2015

# Time Series Similarity Search in Distributed Key-Value Data Stores Using R-Trees

Aleksey Charapko
*University of North Florida*, a.charapko@unf.edu

# TIME SERIES SIMILARITY SEARCH IN DISTRIBUTED KEY-VALUE DATA STORES USING R-TREES

by

Aleksey Charapko

A thesis submitted to the
School of Computing
in partial fulfillment of the requirements for the degree of

Master of Science in Computer and Information Sciences

UNIVERSITY OF NORTH FLORIDA
SCHOOL OF COMPUTING

April, 2015

The thesis, "Time Series Similarity Search in Distributed Key-Value Data Stores Using R-trees," submitted by Aleksey Charapko in partial fulfillment of the requirements for the degree of Master of Science in Computing and Information Sciences has been

Approved by the thesis committee: (Date)

_____ _____

Dr. Ching-Hua Chuan
Thesis Advisor and Committee Chairperson

_____ _____

Dr. Behrooz Seyed-Abbassi

_____ _____

Dr. Sanjay Ahuja

_____ _____

Dr. Roger Eggen

Accepted for the School of Computing:

_____ _____

Dr. Asai Asaithambi
Director of the School

Accepted for the College of Computing, Engineering, and Construction:

_____ _____

Dr. Mark Tumeo
Dean of the College

Accepted for the University:

_____ _____

Dr. John Kantner
Dean of the Graduate School

CONTENTS

FIGURES

# ABSTRACT

Time series data are sequences of data points collected at certain time intervals. The advance in mobile and sensor technologies has led to rapid growth in the available amount of time series data. The ability to search large time series datasets can be extremely useful in many applications. In healthcare, a system monitoring vital signals can perform a search against the past data and identify possible health threatening conditions. In engineering, a system can analyze performances of complicated equipment and identify possible failure situations or needs of maintenance based on historical data.

Existing search methods for time series data are limited in many ways. Systems utilizing memory-bound or disk-bound indexes are restricted by the resources of a single machine or hard drive. Systems that do not use indexes must search through the entire database whenever a search is requested.

The proposed system uses multidimensional index in the distributed storage environment to break the bound of one physical machine and allow for high data scalability. Utilizing an index allows the system to locate the patterns similar to the query without having to examine the entire dataset, which can significantly reduce the amount of computing resources required. The system uses an Apache HBase distributed key-value database to store the index and time series data across a cluster of machines. Evaluations were conducted to examine the system's performance using synthesized data up to 30 million

data points. The evaluation results showed that, despite some drawbacks inherited from an R-tree data structure, the system can efficiently search and retrieve patterns in large time series datasets.

# Chapter 1

# INTRODUCTION

Recent advances in mobile and sensor technologies have led to rapid growth in the amount of time series data. Time series data are sequences of data points collected over certain time intervals. In time series, two consecutive data points differ not only in their respective values, but also in the time they were obtained. Time series data exist in many fields of knowledge, including but not limited to music, engineering, natural sciences and medicine.

Many vital signals collected by medical sensors are time series. Cardiograph (ECG) is one example of time series data used frequently in medicine. The amount of ECG data can be enormous; just one day worth of ECG reflects approximately one hundred thousand heart beats in a day, making an analysis of the entire cardiograph nearly impossible by a doctor [Buza11].

In engineering, data collected from various sensors are often used in testing and analyzing new designs. Such tests can produce enormous amounts of data, depending on the design factors. For instance, during the flutter testing of the Airbus A380, engineers used over a hundred different sensors to capture the oscillation frequencies that were stored for later offline analysis [LMS Test.Lab14]. The data obtained from these sensors are a typical representation of a real-valued time series. Since flutter can potentially

destroy an aircraft, data collected from many tests must be analyzed before the plane is considered safe for further tests and commercial use. Therefore, a system that supports a high quality analysis as quickly as possible is needed.

In music, various features can be represented as real valued parameters progressing over time. For instance, musical notes in a composition can be represented as their MIDI pitch value and duration that change over time to express different musical ideas. Music improvisation systems utilize music represented in such numeric formats to generate new material based on the learned patterns. For example, Pachet used pitch and pitch duration to build a system capable of real time learning and improvising in the style of the musician interacting with the system [Pachet03].

In all of the examples mentioned above, time series data are first collected and then stored for analysis at a later time. The speed of the system used for such analysis is critical for many applications. While researchers may be interested in different information needed in their particular use, the capacity to quickly search the dataset for patterns is useful in many applications. For example, the ability to search for and identify similar subsequences in large datasets is useful for extracting patterns associated with potentially critical conditions, like an equipment failure or important medical state, that require immediate attention.

## 1.1 Problem Statement

The goal of this research is to design a system that provides a fast search and retrieval backbone for large amount of time series data. Existing search methods for time series data are limited in many ways. Generally, systems are designed based on two different approaches to support time series data search. One type of system creates indexing structures to record data in such a way that fast retrieval can be achieved by examining the index structure without scanning through the original time series. But such systems utilizing memory-bound or disk-bound indexes are restricted by the resources of a single machine or a hard drive. The other types of systems that do not use indexes must search through the entire database whenever a search is requested. In this study, a system that takes the advantage of indexing but eliminates the storage limitation by operating in a distributed environment is proposed.

In order to design the search system, the manner in which time series data are used must be examined. In many cases, one might want to search the historical data against the new data coming into the system in order to identify whether the new input fits an existing pattern. This is important when analyzing a performance on complicated equipment by identifying possible failure situations based on the operational history. Similar concepts can be applied to the medical field, in which new data can potentially signal serious health conditions. In music improvisation systems, historical training data provide the basis for improvisation. When a new sequence arrives, the system can find similar

sequences in the training set, and use the existing patterns that follow the similar sequence to generate new musical content [Pachet03].

Most search systems are required to perform search to locate exact patterns in the dataset. Exact search is defined in literature as guaranteed to return all relevant results to the query [Keogh01A] or a search that produces the same results as a sequential scan algorithm for a given similarity metric [Keogh05]. Consider a sequence S = {2, 3, 1, 5, 4, 2, 4, 1, 3} and a search query Q = {5, 4, 2}. The search for Q in S results in the discovery of a single match M = $\{s_i \mid i = 3, 4, 5\}$, where $s_i$ is an element in S with a zero-based index *i*. Unfortunately in many cases such perfect matches do not exist. For example, the recording equipment may have a degree of error and can produce slightly different values while the actual parameter being measured stays the same. Also it is often necessary to identify a condition that can lead to a particular event, but such a condition may produce distinct yet similar patterns from time to time. Perfect match is a bad choice for identifying such conditions, because it will produce a small number of isolated instances for the condition.

A search based on similarity allows us to retrieve patterns that are similar to the query to different extents. This solves the problem of sensor fluctuations and isolated instances of perfect match. One of the most widely used similarity searches is known as the *k* Nearest Neighbors (*k*-NN) search. A *k*-NN query (Q, *k*) will retrieve a match set M consisting of *k* time series segments that are the most similar to the query segment Q. The similarity between Q and the segments in M is often defined through a distance function D. For any

two time series segments $C \in M$, $E \notin M$, $D(Q, C) \leq D(Q, E)$ [Keogh01A]. Consider time series S = {2, 3, 1, 5, 4, 2, 4, 3, 3, 1} with a *k*-NN query Q = {2, 3, 1} and *k* = 2. Such a query will return a set M of *k* time series sequences that are closest to the time series Q. Taking Euclidean distance as the similarity function D, we can expect M to have two segments: {2, 3, 1}and {3, 3, 1}. In order for *k*-NN to produce the exact search, it needs to return *k* most similar sequences to the query entered. For instance, for *k* = 3, the system needs to retrieve the first, second and third most similar sequences based on the defined similarity function. If an algorithm finds the first, second and fourth closest match, it can no longer be called exact search algorithm because it has not found exactly three most similar sequences.

As the amount of data increases, the need to have faster perfect and *k*-NN searches in the time series sequences becomes increasingly more important. In this thesis, the proposed system uses R*-tree multidimensional index stored in a distributed key value data store in order to fulfill these requirements. R*-tree is a multidimensional index in the R-tree family of data structures, and it is commonly used for spatial data analysis. Existing usages of R-tree indexes for time series search have been limited to non-distributed systems. This study extends the limit by storing the search index in Apache HBase, a NoSQL distributed database used for storing key-value data pairs. HBase enables to distribute the index structure across a cluster of computers to provide high data scalability.

The proposed system was evaluated by studying the impact made by various factors on the system's performance. Index dimensionality is known to affect the performance of R-tree indexes. Thus, the effect of dimensionality was tested on the R*-tree index in the distributed environment. Maximum R*-tree node capacity was also tested in another experiment. Each R-tree node can have multiple children nodes. The maximum number of children per node controls the depth of a tree, which can affect the performance of a model. The scalability of the proposed system was evaluated by comparing to the sequential file scan algorithm. In addition, experiments were conducted to examine how data distributions, namely uniform and normal distributions and random walks, affect the performance. Finally, different cache mechanisms were also examined.

The proposed system showed promising results; it was able to outperform the sequential scan algorithm in many of the conducted tests. The system demonstrated the ability to search and index large datasets of millions of data points with high scalability potential, and it also showed many possibilities for further refinement and improvement.

Chapter 2 of this thesis discusses the literature on the subject of time series retrieval. An initial attempt using relational databases for time series search is described in chapter 3. Chapter 4 provides the details on the design and implementation of the proposed system in a distributed environment. The experiments and results for the system evaluation are presented in chapter 5. Finally, possible enhancements for future work are proposed in chapter 6, followed by the conclusions in chapter 7.

# Chapter 2

# LITERATURE REVIEW

## 2.1 Similarity Functions

Similarity between two sequences can be defined in various ways. In addition, the similarity measure is also application dependent. For example, Kahveci and Singh [Kahveci01] pointed out that Euclidean distance, being one of the most widely used distance metrics for time series similarity, can be insufficient for applications in which some time series sequences are constant multiples of each other and should be considered similar.

Non-Euclidean distance functions have been used for establishing time series similarity as well. Perng proposed a new similarity model and an indexing technique based on the 'landmark' events, such as local minimums and maximums, and used these events to reconstruct the time series based on these events [Perng00]. The model supports basic time series transformations, such as scaling, shifting and time warping, and these transformations are then used to compute the similarity between two time series. The authors did not indicate whether the system performs well when trying to find the similarity in a subsequence matching, or finding similarities between subsequences in a large time series dataset.

Chebyshev distance was used fairly often in calculating time series similarity. Agrawal argued that Euclidean-based similarity metric is more sensitive to outliers and used $L_\infty$ distance metric instead for the time series similarity [Argawal95].

Dynamic Time Warping (DTW) is another popular similarity metric used in time series similarity search [Rakthanmanon13]. DTW reduces the impact of different time scales between two sequences. Significant research has been devoted to developing similarity metrics based on DTW; some of the resulting approaches outperform the original DTW [Rakthanmanon13]. According to Keogh et al. in [Keogh05] Dynamic Time Warping cannot be indexed easily because it does not obey triangular inequality.

Some of the similarity measures described in earlier this section are used with specific search algorithms. For example, Dynamic Time Warping measure is used in DTW searches. Others can be used in various contexts. Euclidean distance is mostly universal and can be utilized in various indexing and search systems. Many other similarity measures are described in the time series literature, but the ones mentioned above are most widely used and/or serve as the basis for many other metrics.

## 2.2 Approximate Techniques for Similarity Search

Many researchers have been using approximate techniques for similarity search, claiming that most applications do not need an exact search and that certain error or underreporting is permissible in favor of the improved performance [Keogh01A].

Wang argued that it is not critical to retrieve all matches to the query at the beginning stages for certain time series applications. The exact search is only added after users have refined and narrowed down their search [Wang00]. The authors used least squares approximation to fit the time series subsequences into consecutive line segments. The approximation used in Wang's research allowed for a great speed up at the expense of precision and recall: a speed up of one order of magnitude compared to the exact search yield the result achieving a 60%-70% precision and recall levels. However, the datasets used for this research were fairly small, about 101 thousand data points [Wang00]. Therefore, it is unknown how well the method will scale up.

Park [Park99] used string matching techniques to carryout approximate similarity search. The time series is first broken up into segments. A set of feature vectors is then generated from each segment. Similar feature vectors are grouped together and each group is assigned a symbol to represent it. The sequences of symbols are then used to create suffix tree on which the search is carried out later. The authors claimed that their system performed 6.5 times better than the sequential search, although the dataset of less than a million data points was limited by the modern standards. As a result, it is unclear how the system will perform with more data.

## 2.3 Exact Time Series Similarity Search and Retrieval

Unlike the approximate time series similarity search solutions, the result set of exact search must contain all existing relevant matches and no irrelevant ones. In the case of a

perfect match, the exact search solution finds all time series identical to the query. In contrast, the exact solution in a *k*-NN search retrieves *k* most similar sequences. For example, in the case of a *k*-NN search with $k = 5$, the result set is guaranteed to hold 5 most relevant matches.

For exact search, the retrieval time is the major concern for the system. Many different techniques have been proposed for the problem. Researchers have used various indexing schemes to speed up retrieval at a search time [Keogh01A], [Loh00], [Kahveci01], while others work on improving the performance of immediate solutions that do not require an underlying indexing structure [Rakthanmanon13].

Keogh used a multidimensional indexing structure, namely R-tree, to index time series data [Keogh01A]. Multidimensional indexes are widely used for time series similarity search. A time series subsequence S of size *n* can be treated as a point in *n*-dimensional space, making multidimensional indexing techniques viable for searching and retrieving subsequences from large time series datasets [Keogh01A]. Figure 1 illustrates this concept: (a) a raw time series, (b) the segments of equal size from the raw sequence, and (c) *n*-dimensional points that can be used in various multidimensional indexing algorithms.

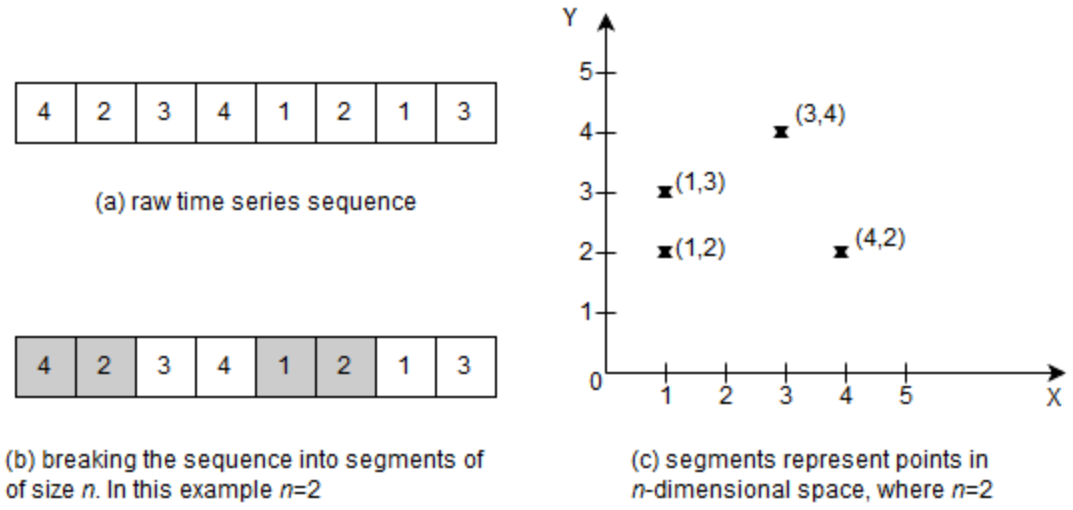



Figure 1. Representing time series as a sequence of *n*-dimensional points.

