Patterns for Indexing Large Datasets

Garima Gaur
Department of Computer Science and Engineering
Indian Institute of Technology
Kanpur, India
garimag@cse.iitk.ac.in

Sumit Kalra
Department of Computer Science and Engineering
Indian Institute of Technology
Kanpur, India
sumitk@cse.iitk.ac.in

Arnab Bhattacharya
Department of Computer Science and Engineering
Indian Institute of Technology
Kanpur, India
arnabb@cse.iitk.ac.in

ABSTRACT

Searching is one of the fundamental tasks in Computer Science. An intuitive way to search is to do it *linearly*, that is, start at the beginning of the dataset and continue till the searched-for item is found or nothing is found. However, as the volume of data increases, the response time of linear search is no longer acceptable. Indexes are designed to search through massive datasets quickly. There are a number of different ways of building complex and advanced indexes. Appropriate selection and modification of indexing structures according to dynamic business requirements is crucial for data-intensive applications. In this work, we present a few basic reusable indexing structures. These structures can be used to create advanced and complex indexing structures with lesser effort and time.

CCS CONCEPTS

• **Information systems → Data access methods**; *Physical data models*; *Data scans*; **Point lookups**; *Query optimization*;

KEYWORDS

Indexing, Hierarchical structure, High-dimensional datasets

ACM Reference Format:
Garima Gaur, Sumit Kalra, and Arnab Bhattacharya. 2018. Patterns for Indexing Large Datasets. In *23rd European Conference on Pattern Languages of Programs (EuroPLoP '18), July 4–8, 2018, Irsee, Germany*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3282308.3282314

1 INTRODUCTION

With the rise in size of the World Wide Web, search engines have gained popularity in the past few decades. Some of the popular search engines, such Google, Bing, Yahoo, etc. have become indispensable. They enable a user to search for relevant information across the entire Internet. The Internet is a huge collection of approximately 60 trillion web pages, and the task of search engines is to quickly search relevant web pages whenever a user queries for them. The response time that these search engines exhibit has contributed to their popularity. The Google search engine answers 3.5 billion queries per day with an average response time of 0.5 seconds. What enables these search engines to find the relevant web-pages from trillions of web pages in just a few milliseconds? The answer is indexing!

Indexing is a process of providing a data structure over the dataset which assists in quickly accessing data. In day to day life, the most common example of an index is that given at the end of a book. The design and optimization of indexing techniques has been an active area of research in the database community for the last 4 decades. Indexes first drew attention back in 1980 when the relational database systems (RDBMs) came into existence.

Developments in software and hardware technology have enabled us to query and process larger and more complex data objects like video, audio, images, etc. With the evolving complexity and size of underlying datasets, it is important to develop newer, more efficient indexing techniques to cater to their specific needs.

In this paper, we have focused on two classes of indexing techniques, hierarchical and high-dimensional. The hierarchical indexing pattern abstracts out the underlying principles involved in designing the indexing techniques which can handle larger datsets. High-dimensional indexing covers the case when the data objects are complex and are represented as points in a high-dimensional vector space.

2 RUNNING EXAMPLE

Suppose there is a collection of toys placed in a shelf. Each toy is labeled with an *ID* number, can be triangular or oval in shape, and is of either red or green color. To quickly locate the toys, the *ID* and shape of each toy have been listed along with the location where the toy is kept, as shown in Figure 1. The location of a triangular toy having *ID* 4 can be found by looking for the pair $(4, T)$ in the list.

For the sake of further discussion, let us set up the terminology. In our example, each toy represents a *data object*. Each data object is characterized by a set of *attributes*; here, *ID*, color and shape are attributes of a toy. The list, shown in Figure 1, is a mapping which for a given pair $(ID, shape)$ produces the location of the corresponding toy. This linear list serves as a *flat* index and assists in quickly accessing the toys. The pair (*ID*, shape) is called the *key* of the index. *Key* is a subset of attributes of the data objects. Here, the toys are represented by $\{ID, shape, color\}$ and the list (index) uses $\{ID, shape\}$ as the key. The key of each data object is inserted in the index.

