CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY

ASSIGNMENT OF MASTER'S THESIS

Title:	Approximate Pattern Matching In Sparse Multidimensional Arrays Using Machine Learning Based Methods
Student:	Bc. Anna Kučerová
Supervisor:	Ing. Luboš Krčál
Study Programme:	Informatics
Study Branch:	Knowledge Engineering
Department:	Department of Theoretical Computer Science
Validity:	Until the end of winter semester 2018/19

Instructions

Sparse multidimensional arrays are a common data structure for effective storage, analysis, and visualization of scientific datasets. Approximate pattern matching and processing is essential in many scientific domains.

Previous algorithms focused on deterministic filtering and aggregate matching using synopsis style indexing. However, little work has been done on application of heuristic based machine learning methods for these approximate array pattern matching tasks.

Research current methods for multidimensional array pattern matching, discovery, and processing. Propose a method for array pattern matching and processing tasks utilizing machine learning methods, such as kernels, clustering, or PSO in conjunction with inverted indexing.

Implement the proposed method and demonstrate its efficiency on both artificial and real world datasets. Compare the algorithm with deterministic solutions in terms of time and memory complexities and pattern occurrence miss rates.

References

Will be provided by the supervisor.

doc. Ing. Jan Janoušek, Ph.D.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague February 28, 2017

Czech Technical University in Prague
Faculty of Information Technology
Department of Knowledge Engineering

Master's thesis

Approximate Pattern Matching In Sparse Multidimensional Arrays Using Machine Learning Based Methods

Bc. Anna Kučerová

Supervisor: Ing. Luboš Krčál

9th May 2017

Acknowledgements

Main credit goes to my supervisor Ing. Luboš Krčál without whom this work could not be finished.

Declaration

In Prague on 9th May 2017 ....

Citation of this thesis

Kučerová, Anna. *Approximate Pattern Matching In Sparse Multidimensional Arrays Using Machine Learning Based Methods.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2017.

Abstract

The main goal of this work is to propose a solution of approximate pattern matching with the use of machine learning based methods. This is done with the help of Locality Sensitive Hashing and existing algorithms. Idea of LSH is used for searching of positions of potential results and their verification is executed as in existing algorithms. Previous work was focused primarily on low dimensional pattern matching. The outcome of this work is an algorithm together with time measures and comparison with already existing solutions. Some of the comparing algorithms were only theoretically designed and not implemented until now. The solution also uses binary format used in a commercial array database.

Keywords Approximate Pattern Matching, Exact Pattern Matching, Locality Sensitive Hashing, SciDB, Regular Grids on Arrays, Multidimensional Arrays

Contents

List of Figures

List of Tables

Introduction

Pattern matching is widely needed because of its usage in the most of the modern databases. Whenever user needs to find specific data matching his requirements, it can be seen as if the query was a pattern that needs to be matched in the data. Imagine an example query: astronomer wants to find all stars in specific part of the night sky with luminosity higher than x, or meteorologist needs to check coordinates of all possible tornadoes on Earth where a tornado can be specified by different humidity, wind change, temperature [28].

This search can be divided into two categories depending on the presence of errors. If no mismatches are allowed it is called *exact pattern matching*, when otherwise the problem is called *approximate pattern matching.* Approximate version of pattern matching can have numerous advantages like allowing search with the spelling errors or only partially specified search.

Majority of the research in this field of both exact and approximate pattern matching was done primarily on one and two dimensional arrays where algorithms that can process higher dimensional arrays are usually low dimensional algorithms converted into high dimensional array spaces or use methods that can reduce dimensionality of both data and patterns.

Modern databases (e.g. SciDB) contain ever increasing number of data because they store whole range of information for example satellites imagery [46]. These data are stored in sparse arrays and thus need to be either preprocessed into special structure like Coordinate format, Compressed Sparse Row format [3], which are commonly used for computations in linear algebra, or a binary mask can be used, which is used for achieving of efficient data storage, and is the implemented solution in this work.

The main goal of this work is to analyse current solutions of pattern matching while implementing three of these proposed in the paper [1]. Next step will focus on creating of a new algorithm based on these and the usage of machine learning methods. Another condition to fulfil is for the algorithm to be able to process data in SciDB binary format.

Structure of this work is as follows, next chapter called Related work 1 is focused on presenting various papers and work done on pattern matching which applies to this work. Chapter 2 is dedicated to analysis of important methods, approaches and also explaining of used terms. It firstly presents specification of the problem 2.1, which is needed for better understanding of the fundamentals, then it focuses on specifying the used data structure 2.2. Next two sections are dedicated to the explanation of similarity hashing 2.3 and similarity measures 2.4. Section 2.5 focuses on the SciDB database together with its data model, used language, Data storage method and use case.

Third chapter explains the problem of multidimensional approximate pattern matching 3 together with its existing solutions and approached methods. Section 3.4 analyses usable machine learning methods for the problem of approximate pattern matching.

Goal of the chapter 4 is to explain all implementation details together with better specification of used algorithms. It also presents input and output format of the data and finally introduces some possible modification.

Chapter 5 is dedicated to the stating of the results of the implemented solutions and their comparisons with the explanation. Next chapter presents the conclusion of this work with a possibility of future work.

Chapter 1

Related work

Theme of this thesis is inspired by the work of Baeza-Yates and Navarro in the field of pattern matching algorithms. In their papers [1], [20], [22], they extend different types of approximate pattern matching, such as in the way of reducing dimensionality by creating filters, or by creating new similarity measures for strings. When mentioning filters, authors of the first string filter Bird and Baker, and their successors J. Kärkkäinen and Ukkonen can not be forgotten.

Among the most important papers and work done in this field belong R. Baeza-Yates with his work on similarity of two-dimensional strings written in 1998 [19] which is focused on computing edit distance between two images while using new similarity measures. Another fundamental paper by R. Baeza-Yates and G. Navarro [20] is concerned with fast two-dimensional approximate pattern matching (also from 1998) where authors construct search algorithm for approximate pattern matching based on filtering of one dimensional multi patterns.

Baeza-Yates also worked with C. Perleberg on the paper published in 1992 [21] dealing with the question of fast and practical approximate pattern matching which they solved by matching strings with their mismatches based on arithmetical operations. Errors here are based on partitioning the pattern. In the next paper written in 1999 by Baeza-Yates and Navarro named Fast multi-dimensional approximate string matching [22] they are extending two dimensional algorithm into n dimensions by creating sub-linear time searching algorithm. This solution turned out to be better than using the dynamic programming.

Lot of progress was made in this field by J. Kärkkäinen and E. Ukkonen with their concentration on filters to quickly discard a large amount of data [52]. In their work from 1994 they concentrate on two and higher dimensional pattern matching in optimal expected time where they try placing a static grid of test points into the text and then eliminate as many potential occurrences of incorrect patterns as possible.

Among other scientists dealing with the question of efficient two dimensional pattern matching belong K. Krithivasan and R. Sitalakshmi [25] who were focused mainly on solving two dimensional pattern matching in the presence of errors as published in their paper from 1987.

Also, there must be mentioned work done in the field of sequence alignment and general work with DNA sequences which is very similar to the task of pattern matching. Of all the scientists involved there can be named P. Sellers with his paper written in 1980 and named: The theory and computation of evolutionary distances: pattern recognition [26], where he is focused on finding pattern similarities between two sequences with the computation time being a product of the sequences length.

Lastly in the book Jewels of Stringology: Text Algorithms published in 2003 [23] with authors M. Crochemore, W. Rytter there are presented various basic algorithms that solves the question of pattern matching but only when considering strings.

All of the papers and work mentioned above are focused mainly on string pattern matching and maximally two dimensional spaces, where when creating a solution for higher dimensional space it is usually generalized version for one or two dimensional space.

Chapter 2

Analysis

Content of this chapter will focus on defining the problem of pattern matching in the first section 2.1, after it the specification of multidimensional arrays will follow 2.2, succeeded by the explanation of similarity hashing 2.3 and similarity measures 2.4.

Another section is devoted to introduction of the scientific database SciDB 2.5 with its binary data format and some possible ways of working with its data.

Last section presents machine learning methods 3.4 especially K Nearest Neighbor and kernel based methods as they can be used for solving or improving of the solution of this problem together with the algorithms presented in the next sections.

2.1 Problem specification

This section will focus on definition of approximate pattern matching.

First thing first, the multidimensional array A is composed of dimensions D and attributes A_p and consists of cells. Dimensions D form the coordinate system for the array. The number of dimensions in an array is the number of coordinates or indices needed to specify an array cell. Each dimension is specified by a unique name, start and end coordinate. Attributes A_p contain the actual data and are specified by a unique name and value type. This is the base data type used in this work and it is also base representation of data in the SciDB [45]. More information about SciDB database can be found in the section 2.5. Main difference between this model and model used by common relational databases is that this model uses dimensions and attributes whilst relational one uses only attributes.

In the table 2.1 there is an example of the two dimensional array with dimensions: NHL team and Season. Attribute values, in the example number of points acquired during the regular season, are inside the cells, while the

	Season				
NHL team	2012–13	2013–14	2014–15	2015–16	2016–17
Colorado Avalanche	39	112	90	82	48
St. Louis Blues	60	111	109	107	99
Pittsburgh Penguins	72	109	98	104	111
New York Rangers	56	96	113	101	102

Table 2.1: Example 2D array data structure

dimension values can be seen on the edges. For simplicity reasons only one attribute is considered in this example data.

Pattern P can be of two types, where the first one is defined as multidimensional array with dimensions $D_p \subseteq D$ and attributes $A_p \subseteq A$ where the number of dimensions $D_p \geq 1$ and number of attributes $A_p \geq 0$. This type of pattern is called template pattern and it can be used when the user has an exact idea about the result. Example of this pattern using the table 2.1 would be to ask which teams managed to get 107 points during one season and only 99 during the next one. Right answer would be St. Louis Blues in the seasons 2015-16 and 2016-17.

And the second type is defined as a condition limiting the values of dimensions and attributes. It consists of a hard constraints in the sense of: Find all parts so that their average value of attribute A_i is lesser than x . This type of pattern is called aggregate pattern, but this type is not used in this work.

As an example of this pattern there can be used the problem of finding weather conditions on earth, which is specified by 2D array with latitude and longitude as dimensions and humidity, wind speed, temperature and other weather related attributes. When in need of finding tornadoes in Europe user would have to use dimensions constraints for Europe and wind and humidity conditions specific for tornado to obtain possible results.

Similarity measure S is a function which quantifies the similarity between two given arrays. It is used to compare the pattern and parts of multidimensional array. Which then gives the information about the similarity of the compared parts and the pattern. Its value belongs in the interval $< 0, 1 >$ where the higher the value the closer the items are, with special case of 1 which means that the items are exactly the same, and 0 meaning there is nothing similar between the items measurable by this function. More about similarity measures is in the section 2.4.

Match M is a subset of array A with dimensions D_m and attributes A_m, where $A_m \subset A_p$ and the value of similarity measure is higher than a certain threshold.

Approximate pattern matching can then be defined as a function with input parameters of pattern P and multidimensional array A which outputs set of matches N with the use of similarity measures S on arrays.

Exact pattern matching is a special case of approximate pattern matching where the dimensions of pattern D_p are equal to the dimensions of match D_m, the attributes A_m are subset of attributes A_p and the value of similarity measure $S = 1$.

Another information about approximate pattern matching and already proposed solutions are explained in the chapter 3.

Solutions to this problem are useful and used in every day life for example in: search in database with errors, finding genes, sequence alignment, image registration, computer vision, object recognition, molecular modeling and so on.

Next chapter focuses on explaining all important terms, algorithms and methods that are commonly used for solving the problem of approximate pattern matching.

2.2 Multidimensional array data structure

Based on [4] the multidimensional array consists of dimensions and cells in contrary to non dimensional array that consists only of cells. Each cell is composed of attributes. To split these array into grid one can use Regularly gridded chunking where chunks have equal shape and do not overlap or Irregularly gridded chunking which is one of the approaches used by the RasDaMan model [68]. To perform selection queries on these arrays there is a need to constraint the query either by dimension or attribute constraint or both of them.

2.2.1 Sparse Data

When data are sparse it means that some attributes can contain 0 or null or empty value. The data can then be viewed as a sparse matrix containing 0 values at the position with no attribute or 0 valued attribute. Because of the nature of this data they offer a big opportunity for using various compressing algorithms to avoid looking through positions with none or 0 values. SciDB [45] treats empty cells like they are non existent i.e. they are not evaluated with the count aggregate. Some Linear Algebra operators process them as 0 values and during join, sort and unpack operations they are disregarded. An example of sparse data can be seen in the table 2.2 where cells containing value 0 are regarded as empty.

2.2.2 Indexing

Indexing improves the speed of data retrieval operation at the cost of additional writes and storage space needed to maintain the index structure. It is used in situations where it is needed to quickly access data without having

Attribute 1	Attribute 2	Attribute 3
0	0	0
1	2	0
1	0	0
0	0	0

Table 2.2: Example of sparse data with three attributes.

to search every position. Common methods of indexing are described in the sections below.

2.2.2.1 Spatial Indexing

Spatial indexing can be done by four main styles. First of them is by using access methods like point or spatial one, which are used for indexing of B-tree with its variants, Multidimensional range tree, Quadtree, KD-tree and R-trees [29]. Another choice is to use Space filling curves which is good for maping the space into one dimension, like Peano-Hilbert, Z-order, row-prime order. Fourth choice is for Tilings of the domain space [8]. These arrays usually have high value locality so the choice is to index individual cels or certain vale ranges as an object. Retrieval of the array indexing of arrays can be done by Depth First Search, but this method is not space efficient [7].

When using Quadtree method one of the ways of creating it is to use the Adaptive Mesh Refinement algorithm described below 2.2.2.2.

2.2.2.2 Adaptive Mesh Refinement

AMR is a function with input parameters of Interval of values, spatial selection and its output is a set of cells with its values. To be returned by the function the cell must satisfy three conditions which are:

- cell must be uncovered
- cell belongs to the spatial selection
- value of the cell belongs in the Interval of values

AMR structure contains levels and refinement ratio [27]. First (coarsest) level is a box (defined by lower and upper corner). Every level can be decomposed into a disjoint union of boxes. A cell contains a pair of information and that is a level it belongs to and its position by coordinates. By uncovered, it is meant that the cell lies in the last (the least coarse) level. Spatial selection needs to be interpreted within the coordinate system of the first level and a cell in the spatial selection if its coordinates belongs inside the first level selection. Interval of values is simply the interval of values that the elements

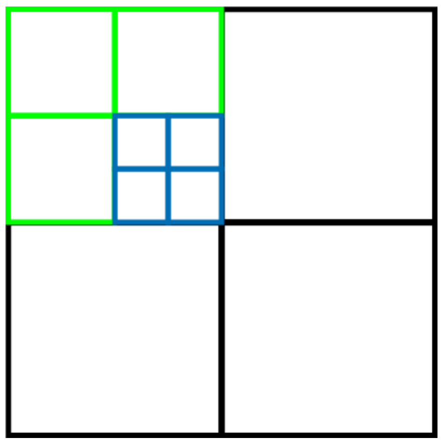


Figure 2.1: Example of AMR decomposition, different levels are represented by different colors

inside AMR structure can have. Values returned by function are in the pair with structure $< cell, value >$.

Construction of AMR Index tree consists of two phases where in the first one the object is split into boxes which create first level and then in the potential valuable boxes creates another level by splitting the box into smaller ones and so on, if the splitting is needed. In the second phase it creates a tree for every box which is covered by another (smaller) boxes.

2.2.2.3 Bitmap Indexing

Bitmap indexing is very useful for its variety. It provides efficient way to evaluate logical conditions on large data sets, because of efficient implementation of the bitwise logical AND, OR and NOT. It also provides significant opportunities for compression (there are also compression algorithms that support compressed domain implementations of the bitwise logical operations and enables query processors to operate directly on compressed bitmaps without the need to decompress). Most of the newer compression algorithms use run-length encoding. Low level bitmaps can be used to index single attribute value and high level ones are used for multiple values. Index structure uses binning strategy which reduces overall number of bitmaps required to index the data, but increases the number of cells later needed to verify. This is called Candidate check.

This index is useful in cases where the values of a variable repeat very frequently, for example attributes that can have only two values (gender).

Another type of bitmap usage is for Equi-width and Equi-depth binning. Encoding can be performed by:

Equality encoding: each bin is encoded by one bitmap

Range encoding: each bitmap encodes range of bins

Interval encoding: each bitmap encodes XOR of two cells

This can be later compressed by Byte-aligned bitmap code or Word-aligned Hybrid algorithms.

