Database Indexing: Time Series Simulation Comparing R-tree and Inverted Index

Shana Weissman
Department of Computer Science
Earlham College
Richmond, IN 47374
syweiss15@earlham.edu

ABSTRACT

It is common knowledge that databases power important research along with the web and other widely used tools. It is less known that a database's driving force is its index. A database's index has the functionality of a traditional index; it is used to find the location of a specific item or value in a pool of information. In this case, the data set is what is indexed. Indexes are complex structures with many variations each having their strengths and weaknesses. This applies to the subset of indexes used for every type of data, for example, text data, relational data, etc. This paper describes a comparison via simulation, SIM_TSDB, of two specific index structures used for time series data. The two indexes are the R-tree and the Inverted Index. SIM_TSDB, written in Python, tests the speed of finding a value's location through the index. Both these structures are not conventionally used for time series databases. Therefore, a large part of my contribution is adapting them for this use. It does not interact whatsoever with an actual time series database. The results answer the question, which of the two indexes should be used depending on what is being done with time series data and the volume of it.

CCS CONCEPTS

• **Time Series Database** → **Index Structures**; • **Time Series Databases** → *Building Indexes*;

KEYWORDS

Time Series, Indexing, R-tree, B+ Tree, Inverted Index, Simulation, Data Base

ACM Reference Format:
Shana Weissman. 2019. Database Indexing: Time Series Simulation Comparing R-tree and Inverted Index. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.475/123_4

1 INTRODUCTION

A major component of any database back-end is the index. Its function is the same as the index of a book. It is used to look up the location of a concept in the book, to save the reader from scanning the entire book for what they are looking for. In a database, it is used to locate specific data points, which databases need to do for many purposes. This paper will focus on locating data points stored in a database for the purpose of querying. The database index itself is a data structure of which there are many variations. The structure depends on what database it is used for, e.g. relational, graph, time series, etc. However, even in a specific database type, there are many different structures used. This paper focuses on the data structures used for indexes in time series databases. Time series data represents sequential measurements taken over time. It is extremely common for data to be measured at specific times especially in astronomy, biology and the web [4]. Time series databases are in need of more focus as it becomes increasingly popular.
Because individual structures have their own traversal algorithms to locate values, different structures are more efficient than others when it comes to certain queries. The time series database index structure that should be used depends on the data and what it will be used for. Due to the high volume of in-use time series database index structures, the scope of this paper pertains to two structures, the R-tree structure[1], and the Inverted Index [6]. The reason for the selection of these specific structures is because they differ widely. Their structures are discussed in the background section for further detail.

This paper attempts to answer the question, in what situations are each of these index structures favorable and when would another option be preferable? In order to answer this question, a simulation is conducted in Python containing three main parts. The three main parts are made up of core simulation containing a query generator, and the two structures and their respective algorithms. The simulation tests the structure's algorithms outside of a database. The simulation does not test the efficiency on retrieval of data, only locating the data along with other methods of the index, such as creation, addition, and removal of values. Because of this, the only data required to perform such tests are the timestamps. These timestamp values are taken from non-synthetic data sets.

The simulation approach to answering the question of which time series database is most fitting for a user's goals has many limitations. Because a simulation of the indexing structures alone, many other factors of a database that affect its usage are not being considered. Storage is a key area of focus when designing an index structure and must be efficient in order for success. Nevertheless, storage is not considered in this paper. Furthermore, it is only comparing two of the many in-use time series index structures.

The second goal of this paper is to explore the use of unconventional index structures for time series data. Both the R-tree and Inverted Index structures were first established and are standards

Conference'17, July 2017, Washington, DC, USA
© 2019 Association for Computing Machinery.
ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

for databases other than time series databases. These structures are adapted to take timestamp values as input rather than their original, intended input. It is valuable to experiment with already working techniques and applying them to other areas for the purpose of exploration.

2 BACKGROUND

2.1 Search Queries

Because this paper discusses the tests of queries across different index structures, it is helpful to the reader to know what these queries represent. Several of the queries used in this paper are intuitive as to their purpose such as an insert or remove. However, the three other tested queries, equality query, bounded range query (range query), and unbounded range query are less obvious. An equality query is when a single value is searched for. What is being searched in a range query is all values between two specified values. For example, a retrieval of all timestamps between the year 2015 and 2016 is a range query. An unbounded range query is a query similar to a range query, but the values being retrieved are not bounded by two values. For example, a retrieval of all values later than a specific time.