Mathematically, each data object O_i is interpreted as a d-dimensional vector $(o_{i1}, \ldots, o_{id})$, where $o_{ik}, 1 \leq k \leq d$, is the value of the k^{th} attribute of the key. In our example, the key is of size 2, and

EuroPLoP '18, July 4–8, 2018, Irsee, Germany

ACM ISBN 978-1-4503-6387-7/18/07…$15.00
https://doi.org/10.1145/3282308.3282314

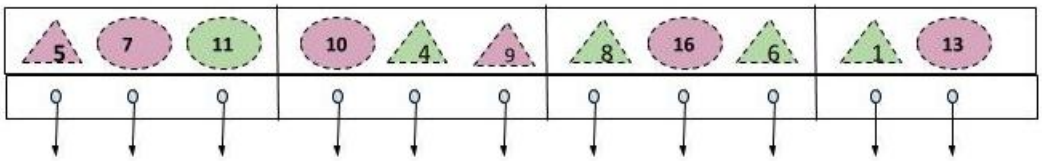

Figure 1: Linear list mapping a toy key to the location of the toy

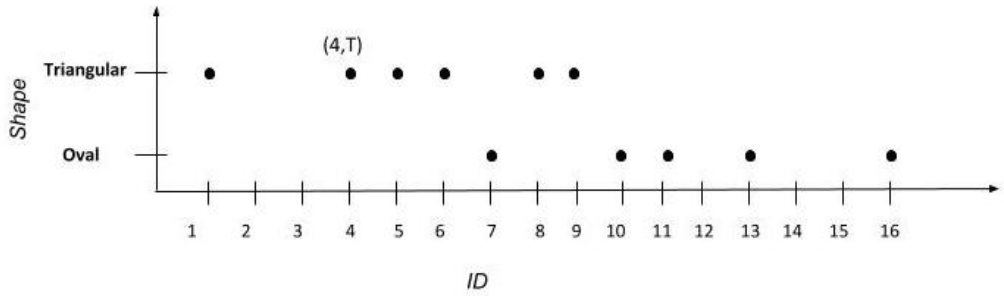

Figure 2: 2-D representation of keys of toys

the key corresponding to each toy t_i is represented as a $2D$ point $(t_i.id, t_i.shape)$, as shown in Figure 2, where $t_i.id$ and $t_i.shape$ are the id and shape of the toy respectively.

3 HIERARCHICAL STRUCTURE

3.1 Intent

To design efficient indexes for large datasets.

3.2 Context

Once the size of datasets is huge, data objects are stored on the disk. A disk is logically divided into numbered pages and each object resides in one of these pages. To read an object, the page containing that data object is copied to the primary memory. This operation of reading a page from disk to primary memory, known as disk I/O, is costly. An index maps an object key to the page number in which it resides.

With the increase in dataset size, the size of index increases too and, hence, the index structure also has to be stored on the disk. If using the index involves a high number of I/O operation then the purpose of building the index is defeated. Thus, the objective of indexing is to access the data object using as few disk I/O operations as possible.

3.3 Problem

Suppose instead of 11 toys we have a collection of 10^4 toys. The flat index will correspondingly have 10^4 entries. Linearly scanning this index to find the location of a toy is time-consuming. The drawback of a *flat* index is that one may need to go through each entry to find the relevant one. Thus, larger the index, more might be the page read operation to access it. The higher number of disk I/O increases the overall index lookup time and, makes it inefficient. We, therefore, need a different strategy to index large datasets.

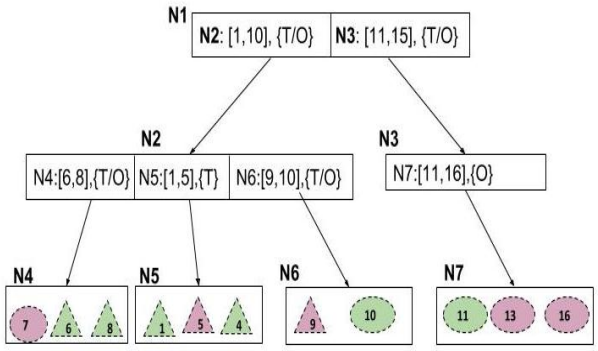

Figure 3: Hierarchical index

3.4 Forces

- Partial loading of the index is not efficient when data is accessed frequently.
- Unnecessarily loading a part of the index which does not contain the key is inefficient.

3.5 Solution

Suppose instead of maintaining the linear list of keys as in Figure 1, we have arranged the keys as shown in Figure 3. To look up a key, say $(4, Triangular)$, we go down the tree, starting from the root node $N1$, choosing the next node to traverse based on the description of the child nodes ($N2$ in this case), and keep traversing down till we reach the leaf node which contains the key, if it exists. Note that as we are traversing down the tree, we are essentially narrowing down the search space, much like in a binary search. This results in a potential $log(n)$ disk page accesses per retrieval.