2.2.2.4 Bucket Indexing

Bucket Indexing methods aggregate data in buckets to save overall size. Data are split into buckets either by having the same or very close index value. Special tree used for storing of this structure is called k-d-b tree and it is a hybrid between k-d tree and B-tree. In its structure one internal node can consist of a set of adjacent regions, leaves can consist of multiple points. It has another variants like LST tree, hB-tree, k-d-b-trie.

2.2.2.5 Grid Indexing

Grid indexing brings it closer to array data model. It splits the domain into non-regularly gridded space and it uses grid file which requires index to maintain linear scale data structure. Algorithm EXCELL creates a modification of grid file allowing it to split the domain space along a single dimension only.

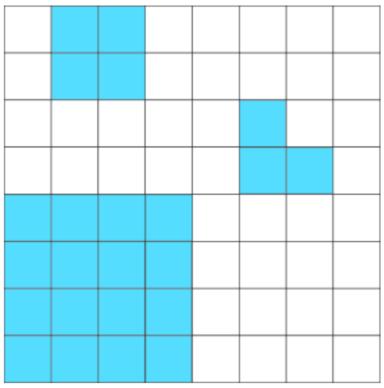
Compressed spatial hierarchical bitmap index [6] creates a spatial hierarchy which consists of nodes, leaves, parents, children, descendants, internal nodes and leaf descendant of a node. The structure itself creates a set of bitmaps where for each node there is a corresponding bitmap and then it creates and stores compressed bitmap for each node, except empty ones which all have one node. This method has one input parameter and it is target block size. Algorithm then ensures that all index files written to the disk are at least block size big which is done by concatenation of compressed bitmap files. This means that the size of the files will be upper bounded while the cost of the bitmap reads will be lower bounded.

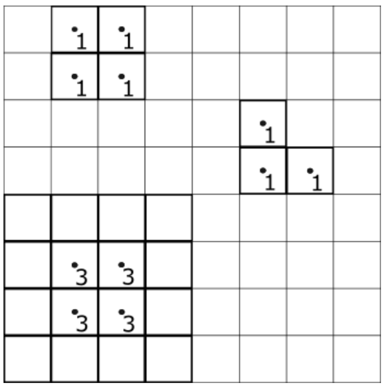
2.2.2.6 Inverted Indexing

Unlike common indexing methods (mentioned above) this methods goal is to create a set of all unique values in the data where each value has a list of the documents containing it. This solutions is mainly used for databases containing text to allow full text searches. However the drawback of this method is in a high processing cost of a newly added elements. [57]

2.2.2.7 Medial Axis Transformation

Before defining Medial Axis Transformation (MAT) there is first needed to explain what a *disk* is. Disk is some specified shape or neighbourhood of some specific point and it also has a radius, it belongs in the object and can not


(a) Objects and cells that MAT algorithm will be performed at



(b) MAT for the collection of objects and cells, maximal block is anchored at the unit-size element at its center

reach out of the object bounds. Object can then be defined as the union of the maximal disks it contains.

Medial axis transform of S consists of the centres of these disks together with their radii. The distance of a point x in S from S is then the length of the shortest path from x to the complement S. MAT can also be defined as the set of all points in S which do not belong to the minimal path of any other point, together with their distances. This can be imagined like constitution of a skeleton.

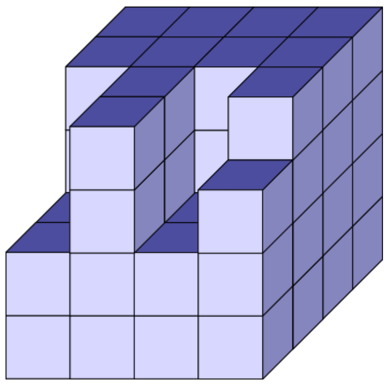
Each object is then defined by the triple (c, d_c, w_c) where c is the cell, s_c is the largest square in which it is contained with width w_c. Set of these triples is called MAT of object S. This is very compact method for simple shapes.

In the field of digital pictures disks are approximated by squares and their orientation depend on the definition of grid distance, which also defines the type of the skeleton.

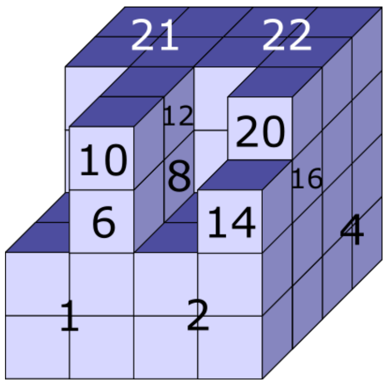
Base types of distances that can be used are:

- Chess-board distance - disks are upright squares
- City block distance - disks are approximation of diagonal squares
- Euclidean distance - disks are discs, also more appropriate in continuous space
- Absolute value distance - disks are diamonds

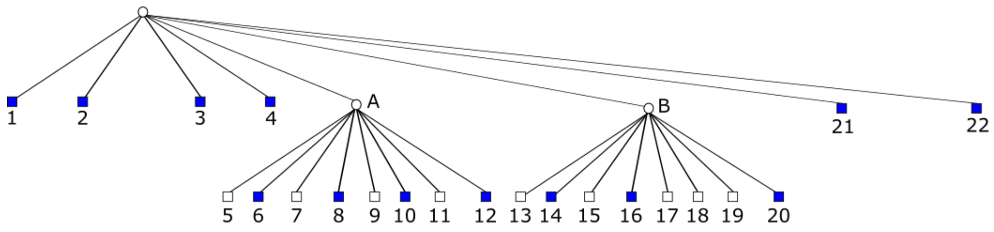
When using irregular grid it can decompose the space using hyperplanes resulting in collection of blocks. By this there can be recreated irregular block sized grid, where no recursion is involved and block decomposition is handled by explicit representation. These blocks are not congruent (this will add access structure containing linear scales which indicates position of partitioning hyperplane).



(a) Object that block decomposition will be used on



(b) Region octree block decomposition of the picture above



(c) Octree block decomposition of the picture in Figure 1

Figure 2.3: Example of octree decomposition.

Representation of block decomposition can be done by using different types of quadtree or octree (2.3). Most commonly used decompositions are:

- Multicoloured quadtree - can contain more than two colours and also more block labels
- Binary quadtree - for black and white values only
- Block specified by a pair of coordinates - specified by upper left corner coordinate and size of the side
- Morton block - encoded by localization code (variant of Morton number), which is power of two and Morton ordering is NW, NE, SW, SE
- Scan of resulting representation from extreme right end to first zero valued bit
- Sorting the values in increasing order - Efficient pixel interleaving compression technique (EPICT)

For more informations about typical usage and types of MAT see D.

2.2.3 Representation of multidimensional data

When representing hierarchical domains there are three categories of different approaches.

First approach is that there can be used the K^n-*treap* which is an extension of k^2-*treap* and it uses k^n-*tree* to store its topology. It is an efficient representation of wide empty areas which splits each dimension into k equal sized parts and saves it as a structure of values and tree structure. It can be searched from root to leaves for aggregated values of a range cells using the depth-first multi-branch traversal.

Second representation is per CMHD which means Compact representation of Multidimensional data on Hierarchical Domains. This algorithm recursively divides matrix into several submatrices. It assumes same height of all the hierarchies. If artificial levels are needed they can be added by subdividing all the elements of a level in just element (itself).

Third possibility is to use Queries where elements of the different dimensions are all on the same level in their hierarchies. Different levels contain different labels. It is required to have top down traversal when searching hash tables for labels provided by the query for different dimensions to locate corresponding nodes. From nodes it traverse each hierarchy upwards to find out its depth and child that must be followed to reach it [60].

2.3 Similarity hashing

Hashing approach is based on transforming the data item into a low-dimensional representation or a short code composed of a sequence of bits.

This method is used to hash similar items into the same buckets, it is used in similarity searching systems. To measure similarity a feature based distance can be used. When hashing data it is the same approach as transforming them into a low dimensional space. Specific hashing approaches used in implementation will be explained in the section Implementation.

Outcomes of this approach can be used in nearest neighbour search problem.

2.3.1 Locality Sensitive Hashing

This method was proposed in 1998 for approximate nearest neighbour search in high dimensional spaces. Research often views this method as: *"probabilistic similarity-preserving dimensionality reduction method from where the hash codes can provide estimations to some pairwise distance or similarity"* [56]. The essential idea of this approach is to have a hash function that will return the same value for nearby items from the data. But to achieve high precision, there are several disadvantages. First of them is that to acquire wanted precision the usage of long hash codes is needed, which reduces the recall. This can be solved by creating multiple hash tables but it also increases storage requirements as well as the query time. Second obstacle is that LSH can be applied only in certain metrics e.g. l_p and Jaccard.

Main usage of this approach is in finding near-duplicate web pages, image detection, clustering and so on.

Types of this method can be split into several categories. First of them is based on the l_p distance: LSH with p-stable distributions, Leech lattice, Spherical LSH, Beyond LSH. Another type is split by the usage of Angle-Based distance: Random projection, Super-bit LSH, Kernel LSH, LSH with learned metric, Concomitant LSH, Hyperplane hashing. Next type of Locality Sensitive Hashing is based on Hamming distance. There are also types dependent on Jaccard coefficient: Min-hash, K-min sketch, Min-max hash, B-bit minwise hashing, Sim-min-hash. Among other similarities hashes belongs: Rank similarity, Shift invariant kernels, Non-metric distance, Arbitrary distance measures. Usability of this method highly depends on the quality of hashing method used.

2.3.2 Learning to Hash

Goal of this method is to learn dependencies in the data and create task-specific hash function that gives compact binary codes to achieve good search accuracy [64]. For achieving this goal, several sophisticated machine learning tools and algorithms have been adapted to hash functions design. Among the most important belong: boosting algorithm, distance metric learning, asymmetric binary embedding, kernel methods, compressed sensing, maximum margin learning, sequential learning, clustering analysis, semi-supervised learning, supervised learning, graph learning, and so on.

This concept can be used for solving of hashing-based ANN search.

2.3.3 Randomized Hashing

This type of hashing covers e.g. the LSH family and has been popular due to its simplicity. There are two main types of this approach based on the problem solution space.

First of them is the Random Projection Based Hashing which is based on an idea of preserving the locality in the original Hamming space.

In machine learning, recent research is focused on leveraging data-dependent and task-specific information for improving efficiency of hash functions like this. Some of the examples are: using kernel learning with LSH [65], boosted LSH [66] and non-metric LSH [67].

Second type is hashing based on random permutations. This method is also known as min-wise independent permutation hashing or MinHash. Originaly the Minhash was applied in clustering and for eliminating of near-duplicates among web documents represented as sets of the words. [11]

2.3.4 Data-Dependent and Data-Independent Hashing

This category of hashing is based on the knowledge about the given dataset. Randomized hashing mentioned in previous section belongs in the category of data-independent hashing as it does not need any special information about the dataset. However this is also a limitation of this approach as these methods have strict performance guarantees and they are not specifically designed for certain datasets. Limitations of this approach caused the creation of data-dependent approach which uses supervision to design better and more efficient hashes [64].

2.3.5 Supervision Hashing

This method can be split into three more categories based on the level of supervision.

First of them is unsupervised which methods try to integrate data properties like distribution into the hash value. Examples of this approach are:

- Spectral Hashing – idea of this hashing is to hash similarity based on Hamming distance and use only a small number of hash bits
- Graph Hashing – this method revolves around hashing graphs i.e. nodes and edges
- Kernelized LSH – this approach is focused on using kernel functions as a hashing functions
- Spherical Hashing – basic idea is to use a hypersphere to formulate a spherical hash function

Second is supervised learning which ranges from usage of kernel methods and metrics (defined in section 2.4) to deep learning. Examples of this approach are introduced in papers [69] and [70].

Last type using semi-supervised learning usually designs hash functions based on both labelled and unlabelled data.

Categories above can again be divided into point-wise, pair-wise, triplet-wise and list-wise subcategories. These hashes can be created either by using information from a single item or an arbitrary number of items [64].

2.3.6 Linear and Nonlinear Hashes

This type is based on the type of function used to create hashes, e.g. when using linear function for hash creation it is linear hash, and vice versa. Because of the computational requirements linear functions are usually used more often than non-linear ones [64].

2.3.7 Weighted Hashing

When using traditional hashing the goal is to map the data into *non-weighted* Hamming space. However sometimes it can be easily observed that different bits are behaving differently [64]. Because of this a technique were designed that learns a specific weight for each bit of the original hash.

2.4 Similarity measures

By using a term similarity there is meant measure of how “close” to each other two instances are by quantifying the dependency between two sequences. The closer they are the larger the similarity value. However there are two ways of approaching the similarity measure. First of them is dissimilarity, which is a measure of how different two instances are. Dissimilarity is larger the more different the instances are. Some scientists use word proximity by which they refer to either similarity or dissimilarity.

Similarity can be measured between strings, numbers, tuples, objects, images and so on, and its value is normalizes, which means in the interval [0,1], where 1 is for exact match and 0 for completely different objects [38].

A similarity measure M is considered a metric if its value increases as the dependency between corresponding values in the sequence. A metric similarity must satisfy following conditions:

- Limited Range: $M(X, Y) \leq S_0$ for some arbitrarily large number S_0
- Reflexivity: $M(X, Y) = S_0$ if and only if $X = Y$
- Symmetry: $M(X, Y) = M(Y, X)$
- Triangle Inequality: $M(X, Y)M(Y, Z) \leq [Z(X, Y) + M(Y, Z)]M(X, Z)$

Where S_0 is the largest similarity measure between all possible X and Y sequences.

In a similar way a dissimilarity measure D is considered a metric if it produces a higher value as corresponding values in X and Y become less dependent. A metric dissimilarity must satisfy following conditions:

- Non-negativity: $D(X, Y) \geq 0$
- Reflexivity: $D(X, Y) = 0$ if and only if $X = Y$
- Symmetry: $D(X, Y) = D(Y, X)$
- Triangle Inequality: $D(X, Y) + D(Y, Z) \geq D(X, Z)$

Typically, if presented with a similarity measure, it is revertible into dissimilarity measure and vice versa.

Generally there is a request for measures to have properties of a metric but they can be quite effective even without being a metric. For example, ordinal measures are not metrics but they are very effective in comparing images captured under different lighting conditions [39].

Similarity measures can be divided into five main categories depending on their domain or character. In the following sections all these categories will be described and their main methods mentioned.

For more informations about similarity measures see: [37] [38] [39].

2.4.1 Numeric methods

These methods can be used for numeric attributes where it can be treated as a vector $\bar{x} = (x_1, \ldots, x_N)$ for attributes numbered $1, \ldots, N$. Some of the most commonly used methods are shown in the list below as described in [12].

- Euclidean Distance: $d_E(\bar{x}, \bar{y}) = \sqrt[2]{\sum_{k=1}^{N}(x_k - y_k)^2}$
- Squared Euclidean Distance: $d_E(\bar{x}, \bar{y}) = \sum_{k=1}^{N}(x_k - y_k)^2$
- Manhattan Distance: $d_M(\bar{x}, \bar{y}) = \sum_{k=1}^{N} |x_k - y_k|$
- Minkowski Distance: $d_{M,\lambda}(\bar{x}, \bar{y}) = (\sum_{k=1}^{N}(x_k - y_k)^{\lambda})^{\frac{1}{\lambda}}$
- Chebyshev Distance: $d_{M,\inf}(\bar{x}, \bar{y}) = max_{k=1,\ldots,N}(|x_k - y_k|)$
- Earth Mover's Distance: $d_{EMD}(P, Q) = \frac{\sum_{i=1}^{m}\sum_{j=1}^{n} f_{ij} d_{ij}}{\sum_{i=1}^{m}\sum_{j=1}^{n} f_{ij}}$, where P has m clusters with $P = (p_1, w_{p1}), (p_2, w_{p2}), \ldots, (p_m, w_{pm})$ where p_i is the cluster representative and w_{pi} is the weight of the cluster. Similarly signature $Q = (q_1, w_{q1}), (q_2, w_{q2}), \ldots, (q_n, w_{qn})$ has n clusters. $D = [d_{ij}]$ is the distance between clusters p_i and q_j. $F = [f_{ij}]$ is the flow that minimizes the overall cost.

2.4.2 Edit-Based methods

First it is needed to define exact and truncated match. Similarity of exact match is equal to 1 if x=y and 0 otherwise. Similarity of truncated match from the beginning is 1 if $x[1:k] = y[1:k]$ and zero if $x[1:k] \neq y[1:k]$, also the similarity of the end of the truncated match is 1 if $x[k:n] = y[k:n]$ and zero if $x[k:n] \neq y[k:n]$. And the similarity of encoded object is 1 if $encode(x) = encode(y)$ and 0 if $encode(x) \neq encode(y)$ [19].