The most commonly used multidimensional indexing structures for time series indexing are R-trees and variations of R-trees, such as R+-tree and R*-tree [Loh00]. The performance of these structures tend to degrade as the dimensionality increases [Keogh01], [Zhou13], and the degradation becomes significant when the number of dimensions reaches 8 to 12 [Keogh01A]. Unfortunately, many applications require search queries of fairly large size; for example, the queries of 1000 data points are quite common. This requirement will mandate the segment size and the number of dimensions for the indexing structure to be large as well [Keogh01A]. In order to improve the performance of indexing techniques, dimensionality reduction is commonly used [Keogh01, Loh00].

Keogh et al. presented a survey of common dimensionality reduction techniques used for time series indexing and similarity searching in [Keogh01B]. These methods include

Discrete Fourier Transform, Singular Value Decomposition (SVD) and Discrete Wavelet Transform (DWT). In addition, the authors proposed a different method for dimensionality reduction called Piecewise Aggregate Approximation (PAA), which breaks down the sequence into equal-sized segments and calculates the mean value for the data in the segment. The vector formed from such mean values becomes the reduced representation of the original sequence. In the paper later published by Keogh, the technique was improved by allowing the segments to be of varying size [Keogh01A].

Multidimensional indexing allows us to easily search a set of sequences for ones that match a query sequence. This type of matching, called whole matching, assumes that all sequences and queries are the same length [Keogh01A]. Because the size of the query is known beforehand, an index can be constructed to tailor the query of that particular size. Subsequence matching is a more difficult problem because of the numerous offsets that the query must be compared in the matching sequence. Keogh [Keogh01B] used a sliding window to match the query to a subsequence of the time series. In [Kahveci01], Kahveci created a multiresolution index. Kahveci used DFT and wavelets to reduce the dimensionality of the time series to a number of different dimensions, and constructed an index for each of these dimensions. The author created a system that indexes information at different resolutions, which allow for a more efficient use of information contained in the query [Kahveci01]. The author claimed that the proposed system addresses another important issue in sequence matching: the query length can be unknown in many applications. In order to solve the problem of the unknown query length, the proposed method is then required to store multiple indexes: a separate index structure needs to be

computed for each resolution level. As a result, the requirement of storage space increases because of the multiple index structures.

Dynamic Time Warping (DTW) has been used as the similarity metric for the time series, but many researchers have neglected the approach due to the general notion of DTW being slow on large datasets [Rakthanmanon13]. DTW relies on dynamic programming and a general DTW algorithm performs in $O(n^2)$ time [Lemire09], which seems less than ideal when scalability is important. However, Rakthanmanon applied a number of optimizations to the DTW algorithm. The optimizations include various data preprocessing and normalization improvements and early abandoning of the computation when no match is possible [Rakthanmanon13]. These changes led to the ability to perform exact *k*-NN searches on the dataset of one trillion data points in reasonable time on a single server built with common hardware [Rakthanmanon13].

## 2.4 Multidimensional Indexing in Distributed Key-Value Data Stores

Distributed Key-Value data stores such as Apache Cassandra and Apache HBase gained a lot of popularity in recent years. HBase is an open source implementation of Google's BigTable, it builds on top of the Apache Hadoop and HDFS to provide scalable and fault tolerant big data store [HBase14].

Research has been carried out in using such data stores for indexing of multidimensional data. Wei used KR*-tree, which is a variant of an R-tree, to index user generated spatial

data [Wei13]. Authors stated that key-value data stores work most efficiently when performing a scan operation and retrieving multiple items whose keys are in the range of the scan operation. With this concept in mind, the authors broke the search space into non-overlapping squares of equal size and used Hilbert space-filling curve to assign each of these squares a Hilbert value. When building the R*-tree, the authors mapped tree nodes with the squares, essentially creating an index underneath the R*-tree index structure and allowing faster retrieval of multiple R-tree nodes that might be relevant to the search. The authors claimed that their approach outperforms the state-of-the-art multidimensional indexing techniques for distributed systems. It is worth noting that Wei et al. focused on two-dimensional spatial data, therefore it is unclear how well the proposed system will perform as the dimensionality increases.

On the contrary, Zhou et al. claimed that R-trees and the variants are not meant for being used in the distributed key-value data stores [Zhou13]. The authors indicated that distributed environments such as Apache HBase are not suitable for R-trees due to their relatively poor performance on random access of a single record in the database. Instead of using traditional multidimensional indexing techniques, the authors used Location Sensitive Hash that is capable of operating on multidimensional data. The authors claimed that their system performs better than traditional multidimensional indexing approaches, especially as the data dimensionality increases. They also stated that the proposed system is capable of handling data with thousands of dimensions, although this claim was not tested in the paper [Zhou13].

## 2.5 Music Systems Relying on Time Series Data

Musical improvisation systems represent another application domain for systems operating on time series data. Pachet in the *Continuator* system used real valued sequences obtained from musical features as the data for the improvisation system [Pachet03]. *Continuator* is a real time music improvisation system. It learns from the input provided by the musician and is able to generate new music of a similar style in response to the musician's query. The system constructs in-memory prefix trees as the user plays into the system. At the query time, system searches the trees for all matches and identifies all ways the query was continued in the past. It then selects one of the possible continuations, transforms it in one of the predefined ways and plays it back as the improvisation to the query sequence [Pachet03]. The biggest disadvantage of the *Continuator* is its limited space for storing historical data. The system learns from the input user provides at the moment, but historical data can be too much for the system to store.

Other improvisation systems utilize data mining techniques to extract important information from the training set for music generation. Halkiopoulos extracted pitch and note duration from the training input, divided the extracted data into variable size segments, and produced feature vectors for each segment [Halkiopoulos12]. Halkiopoulos et al. then grouped the vectors based on their similarity and applied association mining to establish a set of association rules. Such mined rules were later applied to the query to provide a continuation to the query. The authors claimed that the

proposed system, called *POLYHYMNIA*, produced high quality improvisations such that the human listeners were unable to tell whether the improvisation were produced by a machine or a professional musician. *POLYHYMNIA* was trained on a fairly small data set (101 jazz melodies and 414 Bach's Chorales), and it is unknown how bigger training sets will impact the performance of the system and the quality of the improvisation. In addition, the similarity metric developed for the system allows authors to add exceptions to the similarity manually, which can be used as a tool to "fine-tune" the system to a specific dataset.

The two music improvisation systems mentioned earlier in this section differ in the approach taken to solve the problem. But both systems have some common attributes, such as having to utilize real valued sequential data and define similarity between data segments.

# Chapter 3

# AN INITIAL ATTEMPT USING RELATIONAL DATABASE

The *Continuator* system utilizes the prefix search trees in order to perform the lookup on the time series the system has encountered [Pachet03]. As the system learns more information, it needs to maintain larger tree structures that degrade the performance. The *Continuator* is a real time system and it does not need to maintain any historical information.

A search algorithm of the *Continuator* was used as a basis for developing the initial solution to the time series similarity search problem. The algorithm was modified not to rely on main memory but to use relational database instead. Using the database for storing the search trees was intended to improve the scalability beyond the limitations of physical memory installed in the machine. Similar to the original indexing and search model in the *Continuator*, the initial solution was designed to retrieve all possible continuations to the query.

## 3.1 Sequence Indexing and Retrieval

The initial solution expanded the *Continuator* model in many ways. In addition to the relying on relational database for storage, the model was also changed to have lesser number of trees and reuse existing trees or branches whenever possible to save space.

Compound hash digest was created in order to speed up locating the right branch of the needed tree without having to traverse the entire tree structure. The following sections describe the proposed solution for improving the indexing and search model in *Continuator*. More details about the proposed approach can be found in [Charapko14].

### 3.1.1 Basic Model

The process of building a model begins by segmenting the input sequences and assigning a unique identifier to each segment in the incremental order. A reduction function is applied to each segment to allow searching for similar subsequences. For instance, if a reduction function R is applied to sequences S and S' and produces two identical sequences, then S is similar to S':

$$\text{if } R(S) \equiv R(S'), \text{then S is similar to } S'.$$

The result of the reduction function for each segment is parsed from right to left to construct the prefix trees. Every tree node represents an atomic element of the input sequence and maintains a list of continuations from the original sequences. Each node also stores path information in order to identify the branch of the tree when the search is performed. The path information is hashed at each node as well. Storing the path data along with the hash of such path may seem redundant, but such data allows the model to reduce the amount of unnecessary tree traversals and improve look up speed.

Let's consider the input sequence: {*a b c d*}. The proposed model builds the trees containing all possible prefixes for the input. Model construction starts from the right of the sequence by examining node *d*. Element *c* is the prefix of *d*, so node *c* becomes the root of a tree with element *d* in its continuation list. Node *b*, being the prefix of *c* becomes a child of *c* and for similar reason the node *a* becomes a child of *b*, all with node *d* in the continuation lists. For every created edge, a *path hash* value is computed by producing a small hash digest of the two nodes that make up the vertices of the edge. The digest is then appended to the path hash of the previously examined edge in the path. The built tree for the input sequence is illustrated in Figure 2 (a). In the figure, hash digests are labeled as "h" followed by a number to designate different digests. For example, the edge *c*-*b* is labeled as "h1" and the edge *b*-*a* as "h2". As a result, the hash digest for the entire path from node *c* to node *a* is "h1h2". At this point, nodes *a*, *b* and *c* all have a continuation index of 4, which corresponds to the segment *d* from the input.

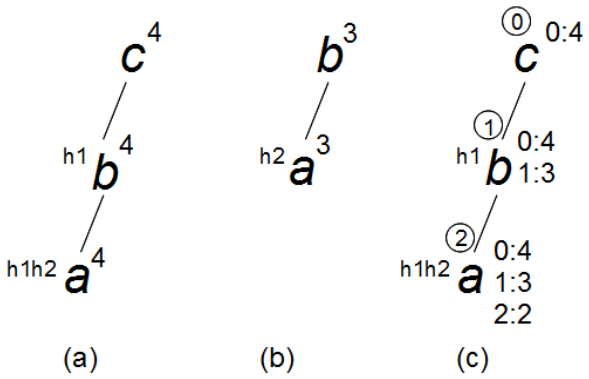



Figure 2. Initial approach model

Parsing of the input sequence continues since not all prefixes have been accounted yet. So far only the prefix structure for node *d* has been placed into the model. The parsing resumes without the last element of the segment, i.e., the sequence {*a b c*} is parsed now. The original *Continuator* model will create a new tree as shown in Figure 2 (b) because no other trees with root *b* exist so far [Pachet03]. The proposed model reuses the existing structure and does not create a new tree. As it can be observed, branch *b-a* already exists in the tree created in the previous step with only one difference being the continuation lists. The model reuses the existing tree structure, but keeps the continuations separate for each case. It is important not to mix the continuation lists from various iterations. For example, the continuations recorded during the parse of {*a b c d*} should not be mixed with those for {*a b c*}. Similarly, we parse sequence {*a b*} and add the appropriate continuations to the corresponding node in order to build the complete tree with all prefixes for the input as shown in Figure 2 (c).

### 3.1.2 Model for Long Time Series

The basic model is limited by the length of the input. In particular, the input length determines the depth of the trees. Therefore, a very deep tree with limited branching can be expected from a very long sequence. A sliding window approach was utilized in order to control the depth of the trees. The goal is to increase the breadth of the trees and to improve the reusability of branches. The input sequence was first divided into half-overlapping windows and the basic model was then used on each window for indexing.

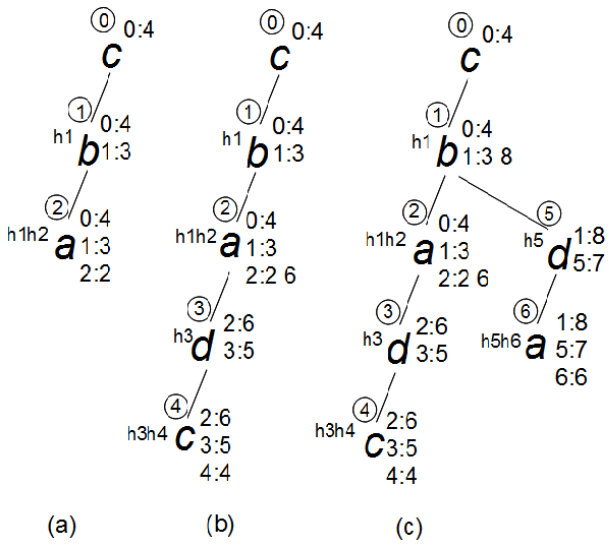



Figure 3. Initial approach on longer input sequences

Suppose a longer input sequence {*a b c d a d b c*} is processed using a sliding window of size four. The window {*a b c d*} is parsed first, producing the model as shown in Figure 3 Figure *3*(a). The same mechanism is applied to the next window, except the newly produced tree is merged with trees already existing in the database. For example, considering the second window {*c d a d*}, the node *d* has a prefix of {*c d a*}. Instead of creating a new tree with root node *a*, an existing tree is continued from the node *a*. The process is repeated until all the prefixes of {*c d a d*} are processed as shown in Figure 3 (b). Note that the node *d* has hash digest "h3" instead of "h1h2h3". This is because edge *a-d* was the first edge created while processing the input window {*c d a d*}. Similarly, the model is updated as in Figure 3 (c) after the last window {*a d b c*} is processed.

### 3.1.3 Retrieving Sequences and Continuations

The original *Continuator* model scans the all the root nodes of the trees created in search for the tree which defining the prefixes of the last element of the query. The tree is then traversed down until following the query from right to left. Unlike the original model, the proposed approach does not traverse the trees; instead it computes the digest for the hash path from the query and uses such digest to retrieve all candidate branches. Once the candidate branches are retrieved they are checked for hash collisions, leaving only the branches that represent the query in the result set. The last element of the branch will contain the list of continuations from the original sequence.

## 3.2 Experiments and Results

The initial solution was tested against a few different datasets. The model is very generic and does not restrict to a certain data type. The performance of the model was studied on the textual data. The compilation of Wikipedia articles was used for the performance evaluation with the smaller dataset consisting of 13.5 thousand words and larger one being 47.8 thousand words. Each word was treated as an atomic value, thus each node in the model represented one word from the articles. The model was built on top of MySQL database running on the windows machine. Sequential scan was chosen as the benchmark, as it is easy to implement and is commonly used for search tasks.

The performance impact from the query length was studied for both intuitive and file scan approaches. A set of 100 queries of same length guaranteed to have at least one match in the input sequence was generated. The same set of queries was tested on both search algorithms and search times recorded. The process was repeated ten times and the average time was computed for each query size. Figure 4 shows the results for the test runs. The intuitive solution outperforms the sequential scan algorithms queries of larger size. Poor performance on the search on queries of length two is most likely due to the very large number sequences and continuations returned, as each sequence had to be checked for hash collisions.

