-MARS-00-04, University of California at Irvine, http://www-db.ics.uci.edu/pages/publications/,* 2000.

[26] K. Chakrabarti, M. Ortega-Binderberger, K. Porkaew, and S. Mehrotra. Similar shape retrieval in mars. *Proceedings of ICME (IEEE International Conference on Multimedia and Expo),* 2000.

[27] Kaushik Chakrabarti and Sharad Mehrotra. Concurrency control in multidimensional access methods. *Technical Report TR-MARS-97-12, Department of Computer Science, University of Illinois,* October 1997.

[28] Kaushik Chakrabarti and Sharad Mehrotra. Dynamic granular locking approach to phantom protection in r-trees. *Proc. of the IEEE International Conference on Data Engineering*, February 1998.

[29] K. Chan and W. Fu. Efficient time series matching by wavelets. *Proceedings of IEEE International Conference on Data Engineering,* 1999.

[30] S. Chandrasekaran, B. Manjunath, Y. Wang, J. Winkler, and H. Zhang. An eigenspace update algorithm for image analysis. *Graphical Models and Image Processing, Vol. 59. No. 5,* 1997.

[31] T. Chiueh. Content-based image indexing. *Proc. of VLDB*, 1994.

[32] K. Chu and M. Wong. Fast time-series searching with scaling and shifting. *Proceedings of PODS*, 1999.

[33] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. *Proc. of VLDB*, 1997.

[34] William G. Cochran. *"Sampling Techniques"*. John Wiley & Sons, 1977. (Third Edition).

[35] G. Das, K. Lin, H. Mannila, G. Renganathan, and P. Smyth. Rule discovery from time series. *Proceedings of KDD Conference*, 1998.

[36] A. Debregeas and G. Hebrail. Interactive interpretation of kohonen maps applied to curves. *Proceedings of KDD Conference*, 1998.

[37] Prasad M. Deshpande, Karthikeyan Ramasamy, Amit Shukla, and Jeffrey F. Naughton. "Caching Multidimensional Queries Using Chunks". In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, pages 259–270, Seattle, Washington, June 1998.

[38] R. Duda and P. Hart. Pattern classification and scene analysis. *Wiley, New York*, 1973.

[39] M. Ester, J. Kohlhammer, and H. Kriegel. The dc-tree: A fully dynamic index structure for data warehouses. *Proc. of ICDE*, 2000.

[40] Martin Ester, Jorn Kohlhammer, and Hans-Peter Kriegel. “The DC-Tree: A Fully Dynamic Index Structure for Data Warehouses”. In *Proceedings of the Sixteenth International Conference on Data Engineering*, San Diego, USA, 2000.

[41] R. Fagin. Fuzzy queries in multimedia database systems. *Proceedings of PODS*, 1998.

[42] Ronald Fagin. Combining fuzzy information from multiple systems. *Proc. of the 15th ACM Symp. on PODS*, 1996.

[43] C. Faloutsos, W. Equitz, M. Flickner, W. Niblack, D. Petkovic, and R. Barber. Efficient and effective querying by image content. In *Journal of Intelligent Information Systems, Vol. 3, No. 3/4*, pages 231–262, July 1994.

[44] C. Faloutsos and et. al. Efficient and effective querying by image content. In *Journal of Intell. Inf. Systems*, July 1994.

[45] C. Faloutsos, H. Jagadish, A. Mendelzon, and T. Milo. A signature technique for similarity-based queries. In *SEQUENCES*, 1997.

[46] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *Proc. of SIGMOD*, May 1994.

[47] Christos Faloutsos and King-Ip (David) Lin. Fastmap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In *Proc. ACM SIGMOD*, pages 163–174, May 1995.

[48] Keinosuke Fukunaga. *Introduction to Statistical Pattern Recognition*. Academic Press, second edition edition, 1990.

[49] V. Ganti, R. Ramakrishnan, J. Gehrke, A. Powell, and J. French. Clustering large datasets in arbitrary metric spaces. *Proc. of ICDE*, 1999.

[50] Y. Garcia, M. Lopez, and S. Leutenegger. On optimal node splitting for r-trees. In *Proc. of VLDB*, 1998.

[51] Philip B. Gibbons and Yossi Matias. “New Sampling-Based Summary Statistics for Improving Approximate Query Answers”. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, pages 331–342, Seattle, Washington, June 1998.