2.2 Traditional Index Structure

2.2.1 B+ tree. The B+ tree is one of the most commonly used structures for commercial databases [13]. Although research is still underway for efficient indexing for time series data, the B+ is commonly used for indexing it's databases. The B+ tree was developed from the B tree, also known as the Binary tree. Much like the B+ tree, the Binary tree is a popular structure for indexes. A simple way to understand the Binary tree is that it is a tree representation of Binary search on an array. In the B tree's structure, each node contains a single entry with a pointer to a file containing information about the entry. A major downside to the B tree is that its height can easily become excessively long, which significantly increases search time. Generally, the height of a B tree grows quickly because of two reasons. Firstly, A Binary tree is limited to only two children per node. Secondly, because each entry is stored in the same node as its pointer, each node can only hold one entry due to the substantial amount of space a pointer requires. Therefore, more nodes are necessary to hold the data stored in the index. The B+ tree is a variation of the B tree that does not have these limitations. The B+'s internal nodes have to ability to have more than two children and each node holds multiple entries. To allow multiple entries in a node, the pointers are held in the leaf nodes.

The B tree's node's left child must contain a value that is less than it's value and it's right child must contain a value that is greater than it's own. The B+ tree follows the same rule, however, there can be multiple entries in each node, and a value q is set for the maximum number of pointers each node contains. Each leaf node, except the node containing the greatest entries, dedicate one of it's pointer to the leaf node that is consecutively greater than it. This is shown in figure 1. In this example, q = 5. Although this figure shows the insertion of value 29, the B+ insertion algorithm is not described, because it is not necessary for the understanding of this paper. A B+ tree performs an equality query rather simply by traversing down the tree until found. On the other hand, the B+

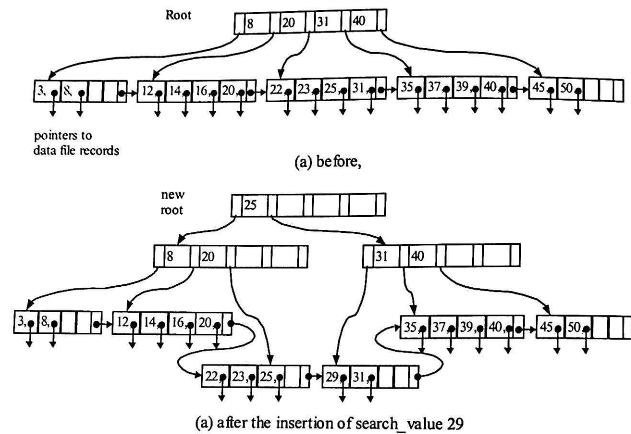


Figure 1: The Structure of the R-tree [12]

tree's algorithms for range and unbounded range queries utilize the pointers in the leaf nodes as to avoid unnecessary traversals.

2.3 R-Tree

The R(rectangle)-tree was built in the 1980s by Antonin Guttman as an extension of the B+ tree. The R-tree was created as a solution to the B+ tree's inability to hold multimedia data, which was beginning to be widely used when it was created. For the most part, in order for the R-tree to be able to hold multimedia data, it needs to be capable of holding high dimensional data. It is now mostly used for spatial data, including GIS data [12] i.e. cities, buildings, etc. The general idea of the R-tree is that data is bounded by rectangles. These rectangles have sub-rectangles for which the lowest level of rectangles contains the actual data values with pointers to files, similar to how pointers are stored in the leaf nodes of the B+ tree. The R-tree represents this rectangle structure; each node represents the area of a rectangle. The structure is shown in figure 2. These rectangles are called minimum bounding rectangles (MBR), they are rectangles with the smallest possible area that bounds it's sub-rectangles. The R-tree is height balanced, meaning that all the leaf nodes are at the same level.

Time series databases are not typically indexed using the R-tree, however, it can be adapted so that it is compatible with time series data. This has been done in the past by mapping timestamps to a higher dimension value [12], SIM_TSDB uses a different approach. For SIM_TSDB, rather than using rectangles, a time range is used for each node in the place of a rectangle's area. Each node represents a range of time. The root node contains the largest time range of all the nodes, each child node narrows down the range until the leaf node is reached. The leaf nodes contain the entries and pointers. Similar to the B+ tree, the R-tree's leaf nodes point to each other chronologically.

