Palestine Polytechnic University
Deanship of Graduate Studies and Scientific Research
Master of Informatics

Genome Database Indexing using a Modified Wavelet Transformation and BTree

Submitted By
Samer Mahmoud Wohoush

In Partial Fulfillment of the Requirements for the Degree
Master of Informatics

June, 2011

The undersigned hereby certify that they have read, examined and recommended to the Deanship of Graduate Studies and Scientific Research at Palestine Polytechnic University the approval of a thesis entitled:

Genome Database Indexing using a Modified Wavelet Transformation and BTree

Submitted by **Samer M. Wohoush**

in partial fulfillment of the requirements for the degree of Master in Informatics

Graduate Advisory Committee:

Committee Chair Name, University:

Dr. Radwan Tahboub, Palestine Polytechnic University Date:_____

Signature:_____

Committee Member Name (Supervisor), University:

Dr. Mahmoud Hasan Saheb, Palestine Polytechnic University

Signature:_____ Date: 30/11/2011

Committee Member Name, University:

Dr. Hashem Tamimi, Palestine Polytechnic University

Signature:_____ Date: 30/11/2011

External Committee Member Name, University:

Dr. Rashid Jayousi, AL-Quds University

Signature:_____ Date: 30/11/2011

Thesis Approved

Prof. Dr. Karim Tahboub
Dean of Graduate Studies and Scientific Research
Palestine Polytechnic University

Date:_____

Signature:_____

Abstract

The main problem of searching the Genome DNA sequences is the large size of sequences and the very high and variant sequences lengths. There are different methods used to enhance sequence searching like using database indexing methods instead of direct access to sequence files.

Our main idea is to provide a suitable access methodology, in time and space, to Genome DNA sequences for searching and comparing while considering the size of the data and the index. The Genome database searching system is needed to give facilities, compact data representation and compression, accurate output, practical to use, and to minimize the number of I/O operations. I/O operations mainly needed at last step to avoid false positives (the sequences that appear to be related but are not related to the searched query). The number of candidate sequences, that need to be checked by database I/O referencing, will be reduced by pruning so no need to search the whole database.

In this thesis, we propose an approach to build a complete index structure that is suitable for large database to do searching with efficient storage space and search time. We use a suitable representation of Genome DNA sequences using n-gram Haar wavelet transformation, and integer conversion for coefficients. A suitable index structure, which is build upon a modified BTree index, is used to hold the integer representation after transformation. We also introduce enhancements that can be followed to increase system efficiency by decreasing index storage size. Our structure is called the Modified Wavelet Transformation and BTree (M-WTBT). The M-WTBT structure allows tuning for a set of parameters so that the index structure is suitable to the available resources. An implementation is done, using a dataset used previously by a set of researches, to approve features and to show the advantages of the M-WTBT structure. Also, the M-WTBT shown to be effective when compare with a set of previous researches.

Keyword: Sequence transformation, sequence compression, large database indexing, Haar Wavelet Transformation, Genome DNA Sequence searching and indexing.

ملخص

المشكلة الرئيسية للبحث في تسلسل الحمض النووي هو الحجم الكبير للسلسلة واختلاف أطوال المتسلسلات، هناك طرق مختلفة لتعزيز استخدام البحث في المتواليات مثل الاستخدام المباشر لملفات بيانات السلسلة الفعلية أو باستخدام أساليب فهرسة قاعدة البيانات التي توفر مجموعة من الميزات و التحسينات وسهولة في الاستخدام.

ان الهدف الرئيسي لهذه الدراسة هو توفير طريقة و الية وصول مناسبة، من حيث سرعة الوصول و حجم الفهرسة، لتسلسل الحمض النووي للقيام بعملية البحث والمقارنة مع الاخذ بعين الاعتبار حجم قاعدة البيانات. يجب ان يتوفر لدى عملية البحث في تسلسل البيانات مجموعة من الميزات الرئيسية مثل امكانية ضغط البيانات، دقة النتائج، و تقليل عدد عمليات الادخال و الاخراج (I/O). ان عمليات الادخال و الاخراج تتم خلال الخطوة الاخيرة للبحث لحل مشكلة الايجابية الخادعة (و هي البيانات التى توحي ان لها علاقة بالمطلوب و هي ليست كذلك) للوصول الي البيانات الفعلية التي تم تخزينها في قاعدة البيانات و التي يجب ان نعمل على تقليلها لاصغر عدد ممكن.

لقد قمنا باقترح نظام فهرسة ذا كفاءة عالية من حيث الحجم و سرعة البحث، مناسب لقاعدة البيانات ذات الحجم الهائل للقيام بعملية البحث. لقد قمنا باستخدام تمثيل مناسب من تسلسل الحمض النووي باستخدام " n-gram Wavelet Transformation المعدل" لتحويل شكل البيانات الي ارقام عوضا عن الاحرف، و تم بناء هيكل مناسب بالاعتماد على BTree مع اضافة بعض التعديلات. تم اضافة بعض التحسينات التي يمكن اتباعها لزيادة كفاءة النظام كما تم اضافة امكانية تعديل مجموعة من المتغيرات ليتلاءم نظام الفهرسة مع المصادر المتاحة للاستخدام. تم تسمية نظام الفهرسة الجديد ب Modified Wavelet Transformation and BTree (M-WTBT). كما تم عمل تصميم لنظام الفهرسة لاثبات فاعلية التحسينات و كذلك لمقارنة ميزات النظام مع مجموعة من الابحاث السابقة.

DECLARATION

I declare that the Master Thesis entitled "*Genome Database Indexing using a Modified Wavelet Transformation and BTree* " is my own original work, and hereby certify that unless stated, all work contained within this thesis is my own independent research and has not been submitted for the award of any other degree at any institution, except where due acknowledgment is made in the text.

Samer Mahmoud Wohoush

Signature:_____ Date:_____

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for the master degree in Informatics at Palestine Polytechnic University, I agree that the library shall make it available to borrowers under rules of the library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgement of the source is made.

Permission for extensive quotation from, reproduction, or publication of this thesis may be granted by my main supervisor, or in his absence, by the Dean of Graduate Studies and Scientific Research when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

Samer Mahmoud Wohoush

Signature:_____ Date:_____

DEDICATION

I dedicate this work to my parents and my wife Bayan. Without their patience, understanding, there continuous support, and most of all love, the completion of this work would not have been possible during my vital educational years. Also their endless patience and encouragement when it was most required.

ACKNOWLEDGMENT

I wish to thank the members of my committee for their support, patience, and good humor. Their gentle but firm direction has been most appreciated. Dr. Mahmoud Saheb was particularly helpful in guiding me toward a qualitative methodology, whose encouragement, guidance and support from the initial to the final level enabled me to develop an understanding of the subject.

I would like to acknowledge the help of Dr. Hashem Tamimi, Dr. Rashid Jayousi and Dr. Radwan Tahboub for their support and assistance especially during the evaluation of my thesis.

This dissertation would not have been possible without the guidance and the help of several individuals who in one way or another contributed and extended their valuable assistance in the preparation and completion of this study.

Abbreviations:

	Abbreviation	Full term
1.	**AB**	Approximate Bitmap
2.	**BBC**	Byte-aligned Bitmap Compression
3.	**BLAST**	Basic Local Alignment Search Tool
4.	**BIC**	Binary Interpolative Coding
5.	**BP**	Base Pairs
6.	**CSV**	Comma Separated Value
7.	**DAWG**	Directed Acyclic Word Graph
8.	**DNA**	Deoxyribonucleic Acid
9.	**DML**	Data manipulation language
10.	**ED**	Edit Distance
11.	**FV**	Frequency Vector
12.	**MRS**	Multi-Resolution index Structure
13.	**M-WTBT**	Modified Wavelet Transformation and BTree
14.	**NCBI**	National Center for Biotechnology Information
15.	**NR**	Numerical Representation
16.	**NP**	Nondeterministic Polynomial
17.	**RABI**	Random Access Blocked Inverted index
18.	**RR**	Relation Row
19.	**RDMS**	Relational Database Management Systems
20.	**SST**	Sequence Search Tree
21.	**WT**	Wavelet Transformation
22.	**WAH**	Word Aligned Hybrid

Table of Contents

List of Figures

Figure Page

List of Tables

Table Page

Chapter 1

Introduction

Large size index structure stores parts of index data at the computer main memory and the rest is stored at lower level storage devices like hard drive. With the high increase of RAM size at the last years, large amount of information can be held at main memory than before.

Nowadays memory size, memory architectures and the distribution of memory over processor cores [3] make a difference to index structures. The result of the new changes of memory structures expected to hold larger index size and faster query answering than before.

Accessing information through index entries is managed by memory paging operation, all index entries divided into an equal pages size and consecutive locations at main memory. Any I/O operation will require loading the page in memory.

1.1 Genome DNA indexing

Bioinformatics data consists of a huge amount of information due to the large number of sequences and the growing amount of Genome day by day, and this data need to be accessed efficiently for many needs. Genome DNA sequence consists of a repeating combination of mainly four characters (A, C, G, T) which are bases of DNA with different lengths. What makes one DNA data item distinct from another is its DNA sequence.

Using a suitable representation of Genome DNA sequences and a suitable index structure, to hold this representation at main memory for efficient processing, will reduce number of disk I/O accesses. I/O operations needed at the end, to avoid false hits. We will reduce the number of candidate DNA sequences that need to be checked by pruning, so no need to search the whole database like most sequential scan methods such as BLAST [31].

Dealing with string of characters for large database is not easy in term of space and access time. Genome databases, like NCBI, have a huge size because of the daily addition of new data. Other large databases that include text and sequences are electronic books, search engines, and biological data.

Most of the data processing on Genome database tries to find small size, efficient digit value that can represents DNA sequences. The large number of I/O operation, when accessing large size database, is very costly in term of space and performance. Accessing the database need to be in minimum amount and at last stage after filtrations to reduce the number of records to be accessed. The Genome database searching methods are either sequence alignment or sequence indexing for exact and not exact matching. Sequence alignment used to compare sequences to find the best alignment, like FASTA*, and match identical subsequences as far as possible. Sequence alignment is a trade-off between search accuracy and execution time and requires high complexity. Different method of alignment uses different methodology, some are using scoring schema for alignment.

A general method used for reducing search cost is using index system to enhance query searching and better retrieve of information. Indexing system aims to fetch a subset of sequences from the database to reduce query computation cost. Genome DNA sequence index methods are based on two bases, either character based or word based indexing. Examples of Genome indexing are hashing, suffix tree [37] which is based on tree structure.

Some indexing methods based on not using the whole data, this means lose of information needed for search process like suffix tree [37]. The main factors that affect indexing are: sequence length, sequence domain, number of sequences, exact or similarity search is needed.

1.2 Proposed indexing structure overview

We aim, through this thesis, to have an efficient index structure for searching the Genome data with a set of properties like: a compact data presentation, practical to use, and less I/O operations to reference original data. Our work has many application domains, like sequence/pattern matching, RDMS indexing, and large scale DB clustering.

We consider the Genome DNA sequence as the key value for genome database. Transforming this key to a digit is required, to increase efficiency by decreasing storage size and being suitable for index structure. Wavelet Transformation technique [50] for Genome DNA sequences is a suitable choice for our needs to perform transformation as it gives us the following advantages:

* BLAST home page, http://blast.ncbi.nlm.nih.gov/Blast.cgi.

Firstly, it saves sequence order while considering amount of overlapping carefully. Secondly, allows transforming characters to digits depending on frequencies of characters which is a key property of Genome DNA sequence. Thirdly, wavelet transformation referring the database one time only even if we used different sliding window sizes.

Little amount of storage is needed after finding the first level wavelet coefficients [50]. The second level, corresponding to second sliding window size, can be calculated depending on the first level instead of referring the original sequence again.

Our approach uses substring searching for matching identical pattern using a sliding window. We reduce candidate list of sequences needed to be checked after pruning. In other words, optimizing the number of I/O operations.

Relational database systems provide different type of indexing like BTree, RTree [24], and hashing. We have started our evaluations, for storage space and search time efficiency, using a set of RDMS indexing types to hold the Genome DNA sequence representation using the wavelet transformation. Then we have used different index types with two situations to hold the sequence representation. First situation is single field indexing and second one is multiple-field indexing.

The building of index structure must take into account the storage space, search time requirement and the query properties. Our structure provides dynamic facilities due to the parameters that can be optimized and refined according to the required needs. This flexibility allows index structure to fit at different environments according to the available resources.

We introduce a new indexing structure; the Modified Wavelet Transformation and BTRee (M-WTBT) that has some properties combined with other algorithms which are suitable for Genome DNA sequence. The main objective of our work is to get advantage of the used technologies to build index structure that can be loaded partially into main memory and defeat other index structure methods efficiency.

The proposed index structure building process, the M-WTBT, can be divided into three parts mainly, according to the techniques being used. The first process is data optimization using Haar wavelet transformation and changing data domain from the character domain to an integer domain. During transformation, which will output the final substring representation value (called the weighted

summation, *Wsum*), the amount of collision will be reduced using an enhancement to reduce number of I/O operations.

Secondly, the *Wsum* values will be loaded into modified BTree structure for indexing and searching. The modified BTree structure is used for efficient storage space, searching time and after evaluation. To save index storage space, an enhancement is introduced by storing *Wsum* difference instead of the actual *Wsum* values. For a query searching, full alignment will be applied for searching. We suggest two options for full alignment, either to use of the well known method (exact searching methods) or using our method for exact full alignment. Our method of full alignment is based on comparing the *Wsum* value with the index data using the *Wsum* value and *Wsum* location or with the *Wsum* value only.

In the last step, parameterization is available to be used. The M-WTBT structure can be adapted to fit at the available resources according to implementation parameters (window size W_x, *Wsum* range parameters, searching query, and the modified BTree branching factor N and number of levels L). Also partially loading of index is followed to minimize main memory requirements.

Our index structure had been implemented using Matlab for data transformation from character domain to integer domain and using C++ programming language to build the index and do searching. The main purpose of doing implementation is to show index structure performance by comparing our results with precious works.

1.3 Thesis organization

The thesis report is organized in six chapters. Chapter 2 provides a review of previous approaches and studies for indexing and searching Genome DNA sequence, studying of different index structures used by relational database management systems (RDMS) for evaluations and increase our understanding of the nature of sequences characters. Also transformation and compression had been discussed and analyzed as a choice for enhancing index performance.

Chapter 3 shows the efficiency of using different relation database management system index types. We have compared the use of transformation with two types of index structure, single-field and multi-fields index structure under the RDMS environment, using for different index types. Experiments results show the need to carefully consider index size and search time while using indexing structure.

We proposed our index structure "M-WTBT" in more details at Chapter 4. An implementation and set of experiments to compare the total size with other methods for the same Genome DNA sequences is accomplished at Chapter5.

Chapter 6 concludes the thesis with a brief discussion of the achieved results and gives some outlook for future work.

Chapter 2

Background and previous work

This chapter provides general introduction and surveys previous work related to Genome DNA sequence searching and indexing. The three main topics that are mostly related to existing solutions are:

- Literature review of previous work and methodologies related to sequence searching and indexing technologies for Genome DNA sequence.
- Data transformation used to preprocess sequences before building index. Main transformation techniques that are applicable for Genome DNA sequences for indexing systems are discussed.
- Full sequence alignment methods are introduced including Global and local alignments, Dot matrix, and Dynamic programming.

2.1 Genome DNA sequence searching and indexing

Genomes of different organisms are represented in long text, called sequence, of characters formed from the four alphabet letters (A, C, G, T) of the DNA bases. The four characters of Genome DNA sequences are adenine - A, cytosine - C, thymine - T, and guanidine – G.

Genome DNA sequences size measured in base pairs (bp), each pair occupies 1byte. The main Genome database systems are:

- Nucleic Acid: 'EMBL'[*], 'DDJB'[†], 'GenBANK'[‡].
- Protein: Swiss-Prot[§], Protein Information Resources (PIR)[**].
- NCBI: The National Center for Biotechnology Information[††].

GenBank database has reached 85Gbp at 2008 [‡‡] and its size is doubling every 18 months, as shown in Figure 2.1.

---

[*] www.ebi.ac.uk/embl/
[†] www.ddbj.nig.ac.jp
[‡] www.ncbi.nlm.nih.gov/Genbank/
[§] http://www.uniprot.org/
[**] http://pir.georgetown.edu/
[††] http://www.ncbi.nlm.nih.gov/
[‡‡] NCBIGenebank, http://www.ncbi.nlm.nih.gov/Genbank/S.

DNA consists of too long chains of nucleotides [78]. Nucleotides are molecules (an electrically neutral group of at least two atoms held together by chemical bonds) that, when joined together, make up the structural units of RNA and DNA. The size of data in the Whole Genome Shotgun (WGS) project is about 109 Gbp [40].The huge growth of sequence data (Figure 2.1) in GenBank can not be avoided and challenging to manage.

The process of searching the Genome DNA sequences to find a match for a query sequence can be seen as follow: given two sequences S_1 of length $|S_1|$ and S_2 of length $|S_2|$, check if there is a full match or semi-match between S_1 and S_2. Solving this issue require $O(|S_1|.|S_2|)$ time complexity and $O(|S_1|+|S_2|)$ space complexity in simple matching algorithm.

For example: if S_i= {GGCGTTAGAAGCGTTTCA}, and Ci= {A, C, G, T}, we want to do searching for Q_1= {AAGCGTTT}, and Q2= {GTTCGATG} over S_i for full matching only.

Finding longest match require passing Q_1 over the total length of S_i and each pass consider one character from Q_1 for matching. So the big O notation will be O ($S_i + W_1$), and the same is true for the second query (Q_2).

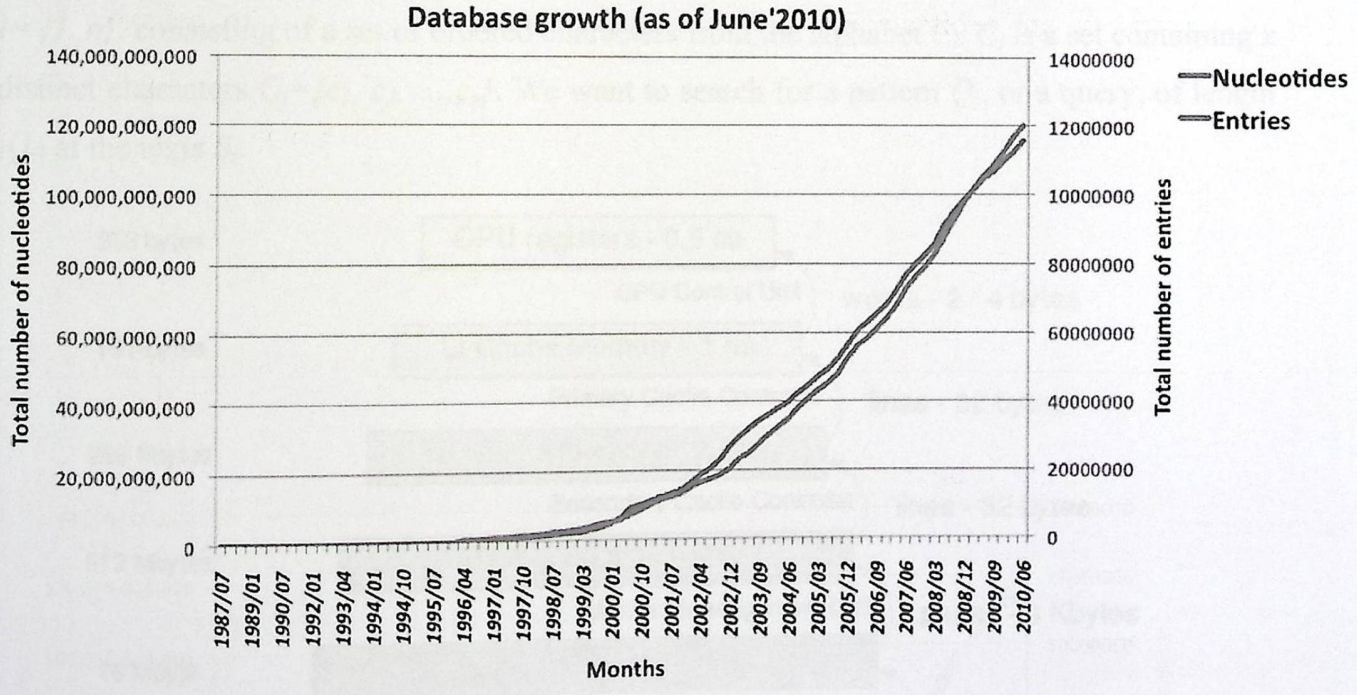


Figure 2.1: Exponential growth of genetic data due to next-generation sequencing technologies [13].

Different applications and search engines with a huge amount of data, like books, portal systems, and genome sequences, must answer queries quickly in spite of the huge data amount. Processing large size data, for different applications, must use indexing structure nowadays to achieve a good performance.

Direct access to data file requires I/O operations to access the disk, which is slower than accessing in the main memory. The total disk access time equals the "Seek time + rotational delay", where both are dynamic functions and take a long time. Different levels and types of memory, according to CPU design, are used in computer architecture which leads to a tradeoff between the size and the performance as shown in Figure 2.2. The closer the memory to the CPU, the less the size and the higher the bandwidth of data transfer.

Most of the data transferred between hierarchy layers, shown in Figure 2.2, is more than what is actually needed, this is called pre-fetching. Pre-fetching is to load more data than needed using the available free blocks at the layer.

The main objective of DBMS performance is to minimize the number of disk accesses (I/O operations). Database indexing is an upper layer of the whole system structure; bellow is the file layer and the disk layer. Each layer has its own manager to enhance the overall performance. Each logical data item will have a representative which will be accessed instead of the physical data stored at disk layer.

The main general idea of Genome DNA sequence searching has a long texts S_i of length $|S_i|$, $i= \{1..n\}$, consisting of a set of ordered characters from the alphabet C_i. C_i is a set containing x distinct characters $C_i=\{c_1, c_2, ...,c_x\}$. We want to search for a pattern Q_i, or a query, of length $|Q_i|$ at the texts S_i.