2.4.2.1 Hamming distance

Hamming distance is defined as the number of positions in which two equal length strings differ. It can be understood as minimum number of substitutions needed to change one string into another or minimum number of errors that could transform one string into the other. This method is used mostly for binary numbers and to measure communication errors. When used within binary numbers it can be counted as number of 1s in XOR of the numbers [13].

For example, the Hamming distance of strings ABC and CBA is 2 as only 1 character matches and 2 would need to be substituted.

2.4.2.2 Edit distance

Edit distance can be understood as minimal number of edits required to transform one string into the other. This method compares strings based on individual characters. As valid edit steps there are counted Insert, Delete and Replace operation, where each edit has a cost of 1. Also as an alternative the smallest edit cost can be counted. Its script is based on dynamic programming algorithm. This distance can also be known as Levenshtein distance [15].

Edit distance of strings ABC and CBA is 2 as 2 positions would need to be edited.

2.4.3 Token-Based methods

Main idea of this method is that every object can be separated into tokens using some kind of a separator. For example space, hyphen, punctuation or special character can be used as a separator in strings. Which is also a problem of this algorithm as lot of both hyphenated and non-hyphenated words are common, the same as words with apostrophes, numbers or periods. Another solution is to split strings into shorter sub-strings of length n and thus creating n-grams [16].

2.4.4 Hybrid methods

Hybrid methods are usually combination of some of previously mentioned methods as presented in [17].

2.4.4.1 Monge-Elkan

Monge and Elkan proposed this simple but effective method for measuring the similarity between two strings containing multiple tokens with the use of internal similarity function capable of measuring the similarity between two tokens a and b. Imagine two texts A and B with their respective number of tokens $|A|$ and $|B|$. The algorithm measures the average similarity of the values between pairs of more similar tokens within texts A and B. The formula is: $ME(A,B) = \frac{1}{|A|}\sum_{i=1}^{|A|} max(sim(A_i, B_j))_{i=1}^{|B|}$, where $sim(a,b)$ is internal similarity function.

2.4.4.2 Extended Jaccard Similarity

If strings contain multiple words then choose words as tokens and use internal similarity function to calculate similarity between sets of tokens as the ratio of the number of shared attributes (AND operator) and the number possessed by OR operator.

2.4.4.3 Tf-idf

Term frequency (tf) weight measures importance of term in a document and inverse document frequency (idf) measures importance in collection (this reflects the amount of information carried by the document). These two measures are then multiplied with some heuristic modifications.

2.4.5 Similarity measures on arrays

When measuring a similarity in 2D array one can use Aho-Corasick's multistring searching algorithm [41] which uses a finite automaton or there can be used an algorithm by Baeza-Yates and Régnier [42] which uses Aho-Corasick's algorithm in every row and Knuth-Morris-Pratt algorithm on every column, searching for the row indices of the pattern. If there is only one pattern to search for there exists optimal algorithm from Kärkkäinen and Ukkonen who improved an algorithm from Baeza-Yates and Régnier so it achieves $O(n^2 \log_\sigma(m)/m^2)$ where σ is the size of an alphabet, n is the size of a text and m is the length of a pattern [52].

On the other hand, when searching for higher number of patterns (thousands) in one string and allowing maximum of one mistake Muth and Manber proposed a solution in [54].

If tasked with probabilistic 2D pattern matching algorithms are proposed by [43] but running time ($O(n^2)$) makes it unusable in practice. Its improvement was introduced by [44] where are two versions, one using Knuth-Morris-Pratt algorithm and second with a variant of the Boyer-Moore algorithm.

If the pattern can be scaled (i.e pattern of a picture appears at a different size), there can be used the work of [47] and [48]. And if it can be rotated some important studies are: [49, 50, 51].

2.4.5.1 KS

Krithivasan and Sitalakshmi (KS) defined the edit distance in two dimensions as the sum of the edit distance of the corresponding row images. Using this model they search a subimage of size $m \times m$ in a large image of size $n \times n$

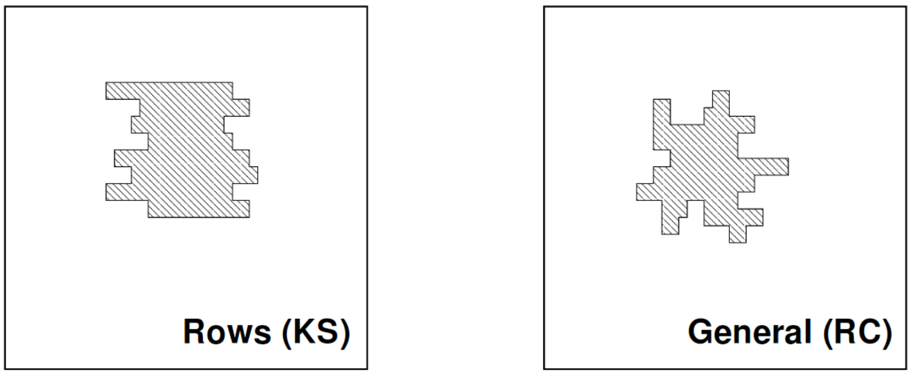


Figure 2.4: KS and RC error models [1]

using a generalization of the classical one-dimensional edit distance algorithm. The KS model can be extended into more than two dimensions, but it allows errors along one of the dimensions only [1]. An example can be seen in the figure 2.4.

2.4.5.2 R,C

In this model, each row (column) is treated as a single string which is compared to other rows using the one-dimensional edit distance, but whole rows can be inserted and deleted as well. Generalization of R,C model is RC model where errors can occur along rows or columns at any time. This model is much more robust and useful for more applications. Example of the RC model is in the figure 2.4 [1].

2.4.5.3 L

If it is needed to change both the row and column at the same time (for example when scaling) there could be used the L-shape for decomposition of the subimage and thus have the same extensions for rows and columns. When comparing L-shapes they are seen as two one-dimensional strings. L-shape string consists of the first (left) j elements of the i-th row and the first (top) $i - 1$ elements of the j-th column.

This idea was first presented by Raffaele Giancarlo [53] where it is used for building of Lsuffix tree with the use of Lstrings which can only be used for square matrices of text. Giancarlo divides the application of his solution into two sections: dictionary matching algorithms and pattern retrieval algorithms.

2.4.5.4 All

This model uses both decompositions at the same time (RC and L) computing the minimal value of all possible cases. It is easy to show that $KS(A, B) \geq$

$R(A,B) \geq RC(A,B) \geq All(A,B)$ and that $L(A,B) \geq All(A,B)$ because each case is a subset of the next. On the other hand, there are cases where $RC(A,B)$ will be less than $L(A,B)$ and vice versa [1].

2.4.5.5 Vector space model

This is an algebraic model using vectors as representation instead of arrays. It is commonly used for handling of text documents in the situations like information filtering, information retrieval and indexing.

The procedure of this model can be split into three stages. First phase is document indexing when the words that do not describe the content are removed (in English words like: the, is) so the document will be represented only by words bearing content. Second phase is term weighting which is based entirely on single term statistics. Term weighting has three main factors and that is: term frequency factor, collection frequency and length normalization factor. These factors are then multiplied together to make the resulting weight. Third stage is the final computation of similarity coefficient which is usually determined by the cosine coefficient by measuring the angle between the document vector and the query vector. Other usable measures can be Jaccard and Dice coefficient.

2.4.6 Domain-Dependent

Among some special similarity measures belong methods processing only one specific domain and thus the name domain-dependent measures. Widely used methods for handling this problem can be divided into categories based on the field like for example: phonetic methods, time and space methods, numerical comparison and binary vectors.

Analysis of two examples is given in the sections below but this work will focus mainly on commonly used measures for arrays. However, if the data for these measures are correctly preprocessed they can be used on arrays.

2.4.6.1 Time and space comparison

This is a very wide field of work mainly because the diversity of categorical data one can use for clustering, classification or any other case in need of some distance measure. To get similarity of time and space objects numerical methods are used to calculate differences in days. In other cases data can be converted (dates of birth into age) and then relevant methods can be used on these new attributes. When approached with geographic locations its similarity is estimated by using the great circle distance which is the shortest distance between two points along the great circle that contains both points. Popular method to compute the great circle distance is the Haversine formula. For time series data such as EEG there is a method for choosing a time interval and compare values of both the time series after the lapse of each interval.

This has several drawbacks because of the fixed interval if the series get out of sync this method can result in exaggerated distance estimates. To avoid this problem there is possible use of Dynamic Time Warping method which is robust enough to speed variations and computes more realistic distance estimate by using recursive formulation. In the list below there are measures useful for categorical data:

- Overlap - Measures number of attributes that match both data instances. Its range is $[0, 1]$ with 0 occurring when there is no match and vice versa.
- Goodall - Attempts to normalize the similarity between two objects. Assigns higher value to a match which is less frequent and then tries to combine similarities based on attribute dependencies. This is very computationally expensive.
- Gambaryan - gives higher value to matches closer to being half similar. The measure is close to Shannon entropy.
- Eskin - Measure assigns higher value to mismatches from attributes with many values.
- Inverse Occurrence Frequency (IOF) - Assigns lower value to mismatches on values with higher frequency. In contrary to inverse document frequency it is computed not on binary matrix but directly on categorical data.
- Occurrence Frequency - Uses opposite weighting compared with IOF (i.e. less frequent mismatches are given lower value)
- Burnaby - Assign low value to mismatches on rare values and vice versa. This method is based on arguments from information theory.
- Lin - Assign higher weight to matches on frequent values and conversely with lower values.
- Smirnov - This measure is based on probability theory based on value's frequency but also the distribution of other values. It means that the value is high for a low frequency of matches and vice versa.
- Anderberg - Assign higher similarity to rare matches, because they should be more important, and low similarity to rare mismatches. This measure cannot be written in equation form.

2.5 SciDB

Nonexistence of usable scientific database was the main reason of the creation of the SciDB. It was created to fulfill requirements given by scientists. Between the main requirements was to be able to contain multi-petabyte amounts of data, the preponderance of the data to be arrays, to be able to perform complex analytics, to have open source code, data can never be overwritten, it needs to be able to do provenance (trace backward), must work with uncertainty (because data comes with error bars) and must implement version control. This system was developed in C++ and supports Linux systems.

2.5.1 Data model

Because of these requirements the native data model chosen was array data where objects are N-dimensional array and not tables. Arrays with integer dimensions are divided into storage chunks which are composed of a stride in each dimension. Also, arrays are uniform, which means that all cells in a given array have the same collection of values [45].

This database supports schema migration, so attributes can be promoted to dimensions and dimensions can be deprecated to attributes. These operations can be performed by the transform command which pushes all dimension values up one to make a slot for new data. It can also reshape and array, flip dimensions for attributes and transform one or more dimensions into polar coordinates.

2.5.2 Language

SciDB was developed to work with functional and SQL-like query language. The functional language is called AFL (array functional language) and the SQL-like is called AQL (array query language) and is then compiled into AFL. Array query language has SQL like construction and is able to compute linear algebra queries. It also contains loose coupling model for analytics which in this case is ScaLAPACK running alongside SciDB performing matrix operations.

2.5.3 Data storage

Arrays are stored on disk in fixed-sized chunks with extra storage chunks where the “not valid” cells are. These chunks are heavily encoded. Chunks are distributed with the use of hashing, range partitioning or a block-cyclic algorithm. It is usually a two level chunk/tile scheme where chunks are split into tiles. The chunking strategy is supporting multidimensional queries in a straightforward way i.e. chunking automatically provides an index, however this suffers from skew problems (e.g. think of density of people living in Manhattan and village). This problem can be solved by creating super-chunks from sparse chunks or by creating variable sized chunks. Because of this the database supports arbitrary number of splits to keep the chunk size below the threshold.

Because of the need to never discard data even when wrong, the storage strategy was chosen as no overwrite. There is a special dimension for each array version. This is managed by special integer counter which begins at zero and increases with each update until array is destroyed. Need to trace backwards is arranged by backward deltas which contain a chain of references to previous versions. When in need of inserting and updating certain cell the database will put new values at the appropriate version in the array while delete operation will simply put “not valid” label to the cell. Version can be tied to wall-clock or special version names.

Design of the stored data was created as shared nothing which enables running local query in parallel across some collection of nodes followed by scatter-gather data shuffle to rearrange data for the next collection of local operations. But chunks can overlap by a user specified amount to support parallel execution of nearest neighbour, feature detection and so on.

Specification of binary file format used for chunk storage is given in the appendix C.

2.5.4 Query processing

Query processing is constrained by three tenets. First of them is to aim for parallelism in all operations with as little data movement as possible. SciDB is fundamentally focused on providing the best response time for AQL which can be helped by redistributing data. It tries to optimize the query parse tree by examining the operations that commute (cheaper of them are pushed down the tree) and examining blocking operations. This operations requires redistribution or cannot be pipelined from the previous operation which means they require a temporary array to be constructed. Second tenet is that incremental optimizers have more accurate size information and can use this to construct better query plans. Optimizer in this case is incremental and it picks the best choice for the first sub-tree to execute and not until it is done it chooses the second sub-tree and so on. Third tenet is to use a cost-based optimizer. SciDB tries to perform simple cost based plan evaluations but it plans only sub-trees. The optimizer provides these steps until there are no more sub-plans: choose and optimize next sub-plan, reshuffle data if required, execute a sub-plan in parallel on a collection of local nodes and collect size information from each local node.

As to the provenance task. The database admin is allowed to specify exact amount of space he is willing to allocate for the provenance data [46].

For more detailed information about SciDB queries processing see appendix E.

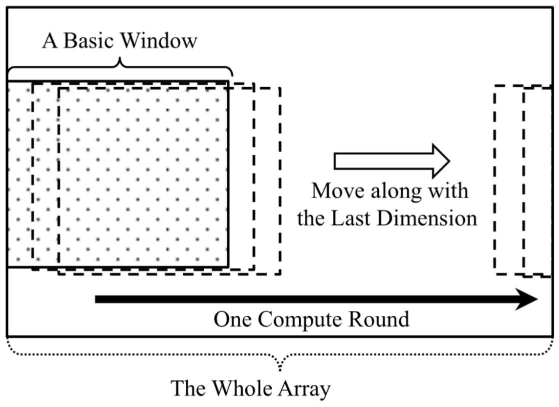


Figure 2.5: Example of a sliding window [2].

2.5.5 Use case

Between most common use cases of this database belong satellite imagery, astronomy applications, genomics and commercial applications.

Everyone can create their own queries because this database supports user defined data types and functions, but with specified amount of overlap. It can also work with uncertain cell values, but not uncertain dimensions and in-situ data, so you can use SciDB without loading your data.

2.5.6 Window aggregation

Window aggregate executes aggregate operators over a sliding window (sub-array), as seen in the figure 2.5. It has five types of operators: MIN, MAX, SUM, average and percentile. Its computation scheme is incremental and has two stages. It tries to maintain intermediate aggregate results of the current area and reuse them. Inside the staged it moves basic window in the first $N - 1$ dimensions, then it moves the window along with the last dimension and incrementally compute the aggregate result for each window. After all windows derived from the basic one are processed it is completed and it returns to the first step until all computation rounds of the basic windows are processed.

Sum and average operators are very similar as average uses sum and divides it by number of the elements. Because of this there will be only sum operator explained. This operator reuses SUM computed in previous windows by creating list structure (SUM-list) which contains sum values of every window unit. It generates basic window, computes its sum and initialize SUM-list. Then it moves the window along with the last dimension with one step on one window unit, scans it and calculates the SUM and updates the SUM-list. Then it proceeds forward and continue to move the basic window down to

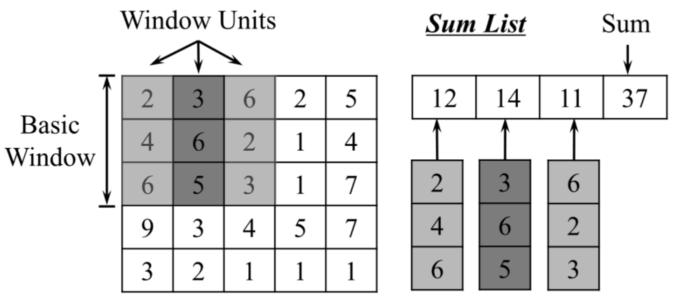


Figure 2.6: Initialization of SUM window [2].