[52] Philip B. Gibbons, Yossi Matias, and Viswanath Poosala. “Fast Incremental Maintenance of Approximate Histograms”. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, Athens, Greece, August 1997.

[53] Phillip B. Gibbons, Yossi Matias, and Viswanath Poosala. “Aqua Project White Paper”. Unpublished Manuscript (Bell Laboratories), December 1997.

[54] T. Gonzalez. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science*, 1985.

[55] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, CA, 1993.

[56] D. Greene. An implementation and performance analysis of spatial data access methods. In *Proceedings of ICDE, pages 606-615*, 1989.

[57] J. Guerin. Selectivity estimation for high dimensional data using ldr. *Information and Computer Science Technical Report, University of California*, 2001.

[58] S. Guha, R. Rastogi, and K. Shim. Cure: An efficient clustering algorithm for large databases. *Proc. of SIGMOD*, 1998.

[59] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. ACM SIGMOD Conf., pp. 47–57.*, 1984.

[60] Peter J. Haas. “Large-Sample and Deterministic Confidence Intervals for Online Aggregation”. In *Proceedings of the Ninth International Conference on Scientific and Statistical Database Management*, Olympia, Washington, August 1997.

[61] Peter J. Haas and Joseph M. Hellerstein. “Ripple Joins for Online Aggregation”. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, pages 287–298, Philadelphia, Pennsylvania, May 1999.

[62] J. Hellerstein, E. Koutsoupias, and C. Papadimitriou. Towards a theory of indexability. In *Proceeding of PODS, 1997*, June 1997.

[63] J. Hellerstein, J. Naughton, and A. Pfeffer. Generalized search trees in database systems. In *Proceeding of VLDB, pages 562-573,* September 1995.

[64] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. “Online Aggregation”. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, Tucson, Arizona, May 1997.

[65] A. Henrich. The lsdh-tree: An access structure for feature vectors. *Proceedings of ICDE*, 1998.

[66] G. R. Hjaltason and H. Samet. Ranking in spatial databases. *Proceedings of SSD*, 1995.

[67] Yannis E. Ioannidis. Personal Communication, August 1999.

[68] Yannis E. Ioannidis and Stavros Christodoulakis. “On the Propagation of Errors in the Size of Join Results”. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, pages 268–277, Denver, Colorado, May 1991.

[69] Yannis E. Ioannidis and Viswanath Poosala. “Balancing Histogram Optimality and Practicality for Query Result Size Estimation”. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 233–244, May 1995.

[70] Yannis E. Ioannidis and Viswanath Poosala. “Histogram-Based Approximation of Set-Valued Query Answers”. In *Proceedings of the 25th International Conference on Very Large Data Bases*, Edinburgh, Scotland, September 1999.

[71] Y. Ishikawa, R. Subramanya, and C. Faloutsos. Mindreader: Querying databases through multiple examples. *Proc. of VLDB*, 1998.

[72] H. V. Jagadish. Linear clustering of objects with multiple attributes. In *Proceedings of ACM SIGMOD, pages 332-342*, May 1990.

[73] H. V. Jagadish. “Linear Clustering of Objects with Multiple Attributes”. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 332–342, Atlantic City, New Jersey, May 1990.

[74] Björn Jawerth and Wim Sweldens. “An Overview of Wavelet Based Multiresolution Analyses”. *SIAM Review*, 36(3):377–412, 1994.

[75] T. Kahveci and A. Singh. Variable length queries for time series data. *Proceedings of ICDE*, 2001.

[76] K. V. Ravi Kanth, D. Agrawal, and A. K. Singh. Dimensionality reduction for similarity searching dynamic databases. *Proc. of SIGMOD*, 1998.

[77] N. Katayama and S. Satoh. The sr-tree: An index structure for high dimensional nearest neighbor queries. *Proc. of SIGMOD*, 1997.

[78] E. Keogh, K. Chakrabarti, S. Mehrotra, and M. Pazzani. Locally adaptive dimensionality reduction for indexing large time series databases. In *Proceedings of 2001 ACM SIGMOD Conference*, 2001.

[79] E. Keogh, K. Chakrabarti, M. Pazzani, and S. Mehrotra. Dimensionality reduction for fast similarity search in large time series databases. *Knowledge and Information Systems Journal*, 2000.