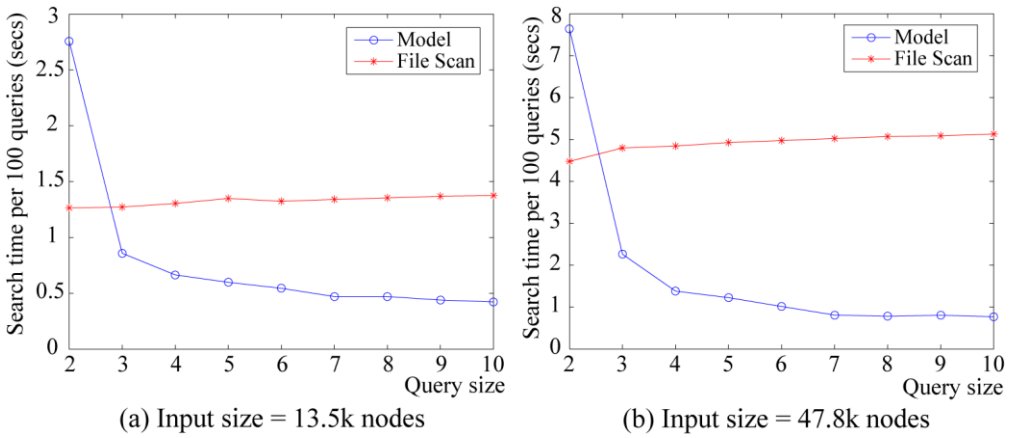



Figure 4. Average search time for varying sizes of query strings

The impact of the result set size was studied in a different experiment. A set of queries of size two was generated where each query was guaranteed to return the same number of results. The performance was measured on both systems and repeated five times for each result set size. The results of this experiment are presented in Figure 5. As can be seen,

increasing the number of results returned by a query negatively impacts the performance of the system. It is also worth noting that the initial solution performed better compared to the sequential file scan on larger dataset.

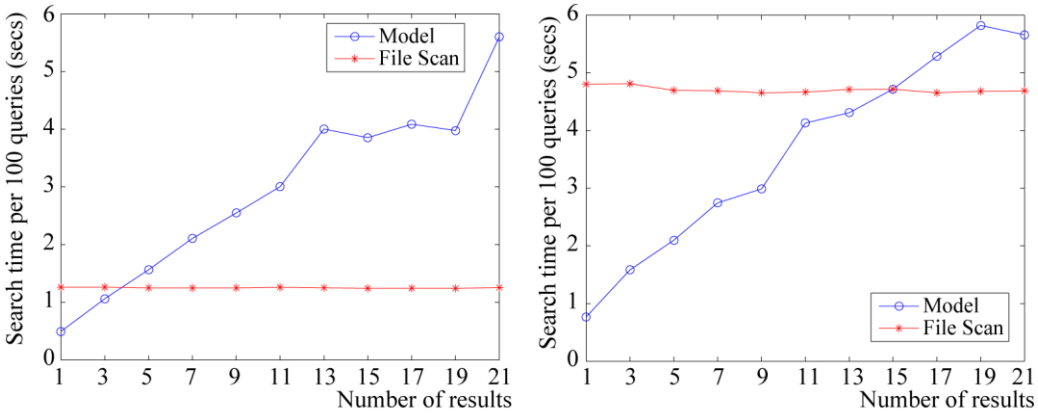



Figure 5. Average search time versus number of results with query size of two.

## 3.3 Conclusions and Discussions on the Initial Solution

The initial approach based on the model used in the *Continuator* has a number of advantages compared to the other solutions to the problem. In particular, it is capable of handling textual and categorical sequences. It was implemented using the free and open source software easily available online without the need of complex configuration or any special knowledge of such software products. The initial approach was faster in most cases compared to regular sequential file scan algorithm working with files stored on a disk.

Unfortunately, this solution is lacking the scalability and performance needed by many applications. The size of the build model is another issue; in the best-case scenario it was four times larger than the raw data. The need to reduce the database interactions for any system relying on the database storage became evident first hand while developing the initial solution, as reducing the number of database read operations played the big role in making the system perform faster than the sequential search.

# Chapter 4

# METHODOLOGY

This chapter first describes the overall architecture of the proposed system, evaluation goals, search and retrieval criteria for the system, and the key components including HBase and R*-tree. It then explains how time series data are processed and indexed in this study. Algorithms for search and retrieval are presented next, followed by the discussion of cache strategies that can potentially speed up the search process.

## 4.1 System Architecture

Existing time series search systems generally suffer from two major problems: inability to scale up or inability to use past computations to speed up future searches. Many of the existing time series systems create an index structure in memory or local disk and perform a search using the index. Such systems are generally limited to the capability of a single machine they reside on, and even disk bound indexes are limited by the main memory limitations of a single machine. For example, the size of the entire indexing structure must be smaller than the size of the memory. Other systems, such as the ones relying on DTW generally do not have indexing capabilities and instead perform database scan whenever a search is requested. These systems heavily utilize early abandoning techniques in order to provide fast retrievals. In many cases, both limitations are present.

For instance, a DTW method proposed in [Rakthanmanon13] uses the resources of a single machine and requires a full data scan for each search.

Many applications require near real-time search performance against large datasets. Such a system generally performs frequent searches against the database with queries that might have a high degree of similarity. For example, a music improvisation system is designed to interact with the musician in real time, and is expected to receive multiple queries within a short time interval. Therefore, a system capable of indexing large time series data and performing fast retrievals is needed.

### 4.1.1 The Proposed System

In this study, a new system is constructed with the capability of indexing multiple large time series and performing fast searches in the indexed space in order to retrieve both perfect and similar matches. The proposed system utilizes distributed key value data store as a backbone to ensure data scalability. An R*-tree index is built on top of the distributed database in order to facilitate the time series search. R-trees and similar structures including R*-trees have been widely used in past research on time series data analysis [Keogh01A].

The system consists of data segmentation component, R*-tree index structure, HBase distributed key value stores and a caching mechanism. R*-tree is a variant of an R-tree optimized for improved performance by building a better quality tree [Beckmann90]. As

a matter of fact, R*-tree adheres to the same rules as the simple R-tree, and the differences are merely in the implementation. Data segmentation is vital for the construction of multidimensional index for time series data, because it breaks time series data into a set of multidimensional points. Since the focus of this study is to evaluate the R*-tree index in the distributed environment, no other data preprocessing such as dimensionality reduction or normalization is performed. HBase serves as a data storage backbone for the system, as such the performance of the HBase is a key factor for the fast operation of the entire system. Caching schemes of the index in the main memory can potentially improve the search time by reducing the number of costly HBase interactions.

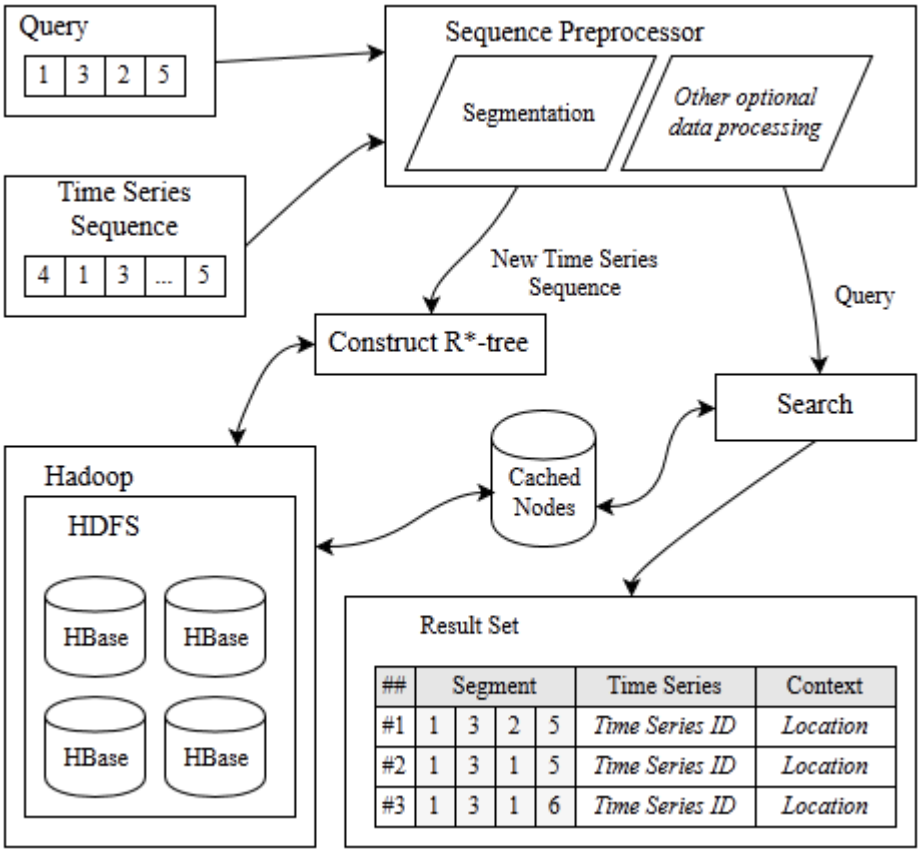



Figure 6. System Diagram

The system diagram is shown in Figure 6. The index construction of a time series starts in a preprocessing phase, during which a time series is segmented into the subsequences of a fixed size. The segments are then treated as index points for constructing the R*-tree index. The resultant index structure is at last written to the HBase storage. Searching for similar subsequences starts at the same preprocessing stage: segmentation. Currently a query length is restricted to the segment length used for index construction. The query is then sent to the search sub-system that seeks similar subsequences in the R*-tree index space. The search interacts with the cache when a tree traversal is needed. If the requested tree nodes are not found in the cache, the search system performs retrievals from the HBase store and updates the cache as needed. At the completion of a search, a set of index points matching the query is returned along with information on where the matches occur in the original time series.

### 4.1.2 Performance Evaluation

Similar index systems in the distributed environments are also utilized for searching spatial data. Comparing with spatial data that contain two-dimensional information, time series data have a much higher dimensionality. Since the performance of the R-trees and its variants is known to degrade in high-dimensionality, the extent of search time deterioration in the key-value data store must be evaluated. In addition to high dimensionality, other properties of time series datasets, such as data distribution and presence of repeated patterns, can have an impact on the performance as well. The proposed system is evaluated with various time series distributions, including uniform

and normal distributions. Since normally distributed data pack most data points relatively close to the mean, it is expected to have negative impact on the performance due to the large number of index node overlaps.

The proposed system is evaluated via the following aspects:

1. Impact of index dimensionality on system performance.
2. Impact of R*-tree node capacity on system performance.
3. Impact of data distribution on system performance.
4. Data scalability of a system in comparison with sequential file scan approach under different index capacities.

### 4.1.3 Search and Retrieval Criteria

A similarity-based search system must be capable of answering at least two types of queries: exact perfect match and exact *k*-NN query. In perfect match, the system is looking for the same subsequence in the database as presented in the query. *k*-NN queries retrieve *k* most similar subsequences ordered by their resemblance to the query.

Similarity between the query *q* and a subsequence *c* is determined by similarity distance function D(*q*, *c*):

D*(q, c)* = a distance measure related to dissimilarity.

Given the choice of R-tree as multidimensional indexing tool, using Euclidean distance as similarity metric is appropriate. Thus D*(q, c)* computes Euclidean distance between query *q* and a potential match *c*. Given the similarity function, perfect match queries can be thought of as searches producing all matches with distance to the query D*(q, c)* = 0.

## 4.2 Background

The proposed system utilizes an R*-tree data structure as its indexing technique. The data structure is built on top of Apache HBase, a distributed key-value database. The rest of this section discusses two crucial components of the proposed system: HBase and R-trees family of indexes

### 4.2.1 HBase

Apache HBase is a NoSQL distributed key-value database built on top of Apache Hadoop and HDFS. HBase is an open source implementation of Google's BigTable [HBase14]. Similar to other key-value databases, HBase can be viewed as mappings between a key and a row with information identified by that key. A collection of rows make up an HBase table and each row in the table has a third dimension generally used to store the previous version of a row. This dimension is often called the time dimension, as it indexes the revision of the record stored in the same row by the time of a record creation. Users have the ability to override the time dimension and use it for purposes not related to revision or history tracking [HBase14].

HBase rows are broken up into column families and a column family consists of columns. Column families and columns allow partial retrieval of a row when not all of its information is required. For instance, an HBase table can have "name" and "address" column families with each column family containing one or more columns. Depending on the situation, a user may choose to retrieve only one of the two column families from the database. Each column family can contain multiple columns, but not every column can be presented in each row. In fact, HBase columns are not specified in advance. Columns are declared at the time data are placed into an HBase table, allowing different rows of the same table to have different columns [HBase14]. Column families on the contrary are defined beforehand and essentially make up the schema of a table. Figure *7* provides an illustration of an HBase table structure.

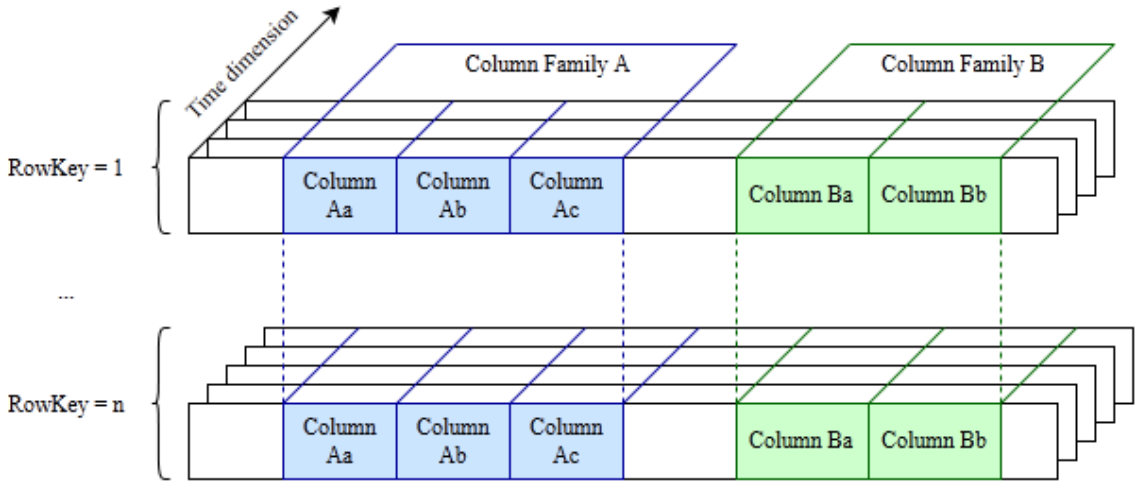



Figure 7. HBase table structure.