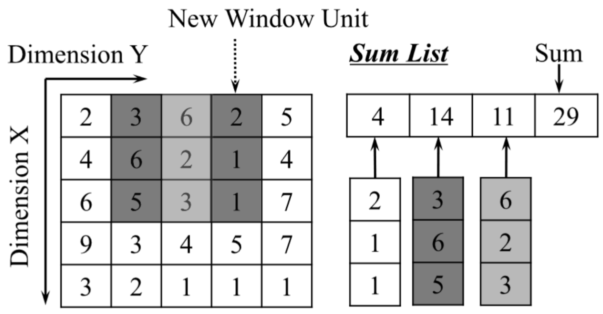


Figure 2.7: Processing of a new unit and updating sum-list [2].

obtain new ones. An example of creating and moving the SUM window can be seen in figures 2.6 and 2.7.

MIN and MAX operators are handled by creating a heap data structure. The operator will generate basic window and scan window units in it and compute MIN/MAX for each one while inserting them into MIN/MAX heap. Then it moves the window along with the last dimension, obtains new window, calculates new MIN/MAX and insert it into MIN/MAX heap again. Finally it checks the heap root and removes it if its corresponding window unit is no longer present in the current window (this step is repeated until it corresponds). Proceed forward and calculate MIN of all windows derived while continuing to move the basic window down.

Percentile operator is a bit different from operators above. it is computed as $n = (\frac{P}{100}) * N + \frac{1}{2}$ and its structure is Self balancing binary search tree (SBST). Steps are: generate basic window, initialize SBST, insert all the values of the basic window into SBST, move the window forward along with the last dimension, insert new elements into SBST and delete old ones (i.e. ones not longer present in the current window). Then move forward and proceed to new basic window [2].

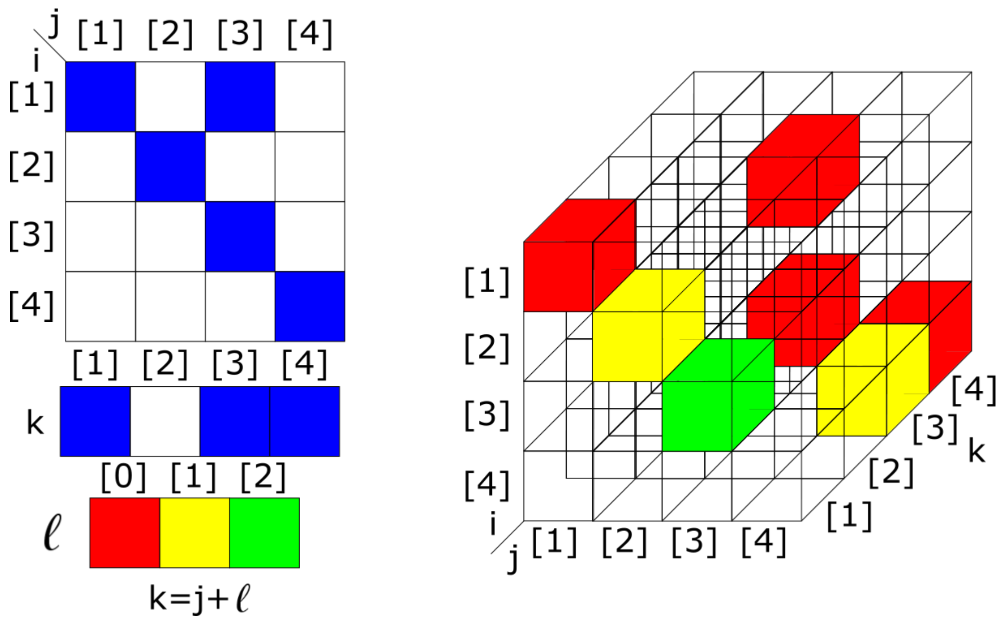


Figure 2.8: Similarity join of 2 different arrays, first 2 dimensional, second 1 dimensional with condition array ℓ. Result is in the 3 dimensional cube.

2.5.7 Similarity join

This join can be used on array data model which means a model where multidimensional array is defined by a set of dimensions and a set of attributes. Each dimension is a finite totally ordered discrete set and all cells in a given array have the same type. It must allow chunking and have shared-nothing architecture. Array join is a join of dimensions and union of attributes. The goal of the similarity join is to find all the pairs of points such that the distance between them is smaller than some Epsilon. When applied on arrays the result dimension is union of joined arrays dimensions, result cells are the ones satisfying the given condition ($k = j + l$ in the figure), this can handle arrays with various dimensionality. Most general algorithm used for this is nested loop join [30].

Chapter 3

Multidimensional Approximate Pattern Matching

As was already stated in the problem specification 2.1 to successfully solve this problem, the user needs to specify certain pattern that will be searched for in the data. There are two options on how to look for the pattern i.e. with or without allowance of errors. According to this condition the problem is said to be approximate pattern matching when errors are allowed and exact vice versa.

Solution of this problem can be used in the field of text searching, pattern recognition or computational biology.

3.1 Existing Solutions

First and easiest approach is to compute the Edit distance between the pattern and data but it can be used only when the problem consists of one dimension and it computes exact pattern match. Another solution (this time for approximate match) can be performed by dynamic programming which tries to find all segments whose edit distance to pattern is maximally k, where k is greater than zero and lesser then complete match. This solution can be improved by filters.

When the problem consists of two dimensions there can be used a variety of methods. Subset of these solutions works only for string data and the main algorithms were developed by: Bird and Baker, Amir and Landau, Zhu and Takaoka, Kärkkäinen and Ukkonen.

Scientists Krithivasan and Sitalakshmi developed The KS model which given two images of the same size, the edit distance is the sum of the edit distance of the corresponding row images. When used for approximate matching the sub-image of size $M \times M$ is searched into a large image of size $N \times N$, where they are using a generalization of the classical one-dimensional algorithm.

When fast searching under the KS model 2.4.5.1 patterns and texts are rectangular. Special distance measure allows errors along rows but not along columns. Variations of this model are Exact Partitioning which is best for larger patterns, Superimposed Automata which is best for small patterns, Counting is best for small patterns and One error. There are two useful functions defined for this model. First of them is $C(m, k, r)$ function which defines cost per text character to search r patterns of length m with k errors, and second is $L(m, r)$ function that says the value for k/m where the one dimensional algorithm does not work any more.

Another option to solve this problem is R model where each row is treated as a single string which is compared to other rows using one dimensional algorithm. Whole rows can be inserted and deleted. Modification of this model is RC model. If this algorithms encounter more than two dimensional problem it can be functional only if it has two hypercubes to compare which are reasonably close in size.

When tasked with a string pattern matching, various methods can be used for improving of the comparisons time, like for example Knuth–Morris–Pratt algorithm described in the section 3.3.

3.2 Multidimensional Pattern Matching Algorithms

To search across multiple dimension there are various algorithms to use, four of them are mentioned here.

First of them is Exact Multidimensional Pattern Matching which divides the space along one dimension to obtain patterns of one dimension less. Then it searches all sub-patterns in each of the dimensional subtexts.

Second one is Fast filter for multidimensional approximate searching. This algorithm splits pattern across every dimension and then it searches sub-patterns the same way as exact solution, but all solutions found are checked by using the dynamic programming.

Third is a stricter filter which says that whenever a piece appears the neighborhood can be checked. Its phase of finding possible solution positions is the same as in the previous algorithm but before checking the neighborhood with the use of dynamic programming the phase known as *preverification* happens, in this phase all pieces of the pattern are checked one by one to see if they match where they should and only when finding sufficient number of pieces (i.e. size of the pattern minus errors that can happen along every dimension) that fit exactly, then dynamic check can happen.

Last algorithm presented is focused on adapting the filter to simpler distances [1].

These algorithms were not implemented by anyone yet, but this thesis provides detailed description of solutions in the Implementation chapter.

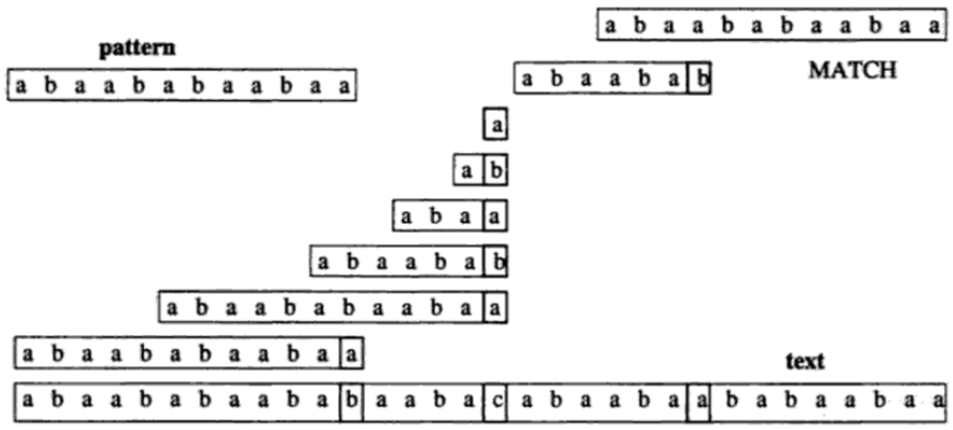

pattern
a b a a b a b a a b a a
MATCH
text
a b a b a b a b a a b a b a a b a c a b a a b a a b a b a a b a a

Figure 3.1: Example of matching of KMP algorithm [23].

3.3 Knuth–Morris–Pratt Algorithm

This algorithm was first presented in 1977 by scientists Knuth, Morris and Pratt [63]. In their work they describe new algorithm usable for a string searching. The main idea of it is to remember for each position of the pattern P the length of the longest suffix P which matches to the prefix of the P. Unlike the naive algorithm that after every mismatch shifts only by one position, which causes some parts of the data to be read again, KMP notices that it is possible to move by more than one position, this information is gained when comparing previous characters. To be able to remember all the extra information the algorithm uses is stored in a pre-computed table. Example of the steps KMP algorithm is shown in the figure 3.1.

3.4 Machine Learning Based Methods

Main use of Machine Learning methods is in situations when in need of data analysis because people can often make mistakes, which make it difficult to find certain solutions. These methods can be successfully applied in these situations and improve both the system efficiency and machine design.

By working with the same set of features (here attributes) these algorithms can be even used when trying discover new data dependencies, classes and patterns [71].

3.4.1 KNN

KNN or k Nearest Neighbors algorithm is very simple method of classifying items based on their distance to the pattern. The main goal of this procedure is to find *k* nearest neighbors to the wanted item and classify it by the major class of its neighbors. This method is widely used since the 1970 and its one of the first and effective methods used for data mining.

First KNN algorithm was based on the Euclidean distance between the items, but it is possible to use other similarity measures (i.e they must meet the 4 basic criteria). Among the most widely used metrics belong: Euclidean distance, Kullback-Leibler distance, chess distance, Manhattan distance and Minkowski distance.

This approach is also known as similarity search, proximity search or close item search. Alternative way is introduced in R-near neighbor search which is using a fixed radius. This algorithm tries to find all items that are within the distance R [56].

One of the usable methods for solving this problem is using the Locality Sensitive Hashing.

3.4.2 Kernel based methods

Most of the machine learning based algorithms was developed for the usage in linear spaces while real word data often require nonlinear methods for dependency detection. By using kernel methods the dimensionality of the space can be reduced to a dot product. In this case the product acts as a similarity function between the pairs of data. The existence of a product like this enables these methods to act without computing the coordinates of the data but only computing the similarity between all of their pairs. Another advantage is that it is usually cheap to compute.

Among the algorithms using this method belong Support Vector Machines, Principal Component Analysis, Ridge regression, Linear Adaptive Filters and others. All linear models can be changed into non-linear by using kernel functions instead of its predictors [58, 59].

All of these methods can be converted to work when using array data structure as mentioned in this paper [10]. Also this paper [18] explains how to define kernels on structured objects like strings. The main idea is to define a kernel between parts of the object by using a suitable similarity measure. Because of this, the kernel function can be designed even for graphs.

Chapter 4

Implementation

This chapter will discuss the implementation. It presents all implemented solutions ranging from brute force exact pattern matching 4.3.1.1 and approximate pattern matching 4.3.2.1 up to exact solution using skipping dimensions 4.3.1.2, approximate algorithm using dynamic check 4.3.2.2, algorithm using preverification phases 4.3.2.3 and solution using various hash functions 4.3.2.4.

Primary languages were C++ and Python. For compilation of *.cpp programs user needs at least compiler of version 2011 (flag `-std=c++11`). All Python scripts were run using version 3.0. A *numpy* library is also needed. A *Makefile* was created in order to simplify the C++ compilation.

Libraries used in C++ were iostream, fstream, string, vector, math.h and chrono which are all included in the C++11 standard.

4.1 Input Data

A large part of the database search revolves around used data structures, so this section will focus on the specifications of used formats and other properties. For this task three Python scripts were implemented and their usage and descriptions are in the sections below focused on generating of data 4.1.3, data filtering and chunking 4.1.4, all of these scripts expect CSV data as input format, because it is one of the most common data storage format. However first section 4.1.2 is explaining input data format which the scripts can process.

4.1.1 Real datasets

Implementations were also tested using some real datasets [1]. However there is a need of preprocessing of the data, as they are missing proper headers. The chunking can be done by the provided script. After these operations the datasets can be used the same way as testing data.

[1]Provided by `archive.ics.uci.edu/ml/datasets`

4.1.2 Format

Data format (file extension) that can be successfully processed is of types CSV and BIN where CSV data must have a header in the format:

- 0:DIMX$_0$, 1:DIMX$_1$, ..., N:DIMX$_N$, ATTR1:TYPE, ATTR2:TYPE, ..., ATTRL:TYPE

In this format $(0, N)$ is an order of the dimension whilst $(X_0, X_1, \ldots, X_n)$ is a number of unique values that are in this dimension. $(ATTR1, \ldots, ATTRL)$ specifies the name of the attribute and TYPE parameter contains the type of the values in the attribute, only one type per attribute is allowed.

Various examples of this format can be:

- 0:DIM5, 1:DIM5, 2:DIM5, TIME:FLOAT - this data has three dimensions where each of them contains 5 values and one attribute named TIME using type float.

- PLACE:DIM3,TYPE:DIM4,PRIZE:INT - this could be a specification of races where there are three different places, four different types and the prize for the winner is specified in attribute PRIZE.

When the user needs binary data, another script can be used to create chunked and binary data from the original CSV file. This script is specified in the section called Chunking 4.1.4.

Following this specific header are usual CSV data. Be aware that the data can be sparse, which means there can be missing cells or attributes.

4.1.3 Generating Data

Data generating is taken care of by two scripts where both of them generate hyper-cubic data (this is a simplification for the analysis of the algorithms, although algorithms can be used for different lengths of dimensions and patterns). First of them will generate data where no cell is missing or NULL value. When in need of sparse data, second script named can generate them with the help of multidimensional Gaussian functions.

The actual generating is based on random generated three Gaussian distributions with the centers set in three different positions

An example of generating data in two dimensional space is in the figure 4.1 where the ellipses G_1, G_2 and G_3 represents where the data are going to be generated, S_1, S_2 and S_3 are the centers of the ellipses.

For pure testing purpose there is also a script which is prepared for creating patterns at specific positions of the data.

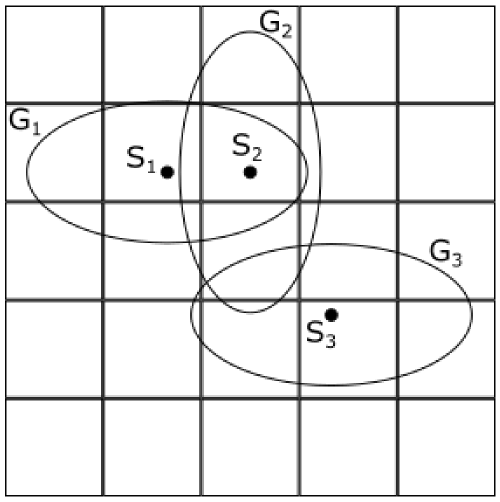


Figure 4.1: Example of the space filling with Gaussian functions. [1]

4.1.4 Chunking

All tested data are represented in binary format and chunked into files which represent a set number of cells based on the dimension sizes.

Attribute values are added into the relevant file, which is determined by the dimension values. Created files have their names same as the input file with the dimension numbers added to signify which data are stored in them. For example, if the input data are two dimensional, there are two possible options to perform chunking based on the size of the data and maximal chunk size. First, if the data are small enough, the chunking would use only one bin file. Otherwise, for each "row" a single bin file would be created to easily determine what data are stored in them.

As was already specified SciDB has its own binary format and the chunking uses exactly the same format. This means there is always one special file with meta information about the chunked binary files. In meta file there can be found header of the original data as this file is not binary encoded. There is also a mask file that specifies if a cell is in the data or not by using one bit per each cell (0 means the cell is not in the data and vice versa). Other files created by the script are of a binary type that contain a sequence of cell values. For the sake of testing attributes were only of integer type, but implementation can also work with other values.