[80] E. Keogh and M. Pazzani. An enhanced representation of time series which allows fast and accurate classification, clustering and relevance feedback. *Proc. of KDD Conference*, 1998.

[81] F. Korn, H. Jagadish, and C. Faloutsos. Efficiently supporting ad hoc queries in large datasets of time sequences. *Proc. of SIGMOD*, 1997.

[82] F. Korn, N. Sidiropoulos, and C. Faloutsos. Fast nearest neighbor search in medical image databases. *Proc. of VLDB*, 1996.

[83] M. Kornacker, C. Mohan, and J. Hellerstein. Concurrency and recovery in generalized search trees. In *Proc. of SIGMOD*, 1997.

[84] I. Lazaridis and S. Mehrotra. Progressive approximate aggregate queries with multi-resolution tree structure. *Proc. of ACM SIGMOD Conference*, 2001.

[85] Ju-Hong Lee, Deok-Hwan Kim, and Chin-Wan Chung. “Multi-dimensional Selectivity Estimation Using Compressed Histogram Information”. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, pages 205–214, Philadelphia, Pennsylvania, May 1999.

[86] K. Lin, H. V. Jagadish, and C. Faloutsos. The TV-tree - an index stucture for high dimensional data. In *VLDB Journal*, 1994.

[87] Richard J. Lipton, Jeffrey F. Naughton, and Donovan A. Schneider. “Practical Selectivity Estimation through Adaptive Sampling”. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 1–12, Atlantic City, New Jersey, May 1990.

[88] D. Lomet. A review of recent work on multi-attribute access methods. In *SIGMOD Record*, Sept. 1992.

[89] D. Lomet. Key range locking strategies for improved concurrency. In *VLDB Proceedings*, August 1993.

[90] D. Lomet and B. Salzberg. The hb-tree: A multiattribute indexing mechanism with good guaraneed performance. *ACM Transactions on Database Systems*, 15(4), 1990.

[91] Yossi Matias, Jeffrey Scott Vitter, and Min Wang. “Wavelet-Based Histograms for Selectivity Estimation”. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, pages 448–459, Seattle, Washington, June 1998.

[92] Yossi Matias, Jeffrey Scott Vitter, and Min Wang. “Dynamic Maintenance of Wavelet-Based Histograms”. In *Proceedings of the 26th International Conference on Very Large Data Bases*, Cairo, Egypt, September 2000.

[93] J. Melton and A. R. Simon. Understanding the new sql: A complete guide. *Morgan Kauffman*, 1993.

[94] MIT. Project voyager. *http://www.media.mit.edu/pia/voyager*, 2001.

[95] C. Mohan. ARIES/KVL: A key value locking method for concurrency control of multiaction transactions operating on b-tree indexes. In *Proceeding of VLDB*, August 1990.

[96] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM TODS*, Vol. 17, No. 1:94–162, March 1992.

[97] G. Moody. Mit-bih database distribution. *http://ecg.mit.edu/index.html*, 2000.

[98] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[99] M. Muralikrishna and David J. DeWitt. "Equi-Depth Histograms for Estimating Selectivity Factors for Multi-Dimensional Queries". In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, pages 28–36, Chicago, Illinois, June 1988.

[100] Apostol Natsev, Rajeev Rastogi, and Kyuseok Shim. "WALRUS: A Similarity Retrieval Algorithm for Image Databases". In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, Philadelphia, Pennsylvania, May 1999.

[101] M. Ng, Z. Huang, and M. Hegland. Data-mining massive time series astronomical data sets - a case study. *Proceedings of Pacific-Asia KDD Conference*, 1998.

[102] R. Ng and J. Han. Efficient and effective clustering methods for spatial data mining. *Proc. of VLDB*, 1994.

[103] R. Ng and A. Sedighian. Evaluating multidimensional indexing structures for images transformed by principal component analysis. *Proc. of SPIE Conference*, 1996.

[104] B. Nichols, D. Buttlar, and J. P. Farrell. *Pthreads Programming*. O'Reilly & Associates, 1996.

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

[106] C. Olston and J. Widom. Offering a precision-performance tradeoff for aggregation queries over replicated data. *Proc. of VLDB Conference*, 2000.

[107] B. Ooi, C. Goh, and K. Tan. Fast high-dimensional data search in incomplete databases. *Proc. of VLDB*, 1998.