The insert algorithm for inserting a value into the R-tree [13], does not rely on the structure containing rectangles. Therefore, the insert algorithm is not changed for this scenario. The search algorithms require consideration as well in order to ensure their success for time series data. Because this adaptation of the R-tree contains leaf nodes that use pointers to keep an account of the order of the leaf

nodes, range and unbounded range queries can be easily performed. While the insertion algorithm is complex, the search algorithm is simple and easily implemented.
Because the application of the R-tree on time series data is one dimensional like the B+ tree, it is important to emphasize the differences between the B+ tree and the SIM_TSDB application of the R-tree. The B+ tree has a maximum number of children allowed, whereas the R-tree does not. Furthermore, due to the concept of the MBR, the insertion algorithm for the R-tree takes into account the smallest possible range each node needs to represent it's sub-tree, while the B+ tree's internal nodes do not represent values in the same way. This implementation of the R-tree uses different algorithms and node structures than the B+ tree.

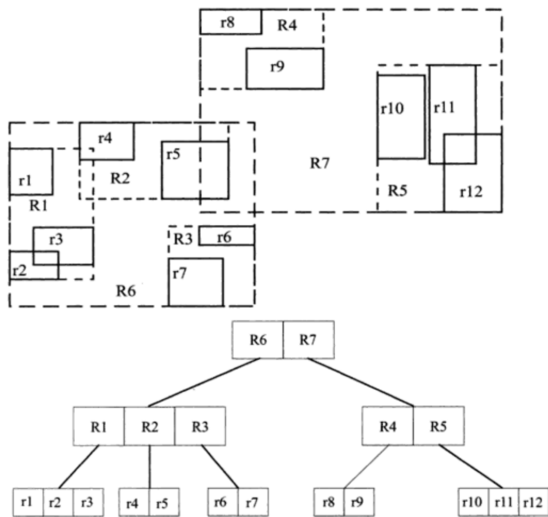


Figure 2: The Structure of the R-tree [12]

2.4 Inverted Index

The Inverted Index structure is a well-established text database index. It has been adapted for the use of time series data[6] prior to SIM_TSDB. When used in it's traditional, text database framework, it consists of a lexicon describing every word in the database. This lexicon is stored in an array and it alphabetically ordered. Each value in the array is mapped to a file obj, known as an inverted index, containing the information about this word. The method of adapting the Inverted Index used for SIM_TSDB is intuitive. Rather than the lexicon array containing a list of words, it contains each unique timestamp. To allow better searching, the values in the timestamp lexicon will be ordered from earliest to most recent. A figure of the Inverted Index is shown below.

A strength of the Inverted Index is that, since it is very simple, it is easy and relatively fast to implement. However, the adaptations it undergoes to be used for a time series database add complexities that a traditional time series index does not have.

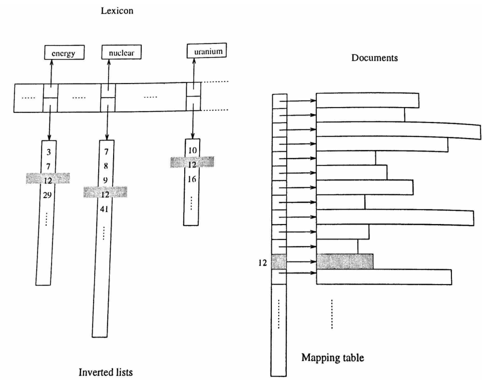


Figure 3: The Structure of the Inverted Index [2]

3 METHODOLOGY

3.1 Overview

The general workflow of the simulation is shown in 4. It takes a CSV file of a time series data set as input, creates two index structures for the data set, queries each structure and stores the results. Three different data sets were used for testing. The three main divisions of the simulation consist of the two data structures along with a driving function. The driving function contains a query generator which generates and carries out queries. It also serves as a communication line between the simulation and the database that stores the simulation results.