Because HBase is a distributed system, it resides on multiple machines and can be scaled up if more computers become available. The system is fault tolerant, meaning that the malfunctions in a certain number of machines will not cause the entire database to go offline. According to Mathur, Google's BigTable can be viewed as a B-tree, where nodes of a tree can be distributed across multiple computers [Mathur11]. Figure *8* shows a simplified structure of the BigTable with omitted time dimension for data and indexes. All the data in such a storage system are located in the leaf nodes of the tree, while other nodes are used to route the request to manipulate the data to the leaf nodes. Since HBase is an open source implementation of BigTable, its structure is similar to the one in Figure *8*.

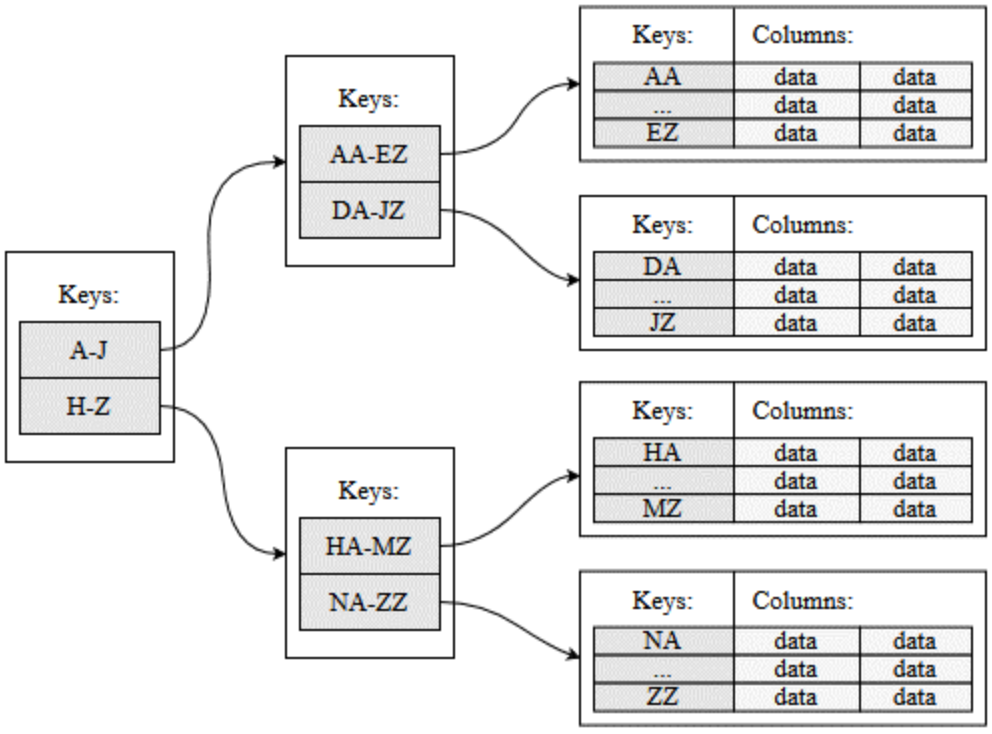



Figure 8. Simplified schematic representation of BigTable with time dimension omitted.

### 4.2.2 R-tree

R-tree is a tree data structure commonly used for storing and indexing of spatial data. It has also been used as a tool for indexing multidimensional data. Since time series can be easily represented as a set of *n*-dimensional data points, R-trees have been extensively used in prior research for time series similarity search. However, such usages almost all resided in a single machine, not in a distributed environment.

The original R-tree was presented by Guttman in [Guttman84] as a tool to index complex spatial objects consisting of multiple *n*-dimensional points. In an R-tree, leaf nodes contain a set of records in the form of:

(*R, data-pointer*),

where *R* is the *n*-dimensional minimum bounding rectangle (MBR) for the object referred by the *data-pointer.* Non-leaf nodes slightly differ from leaf nodes, containing records in the form of:

(*R, child-node-pointer*),

where *R* is the minimum bounding rectangle covering all rectangles in the child node and the *child-node-pointer* is a link to a child node.

As defined by Guttman in the original work [Guttman84], R-tree must satisfy the following properties:

**Property [1].** Every node can contain between *m* and *M* records, where *m* and *M* are respectively minimum and maximum node capacity, unless it is also a root node.

**Property [2].** For each entry in the node, *R* is the smallest rectangle that contains the multidimensional objects or rectangles in the child node.

**Property [3].** The root node must contain at least two children nodes, unless the root is also a leaf node.

**Property [4].** All leaf nodes are on the same level of the tree.

Figure *9* illustrates an R-tree structure and provides a visualization of the index in 2-dimensional space. For instance, it can be observed in (a), rectangle R3 points to the leaf nodes R8 and R9, and in (b), rectangles R8 and R9 are inside the rectangle R3.

R-tree was originally developed to be used in the disk storage, but it has been successfully ported to other environments, such as main memory. Some success with R-trees was achieved in the distributed key-value stores as well. Wei built R-tree index on top of the Apache Cassandra [Wei13]. The trick to using any R-tree variants in distributed database like HBase and Cassandra is in the key choice for the tree nodes, since it is significantly more efficient to retrieve a range of objects with keys located nearby in the key space than retrieve multiple objects one at a time [Wei13]. It is extremely important to have proper assignment of keys to the tree nodes in a way to allow for retrieval of all children nodes in one database scan.

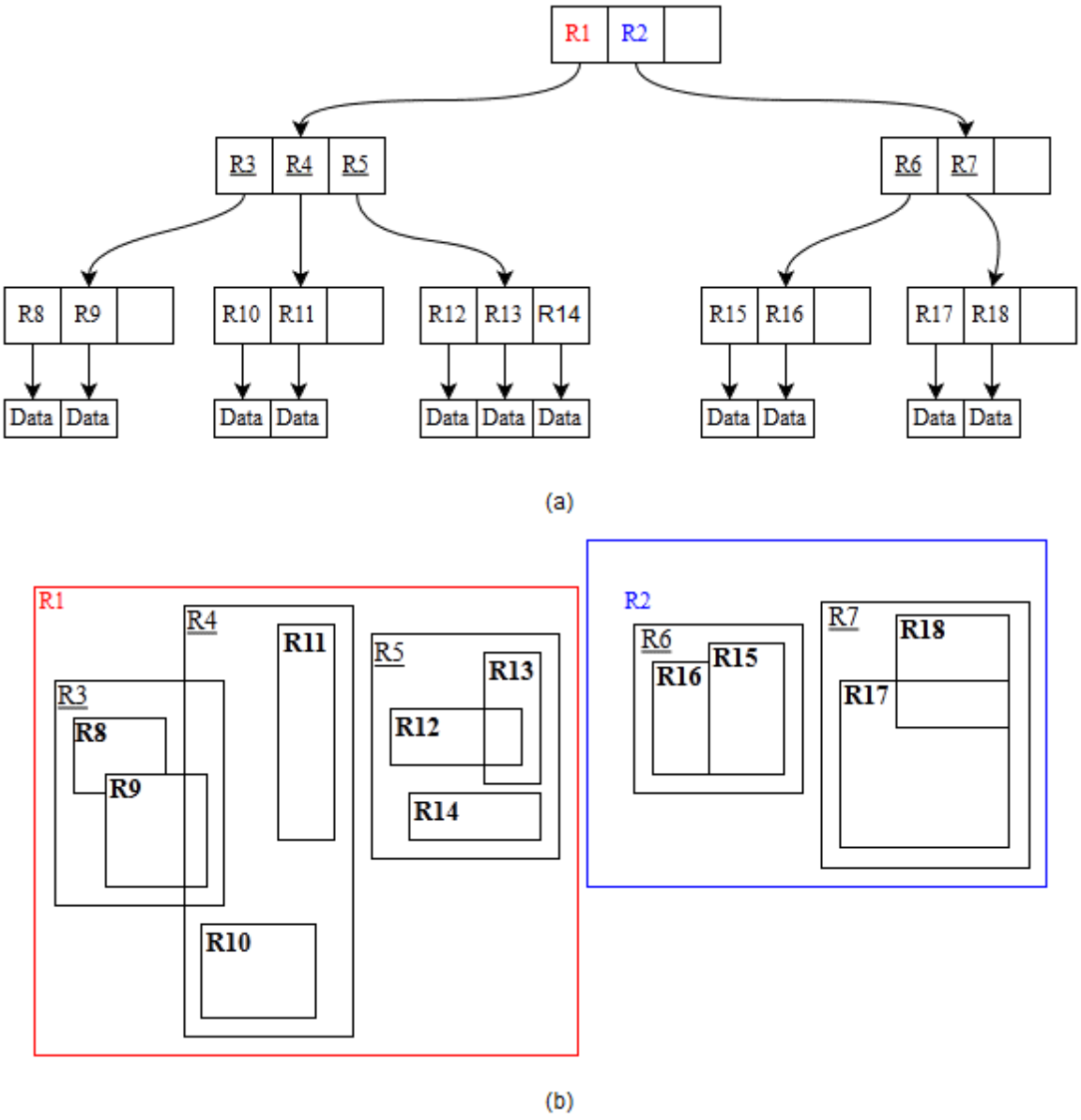



Figure 9. (a) R-tree structure and (b) visualization in the 2-dimensional space.

In addition, reducing the number of traversed tree nodes can improve the performance. One way to accomplish this is by allowing the tree to grow broader to reduce the depth of the tree. The maximum number of items each node can hold was originally predetermined by the size of the disk block in order to minimize the number of disk I/O operations. In distributed key-value stores, the tree is more limited by the network latency

and synchronization costs of the distributed system. Therefore, reducing the number of times the database is accessed by making the tree broader than originally designed for disk storage can improve the performance. Another way to reduce database access is by selecting the correct branch of the tree without having to traverse from the root node each time. Wei proposed an additional index based on the Hilbert space-filling curve to allow querying the rectangles only from the region in which a query point is located [Wei13].

In [Beckman90], a system using R*-tree version of the index is proposed to optimize the node split algorithm in order to produce splits with smaller overlaps. Another implementation change is the addition of node reinsertions into the algorithm. Node reinsertion happens when a child is being added to the parent node already at its maximum capacity. Instead of performing a node split, R*-tree algorithm removes certain children nodes from the parent node and adds these children back to the tree using regular tree insertion algorithm. Such reinsertion allows some children that were generally added earlier in the construction process to find a better parent node to reduce the node overlap. Reinsertions can be done only once on each tree level per data point in order to prevent infinite loops. Because of the reinsertion policy, tree node splits can only happen during reinsertion and never occur when new data points are added.

## 4.3 Time Series Preprocessing

Incoming data can be processed in many different ways before the construction of an index structure. The data reprocessing has a major impact on the system performance in

terms of both efficiency and accuracy. The most common steps to process incoming time series data include segmentation, normalization and dimensionality reduction. Segmentation partitions the time series into manageable subsequences that are later used to construct index structures. Normalization can be applied to the entire series or segments in order to reduce the negative effects of outliers and noise, and it also allows for similarity search on sequences in different scales or with constant up or down shifts [Loh00]. Dimensionality reduction is often used in the field of time series similarity search in order to reduce the index size and improve search performance [Keogh01A]. In the scope of this work, the effects of normalization and dimensionality reduction are not studied.

Unlike dimensionality reduction or normalization, segmentation of time series is a necessary step for the construction of multidimensional index structures such as R-trees. Since the R-tree index is constructed from a set of multidimensional objects, the time series $S = \{s_1, s_2, s_3,...,s_m\}$ of arbitrary length $m$ needs to be broken up into a set of segments of length $n$, where $n$ is the number of dimension in the R-tree index, $n \leq m$. As a result, each segment of the form $C = \{c_1, c_2, c_3,..., c_n\}$ can be seen as an $n$-dimensional point that can be used for constructing the $n$-dimensional indexing structure [Keogh01A].

Segments obtained from the original sequence can overlap with each other. The overlap is usually achieved by the means of a sliding window approach. In order to make sure the R-tree model can find all subsequences for a given query, the segmentation has to be performed with a window sliding by one data point at a time. In contrast, if segmentation

is done without overlaps, each segment can be seen as an independent entity or a point of a multidimensional time series. Figure *10* shows how segmentation overlaps can affect the search results. As shown in (a), segmentation with no overlaps produces less segments. It does not capture all subsequences of the original time series, and thus segment {4,3} is found only once. Segmentation shown in (b) uses overlaps to create segments for all subsequences from the inputs. Thus segment {4, 3} can be found twice, just as subsequence {4, 3} is present twice in the input.

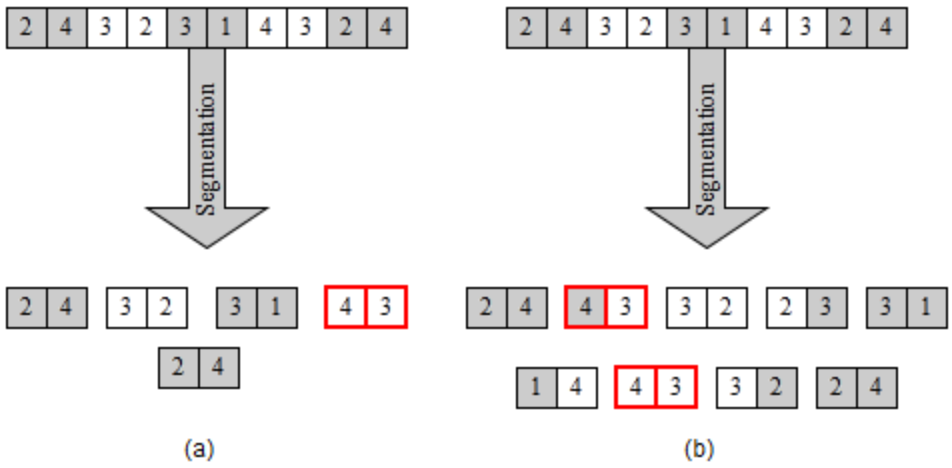



Figure 10. Segmentation overlaps. (a) Segmentation with no overlaps. Only one segment {4,3} is found despite the fact that sequence {4,3} occurs twice. (b) Segmentation with overlap. Two segments {4,3} are found.

## 4.4 Index Construction

The indexing structure is constructed from the input data previously segmented into a set of subsequences. Depending on the application, subsequences can have various degrees

of overlaps. Once the input sequence is segmented, an R-tree index is constructed and saved to the database. The rest of this section discusses the construction of a basic R-tree and the manner in which the R-tree is saved to an HBase distributed data store.

### 4.4.1 Basic R-tree Index

The proposed indexing system uses R*-tree variant of the basic R-tree multidimensional index structure. Both versions of the tree are very similar. Both adhere to the same rules outlined in [Guttman84] and the properties described in the previous section. R*-tree mainly differs from the basic R-tree in the implementation. Understanding the index construction of an R-tree is important for the comprehension of the changes introduced by R*-tree optimizations.

The proposed indexing system implements R*-tree variant of the indexing structure, but the generic structure of the model remains identical to the R-tree. Non-leaf (or general) nodes group children nodes located close to each other. Unlike the generic R-tree and R*-tree models, the leaf nodes in the proposed system do not store pointers to the data items. Instead, the pointers to data nodes are contained in the leaf nodes. Data nodes are capable of storing multiple data pointers. This design allows the system to use the same node to index multiple identical segments so that the size of the tree can be reduced to improve overall performance. Both general and leaf nodes of the index structure must store a few key pieces of information needed for the index construction and operation: a list of pointers to child nodes, the position of the MBR for a node and the dimension of the

MBR. Data nodes contain the indexed point and a list of references to the actual data represented by the node.