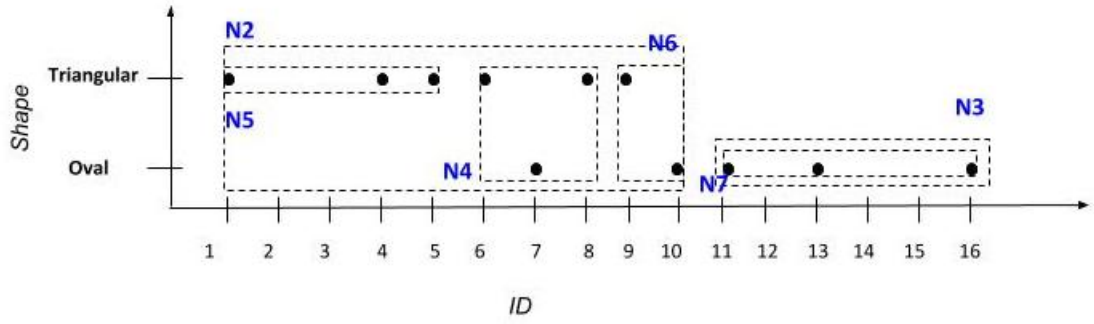

Figure 4: Hierarchical index partitioned the space

In full generality, hierarchical indexes are tree-like structures, where internal nodes contain the child partition descriptor to guide the traversal, and leaf nodes contain the key and the pointer to the corresponding object location. Each object key belongs to only a single leaf node.

The geometrical representation of a hierarchical index is shown in Figure 4, where each dotted enclosing rectangle corresponds to a node in the index and the parent-child relationship is captured as a containment of enclosing regions. The data is partitioned by some notion of proximity of objects.

3.5.1 Discussion. Before we explain the working of the index, let us discuss some other aspects of hierarchical indexes,

- *Fanout of nodes*: A hierarchical index is designed so that each node resides in a single disk page. So, the fanout of a node is dependent on how many node descriptors a page can hold. In our toy index, a descriptor is the range of attributes of the key (ID, shape). Let the average size of a descriptor be D bytes. This size can be expected to scale linearly with the number of dimensions of the *key*. The number of children a node can have can be approximated by

$$\text{Node fanout } (f_{out}) \approx \frac{P}{D}$$

  where P is the size of a page in bytes. Since the number of page accesses is correlated to the height of the tree

$$\text{No. of pages} \approx log_{f_{out}}(n)$$

  which is logarithmic in the size of the dataset.
  On the other hand, a flat index would store a fixed number of keys on each page and, thus, the expected number of page accesses for retrieval would be linear in the size of the dataset. It is apparent then that hierarchical indexes scale much better than flat indexes in terms of page access.
- *Partitioning methodology*: Based on which aspect is used to create partitions, hierarchical indexes are classified into data-partitioning and space-partitioning indexes. In data-partitioning, the partitions are created based on the data while trying to maintain a balanced distribution of data across partitions. The partitioning shown in Figure 4 is data-driven.
  On the other hand, a space-partitioning strategy partitions the space into subspaces without taking into account the data distribution in the subspace, as shown in Figure 5. Since, the density of data is not taken care of while partitioning, it may so happen that some of the subspaces created are empty. Space-partitioning indexes might be space inefficient, but are still are used in various applications because they are easier to create.

3.5.2 Structure. A hierarchical index is a tree-like structure, where internal nodes specify partitions of the data space which can be dependent on the order in which new objects are inserted and the leaf nodes contain keys and pointers to the actual data.

As discussed in Section 3.5.1, the maximum number of records that can reside in a node depends on the size of a record key and the size of a disk block. Further, a lower bound on the data stored per node is imposed to avoid inefficiency in space usage. This implies that our structure has a lower bound on the number of children of internal nodes and a lower bound on the number of records in a leaf node.

3.6 Dynamics

If the database is dynamic then ease of inserting and deleting an object from the index becomes crucial. Efficiency of updating the index serves as the basis of the choice of indexing technique. We now briefly discuss these operations.