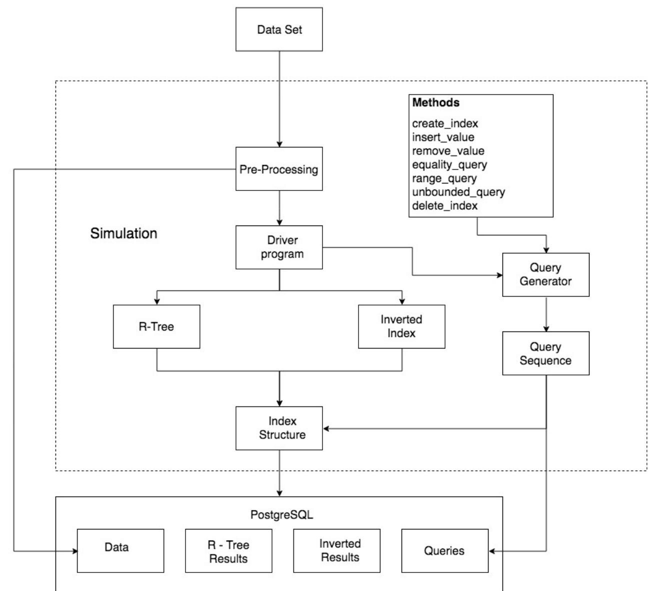


Figure 4: Design of Simulation

3.2 Data Pre-processing

In order to test and debug the simulation, data is needed. Often, finding data a data set that fulfills certain criteria is easier said than

done. Additionally, it is generally a given that unprocessed data needs to be cleaned so that it can run smoothly through a program. Because this simulation is only testing the index structures of time series databases, the only needed part of each data set are the timestamps. This significantly cuts down on the amount of data cleaning required, however, the unclean data can cause issues when filtering out the timestamps out of its complete data set containing other information. For example, extra commas inside of entries will miss-align the columns putting a non-timestamp value in a timestamp column. Pre-processing of the data is needed to fix issues like this. The code itself is not very long to extract these specific values, but the process of finding all errors when ingesting the data and correcting them can be time-consuming. This is not done manually, however, there are typically many typos and errors in the data that must be worked around one way or another in the ingesting program. Another factor that needs to be accounted for in the pre-processing of the data is re-formatting of the timestamp. This is done automatically using the `datetime` Python module in a simple program as part of the driver function. Converting the timestamps into a consistent format helps ensure that the rest of the simulation will run smoothly.

3.3 Simulation

3.3.1 Index Structures. Each of the two index structures are represented in their respective modules. Both modules contain query methods that are carried out by the driver program. These specific query methods are, for the most part, universal across index structures. The methods in table 1 are used in both index structures.

Method	Description
`create_index`	Creates an empty instance of an index structure.
`insert_value`	Inserts a timestamp into a given index structure.
`remove_value`	Removes a specific value from a give index structure.
`equality_query`	Queries the index for one specific value
`bounded_query`	Performs a bounded range query for a range of values between two times.
`unbounded_query`	Performs an unbounded range query for a range of values that are not between two times
`delete_index`	Deletes entire instance of the index

Table 1: Index Structure Methods

The algorithms for these methods are adapted from [13]. The algorithms for the R-tree structure are much more complex than that of the Inverted Index. The `equality_query`, `bounded_query` and `unbounded_query` are special cases when it comes to the Inverted Index. The Inverted Index is a text database index that has been adapted for a time series database. In order to create these methods, the Inverted Index is sorted by time. Binary search is used to find the bounds of the ranges or the specific value. The adaptation of the R-tree uses a method similar to the B+ tree for the `bounded_query` and `unbounded_query`.

3.3.2 Driver Program. As mentioned in the overview, the driver program pre-processes data, creates the index structures, queries these structures and stores the results. In order to secure accurate results, the number of queries tested must be large. In order to avoid manually authoring a large number of queries, a query generator is used. The generator runs 500 iterations of randomly-generated queries for each of the three data sets on both of the index structures and stores the results. Each iteration runs each query except the `create_index` and `delete_index` methods. This system queries for a random time value between the earliest and latest timestamp in the data set.

For queries requiring two timestamps, such as the bounded and unbounded queries, a second random value is used.

3.4 Storing Data

A separate database to store values is necessary for several different functions. Most importantly, it is needed to store the results, since there is a large amount of data that needs to be analyzed following each simulation. Each data set stores 5,000 query results alongside the timestamps from each data set. The size of each data set can be found in the results section. Furthermore, the cleaned timestamp values need to be stored prior to creating an index. These values are also used for the query generator so that it can generate queries based on the actual data it is querying. A relational PostgreSQL database is chosen to store this data. It is a suitable option because it communicates well with Python, and can easily compare the queries between the two structures seeing as it is relational. The module `psycopg2` is used to write PostgreSQL commands and alter the database directly from the driver program. Each data set has its own database with four tables which have the same layout. The first table holds the timestamp values. The next two tables hold the results from querying, one table is for the Inverted Index and one for the R-tree. The last table holds all the queries that were tested on each index structure. The tables with the results consist of a column with a query id value and the time it took to complete the query. The query table holds two columns with a query id and the query corresponding to said query id.