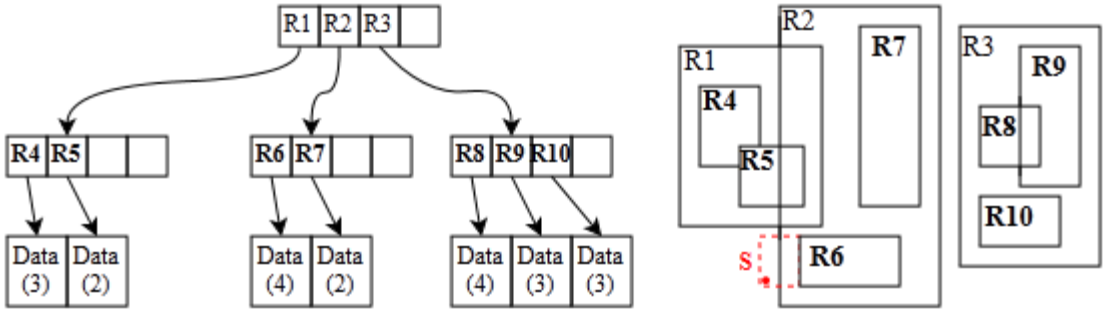
Both leaf and general nodes can be seen as records in the form of:

*(MBR-location, MBR-dimension, List of child-node-pointers),*

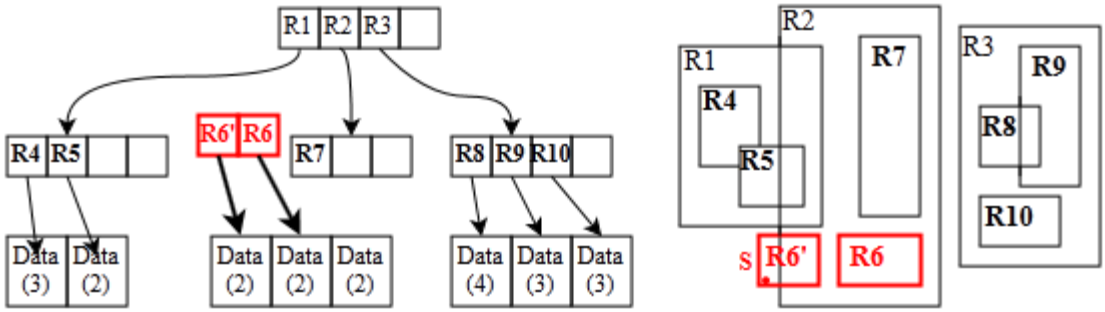
where *MBR-location and MBR-dimension* are the position and dimension of the minimum bounding rectangle for a node, and *child-node-pointer* is pointer to a child of a node.

The basic R-tree model is constructed one data segment at a time. Each segment is treated as a multidimensional point. Generally, the addition of a new segment to an index consists of finding the most suitable leaf node for storing the segment and then adding the segment to the leaf node. In the case where a segment addition causes an MBR of the leaf to change, the change is propagated upwards to the parent node and eventually reaches the root node if needed. Sometimes the most suitable node for a segment can be at its maximum capacity. In this case, the node is split in two nodes and the split nodes replace the old one at the parent level. The split propagates upwards by dividing any parent nodes along the way as needed. Figure 11 illustrates the process of adding a segment S to an existing R-tree. In Figure 11 (a) the existing model is searched for a leaf node to hold newly added segment S. Node R6 is such a node in the example, because it requires the smallest MBR increase to contain S. Node R6 is at the maximum capacity and cannot

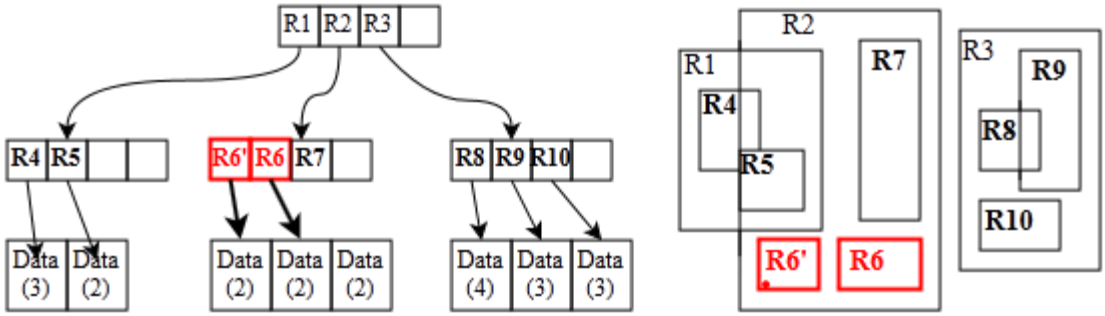
hold segment S. A node split is performed in Figure 11 (b) producing nodes R6 and R6'. The node split is propagated to R2, a parent node of R6 in Figure 11 (c). MBR of R2 is increased to cover R6 and R6'. At this time the propagation stops, since the root node is reached and no root split is needed.




(a) Left: R-tree with node capacity of 4. Number of data nodes each leaf has is shown in parentheses. Right: Data segment S is added. Dashed line shows how much R6 needs to extend to contain S.




(b) Left: Leaf R6 already contains its mixumum number of children, so it splits into R6 and R6'. Right: Shows R6 and R6' after split. R6' is still not contained by R2.




(c) Left: R6 split into R6 and R6' and the split is propogated to the parent node R2. Right: MBR of node R2 is extended to cover R6 and R6'.

Figure 11. Adding a new segment to an R-tree.

The proposed system takes advantage of a few R*-tree optimization techniques aimed at improving the quality of the tree by minimizing the overlap between the nodes on the same level. One of the biggest changes introduced by an R*-tree is a better node split algorithm. The improved node split algorithm that minimizes the overlap between the rectangles leads to a smaller number of visited nodes when performing the search. Another optimization aims to improve the quality of the tree by deleting and reinserting certain nodes back to the tree. The optimization triggers when the node overfills and needs to be split. A predefined percentage *p* of nodes is removed from the overfilled node and reinserted to the tree again. This node reinsertion allows many of the nodes to be placed in a better spot within the index structure, which reduces the overlap and eventually improves the search performance [Beckmann90]. The process allows only one reinsertion per tree level for each added segment, eliminating the possibility of entering an infinite loop of node reinsertions. Node reinsertion occurs only when a node reaches its maximum capacity. After the reinsertion completes, a node that triggered the reinsertion is no longer full, because a portion of its children has been removed and placed into different nodes of a tree, thus the node starting the reinsertion procedure is not split. However, node splits happen when a node reaches its capacity and reinsertion procedure cannot be invoked because the procedure has been used prior on the same tree level. The nodes for reinsertion are picked from an overfilled node by calculating the distances of child nodes from the center of an overfilled node and selecting certain percentage *p* of nodes with highest distances from the center. In the original R*-tree research, the authors claimed that $p = 30\%$ gives the best improvement in search performance [Beckmann90].

The algorithms used for index construction are presented below. More information on building R-trees can be found in [Guttman84] and [Beckmann90]. Figure 12 lists an algorithm used as an entry point for adding a new index point. The algorithm resets any node reinsertion restriction that could have been set by the previous data insertion (lines 1-3). On lines 4 to 7, it is decided whether there is a necessity to proceed and to create a new index point or simply to add new data pointer to the list of pointers in the matching index node, in case the index for the data point already exists. On line 8, *ADD_TO_LEAF* algorithm is invoked and the data-node S is added to the index. If needed, the tree is grown on lines 9 to 12.

```
Algorithm INSERT (inserts a data segment to the R-tree)
Input: R-tree root R, Data-node S,
     Maximum number of children per node MAX,
     Minimum number of children per node MIN
Output: updated R-tree
Method:
1.   if not reinserting node then
2.      reset reinsert restriction for all levels
3.   end if
     //check if index for S already exists
4.   matchNode = Node M such that D(M, S) = 0
5.   if matchNode ≠ empty then
6.      add pointer to data of S to list of data pointers of
     matchNode
7.   else
8.      splitNodes = ADD_TO_LEAF(R, R, S, 1, MAX, MIN)
9.      if |splitNodes| == 2 then
           //root node has split, grow the tree.
10.        R = new Root Node
11.        make splitNodes be children of R
12.     end if
13.  end if
14.  return R //return back the root of modified R-tree
```

Figure 12. R*-tree node insertion algorithm.

Figure 13 shows the *ADD_TO_LEAF* algorithm. This algorithm is responsible for

traversing the tree to the most appropriate leaf node for a data node being inserted. Lines 2 to 9 are recursive base case of the algorithm and are responsible for adding new data point to the leaf node. Line 8 invokes the OVERFLOW_TREATMENT procedure when the leaf node reaches its maximum capacity. The recursive section of the algorithm is on lines 10 through 26. The next traversal step is computed on line 11 by finding the children that require the smallest rectangle increase to accommodate the new node. A recursive call is made on line 12, while lines 13 to 25 update the traversed nodes as the algorithm unwraps the recursion calls. Similar to reaching the capacity at the leaf level, non-leaf nodes can be subjected to overflow. Therefore, the overflow treatment is invoked for non-leaf nodes on line 23.

```
Algorithm ADD_TO_LEAF (inserts a data segment to leaf node)
Input: R-tree root R, R-tree node N, Data-node S, Current tree level L,
    Maximum number of children per node MAX,
    Minimum number of children per node MIN
Output: list of modified nodes T
Method:
 1.   T = {Ø}
 2.   if N is leaf node then
         //we reached leaf node, so add data node for segment S
 3.      create new data node D for segment S
 4.      add S to children of N
 5.      if |N.children| <= MAX then
 6.         add N to T
 7.      else
            //adding a child caused node to overfill
 8.         T = OVERFLOW_TREATMENT(N, L, R, MAX, MIN)
 9.      end if
 10.  else
         //Non leaf node, so go down to the leaf
 11.     C = child of N requiring smallest MBR increase to fit S
 12.     splitNodes = ADD_TO_LEAF(R, C, S, L + 1, MAX, MIN)
 13.     if |splitNodes| == 1 then
            //no nodes have split
 14.        recompute dimensions and position of MBR of N
 15.        add N to T
 16.     else
            //deal with consequences of node split
 17.        delete C from children of N
 18.        add each node of splitNodes to children of N
 19.        if |N.children| <= MAX then
               //split stops here
 20.           recompute dimensions and position of MBR of N
 21.           add N to T
 22.        else
               //handle overflow of N
 23.           T = OVERFLOW_TREATMENT(N, L, R, MAX, MIN)
 24.        end if
 25.     end if
 26.  end if
 27.  return T
```

Figure 13. ADD_TO_LEAF algorithm recursively reaches leaf node and adds data to it.

When an R-tree node reaches its maximum capacity, an OVERFLOW_TREATMENT routine is invoked. During the overflow treatment, a subset of children of a full node can be removed from the model and resinserted again. This node reinsertion can only be done once per tree level for each new data node. Lines 2 to 9 of the

OVERFLOW_TREATMENT algorithm shown in Figure 14 enforces reinsertion rules and select the best candidates for reinsertion. Line 4 sorts children of the full node by the distance from the node's center, then a certain percentage *P* of the children furthest from the full node's center is removed and reinserted back into the indexing model. The reinsertion identifies a better place for nodes that have been added early in the model construction and may no longer fit well with their current parent [Beckman90].

```
Algorithm OVERFLOW_TREATMENT (handles overflow in a node)
Input: Overflowing node N, Node level in R-tree L, Root Node R
     Reinsertion percentage P,
     Maximum number of children per node MAX,
     Minimum number of children per node MIN
Output: list of modified nodes
Method:
 1.   List of nodes for reinsertion I = {Ø}
 2.   if L > 0 AND can reinsert on level L then
 3.      restrict reinsertion on level L
 4.      sort children of N by distance from center of N
 5.      add MAX * (P / 100) nodes to I
 6.      remove I from children of N
 7.      REINSERT (R, I, MAX, MIN)
 8.      return empty
 9.   end if
 10.  return SPLIT(MAX, MIN, N)
```

Figure 14. OVERFLOW_TREATMENT algorithm performs reinsertions or node splits.

Node reinsertion is accomplished in a recursive manner. Since only data nodes can be inserted into the tree, when a REINSERT procedure, shown in Figure 15, is called on a non-data node, the algorithm recursively reaches all data nodes in the current R*-tree branch and reinserts such nodes back to the model. Lines 1 to 3 show the recursive base case where a data node is inserted back to the index. Lines 3 to 7 represent the recursive case: REINSERT procedure is invoked for each child of a node *N*.

```
Algorithm REINSERT (reinserts a data node to the R-tree)
Input: R-tree root R, Node N,
     Maximum number of children per node MAX,
     Minimum number of children per node MIN
Output: modified R-tree with Segment node S
Method:
 1.   if N is data node then
 2.      INSERT(R, N, MAX, MIN)
 3.   else
         //not a data node, so traverse to closer to data node
 4.      for each child in children of N
 5.         REINSERT(R, child, MAX, MIN)
 6.      end for
 7.   end if
 8.   return R //return back the root of modified R-tree
```

Figure 15. Algorithm REINSERT. Reinserts some of the nodes back to R-tree.

Figure 16 presents a basic overview of an algorithm used for splitting two nodes. In order to carry out a node split, we first choose the axis or dimension along which to perform the split. Generally, a metric is used to pick the best axis to ensure that the split along the chosen axis has the least amount of overlap. Two groups of nodes are chosen by separating the nodes along the axis of split on line 2. The groups are selected to have minimum overlap among all other groups along the split axis. Lines 3 to 6 finalize the split and return two resultant tree nodes that can be added to the model.

```
Algorithm SPLIT (splits node in two)
Input: Node N,
     Maximum number of children per node MAX,
     Minimum number of children per node MIN
Output: list of two resultant nodes with all children of N split
Method:
1.   choose axis along which the node is to going to be split.
2.   assign children of N to two groups such that the overlap
      between two groups is minimal along the chosen axis, while
      ensuring each group has between MIN and MAX nodes.
3.   create empty nodes S1 and S2 of the same node type as N
4.   assign first group to be children of S1
5.   assign second group to be children of S2
6.   return {S1, S2}
```

Figure 16. Algorithm SPLIT breaks a full node into two smaller nodes.

### 4.4.2 R-tree in HBase

Storing the R-tree model in a distributed key value data-store such as HBase is in fact very similar to disk storage. Instead of storing nodes in the disk pages, tree nodes are saved into the HBase rows, and each row can be found by a unique identifier. Similar to disk storage, the index in HBase must be grouped together for faster retrieval. The proposed system groups records by their identification numbers to have all children of a node occupy a block of consecutive identification numbers. This allows faster retrieval of child nodes using HBase scan operation. However, unlike the disk storage, each node stores the pointer to the children and not the children MBRs and locations. This allows us to retrieve all children at once and then perform in memory processing to decide which children nodes are expanded next. This approach is intended to reduce the number of

HBase interactions. In addition, performing a scan operation in order to get a range of records is far more efficient than returning the same number of records one record at a time [Wei13].