The main advantage of the chunking is that the algorithm does not need to hold all data in the memory and instead read only currently processed sections.

4.2 Output Format

After successfully running the executables in the terminal window the program will write in the standard output stream. First it will write Finding, meaning that the part of the program that searches for solutions started and all initial settings and reads are done. When finished with find phase program will output the time it took to find the results and after this it will write coordinates of the found solutions. They will be written in the form of list of all possible dimension coordinates where the desired pattern can be found while not violating the constraint about maximum number of errors.

Usual output would contain an array of found solutions, however as this work is concerned about the process of finding, the result serves mostly a confirmation role. If the algorithms were used in real system, it would be simple to modify the output to suit the system needs.

4.3 Pattern Matching

In this section eight implemented solutions of pattern matching are presented. First part 4.3.1 is focused on two solutions used for solving of exact pattern matching, and second part 4.3.2 is dedicated to approximate algorithms.

4.3.1 Exact pattern matching

Every other algorithm is a type of pattern matching. This section will present implemented exact solutions. Exact pattern match means there are no errors and the pattern is present in the data without changes. Two solutions were implemented as reference solutions. First of them is solution using naive brute force algorithm. Second is algorithm by Navarro which skips first dimensions and thus have smaller both patterns and data to compare.

4.3.1.1 Brute Force

As the name suggests this algorithm goes through every position in data and starts to compare the pattern from the "top-left" corner until it encounters error. This solution is very simple, on the other hand there is obviously no possibility not to find every position of the pattern.

Just to clarify things, the algorithm iterates over every dimension and in it over every value, thus the returned positions are sorted.

4.3.1.2 Baeza-Yates and Navarro modification

This algorithm was proposed by Baeza-Yates and Navarro [1]. Its process is similar to the brute force algorithm, however one dimension which is present in both the pattern and the data is selected. While iterating through this dimension instead of taking steps of length 1, steps of length m are taken.

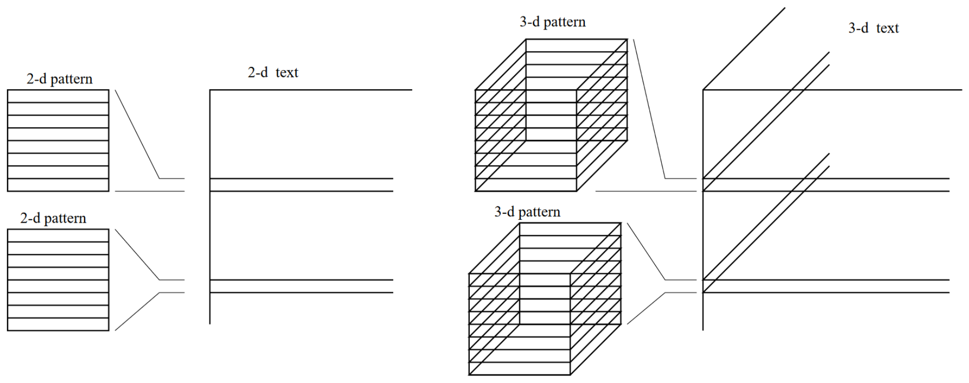


Figure 4.2: Illustration of comparison between pattern and data. [1]

Not to miss any possible solution, every "row" of pattern is compared with the given part of the data. This process is better illustrated in the figure 4.2.

Search cost in the worst case corresponds to verifying all text positions the same way as brute force algorithm, which gives $\mathcal{O}(rm^d n^d)$ where r is the number of searched patterns (in this case 1), m is the size of pattern in one dimension and n is the size of data in one dimension, for simplicity the algorithms will work only with square shaped data, d specifies the number of dimensions [1].

Considering that both pattern and data have all dimensions with the same sizes and the selected dimension is arbitrary, the first one is always used for easier implementation. This modification reuses parts of data instead of using every single cell.

4.3.2 Approximate pattern matching

Previous two solutions could us a simple identity function to recognize, whether the solution is valid or not. If an approximation is taken into account, the algorithm has to measure the difference between the pattern and the currently tested position. There exist several options to do that. In the next section all implemented variants are presented.

In every solution an input value k is added, which represents the maximal tolerated error.

4.3.2.1 Brute Force

First and again the simplest solutions to iterate over every part of the data and compare one coordination of the pattern of another until either whole pattern is checked or the k number of errors are found.

This method obviously always finds every valid solution so the number of False Negatives is 0. However it is linearly dependent on the selected error, so it is not suitable for large patterns with higher error range. On the other hand,

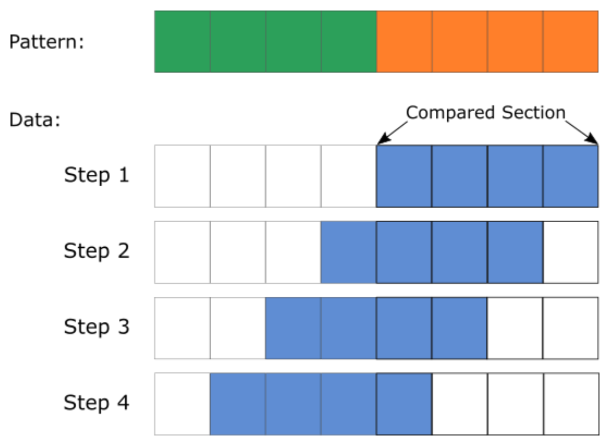


Figure 4.3: Example of sliding window.

the implementation is very simple again and it serves as a solid confirmation of the correctness of other solutions.

4.3.2.2 Fast Filter Searching

A less demanding solution was proposed again by Baeza-Yates and Navarro [1]. Its idea is similar to the modification made in the exact algorithm, however in this case an iteration in all dimensions present in the pattern are influenced.

The data are divided into sections, which are the same size as the pattern. The iteration through positions in dimensions not present in pattern is the same as in brute force algorithm. However in the other dimensions, both the section of data and the pattern is divided into the same number of parts and the comparison is made only between the “bottom-right” corner of data section and all pattern parts. Not to miss any possible positions, the compared part of the data has to be moved always by one cell part size - 1 times. Again, for better illustration a figure 4.3 is provided. In the picture length of the pattern is 10 and is split into two parts. Each part of the pattern is then compared with the “bottom-right” corner of the section and then with the parts moved by one until the compared part would not be overlapped with the first part as shown in the figure.

In current implementation all dimensions are split into m parts, except the last, which is split into j parts, where:

$$j = \lfloor k^{\frac{1}{d-1}} \rfloor + 1$$

this value is recommended, so it was not tampered with. This relatively small parts are compared by basic identity function and the positive results are remembered. The whole process usually returns more results than it should, because only a small parts of the pattern are matched. So further verification is needed.

In this case a dynamic programming method is utilized. To perform the dynamic programming check, multiple approaches are available. Completely implemented and tested was only Edit distance, with framework prepared in code for R, C and RC measures as specified in section 2.4. Following methods also use this step, so the dynamic programming was reused.

Usually the dynamic programming confirmation would be made right after finding each position. However considering that all implemented algorithm use chunked data and thus not every section of the pattern is necessarily in the memory, these stages are split and the whole data are searched through for the partial matches and after that the dynamic programming check is performed on all found positions.

When using the ideal j value, the total cost of search and verification is:

$$n^d k^{\frac{d}{d-1}} \Big(\frac{1}{m^{d-1} k^{\frac{1}{d-1}}} + \frac{1}{\sigma^{m/k^{1/(d-1)}}} + \frac{d! m^{2d}}{\sigma^{m^d / k^{d/(d-1)}}} \Big)$$

as stated in [1], where σ is the cardinality of the alphabet. This algorithm is then sublinear for the number of errors

$$k < (m/(d \log_\sigma m))^{d-1}$$

and it is not sublinear but still better than dynamic programming for

$$k \leq m^{d-1}/(2d \log_\sigma m)^{(d-1)/d}$$

Expected cost of verification the third addend while the elements before express the cost of the find phase. For more informations and explanation of course of action leading to this formula see [1].

Concerning number of False Positives and False Negatives, the verification phase should prevent the existence of False Positive results. This also applies to algorithms Stricter Filter and Hashed Stricter Filter explained in the next sections. However, there is a possibility of False Negatives because the algorithm requires strict match of a specific part, which in reality could be just the part with the allowed error.

4.3.2.3 Stricter Filter

A slightly modified version of previous method is called Stricter filter and was proposed in the same paper. The disadvantage of previous solution is, that if the first partial matching finds a large number of False Positives, the dynamic programming is then a bottleneck, because it is computationally expensive to perform.

Therefore, before the last stage a preverification step is added. It has the same goal as the dynamic programming, however is much simpler and thus much faster. It should filter a large portion of results, which will not be necessary to go through by the dynamic programming.

The idea is to check the found area cell by cell and remove it from solutions if more than k are found. The preverification can end before iterating through all cell if:

1. $k + 1$ errors are found, i.e. the position is not a solution.
2. Less than $k - q$ cells remain to compare, where q is the number of already found errors.

New search cost is then:

$$n^d \Big(\frac{j^{d-1}}{m^{d-1}} + \frac{j^d}{\sigma^{m/j}} + \frac{j^d jk}{\sigma^{(m/j)^d}} + \frac{\binom{j^d}{jk} d! m^{2d}}{\sigma^{m^d - km^d/j^d - 1}} \Big)$$

which is the sum of all steps in this algorithm: find phase (first two parts), preverification phase (third addend) and dynamic check phase (last addend).

4.3.2.4 Hashed Stricter Filter

All of the previous approaches were either the simple obvious solutions or previously proposed by Baeza-Yates and Navarro. The last solution is built upon the Stricter Filter and modifies one significant feature. The problem with Stricter Filter solution is, that there has to be the partial identity for arbitrary error value.

So instead of using the comparison of the actual sections of pattern with the data a hash function is implemented. Firstly, all parts of the pattern are hashed, so it is done only once. The same cannot be done with the whole data as only some parts should be hashed and the data are not loaded at the start as they are chunked in smaller files.

Three similar solutions are presented and each of them will be individually presented in its own paragraph below. The rest of the processes stays the same as in the Stricter Filter, so preverification and dynamic programming check is performed.

The assumption is, that the added hashing may increase the computation speed, because a lot shorter values are compared instead of whole sections. However the part size will have high impact on the improvement, because the higher the ratio between part size and hash size the bigger the improvement. Another possible outcome is to find even more partial results for preverification, because the the hash function will probably make some collisions.

The best case scenario thought is that the collisions will be for similar sections and thus possibly valid positions, which were disregarded by the Stricter Filter method. This is the reason for usage of SimHash, while the use of the other hash function is supported by the idea that when the data are created randomly there is a little chance for the orders of numbers to be repeated and thus creating less collisions.

The complexity of this algorithm is the almost the same as of Stricter Filter algorithm, but the find phase elements are reduced by the probability of the hash reusage.

Number of the created hashes for SimHash and LSB hash discussed in the sections below is:

- Pattern – $m^{d-1} * j$
- Data – $(\frac{n}{m})^{d-1} * \frac{m}{j}$

The number of False Negatives is similar as in the previous approximate solutions with the difference that hashes may return some previously not found positions due to the hash collisions.

4.3.2.4.1 Naive SimHash First implemented variant uses SimHash which is created by hashing attributes needed to compare (attributes present in the pattern) and then combining the partial hashes into one hash where each bit position contains 1 if there is more 1s than 0s in the same bit position of all partial hashes. The hash used for the partial hashing is simple identity as the testing data use only integer values with the highest value of 255, so it is basically an eight bit number. The actual hashing of data is performed every time a comparison is required between pattern and data. In the table 4.1 is an example of the SimHash, with the final value of 00101000.

i	Value	Bit value							
0	43	0	0	1	0	1	0	1	1
1	56	0	0	1	1	1	0	0	0
2	11	0	0	0	0	1	0	1	1
3	167	1	0	1	0	0	1	1	1
4	204	1	1	0	0	1	1	0	0
5	38	0	0	1	0	0	1	1	0
6	64	0	1	0	0	0	0	0	0
7	249	1	1	1	1	1	0	0	1
# 1s:		3	3	5	2	5	3	4	4
# 0s:		5	5	3	6	3	5	4	4
Hash:		0	0	1	0	1	0	0	0

Table 4.1: Example of Naive SimHash.

4.3.2.4.2 SimHash Second implementation varies only a slightly from the first one. The data are hashed using SimHash as well, but only once as they are read from the chunked file, and only the parts to be compared. This reduces the number of hash computations, however it requires another matrix to keep the hashes in memory. The main idea for this hash is to be able to reuse once computed hash values instead of creating them again.

4.3.2.4.3 LSB Hash Third solution uses a variation of the least significant bit hash (LSB), but the position of the LSB is moved after every step. This hash is constructed by using this formula: $h(x_i) = A_i \ \&_{bit} \ 2^i$, where A_i is the i-th value to be hashed, h is the the hash function and x_i is the i-th position of the hash. Main drawback of usage of this hash is that only one bit is taken from every value which means for small compared parts (smaller than eight) it will create higher number of collisions. On the other hand, for longer parts, the hash will actually be longer and thus may create less collisions with the drawback of longer comparison (but still not as long as without using any hash). In the table 4.2 is an example of the hash, with the final value of 10000111.

i	Value	Bit value	Hash
0	43	0010101**1**	1
1	56	001110**0**0	0
2	11	00001**0**11	0
3	167	1010**0**111	0
4	204	110**0**1100	0
5	38	00**1**00110	1
6	64	0**1**000000	1
7	249	**1**1111001	1
			10000111

Table 4.2: Example of Modified LSB hashing.

This explanation uses an assumption, that each cell contains only one integer attribute, in the actual implementation, the hashes for each attributes are concatenated.

4.4 Possible modifications

Some other modifications were considered and two of them were analysed, whether it could have a positive impact on already implemented solutions. First modification consist of changing the comparison between the section from the naive algorithm to KMP as the sliding part of the data may be considered as a longer array and the comparison could return only a position in the slide. However considering that the compared sizes are $\frac{m}{j}$ and $2\frac{m}{j} - 1$, and the non-repetitive type of data, the creation of the extra table as specified in 3.3 would add another space requirement and the speed increase would probably not occur, because mostly, the check ends after single value comparison. If these methods were used data, where a repetition of subsequences was expected and the part size was set to sufficient value, it may prove to be faster.

Second modification was to use hashing even for preverification phase. As the pattern was already hashed it seemed as a viable option. However only a very small portion of the data is hashed at the start of the preverification, as only a few sections are used in comparison and also some of the hashed parts were overwritten in cache. Creating hashes for all required parts of the data would take more time than the actual preverification in current form, so it was not used. If the data were for some reason hashed before, the idea could prove more plausible. For example, if some form of indexing was used.

CHAPTER 5

Results

This chapter will focus on presenting and discussing the results of all eight implementations (two for exact pattern matching and six for approximate pattern matching). First part 5.1 is focused on explaining all factors that the time could be dependent on. Next part 5.2 will discuss measured times and its dependency on size and density of the file, number of dimensions and number of pattern matches. For this task, eight testing files were generated using the Gaussian generating script 4.1.3. The last part 5.3 will focus on explaining memory usage and the chunking algorithm.

5.1 Problem dimensionality

Among the dimensions of the problem which could be worth measuring belong: size of the dataset and pattern, density of dataset, number of dimensions, number of errors allowed, algorithm used, size of the chunks and number of occurrences of the pattern in the data.

From all of these categories five were chosen for testing and they are: size of the dataset, density, number of dimensions, occurrences of the pattern and number of errors. For each of the categories, all implemented solutions were measured. For the sake of the testing eight files were created with specific values based on these categories. One of the files was called standard file and is referenced in the text also by the name standard file. This standard file was compared with all other files and its parameters were:

- size – 256 MB
- density – 50 %
- number of dimensions – 3
- occurrences of pattern – 0.1 %
- number of errors - 0, 4, 16, 64

5.2 Time complexity

Time measuring was done with the help of chrono C++ library. In the first figure 5.1 there can be seen times of all implemented algorithms achieved for the input file with standard parameters and standard pattern of size $16 \times 16 \times 16$.

Abbreviations on the x axis correspond with these algorithms:

- BE – Brute Force for exact pattern matching
- SE – Baeza-Yates and Navarro algorithm for exact pattern matching
- BA – Brute Force for approximate pattern matching
- FFS – Fast Filter Searching
- SF – Stricter Filter
- NSH – Naive SimHashed Stricter Filter
- SH – SimHash Stricter Filter
- LSB – LSB Hash Stricter Filter

Y axis units are always seconds.