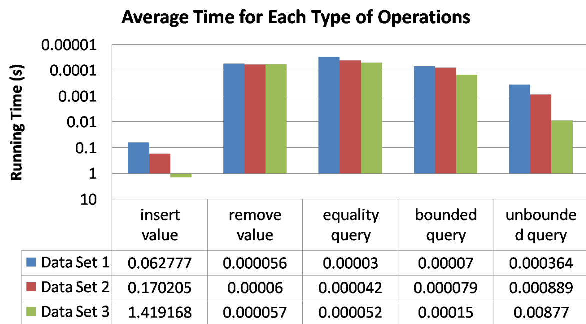
4 RESULTS

4.0.1 Inverted Index. The following figures show the results from the Inverted Index testing.

Method	Data Set 1	Data Set 2	Data Set 3
`insert_value`	0.062777	0.170205	1.419168
`remove_value`	0.000056	0.00006	0.000057
`equality_query`	0.00003	0.000042	0.000052
`bounded_query`	0.00007	0.000079	0.00015
`unbounded_query`	0.000364	0.000889	0.008774
Data Set Size	8,089	21,165	160,214

Table 2: Index Structure Methods

Table 2 shows the average seconds for each query as a result of the query generator. 500 queries were performed on each type.



	insert value	remove value	equality query	bounded query	unbounded query
Data Set 1	0.062777	0.000056	0.00003	0.00007	0.000364
Data Set 2	0.170205	0.00006	0.000042	0.000079	0.000889
Data Set 3	1.419168	0.000057	0.000052	0.00015	0.00877

Figure 5: Logarithmic Chart of Inverted Index Results

Figure 5 is a logarithmic chart representing table 2.

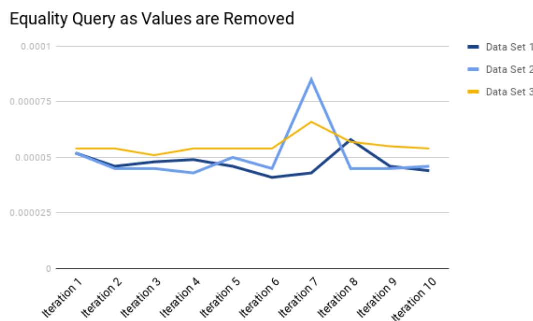


Figure 6: Equality Query Series Results

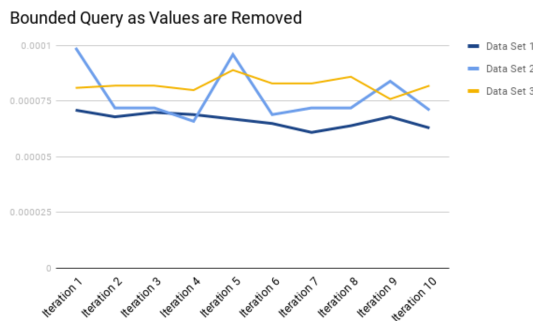


Figure 7: Bounded Query Series Results

The above tables show the change in time for each search query over the removal of values. Each iteration represents the removal of 500 random values.

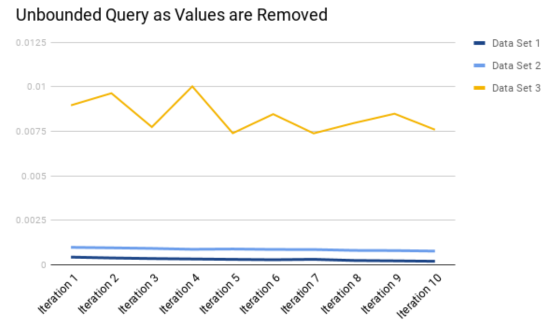


Figure 8: Unbounded Query Series Results

5 DISCUSSION

Databases serve as mechanisms for such things as holding, querying, adjusting, updating data. Different strategies for how a database functions serve better depending on the variation in the data type and volume. On examining the results for the Inverted Index, there is a clear indication that the time for the `insert_value` and `unbounded_query` are affected by the size of the data set. In chart 2 there are different results for each of the three data sets for those two queries. When taking into account the sizes of the three data sets, Data Set 3 being by far the largest, the time increases with the size of the database. The results for the unbounded query series further support these findings. Generally, removing the values from the index didn't have much of an effect on the query times, but the large gap between Data Set 3 and the other two data sets shows how the difference in size affects the time of the unbounded query.