Unique identifiers for HBase rows are reserved in advance for each tree level. Since the maximum number of nodes at each level can be calculated in advance, knowing the maximum node capacity allows each level of the tree to have its HBase identifiers reserved in advance. Even when the level is not full, it is guaranteed to have enough identifier space available to reach the maximum level capacity while still preserving the continuous nature of the identifiers.

## 4.5 Search and Retrieval

The main purpose of building an indexing structure like R-tree is to support multidimensional data search and retrieve. In this study, such data are segments from a time series sequence. Generally, there are few kinds of queries a search system can answer: perfect match and *k*-NN. In this work, *k*-NN searches are discussed the most, since a perfect match can be seen as a special case of the *k*-NN search where all results have the same similarity to the query. The searches are formally defined as follows.

**Definition [1].** Exact perfect match search is a search returning all subsequences of the original input with similarity distance $D(q, c) = 0$ between a query $q$ and a potential match $c$.

**Definition [2]**. Exact *k*-NN search is a search returning *k* closest segments to the query *q* as defined by the similarities distance function *D* in ascending order starting from the match *c* with the smallest distance *D(q, c)*. The results set must contain *k* closest matches without any omissions.

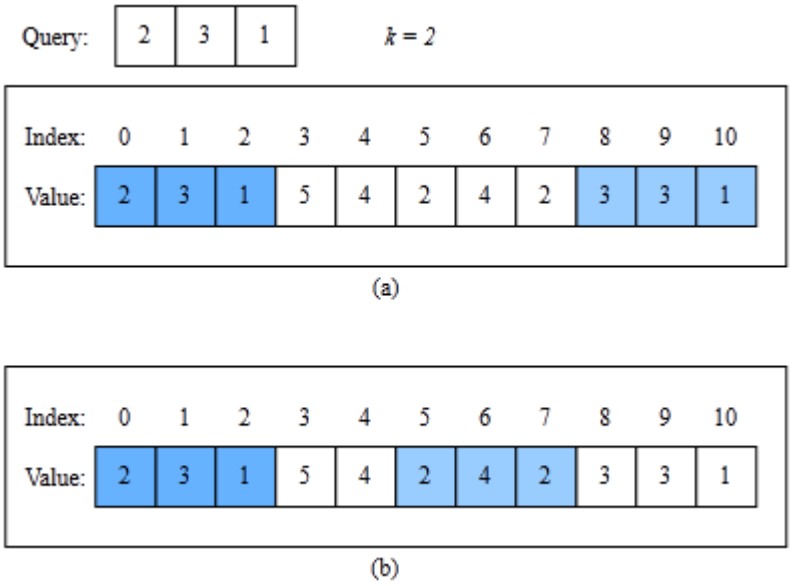



Figure 17. Exact and approximate *k*-NN examples. (a) Exact *k*-NN with $k = 2$. (b) Approximate *k*-NN with $k = 2$

Exact perfect match and *k*-NN searches should not be confused with an approximate search. In the approximate perfect search, the result set might not contain all matching subsequences of the original time series. Approximate *k*-NN search can omit some of the relevant results. Figure 17 illustrates the difference between exact and approximate search results. In Figure *17* (a), the exact *k*-NN search is performed against a dataset with *k=2*. Two closest matches {2, 3, 1} and {3, 3, 1} are retrieved. Figure *17* (b) illustrates

what an approximate *k*-NN search can return. As can be seen, segment {2, 4, 2} is returned instead of {3, 3, 1} even though the distance of segment {3, 3, 1} to the query is smaller than the distance from {2, 4, 2}. The proposed system focuses on exact searches, making the results in Figure *17* (b) unacceptable.

```
Algorithm KNNSEARCH (performs a k-NN search)
Input: R-tree root R, number of results to return k, Query Q;
Output: a list of closest index points I;
Method:
    //initialize
1.  MinPriorityQueue PQ = empty
2.  List candidates = empty
3.  List results = empty
4.  PQ.add(R, 0) //adding root to queue with priority 0
5.  while PQ not EMPTY
6.    top = PQ.pop
7.    foreach subsequence cs in candidates
8.      if D(Q, cs) <= MINDIST(Q, top) then
9.        results.add(cs)     //adding item to the result list
10.       candidates.remove(cs) //removing item from candidate list
11.       if |results| == k then return results //we are done here
12.     end if
13.   end foreach
14.   if top is DATANODE then candidates.add(top)
15.   else if top is LEAFNODE then
16.     foreach child data node cdn in top
            //add child to queue with priority of D(Q, cdn)
17.       PQ.add(cdn, D(Q, cdb))
18.     end foreach
19.   else // top is general, non-leaf node
20.     foreach child node cn in top
            //add child to queue with priority of MINDIST(cn, Q)
            //where MINDIST is minimal distance from Q to MBB of cn
21.         PQ.add(cn, MINDIST(cn, Q))
22.     end foreach
23.   end if
24. end while
25. return results //return results in case we never got k matches
```

Figure 18. Exact *k*-NN search algorithm.

In order to perform searches in the indexed space, a slightly modified version of the algorithm in [Keogh01A] is used. The modified algorithm is presented in Figure 18. The algorithm operates by always starting from the root of the tree. The root node is added to the minimal priority queue in line 4. The algorithm iterates as long as there are items left in the priority queue. In each iteration of the main loop, the top node of the queue is either expanded with all the children added to the queue in lines 15 to 23, or added to the list of potential candidates on line 14, in case the top node is a data node. Each iteration of the main loop also causes the list of candidates to be checked in lines 7 to 13. If the distance from the query to the candidate is smaller or equal to the distance from the query to the top node of the queue, then the candidate is added to the list of results.

The condition in line 7 is a very crucial part of the algorithm. The algorithm performs a search satisfying the exact *k*-NN search criteria only if an indexing structure can guarantee that all children of the *top* node of the priority queue have greater or equal distance from the query than that the distance to the *top* node from the query. Since no dimensionality reduction is used in the proposed system, all the children of the *top* node are at least as far away from the query as the node itself. *MINDIST(node, query)* function is very similar to the similarity distance function D(*q*, *c*) and it computes the minimal Euclidean distance from the axis aligned MBR of a node to the query. Since non-data nodes contain multiple points in the multidimensional space, it is no longer possible to use function *D(q,c)* to compute the distance between a query and a node, because D(*q*, *c*) only computes Euclidean distance between two points.

Once *k* matches are found, the algorithm returns the list of results in line 11. If no *k* results are reached, then the index structure does not contain *k* indexed points and the algorithm returns with the full list of indexed points ordered by their similarity to the query in line 25.

## 4.6 R-tree Node Cache

The proposed system employs a simple caching mechanism to minimize the number of interactions in HBase data store to improve system performance. The cache operates by storing several number of cache pages in the main memory. As the cache gets full, the least recently used pages are removed from the cache to free up space for the other pages. Each cache page represents a list of children of some other node. When requested to retrieve the children nodes, the system performs a cache look-up and tries to find the page with children nodes. If such a page does not exist in the cache, an HBase scan operation is performed to retrieve the list of children nodes. The retrieved list is then placed into a cache page and its usage time is marked.

When the maximum number of pages is reached, cache pages not used recently are removed. The cache system protects certain cache pages from ever being deleted. Generally, protected cache tables store tree nodes located close to the root of the tree. These cache pages have a high chance of being used on subsequent queries, thus their removal needs to be avoided. Cache pages farther away from the root are not protected from deletion Cache pages with nodes farther from the root have higher chance of being

removed from cache, because these cache tables represent specific branches of the traversed tree and have smaller chance of being used in subsequent searches.

Cache size is one of the most prominent and important parameters as it regulates the maximum number of nodes cached by the system. Higher capacity allows for more indexes to be stored in the memory, which reduces the need to request data from an HBase and further improves the overall performance. Unfortunately, high node capacity might not always be beneficial as it will boost the performance only when being utilized close to the maximum capacity. Working with small datasets or doing small number of searches might not fill the cache to its capacity; therefore, there are no benefits when compared to smaller caches.

Chapter 5

# EXPERIMENTS AND RESULTS

## 5.1 Testbed

All the experiments were carried out on the Hadoop cluster of 7 Dell Optiplex 755 machines with dual core Intel CPUs, and 2GB of RAM. The computers were running under Linux CentOS 6.4 operating system. Apache Hadoop and HBase systems were running under Java version 7. A client application was running on the gateway computer connected to the same local network as the cluster. Figure *19* illustrates the schematic layout of the Hadoop environment used as the testbed.

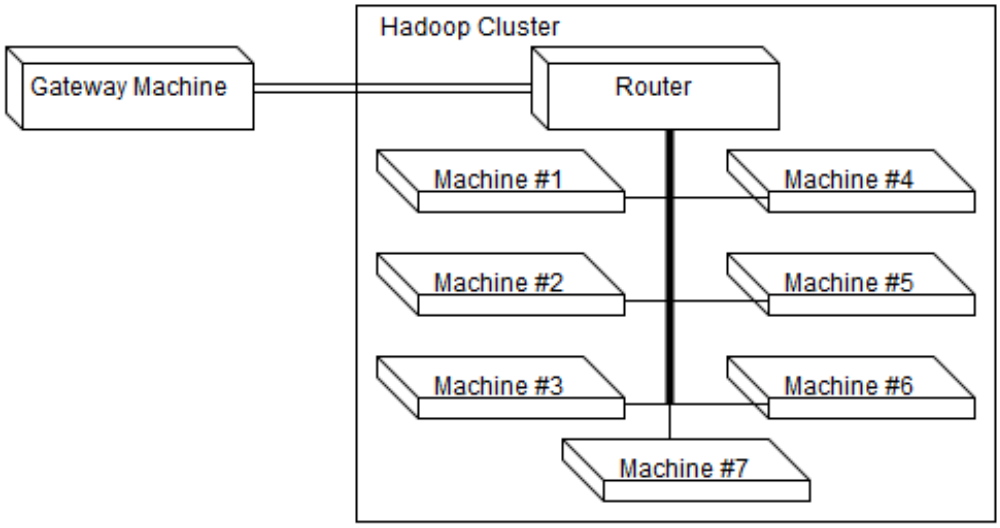



Figure 19. Testing environment.

## 5.2 Impact of R-tree Dimensionality on Performance

R-tree based indexes are susceptible to significant performance degradation as the number of dimensions in the tree grows [Keogh01A]. In this experiment, the impact of increasing dimensionality on the R*-tree stored in the distributed key-value database is tested.

A uniformly distributed time series of 5 million integers was used for this experiment. The dataset was preprocessed into the overlapping segments of controlled length. The segments were overlapped in such a way to guarantee exact search. For example, for the segment size of $n$ items, the overlap is $n-1$ elements, thus each consecutive segment contains $n$-1 last data points of the previous segment. Six models were constructed with dimensions ranging from $n$=4 to $n$=9. Other parameters of the tree, such as minimum and maximum node capacities and reinsertion percentage, remained constant. A set of 100 random queries was generated for each dimensionality tested. Due to the restriction of query size being equal to the dimensionality of the tree, queries with different sizes must be created for R-trees with different dimensionality. However, queries were constructed in such a way to ensure common properties for all dimensions: each of the generated queries used for testing was guaranteed to be found once in the dataset. It is also worth noting that despite the different dimensionality of the tested models, each indexing

structure had the same number of data nodes stored. *k*-NN exact search with *k=5* was performed for each query to measure the execution time and the number of HBase communication of the six models. All caching mechanisms were disabled for this test.

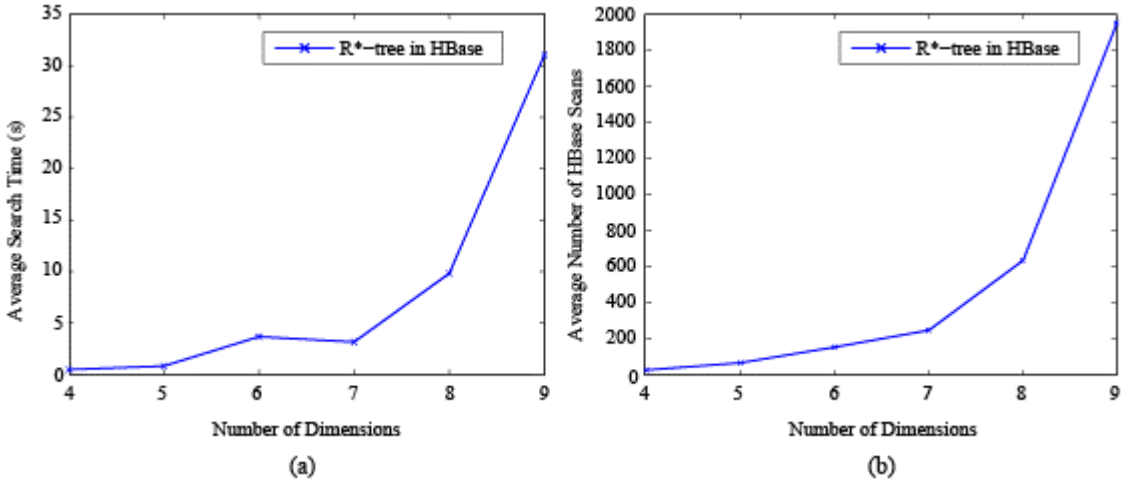



Figure 20. The impact of index dimensionality on R*-tree performance in the distributed key-value data store: (a) the average search time for different index dimensionalities, and (b) the average number of HBase scans for different index dimensionalities.

Figure *20* shows the average search time and average number of HBase interactions for each tested model. It can be observed that the performance degrades very quickly as the number of dimensions increases as shown in Figure *20* (a). Figure *20* (b) also shows that the number of HBase scans grows quickly as the dimensionality increases. Because all the models index the same number of data-points with the same number of data nodes used, the increasing number of HBase interactions suggests higher overlap of the multidimensional rectangles when the number of dimensions increases. Higher overlap

means that a potential match can be located in a larger number of rectangles, making the algorithm examine larger index space. The result of this experiment explains the main reason why R-trees and derivatives have not been used as often for data with higher dimensionality. Since time series indexing requires processing large segments, R-tree based structures often employ certain dimensionality reduction techniques to stay within a more optimal performance range for the index. The average search time at $n$ = 7 slightly improves compared to the dimensionality of $n$=6, despite the increase in the average number of HBase scans. Such behavior can be attributed to a number of factors, such as tasks running in the background in the operating system, Hadoop and HBase, and/or activities from other users accessing the shared cluster used for testing.