Yellow color symbolizes the time needed for finishing of preverification and red color means the time of dynamic check.

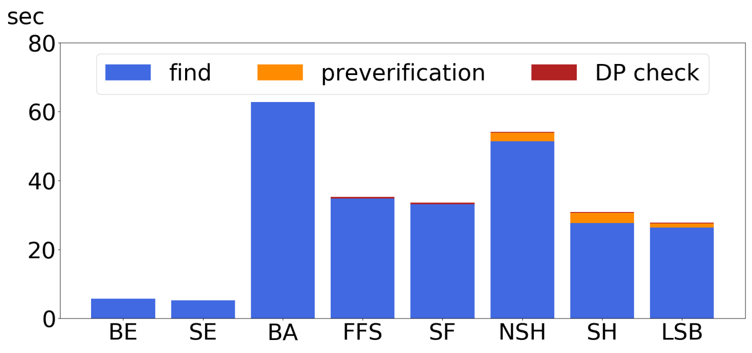


Figure 5.1: Computation time for all implemented algorithms using the standard input file.

As can be seen in the figure 5.1, solutions of exact approach achieve higher performance than approximate solutions. This is because whenever exact algorithms encounter a mistake they can dismiss this possible solution while approximate approach has to check until k errors are found. Because of this distinct difference exact algorithms are not considered in another graphs.

Dynamic check has such a low times because of the low number of pattern occurrences in the data and thus very quick verification. Preverification times are higher when using hashing algorithms because of the higher number of possible solutions found.

5.2.1 Dependency on data density

Measuring in this section was done using 3 different files. One of them is the standard file specified in the first section, other 2 files were different only in the density of generated data and its values were 100% and 2%. When the file is 100% dense it means there are no missing values, while with the 50% density on half of the values is present and similarly with 2% density.

In the figure 5.2, green color represents dense data, orange 50% density and blue 2% density. It is clearly visible that lesser the maximum error allowed the higher the dependency on the data density. Times achieved when allowing up to 4 errors is multiple times worse for the least dense data. This happens because when the density is low there is a lot of positions to check even when there are no data in them, which is caused by the incorrect data manipulation in the implementations, as they are checking each position even when there are continuously missing data. Because of the high sparsity of data, there is much more positions to check in the low dense file which dimensions were $656 \times 656 \times 656$ than in the sparser ones (the dense file dimensions were $256 \times 256 \times 256$ which worsens the time even more).

However there can be seen that solutions using hashes can reach better times than other algorithms especially in the data with low density. Reason for this is that non-hash algorithms can take a while to reach a position with no data, but when using second and third implementation the overall comparison is quicker because the value of the hash is NULL if some of its value is not in the data. Which means that a lot more positions can be skipped directly.

Solution using variant of LSB hash can be very successful for low errors allowed but rapidly worsens with growing error rate. This is why there is such a peak for error 16 and also its time is not used in the last graph for better scaling. In the last graph the brute solution is also omitted.

5.2.2 Dependency on dimensionality

Three different files were used based on the number of dimensions which were: two, three and four. In the figure 5.3 the green color belongs to the 2D file, orange is for the standard three dimensional file and blue shows four dimensions. It is clear that the bigger the number of dimensions the higher times. This happens because for more dimensions the algorithm gets into higher recursion levels than otherwise.

Second important thing to see in these graphs is that for low error rate and lower dimensionality hash algorithms are way worse than other non-brute

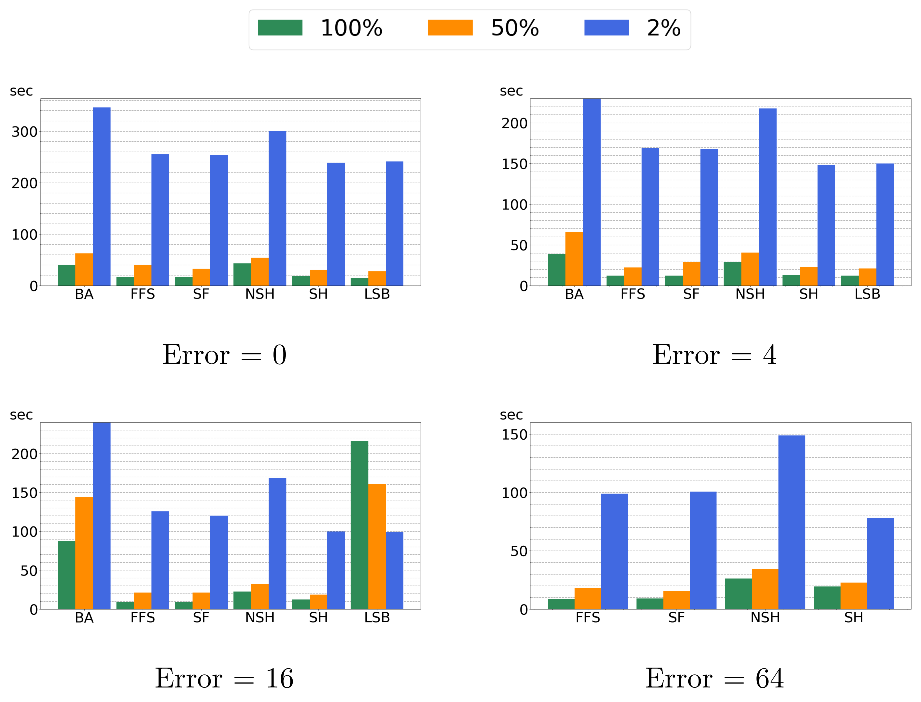


Figure 5.2: Comparison on varying density of data

solutions. This is caused by a high number of collisions and thus more expensive preverification. Because of the low data dimensionality there is only last dimension of data in the memory that will be hashed which means there is no reusage of already created hashes.

The higher the number of dimensions in chunks the better the reusage rate of hashing algorithms thus better scaling.

5.2.3 Dependency on frequency of pattern occurrence

Difference between input files were this time in the number of pattern occurrences. Standard file had the occurrence rate set to 0.1% while the other two had 1% and 0.01%. In the figure 5.4 there can be seen only a slight difference in the times of comparing algorithms. This difference is caused only by the preverification step and dynamic checks as they need to check more possible solutions thus lasting longer.

The big leap of LSB algorithm between error 4 and 16 is caused by the high number of collisions caused by the change of j value, which is discussed more in the section focusing on the dependency on error rate 5.2.5.

As there is not much else to see in this figure it is possible to notice the general differences between the used algorithms. Brute algorithm is by expectation the worst solution, then both Navaro and Baeza-Yates are very similar

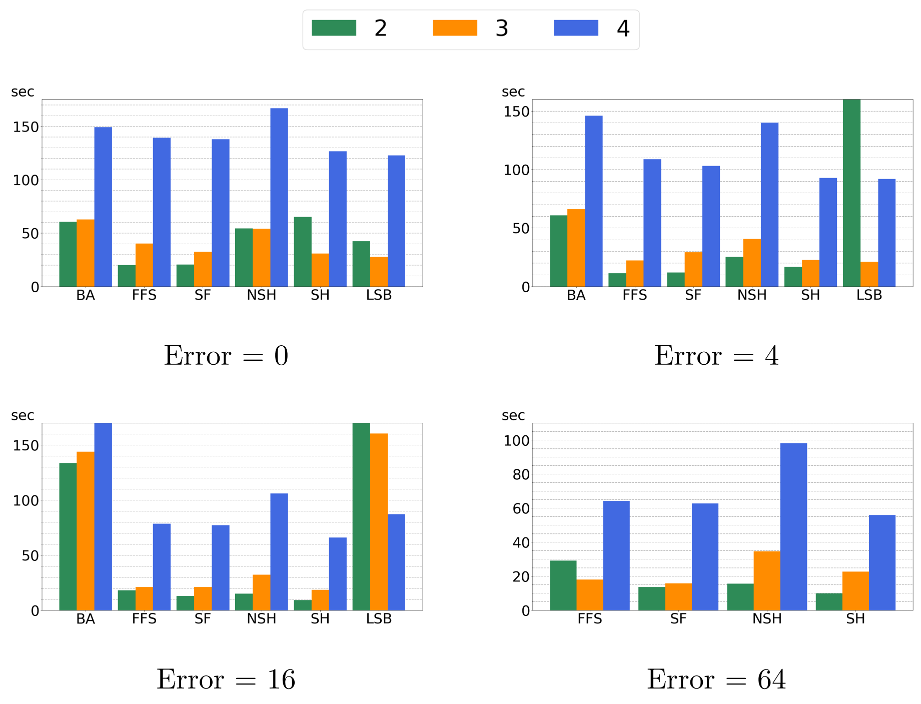


Figure 5.3: Comparison on varying number of dimensions.

because the find part does not find any false positive so the preverification is not necessary to perform. Then from the implemented solutions the first algorithm using hashing does the same number of comparisons, however the hashing function is more time consuming, on the other hand if the hashes are performed only once it significantly improves the find time while also increasing the preverification time as there are more possible solutions to check, but in the end the overall time is still a little lower unless the part size is too small for hashing to be worth it.

Stricter Filter algorithm is always slightly better than FFS because preverification step is quicker than dynamic check. When using LSB solution, its even more dependent on the part size because the hash length relies on it, so the quality of LSB lies mainly in lower error rates.

5.2.4 Dependency on size of dataset

In the figure 5.5 there can be seen comparison of standard (orange) file and file of the size 4 GB but measured twice, first with the pattern of size $64 \times 64 \times 64$ and second with the standard pattern (size $16 \times 16 \times 16$), the large pattern was selected so the ratio of pattern size and file size is more similar. Also these figures are displayed with logarithmic scale due to the large difference in time.

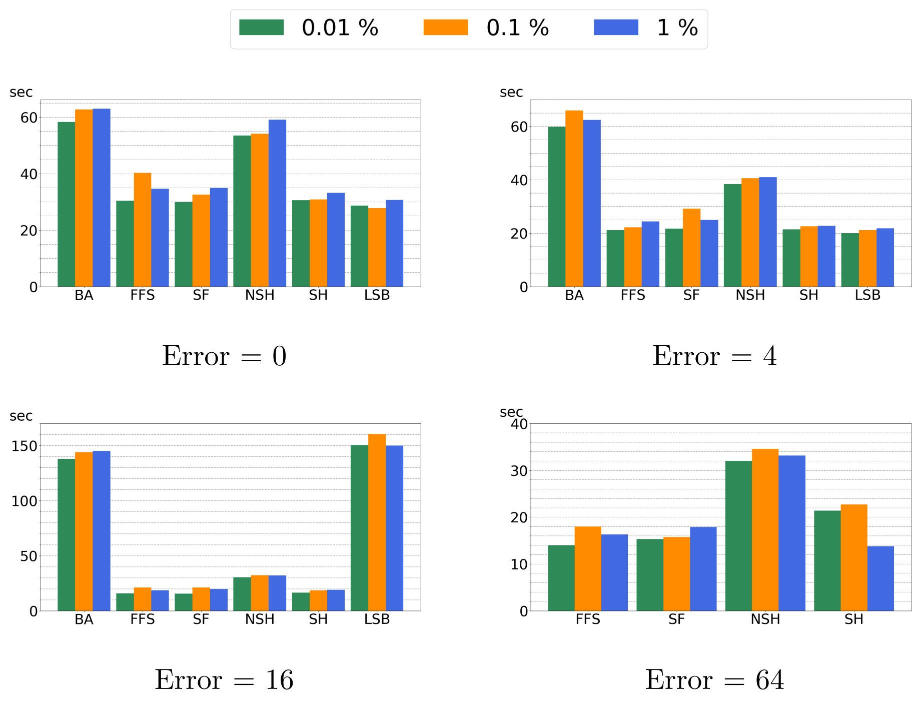


Figure 5.4: Comparison using varying frequency of pattern occurrence.

As the file contains approximately 18 times more positions to check the time is about 20 times longer for the bigger file. This is probably due to the increased need of handling the cache and also may be affected by the number of found partial solutions because the file contains just pseudo randomly generated data.

An effect of a difference in the pattern size, which was not tested by itself, is also visible in these figures. As expected it has only a slight effect on the final times and probably mostly due to inconsistencies in number of found positions due to the nature of the data.

5.2.5 Dependency on error rate

In all the figures mentioned in previous sections, there can be seen that concerning approximate brute force solution the higher the error rate the higher the times. This is caused by the course of action of the algorithm used, i.e. it needs to check more data before the maximum number of errors is found.

Both of the Navaro and Baeza-Yates algorithms act very similarly in a way that their run time decreases the bigger the error rate. This is caused by decreased find time as it needs to compare smaller parts. The drawback of this algorithms in this part could be in finding more solutions but this is not the case.

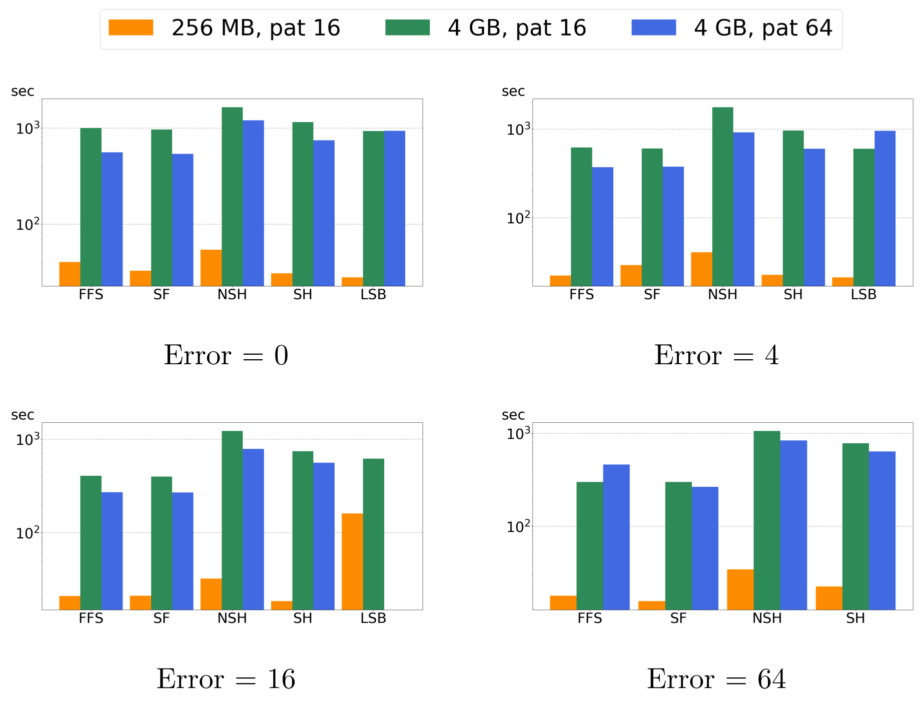


Figure 5.5: Test of various file and pattern size.

In the solutions using hashing the find parts behave the same as for previous solutions in term of increasing the speed however the number of returned possible solutions to preverify is considerably higher, so the overall speed of find parts increases while the preverification decreases with the higher error rate. In the LSB algorithm the number of collisions is so high that the preverification cost gradually outweighs every other part. This can be seen in the figures: 5.6 which shows the the change of computation time based on the error rate and figure 5.7 showing the difference between hashed and non-hashed find phase. The noticeable changes are caused by the change of j, where first change is when the j value changes to 2 then the change to 4 and then when $j = 8$. That is because there is still the same number of comparisons but smaller parts to compare.

Even though the j number is optimized for the FFS and SF algorithms the usage of hashing changes some dependencies and thus the change of the calculation of j could offer better scaling.

5.3 Memory usage

Implemented solutions uses a cache like memory system with the size of $C = max(64, m)$ where m is the size of the pattern. This means that

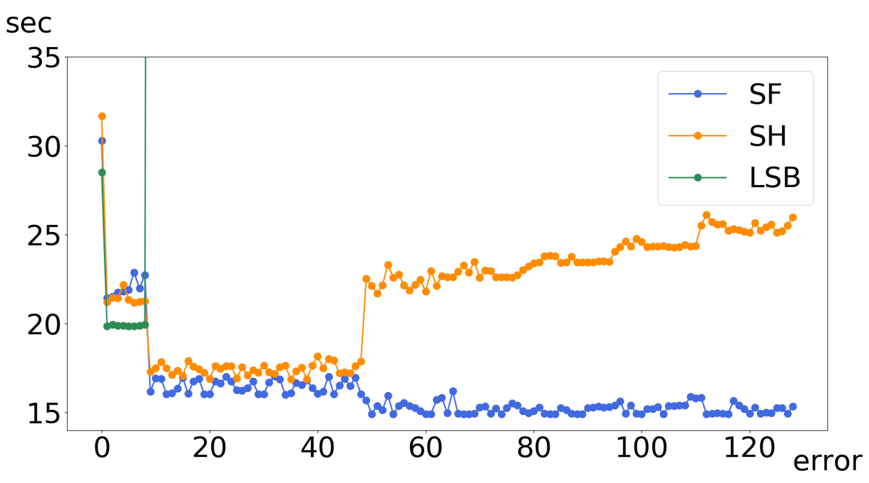


Figure 5.6: Dependency of computation time of selected algorithms on error rate using the standard input file .