6 CONCLUSION AND FUTURE WORK

This project has designed and tested an adaptation of the Inverted Index and designed the an adaptation of the R-tree for time series data. The first step in future work is to complete tested for the R-tree. Because the sample size is rather small, testing the simulation on more data sets of larger size would be more telling on the effect of the different index structures and their strengths and weaknesses. Furthermore, the visualizations created are rather simple due to time constraints. Putting more thought and time into making particularly appealing visuals would have been ideal and helpful for spreading the results of the simulation. The simulation itself also only tests two index structures out of a wide array of existing time series index structures. There are many different ways of testing the efficiency of index structures, for example rather than timing the methods, an in-depth analyses of each algorithm's complexity would provide useful information.

REFERENCES

[1] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. 1996. The X-tree : An Index Structure for High-Dimensional Data. In *VLDB*.
[2] E. Bertino, B.C. Ooi, R. Sacks-Davis, K.L. Tan, J. Zobel, B. Shidlovsky, and D. Andronico. 2012. *Indexing Techniques for Advanced Database Systems*. Springer US. https://books.google.com/books?id=XqLhBwAAQBAJ
[3] Christian Böhm, Stefan Berchtold, and Daniel A. Keim. 2001. Searching in High-dimensional Spaces: Index Structures for Improving the Performance of

Multimedia Databases. *ACM Comput. Surv.* 33, 3 (Sept. 2001), 322–373. https://doi.org/10.1145/502807.502809

[4] Alessandro Camerra, Jin Shieh, Themis Palpanas, Thanawin Rakthanmanon, and Eamonn Keogh. [n. d.]. Beyond one billion time series: indexing and mining very large time series collections with iSAX2+, journal=Knowledge and Information Systems, year=2014, month=Apr, day=01, volume=39, number=1, pages=123–151, issn=0219-3116, doi="10.1007/s10115-012-0606-6", url=https://doi.org/10.1007/s10115-012-0606-6. ([n. d.]).

[5] Philippe Esling and Carlos Agon. 2012. Time-series Data Mining. *ACM Comput. Surv.* 45, 1, Article 12 (Dec. 2012), 34 pages. https://doi.org/10.1145/2379776.2379788

[6] Lazin Eugene. 2017. Inverted Index. https://akumuli.org/akumuli/2017/11/17/indexing/

[7] Lazin Eugene. 2018. Akumuli. https://github.com/akumuli/Akumuli.

[8] Joseph M. Hellerstein, Elias Koutsoupias, and Christos H. Papadimitriou. 1997. On the Analysis of Indexing Schemes. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '97)*. ACM, New York, NY, USA, 249–256. https://doi.org/10.1145/263661.263688

[9] M.J. Hernandez. 2013. *Database Design for Mere Mortals: A Hands-on Guide to Relational Database Design*. Addison-Wesley. https://books.google.com/books?id=_n4hBQAAQBAJ

[10] Eamonn Keogh, Kaushik Chakrabarti, Michael Pazzani, and Sharad Mehrotra. 2001. Locally Adaptive Dimensionality Reduction for Indexing Large Time Series Databases. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data (SIGMOD '01)*. ACM, New York, NY, USA, 151–162. https://doi.org/10.1145/375663.375680

[11] David Maier and Gottfried Vossen. 1993. *Query Processing for Advanced Database Systems* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[12] Y. Manolopoulos, A. Nanopoulos, A.N. Papadopoulos, and Y. Theodoridis. 2010. *R-Trees: Theory and Applications*. Springer London. https://books.google.com/books?id=1mu099DN9UwC

[13] Yannis Manolopoulos, Yannis Theodoridis, and Vassilis J. Tsotras. 2000. *Advanced Database Indexing*. Kluwer Academic Publishers, Norwell, MA, USA.

[14] Last Mark, Mark Last, Horst Bunke, and Abraham Kandel. 2004. *Data Mining in Time Series Database*. World Scientific Press.

[15] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (01 Jun 1996), 351–385. https://doi.org/10.1007/s002360050048

[16] Michail Vlachos, Marios Hadjieleftheriou, Dimitrios Gunopulos, and Eamonn Keogh. 2003. Indexing Multi-dimensional Time-series with Support for Multiple Distance Measures. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '03)*. ACM, New York, NY, USA, 216–225. https://doi.org/10.1145/956750.956777

[17] Kostas Zoumpatianos, Stratos Idreos, and Themis Palpanas. 2014. Indexing for Interactive Exploration of Big Data Series. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. ACM, New York, NY, USA, 1555–1566. https://doi.org/10.1145/2588555.2610498