3.6.1 Insertion. Whenever a new record is inserted, the hierarchical structure is traversed in a top-down manner to find a leaf node in which the data record can be added. Suppose, in our running example, a new toy $(15, triangular, Red)$ is added into the collection. To update the index, we traverse the index from root node $N1$ to appropriate node $N7$ and insert the key. Since, $N7$ was already full, this latest insertion would cause *overflow*. In this case, the node $N7$ is split to create two new sibling nodes $N8$ and $N9$ as shown in Figure 6. The data objects residing in the original node are distributed into these new nodes. Various strategies can be employed to distribute the data from the original node. For ease of exposition here, we randomly distribute the keys to $N8$ and $N9$.

Recall that each internal node also lies entirely on a single page. The split shown in Figure 6 can thus further cause node $N3$ to overflow because of the added partition descriptor to it's contents.

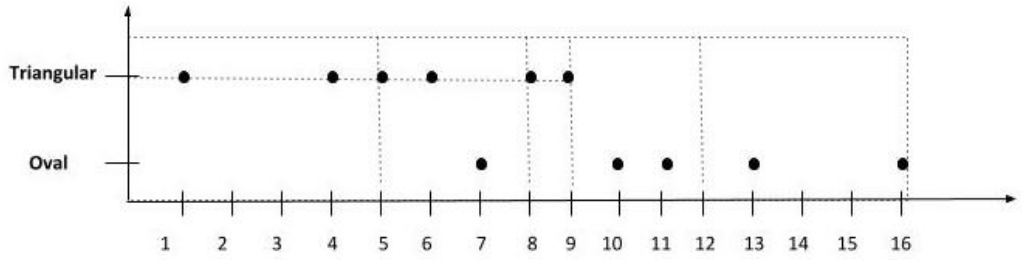

Figure 5: Partitions created by *space-partitioned* indexing

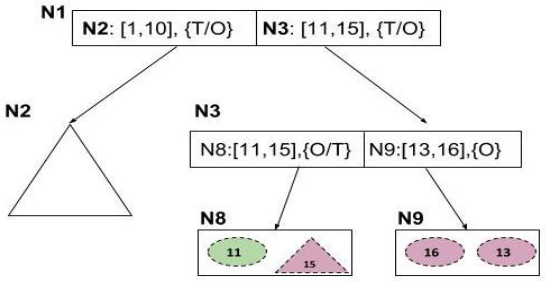

Figure 6: Updated hierarchical index after insertion of key $(15, triangular)$

In this case, we employ a similar split on $N3$ and propagate the change to it's parent (in this case $N1$).

Lastly, it is important to note that the splitting methodology dictated by particular applications may create sibling nodes with overlapping descriptors. We showcase this in the aforementioned split into $N8$ and $N9$ in Figure 7. Such overlap happens organically in an R-tree, which is discussed in Section 3.8. Such overlapping nodes are detrimental to the performance of the index. Suppose we wanted to search for the key $(13, triangular)$. Starting from $N1$, traversing down to $N5$, we would explore both $N8$ and $N9$ as either could have the key. This clearly leads to more disk I/O. Such splitting, while possibly unavoidable in certain applications, does mar the efficiency of the index.

3.6.2 Deletion. An object is deleted by first locating its key in the index and then deleting the key from the node containing it. Generally, a deleted key is marked as "deleted" and, then later all such keys are removed from the index in batches.

Deletion of a record may cause the number of records in the affected node to fall below the lower bound imposed. If this happens then the node is merged with one of its siblings. In case the merged node overflows, we would employ a split as in the case of insertion. Also, a merge propagated to its parent node can result in underflow there which we would resolve by another merge at that level.

Both, the splitting and merging of nodes, thus, propagate their effects in bottom-up fashion.

3.7 Consequences

3.7.1 Advantages.

- **Fast data point retrieval:** In the case of simple splitting methods that do not result in overlapping node descriptors, O(log n) disk accesses are performed.
- **Easy updates:** Updating index is fairly easy when new data points are introduced.
- **Load only relevant part of the index:** Index tree provides the search path such that only relevant part of the index needs to be accessed.

3.7.2 Liabilities.

- Nodes, like $N3$ in our example, might have some unused space.
- Deletions of data points do not necessarily decrease the index size.
- For small datasets and very high-dimensional datasets, linear scan outperforms hierarchical indexes.