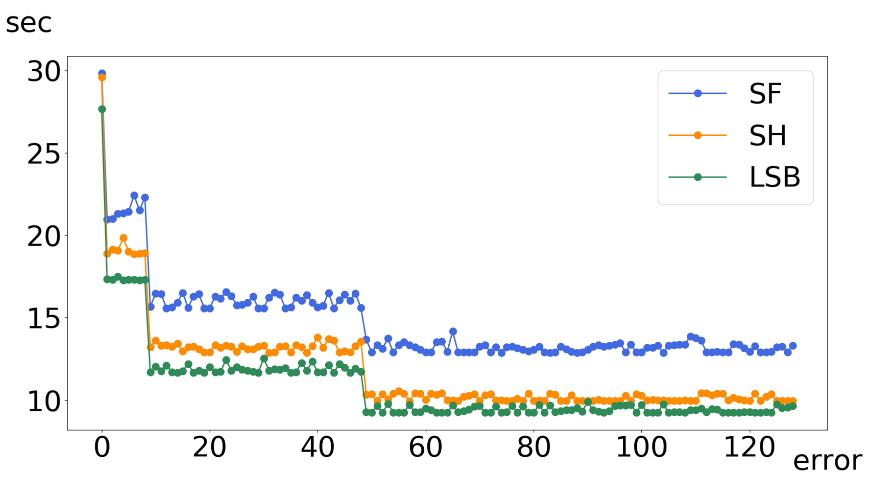


Figure 5.7: Computation time of only the find phase.

when the requested file is not in the memory it will be loaded into cache and if the cache is full the oldest data are discarded.

When taking brute solution as a base line that uses only cache and pattern memory, FFS and SF algorithms do not use any added memory except for dynamic check that needs only extra space for creating of matrix of size $m \times m$ for the alignment purposes. SH algorithm needs extra space only for the one data hash at a time and hashed all parts of the pattern so the increase is negligible. On the other hand, both NSH and LSB need to remember all hashes for the currently used data chunk, but because it is stored in the same structure as the data the complexity is increased to a theoretical maximum of $2 \times C \times N$ where N is the maximum number of cells stored in one chunk file. However the number of hashes is lower than the number of cells, because in this stage, as was explained in the implementation, only a part of the data are actually compared with the pattern.

5.3.1 Binary chunking

Performed data chunking creates one meta file, which contains the header presented in the implementation and an information needed to recognize the bin files containing the actual data, which are also created by chunking. Last, and usually the largest, file is a mask file. The size of this file is the main reason for low density data files to be less effectively chunked. As the mask always contains a bit for each cell, its size for the file with 2 % density is more than 70 % of the total size of chunked data. On the other hand, in the dense file the mask only takes up about 10 % of the total size.

Memory usage of algorithms using chunked data is $\mathcal{O}(N * C)$.

Conclusion

The goal of this work was to get to know various methods of solving pattern matching problem, because of its diverse usability in the modern databases. Common solutions of this problem can be split into two categories based on the allowance of mistakes in the search. According to this criterion it can be called exact pattern matching or approximate pattern matching, where this work was more focused on solving of the approximate version as it can have numerous advantages.

As was presented in the analysis part of the work majority of the research is focusing primarily on one or two dimensional data and algorithms processing higher dimensional spaces are usually low dimensional algorithms generalized into high dimensional spaces or methods reducing the dimensionality of both data and pattern.

Outcome of this work is an algorithm allowing of approximate pattern matching in sparse high dimensional data using the SciDB binary format and chunking. Presented method solving this problem is based on the algorithm named Stricter Filter by Navarro and Baeza-Yates but with the usage of Locality Sensitive Hashing approach.

The algorithms were measured using pseudo randomly generated data, but also using some real world datasets. Finally eight algorithms were implemented, two for solution of exact pattern matching (brute force and Exact Multidimensional Pattern Matching algorithm) and six for approximate pattern matching, where two of the methods by Navarro–Baeza-Yates were only theoretically designed (FFS and SF algorithms) but not implemented until now. Then there is one brute solution and three methods using hashes.

Due to the properties of exact pattern matching its solutions achieved lower times than approximate approaches, however they were not the main concern of this work so they were not researched further. FFS and SF algorithms proved to be about twice as fast as approximate brute solution in average case. NSH algorithm proved to be worse than other non-brute approximate solutions, because it hashes data every time the hash is needed, while the

SH and LSB algorithms hash the data only once which can improve the time needed for finding of the possible solutions to preverification and dynamic check, because of the possible reusability of the hashed value. This approach is highly dependent on the quality of the used hashing function as the collision rate hinges on it.

A weak point of the implementation is a work with the sparse data, although it is usable even on the data with arbitrary sparsity. However it is not well optimized for the data with very low density so it could be a goal of the future work.

Bibliography

[1] Baeza-Yates, R.; Navarro, G. New models and algorithms for multidimensional approximate pattern matching. *J. Discret. Algorithms*, volume 1, no. 1, 2000: pp. 21–49.

[2] Jiang, L.; Kawashima, H.; et al. Efficient Window Aggregate Method on Array Database System. *Journal of Information Processing*, volume 24, no. 6, 2016: pp. 867–877.

[3] Saad, Y. SPARSKIT: A basic tool kit for sparse matrix computations. 1990, [Online; accessed 29-04-17]. Available from: `https://ntrs.nasa.gov/search.jsp?R=19910023551`

[4] Samet, H. *Foundations of multidimensional and metric data structures.* Morgan Kaufmann, 2006.

[5] Peleg, S.; Rosenfeld, A. A Min-Max Medial Axis Transformation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, volume PAMI-3, no. 2, March 1981: pp. 208–210, ISSN 0162-8828, doi:10.1109/TPAMI.1981.4767082.

[6] Nagarkar, P.; Candan, K. S.; et al. Compressed spatial hierarchical bitmap (cSHB) indexes for efficiently processing spatial range query workloads. *Proceedings of the VLDB Endowment*, volume 8, no. 12, 2015: pp. 1382–1393.

[7] Bayer, R. *The universal B-Tree for multidimensional Indexing.* Mathematisches Institut und Institut für Informatik der Technischen Universität München, 1996.

[8] Belussi, A.; Faloutsos, C. Self-spacial join selectivity estimation using fractal concepts. *ACM Transactions on Information Systems (TOIS)*, volume 16, no. 2, 1998: pp. 161–201.

[9] Papadopoulos, A. N.; Manolopoulos, Y. Multiple range query optimization in spatial databases. In *East European Symposium on Advances in Databases and Information Systems*, Springer, 1998, pp. 71–82.

[10] Evgeniou, T.; Micchelli, C. A.; et al. Learning multiple tasks with kernel methods. *Journal of Machine Learning Research*, volume 6, no. Apr, 2005: pp. 615–637.

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

[12] Cha, S.-H. Comprehensive survey on distance/similarity measures between probability density functions. *City*, volume 1, no. 2, 2007: p. 1.

[13] Norouzi, M.; Fleet, D. J.; et al. Hamming distance metric learning. In *Advances in neural information processing systems*, 2012, pp. 1061–1069.

[14] Skopal, T.; Krátkỳ, M.; et al. A new range query algorithm for Universal B-trees. *Information Systems*, volume 31, no. 6, 2006: pp. 489–511.

[15] Ukkonen, E. On approximate string matching. In *Foundations of Computation Theory*, Springer, 1983, pp. 487–495.

[16] Wang, J.; Li, G.; et al. Fast-join: An efficient method for fuzzy token matching based string similarity join. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, IEEE, 2011, pp. 458–469.

[17] Cohen, W.; Ravikumar, P.; et al. A comparison of string metrics for matching names and records. In *Kdd workshop on data cleaning and object consolidation*, volume 3, 2003, pp. 73–78.

[18] Hofmann, T.; Schölkopf, B.; et al. Kernel methods in machine learning. *The annals of statistics*, 2008: pp. 1171–1220.

[19] Baeza-Yates, R. A. Similarity in two-dimensional strings. In *International Computing and Combinatorics Conference*, Springer, 1998, pp. 319–328.

[20] Baeza-Yates, R.; Navarro, G. Fast two-dimensional approximate pattern matching. In *Latin American Symposium on Theoretical Informatics*, Springer, 1998, pp. 341–351.

[21] Baeza-Yates, R. A.; Perleberg, C. H. Fast and practical approximate string matching. In *Annual Symposium on Combinatorial Pattern Matching*, Springer, 1992, pp. 185–192.

[22] Navarro, G.; Baeza-Yates, R. Fast multi-dimensional approximate pattern matching. In *Annual Symposium on Combinatorial Pattern Matching*, Springer, 1999, pp. 243–257.

[23] Crochemore, M.; Rytter, W. *Jewels of stringology: text algorithms.* World Scientific, 2003.

[24] Kärkkäinen, J.; Ukkonen, E. Two- and higher-dimensional pattern matching in optimal expected time. *SIAM Journal on Computing*, volume 29, no. 2, 1999: pp. 571–589.

[25] Krithivasan, K.; Sitalakshmi, R. Efficient two-dimensional pattern matching in the presence of errors. *Information Sciences*, volume 43, no. 3, 1987: pp. 169–184.

[26] Sellers, P. H. The theory and computation of evolutionary distances: pattern recognition. *Journal of algorithms*, volume 1, no. 4, 1980: pp. 359–373.

[27] Zou, X.; Boyuka, D. A., II; et al. AMR-aware in Situ Indexing and Scalable Querying. In *Proceedings of the 24th High Performance Computing Symposium*, HPC '16, San Diego, CA, USA: Society for Computer Simulation International, 2016, ISBN 978-1-5108-2318-1, pp. 26:1–26:9, doi:10.22360/SpringSim.2016.HPC.012.

[28] Byna, S.; Wehner, M. F.; et al. Detecting atmospheric rivers in large climate datasets. In *Proceedings of the 2nd international workshop on Petascal data analytics: challenges and opportunities*, ACM, 2011, pp. 7–14.

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

[30] Zhao, W.; Rusu, F.; et al. Similarity Join over Array Data. In *Proceedings of the 2016 International Conference on Management of Data*, ACM, 2016, pp. 2007–2022.

[31] Chávez, E.; Marroquín, J. L.; et al. Spaghettis: An array based algorithm for similarity queries in metric spaces. In *String Processing and Information Retrieval Symposium, 1999 and International Workshop on Groupware*, IEEE, 1999, pp. 38–46.

[32] Baumann, P.; Merticariu, V. On the efficient evaluation of array joins. In *Big Data (Big Data), 2015 IEEE International Conference on*, IEEE, 2015, pp. 2046–2055.

[33] Böhm, C.; Braunmüller, B.; et al. Epsilon grid order: an algorithm for the similarity join on massive high-dimensional data. In *ACM SIGMOD Record*, volume 30, ACM, 2001, pp. 379–388.

[34] Dittrich, J.-P.; Seeger, B. GESS: a scalable similarity-join algorithm for mining large data sets in high dimensional spaces. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, ACM, 2001, pp. 47–56.

[35] Jacox, E. H.; Samet, H. Metric space similarity joins. *ACM Transactions on Database Systems (TODS)*, volume 33, no. 2, 2008: pp. 1–38.

[36] Koudas, N.; Sevcik, K. C. High dimensional similarity joins: Algorithms and performance evaluation. *IEEE Transactions on Knowledge and Data Engineering*, volume 12, no. 1, 2000: pp. 3–18.

[37] Jarvis, R. A.; Patrick, E. A. Clustering using a similarity measure based on shared near neighbors. *IEEE Transactions on Computers*, volume 100, no. 11, 1973: pp. 1025–1034.

[38] Goshtasby, A. A. *Similarity and Dissimilarity Measures*. London: Springer London, 2012, ISBN 978-1-4471-2458-0, pp. 7–66, doi:10.1007/978-1-4471-2458-0_2. Available from: `http://dx.doi.org/10.1007/978-1-4471-2458-0_2`

[39] Naumann, F. Similarity Measures. 2013, universität Potsdam Lecture. Available from: `https://hpi.de/fileadmin/user_upload/fachgebiete/naumann/folien/SS13/DPDC/DPDC_12_Similarity.pdf`

[40] Dekhtyar, A. Distance/Similarity Measures. 2009, course CSC 466, California Polytechnic State University Lecture. Available from: `http://users.csc.calpoly.edu/~dekhtyar/466-Spring2012/lectures/lec09.466.pdf`

[41] Aho, A. V.; Corasick, M. J. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, volume 18, no. 6, 1975: pp. 333–340.

[42] Baeza-Yates, R.; Régnier, M. Fast two-dimensional pattern matching. *Information Processing Letters*, volume 45, no. 1, 1993: pp. 51–57.

[43] Karp, R. M.; Rabin, M. O. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, volume 31, no. 2, 1987: pp. 249–260.

[44] Zhu, R. F.; Takaoka, T. A technique for two-dimensional pattern matching. *Communications of the ACM*, volume 32, no. 9, 1989: pp. 1110–1120.

[45] Stonebraker, M.; Brown, P.; et al. SciDB: A Database Management System for Applications with Complex Analytics. *Computing in Science Engineering*, volume 15, no. 3, May 2013: pp. 54–62, ISSN 1521-9615, doi:10.1109/MCSE.2013.19.

[46] Stonebraker, M.; Brown, P.; et al. The architecture of SciDB. In *International Conference on Scientific and Statistical Database Management*, Springer, 2011, pp. 1–16.

[47] Amir, A.; Butman, A.; et al. Real scaled matching. *Information Processing Letters*, volume 70, no. 4, 1999: pp. 185–190.

[48] Amir, A.; Calinescu, G. Alphabet independent and dictionary scaled matching. In *Annual Symposium on Combinatorial Pattern Matching*, Springer, 1996, pp. 320–334.

[49] Fredriksson, K.; Ukkonen, E. A rotation invariant filter for two-dimensional string matching. In *Annual Symposium on Combinatorial Pattern Matching*, Springer, 1998, pp. 118–125.

[50] Fredriksson, K.; Navarro, G.; et al. An index for two dimensional string matching allowing rotations. In *IFIP International Conference on Theoretical Computer Science*, Springer, 2000, pp. 59–75.

[51] Fredriksson, K.; Navarro, G.; et al. Fast filters for two dimensional string matching allowing rotations. 2000, [Online; accessed 29-04-17]. Available from: `http://swp.dcc.uchile.cl/TR/1999/TR_DCC-1999-009.pdf`

[52] Kärkkäinen, J.; Ukkonen, E. Two and Higher Dimensional Pattern Matching in Optimal Expected Time. In *SODA*, volume 94, 1994, pp. 715–723.

[53] Giancarlo, R. A Generalization of the Suffix Tree to Square Matrices, with Applications. *SIAM Journal on Computing*, volume 24, no. 3, 1995: pp. 520–562.

[54] Muth, R.; Manber, U. Approximate multiple string search. In *Annual Symposium on Combinatorial Pattern Matching*, Springer, 1996, pp. 75–86.

[55] Paradigm4, Inc. SciDB Reference Manual. 2017, [Online; accessed 29-04-17]. Available from: `http://www.paradigm4.com/HTMLmanual/15.7/scidb_ug/index.html`

[56] Wang, J.; Shen, H. T.; et al. Hashing for Similarity Search: A Survey. *CoRR*, volume abs/1408.2927, 2014. Available from: `http://arxiv.org/abs/1408.2927`

[57] Xia, Y.; He, K.; et al. Joint inverted indexing. In *Proceedings of the IEEE International Conference on Computer Vision*, 2013, pp. 3416–3423.

[58] Hofmann, T.; Schölkopf, B.; et al. Kernel methods in machine learning. *The annals of statistics*, 2008: pp. 1171–1220.

[59] Wang, X.; Pedrycz, W.; et al. *Machine Learning and Cybernetics: 13th International Conference, Lanzhou, China, July 13–16, 2014. Proceedings*. Communications in Computer and Information Science, Springer Berlin Heidelberg, 2014, ISBN 9783662456521. Available from: `https://books.google.cz/books?id=FHq1BQAAQBAJ`

[60] Brisaboa, N. R.; Cerdeira-Pena, A.; et al. Efficient Representation of Multidimensional Data over Hierarchical Domains. In *International Symposium on String Processing and Information Retrieval*, Springer, 2016, pp. 191–203.