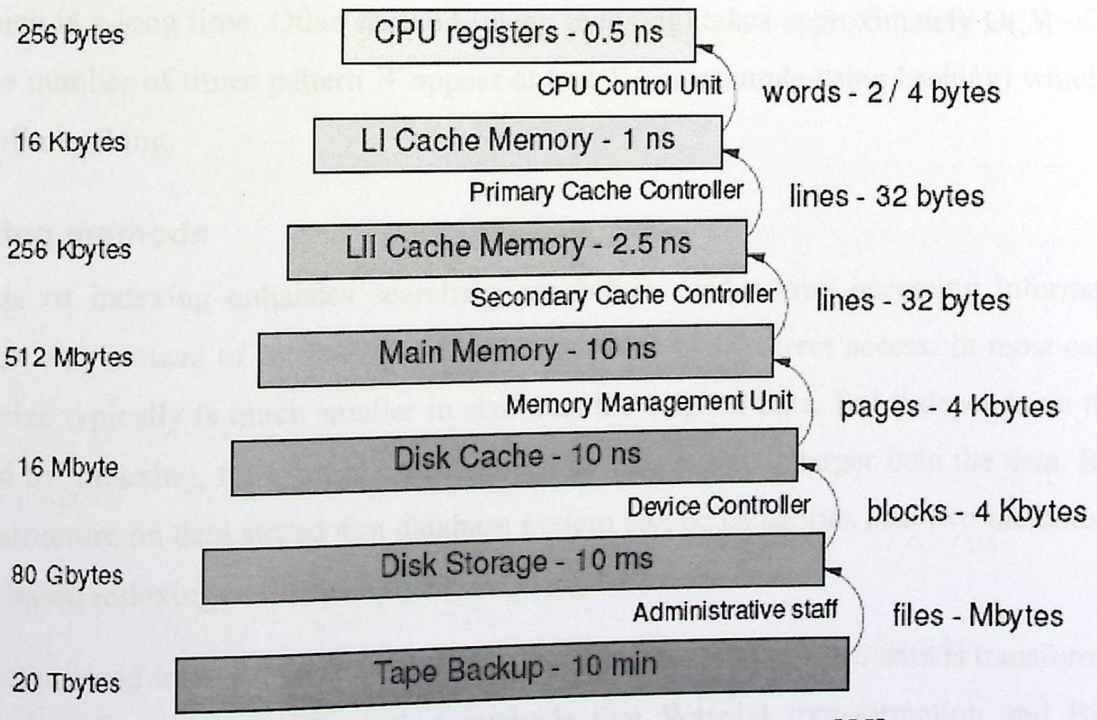


Figure 2.2: Memory Hierarchy at different layers [55].

2.1.1 Sequence searching methods

Sequence searching field has vast a large number of researches and extensively studied in last years. Concerning the Genome DNA sequences most works are based on transforming the sequence and building index structure on the transformed format. Efficient searching technique need to work on efficient index structure, in term of space and time, to defeat other methods for searching.

This section introduces different methods and structures of previous work related to data searching and indexing. The main two approaches for DNA searching methods are sequential scan and indexed based searching methodology. Full text based indexing and different methods based on tree structure, like SST Bed-tree and DMP tree, have been discussed through the following sections. The interesting point about most different Genome DNA sequences searching methods is the availability of compressed versions of searching and indexing techniques.

Finally, we briefly introduce main relational database management systems (RDMS) indexing types, like bitmap and hashing, used by different commercial RDMS applications. More details concerning RDMS are introduced in Chapter 3.

2.1.2 DNA sequence indexing

The search time for full matching will take $O(|S|+|W|)$ complexity, for a two substrings S and W, which is a long time. Other method (using indexing) takes approximately $O(|S|+£)$ where $£$ is the number of times pattern W appear at text S (for example using hashing) which is less than full matching.

Indexing methods

The use of indexing enhances searching capabilities and allows accessing information in efficient way instead of retrieving original larger data using direct access. In most cases, the index size typically is much smaller in size than the original data. But it depends on the used method for indexing, for example the inverted file index size is larger than the data. Building index structure on data stored at a database system can be categories into two methods, either vector based indexing or distance based indexing.

- Vector based indexing [45]: before building the index, the original data is transformed to a new format using transformation methods like Wavelet transformation and Burrows-Wheeler Transformation. The index structure builds upon the new transformed data from the original sequences, which is called a feature representation of original data. Picking

the transformation method is important and related to the type of the data. Data compression can be used either before the transformation of the original data, or after the transformation on the new feature data. When comparing by a new query for matching, the distance function used reflect the real distance from the original data. Most distance functions used for vector based indexing are less than or equal the edit distance (discussed later), in other words a lower bound to edit distance.

- Distance based indexing [3, 51]: database items are put into categories; each category has one item representative. The representative is chosen by comparing items with each other using a measurement method. For new query, distance will be measured according to representatives, then it will be compared with each sequence belong to the picked category.

The measurement used to calculate distance between sequences is mainly based on Edit Distance (ED), bellow we will discuss ED in more details. Our work is based on vector based indexing.

Genome DNA sequence index building

To build the index for Genome DNA sequence we have two ways, either Full-Text indexing or Word-based indexing:

Full-Text indexing [79]: build index for the whole sequence, each location (or character) at sequence corresponds to an appearance of at least one representation at the index, like inverted files, and Bitmap indexing.

Word-based index [41]: sequence processed as a list of words (of length W_x), where the location at the word of one character does not corresponds to any representation at the index like Suffix tree, Suffix Array, and BTree.

Our index structure will follow Word-based method. Building indexing structure strategies needs to consider the following important factors:

- Filtration step: decreasing number of candidates will optimize the number of I/O operations because of the large size of original data needed to be referenced which is not loaded at memory.
- The effect of query length on searched text (Genome DNA sequence): The DNA query length (related to the sliding window W_x for word-based indexing) affects mainly the index size and the searching method. Increase W_x value will increase index size, Section

3.6. Searching the database index using W_x value for a query with length $2W_x$ can be done in parallel by splitting query into two equal parts and do intersection between candidates. If query length is $3W_x$ and index is build using different window sizes (W_x, $2W_x$... nW_x), searching can be done more than one time using different W_x values (W_x and $2\ W_x$). Then intersecting between candidates will be applicable to reduce I/O operations.

- Index size must be small enough to fit in memory, or part of the index, if needed. The use of compression on index data to decrease the size while not increasing calculation rapidly which may slow down searching time.

Edit Distance

Edit Distance (ED) [73] is a standard distance method used to measure the similarity between two compared items (for Genome, it is called sequences). The edit operations used for measurements are inserting, deleting, and updating. Other types of distance measurements are weighted edit distance, Jaccard Distance, Jaro-Winkler Distance, Hamming distance, and TF/IDF Distance [73].

Edit distance measured by the number of operations needed to transform a sequence S to become identical to sequence W for any two sequences S, and W with lengths $|S|$, and $|W|$. The main problem with ED is the high complexity, $O(|S|\cdot|W|)$. This complexity makes ED not practical to be used with long sequences, and need to be optimized for different applications. Bellow the detailed algorithm for ED:

Algorithm 2.1: The ED algorithm steps:

```
Input:
     Two sequences (S and W) sequences, lengths are |S|, and |W|.
     Processing matrix M_LSxLW
Output: number of inserting, deleting, and updating operations.
Process:
     If LS=0 then return LQ.
     If LQ=0 then return LS.
     Construct comparison matrix of size (|S| rows, |W| columns).
     Fill first row with W chars, and first column with S chars.
     Loop i = 1 to |S|
          Loop j = 1 to |W|
               Compare S[i] <> W[j] as
                    (if S[i] = W[j]) {cost = 0} else {cost = 1}
               Now store value at matrix cell d[i,j] = MIN of{
                    M[i-1,j] + 1 OR
                    M[i,j-1] +1 OR
                    M[i-1,j-1]+cost
               }
     Return distance at d[|S|,|W|]
```

The main application domain for ED is Spell checking [73], Speech recognition, Handwriting Recognition, DNA analysis, Plagiarism Detection, File Revision (track changes), and Remote Screen Update (remote updates from more than one user).

Almost all works try to build a lower bound (D) for the edit distance (ED). In general we can say that for three strings S_1,S_2 and S_3, if $D(S_1,S_2) > D(S_1,S_3)$, then $ED(S_1,S_2) > ED(S_1,S_3)$) for any distance measurement method D. This is the core principle that all distance methods are based on.

2.1.3 Search query measurement

Searching methods for substrings at a sequence can be divided into two groups; approximate matching and exact matching. Approximate matching uses different algorithms and will lead to different results of searching and requires further filtration. Exact matching will always get one answer for the same keywords.

To find the similarity between any two sequences, there are two main methods according to the measurement being used:

- Equality selection queries: measurement is based on equality operation, only record with an exact specified value according to given value in the field is considered valid and returned.
- Range selection queries: no exact matching is required and can be one of the following two types:
    1. K-nearest neighbors (KNN) queries: This method finds the k sequences that are closest to the query sequence according to a specific distance function like Edit Distance or any other distance functions.
    2. Range queries: For a given range from a given distance value, find all the sequences that are within this range, no matter of the numbers of returned sequences.

The use of both methods, KNN and range queries, are related to similarity threshold value specified according to user needs. For our work we do exact measurement not similarity measurement so none of these methods will be used.

2.1.4 Sequential scan methods:

Sequential scan means to scan each data record, starting from first location till the end, for comparison. We will introduce the main methods used for sequential scan of Genome DNA sequences, which include Basic Local Alignment Search Tool (BLAST), Dynamic programming, Pair wise Sequence Similarity, Binary masks, Multi-Resolution index Structure (MRS), Random Access Blocked Inverted index (RABI), Self-index, and Scaling indexing.

Basic Local Alignment Search Tool (BLAST)

BLAST [31] technique used to find local similarity [22] between the query and the database and not the global similarity (see Section 4.2.6.1 for more details about local and global similarity). BLAST used heavily for biological homology searching on genomes and protein sequences since 1990 [61].

BLAST technique requires a string matching tool that passes through two phases:

The first phase: search all database sequences for an exact matching according to a fixed pre-specified matching length W_x (3 for protein and 11 for DNA). Suppose the exact matching is founded at the location *i*. The value of W_x presents a tradeoff between the sensitivity of the searching time and memory requirements. Increasing W_x value lead to increase memory requirements and decrease searching accuracy. While decreasing W_x value lead to more false negative candidates and more computations which will slow down the system.

The second phase: using a threshold *t*, continue searching after the exact match at both direction, left and right, for distance more than *i* and before *i-w* till threshold *t* is exceed. It stores pointer for location *i* to speedup first phase, so space needed is more than the database size. Searching time is liner with respect to sequence length, which makes this method not practical with large sequences length searching.

There are several different portals applications available online for sequence comparison (BLAST, and FASTA*) that can be used with different scoring systems (PAM250 [74], and BLOSUM62 [16]) and different databases (NCBI, SWISS-PROT, and GenPept).

BLAST has a set of variations like Nucleotide-nucleotide BLAST (blastn), Protein-protein BLAST (blastp), Position-Specific Iterative BLAST (PSI-BLAST), Nucleotide 6-frame

* FASTA Sequence Comparison at the U. of Virginia, http://fasta.bioch.virginia.edu/fasta_www2/fasta_list2.shtml, 2006.

translation-protein (blastx), Nucleotide 6-frame translation-nucleotide (tblastx), Protein-nucleotide 6-frame translation (tblastn), large numbers of query sequences (megablast).

Below is a little introduction about some of the BLAST variations [31, 61]:

- Blastn: This program search for a DNA sequence, it returns the most similar sequences to a given query.
- Blastp: This program is similar to Blast, but it works for protein sequences instead of DNA sequences.
- PSI-BLAST: This program is used to find distant relatives of a protein. As it includes related proteins in the search, PSI-BLAST is much more sensitive in picking up distant evolutionary relationships than a standard protein-protein BLAST.
- Tblastx: This program is the slowest of the BLAST family and its purpose is to find very distant relationships between nucleotide sequences.
- Megablast: It is used to compare large number of sequences; "megablast" is much faster than running BLAST multiple times.

BLASTn and BLASTp are the most commonly used because they use direct comparisons, and do not require translations.

Dynamic programming

Dynamic programming [30, 48] has time and space complexity of $O(|S|\cdot|W|)$ for two strings S and W of lengths $|S|$ and $|W|$. And for database comparisons Dynamic programming needs a matrix of size $|S|x|W|$. Hence for long sequence and large database this method will not be practical in term of both space and time. Dynamic programming calculates the distance between S and W using a heavy computation method; the edit distance.

Pair-wise sequence similarity

To compare the similarity between two sequences, the naive method is to compare one character from the first sequence with the corresponding character from the second sequence. This process is repeated through the whole sequences length, where the number of possible alignments is exponential in the length of the sequences. This method called pair wise comparison [24, 45] with $O(|S_i|\cdot|W|)$ space and time complexity, for two sequences S_i and W with length $|S_i|$ and $|W|$.

The comparison between the two characters can be measured by two methods. First method is a match or not matches between the two compared characters between sequences. Second method is a similarity score value from 0 to 1 shows percentage of matching using a similarity matrix.

Other pair-wise alignment methods uses scoring functions where different scores can be assigned to different alignments and some methods consider the gaps for scoring.

Binary masks

The use of r Binary masks [30], $M_1, M_2 ... M_r$ of size m, to move through S of size n, by word size W_x. The complexity of binary masks is $O(nmr/W_x)$. And for a large value of m, this complexity will be very close to dynamic programming.

Dictionary based index

For a database of sequences S_i, $i=1,2,...,n$, create an index structure of size n corresponding to database size using a predefined query lower bound length (L). All substrings of length L mapped to integer using a hashing function. For queries larger than L, split the query into sub-queries, then search each sub-query alone and combine the results. This method indexes all possible strings of a pre-specified length L. This method is called dictionary based [30] and referring to reference, its index size is larger than the database.

Multi-Resolution index Structure (MRS)

The Multi-Resolution index Structure (MRS) [31, 19] uses a sliding window of sizes 'W_{x1}, W_{x2},.. W_{xj}'. The window will slides c times, which is the capacity of the box, as shown in Figure 2.3.

MRS seeks the result set in different resolution levels. However, the authors of [31] only focus on the cost of MRS, and do not evaluate the filtrations efficiency of their proposed technique. This process is applied on all database sequences, starting from W_{x1} till W_{xj}. Figure 2.4 shows the final structure after applying W_x over all sequences using MRS method. Each box at Figure 2.4, like T1, 1, is the output represented by Figure 2.3. After scanning all sequences (s1 till sd) and scanning using all window sizes (W_{x1} till W_{xj}), we get the resulted displayed at Figure 2.4.

Random Access Blocked Inverted index (RABI)

The use of inverted index for retrieval of full text systems for full text retrieval systems improve time and space for large scale data size.

The random access blocked inverted index (RABI) proposed to improve the retrieval performance for the inverted index in large-scale full-text systems. The blocked inverted index [76] consists of index file (distinct terms over all data) and a set of inverted lists (posting files) with large-scale full-text system.

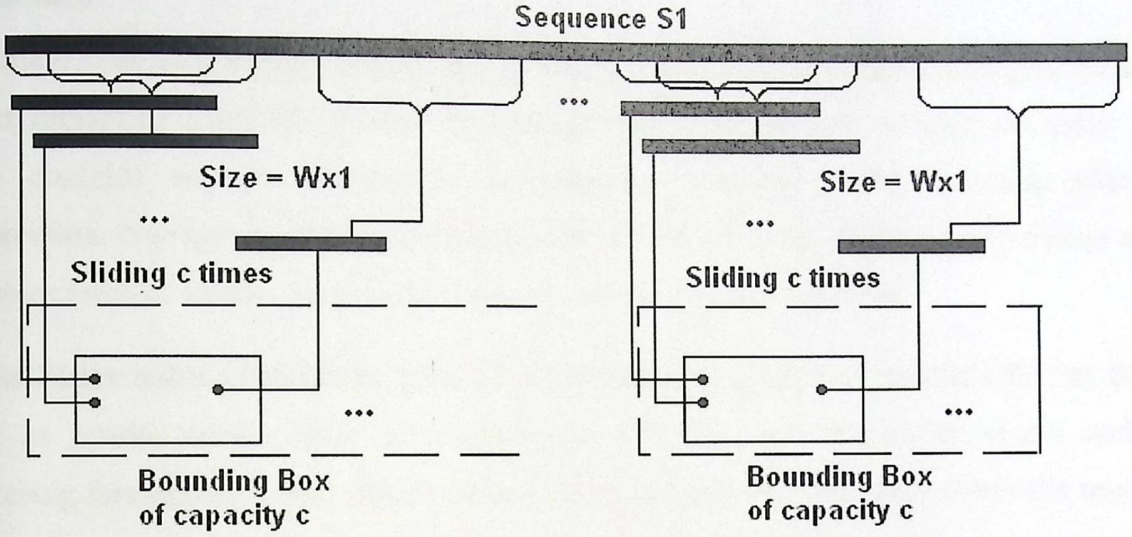


Figure 2.3: MRS internal nodes structure.

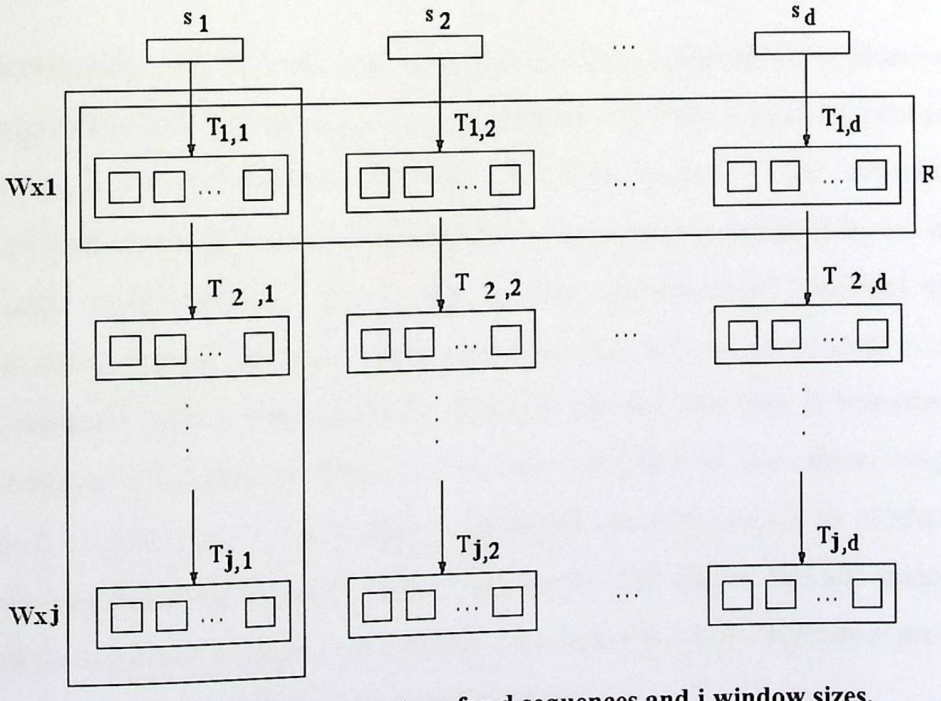


Figure 2.4: MRS index structure for d sequences and j window sizes.

RABI method solves two problems: The high storage overhead and considering search query structure with differentiation between short and long queries. This method follows blocked inverted index with skipping approach and propose the random access blocked inverted index (RABI) which enhance space and storage efficiency. This method divides index into blocks and do compression to different parts of the block using encoding method. For compression it uses Binary Interpolative Coding (BIC) method. Access is done at both levels block level and inner block level.

Self-index

Self-index [28] is a highly compressed full-text index, uses delimiters and give an upper bound, limited by a number of bits of permanent space in worst case, to build the index. Self-index contains enough information to efficiently reproduce text substring after the compression. Storage experiments compare the effects of using stuffing performance and it examines three processes: the construction, the recover, and the retrieval.

Minute Space index (FM-index) is the first self-index developed to provide efficient storage space as results shows, close to compression methods, and to provide search and text recovering functionality with almost optimal time complexity. FM-index combines sequence compression with indexing, not with original sequences, all process is done on the compressed data. FM-index returns two values, the number and locations of occurrences of query W at text S.

The main functionalities of FM-index based upon the relationship between Burrows-Wheeler compression algorithm and the suffix array data structure as follow: start by process the input text T[1,..,n] using Burrows-Wheeler Transform (BWT) to produce L (the permutations of T), Run Move-To-Front encoding (as is" each symbol in the data is replaced by its index in the stack of "recently used symbols". For example, long sequences of identical symbols are replaced by as many zeroes, whereas when a symbol that has not been used in a long time appears, it is replaced with a large number. Thus at the end the data is transformed into a sequence of integers; if the data exhibits a lot of local correlations, then these integers tend to be small.") on L to produce L'. Then apply run-length encoding on L' to produce L''. Last step is to apply variable-length prefix code. This work [28] shows the advantages of using FM-index with the addition of adding delimiters. FM-index is a CPU intensive process; so the use of parallel processing can enhance the performance.

FM-index is a sort of compressed suffix array that takes advantage of the data compression properties in order to minimize space close to the best. It is difficult to build the FM-index for large text because of the large memory requirement. To overcome this limitation, a large text is split according to an overlapping.

Mathematical expression indexing

Handle mathematical structural text and mathematical operation [26] by indexing real-world scientific documents containing mathematical notation based on full text searching. The mathematical indexing address the following issues, extraction and storing of mathematical notation, and ranking function for notations relevancy. This method differs from previous methods for mathematical notations, it starts searching by exact matching. If the exact matching for the notation is not successful, the searched query is generalized and search is repeated again.

BNDM matching algorithm

BNDM [9, 16] is a matching algorithm by [18], used for exact string matching, it is an update over the Backward Nondeterministic DAWG [23]. DAWG is a directed graph, uses the smallest suffix pattern, and extremely fast word searches. The alignment process is done by N-gram word size. This method has good performance for patterns of length range between 5 and 30. BNDM algorithm scan string from right to left for matching either a full matching of the query string or no matching even for substring, of a predefined length W_x. The matching is done using a sliding window with amount of overlapping defined previously. Figure 2.5 shows an example for BNDM. Notice that after step 4 at Figure 2.5, it fails to recognize the next character (A), so shift the window to the last location and start search again.

Query = [AACCAAC]

1. S=[ACCACAACCAAC]
2. S=[ACCACAACCAAC]
3. S=[ACCACAACCAAC]
4. S=[ACCACAACCAAC]

FAIL after point '4'

Wx

5. S=[ACCACAACCAAC]
6. S=[ACCACAACCAAC]
7. S=[ACCACAACCAAC]
8. S=[ACCACAACCAAC]
9. S=[ACCACAACCAAC]
10. S=[ACCACAACCAAC]
11. S=[ACCACAACCAAC]

Figure 2.5: BNDM example for sequence S to search for Query [AACCAAC].

Scaling indexing:

The scaling pattern [27] method based on enlarging or shrinking W_x value until an extent match is founded, to build index using this pattern. Scaling indexing is almost the same as suffix array. It takes $O(|S_i|)$ time for processing step and $O(|W|+M+logm)$ time for query

searching step where $|S_i|$: length of sequence, $|W|$: the query W length, M: the locations query W exists at in different sequences, m: length of $|S_i|$ under run-length representation [42] (see Run length encoding Section 2.3.3 for more details).

2.1.5 Full text indexing

Full text searching using N-gram [21, 14], of N characters, processed for searching starts with 2-gram index then supplement with higher N-gram index. Frequently used terms are selected for incremental indexing. The incremental index has two main functional engines, the search engine and the index creation engine.

For a long sequence, the number of "AND" operation is large which cause low performance for searching. The incremental indexing solves this problem by carefully selecting search terms using search intensive approach. Experimental results, done by [21] shows the effectiveness of incremental index even for further added terms. This method builds the incremental index upon subset of terms, not for all terms, to save space and provide efficiency for most search terms. The Genome DNA sequence indexing requires the index to be ready before searching, as the whole data is ready.