3.8 Known Uses

- **B-tree [1]:** B-tree is one of the most commonly used data structure in various database management systems and file systems such as Microsoft's NTFS, Apple's HFS+, Mysql. Prior to the use B-tree the indexes were sequential files containing keys of records. For locating a record, the system needs to read the entire index file to search for the corresponding key. B-trees are designed based on the idea of maintaining a parent-child relationship, i.e., a hierarchy, among the record keys. Such a hierarchy enables the user to fetch only those blocks of the index file which are essential to navigate the system to the desired record.
- **R-tree [6]:** R-trees are similar to B-trees; however, they are capable of indexing multi-dimensional objects. R-trees partition multidimensional data into regions called minimum bounding regions (MBR), and maintain a hierarchy among these MBRs.

4 HIGH-DIMENSIONAL STRUCTURE

4.1 Intent

Designing indexing techniques to efficiently access high (≥ 8) dimensional data objects.

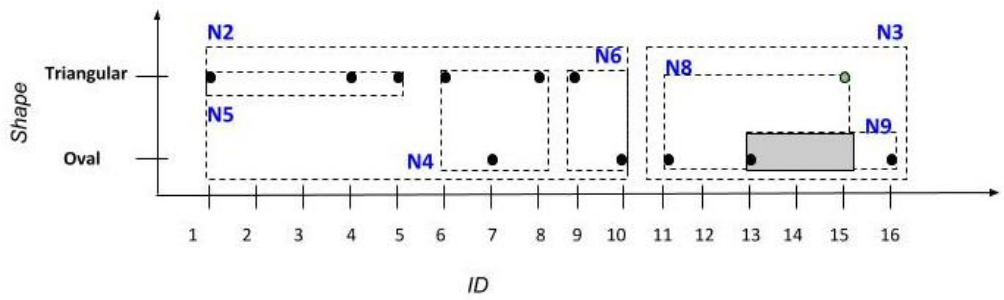

Figure 7: Overlapped space generated after the insertion of key $(15, triangualr)$

4.2 Context

Datasets with complex object types such as text, videos, photos, etc., are represented as a collection of high-dimensional points where each data object is transformed to a high-dimensional vector. The data space is generally modeled as a unit hyper-cube. The nature of the problem - like, say, the data distribution involved - changes as the dimension of the space increases.

4.3 Problem

Recall from Section 3.5.1, that the size of a partition descriptor in a hierarchical index is dependent on the number of dimensions of the key. Therefore each node can accommodate lesser number of partition descriptors. Due to this, insertion operation causes node overflow more frequently. Depending upon the splitting methodology, this might also result in a greater number of overlapping siblings. Thus, as the dimensionality of the data increases, we might end up exploring multiple paths while performing a search operation. This makes the existing hierarchical indexing techniques ineffective for high-dimensional data.

4.4 Forces

F1 Hierarchical index structures are balanced as they expect the data to be roughly uniformly distributed, but data points in high-dimensional spaces are generally closer to the boundary of the space, a phenomenon which is commonly known as the *curse of dimensionality*.

F2 It is commonly observed that not all the dimensions (attributes) of a high-dimensional data objects have the same information quotient. Higher the information quotient higher will be the variance in the value of that attribute across objects. This property of high-dimensional data can be leveraged for indexing them.

4.5 Solution

One way to address the problem is to use the standard dimensionality reduction techniques such as SVD [5], PCA [8], IsoMap [10], etc., and embed the high-dimensional data space into some lower dimensional space and then use the usual hierarchical approach, discussed in Section 3, to index the data points. Force **F2** drives this solution.

The other category of solutions is based on adopting different strategies for partitioning the space. One way is to avoid the node

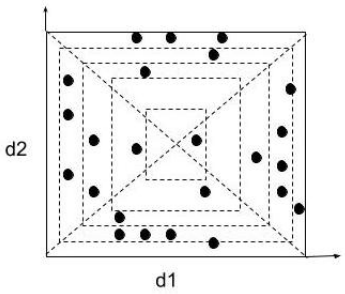

Figure 8: Structure driven by partitioning the space differently [4]

partitioning by employing special nodes which are flexible in size unlike usual internal nodes of hierarchical indexes. These nodes, while spread over multiple pages on the disk, are still guaranteed to be a contiguous block and thus do not add too much of an overhead in terms of disk I/O. This can reduce the number of overflow events.

Another approach targets the uneven density of data points mentioned in force **F1**. This can be done by partitioning the space so as to match the density of partitions with the *expected* density of data points in any region. This can help create more balanced indexes. This category of solutions is a special class of hierarchical indexing.