[108] J. Orenstein and T. Merett. A class of data structures for associative searching. In *Proc. Third SIGACT News SIGMOD Symposium on the Principles of Database Systems, pages 181-190*, 1984.

[109] Jack A. Orenstein. "Spatial Query Processing in an Object-Oriented Database System". In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, pages 326–336, Washington, D.C., June 1986.

[110] M. Ortega, Y. Rui, K. Chakrabarti, S. Mehrotra, and T. Huang. Supporting similarity queries in mars. *Proc. of ACM Multimedia 1997*, 1997.

[111] M. Ortega-Binderberger, Y. Rui, K.Chakrabarti, S. Mehrotra, and T. Huang. Supporting ranked boolean similarity queries in mars. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, November 1998.

[112] V. Hadzilacos P. A. Bernstein and N. Goodman. Concurrency control and recovery in database systems. *Addison Wesley*, 1987.

[113] C. Papadimitriou. The theory of database concurrency control. *Computer Science Press*, 1986.

[114] T. Pavlidis. Wavelet segmentation through functional approximation. In *IEEE Transactions on Computers*, July 1976.

[115] Viswanath Poosala and Venkatesh Ganti. "Fast Approximate Answers to Aggregate Queries on a Data Cube". In *Proceedings of the Eleventh International Conference on Scientific and Statistical Database Management*, Cleveland, Ohio, July 1999.

[116] Viswanath Poosala and Yannis E. Ioannidis. "Selectivity Estimation Without the Attribute Value Independence Assumption". In *Proceedings of the 23rd International Conference on Very Large Data Bases*, Athens, Greece, August 1997.

[117] Viswanath Poosala, Yannis E. Ioannidis, Peter J. Haas, and Eugene J. Shekita. "Improved Histograms for Selectivity Estimation of Range Predicates". In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 294–305, Montreal, Quebec, June 1996.

[118] K. Porkaew, K. Chakrabarti, and S. Mehrotra. Query refinement for content-based multimedia retrieval in MARS. *Proceedings of ACM Multimedia Conference*, 1999.

[119] D. Rafiei. On similarity-based queries for time series data. *Proceedings of ICDE*, 1999.

[120] J. T. Robinson. The k-d-b-tree: A search structure for large multidimensional dynamic indexes. In *Proc. ACM SIGMOD*, 1981.

[121] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. *Proceedings of SIGMOD*, 1995.

[122] N. Roussopoulos, Y. Kotidis, and M. Roussopoulos. Cubetree: Organization of and bulk incremental updates on the data cube. *Proc. of SIGMOD*, 1997.

[123] Y. Rubner, C. Tomasi, and L. Guibas. "A Metric for Distributions with Applications to Image Databases". In *Proceedings of the 1998 IEEE International Conference on Computer Vision*, Bombay, India, 1998.

[124] Y. Rui, T. Huang, and S. Mehrotra. Content-based image retrieval with relevance feedback in mars. *Proc. of IEEE Int. Conf. on Image Processing*, 1997.

[125] Y. Rui, T. Huang, M. Ortega, and S. Mehrotra. Relevance feedback: A power tool in interactive content-based image retrieval. *IEEE Tran on Circuits and Systems for Video Technology*, September, 1998.

[126] B. Salzberg. Access methods. In *ACM Computing Surveys, Vol. 28, No. 1*, March 1996.

[127] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley Publishing Company, Inc, 1990.

[128] S. Sarawagi. Indexing olap data. *IEEE Data Engineering Bulletin, Volume 20*, 1997.

[129] Sunita Sarawagi and Michael Stonebraker. "Efficient Organization of Large Multidimensional Arrays". In *Proceedings of the Tenth International Conference on Data Engineering*, pages 328–336, Houston, Texas, February 1994.

[130] Carl-Erik Särndal, Bengt Swensson, and Jan Wretman. *"Model Assisted Survey Sampling"*. Springer-Verlag New York, Inc. (Springer Series in Statistics), 1992.

[131] T. Seidl and H. Kriegel. Optimal multistep k-nearest neighbor search. *Proc. of ACM SIGMOD*, 1998.

[132] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-tree: A dynamic index for multi-dimensional objects. In *Proc. VLDB*, 1987.

[133] H. Shatkay and S. Zdonik. Approximate queries and representations for large data sequences. In *Proceedings of ICDE*, 1996.

[134] M. Shevchenko. http://www.ikki.rssi.ru/. *Space Research Institute, Moscow Russia*, 2000.