[61] Afrati, F. N.; Sharma, S.; et al. Computing marginals using MapReduce. *Journal of Computer and System Sciences*, 2017.

[62] Soroush, E.; Balazinska, M.; et al. Efficient Iterative Processing in the SciDB Parallel Array Engine. In *Proceedings of the 27th International Conference on Scientific and Statistical Database Management*, SSDBM '15, New York, NY, USA: ACM, 2015, ISBN 978-1-4503-3709-0, pp. 39:1–39:6, doi:10.1145/2791347.2791362. Available from: `http://doi.acm.org/10.1145/2791347.2791362`

[63] Knuth, D. E.; Morris, J. H., Jr; et al. Fast pattern matching in strings. *SIAM journal on computing*, volume 6, no. 2, 1977: pp. 323–350.

[64] Wang, J.; Liu, W.; et al. Learning to Hash for Indexing Big Data — A Survey. *Proceedings of the IEEE*, volume 104, no. 1, Jan 2016: pp. 34–57, ISSN 0018-9219, doi:10.1109/JPROC.2015.2487976.

[65] Kulis, B.; Grauman, K. Kernelized locality-sensitive hashing for scalable image search. In *Computer Vision, 2009 IEEE 12th International Conference on*, IEEE, 2009, pp. 2130–2137.

[66] Shakhnarovich, G. *Learning task-specific similarity*. Dissertation thesis, Massachusetts Institute of Technology, 2005.

[67] Mu, Y.; Yan, S. Non-Metric Locality-Sensitive Hashing. In *AAAI*, 2010, pp. 539–544.

[68] Baumann, P.; Dehmel, A.; et al. The multidimensional database system RasDaMan. In *Acm Sigmod Record*, volume 27, ACM, 1998, pp. 575–577.

[69] Kulis, B.; Jain, P.; et al. Fast similarity search for learned metrics. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, volume 31, no. 12, 2009: pp. 2143–2157.

[70] Liu, W.; Wang, J.; et al. Supervised hashing with kernels. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, IEEE, 2012, pp. 2074–2081.

[71] Kotsiantis, S. B. Supervised machine learning: A review of classification techniques. *Informatica (Ljubljana)*, volume 31, no. 3, 2007: pp. 249–268.

[72] Python Software Foundations. Python Laguage Reference, version 2.7. 2010–. Available from: `https://www.python.org/`

[73] Victor, S. Generate pseudo-random numbers. 1990–, [Online; accessed 29-04-17]. Available from: `https://docs.python.org/2/library/random.html`

[74] Oliphant, T.; et al. NumPy. 2006–, [Online; accessed 29-04-17]. Available from: `http://www.numpy.org/`

[75] Jones, E.; Oliphant, T.; et al. SciPy: Open source scientific tools for Python. 2001–, [Online; accessed 29-04-17]. Available from: `http://www.scipy.org/`

[76] Hunter, J. D. Matplotlib: A 2D graphics environment. *Computing In Science & Engineering*, volume 9, no. 3, 2007: pp. 90–95, doi:10.1109/MCSE.2007.55.

[77] C++ Standards Committee; et al. iso/iec 14882: 2011, standard for programming language c++. Technical report, ISO/IEC, 2011. Available from: `http://www.open-std.org/jtc1/sc22/wg21`

Appendix **A**

Acronyms

AFL Array Functional Language

AMR Adaptive Mesh Refinement

ANN Approximate Nearest Neighbour

AQL Array Query Language

BA Brute Force solution of approximate pattern matching

BE Brute Force solution of exact pattern matching

CSV Comma Separated Values

DNA Deoxyribonucleic Acid

EEG Electroencephalogram

FFS Fast Filter Searching for approximate pattern matching

KMP Knuth–Morris–Pratt

KNN K Nearest Neighbours

KS Krithivasan and Sitalakshmi

LSB Least Significant Bit

LSH Locality Sensitive Hashing

MAT Medial Axis Transformation

NSH Naive SimHashed Stricter Filter

RasDaMan Raster Data Manager

RC Row, Column

RNN R-Near Neighbour

SBST Self Balancing Binary Search Tree

SE Baeza-Yates and Navarro solution of exact pattern matching

SF Stricter Filter for approximate pattern matching

SH Improved SimHash

SQL Structured Query Language

Appendix B

Contents of enclosed USB

```
README.txt ...................... the file with USB contents description
├── scripts ......................... the directory with Python scripts
├── include ........................ the directory with header C++ files
├── src ................................... the directory of source codes
└── text ...................................... the thesis text directory
    ├── DP_Kucerova_Anna_2017.pdf ........... the thesis text in PDF format
    ├── DP_Kucerova_Anna_2017.tex ........... the thesis text in TEX format
    └── img ................................ folder with images used in text
```

Appendix **C**

Binary SciDB file format

According to SciDB manual web page [55] each binary file represents 1D array, where there are no empty cells in the array (it is dense), but there can be null values for attributes that are nullable. When creating binary files the rules for SciDB attributes are:

- Attributes in a binary file appear in the same left-to-right order as the attributes of the corresponding array's schema.
- A fixed-size attribute of length n is represented by n bytes (in little-endian order).
- A variable-size attribute of length n is represented by a four-byte unsigned integer of value n, followed by the n data bytes of the attribute. String attributes include the terminating NUL character (ASCII 0x00) in both the length count and the data bytes.
- Whether fixed or variable length, a nullable attribute is preceded by a single byte. If a particular attribute value is null, the prefix byte contains the "missing reason code", a value between 0 and 127 inclusive. If not null, the prefix byte must contain 0xFF (-1).
- Even if a nullable attribute value is in fact null, the prefix byte is still followed by a representation of the attribute: n zero bytes for fixed-length attributes of size n, or four zero bytes for variable-length attributes (representing a length count of zero).
- Binary data does not contain attribute separators or cell separators.

Among the other characteristics of this format belong:

- Each cell of the array is represented in contiguous bytes.
- There are no end-of-cell delimiters.

- A fixed-length data type that allows null values will always consume one more byte than the data type requires, regardless of whether the value is null or non-null.
- A fixed-length data type that disallows null values will always consume exactly as many bytes as that data type requires.
- A string data type that disallows nulls is always preceded by four bytes indicating the string length.
- The length of a null string is recorded as zero.
- Every non-null string value is terminated by the NUL character. Even a zero-length string will include this character.
- The length of a non-null string value includes the terminating NUL character.
- If a nullable attribute contains a non-null value, the preceding null byte is -1.
- If a nullable attribute contains a null value, the preceding null byte will contain the missing reason code, which must be between 0 and 127.
- The file does not contain index values for the dimension of the array to be populated by the LOAD command. The command reads the file sequentially and creates the cells of the array accordingly.

Each cell must contain a data type recognizable by SciDB (e.g. native types, user-defined types). Storage format is assumed to be the x86_64 little-endian.

Appendix D

Usage and Types of MAT

Typical usage for this algorithm is at nearest neighbour finding query. In this task the goal is to locate neighbour of the element, which can be done by following ancestor links until finding nearest common ancestor and then descending to the leaf. For this task there can be used the Spaghettis algorithm [31] which has only one parameter and it is the number of pivots used. As because the parameter it is clear that the algorithm is pivot based, mapping metric space to k-dimensional vector space. Its competitive algorithm is AESA (Approximating Eliminating Search Algorithm). The distance function of this algorithm has metric properties. Similarity queries solved can be divided into two categories and these are range queries (retrieve all elements which are within distance r to q) and nearest neighbour queries (retrieve the closest elements to q in U). The algorithm steps are: For each pivot calculate and save the distances to each database element. Sort each array saving the permutation with respect to the preceding array (from array to array only the successful traversals are followed). Given k intervals, define k sets (k is number of pivots). Obtain the index intervals corresponding to each set and follow each point through the pointers to find out if it falls inside all the index intervals. If a point falls inside all the index intervals it is in the intersection.

There is also a modification for nearest neighbour queries by selecting unchecked point do range query and if it returns more than one element repeat with lower range else stop. In this case random selection is not totally ineffective.

There is range of types of MAT algorithm, varying from different measures to data organization.

D.0.0.0.1 SPAN The Spacial Piecewise Approximation by Neighbourhoods is the generalized version of MAT algorithm trying to compute maximal homogeneous disks. It is the same for pictures with two values (white/black) as there it is looking for constant values. But it is expensive to compute.

D.0.0.0.2 GrayMat This algorithm uses Gray-weighted distance. To use it effectively there is a need to segment the picture into two classes. First class is marked as 0 (e.g. background of the picture without objects) and second class is labelled as non 0 (e.g. objects in the picture, not the background).

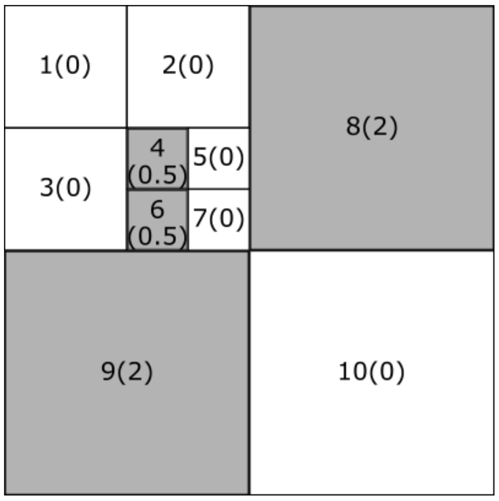
D.0.0.0.3 GradMat In this approach there can be computed a specific score for each point P of a picture based on the gradient magnitudes at all pairs of points that have P as their midpoint. These scores are high for points that lie midway between pairs of anti-parallel edges. They define a weighted ”medial axis”. Advantage of this algorithm lies in its sensitivity to noise and irregularities in region edges.

D.0.0.0.4 MMMAT This variation of the MAT uses Min Max algorithm and is insensitive to noise. It is based on the fact that the MAT of a set S can be constructed by a process of iterative shrinking and re-expanding which corresponds with local MIN and MAX operations. When this construction is used it can be applied to gray-scale pictures. [5]

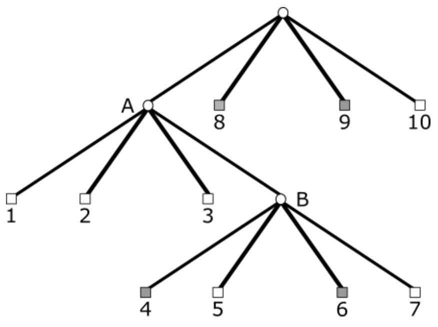
D.0.0.0.5 QMAT Mentioned method is appropriate for processing pictures containing only binary values. It can be imagined that this picture is like an array of $2^n x 2^n$ pixels, which is going to be repeatedly subdivided into quadrants until blocks which consist of a single value are obtained. Quadtree skeleton then consists of the set of black blocks in the image satisfying three conditions and its radius. The conditions are:

- Whole image is spanned by the skeleton
- Elements of skeleton are the blocks with the largest distance transform values
- No block in the set of blocks but not in the tree requires more than one element of the tree for its subsumation.

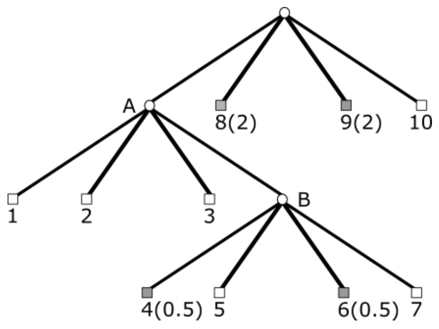
Each created tree is unique and also has distance value stored in each black node (in white nodes there is no need to store it as there would be 0 value). Example of this method is in this figure D.1.



(a) Sample Image for QMAT processing.



(b) Quadtree of the Sample Image.



(c) QMAT tree of the Sample Image.

Figure D.1: Example of QMAT processing.

Appendix E

ArrayLoop System

ArrayLoop is a middle-ware system that translates queries expressed in the model into queries executable by the SciDB. It uses iterative array model, which means there is an array being modified during iteration. Termination condition of this function is represented as an AQL function. This model is interested in major steps that can be decomposed into a minor ones which gives us possibility of parallel evaluation. By major steps its meant that function operates on all cells in array and minor step is otherwise. First of all there is need to assign what to do which is done by *Pi* function. Then it is needed to compute aggregate function separately for each group. This computation can be done with the help of an algorithm presented in [61] and described in the section below E.1. After that the records are updated with the new values and this generates new subset containing the results. For best usage of this algorithm was developed FixPoint function where user specifies the logic of iterative algorithm and it will translate it into ArrayLoop which prepares it into AQL and then SciDB can finally process it.

To transform the function into the AQL format, ArrayLoop first automatically expands and rewrites the operation into an incremental implementation, if it is possible, and efficiently computes the last state of iterative array with the use of the updated cell in each iteration. Then it extracts (for each array chunk) the overlap area of neighbouring chunks and stores this overlaps together with the original chunk which provides both the core and overlap data to the operator during processing. In the final step it identifies outlines of the structures on a low resolution array and then refines details onto high resolution arrays. [62]

E.1 Computing marginals

As a marginal of a data cube it can be understood the aggregation of the data in all those tuples that have fixed values in a subset of the dimensions of the cube. Order of the marginal is the number of dimensions over which

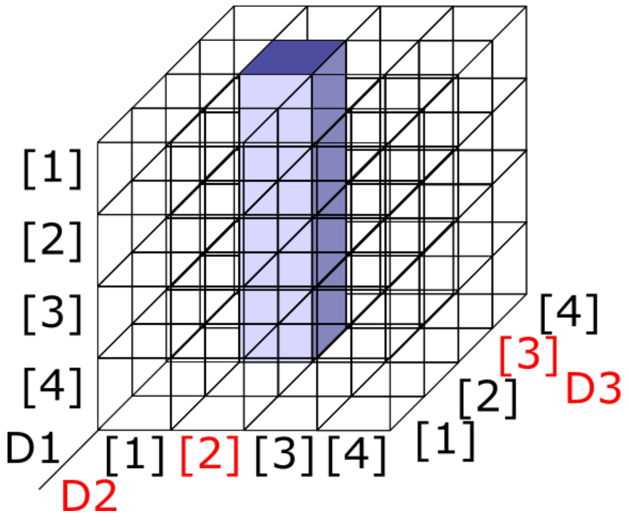


Figure E.1: Example of DataCube and selecting dimensions where $y = 2$ and $z = 3$.

aggregation is happening. Mapping of reducers q and replication rate r is that no reducer associated with more than q inputs and for every output there is some reducer that is associated with all the inputs that output needs for its computation. Dimensions in this case have same number of different values. Function $C(n, m, k)$ defines minimum number of sets of size m out of n elements that every set of k out of the same n elements is contained in one of the sets of size m, and it is called the covering number. Different sets of n, m, k are handled differently but when approached with unknown values it can be solved as: one group of handles contains dimensions to $m - k + 1$ plus any $k - 1$ dimensions from group 2, and others are formed recursively to cover the dimensions of group 2 and have none of the members of group 1. This problem can be generalized as a problem with weights of dimensions and dividing dimensions into several groups. [61]

Example of computing marginal in three dimensional space (as seen in the pictures E.1 and E.2):
SELECT SUM(V)
FROM DataCube
WHERE D1 = x, D2 = y, D3 = z;

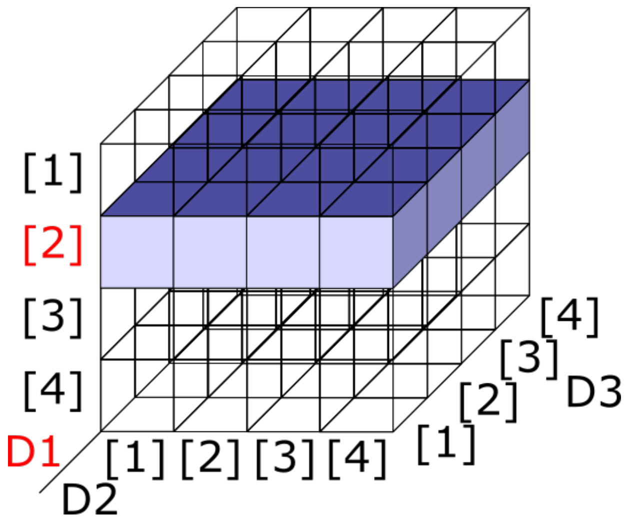


Figure E.2: Example of DataCube and selecting dimensions where $x = 2$.