Programming kit

An ontology kit for full text searching [64] focuses on finding words related to a certain concept (using relevancy ranking function) from a set of concepts. This kit consists of three layers:

- Full text layer for full text indexing (a Word-based index) and full text searching,
- Ontology layer for concept definition and ontology maintenance, and
- User layer for the programming issues.

This kit made use of Apache Lucence and Jena development kits [http://lucene.apache.org/]. They work to get a relevancy ranking between documents that meet the query, which may be met by large number of documents. This kit is a sample of development kits used for evaluating index structure, this kit process depend on relevancy ranking rather than accurate matching.

File signature

The text is divided into a set of logical blocks; each block contains a number of distinct words which is called the signature. The number of distinct words determines signature width. The

word yields the word signature which is a set of bits. The use of signature of files allows the use of parallel hardware architecture [27, 67] for full text searching. This method control false drops, uses a suitable hashing function with buffer, and it has a good storage overhead. The parallel process can be applied for processing database documents and between documents too. It provide an option for don't care characters, if needed.

The Genome database containing Genome DNA sequences of different lengths, while file signature need database records to be of the same length [60] to have a good performance. The signature width needs to be considered carefully as DNA sequence has variant lengths, there is a maximum length and minimum length through the whole sequences.

If it is set to the maximum DNA sequence length, the index size will be very large and if set to average sequence length, the false match candidates will increases.

A proper method for selecting the signature width is to divide the database into sub databases with similar lengths. As we can see, file signature indexing method faces a serious problem with large number of Genome DNA sequences and with large sequence lengths; it results in a large index size.

2.1.6 Tree based indexing

Index structure can be developed depending on tree structure that can be used for index entries or for the input data. If tree structure is used for holding index entries, Genome DNA sequence will be transformed to new format mainly before filling the tree. For the second case, when using tree structure to hold the actual Genome DNA sequences, tree size is huge mostly and will need compression and efficient searching method. Through this section we will discuss the following tree structures: Suffix array and suffix tree, Sequence Search Tree (SST), Bed-tree, Cache-conscious SR+-tree (CSR+-tree) index, and DMP tree.

Suffix array and suffix tree

Building suffix array [62] requires O(n) time complexity and O(n) space complexity for constant size text and the Suffix tree requires O($n \cdot \log n$) time complexity and O(n) space complexity. Suffix array and Suffix tree are suitable for pattern searching. A Compressed Suffix Array [20] introduced that uses the output array of Burrows-Wheeler Transformation (BWT) with complexity O($n(\log_{\Sigma} n)$). The main function of compressed Suffix Array depends on developing a new algorithm using terminators. Notice that the complexity for

compressed suffix array is less than that for suffix array. (Conventional suffix tree and array takes $O(n \cdot \log n)$).

Suffix array scans the database using a window (W_x with overlapping amount Δ) and count repetition of all possible k-tuples of width equal W_x. It stores result at a vector of size σ^k (σ referred to the alphabet chars A, C, G, T). Then it indexes those vectors in hierarchical binary tree. To compare new query with those vectors, Suffix array uses ED distance method. Suffix array is 25 to 50 times faster than BLAST. Disadvantage of Suffix array method is the allowing of false drops and index size increase linearly with k value.

[72] Announces the suffix array which is an integer array to replace suffix tree, in 1990. While from that time suffix array had been neglected because of the slow construction phase.

Suffix array consists of lexicographical sorted suffixes of input data and a list contains starting location for each suffix. It takes $O(n \cdot \log n)$ to construct the array which is not efficient to replace suffix tree. Compressed version [71] of suffix array was used as a proposed solution.

Any edit operation on input data leads to rebuild the suffix array again. The work of [71] focus on solving this point, their algorithm is based on dynamic-Wheeler transformation (DWT) algorithm. Suffix array require almost 10 bytes per input letter, this limitation prevent suffix array from being used for the large size input data. New updated on suffix array methods, like [33, 75, 71], introduced to save space by encoding redundant information using compression.

After building the suffix array, the update operation is so much costly. Some algorithms like Dynamic extended suffix arrays by [43] maintain the already built suffix array, for the input sequence, to handle this issue by considering only the edit operations that are affecting the input sequence.

Some methods, like the work of [40], try making enhancement for the Suffix tree constructing method with data size larger than the available memory. Suffix tree belong to full-text indexing, this type make index for all different substring of the input sequence and produce suffix tree with size larger than the input sequences size.

As the size of input data increases, like Genome databases, old methods performance for building suffix tree decrease linearly with the linear increase in size. Two main problems

facing efficient construction of suffix tree: random traversal of the tree during construction and massive random access to input strings which lead to high I/O disk reference.

The increase in memory size in last years allows the building of suffix tree in better performance. The work of [40] announces a method to build suffix tree for input sequence larger than memory size. Input sequence and suffix tree both will be stored at disk but with minimum I/O reference. This work builds the suffix tree for 12 GB of real DNA sequences in 26 h on a single machine with 2 GB of RAM.

The string Suffix Binary Search Tree (BST) construction algorithm is used for efficient external-memory suffix tree construction for very large input data. The BST algorithm works as follow: partitioning the input (using the 'input data to memory size' ratio) for building suffix tree, sorting the suffixes in each partition pair, and efficiently merging the sorted suffixes into a suffix tree. BST minimizes the random access to the input string.

Construction of a suffix BST for an n-long string can be achieved in $O(n \cdot h)$ time, where h is the height of the tree. It performs sequential access on the suffix tree and input sequence stored at storage device instead of random access.

Experiment result shows the good performance that BST algorithm has in building suffix tree while the 'input data to memory' ratio increases. Figure 2.6 summarizes the two structures of Suffix Tree and Suffix Array.

Sequence Search Tree (SST)

Using SST [17, 28], the sequence S will be divided into substrings according to window size Wx using multiple offsets Δ.

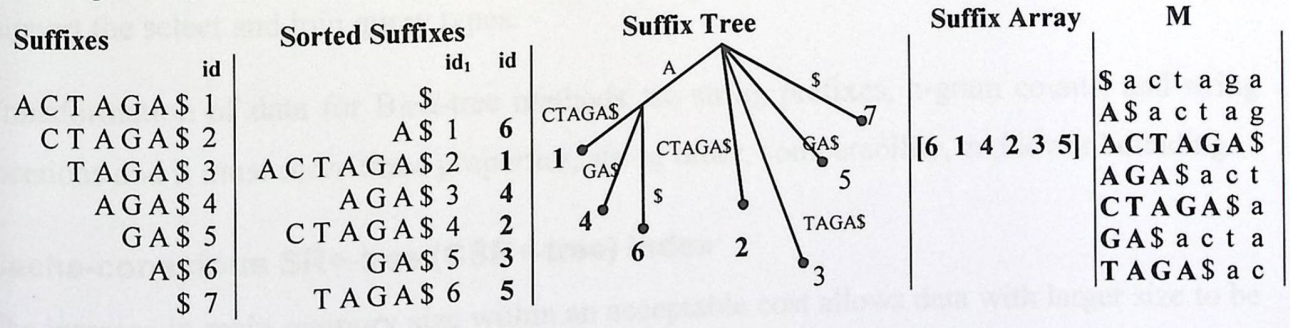


Figure 2.6: Suffix tree and Suffix array structure.

(tree, numbers from first left, array from second left)

Typical W_x values are less than *1000* and $5 \leq \Delta \leq W_x/2$. Each substring results from using W_x is mapped into a vector of dimension 4^k, k represent the frequency of occurrence of the k-tuples.

(The I'th entry in the vector is the number of occurrences of tuple I in that window, windows= W_x, tuple =x).

SST in general, searches the Genome database of sequences for near-exact matches. It scans the database according to window size W_x and map results to a vector, then hierarchical clusters built using k-means algorithm. For any new query need to be searched against the database, the cluster mean is compared and neglect clusters that are far away from the new query according. It is searching for nearest-neighbor windows in the database and can be used as a filtration process.

Some disadvantages of SST are the complexity of calculations, false clustering, and time complexity is proportional to database size.

B$_{ed}$-tree

String search for similarity (not an exact match) can be divided into two types, search based on measurement like edit distance based search and search based on tokens. Distance based search, where comparison between strings depends on number of delete, insert, and updates operations, converts one string to be identical to the second string. And search based on tokens, like n-gram search based, compares two strings, substrings, or words, of a predefined width (W_x) with each other.

The Bed-tree [78] index structure is a B+Tree based index structure for string similarity search (using range queries) using the edit distance measurement, and normalized edit distance methods for similarity search queries. The main use of Bed-tree is by select and joins queries, Bed-tree works on data transformed from string domain to a new domain to support the select and join query types.

Transformation of data for Bed-tree methods are string prefixes, n-gram counts, and string locations and it must serve three properties, string order, comparability, and lower bounding.

Cache-conscious SR+-tree (CSR+-tree) index

The increase in main memory size within an acceptable cost allows data with larger size to be loaded at main memory which was not applicable in previous time. Many index structures like SS-tree [7], and SR-tree [47] was large enough to not fit at main memory. Memory cache is a factor that has big effect on index structures, cache memory levels L_2 and L_3 size and topologies affect miss operations. Miss operation causes a main memory reference which is slower than cache.

The cache-conscious SR+-tree (CSR+-tree) [77] index structure is used for height-dimensional similarity search, where index is loaded at main memory. The distance schema used does not require decompressing index items. This method eliminates the need for decompression before computing distance function on original data with the query in case input data is very large.

The CSR+-tree is based on the quantized (taking a seemingly infinite range of possible values and dividing it into specific steps) bounding spheres (QBSs) that approximate the bounding spheres (BSs) which are the data points and the minimal bounding around points. The QBS is an enhancement over the quantized bounding rectangle (QBR). They apply dimensionality reduction to the input data first, and then build an index structure over the reduced data set.

Dynamic m-way prefix tree (DMP)

DMP [46] tree is a general matching and prefix matching (Example: SA = {"ab", "abd", "abdf", "abz"} P = "abd") method. DMP functions upon the assumption that all strings consist from the alphabet $\sum$ with different lengths, and some of the strings are prefix of other strings. DMP-tree become BTree if no prefixing is exists between strings. DMP tree does not allow substring to be in a higher level than another substring in case it is a prefix of it. This requires a special procedure for node splitting. A word splitting is the difference between DMP-tree and BTree and makes BTree a special case of DMP-tree, where both built bottom up.

Main applications of DMP-tree are Routing, online dictionary, spell checking, general pattern matching, and telephone directory lookups data.

DMP tree first step is string sorting, where other method does not provide easily and efficient sorting when strings are of different lengths.

Experiment result show that the average height of DMP-tree is O ($Log_M N$), Logarithm of base M, for large N and big branching factor M in the internal nodes, this means that its performance is close to BTree. Search time for BTree is $Log_M N$, where N: number of elements and m: branching factor while the initial DMP tree will have the height $Log_M \frac{(\max imum\ string\ length - \min imum\ string\ length)}{M-1}$.

DMP tree is efficient memory usage and good performance with large scale data set and aimed to solve two problems: providing better search time than the binary tree (worst case for search is O (W_x), and solving the bad effects of the update operations on the index structure.

2.1.7 RDMS indexing

RDMS indexing systems is a tool of selecting data from the relational database using a key (index) and direct access input/output operations. Indexing supports simple (single variable) and composite (multiple variables) indexing systems. The Bitmap indexing, Encoded Bitmap indexing, and Hashing will be discussed during this section (refer to Section 3.3 for more details of RDMS index types). See appendix C for detailed list of different index types used by main RDMS like MSSQL server, MySQL, and Oracle.

Bitmap index

Bitmap index consists of distinct values divided into a set of vectors and used mainly for query optimization. The distinct values at the data determine the number of bitmap vectors. For indexing, each distinct value will need a bitmap vector [52], a bitmap vector will contains the value 1 for a relation row if match is found; other wise 0 will be the value. The size of bitmap index depends on the data cardinality; increasing cardinality will increase bitmap index size.

For a large cardinality, main memory will not be able to hold the whole bitmap index size. Different methods for updating index structure to do bitmap compression proposed to solve the increase of index size issue. First method: Binning [34] or bit slicing is used to update the bitmap index structure. Binning principle based on using values ranges instead of explicit values, so the relation row will belong to a range of value rather than having explicit value. Second method: Bit slicing use a mapping table and ordered bit values for encoding, *3* bit encoding provide $2^3 = 8$ distinct relation field values like the encoded bitmap index, which will be discussed later.

Figure 2.7 shows an example of using bitmap index for the 'Dept' field which consists of five distinct values (d) which are {1, 2, 3, 4, 5}. As figure shows at right side, index size equal to number of rows (r) multiplied with number of columns (d) multiplied of cell size (which is 1 byte) $= r \cdot d \cdot 1$.

Encoding Bitmap

If Bitmap index applied for big range cardinality field, it will need bitmap vectors equal to the number of field distinct values which will be a large number. To decrease this number, Encoded bitmap [42] can be used. Figure 2.8 shows an example of encoded Bitmap index, as we can see the number of Bitmap vectors decreases by a logarithm relation to the base '2' as

we used 0/1 value only. Storage space will be reduced; we need 3 vectors and the mapping table.

Mapping table size

The mapping table consists of two columns 'A', and 'B'. Column 'A' represents the target and column 'B' contains the distinct Bitmap vectors. In general, we can say that bitmap index can be used with database if the update operation done rarely or at schedule time. In other word update, insert, and delete operation done at a time where no reading is required (low DML operations).

Hashing

The main idea of hashing [5] is to build a table using a certain hashing function (h(key) = value). For example h(AACG) = {2,1,1,0}, the hashing function is the frequency for the DNA characters {A,C,G,T}. Hashing table allow the direct access of data, it needs one I/O operation to access data item.

EMP_ID	Name	Dept
1001	AAAAA	1
1002	BBBBB	2
1003	CCCCC	3
1004	DDDDD	4
1005	FFFFF	5
1006	WWWWW	6

1	2	3	4	5
1	0	0	0	0
0	0	1	0	0
0	0	1	0	0
1	0	0	0	0
0	1	0	0	0
0	0	0	0	1

Figure 2.7: Bitmap index build on Dept field.

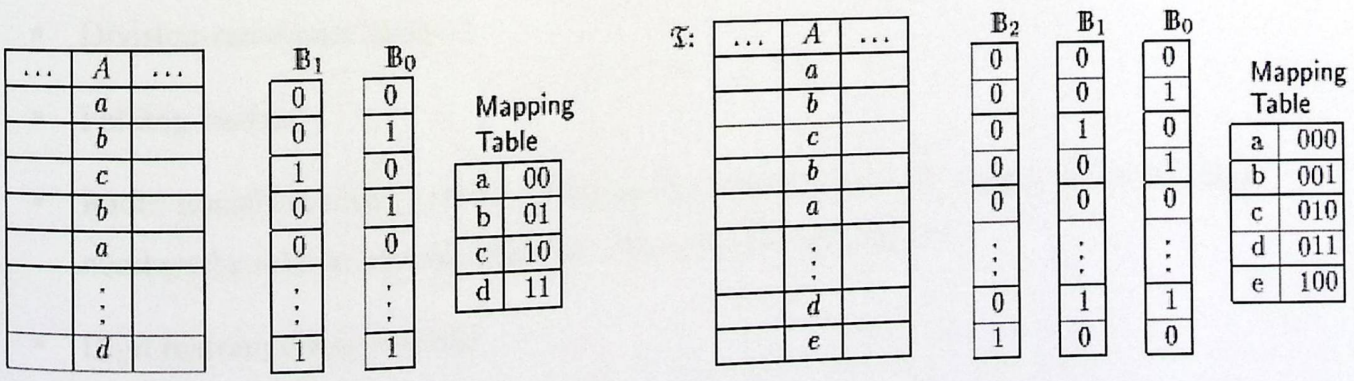


Figure 2.8: Encoded Bitmap index on the field "A".

Figure 2.9 shows the general idea behind hashing that ends up by storing data at block *i*, where *i* range from *0* to *n*.

Figure 2.9 part (b) shows another approach for storing data item, instead of storing data item directly we store another key like a pointer which has the reference to the data item.

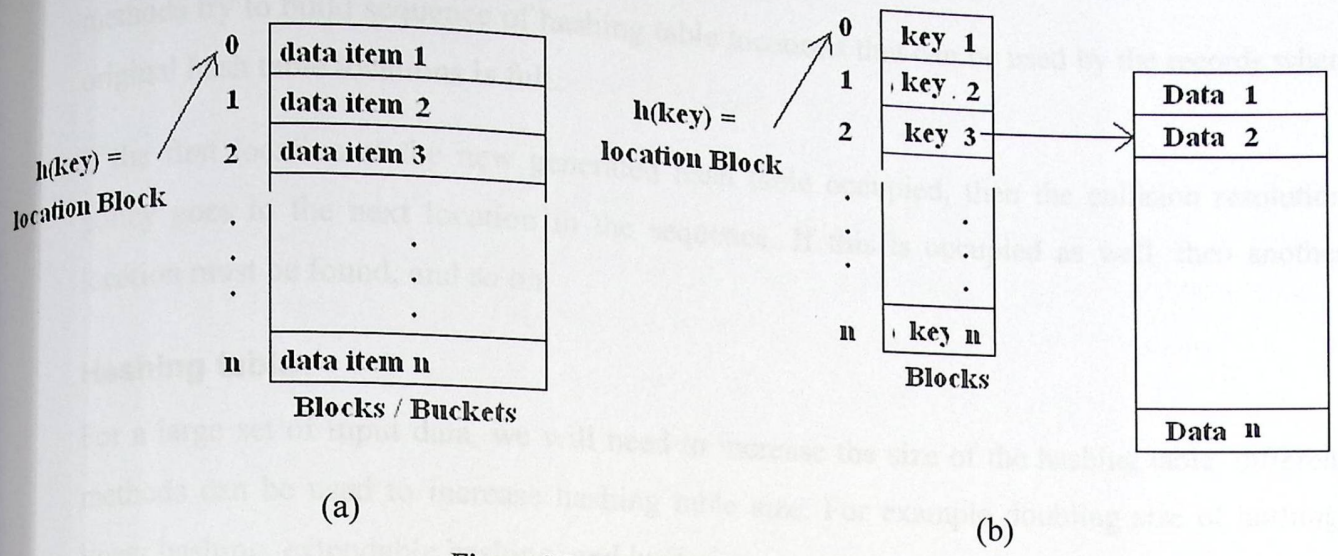


Figure 2.9: Hashing data items to data blocks.

Hashing function

The main purpose for the hashing function is to store data item at distinct and uniformly distributed location by an easy calculation. The general idea is that we have the data item '*x*' that will be stored at block h(*x*). One simple hashing function is the 'mod' over number of blocks. If the hashing function sends too much data item to the same block this will result in overflow, as storage space will not be enough as capacity is exceeded. We will need a backup storage space for overflow called overflow blocks, as shown in Figure 2.10.

The best choice for the hashing function is a random selection from a list of functions, this option make the hashing structure to behave well regardless of the value of the keys. Bellow are relatively simple hash functions that can be used:

- Division-remainder method.
- Folding method.
- Radix transformation method: changing the number base for digital values (decimal numbered could be transformed into a hexadecimal numbered).
- Digit rearrangement method.

Collision

Collision occared when the function h(key) put data item at the same location, which is occupied by another item. There are different methods to solve collision, called collision resolution, during the addition of new item. The aim is to find a free location in the hash table in case that home location for the data record is already occupied. Collision resolution

methods try to build sequence of hashing table locations that can be used by the records when original hash table locations is full.

If the first location at the new generated hash table occupied, then the collision resolution policy goes to the next location in the sequence. If this is occupied as well, then another location must be found, and so on.

Hashing table

For a large set of input data, we will need to increase the size of the hashing table, different methods can be used to increase hashing table size. For example doubling size of hashing, linear hashing, extendable hashing, and buffering.

The work of [35] "Fast exact string matching algorithms" is a string matching using hashing of n-gram string (word of length equal to *n*). They make adaptation to [65], instead of multiple string matching algorithms, a single string matching algorithm is used. This method is efficient with short word length from a small size alphabets comparing to the well known fast algorithms.

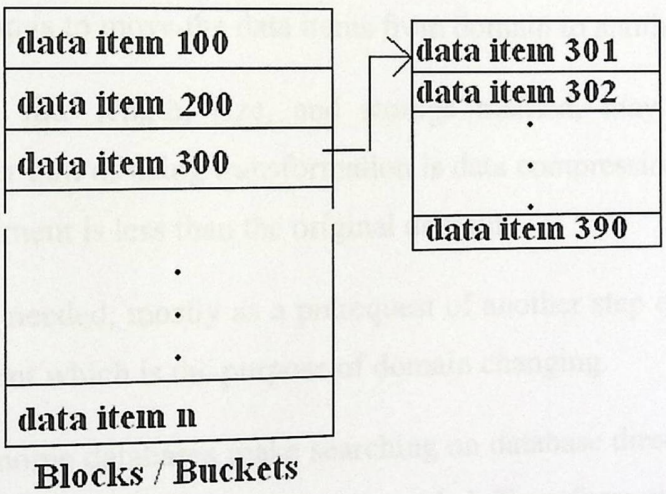


Figure 2.10: Overflow blocks for Hashing method.

Introduction to Tree structure

Tree is an undirected linked vertices where the link between vertices represents a path that is not a cycle. Tree structure has one root, the node at the top of the tree, for a set of nodes or items belong to the same root. Tree degree refers to the number of links for each vertex.

Tree structure used by computer processing has a directed links. Tree can be classified in different types [11]:

- Undirected tree: no specific direction between nodes like forward or backward.
- Irreducible tree: a tree where no vertex has a degree 2.

- Forest tree: it is undirected tree consists of disjoint groups of trees.
- Directed tree: all vertexes are directed to one vertex (the root) or directed out from the root node.
- Rooted tree: if all vertexes are connected a one starting node (called the root node) and the direction from vertexes and the root is either toward or away from the root. This type has spatial properties which is the ordering of neighbor nodes for each vertex. A tree without a root is called a free tree.
- n-ary tree: n refer to the number of children for each vertex, binary tree is an 2-ary tree.

2.2 Sequence transformation

This section will introduce main transformation methods that are applicable to be used with Genome DNA sequences. We will introduce Burrows-Wheeler Transform (BWT), Wavelet Transformation (WT) [30].

Suppose the input data is S_i, $i=1..n$, and let S_i domain be $\sum$. Sometimes it is hard to manage the input data directly, instead changing data domain may solve the problem. The aim of sequence transformation is to move the data items from domain to another domain.

Input data properties, like length, size, and storage method, may be chancged during transformation. Another aim of using transformation is data compression, spceillay if the new domain storage requirement is less than the original domain.

Data transformation is needed, mostly as a prerequest of another step of processing, to make processing more efficient which is the purpose of domain changing.

The extrem size of Genome databases make searching on database directly very complex and inefficient, as a result sequence transformation is needed. Transformation methods is used to reduce the time and the space needed for managing original data.