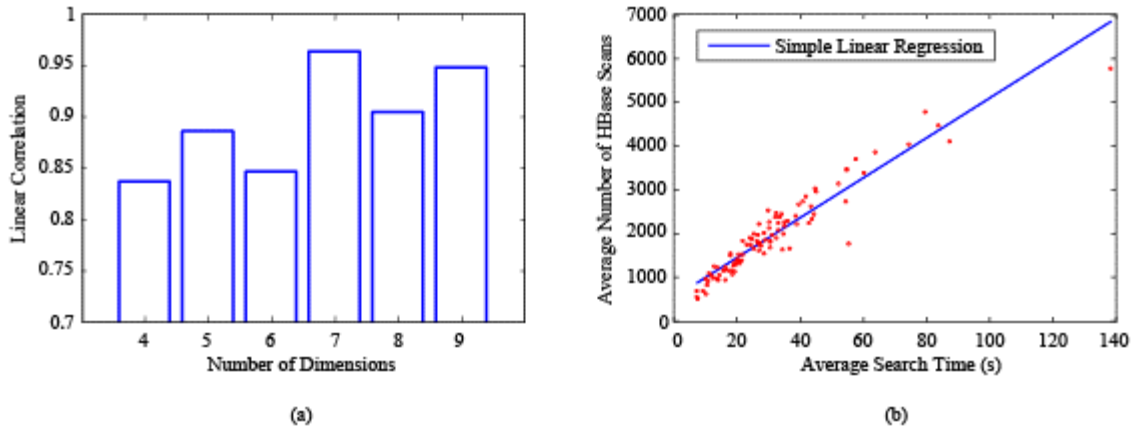



Figure 21. Linear correlation between execution time and the number of HBase scans: (a) linear correlation at different number of dimensions in the index and (b) simple linear regression between average search time and average number of HBase scans at number of dimensions $n$=9

Figure *21* (a) shows the linear correlation between the average time and the number of database interactions needed to perform the query. Strong correlation, reaching above 0.9 in some cases, suggests that the execution time is highly and positively related to the number of database interactions. Such a relation is expected because each database interaction takes a significant amount of time to complete. It is important to mention that the execution time can be impacted by other parameters as well. Figure *21* (b) plots all data points for queries executed on the model with dimensionality *n=9.* Strong linear dependence can be observed between the search time and the number of HBase scan operations.

## 5.3 Impact of Node Capacity on Performance

The capacity of a node in the R-tree based index can play a major role in the system performance. Traditionally, the capacity was regulated by the disk page size so that the index node can only occupy one page to reduce the disk read and write overheads. In contrast, increasing the node capacity in the distributed key-value stores can improve retrieval efficiency. Nodes with higher capacity enable retrieving more index points at once, thus shortening the time it takes for the search task to complete by reducing the number of database interactions.

In this experiment, the same uniformly distributed five million data point integer time series was used. The time series was preprocessed into segments of size *n=5* with an overlap *o=4.* Individual models were built for each of the tested maximum node

capacities. The minimum node capacity was set to always be a half of the maximum capacity to ensure similar conditions for node splits. Index dimensionality at *n=5* and reinsertion percentage *p=30%* were kept constant for all tested models in this experiment. 100 random queries were generated for the evaluation, and the same set of queries was used for each node capacity. Similar to the previous experiment, all caching was disabled and execution time and number of database interactions were recorded.

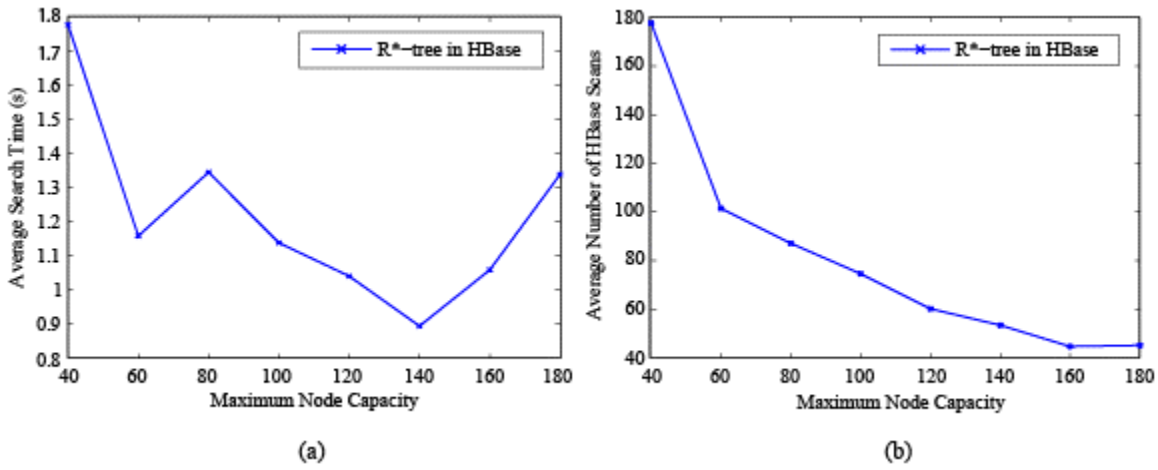



Figure 22. The impact of node capacity on system performance measured by (a) the average search time and (b) the average number of HBase scans.

The average search time and number of HBase requests are shown in Figure *22*. The performance improves as the node capacity increases until it reaches 140 children per node as shown in Figure *22* (a). A noticeable slowdown can be observed at 160 children per node. The initial performance improvement can be explained by the dramatic reduction in the number of database interaction as the node capacity increases. This is because higher capacity nodes create a broader tree, which generally reduces the number

of nodes that must be traversed down to reach data nodes. As the node capacity continues to increase, the tree becomes too broad and each node contains many children that are too far away from each other. As a result, each HBase scan retrieves large volumes of information not needed for the search, even for a fairly small number of database requests. The large unnecessary volume of information increases network loads and computation costs, and ultimately degrades the overall performance of an index.

## 5.4 Impact of Dataset Properties on Performance

In this section, three datasets with different distributions were tested to learn about how the nature of the data impact on the performance of a system. A time series similarity search system can be exposed to datasets with different properties, and the system performance can vary due to the different nature of the dataset. In this experiment, three data sets (each consisting of five million data points) were generated to represent three different properties: uniform distribution, normal distribution and simple symmetric random walk time series. Random walk time series have been used to approximate certain types of financial time series [Rakthanmanon13]. For each dataset, the indexes were constructed with same node capacity *max=140* and *min=70*, reinsertion percentage *p=30%* and index dimensionality *n=5.* All caching mechanisms were disabled during this test.

As can be seen in the Figure *23*, dataset properties can play a very dramatic role in the performance of an R-tree based index system in the distributed key-value environments.

The performance of the system is almost an order of magnitude slower for normally distributed time series than uniformly distributed data. Such system's behavior is likely due to the fact that most of normally distributed data tend to reside in the same section of the multidimensional indexed space, thus more overlaps occur closer to the center of an index space. This hypothesis of having high overlaps is confirmed by the large number of database interactions. The large number of interactions tends to happen when multiple neighboring nodes are examined, suggesting an overlap between such neighbors.

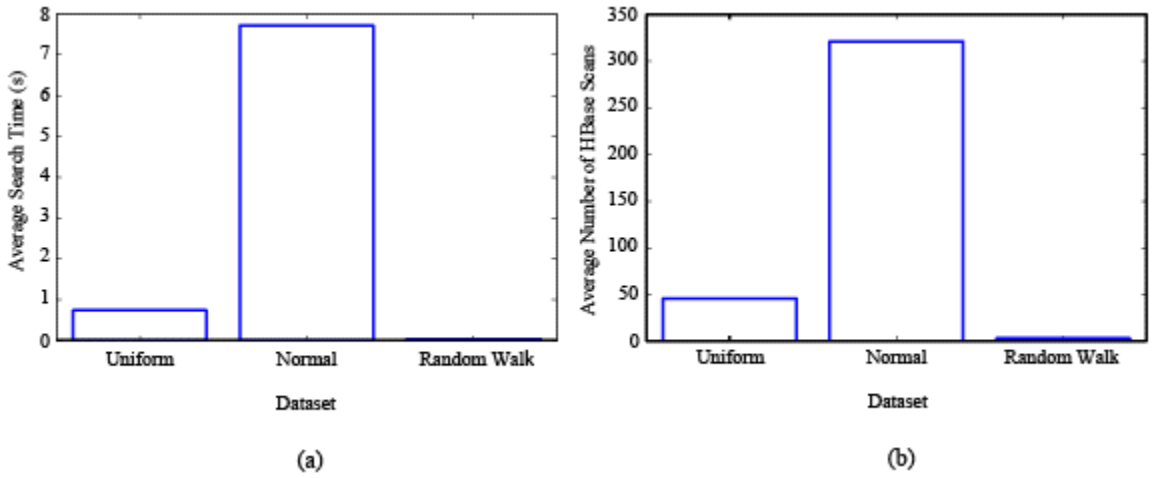



Figure 23. Performance difference in (a) average search time and (b) average number of HBase requests for datasets generated using uniform distribution, normal distribution, and symmetric random walk.

However, the system demonstrated outstanding performance on random walk dataset. Such result can be explained by the presence of high number of repeated subsequences in the simple symmetric random walk data. For example, there were only 72,995 unique

segments of length *n=5* in the tested sample of five million data points. This small number of unique segments results in a construction of a very small index tree that consequently is able to provide very fast search times.

## 5.5 Impact of Dataset Size on Performance

In this experiment various data sizes have been tested against the R-tree based indexing system. For a comparison reason, the same dataset was tested against a simple sequential file scan algorithm performing *k*-NN search on a time series under the Euclidean distance. Several various cache capacities for the HBase R-tree index were evaluated as well. For the experiment, a uniform time series of 30 million integer data points was used. The indexing models were constructed in five million increments, starting with a model of just five million numbers. All models were constructed with the same node capacities *max=140* and *min=70*, reinsertion percentage *p=30%* and dimensionality *n=5*.

A set of 100 test queries was generated for each test size and the same set was used for all tests on a given time series size. Each query was guaranteed to have at least one perfect match in the dataset. In addition, perfect matches were uniformly distributed across the five million blocks of data points. For example, in a ten million dataset, half of the perfect matches to the queries are found in the first block of five million numbers while the other half of the quires has perfect matches in the second five million block of data points.

The cache can play significant role in the performance of a system, as it allows a big reduction in the number of database look-ups. Multiple cache parameters can be configured and tested, but the cache size is by far one of the most important ones. In this experiment, three different cache size were used: small with just 1000 pages, moderate cache with maximum capacity of 5000 pages and big cache capable of holding up to 10000 cache pages. All other cache parameters were kept at the default values.

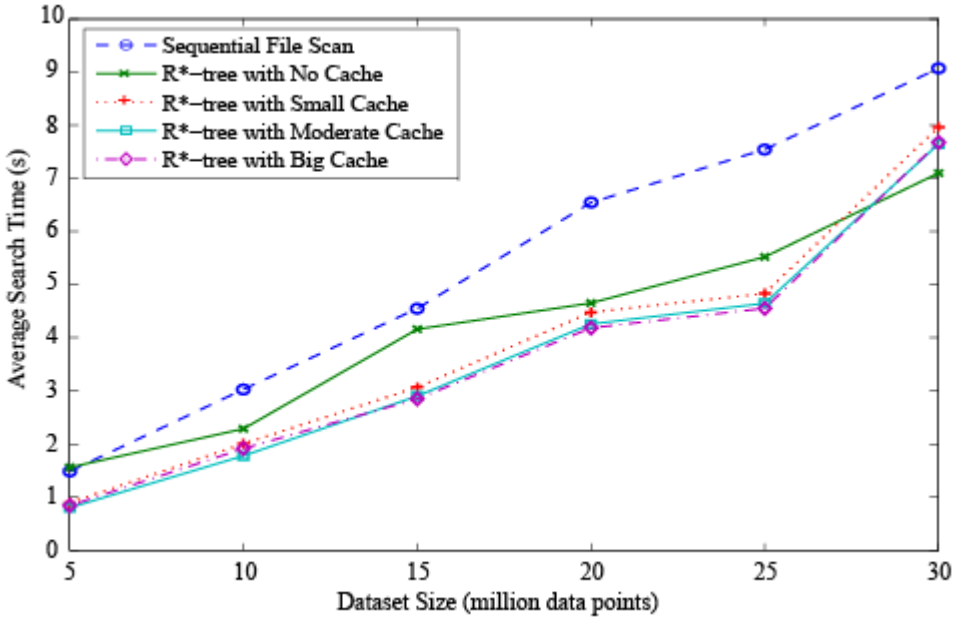



Figure 24. Performance in average search time with respect to dataset size.

The average search time of the system with various cache schemes and of the sequential file scan is shown in the Figure *24*. As can be seen, the R-tree index in HBase generally performs better than the sequential *k*-NN algorithm. As the data size increases, the R-tree based model outperforms the sequential file scan. At 30 million data points cached

models show poor search performance compared to the non-cached version, yet the number of HBase scans decreased due to the cache as can be observed on the Figure *25*. Such performance fluctuation can be attributed to a combination of reasons. The system used for testing is a shared cluster with other users being able to access the cluster. In addition a handful of jobs are performed on the background by Hadoop, HBase and possibly even operating system. These background activities can cause some variation in the network latency and HBase requests queue performance. It is also important to consider the overhead of managing cache, which could have contributed to the performance artifact observed. An overall trend for performance change is observed to follow a linear pattern. Such linear scalability can be explained by the nature of tree construction process for such large time series. As the dataset size increases, the model starts to be built by parts, where each part is independent of each other. Such process causes high overlap between branches created from different parts resulting in the need to examine all such independent branches.

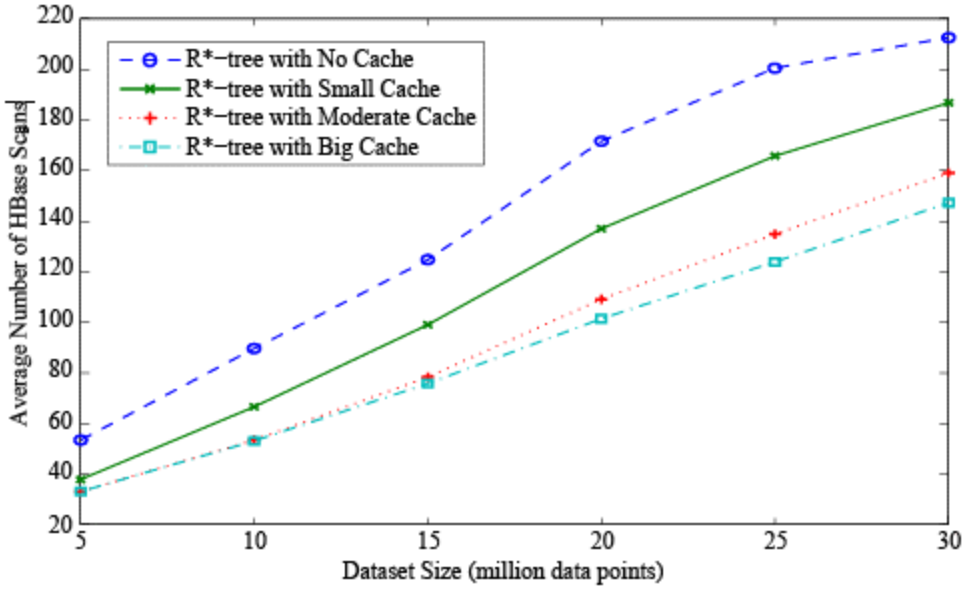



Figure 25. Effect of cache size on the amount of database requests.