[135] J. Srinivasan, R. Murthy, S. Sundara, N. Agarwal, and S. DeFazio. Extensible indexing: A framework for integrating domain-specific indexing schemes into oracle8i. *Proceedings of ICDE Conference*, 2000.

[136] E. Stollnitz, T. DeRose, and D. Salesin. *Wavelets for Computer Graphics – Theory and Applications*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1996.

[137] Eric J. Stollnitz, Tony D. DeRose, and David H. Salesin. *"Wavelets for Computer Graphics – Theory and Applications"*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1996.

[138] M. Stonebraker, J. Frew, K. Gardels, and J Meredith. The sequoia 2000 storage benchmark. *Proc. of SIGMOD*, 1993.

[139] Michael Stonebraker and Dorothy Moore. Object-relational dbmss: The next great wave. *The Morgan Kaufmann Series in Data Management Systems, Jim Gray, Series Editor*, 1996.

[140] A. Szalay, P. Kunszt, A. Thakar, and J. Gray. Designing and mining multi-terabyte astronomy archives: The sloan digital sky survey. *Proc. of SIGMOD*, 2000.

[141] Illustra Information Technologies. Illustra reference manual, illustra server release 2.1. June 1994.

[142] M. Thomas, C. Carson, and J. Hellerstein. Creating a customized access method for blobworld. *Proc. of ICDE*, 2000.

[143] Purdue University. Pervasive location aware computing environments (PLACE project). *http://www.cs.purdue.edu/homes/seh/PLACE.html*, 2001.

[144] Jeffrey Scott Vitter and Min Wang. "Approximate Computation of Multidimensional Aggregates of Sparse Data Using Wavelets". In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, Philadelphia, Pennsylvania, May 1999.

[145] Jeffrey Scott Vitter, Min Wang, and Bala Iyer. "Data Cube Approximation and Histograms via Wavelets". In *Proceedings of the Seventh International Conference on Information and Knowledge Management*, pages 96–104, Bethesda, Maryland, November 1998.

[146] R. Weber, H. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high dimensional spaces. *Proc. of VLDB*, 1998.

[147] A. Weigend. The santa fe time series competition data. *http://www.stern.nyu.edu/ aweigend/Time-Series.SantaFe.html*, 1994.

[148] D. Welch and P. Quinn. Machco project. *http://wwwmacho.mcmaster.ca/Project/Overview/status.html*, 1999.

[149] D. White and R. Jain. Similarity indexing with the ss-tree. *Proc. of ICDE*, 1995.

[150] D. White and R. Jain. Similarity indexing: Algorithms and performance. *Proc. of SPIE*, 1996.

[151] D. Wu, D. Agrawal, and A. Abbadi. A comparison of dft and dwt based similarity search in time-series databases. *Proc. of CIKM*, 2000.

[152] D. Wu, D. Agrawal, A. Abbadi, A. Singh, and T. Smith. Efficient retrieval for browsing large image databases. *Proc. of CIKM*, 1996.

[153] B. Yi and C. Faloutsos. Fast time series indexing for arbitrary lp norms. *Proceedings of VLDB*, 2000.

[154] T. Zhang, R. Ramakrishnan, and M. Livny. Birch: An efficient data clustering method for very large databases. *Proc. of SIGMOD*, 1996.

Vita

Kaushik Chakrabarti received his BTech degree in Computer Science and Engineering from the Indian Institute of Technology, Kharagpur in 1996 and MS degree in Computer Science from the University of Illinois at Urbana Champaign in 1999. He is currently finishing his PhD degree in Computer Science at the University of Illinois at Urbana Champaign. His research interests include multimedia databases, information retrieval, decision support systems, data mining and database systems for internet applications like E-commerce and XML. He has published more than 25 technical papers in the above areas. His paper titled ”Locally Adaptive Dimensionality Reduction for Indexing Large Time Series Databases” received the 2001 ACM SIGMOD best paper award. His paper titled ”Approximate Query Processing Using Wavelets” was adjudged one of the best papers of the 2000 VLDB Conference (invited to the ”Best Papers of VLDB 2000” Special Issue of the VLDB Journal). He was elected into the Honor Society of Phi Kappa Phi in 1998 for having perfect GPA (4.0/4.0) in graduate school. He is a member of the ACM, ACM SIGMOD and ACM SIGKDD.