Transformation is used as a prior step before the compression algorithm, where instead of applying compression on the actual data, transformation improves compression by doing feature extraction. The transformation method can be used according to the desired compression algorithm that will be applied. Choosing the type of transformation is highly related to the type and the size of the data, which the transformation will be applied on. For our work, we have used WT on the Genome DNA sequences and not using BWT as described in next section.

2.2.1 Burrows-Wheeler Transform (BWT)

For an input string S_i= $\{S_1, S_2 \ldots S_n\}$, BWT* build all cyclic permutations for S_i by cyclic rotation over all items. The cyclic permutation does not change the order of items and each time one character from the end moved to first location.

For example for $S_i = \{ATGACA\}$ the cyclic permutations are: `ATGACA, AATGAC, CAATGA, ACAATG, GACAAT, TGACAA.`

BWT output is the last character of each permutation after ordering. There is no need to store all cyclic permutations for the original string S_i, a pointer or index for the start location is enough.

For the previous example, after ordering:

AATGAC-→{0} CAATGA-→{3}

ACAATG-→{1} GACAAT-→{4}

ATGACA -→{2} TGACAA-→{5}

Output of BWT for S_i = {C, G, A, A, T, A}, note that first column from left is the string S_i after ordering {AAACGT}.

Figure 2.11 is another example for BWT for input sequence {ACAACGT}.

A third example for the input sequence "AGTACA". where right rectangle shows the output at Figure 2.11.

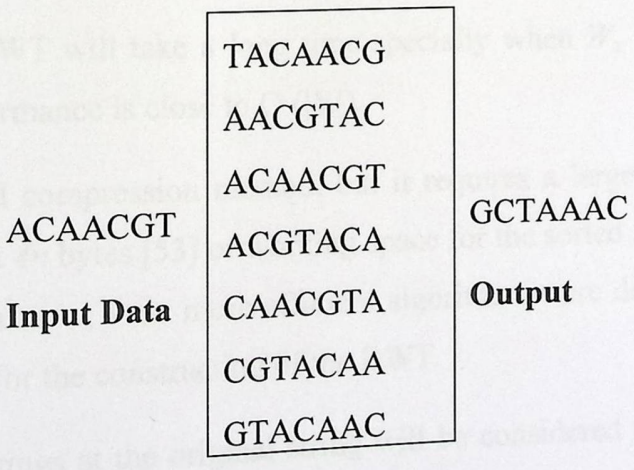


Figure 2.11: BWT applied on the sequence "ACAACGT".

*Portal website using BWT method: http://bwtools.polsl.pl/BWtrs/RepeatSearcher

1	A	A	G	T	A	C
2	A	G	T	A	C	A
3	A	C	A	A	G	T
4	G	T	A	C	A	A
5	C	A	A	G	T	A
6	T	A	C	A	A	G

Figure 2.12: BWT applied on substring "AGTACA".

Reconstruction of S_i after BWT

BWT transformation is reversible; the original order of the data elements can be restored with no loss of original items. Given BWT(w) for w and an index i, it is possible to recover the original word w.

The following figure shows an example of reversing BWT for the previous example at Figure 2.11:

TACAACG	TACAACG	TACAACG	TACAACG	TACAACG	TACAACG
AACGTAC	AACGTAC	AACGTAC	AACGTAC	AACGTAC	AACGTAC
ACAACGT	ACAACGT	ACAACGT	ACAACGT	ACAACGT	ACAACGT
ACGTACA	ACGTACA	ACGTACA	ACGTACA	ACGTACA	ACGTACA
CAACGTA	CAACGTA	CAACGTA	CAACGTA	CAACGTA	CAACGTA
CGTACAA	CGTACAA	CGTACAA	CGTACAA	CGTACAA	CGTACAA
GTACAAC	GTACAAC	GTACAAC	GTACAAC	GTACAAC	GTACAAC

Figure 2.13: BWT reversible on the sequence "ACAACG".

The use of BWT with Genome DNA sequence searching suffers from the following obstacles:

- It is not easy to do sorting for sequence permutations; we have small set of alphabet, a repeated sequences, and different window lengths W_x.
- Sorting with BWT will take a long time specially when W_x is a large value, sorting algorithm performance is close to O ($|S_i|$).
- BWT is a good compression method, but it requires a large size of memory. BWT requires at most $4n$ bytes [53] of working space for the sorted substrings (suffix array). Recently several new, even more efficient algorithms were developed that reduce the space required for the construction of the BWT.
- Repeated substrings at the original string will be considered as a new item by BWT, which is not needed for further indexing and will increase storage space.

Figure 2.14 show a comparison of different index structure sizes (BWT, Suffix tree, Suffix array, and Hashing) [4]. K-mer is a substring of length k characters and the hash table stores the number of occurrence of each k substring length.

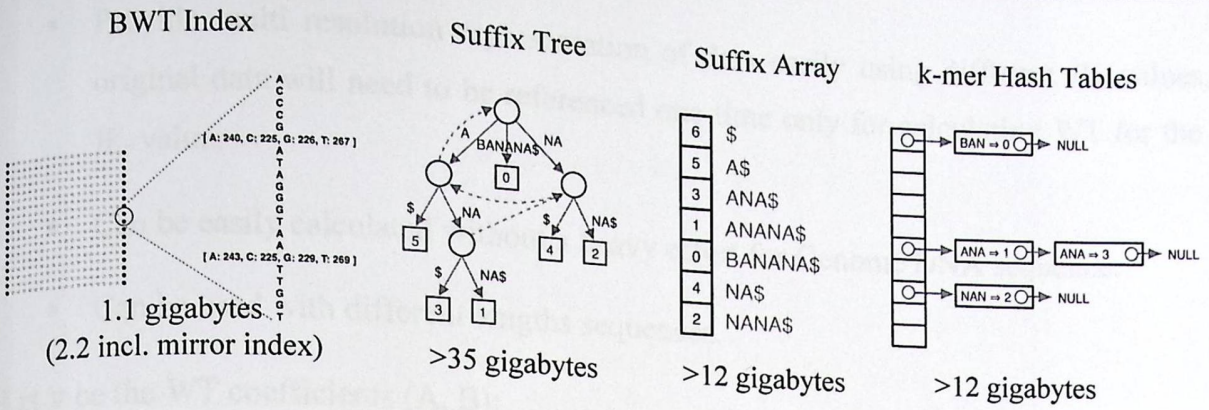


Figure 2.14: Comparison of index size for BWT, Suffix tree, Suffix array, and k-mer Hash table [4].

2.2.2 Haar Wavelet Transformation [30]

Let S a set of characters, of length $|S|$, from the domain $D = \{c_1, c_2, \ldots, c\alpha\}$ which is a set of α characters ($\alpha = 4$ for DNA sequences). Let Oci be the number of occurrences of character ci at any sequence built from the domain D, then the character frequency vector (FV) for S is $FV(S) = \{O_{c1}, O_{c2}, \ldots, O_{c\alpha}\}$. Number of elements at FV equals the number of elements of D(α):

$$\sum_{i=1}^{\alpha} Oci = |S| \quad \ldots\ldots\ldots\ldots (2.1)$$

If we want to compare any two different sequences, like S and W, from the same domain and the same length without considering the character order, FV is a good and easy tool for this need. But if we need to do comparison and save order of character, FV will only give a good indication, not an accurate answer.

To save order for characters, Haar wavelet transformation [30] (WT) can be used. WT can be constructed at different levels {1, 2… k} and consists of two coefficients {A, B}, the frequency coefficient "A" and difference coefficient "B", to provide a representation for the input data.

The coefficient A equal FV for the sequence S, where coefficient B is the frequency difference of FV for two parts, each part is half of sequence S. let S be divided into S_1 and S_2 where $S=S_1 U S_2$. Then A is the FV for S and B is the difference between FV(S_1) and FV(S_2). B coefficient saves character order.

The main properties of WT are:

- Saving characters order for the input text but not absolute. There are some cases where character order is not saved as will be shown bellow.

- Provide multi resolution representation of data easily using different W_x values, the original data will need to be referenced one time only for calculating WT for the first W_x value.
- Can be easily calculated without a heavy effort for Genome DNA sequence.
- Can be used with different lengths sequences.

Let v be the WT coefficients (A, B):

$v_{k,i} = (A_{k,i}, B_{k,i})$.....(2.2),

where

$$B_{k,i} = \begin{cases} 0 & k = 0 \\ A_{k-1,2i} - A_{k-1,2i+1} & 0 < k \le \log_2 n, \end{cases} \quad \text{.....(2.3)}$$

$$A_{k,i} = \begin{cases} f(c_i) & k = 0 \\ A_{k-1,2i} + A_{k-1,2i+1} & 0 < k \le \log_2 n, \end{cases} \quad \text{.....(2.4)}$$

Example of the construction of wavelet coefficients:

Let a sequence $S = \{AATGATAC\}$, divide S into s_1 & s_2 and calculate the two coefficients:

$S = AATG + ATAC = s1 + s2$

FV(s1) = [2, 0, 1, 1], and FV(s2) = [2, 1, 0, 1], then

A(S)= FV(s1)+FV(s2) = [4, 1, 1, 2] and B(S)= FV(s1)-FV(s2) = [0, -1, 1, 0]

And WT={(4 1 1 2),(0 -1 1 0)}

Another example to show the not absolute saving of order using WT:

Let $S=\{AATG\ ATAG\}$, then FV(s1)=(2011), FV(s2)=(2011), and WT={(4 0 2 2), (0 0 0 0)}

For $S=(GTAA\ TAGA)$, then FV(s1)=(2011), FV(s2)=(2011), and WT={(4 0 2 2), (0 0 0 0)}

For two different S values we get the same WT output value.

2.3 Hardware storage efficiency

Using the correct hardware structure is important to enhance the overall system performance, especially when the data size is huge. We can divide hardware issues needs into two categories, the internal hardware structure and the external hardware structure.

Internal structure is related to the internal hardware nodes specifications like server memory, RAID controller, number of CPUs, bus speed, storage size and speed, and I/O throughput. It is important to have an adaptive internal hardware structure to do tuning and parameterization according the system needs.

External structure is how these nodes (like servers) will be connected together. For example the uses of storage architecture like SAN and NAS, running more than one server together (clustered servers) for the database system to manage the data and the indexing. Storage internal specification need to be considered carefully, like internal switches type and speed, links type and speed, cache size, storage size and speed of the backend storage devices (for example fiber of SAS hard drives).

Another factor to worry about is the database management utilities which are related to the hardware specification like: logical page size, buffers, Joins, LOG archiving, and the effective use of database distribution. Our index structure gives user a flexibility to change parameters according to the available resources, see Chapter 6 for more details.

2.4 Full sequence alignment

Sequence alignment [48] used for comparison purposes. It is used to mark pattern of difference, finding common substrings, check if the sequences evolved from the same source sequence, and find the similar sequences from the database to the query sequence we have.

Given two sequences S_1=*(c[1]c[2]...c[L1])* and S_2= *(x[1]x[2]...x[L2])*, where c[i] and x[j] for i=*1..L1* and j=*1..L2* is a character. The alignment of the two sequences refers to assign gaps to positions at index i in S_1 and at j in S_2 so as to map up each character in one sequence with either a gap or a character in the other sequence. Thus, there can be many possible alignments between the two sequences.

For example, given the following two sequences:

S_1 : *GCGCATGGATTGAGCGA* and

S_2 : *TGCGCCATTGATGACCA*

One possible alignment is:

-GCGC-ATGGATTGAGCGA

TGCGCCATTGAT-GACC-A

An example of sequence alignment using GAP:

S1: G G T C C A T C G G T T T T C A G G G G
S2: G C T - - A T - - - T T T T C A G G G G

Gaps will not be considered to our work because it corresponds to add or delete of a character (A, C, G, T) at the sequence. We don't need to make any change, like add or delete operation, to make the similarity between sequences we need full alignment without any changes.

The main purpose of sequence alignment is to measuring the similarity between two sequences. DNA sequences length is very large and database size increases exponentially, so sequence alignment become time and space consuming.

2.4.1 Main DNA sequence alignment methods

There are mainly three types of sequence alignment: Global alignment, Local alignment, and multiple sequence alignment. Some types of sequence alignment based on scoring schema, the score is based on the goodness of alignment according to a specified criteria. Through this section we will discuses the following alignment methods: Global and Local alignment, Dot-matrix alignment, alignment using Dynamic programming.

Global and local alignments

Global alignment: given two sequences from the same domain, like DNA sequences from the alphabet {A, C, G, T}, find the similarity between the two sequences through the entire length of both sequences. For example Needelman-Wunsch [57] algorithm is a global alignment method. The following is a simple example of global alignment method:

```
ACGRGRGRACAACAGTTGAGGTCA
  |   |   |           | | | | | |
GCAAGTARTTGCACTAACAGGTCA
```

Global alignment is suitable to use when two sequence have the same length, and have a degree of similarity.

Local alignment: Given two sequences, find the substrings that match well. This method not force matching through the whole sequence length, substring matching is enough.

The following is a simple example of local alignment method:

```
ACGRGRGRACAACAGTTGAGGTCA
                | | | | | |
GCAAGGRGCAACAATAACAGGTCA
```

For example Smith-Waterman algorithm [59] is a local alignment method used for optimal local alignment with the highest accuracy.

Local alignment is used with sequences of different lengths, and has a small degree of similarity.

Dot-matrix methods*

This method builds a 2D matrix using two sequences. One sequence is at the vertical axis and another sequence is at the horizontal axis.

At each location at the matrix, if there is a match between vertical and horizontal axis, this means there is an identical value. At each identical value a mark is placed. To find similarity between the two sequences, diagonal lines are marked to indicate the similarity.

Dynamic programming

For a two sequences S_1, and S_2 of lengths L_1, and L_2 respectively, dynamic programming will build a scoring matrix SM(i,j) by aligning S_1 and S_2. Where $i \in \{1.. L_1\}$ and $j \in \{1..L_2\}$. We will have three cases when doing aligning of S_1 and S_2 as follows:

Figure 2.20 shows an example of Dot-matrix method:

	B	A	S	K	E	T	B	A	L	L
B	*						*			
A		*						*		
S			*							
E					*					
B	*						*			
A		*						*		
L									*	*
L									*	*

Figure 2.15: An example of Dot-matrix method*.

1. Matching a character from S_1 with a character from S_2:

SM(i,j) = SM(i-1,j-1)+ Match(S_1 [i], S_2 [j])……..(2.5)

Where,

Match(S_1 [i], S_2 [j]) = x if S_1 [i]= S_2 [j], else Match(S_1 [i], S_1 [j])=-y.

2. Matching a gap from S_1 to S_2:

SM(i,j) = SM(i-1,j)- penality……..(2.6)

3. Matching a gap from S_2 to S_1:

SM(i,j) = SM(i,j-1)- penality. ……..(2.7)

* Rasmus Pagh,, Database Tuning, http://www.itu.dk/people/pagh/DBT09/07-text-indexing.pdf, Spring 2009.

Now we have the following matrices SM(i-1,j-1), SM(i-1,j), and SM(I,j-1) we can find the optimal SM(i,j) as follow:

SM(i,j) = max of: 1. SM(i-1,j-1)+ Match(S1[i],S2[j]) .....(2.8)

2. SM(i-1,j)- g .....(2.9)

3. SM(i,j-1)- g .....(2.10)

Where g is the gap penalty.

So if we want to find SM(L_1, L_2) we need to calculate SM(i,j) over all $1 \leq i \leq L_1$ and $1 \leq j \leq L_2$. This means we will need a matrix of size $L_1 \cdot L_2$ to find the optimal alignment.

2.5 Summary

The large size databases require a large number of I/O references, which need to be minimized using indexing. The increase in memory size nowadays and dropping in cost allow larger index to be loaded into the memory. Sequential scan through the whole sequences, like BLAST and dynamic programming, needs a large storage space, very high memory consumption, and slow search time.

Building index structure, using vector based indexing, and not using distance based indexing, insure that exact matching will be used for searching. There are two ways for building the index for Genome DNA sequence, either Full-Text indexing or Word-based indexing; our index structure will follow Word-based method.

For the search query, we will use equality selection queries using exact measurement, range selection queries can be used also if similarity search is needed for any purpose. Full text indexing is suitable for similarity search mainly; if it is applied for exact matching it will require extra calculations like file signature indexing. System requirements always changed, so static index structure may face some obstacles; we need to think for future upgrade. Loading the whole structure for processing is hard and very costly process, so we need partial loading to be applicable according to a predefined criterion.

Tree based indexing suffer from the heavy work during construction step and massive I/O references. Also Tree structures, like suffix tree and SST, has a limitation on W_x values and search time is proportional to input data size. The tree structure that is suitable for Genome DNA sequence indexing is BTree.

Generally speaking, if exact matching is needed, the whole input data must pass though indexing. And if similarity matching is needed, partial part of input data will be passed to

indexing. Genome DNA sequence transformation is needed before indexing. We will use wavelet Transformation because of its suitable properties over BWT transformation for Genome DNA sequence indexing.

Different compression methods can be used either before or after transformation over the *Wsum* values result from transformation.

Evaluating different types of indexing using RDMS

The main problem of searching the DNA sequences is the large size of sequences and the high and variable sequence lengths. We need to have a suitable index for searching DNA sequences efficiently, with suitable index size and searching time. The number and type of relation fields, where index is build on, has a big effect on the index size and search time.

This chapter discusses the efficiency of using different relation database management system index types and the factors that are need to be considered when constructing index structure for large scale data. Our experiment uses the n-gram wavelet transformation upon one field and multi-fields index structure under the RDMS environment. Results show the need to consider index size and search time carefully.

This chapter includes the following parts: Data samples and methodology followed to transform sequences for the experiments, algorithm followed to do the experiments, introduction about indexing types used by RDMS, experiment results and discussion, last part is space and time complexity and conclusions.

3.1 Data and methodology

The data used in our experiments is shown in Table 3.1, we have selected different types (Species kingdom) including Archaea, Eukaryotic, and Bacteria. The sample file contains DNA sequences only and is of different sizes as show in Table 3.1.

Table 3.1: Data samples used by experiments

Species kingdom	Accretion No.	Seq. length	Size
Archaea	NC_010315.fasta	1,051	2KB
	NC_008318.fasta	15,717	16KB
	EU881703.1.fasta	28,643	29KB
	NC_006389.fasta	33,927	34KB
	AY596291.1.fasta	33,446	34KB
	NC_013966.fasta	63,034	630KB
	NC_011766.fasta	1,365,223	1353KB
	NS_000190.fasta	2,082,083	2000KB
Eukaryotic	AP009202.1.fasta	16,240	17KB
	NC_009684.fasta	16,604	17KB
	NC_010093.fasta	153,819	153KB
	NC_013009.fasta	879,977	872KB
Bacteria	NC_011841.fasta	30,652	31KB
	NC_009471.fasta	37,155	37KB
	NC_013210.fasta	191,799	190KB
	NC_009926.fasta	374,161	371KB
	NC_013009.fasta	879,977	872KB

3.1.1 Design

Our approach start by converting DNA sequence into eight columns vectors corresponding to the two wavelet transformation factors; A and B. Calculation like data center, four k-means, four vectors of size v with fixed size of eight-columns instead of a sequence of size n which is variable and long is less in storage size. This transformation has been used for computing second coefficient wavelet transformation with six different windows W_x of sizes 8, 16, 32, 64, 128, and 256 chars Figure 3.1 shows the conversion steps.

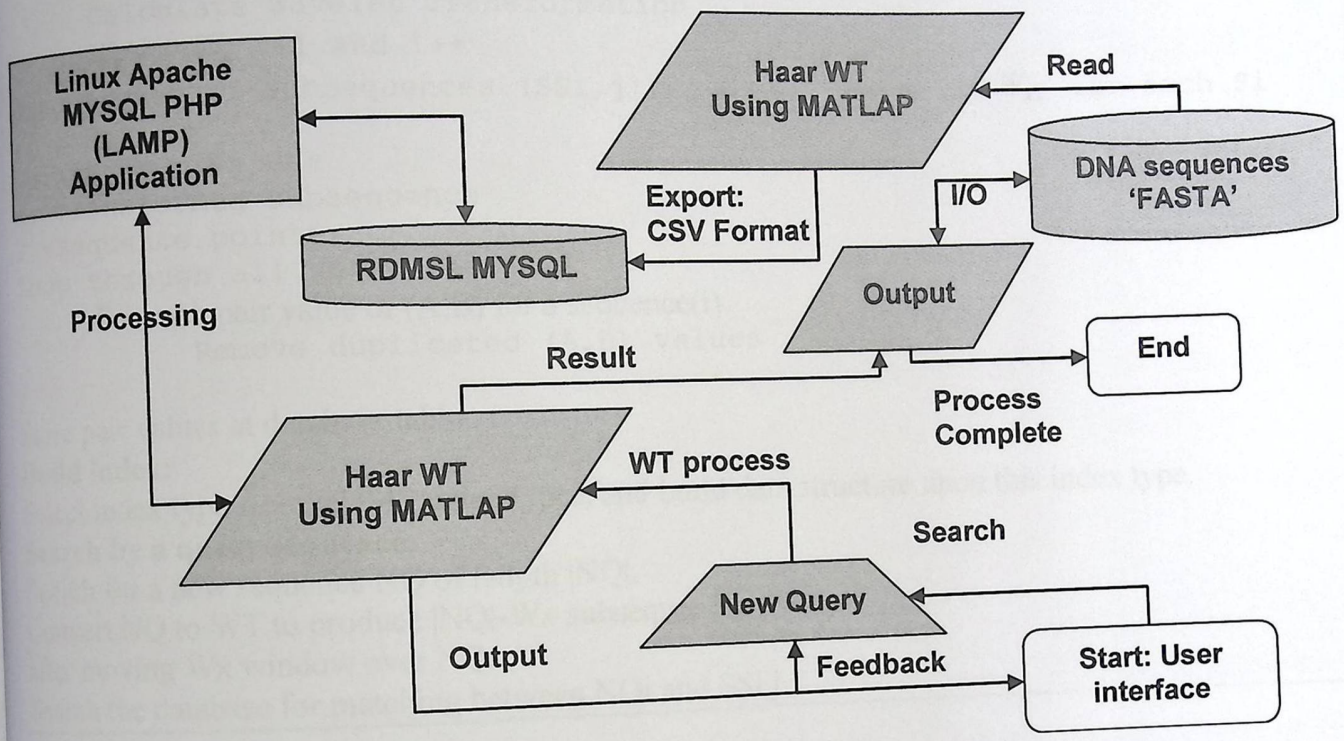


Figure 3.1: General schema chart shows transformation, building index structure, preprocessing new query, and comparison

After transformation we build an index which will be used for searching. Transforming data sequences to numerical representation (NR) will be accomplished.