4.6 Structure

Based on which pre-decided partitioning approach we use, different indexing techniques are designed, for instance, the pyramind technique, discussed in Section 4.8, creates more partitions near the boundaries of the data space as that is an area with high data point density.

Modified hierarchical structures is shown in Figure 9. The blue colored nodes are *special* flexible nodes along with the normal ones. These special nodes do not have fixed sizes and are usually bigger than normal nodes.

4.7 Consequences

4.7.1 Advantages.

- Gains from low-dimensional indexing techniques do not have to be completely forgone.

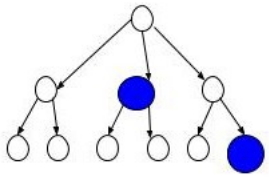

Figure 9: Structure with *modified* nodes

- Complexity in implementation over the low-dimensional techniques lies mostly in the setup:
  - Mapping to a lower dimensional space
  - Establishing heuristics as to which nodes are special
  - Deciding an initial space partitioning scheme

4.7.2 Liabilities.

- Due to the inherent complexity resulting from the large number of dimensions, index update operations are costlier.
- Requires more upfront knowledge of both the data space and distribution.

4.8 Known Uses

- **X-tree [3]:** X-tree, a widely used indexing technique to handle high-dimensional data objects, is influenced by hierarchical and sequential indexing techniques. An X-tree is a modified R-tree with special *supernodes*. Supernodes, unlike R-tree nodes, are not of fixed size. Whenever an insertion results in the splitting of a node, supernodes are created so as to avoid splitting. It is an effective indexing choice for high (> 4) dimensional data.
- **Pyramid Technique [2]:** This technique proposes a different way of partitioning the high-dimensional space. In a d dimensional space, $2d$ pyramids are built, 2 pyramids corresponding to each dimension, as shown in Figure 8. Each pyramid (partition) is further sliced off parallel to the base of pyramid to create smaller partitions such that each subpartition can reside in a disk page. Many variants of this technique exist, such as the Extended pyramid technique [2], IMinMax index [9], iDistance [7].

5 CONCLUSIONS

We have presented two patterns which broadly cover the indexing techniques for large datasets. The provided patterns are generic, in the sense that they do not adhere to any particular data model. The work is presented with the hope of serving a dual purpose: helping students to understand the basic ideas behind the existing indexing techniques, and to assist practitioners to design hybrid indexing techniques which are suitable for the problem at hand.

6 ACKNOWLEDGMENT

We would like to thank our shepherd Brigit van Loggem for her valuable inputs on arranging the information in a more comprehensible manner. Special thanks to the discussion group at EuroPLop'18 whose suggestions have shaped the final version of this paper.

REFERENCES

[1] R. Bayer and E. McCreight. 2002. *Organization and Maintenance of Large Ordered Indexes.* Springer Berlin Heidelberg.
[2] Stefan Berchtold, Christian Böhm, and Hans-Peter Kriegal. [n. d.]. The Pyramid-technique: Towards Breaking the Curse of Dimensionality. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data (SIGMOD '98).*
[3] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. [n. d.]. The X-tree: An Index Structure for High-Dimensional Data. In *Proceedings of the 22th International Conference on Very Large Data Bases (VLDB '96).*
[4] Arnab Bhattacharya. 2014. *Fundamentals of Database Indexing and Searching* (1st ed.). Chapman & Hall/CRC.
[5] G. H. Golub and C. Reinsch. 1971. *Singular Value Decomposition and Least Squares Solutions.* Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-86940-2_10
[6] Antonin Guttman. [n. d.]. R-trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data (SIGMOD '84).*
[7] H. V. Jagadish, Beng Chin Ooi, Kian-Lee Tan, Cui Yu, and Rui Zhang. 2005. iDistance: An Adaptive B+-tree Based Indexing Method for Nearest Neighbor Search. *ACM Trans. Database Syst.* 30, 2 (June 2005), 364–397. https://doi.org/10.1145/1071610.1071612
[8] I.T. Jolliffe. [n. d.]. *Principal Component Analysis.* Springer Berlin Heidelberg.
[9] Beng Chin Ooi, Kian-Lee Tan, Cui Yu, and Stephane Bressan. 2000. Indexing the Edges&Mdash;a Simple and Yet Efficient Approach to High-dimensional Indexing *(PODS '00).*
[10] Joshua B. Tenenbaum, Vin de Silva, and John C. Langford. 2000. A Global Geometric Framework for Nonlinear Dimensionality Reduction. (2000).