In general, the models with cache outperformed the one without cache as shown in Figure 25. Even the model with the small cache option is cable of producing noticeable reduction in the number of HBase interactions as demonstrated in Figure *25*. Increasing cache size may not always result in performance leaps. For instance, if cache size is very large and the number of requested queries is relatively small, the cache can take a long time to fill-up. Another example in which cache may not be helpful involves small dataset size. Small dataset size can reduce the benefit a larger cache may have on the performance of a system, because small dataset allows to cache a higher portion of non-data nodes without even reaching the maximum number of pages. Such effect can be observed when comparing moderate and big maximum cache capacities on a smaller dataset of five million numbers. Both cache size have enough capacity to keep adding

new pages to cache without ever having a need to reclaim cache space. As a result, the number of HBase interactions for a set of a hundred queries stays the same for moderate and big cache sizes on a dataset of five million, effectively having the same impact on the performance of a system.

It is worth mentioning the fact that both algorithms, HBase R*-tree and file scan, are sequential in their nature. The tested model uses HBase merely as a storage mechanism, and no computations are performed in parallel or concurrent manner while searching except for those that might happen internally in the HBase. Both the HBase system and file scan *k*-NN were tested on the same gateway machine. Both approaches can be transformed into parallel solutions by traditional methods, like threading or Message Passing Interface (MPI), and by the utilization of MapReduce. However, since the focus of this work is on evaluating the pure performance of R-tree structures in the HBase database, it was decided not to make such parallel adaptations because parallelization of the algorithms would have masked the core algorithmic traits and deficiencies of the R-tree and R-tree derived indexes in the distributed key-value data stores.

# Chapter 6

# FUTURE WORK

As was demonstrated, an R-tree base time series search and retrieval system running in the distributed key-value data store can handle large time series. However, the performance of such systems greatly vary on a number of parameters, ranging from the property of the input sequences to the limitation of the implementation, such as construction of large indexes by parts and the limitation of the multidimensional indexing used in the system. The indexing method adopted in the proposed system restricts the queries to be the size of index dimensionality. R-tree can be constructed for any number of dimensions, but the performance degrades noticeably at high index dimensionality. The future work must be done in the direction of minimizing or entirely eliminating the limitations of the indexing technique. Certain non-index level optimizations, such as dimensionality reduction can mitigate some of the drawbacks of the index. Unfortunately, such optimizations can mask underlying index problems and are not the solutions to the unstable performance. Possible changes and improvements for the continuing work are discussed in the chapter.

## 6.1 R-tree Node Overlap

R-tree is a dynamic structure and it performs many update operations when adding new indexing points. Unfortunately, such operations are costly when the structure is stored in

the distributed environment such as HBase. As a result, R-trees used in such environments are typically constructed by parts and then the built small sub-trees are saved into the database [Cary09]. However, by doing so the R-tree index loses its dynamic nature: once the tree is written to the database its structure cannot be efficiently modified. Inability to alter the index structure after it has been placed into the key-value store leads to high node overlap. High node overlap occurs because each of the small sub-trees is likely to have significant if not complete overlap with each other, resulting in the need to inspect more branches while performing search operations.

Even without such piece-by-piece tree constructions, nodes in the tree can still overlap. The overlap becomes greater as the dimensionality of the tree increases. Both R-tree and R*-trees are susceptible for such behaviors. Even considering R*-tree optimizations aimed at reducing the node overlap, the overlaps cannot be eliminated entirely without knowing all the data beforehand and performing some additional preprocessing [Sellis87]. High overlap is one of the main reasons behind the dramatic performance degradation as the number of dimensions increases. As a result, a better mechanism to control node overlaps in R-tree derived structures is needed.

R+-tree and its variants offer a partial solution to a problem at the expense of storage required for the index. R+-trees address the overlap issues by disallowing node overlaps and enabling nodes on the same level to share children [Sellis87]. Traversing the tree while performing search does not incur the negative costs of scanning neighboring nodes and the original R+-tree research claims 50% reduction in disk access compared to the

regular R-tree [Sellis87], although it is not known how well R+-tree variant will perform in the distributed key value data store environment.

## 6.2 Dimensionality Reduction

Dimensionality reduction offers many benefits to the time series search system using multidimensional indexing techniques. When applied to the R-tree based system, dimensionality reduction allows indexing time series broken up into large segments while staying in the optimal dimensionality of the indexing structure. This approach can significantly boost the performance of the index structure and reduce the size of an index. Unfortunately, dimensionality reduction cannot be considered an ultimate solution to the R-tree high dimensionality problems, as it simply maps the high dimensional data points to a lower dimensional index, allowing more optimal index operations. In addition, searching the index space of dimensionality reduced data requires a few extra steps to ensure the proper operation of an exact perfect match and *k*-NN searches.

## 6.3 Parallel Retrieval

Choosing a distributed storage system like HBase allows for the creation of a highly parallelized algorithm for both storage and retrieval of data. As mentioned in the previously chapters, a time series is broken into large chunks and each chunk is then used to build a tree. Therefore, parallel processing can take the advantage of the independence between chunks in the piece-by-piece construction of the tree for large sequences.

Because of this nature in the tree construction, it is very easy to perform parallel computation on each of the trees and aggregate the results. HBase can be a source of data for MapReduce [HBase14] tasks, allowing for work to be distributed in the cluster.

Alternatively, the distributed nature of HBase key-value data store should allow for multiple concurrent searches from different clients to be performed at the same time. The ability to serve multiple searches in parallel is especially important for commercial applications targeted at serving multiple users at any given point of time.

## 6.4 Dynamic Query Length

In the experiments conducted in this work, the query length was always fixed to the dimensionality of an index. This is a common problem of indexes using variations of R-tree structures. Commonly multiple indexes are built to support various resolutions and query size, but such approaches consume a lot of storage space and require the search to be performed on different indexes [Rakthanmanon13]. Perfect match searches utilizing partial queries are possible. However, since the search can only be performed for a query of a predefined size, any query of a longer length will be truncated to match the dimensionality of the index. This causes some portion of the query never being used to search the index space, since the truncated part is only utilized in results validation process.

## 6.5 Caching

Main memory of a computer generally operates faster than disk storage. As it has been shown, utilizing cache can improve the performance of the system and reduce the amount of HBase scan operations. Improving the caching subsystem to be smarter at what cache pages must be kept and what pages can be discarded will increase the overall performance of the system, since it will be possible to keep more useful R-tree nodes in the main memory of a computer.

In addition to cache improvements, it is important to study how various cache parameters impact the performance of a time series similarity search system. Many various parameters, such as cache capacity, the amount of tables cleared each time cache needs to reclaim memory, various priorities for cache reclamation and others must be tested. Simply testing the impact of these cache variables and their combination is an immersive task that can help to establish optimal cache configurations for various data sizes.

# Chapter 7

# CONCLUSIONS

In this thesis, an R-tree based indexing system in a distributed data store was proposed and evaluated for searching and retrieving time series data. The system was constructed using HBase distributed NoSQL database, which runs on top of an Apache Hadoop cluster, to store and retrieve R*-tree multidimensional index. This design not only eliminates the limitation of memory/disk space but also provides scalability. Similar index structures have been used in the past for time series search and retrieval, but these systems typically reside on a single machine and not in distributed environment.

The proposed system can perform efficient similarity search against large time series. For instance, the average search time for a uniform dataset of five million data points was 0.799 seconds, almost twice as fast as the sequential scan algorithm. However, it was observed in the experiment result that the R-tree implementation in the distributed environment suffers from performance degradation at high dimensionality, which is a common limitation of the R-tree family of multidimensional indexes. Data distribution and the presence of repeated patterns in the time series also have major impact on the performance of the system.

The performance degradation due to the high dimensionality in the index is probably the most prominent reason why most researchers limit the usage of R-tree to spatial, 2-dimensional data. Very little literature was found showing that R-tree is used for time series data because time series data in most cases require higher index dimensionality. Although techniques in dimensionality reduction have a potential to mitigate the problem of inefficiency at higher index dimensionality, such approaches do not address the root of the performance degradation but only mask and delay the appearance of an issue.

The system is also impacted by the properties of the input data. Certain datasets containing large number of repeated subsequences were reported with exceptional performance. Other datasets that exhibit higher data concentration in smaller regions of index space, such as normally distributed data, tend to receive poor performances with R-trees because of the high node overlap issue. Big performance variability due to the data distribution and the presence of repeated patterns implies the need of evaluating whether an R-tree or similar structure can be used efficiently with a dataset. .

Many improvements and optimizations can be done to mitigate the shortcoming of the indexing structure. Utilizing dimensionality reduction techniques can allow indexing larger datasets at higher dimensionality while keeping the index at its optimal performance for a wider set of operating conditions. Algorithms used to retrieve the similar segments can be changed to allow parallel execution improving resource

utilization and retrieval speed for low-tenant systems. A caching system can help speed up the performance significantly by reducing the number of database interactions while searching and retrieving, but it is at the expense of using more main memory.

In addition to the improvements mentioned above, solving a node overlap problem is likely to provide the most scalable solution to the issues of performance degradation and performance variability. The usage of R+-trees or similar approaches can potentially address the overlap problem at the expense of increasing storage space consumption.

The R-tree index in the HBase environment is a feasible solution for time series search, especially if the improvements outlined above are implemented. HBase offers a number of advantages over other methods, such as the ability to easily distribute and parallelize algorithms and to provide concurrent accesses to multiple users. R-tree index is fairly easy to construct. It can achieve very good performance on certain kind of data, and stays flexible enough to handle time series with various properties.

# REFERENCES

Print Publications:


[Agrawal95]
Agrawal R., Lin K., Sawhney H. S., & Shim K., "Fast similarity search in the presence of noise, scaling, and translation in time series databases," *VLDB*, Zurich, Switzerland, 1995.

[Beckmann90]
Beckmann, N., Kriegel, H. P., Schneider, R., & Seeger, B. (1990). "The R*-tree: an efficient and robust access method for points and rectangles*"* (Vol. 19, No. 2, pp. 322-331). ACM.

[Buza11]
Buza, K., Nanopoulos, A., Schmidt-Thieme, L., & Koller, J., "Fast classification of electrocardiograph signals via instance selection," *Healthcare Informatics, Imaging and Systems Biology (HISB), First IEEE International Conference on* (pp. 9-16). IEEE, 2011.

[Charapko14]
Charapko, A. and Chuan, C.-H., "Indexing and retrieving continuations in musical time series data using relational databases," the 10th IEEE International Workshop on Multimedia Information Processing and Retrieval, in conjunction with IEEE International Symposium on Multimedia, December 10-12, 2014.

[Cary09]
Cary, A., Sun, Z., Hristidis, V., & Rishe, N. "Experiences on processing spatial data with mapreduce," *Scientific and statistical database management* (pp. 302-319). Springer Berlin Heidelberg, 2009.

[Halkiopoulos12]
Halkiopoulos, C., & Boutsinas, B., "Automatic Interactive Music Improvisation Based on Data Mining," *International Journal on Artificial Intelligence Tools*, *21*(04), 2012.

[Guttman84]
Guttman, A. *R-trees: a dynamic index structure for spatial searching*, ACM, Vol. 14, No. 2, pp. 47-57, 1984.

[Kahveci01]
Kahveci, T., & Singh, A. "Variable length queries for time series data," *Data Engineering. Proceedings. 17th International Conference on*, IEEE, pp. 273-282, 2001.

[Keogh01A]
Keogh, E., Chakrabarti, K., Pazzani, M., & Mehrotra, S., "Locally adaptive dimensionality reduction for indexing large time series databases". *ACM SIGMOD Record*, *30*(2), pp. 151-162, 2001.

[Keogh01B]
Keogh, E., Chakrabarti, K., Pazzani, M., & Mehrotra, S., "Dimensionality reduction for fast similarity search in large time series databases," *Knowledge and information Systems*, vol. 3, no. 3, pp. 263-286, 2001.

[Keogh05]
Keogh, E., & Ratanamahatana, C. A., "Exact indexing of dynamic time warping." *Knowledge and information systems*, *7*(3), pp. 358-386, 2005.

[Lemire09]
Lemire, D., "Faster retrieval with a two-pass dynamic time warping lower bound," *Pattern Recognition* vol. 42 no 9, pp. 2169-2180, 2009.

[Loh00]
Loh, W. K., Kim, S. W., & Whang, K. Y., "Index interpolation: an approach to subsequence matching supporting normalization transform in time series databases," *Information and Knowledge Management. Proceedings. Ninth International Conference on,* ACM, pp. 480-487, 2000.

[Mathur11]
Mathur, A., Mathur, M., & Upadhyay, P., "Cloud Based Distributed Databases: The Future Ahead," *International Journal on Computer Science and Engineering*, *3*(6), 2477-2481, 2011.

[Pachet03]
Pachet, F., "The continuator: Musical interaction with style," *Journal of New Music Research*, *32*(3), pp. 333-341, 2003.

[Perng00]
Perng, C. S., Wang, H., Zhang, S. R., & Parker, D. S., "Landmarks: a new model for similarity-based pattern querying in time series databases," *Data Engineering. Proceedings. 16th International Conference on,* IEEE, pp. 33-42, 2000.

[Rakthanmanon13]
Rakthanmanon, T., Campana, B., Mueen, A., Batista, G., Westover, B., Zhu, Q., Zakaria, J. & Keogh, E, "Addressing big data time series: Mining trillions of time series subsequences under dynamic time warping". *ACM Transactions on Knowledge Discovery from Data (TKDD)*, *7*(3), 10, 2013.

[Sellis87]
Sellis, Timos, Nick Roussopoulos, and Christos Faloutsos. "The R+-tree: A dynamic index for multi-dimensional objects," 1987.

[Wang00]
C. Wang, S. Wang, "Supporting content-based searches on time series via approximation," *Scientific and Statistical Database Management. Proceedings. 12th International Conference on,* 2000.

[Wei13]
Wei, L. Y., Hsu, Y. T., Peng, W. C., & Lee, W. C., "Indexing spatial data in cloud data managements," *Pervasive and Mobile Computing*, 2013.

[Zhou13]
W. Zhou, J. Han, Z. Zhang. Z. Xu, J. Dai. "HDKV: Supporting efficient high-dimensional similarity search in key-value stores," *Concurrency and Computation: Practice and Experience,* 25(12), pp. 1675–1698, 2013.

Electronic Sources:

[Hbase14]
Apache, "Apache HBase" http://hbase.apache.org/, 2014, last accessed October 12, 2014.

[LMS Test.Lab14]
Siemens, "Airbus uses LMS Test.Lab to improve and streamline its flutter analysis process", http://www.plm.automation.siemens.com/CaseStudyWeb/dispatch/viewResource.html?resourceId=40536, 2014, last accessed October 5, 2014.

# VITA

Aleksey Charapko expects to receive his Master of Science in Computer and Information Sciences degree in the spring of 2015. Aleksey has been working as a graduate research assistant for Dr. Ching-Hua Chuan for over two years. The work performed as a research assistant resulted in a number of conference and journal publications in the field of music information retrieval. Aleksey has been working as an application developer since 2012. He has been responsible for designing, developing and maintaining custom software products using the technologies such as PHP, Java, Xojo, ASP, MySQL and SQL Server.