The aim of using different window sizes W_x is to have different resolution levels of representation of a sequence; we aim, through using different window sizes, to find the values of the window sizes where index structure remain stable, in other word we need the window size where space and search time is optimal. We assume the windows sizes W_x to be $2x$. By this assumption after finding the first order wavelet transformation by scanning the database by window W_{x1}, we can find the wavelet transformation for window values W_{xI} for i>1 depending on previous window value (W_{x1}) and no need to scan the database again.

3.1.2 RDMS indexing evaluation algorithm:

Input:
Database of n sequences, n is a large value.
Each sequence will be donated by S_i, i ∈ [1, .., n] with length L_i.
Preprocessing:
We have different window sizes (W_x), x =1, …,6
Transform each Si into number format using Wavelet Transformation (WT), Haar wavelet transformation for Wi+1 can be calculated from Wi as:
(A1,B1),(A2,B2) -> (A1+A2,A1-A2)

```
Initialize i=0
For each Wx value from (Wx min,…, Wx max){
    Slide window Wx over Si
    Calculate Wavelet Transformation
    Wx' = Wx' +1 and i++
Output: set of subsequences (SSi,j), j ϵ[1,…, m],m=Li/ Wx, for each Si
}
Output: two values
1. Transformed subsequence
2. Sequence pointer
Loop through all sequences (i){
    For each pair value of (A,B) for a sequence(i)
        Remove duplicated (A,B) values
}
```

Store pair values at database table.
Build index:
Select index type from RDMS index types, and build data structure upon this index type.
Search by a query sequence:
Search for a new sequence NQ of length |NQ|.
Convert NQ to WT to produce |NQ|-Wx subsequences (NQi)
after moving Wx window over NQ.
Search the database for matching between NQi and SSi,j.

3.2 Constructing Wavelet coefficients

Each NR row corresponds to a DNA sequence, consists of 8 columns vector. The columns is the wavelet second order coefficients (A,B), A is a 4 columns represent frequency of chars (A,C,G,T) second part B is the difference. Example bellow describes how wavelet [30] works:

$v_{k,i} = (A_{k,i}, B_{k,i})$..... ....(3 .1), *where*

$$A_{k,i} = \begin{cases} f(c_i) & k = 0 \\ A_{k-1,2i} + A_{k-1,2i+1} & 0 < k \le \log_2 n, \end{cases} \quad , B_{k,i} = \begin{cases} 0 & k = 0 \\ A_{k-1,2i} - A_{k-1,2i+1} & 0 < k \le \log_2 n, \end{cases}$$

For a sequence u: [$ACTC\ \ TAGC$], consider frequency is done by the order (A,C,G,T) =(2, 3, 1, 2), divide u into two equal parts and recalculate frequency again then do subtraction between the two parts, you get [1201, 1111] →(2 3 1 2, 0 1 –1 0).

The sequences will be represented using six window sizes, W_i for i={1, 2, 3, 4, 5, 6}, Each window size W_i representation will correspond to a final matrix for each sequence, this means we will have six matrices corresponds to W_i value.

Then we upload the data on a rational database system (RDMS). We used RDMS to get advantage of RDMS indexing systems like BTree, and Hash. Before uploading data to database tables, all repeated rows had been eliminated to make index size less since there is no need for the repeated data rows. Table 3.3 shows results of transformation and percentage of repeated data.

Seven types of indexing have been used for evaluation. The types are index on primary key, normal index, primary index, full-text, unique, Hash, and BTree. Our experiments done using two PC's, one with 1GB memory 2GHz, second one is 2 x1GHz CPUs 4GB memory. RDMS used is MYSQL v5.0.1 [26, 21] to store index in, webserver is Apache and language script used is PHP v5 for testing index reach.

3.3 RDMS index types

Full-text index allow search for natural language text, some features are: Excludes partial words and words less than x characters in length (3 or less), words that appear in more than half the rows, Hyphenated words are treated as two words, Rows are returned in order of relevance, descending, words in the stopword list (common words) are also excluded from the search results. Full-text had been used to achieve high performance XML indexing [9].

"Normal" Indexes are the basic index type over most RDMS and requires data field to be ordered, Normal Index have no constraints such as uniqueness. Unique Indexes are the same as "Normal" indexes with one difference: all values of the indexed column(s) must only occur once. Primary index are unique indexes for primary keys.

BTree index, for n keys values, constructed by build a tree with height (h) and a degree (t). Where the degree (t) is greater than or equal to 2. The worst case of BTree is O($logn$) comparisons. Number of branches for BTree index is larger than the number of branches of other balanced tree structures. Number of branches for a tree controls the logarithm base of complexity ($Logn$ of base x where x equal the number of branches). So the base of logarithm tends to be large than required by other tree structures. And what this mean, it means that if we have n key values and we want to build a tree of base x, x branches, as we increase x number of nodes visited during search tends to be smaller. BTree tend to have smaller heights than other trees with the same number of key values. Path to leaf node not exceeding $Log_{n/2}K$ while a binary tree is Log_2K, where Search k-key values are $K_1, K_2, \ldots, K_{n-1}$.

BTree make all nodes full at least to a minimum percentage to save space and reducing number of disk references. Space complexity of BTree is O(L/B), where L: length of the sequence and B: block size [15].

In Hash index type, bucket reached by key through a set of functions (h functions). Records with different key values map to same bucket; thus entire bucket has to be searched sequentially to locate record.

Bucket Overflows causes by insufficient buckets and distribution of records (Overflow chaining) Collision handling with O(1) complexity, for worst cases performance may deteriorate to O(n). An ideal hash function is uniform/random and worst map to one bucket. Space complexity for formal Hash function is O($n \cdot \log n$) [15, 21], where n: number of keys. Hashing functions divided into two types, Uniform distribution: all buckets have the same number of search-key values. Random distribution: on average, at any given time, each bucket will have the same number of search-key values, regardless of the current set of values.

Primary index and Unique index both can consists of one or more fields, and both can be clustered/non-clustered indexes. The difference is that Primary cannot be Null while Unique can be, there can be only one Primary index on a table but you can have more than one Unique index.

Cluster indexing used when we have more than one table need to be referenced using join query. The efficient index is to store tables' data in the same block.

A clustered index sorts data physically while non clustered index does not. We can add one clustered index per table only, but it is possible to have multiple non clustered indexes in a single table. Non-clustered index used when table does not contains too many unique values in contrast to clustered index which is used when table contains large number of unique values.

3.4 Experimental results

We used two approaches for index evaluation. In the first approach, we created the index on one field representing the coefficients of WT and investigate the effect of changing the type of the index on the response time, which is measured in millisecond. Table 3.2-a shows the response time. For the second approach, search field is spitted into two parts mainly which are the wavelet transformation coefficients (A, B). Each part consists of four columns. Table 3.2-b shows the results of this approach.

For all index types we do search for the worst case if applicable or randomly. "Default on pk" ordered by order of entry, worst case is the last entry. The same is true for normal index, primary index, and unique index.

Table 3.2-a: Evaluation of sample data (under six resolutions W) using different index types.

Index type	W1	W2	W3	W4	W5	W6
DEFAULT ON pk	0.002494	0.029923	0.233727	0.677167	0.9619	1.1533
Normal Index	0.01046	0.1679	1.3807	4.05	5.8911	6.3502
PRIMARY	0.093462	0.190009	1.4703	4.233	6.0642	6.4283
Fulltext	0.002924	0.027733	0.224	0.248133	1.082667	1.501533
UNIQUE	0.009996	0.1726	1.406	4.0779	5.931	6.357
Hash	0.0107	0.167433	1.3728	4.1471	6.016033	6.344233
BTree	0.010133	0.1664	1.709933	4.1319	6.0156	6.337333

Table 3.2-b: Applying indexes for eight columns search fields.

Index type	W1	W2	W3	W4	W5	W6
DEFAULT ON pk	0.018718	0.010034	0.067594	0.209389	0.28135	0.27995
Normal Index	0.001046	0.001547	0.001531	0.001733	0.001544	0.001677
Hash	0.001588	0.001561	0.001646	0.001609	0.001599	0.001651
BTree	0.001605	0.001663	0.001614	0.001688	0.001605	0.001522

Through all experiments, the searching process is applied using the same value, while changing index type, so we can results correctly. For Hashing index type, most database engine uses random hash function, we do the experiment by randomly picking values then the average access time is calculated. BTree index, which is the most popular index over database systems, depends mainly on sorting the data.

Table 3.3 shows percentage number of returned references to the whole database size while changing window size

Table 3.3: Error amount at each resolution used corresponding to amount of reduction.

W	with duplication	No duplication	References%
W1	7069	1250	0.81
W2	73562	23283	0.65
w3	325701	192058	0.41
w4	662746	553933	0.15
w5	813987	796371	0.02
w6	836376	835106	0.002

$$Ratio = \frac{sequences\ retrieved}{total\ size\ of\ dataset} \quad \text{.....(3.2)}$$

Figure 3.2 shows that, depending on input data properties, while increasing sliding window size the size of index is the same even if we don't remove duplicated values.

3.5 Analysis

We have applied indexes in two different ways, one field index and multiple fields' index. When using one field index, the best performance achieved was using default index and Full-text. When we used BTree or primary index we get the worse performance over all for one field indexing. Almost all other types of indexes give performance close to BTree index.

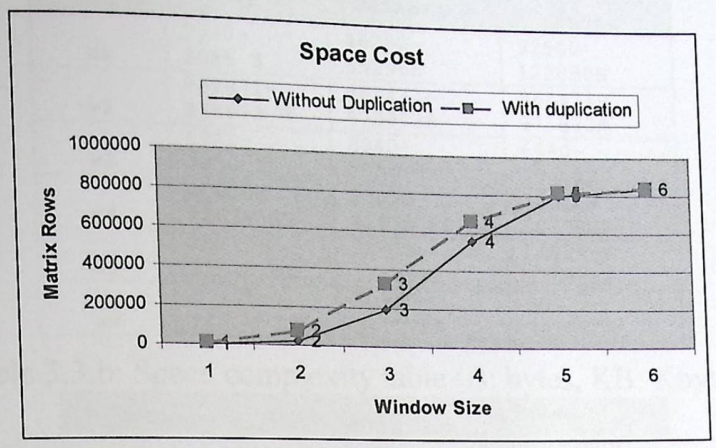


Figure 3.2: Space cost.

On the other hand when we used multiple fields index, we got much better results as shown by Table 3.2-b. Hash, BTree, and normal index on the eight fields give better results when compared with a single field index by Table 3.2-a.

Multiple field index cause overhead for: calculation and index address updates in case that amount of updates are high and overhead for write operations and disk referring. But if compared amount of overhead with multiple indexes (merge index) case, this overhead is much less in size. For DNA database, updates operation is much less than insert operation and can be neglected. To reduce size of WTR, singular value decomposition (SVD) [58] as a preprocessing step before building index structure for the genome database cab be used.

Window size (resolution) affect mainly needed I/O references. When we increase window size the I/O reference operation decreases as shown in Table 3.3. Changing the resolution of the wavelet transformation resolution from low value to high value (from 8 char to 256 char) leads to increase in size of the number of the wavelet coefficients and time to scanning the database. From Table 3.3 when using w1 we get 81% of overall database reference while for w4 this percentage goes down to 15%. Changing window size affects I/O reference percentage directly so as this percentage can be used as a threshold according to application needs.

3.6 Index space complexity

Table 3.3 shows the space complexity of using one column index with the following index types: full-text, primary index, 8 column index of type's unique, primary, and normal index. The range values (for example *6940-7850KB*) represent the space occupied by the data and the index respectively.

Table 3.3.a: Space complexity table (B: bytes, KB: Kbytes).

W value	UNIQUE	Primary-8	Index
w1	2048-2085 B	46250-55296B	92500-122880B
w2	861471-975872 B	861471-975872B	861471-975872B
w3	6940-7850 KB	6940-7850	6940-7850KB
w4	20015-22638 KB	20015-22638 KB	20015-22638KB
w5	28775-32545 KB	28775-32545KB	28775-32545KB
w6	30175-34128 KB	30175-34128KB	30175-34128KB

Table 3.3.b: Space complexity table (B: bytes, KB: Kbytes).

W value	Full-text	Primary
w1	25000-23552B	25000-15360B
w2	465660-435200 B	475812-241664B
w3	3751-4715 KB	4334-1928KB
w4	10974-14913 KB	13137-5561KB
w5	16253-22866 KB	19364-7995KB
w6	17931-26848 KB	21193-8384KB

Data size: is highest when using 8-columns index structure, low value when using one field index.

Index size: when using 8-columns almost the data and the index size are the same. When using one field index, data and index sizes are relatively the same too.

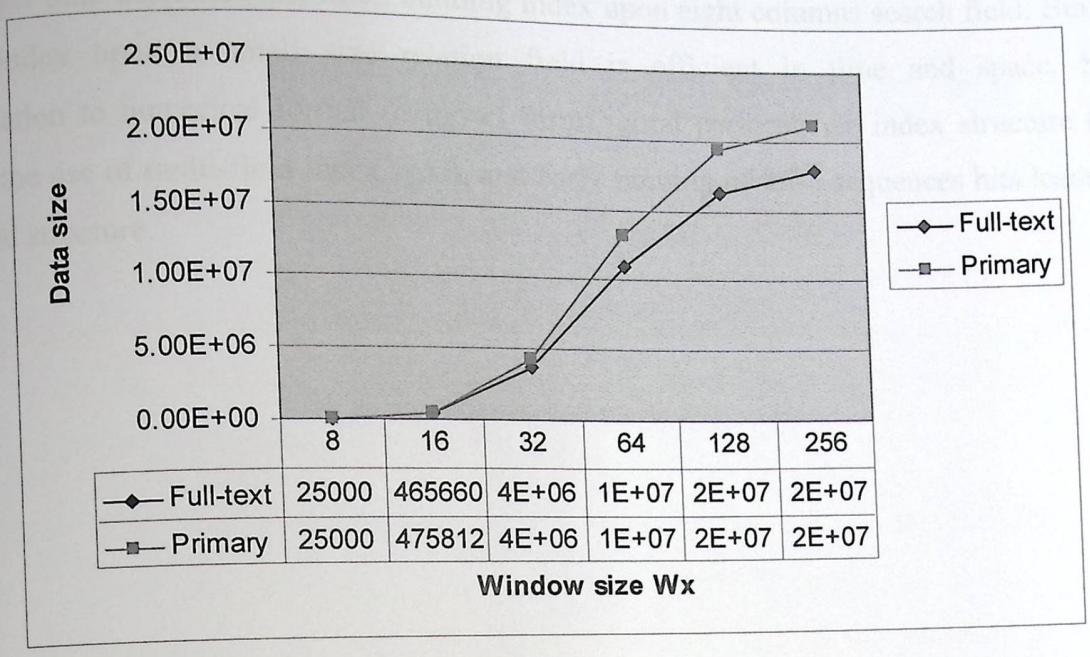
When comparing time with size for one field index, we found that best time performance achieved by full-text but full-text had high space requirement. For 8-column index structure best time achieved by normal, hash, and BTree index.

Access time for 8-column is better than that of one field index but index size equal or more than data size which is a large value as shown by Figure 3.3 b. Lowest index size is primary and Full-text as shown in Figure 3.3 a.

3.7 Summary

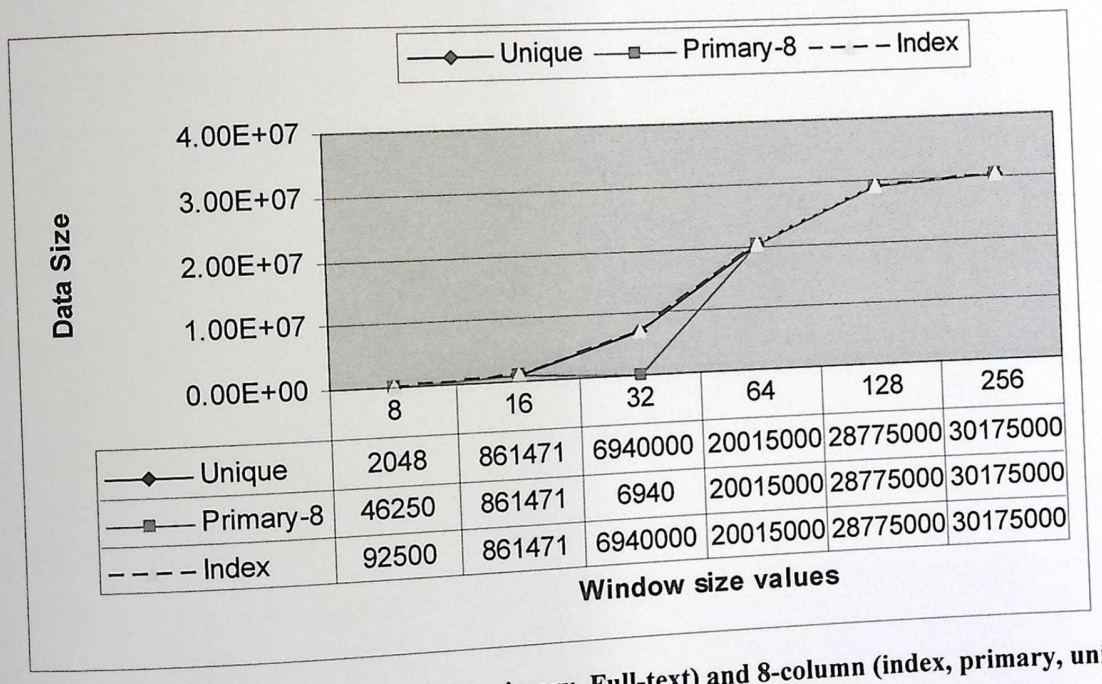
Our evaluation shows that using multi-fields index improve performance over all types of indexing in spite of the type of index we used. The first experiment shows that using specialized index type like full-text or primary index in integer fields give the best performance over using BTree or Hash indexing.

Different window sizes provide multi-resolution index structure. This property gives user a threshold value to determine his needs, and support queries of different sizes. Through our work, we see that no need, when doing query search, to scan the whole database. Instead of scanning the whole database a subset of sequences, which we call candidate sequences, will be referenced from the database after the filtration step. By this way we have minimized the number of disk pages that will be visited at the final stage.



	8	16	32	64	128	256
Full-text	25000	465660	4E+06	1E+07	2E+07	2E+07
Primary	25000	475812	4E+06	1E+07	2E+07	2E+07

Figure 3.3 a: Space cost for all one field (primary, Full-text) and 8-column (index, primary, unique)



	8	16	32	64	128	256
Unique	2048	861471	6940000	20015000	28775000	30175000
Primary-8	46250	861471	6940	20015000	28775000	30175000
Index	92500	861471	6940000	20015000	28775000	30175000

Figure 3.3 b: Space cost for all one field (primary, Full-text) and 8-column (index, primary, unique)

Space and time complexity show that using special type of index (like Full-text) or using the primary index, of one field, leads to decrease index size, like the full text index when using w6 compared with unique index for the same window size as shown by Table 3.3-a and 3.3-b. And a higher access time compared to eight fields index type, which lead to larger index size but better access time. This is true, as the Full-text get advantage of its properties as a special index for the search field and the primary index is on integer field, which is less in size than the 8 columns (64.303 compared with 29.577 about one half). This means that a good representation of search field must occupy less space. Small size index, which can be fit in memory, allow the use of in-memory searching mechanisms which gives fast searching time.

From the discussed results, we can see that we need to try to find a less size index structure. Index size is larger than database size, when building index upon eight columns search field. Building the primary index upon a small size relation field is efficient in time and space. Sequence transformation to numerical format (compact form), good performance index structure (size and time and the use of multi-field index type), and early pruning of false sequences hits leads to build the desired structure.

Chapter 4

Modified Wavelet Transformation and BTree (M-WTBT) specification

This chapter will introduce our work to build an index structure for a DNA Genome characters. This chapter includes two main sections: the DNA sequence transformation process from character domain to integer domain using WT with some modification, and the index structure, used to hold the transformed data, based on modification of BTree structure. We called it Modified WT and BTRee M-WTBT index structure.

Section 4.1 discuses two issues: the modification on calculating the wavelet transformation (WT) coefficients to have the final sliding windows representation, *Wsum* values, and the analysis for three suitable cases to store two fields, the *Wsum* values and the sequence numbers using a relation at Section 4.1.2.

Section 4.2 compares the use of two different structures, the BTree and B+Tree strcutures, to hold the *Wsum* values. And the advantages of BTree over B+Tree for Genome DNA sequence needs are shown.

In the last section, Section 4.3, we compare index structure based on Radix tree with the index structure based on modified WT and BTree. The aim of the comparison is to calculate the efficiency of index size of suggested structures. In the last part of this section we discuss the sequence candidate alignment of sequences, which is the last processing step.

4.1 Sequence transformation

For a set of Genome DNA sequences of n sequences, we need to represent the Genome DNA sequence at an integer format to get advantage of integer representation. We have used WT because it saves string order, can be calculated efficiently for larger window sizes and other properties mentioned in Chapter 2. The following two sections will show the process of transformation and the structure we used to hold the transformed vales.

4.1.1 Sequence transformation using modified WT to reduce space complexity

The general idea behind string transformation is to scan the database of sequences pattern by pattern, using window of size W_x, because it is hard to scan the whole sequence at one time. Then we apply WT on these substring, we use n-gram Haar wavelet transformation. Where n = 1, which means each overlap amount equal to one character.

The output of WT has two parts, called coefficients {assume the first coefficient is α and the second coefficient is β}, α coefficient is the frequency value for the genome DNA characters (A, C, G, T), and β coefficient is the frequency difference between two equal parts (after dividing the substring into two equal substring and calculating frequency for each part). Each coefficient consists of four fields {(A1, A2, A3, A4), (B1, B2, B3, B4)} [50, 30]. Refer to Section 2.2.2 for more details.

We want to convert the four fields of each coefficient to one integer value, using a hashing function, to represent the original data to save space, reduce collision, and improve search time.

The hashing function that we use is:

α = A1.v + A2.w + A3.t + A4.r.....(4.1)

β = B1.v + B2.w + B3.t + B4.r.....(4.2)

Aims for this transformation are:

1. Combining WT coefficients: combine the four fields for each coefficient (A1, A2, A3, A4 and B1, B2, B3, B4) to one integer value (α and β).

2. Decreasing amount off collision: using the conversion parameters (v, w, t, r), which are integers of different values, to decrease collision. We suggest the following strategies for picking values for the conversion parameters:

    - If the conversion parameters have the same value (for example v=2, w=2, t=2, r=2), α part will be identical between all substrings and we want to decrease amount of collision. We used the values v=16, w=8, t=4, r=2 according to experiments.

      Let we consider the two substrings at different locations 'i' and 'j'

      α[i] = A1[i].v + A2[i].v + A3[i].v + A4[i].v

      = v.(A1[i]+A2[i]+A3[i]+A4[i])

      α[j] = A1[j].v + A2[j].v + A3[j].v+ A4[j].v

      = v.(A1[a]+A2[a]+A3[3]+A4[4])

      Referring to the wavelet transformation properties of Genome DNA sequence, we have that:

      A1[i]+A2[i]+A3[i]+A4[i] = W_x and B1[j]+B2[j]+B3[j]+B4[j] = W_x for any window size W_x, Equations 4.3 and 4.4 become:

      α[i] = v.(W_x) and α[j] = v.(W_x). .....(4.3)

The output of WT has two parts, called coefficients {assume the first coefficient is α and the second coefficient is β}, α coefficient is the frequency value for the genome DNA characters (A, C, G, T), and β coefficient is the frequency difference between two equal parts (after dividing the substring into two equal substring and calculating frequency for each part). Each coefficient consists of four fields {(A1, A2, A3, A4), (B1, B2, B3, B4)} [50, 30]. Refer to Section 2.2.2 for more details.

We want to convert the four fields of each coefficient to one integer value, using a hashing function, to represent the original data to save space, reduce collision, and improve search time.

The hashing function that we use is:

α = A1.v + A2.w + A3.t + A4.r.....(4.1)

β = B1.v + B2.w + B3.t + B4.r.....(4.2)

Aims for this transformation are:

1. Combining WT coefficients: combine the four fields for each coefficient (A1, A2, A3, A4 and B1, B2, B3, B4) to one integer value (α and β).
2. Decreasing amount off collision: using the conversion parameters (v, w, t, r), which are integers of different values, to decrease collision. We suggest the following strategies for picking values for the conversion parameters:
   - If the conversion parameters have the same value (for example v=2, w=2, t=2, r=2), α part will be identical between all substrings and we want to decrease amount of collision. We used the values v=16, w=8, t=4, r=2 according to experiments.

     Let we consider the two substrings at different locations 'i' and 'j'

     α[i] = A1[i].v + A2[i].v + A3[i].v + A4[i].v

     = v.(A1[i]+A2[i]+A3[i]+A4[i])

     α[j] = A1[j].v + A2[j].v + A3[j].v+ A4[j].v

     = v.(A1[a]+A2[a]+A3[3]+A4[4])

     Referring to the wavelet transformation properties of Genome DNA sequence, we have that:

     A1[i]+A2[i]+A3[i]+A4[i] = W_x and B1[j]+B2[j]+B3[j]+B4[j] = W_x for any window size W_x, Equations 4.3 and 4.4 become:

     α[i] = v.(W_x) and α[j] = v.(W_x). .....(4.3)

- Concatenating of four fields: this method requires more storage space but also illuminate collision (A1A2A3A4, B1B2B3B4).
- Use strategy number '1' with the addition of row number to decrease collision amount.

The last step is to combine the two values (α and β) to produce the *Wsum* value. We consider the summation of α and β coefficients according to the following equation:

$Wsum = \alpha.wa + \beta$ ..... (4.4)

The aim of multiplying α with '*wa*' instead of $\alpha+\beta$ is to solve the problem of false positive to reduce collision. '*wa*' is any number lead to exceed largest $\alpha+\beta$ value, for example if largest $\alpha+\beta$ value is 120,000 then α . *wa*>120000. Duplicated *Wsum* values is removed to save space, duplicated *Wsum* values for the same sequence means the same substring and we do not want to lose space and search time for duplicated *Wsum* values which will point to the same sequence. Any other options, listed in point 2 at the previous page, can be followed also.

Let's consider the two coefficients α, and β for two different *Wsum* values at positions '*i*'' and '*j*',

This case means we have false positive, if we take the summation of $\alpha+\beta$ only, we will get the same value for two different *Wsum* values. To solve this problem we consider using weight to one coefficient, or we can give two different weights to both coefficients. We follow strategy number '1'.

For example, for the following substrings: s1 = ATTCAGAT, and s2=TTCAGATG

WT (s1)={66,-10} and WT(s2)={54, 2},

If we take *Wsum*= $\alpha+\beta$ then, 66 -10 = 54+2 = 56.

We want to minimize amount of collision to the minimum, according to the available resources. We allow the user to set the values of *v,w,t,r* , and *wa*. Changing those values will affect the amount of storage required and the needed memory. For example if *v,w,t,r* values increases, the number of the distinct *Wsum* values will increase too.

The calculation of WT coefficients, we have reduced the amount of calculation to 25% by neglecting one character from the four DNA characters {A, C, G, T}. We have neglected this character from character frequency counting, character difference factor (A and B) calculation, where the calculations affected by not considering the character "A" are two places:

- At the 'If statement' for comparing each DNA characters (A, C, G, T) to calculate the frequency for each character.

- At calculating the WT coefficients { α, β } fields A1 and B1 used by the hashing function at Equation 4.1 and 4.2.

The same will be done when searching for new query, to find the *Wsum* values, for comparison with the whole database.

For example: let S1 = { GGTAAACGGCTAAT } and S2={ AAACTGGAAGGCTT } then, S1 coefficients: (0,2,4,3) (0,0,0,1) and if considering "A" character (5,2,4,3) (-1,0,0,1), and S2 coefficients: (0,2,4,3) (0,0,0,1) and if considering "A" character (5,2,4,3) (-1,0,0,1).

4.2 M-WTBT Index structure analysis and specifications

What we will do next is to store *Wsum* value at a suitable and efficient index structure, where *Wsum* value may exists at different sequences.

The main fields of interest for index structure building are the *Wsum* data and sequence numbers. To create an index on a field or a combination of fields, choosing the right field depends on set of factors:

- If a small set of columns are referenced by the query, put all that columns on an index.
- When building multiple columns index, use high cardinality fields (like *Wsum*) first then less cardinality fields. Using the highest cardinality firstly is good because query reference this field too much.
- Considering the field size and data values at this field, like if it is a key or not key with small size or large size and if it holds character data or integer data.
- Additional index may be required if query search is too slow or data size increased significantly. Statistics and system monitoring is required to support any change for indexing.

We have two fields, *Wsum* and sequence number (Seq#), index will be build using both of them. If other fields need to be added to the relation, index structure will not be affected as query search needs *Wsum* and Seq# fields only.

In general, we have three cases for constructing the index using *Wsum* and *Seq#* as shown in Table 4.1. The three cases are:

Case (a) allows the repetition of *Wsum* value and preventing repetition of sequence number. Case (b) allows repetition for sequence number and have distinct *Wsum* values. In case (c), the two fields work as a composite key field.

e 4.1 is a sample data from Eukaryotic (NC_009684, NC_010093, NC_013009) sequences*.

next three sections will discuss the three cases in details; we will analyze the total size of the x for each case. As we said before, the table consists of two fields the *Wsum* and sequence ds. The total size for each case comes from *Wsum* field size and sequence number (*Seq#*) field

Table 4.1: Index structure using *Wsum* and Seq# fields.

Substring	Seq#
549	1
511	1
539	1
551	2
514	2
558	2
559	3
486	3
561	3
511	5
511	6

Original Data

Wsum	Seq#
511, 539, 549	1
551,514,558	2
486,559,561	3
539,511	5
511	6

(a)

Wsum	Seq#
514	2
486	3
549	1
551	2
558	2
539	5,1
561	3
511	5,6,1
559	3

(b)

Wsum	Seq#
514	2
486	3
549	1
551	2
558	2
539	1
539	5
561	3
511	1
511	5
511	6
559	3

(c)

The following definitions are used through the analysis of the three cases.

Assume:

n: number of sequences.

R_1: number of rows for case (a) = n.

R_2: number of rows for case (b) = number of distinct *Wsum*.

R_3: number of rows for case (c), R_3=R_2 + repeated *Wsum* values.

The total size of the index for the three cases will be:

Total index size = *Wsum* filed size + sequence number field size.

4.2.1 Case (a): Unique sequence number

Field Seq# size:

This column contains one value which is the sequence number at the database. Suppose that this value range from '*a*' to '*b*' where '*b*' is a maximum sequence number. Suppose that '*b*' needs *x* bytes for storage. To calculate the exact number of bits to represent sequence number, the

* NCBIGenebank, *http://www.ncbi.nlm.nih.gov/Genbank/S.*

Table 4.1 is a sample data from Eukaryotic (NC_009684, NC_010093, NC_013009) sequences*.

The next three sections will discuss the three cases in details; we will analyze the total size of the index for each case. As we said before, the table consists of two fields the *Wsum* and sequence fields. The total size for each case comes from *Wsum* field size and sequence number (*Seq#*) field size.

Table 4.1: Index structure using *Wsum* and Seq# fields.

Substring	Seq#
549	1
511	1
539	1
551	2
514	2
558	2
559	3
486	3
561	3
511	5
511	6

Original Data

Wsum	Seq#
511, 539, 549	1
551,514,558	2
486,559,561	3
539,511	5
511	6

(a)

Wsum	Seq#
514	2
486	3
549	1
551	2
558	2
539	5,1
561	3
511	5,6,1
559	3

(b)

Wsum	Seq#
514	2
486	3
549	1
551	2
558	2
539	1
539	5
561	3
511	1
511	5
511	6
559	3

(c)

The following definitions are used through the analysis of the three cases.

Assume:

n: number of sequences.

R_1: number of rows for case (a) = n.

R_2: number of rows for case (b) = number of distinct *Wsum*.

R_3: number of rows for case (c), $R_3=R_2$ + repeated *Wsum* values.

The total size of the index for the three cases will be:

Total index size = *Wsum* filed size + sequence number field size.

4.2.1 Case (a): Unique sequence number

Field Seq# size:

This column contains one value which is the sequence number at the database. Suppose that this value range from '*a*' to '*b*' where '*b*' is a maximum sequence number. Suppose that '*b*' needs *x* bytes for storage. To calculate the exact number of bits to represent sequence number, the

* NCBIGenebank, http://www.ncbi.nlm.nih.gov/Genbank/S.

maximum value needs more number of bits than the minimum value, we used the following equation:

$$Number\ of\ bits = \begin{cases} 1 \ if \ value \ = 1 \\ \lceil Log_2 n \rceil \ for \ other \ values \end{cases}$$

Let's neglect the first case when the sequence value is 1, the total space size will be:

$$Space\ Size = \sum_{x=1}^{n} Log_2 n$$

We will consider that all values at the sequence field have one size, which is the maximum size:

$$Sequence\ number\ \ size = R_1 \cdot Log_2 n \quad \text{.....(4.5)}$$

Field Wsum size:

Suppose the number of *Wsum* values (count of *Wsum* values) in each Relation Row (*RR*) is called *cWsum(j)* where *j* is the *RR* number, each *RR* contains a set of *Wsum* values. *Wsum* value is within a range (*x* to *y*) and will need *Wsum_size* bytes space for each value. *Wsum_size* value is the number of bytes needed to store maximum value (*y*), we will use this value, which is more than needed as some small values will need less number of bytes than *Wsum_size*:

cWsum(j) = count (Wsum) at *RR j*, where j=1 to R_1,

$$space\ \ size = sizeof(Wsum) \cdot \sum_{j=1}^{j=n} cWsum(j) \quad \text{.....(4.6)}$$

$$Total\ \ space\ \ size = R_1 \cdot Log_2 n + \text{Wsum\_size} \cdot \sum_{j=1}^{j=n} cWsum(j) \quad \text{.....(4.7)}$$

4.2.2 Case (b): Unique *Wsum* values

The number of distinct *Wsum* vales for all database sequences equal R_2. There is a relation between R_2, W_x, and the window overlapping amount ∇. When we increase W_x value, R_2 increases too as *Wsum* values become more distinct as shown at Chapter 3. And decreasing overlapping amount will decrease number of output substrings and hence *Wsum* values.

Now comparing R_2 with the number of elements in column *Wsum* in case (a):

$$number\ of\ elements\ at\ column\ Wsum = \sum_{j=1}^{R1} cWsum(j)$$

$$R_2 \le \sum_{j=1}^{j=R1} cWsum(j) \quad \text{.....(4.8)}$$

This is true because both fields, the *Wsum* field at case (a) and the *Wsum* field at case (b) contains the same *Wsum* values with one difference only, no repeated items of *Wsum* are allowed at *Wsum* field in case (b).

Field Seq# size:

Suppose that the number of sequences values at each *RR* is cSeq(j) at row *j*, each *RR* contains a set of sequence numbers within the range (*1* to *n*) and up to Log(*x*) bits space for each value is needed.

cSeq(j) = count (sequence) at j, where j=1, …, R_2, and

$$\sum_{j=1}^{j=R2} cSeq(j) = \sum_{j=1}^{j=n} cWsum(j) \quad \text{.....(4.9)}$$

This is true, if we count the number of items at the field *seq#* at case (b), it will equal to the number of items in the field *Wsum* in case (a). Bellow is the proof:

In case (a) each *Wsum* value may belong to more than one sequence (like the value '511' at Table 4.1) and this is represented by duplicated *Wsum* values at column *Wsum* (suppose for each *Wsum(i)* value the amount of duplication = *di*). *'di'* equal to number of occurrence of this *Wsum* value at different sequences and not at one sequence.

In case (b) each *Wsum* value appears only once, so if two sequences share same *Wsum* value, like sequences 1, 5, 6 share substring 511, then *seq#* column will contain duplicated values (*cSeq*) and equal to d_i.

And to calculate *Seq#* field size, the total size of sequence values is: $\sum_{x=1}^{n} Log_2 n$ and the size of one sequence is *Logn* of base 2, for total n sequences.

$$Seq\#\ field\ size = sequence\ Size\ .\ number\ sequences$$

$$Seq\#\ field\ size = Log_2 n \cdot \sum_{j=1}^{j=R2} cSeq(j),$$

$$and\ refering\ to\ (4.9): \sum_{j=1}^{j=R2} cSeq(j) = \sum_{j=1}^{j=R1} cWsum(j)$$

$$space\ field\ size = Log_2 n \cdot \sum_{j=1}^{j=R1} cWsum(j).$$

And let the variable Σ equal to:

$$\Sigma = \sum_{j=1}^{j=R2} cSeq(j) = \sum_{j=1}^{j=R1} cWsum(j)$$

$$Then\ R_3 = \sum_{j=1}^{j=n} cWsum(j) \text{.....(4.10)}$$

Field *Wsum* size:

This column contains distinct *Wsum* values over all sequences.

$$Wsum\ field\ size = R_2 \cdot Wsum\_size$$

$$Total\ size = R_2 \cdot \text{Wsum\_size} + Log_2 n \cdot \sum_{j=1}^{j=R1} cWsum(j) \text{.....(4.11)}$$

4.2.3 Case (c): Composite key

Field Seq# size:

Referring to Equation (4.5),

$$Seq\#\ field\ size = R_3 \cdot Log_2 n \ , where\ R_3 = \sum, then$$

$$Seq\#\ field\ size = \sum \cdot Log_2 n \text{ .....(4.12)}$$

Field *Wsum* size:

This column contains *Wsum* values for all sequences Si, for i=1..n, repetition is allowed at this field. Suppose the amount of repetition is *RWsum*, then the total number of items;R_3 is:

$$R_3 = R_2 + RWsum.$$

$$Wsum\ Field\ size = R_3 .\ Wsum\_size$$

$$= SUM .\ Wsum\_size \ (referring\ to\ Equation\ 4.11)$$

$$Wsum\_size\ Total\ size = SUM \cdot Log_2 n + SUM \cdot \text{Wsum\_size} \text{.....(4.13)}$$

By using $\sum$ according to Equation 4.13 and let X= $Log_2 n$ for Equation 4.7 and Equation 4.11,we get:

$$Total\ storage\ size\ case(a) = R_1 \cdot X + Wsum\_size \cdot SUM = T_1$$

$$Total\ storage\ size\ case(b) = R_2 \cdot Wsum\_size + X \cdot SUM = T_2$$

$$Total\ storage\ size\ case(c) = SUM \cdot Wsum\_size + X \cdot SUM = T_3$$

Now we will compare the three cases regarding required space and the main operation; the insertion operation.

4.2.4 Comparing storage size:

We know that R_2<*SUM* from Equation 4.4 and we can say that *Wsum_size*> *X*, where *X* represent number of bits to represent sequence number and *Wsum _size* is the number of bits to store *Wsum* value. The total number of *Wsum* values exceeds the total number of sequences, as each sequence produce set of *Wsum* values.

From the study of Genome DNA databases, the total number of sequences exceeded 150 million* this means that the maximum number of *Wsum* values will exceed this value. Referring to the total storage size for the three cases (T_1, T_2, and T_3), the variables that we need to compare are R_1, R_2, and R_3 to find the best storage size between them.

Generally speaking, if the number of sequences (*Seq#*) is larger than the number of *Wsum* values, then $R_1 > R_2$, otherwise $R_1 < R_2$, where $R_3 > R_1$ and $R_3 > R_2$. This means that $\sum > R_1$ and $\sum > R_2$ as $\sum = R_3$ according to 4.11.

Referring to Chapter 3, we found that increasing W_x value leads to have a distinct *Wsum* values overall sequences and number of common *Wsum* values over all sequences decreases sharply.

We have two situations concerning the values *Wsum* as follow:

1. *Wsum* values are not repeated: this means that the field *Seq#* at case (b) will hold one value mostly. At this case:

$R_2 > R_1$ and $X < Wsum\_size$, suppose that $R_2 = a \cdot R_1$ and we have n sequences, suppose that each sequence will have k *Wsum* values then:

Assumed $X = Log_2 n$,

$$\begin{aligned} Wsum\_size &= Log_2(n \cdot k) \\ &= Log_2 n + Log_2 k \\ &= X + Log_2 k, \end{aligned}$$

and

$$\begin{aligned} T_1 &= R_1 \cdot X + (X + Log_2 k) \cdot SUM \\ &= R_1 \cdot X + X \cdot SUM + Log_2 k \cdot SUM \\ T_2 &= R_2 \cdot (X + Log_2 k) + X \cdot SUM \\ &= R_2 \cdot X + X \cdot SUM + Log_2 k \cdot R_2 \end{aligned}$$

and if substitute $R_2 = a \cdot R_1$ at Equation 4.7 and at Equation 4.11,

$$T1 = R_1 \cdot X + [X + Log_2 k] \cdot SUM = R_1 \cdot X + X \cdot SUM + Log_2 k \cdot SUM$$
$$T2 = R_1 \cdot a * [X + Log_2 k] + X \cdot SUM = a \cdot R_1 \cdot X + X \cdot SUM + Log_2 k \cdot R_1 \cdot a$$

And we know that: $\sum > R_2$ from Equation 4.9, this means that T_1 and T_2 almost have the same value.

2. Wsum values are repeated

*Denise Casey, Human Genome Management Information System, http://www.ornl.gov/sci/techresources/Human_Genome/publicat/primer/fig3.html

$R_2 > R_1$ and $X < Wsum\_size$, suppose that $R_2 = (a/v) \cdot R_1$ and we have n sequences, suppose that each sequence will have (k/z) $Wsum$ values then:

$X = Log_2 n$, and

$Wsum\_size = Log_2(n \cdot \frac{k}{z}) = Log_2 n + Log_2 \frac{k}{z} = X + Log_2 \frac{k}{z}$, then

$T_1 = R_1 \cdot X + (X + Log_2 k) \cdot SUM = R_1 \cdot X + X \cdot SUM + Log_2 \frac{k}{z} \cdot SUM$

$T_2 = R_2 \cdot (X + Log_2 k) + X \cdot SUM = R_2 \cdot X + X \cdot SUM + Log_2 \frac{k}{z} \cdot R2$

This means that T_1 and T_2 almost have the same value.

And for case (c) using Equation 4.13:

$$Total\ space\ size\ case(c) = T_3 = SUM \cdot Wsum\_size + X \cdot SUM$$

For comparing T_2 and T_3, we need to compare R_2 and $\sum$, all other values ($Wsum\_size$, X) are identical between T_2 and T_3. We know that $\sum > R_2$, from Equation (4.8) so case (c) is the worst case and will not be used.

Example: suppose R_1=120000000, $R_2 = R_1 \cdot 100$, X= $Wsum\_size$ =16, $\sum = R_2$,

T_1=1.9392E+11 and T_2= 3.84E+11 and the difference is about 50% (refer to appendix E for more details).

While if we suppose that X=16 and r=32:

T_1 = 3.8592E+11 and T_2= 5.76E+11 where the difference is 33%.

Conclusion of comparing 'a', 'b', and 'c' cases

After comparing the storage size for cases ('a','b', and 'c'), we found that case 'c' is the worst case and case 'a' is the best. But we will follow case 'b' in our index construction as case 'a' suffers from the following disadvantages:

- *Wsum* value is not applicable for search, it is not unique and repetition is allowed.
- No suitable index structure or a clear index structure can be used. Case (a) can be used for searching if we consider using composite index, which require more space for the index size.
- The storage space needed for the total storage size is close to case (b), and this makes case (b) a candidate case.

4.3 BTree Index structure based on transformation

Figure 4.1 shows our idea to store *Wsum* values, according to case (b) implementation. Each node consists of three parts: *Wsum* values, linked list pointer, and a pointer to lower level of the index. The linked list holds the number of the sequence that contains the *Wsum* value.

This structure is a modified classical BTree structure; it is a multi-level index structure starts from level 1 till level L. The fact that the *Wsum* value may belong to more than one sequence lead to the use of a linked list, as shown in Figure 4.1. The difference between this structure and the BTree is the use of linked list to point to more than one DNA sequence. For example the *Wsum* value 245 linked with more than one sequence.

This section will discuss three subjects. Firstly: general introduction to tree structure, secondly: comparing the use of a modified BTree with a modified B+Tree structure to build index for the *Wsum* values. Thirdly: comparing building index structure based on wavelet transformation with index structure based on tree without the use of transformation.

4.3.1 Introduction to Tree structure

Tree is an undirected linked vertices where the link between vertices represents a path that is not a cycle. Tree degree refers to the number of links for each vertex. Refer to Section 2.1.7.3 for more details.

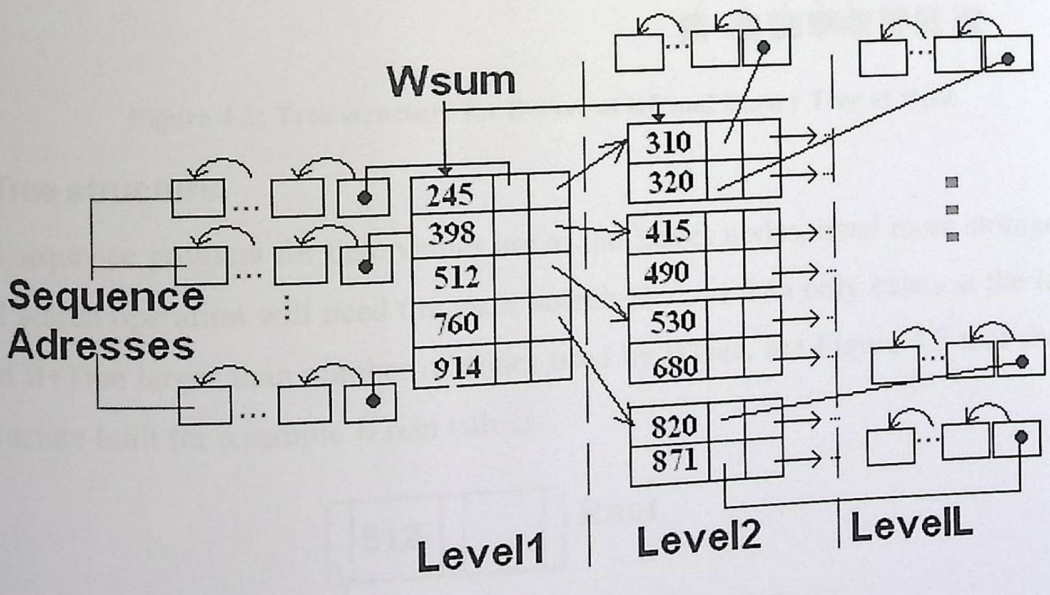


Figure 4.1: Index structure, modified B-Tree

4.3.2 BTree structure

The most efficient trees, for our needs, are the B+Tree and BTree over other types we have studied including: Binary Tree, Red-black tree, AVL-Tree, RB-Tree,RTree, Suffix tree, AA-Tree, and R-Tree. Number of branches, also called tree order, makes the difference between BTree and binary

tree; binary tree has two branches only not more as shown in Figure 4.2. For a large number of keys, binary tree will have a higher depth than a tree with larger number of branches.

BTree can store many keys in a few levels, and increasing number of branches will decrease number of tree levels. Figure 4.2 shows the BTree structure compared with binary tree structure. Access time is related to the number of nodes we need to pass through till reach the desired record. BTree reduces access time compared with binary tree.

In large index size, if not fit in random access memory (RAM), will be stored at a storage medium. BTree minimize the number of times this medium need to be accessed to allocate the needed record, thereby speeding up the process.

The access time for storage medium, like hard disk, is longer than the access time for RAM because of the mechanical devices. Mechanical device read and write operation is very slow compared with RAM.

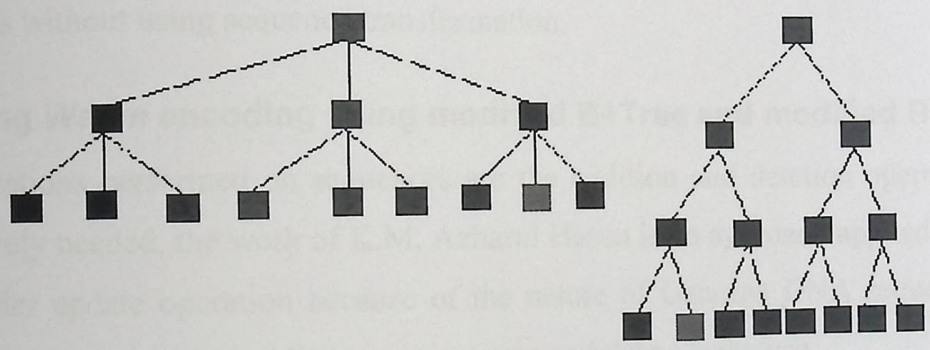


Figure 4.2: Tree structure for B-Tree at left and Binary Tree at right

4.3.3 B+Tree structure

The actual sequence pointers for data values are at the leaves nodes, need more storage space than BTree. All search operation will need the same search time, as data only exists at the leaf. Number of nodes at B+Tree larger than number of nodes used by BTree, see Figure 4.3 that shows B+Tree general structure built for a sample *Wsum* values.

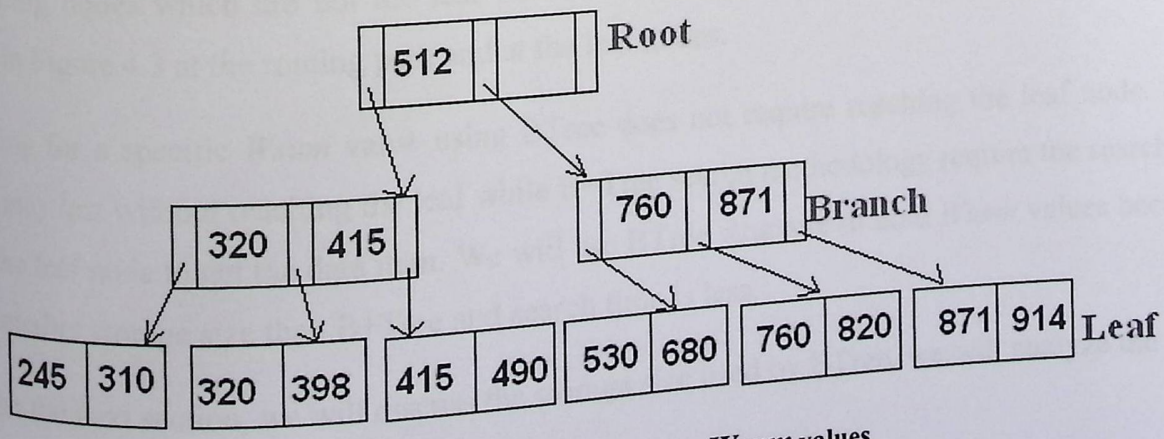


Figure 4.3: B+Tree example for *Wsum* values

tree; binary tree has two branches only not more as shown in Figure 4.2. For a large number of keys, binary tree will have a higher depth than a tree with larger number of branches.

BTree can store many keys in a few levels, and increasing number of branches will decrease number of tree levels. Figure 4.2 shows the BTree structure compared with binary tree structure. Access time is related to the number of nodes we need to pass through till reach the desired record. BTree reduces access time compared with binary tree.

In large index size, if not fit in random access memory (RAM), will be stored at a storage medium. BTree minimize the number of times this medium need to be accessed to allocate the needed record, thereby speeding up the process.

The access time for storage medium, like hard disk, is longer than the access time for RAM because of the mechanical devices. Mechanical device read and write operation is very slow compared with RAM.

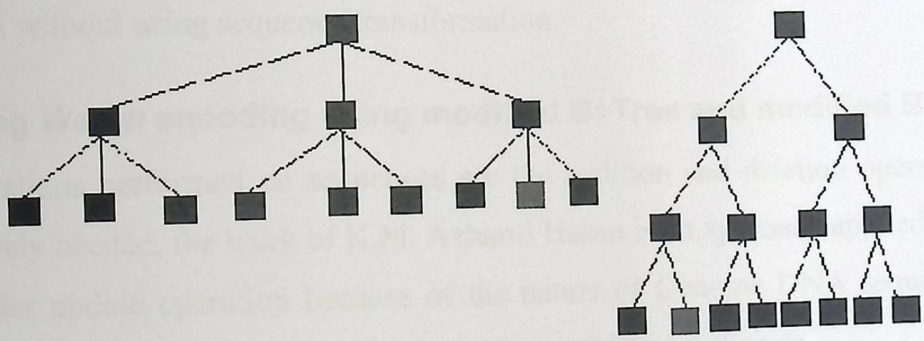


Figure 4.2: Tree structure for B-Tree at left and Binary Tree at right

4.3.3 B+Tree structure

The actual sequence pointers for data values are at the leaves nodes, need more storage space than BTree. All search operation will need the same search time, as data only exists at the leaf. Number of nodes at B+Tree larger than number of nodes used by BTree, see Figure 4.3 that shows B+Tree general structure built for a sample *Wsum* values.

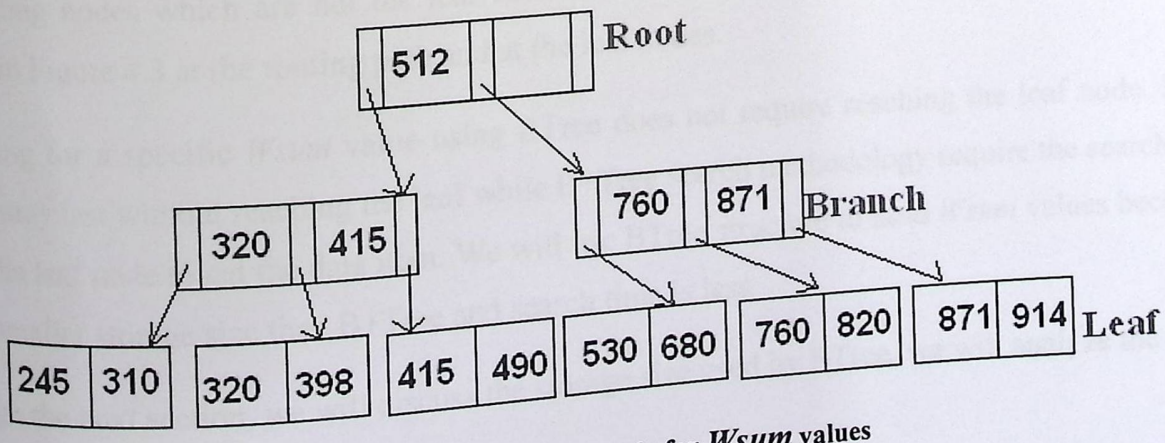


Figure 4.3: B+Tree example for *Wsum* values

There are many different tree structures, like BTree, Red-black tree, AVL-Tree, RB-Tree, RTree, Suffix tree, and AA-Tree, that can be used to build index for the *Wsum* values.

BTree structure is suitable for situation with large number of lookups and a little amount of update operation while AVL-Tree is suitable for large amount of lookups and a lot of modifications, RB-Tree suitable for little number of lookups and a lot of modifications.

RTree [38] is suitable for approximate queries like K-nearest neighbor (KNN) query and range query. Suffix tree produces large set of permutations and this number increase when using different window sizes.

BTree get advantage of the cache, it reduces cache miss [60] especially when node size equal cache line size because it put multiple keys in one node.

The following section analyzes the advantage of using BTree over B+Tree to build a tree structure for the *Wsum* values. Next section compares the M-WTBT index structures with the tree based index structures without using sequence transformation.

4.4 Comparing *Wsum* encoding using modified B+Tree and modified BTree

The main operations performed on sequences are the addition and deletion operation, the update operation is rarely needed, the work of K.M. Azharul Hasan is an approach applied for B+Tree. We will not consider update operation because of the nature of Genome DNA sequences, almost all operations on database is either adding new sequence or delete a sequence.

Our work for building index structure requires using the modified BTree instead of B+Tree to increase indexing efficiency. The main differences between the modified BTree and the modified B+Tree are in index storage space and searching methodology.

B+Tree require more storage space size because of the rout nodes and the storage of data values at leaf nodes only. This means that extra nodes are needed for B+Tree than BTree. Those nodes are the routing nodes which are not the leaf nodes. For example the *Wsum* values {415, 760, 871} shown in Figure 4.3 at the routing path and at the leaf nodes.

Searching for a specific *Wsum* value using BTree does not require reaching the leaf node. BTree search may last without reaching the leaf while B+Tree search methodology require the searching to reach the leaf node to get the data item. We will use BTree structure to hold *Wsum* values because it needs smaller storage size than B+Tree and search time is less.

Through the next section, we will discuss the storage size used by BTree, we will analyze the search time and the index storage size.

4.4.1 Tree search time:

Number of nodes [68] or index keys for a tree in general (see proof at Appendix A):

$$\frac{[N^L - 1]}{N - 1} \quad \text{.....(4.14)}$$

Where:

N: number of branches and

n: number of *Wsum* values, which are the actual index entries and

L: number of levels.

And:

$$L \leq Log_N n \quad \text{.....(4.15)}$$

Equation 4.15 is not applicable for the B+Tree structure. The number of nodes at B+Tree is larger than n because of routing nodes.

Searching time for a tree [68] equal the tree depth 'L' plus the search time through each node that passed through. Searching though node can be done by different ways, there is sequential searching that requires at maximum N comparisons:

$$\text{Search time} = O(N \cdot L) = O(N \cdot Log_N n) \quad \text{.....(4.16)}$$

And if binary search is used instead of sequential searching, search time complexity becomes:

$$= O(Log_2 N \cdot Log_N n) \quad \text{.....(4.17)}$$

4.4.2 Index storage size:

Each node, at each level except the last level, contains the following parts: the node (contains *Wsum* value, Next Level Pointer *NLP*, and substring position), sequence pointer, and node contents counter as shown by Figure 4.4. The substring position will be used for full-alignment at the last step.

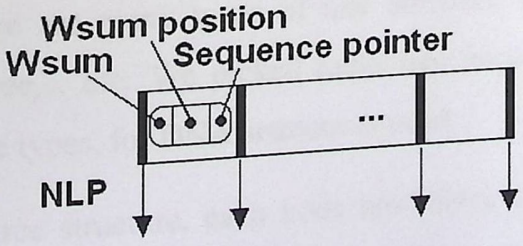


Figure 4.4: Tree node structure.

By referring to Section 4.1.2, the index size parts are:

a. Sequence pointer (*SP*) size equal to *X*, for one node it equals (N-1)·X.

b. *Wsum* value size = r bit.

c. NLP: it depends on L and number of nodes, number of pointers (NP) equal:

Number of keys/node capacity =n/N and size for all nodes = $NP \cdot Log_2 NP$ bits.

d. *Wsum* position:

We need ($L_i - W_x$) position values, each value need log ($L_i - W_x$) bits for encoding, the total cost of adding position value to the index node is $(L_i - W_x) \cdot Log_2(L_i - W_x)$.

e. Node Counter (NC) indicate number of items filled at node, up to a maximum value (N-1). In our work we did not use counter, instead we check all pointers. The internal pointers will be checked till find the required key or found nothing.

The total size of the whole structure will include nodes and pointers where:

One Node total size = *Wsum* size+ sequence pointer (N-1) · *Wsum_size* + (N-1)·X

So the total size of the index structure for one node using the modified BTree is:

Total size of one node = $(\frac{L_i - W_x}{N-1})$ ·(N-1)·(*Wsum_size*+ SP + *Wsum* position) + NLP …(4.18 a)

Where:

$NP = n/N$, $n = Li - W_x$, number of nodes = $(n/N-1)$, L_i is the sequence length.

And total number of nodes is = (L_i-W_x)/(N-1) …(4.18 b), then

$$=(L_i - W_x)[Wsum\_size + \mathrm{SP} + Log_2(L_i - W_x)] + \frac{n}{N-1} \cdot N \cdot Log_2(L_i - W_x) \quad \text{.....(4.18)}$$

And the *Wsum* position value is not used:

$$(L_i - W_x)[Wsum\_size + \mathrm{SP}] + \frac{n}{N-1} \cdot N \cdot Log_2(L_i - W_x) \quad \text{.....(4.19)}$$

4.5 Comparing radix tree with Wavelet transformation

For a database of sequences, the Genome DNA sequence need to be represented by a character format instead of integer format, using a tree structure, to compare it with our work based on wavelet transformation. There are many types of tree structure that can be used, balanced, un balanced tree, BTree, RB-Tree… etc. We picked radix tree because it is the most suitable tree structure, over the studied tree types, for DNA sequences need.

Figure 4.5 shows the radix tree structure, each node hold three characters and can be presented using one character as shown in Figure 4.6. Each path from root till leaf represent a character permutation, this figure uses 3-gram windowing.

Figure 4.6 shows all permutations, from the DNA genome characters, for W_x=4 for substrings that start with the character "A" as a root node. For the whole permutations we need another three tree starts with the characters {C, G, T} as a different root node. We will compare WT and Tree based index structure in term of index size, transformation cost, and search time.

The window size W_x affects the path at the tree. Increasing W_x will require adding more nodes to the path. For example if W_x =3, number of passing nodes is 3 nodes from the root, while if W_x =8 number of passing nodes will be 8 nodes from the root.

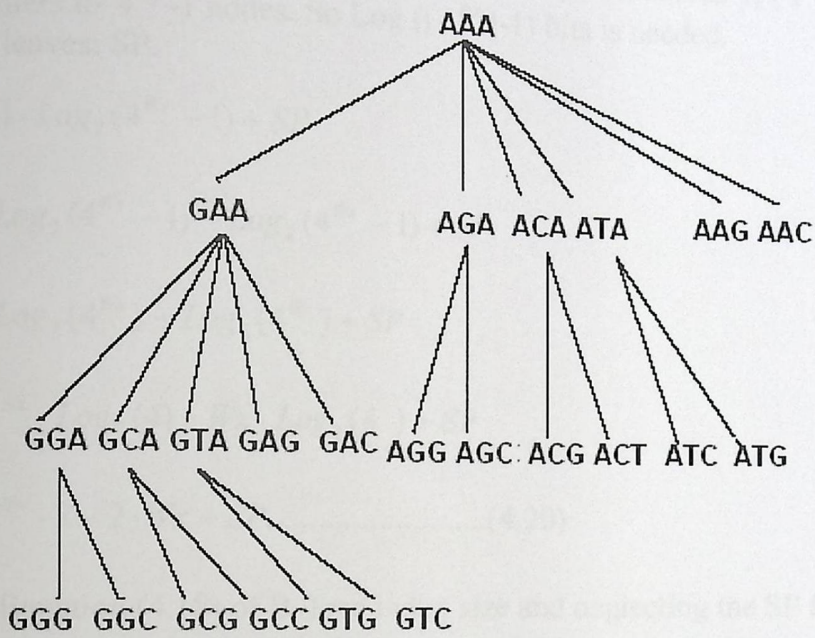


Figure 4.5: Radix Tree based encoding for DNA sequence using Wx window size.

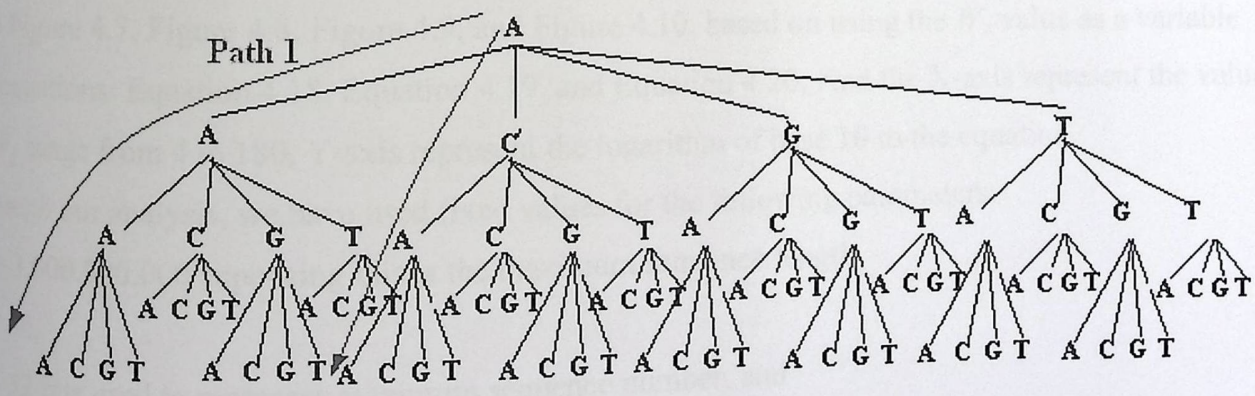


Figure 4.6: 'A' Tree root for DNA sequence holds one character at each node

Transformation cost will be neglected because it is a one time cost used when building the index. The following analysis is for the Index structure based on Radix tree to compute the total storage size in order to be compared with Equation 4.18.

4.5.1 Index size

The number of levels for the tree depends on the size of the window (W_x) used to scan the sequence. The total size of the index is the nodes, links between nodes (pointers), and lastly the linked list at

the last level that points to the sequences. The linked list field is identical to "SP" used at Equation 4.18. The total index size can be calculated as follows:

1. Number of nodes for a tree:

$\frac{[N^L - 1]}{N-1}$, where L = W_x

2. Size of each node is 8bit [11], referring to 128 ASCII*, to represent each character of the DNA sequence {A, C, G, T}.

3. Number of nodes links of one tree is equal the number of permutations-1, (4^{Wx})-1. And size of link is, we need pointers to 4^{Wx}-1 nodes. So Log ([4^{Wx}]-1) bits is needed.

4. Linked list at the leaves: SP.

$$\frac{[N^{Wx} - 1]}{N-1} + (4^{Wx} - 1) \cdot Log_2(4^{Wx} - 1) + SP$$

$$= \frac{[N^{Wx} - 1]}{N-1} + 4^{Wx} \cdot Log_2(4^{Wx} - 1) - Log_2(4^{Wx} - 1) + SP$$

$$\cong \frac{[N^{Wx} - 1]}{N-1} + 4^{Wx} \cdot Log_2(4^{Wx}) - Log_2(4^{Wx}) + SP$$

$$= \frac{[N^{Wx} - 1]}{N-1} + Wx \cdot 4^{Wx} \cdot Log_2(4) - Wx \cdot Log_2(4\ ) + SP$$

$$= \frac{[N^{Wx} - 1]}{N-1} + Wx \cdot 4^{Wx} \cdot 2 - 2 \cdot Wx + SP \quad \text{.....(4.20)}$$

Compared with the Equation (4.18) of B-Tree index size and neglecting the SP field:

$$Wsum\_size\ (Li - Wx)[(N-1) \cdot Wsum\_size + SP + \frac{n}{N} \cdot Log_2(n/N)\ ] + (Li - Wx) \cdot Log_2(Li - Wx)$$

The Figure 4.7, Figure 4.8, Figure 4.9, and Figure 4.10, based on using the W_x value as a variable for equations: Equation 4.18, Equation 4.19, and Equation 4.20. And the X-axis represent the values of W_x range from 4 to 180, Y-axis represent the logarithm of base 10 to the equation.

Through our analysis, we have used fixed values for the following parameters:

Li = 3.000.000.000 supposing this is the maximum sequence length,

N = 50,

X = 32 bits used to represent maximum sequence number, and

Wsum_size=32 bits.

Changing the W_x value affects Equation 4.18 at two parts:

$((L_i - W_x)[(N-1) \cdot Wsum\_size$, and $(L_i - W_x) \cdot Log_2(L_i - W_x))$

And affects Equation 4.20 at three parts: $\left(\frac{[N^{Wx} - 1]}{N-1},\ Wx \cdot 4^{Wx} \cdot 2, \text{and}\ 2 \cdot Wx\right)$.

The effect of the W_x value at Equation 4.18 is decreasing the value of L_i by subtraction $(L_i - W_x)$.

* ASCII character codes and html, http://www.asciitable.com/, 2011.

But the effect of W_x at Equation 4.20 is increasing by power of (N^{Wx}) and multiplication with W_x ($W_x \cdot 4^{Wx} \cdot 2$). So increasing W_x value will decrease the output of Equation 4.18 and increase the output of Equation 4.20.

Increasing the W_x value affects the index structure based on "WT transformation & modified BTree-WTMB" by decreasing the storage size and affects the index structure based on Radix tree structure by increasing the index size, see Figure 4.7 and Figure 4.8.

The change of the value of Equation 4.18 by increasing W_x value is about 12% (18.03 at W_x =4 and the last value is 16.26 at W_x =180). On the other hand, the effect of W_x at Equation 4.20 is larger, it is 24 times larger (the first value is 11.90 and the last value is 304.12). See appendix B for detailed values over W_x values for the three Equations 4.18, 4.19, and 4.20.

Changing W_x value not affecting our structure a lot, but it increases index structure based on tree.

Notice the sharp increase of index size while increasing W_x value at Figure 4.7. Referring to Figure 4.8, there is a little change on the index size during increasing W_x value, it is almost stable.

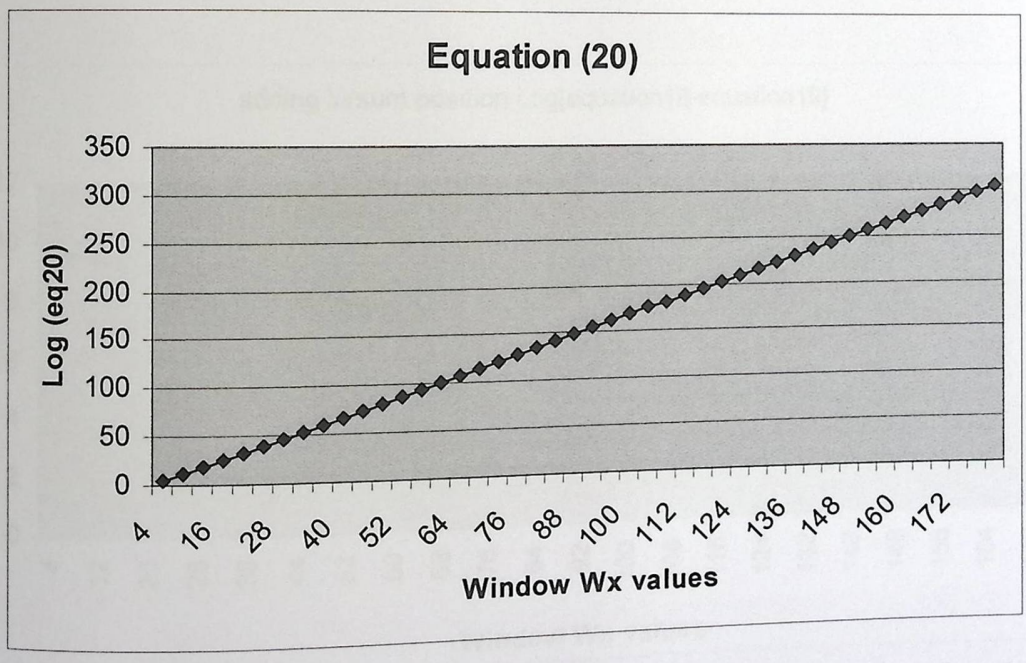


Figure 4.7: The output values for the logarithm of Equation 4.20 by changing Wx value

Figure 4.9 shows the effect of adding *Wsum* position to the index structure. This figure represents the logarithm difference between the two Equations 4.19 and 4.18 (the difference in size Equation 4.18 – Equation 4.19). As we can see from the figure, the difference of size between adding the *Wsum* position or not adding *Wsum* position start at value 10 and ends around value 8 from a maximum value of 10. Increasing W_x value will decrease the difference of size between the two equations, start by 6,656,295,936 at W_x =4 and end by 120,362,768 at W_x =180.

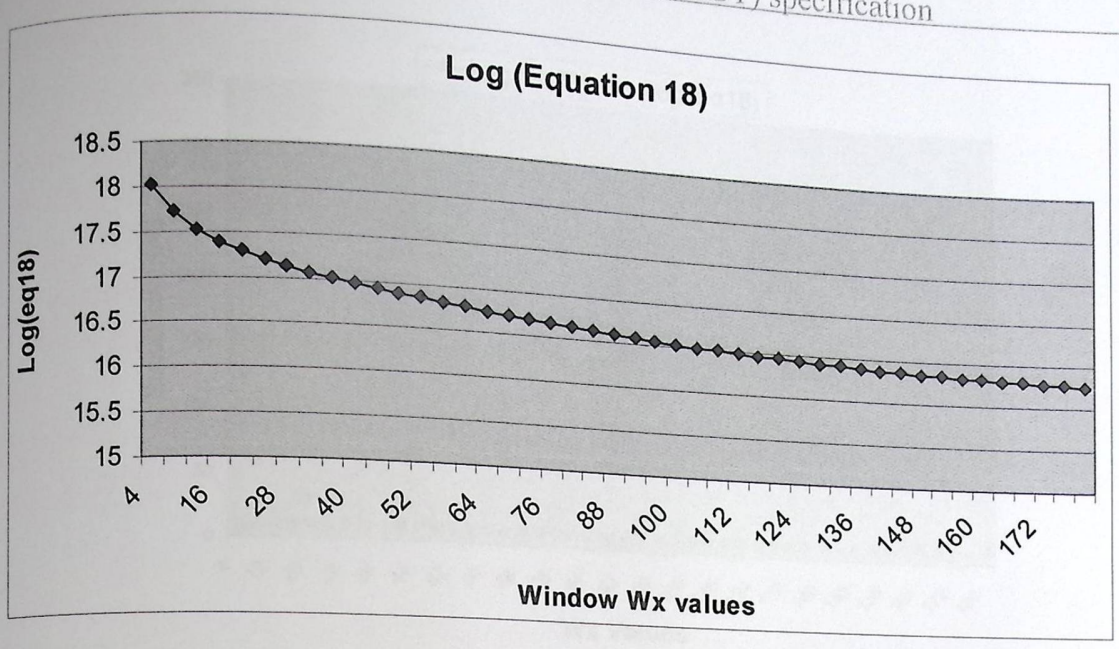


Figure 4.8: The output values of the logarithm of Equation 4.18 by changing Wx value.

Figure 4.10 shows the two curves of comparing the radix tree, referring to Equation 4.20, and the Wavelet Transformation for index structure, referring to Equation 4.18. Figure 4.18 displays two curves for the logarithm of Equations 4.18, and 4.20. As we can see, our index storage is almost stable during increasing window sizes, but the index size based on tree increases quickly.

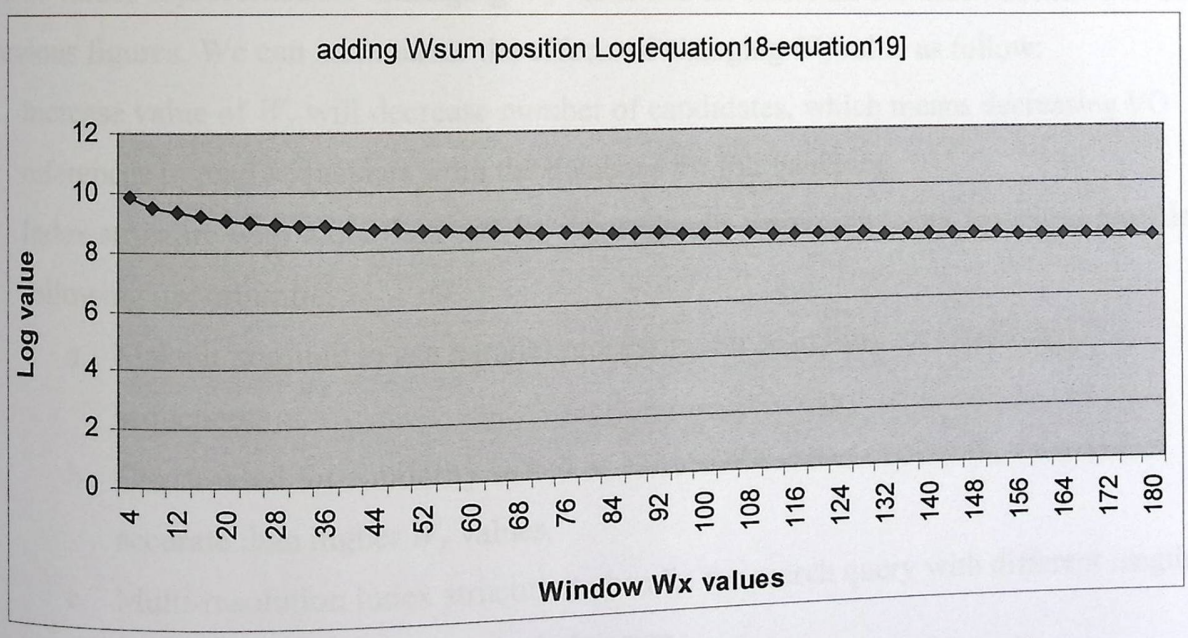


Figure 4.9: The logarithm of the difference between Equations 4.18 & 4.19.

4.5.2 Search time

The search time depends mainly on the number of comparisons needed. For tree based structure, number of comparisons required equal to W_x, which is the tree height. And for the modified WT, it needs W_x comparisons too, so both structures require the same number of comparisons.

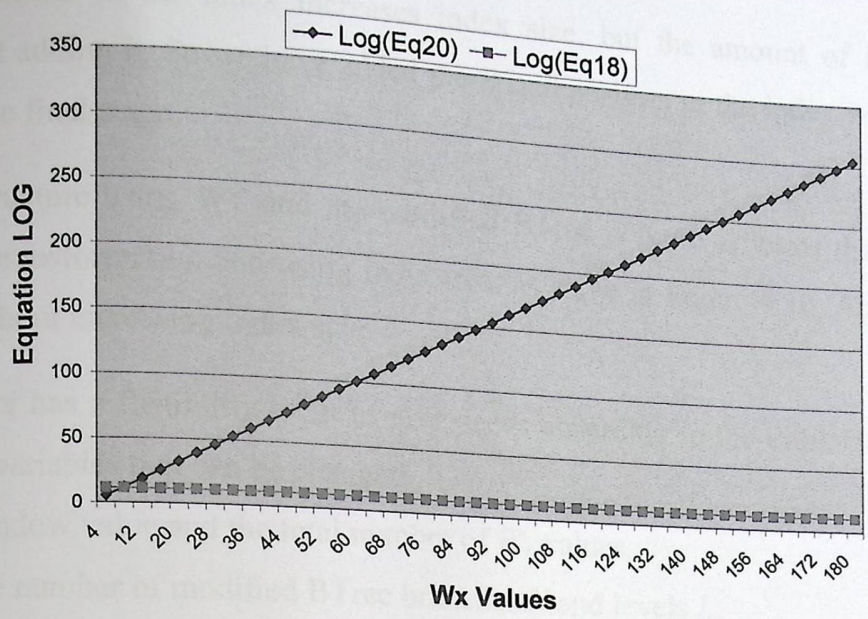


Figure 4.10: The logarithm of Equations 20&18 while changing Wx value.

4.6 Conclusion of comparing M-WTBT and Tree based without transformation

We have used W_x as a variable value and other parameters can be divided into two categories, the database data related and index related. The database related is L_i "sequences length", n sequences, and X bits for sequence numbers. The index related variables are N branches, and $Wsum\_size$ bits for $Wsum$ values representation. Changing W_x value has an effect on the index structure as seen by the previous figures. We can summarize the effects of changing W_x value as follow:

1. Increase value of W_x will decrease number of candidates, which means decreasing I/O references to read sequences from the database for full matching.
2. Index structure with more than one W_x values (multi-resolution index structure) provide the following opportunities:
    a. Make it possible to use parallel processing for searching for candidate DNA sequences.
    b. Can be used for similarity search as a threshold value, smaller W_x value is less accurate than higher W_x values.
    c. Multi-resolution index structure to handle the search query with different lengths.
3. Increasing W_x value will increase the index size.
4. The building time for index structure for more than one W_x value is higher than index structure with one W_x value. The use of WT will enhance this issue; using WT we can do transformation for a higher W_x value with no need to reference original sequence again.

Picking the right value of W_x is a user issue, depending on his needs. Index structure with more than one W_x value allows the use of parallel processing and intersection to have set of valid candidate for final referencing.

Adding *Wsum* position to the index increases index size, but the amount of increase is small compared with not adding it. So we have added the *Wsum* position to the index which will be used for alignment at the final stage.

Building index structure using WT and the modified BTree is more efficient than building index structure without transformation and using radix tree, as show in Figure 4.10. And it allows using large W_x value without increasing index size.

Our index structure has a flexibility to fit to user needs according to the available resources. User has the following variables that can be changed:

- Window value and the total number of W_x values.
- The number of modified BTree branches N and levels L.
- We used 1-gram WT, using n-gram WT is also applicable and will reduce index size. The value of n also a user defined.

We conclude that using our index structure is more effective over index structure built without character transformation and using Radix like tree based, and provides the ability to build index with flexibility according to user resources.

4.7 Enhancement for saving index size

In the process of designing and analysis of this method, a few ideas were generated for expanding and improving this work.

In order to decrease index size instead of storing *Wsum* values at the modified BTree, which we suppose it is up to 32bits, we will use the *Wsum* difference between values. The following steps explain this idea:

The first step is *Wsum* values ordering: put all *Wsum* values in ascending order, $W_s = \{W_1, W_2, \ldots, W_n\}$.

Second step is building difference list according to the following $d = \{W_1, W_2\text{-}W_1, \ldots, W_n\text{-}W_{n-1}\}$.

Last step is the modified BTree filling: select the first value from the order list Ws to be the root, and select N values following the root to be filled at level one node.

Figure 4.11 and Figure 4.12 is an example of how the modified BTree index structure will store keys. The aim of using *Wsum* difference instead of the actual value is to save index storage size because *Wsum* difference will need lower number of bits than the actual *Wsum* value needs.

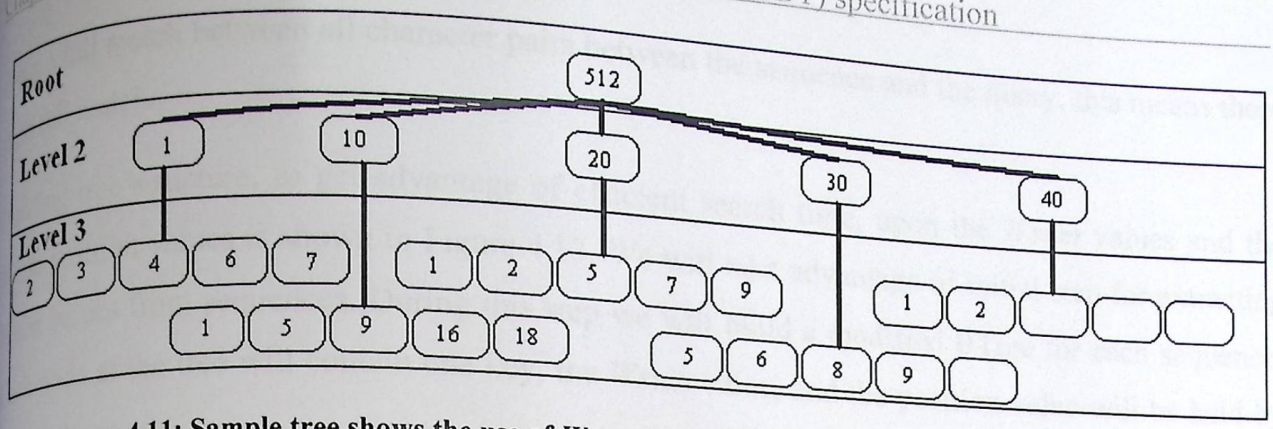


Figure 4.11: Sample tree shows the use of *Wsum* difference (1) instead of the actual *Wsum* values.

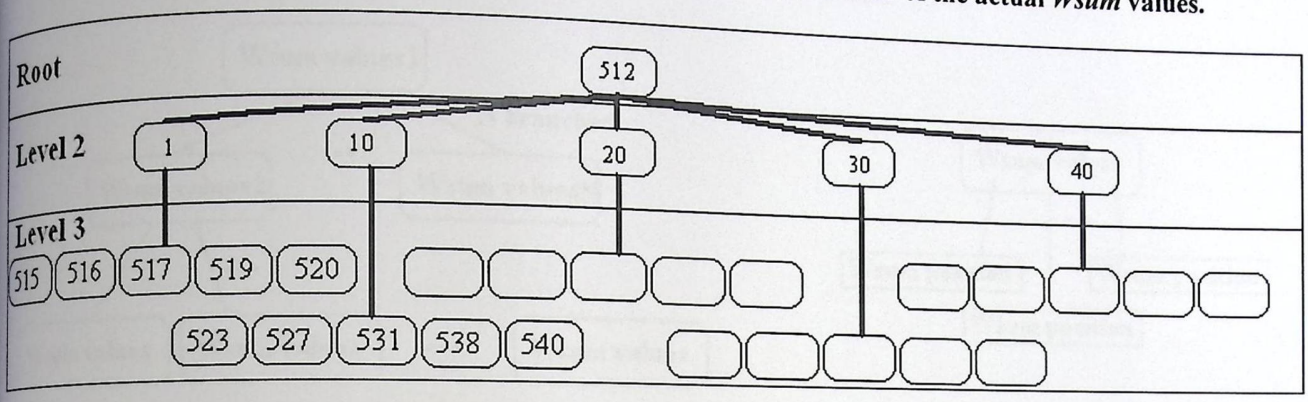


Figure 4.12: Sample tree shows the use of Wsum difference (2) instead of the actual Wsum values.

For example if new query need to search for the *Wsum* value 523, searching path will start from the root (512) then we compare the difference between 523 and the root (which is 11). Then compare the difference with the next level nodes, needed node will be the next node containing value 10. Then difference will be 11-10=1, and by comparing with the last level, first node after node 10 that contains the value 1 is our need. The path will be (root -> 10 -> 1).

4.8 Our method for sequence full alignment

After filtration step, a list of candidate sequences is ready to be checked against the query. Full alignment, with no gap, is needed on those candidates with the query sequence to have the final output, either a matched sequence/s or no match sequence/s.

We will discuses two situations, the index structure with *Wsum* position and the index structure without *Wsum* position value, and show the use of alignment in both cases in the following two sections.

4.8.1. Full alignment with *Wsum* position value

We need to do full alignment using the position value between the candidates and the query sequence.

If Pair-wise alignment is used between the two sequences; the alignment will be done character by character. This method starts at a common substring then do comparisons in both directions, before and after the common substring. In other words comparison will be done left and right. If the result

is not a full match between all character pairs between the sequence and the query, this means there is no full match.

Building tree structure, to get advantage of efficient search time, upon the *Wsum* values and the *Wsum* position values is shown in Figure 4.13. We will take advantage of initial step for extracting *Wsum* values from sequences. During this step we will build a modified BTree for each sequence. Each node at the tree will contain one key, the *Wsum* value, and the position value will be held by the pointer.

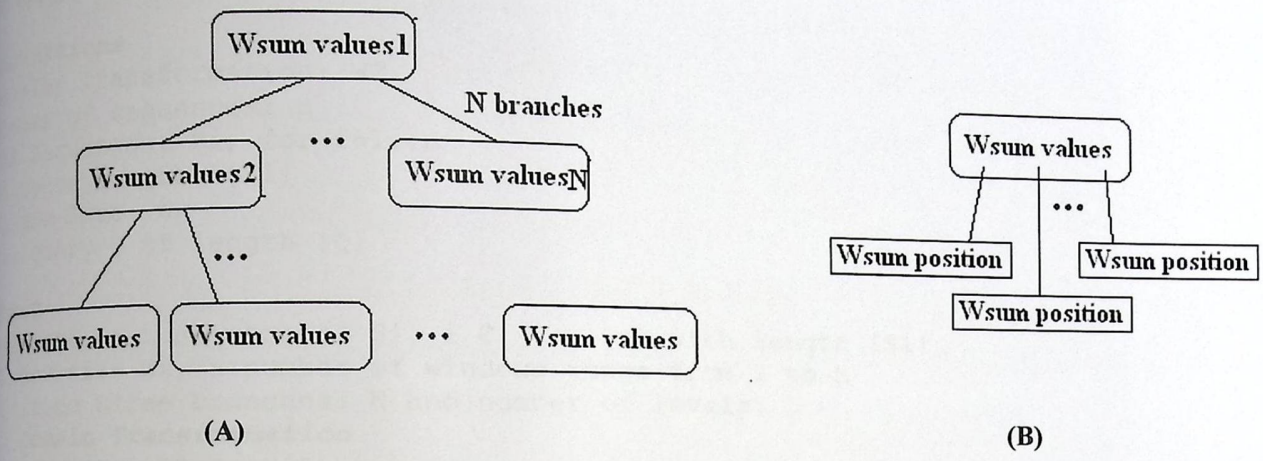


Figure 4.13: Modified BTree for each sequence with *Wsum* position value.

For each sequence, the modified BTree is used to hold the *Wsum* value at nodes and pointers are used to indicate the *Wsum* position for each *Wsum* value, in case that the *Wsum* value appears more than one time at the sequence.

Figure 4.13 B is the internal structure at each node. Each node has pointers to *Wsum* position for each *Wsum* value, so the number of position pointers for each node is N-1 pointers. Last step is to do the alignment starting from the given *Wsum* value, again full alignment is needed, any well known method for BTree comparison can be used at this step.

Suppose we have two trees A and B, and A= $\{A_{root}, A_{left}, A_{right}\}$, B= $\{B_{root}, B_{left}, B_{right}\}$

$A=B \leftrightarrow A_{root}=B_{root} \wedge A_{left}=B_{left} \wedge A_{right}=B_{right}$,

A recursive processing on both trees is needed and compare each single node, and as soon as one value is differ, stop scanning and return not equal trees. Bellow is a simple pseudo code:

Algorithm 4.1
Tree alignment algorithm:

```
int compare_tree_node( BTree A, BTree B){
if (A is empty) {
    return -1;}else{
    if (B is empty) {
        return -1;}else{
    check if Aroot = Broot && Aleft=Bleft && Btight=Bright
    return 1;
}//end else
}//end compare tree node
```

4.8.2 Alignment with *Wsum* value only

This case assumes no *Wsum* position is added to the index structure during the index building. In this case we can follow one of the well known methods used for sequence alignment. These methods include global alignment, dynamic programming, Basic Local Alignment Search Technique (BLAST variants), and First Fast Sequence Searching Algorithm (FASTA).

Algorithm 4.2

M-WTBT Index structure algorithm:

Definitions
Wavelet Transformation: WT
Number of sequences: n
DNA Sequences: Si, for i=1..n
Sequence length: |Si|
Window Size: Wx
New query Q of length |Q|

Input
Database of n sequences, Si, i ∈ [1... n] with length |Si|
Window size Wx, x:number of windows range from 1 to h
Modified BTree branches: N and number of levels: L

1. Domain Transformation
For each Wx value, x=1..h
  For each Si
    Scan Si using Wx to calculate WT-1 gram coefficients {A_cof, B_cof}

2. Integer domain transformation
    A_cof = A1.v + A2.w + A3.t + A4.r
    B_cof = B1.v + B2.w + B3.t + B4.r
    Calculate *Wsum* : *Wsum*=(A_cof.0.5+B_cof)
    Remove duplicated *Wsum* values

3. Modified BTree index structure (Wsum)
    Fill the modified BTree index structure with *Wsum* values and sequence pointers

4. Query Searching
    Search for a new sequence Q of length |Q|.
    For each Wx value, x=h..1 such that Wx<|Q| do
    Convert Q to WT using step2: output is |Q|/Wx subsequences (NQi)
    Search the index structure for matching according to section 4.10
    End loop if match is found or continue till last value of x

Chapter 5

Evaluating M-WTBT index size (implementation details)

Through this chapter we will compare the M-WTBT index structure size and properties with different previous works using the following sample data: chromosome 18, chromosome 20, and chromosome 22. A set of experiments was done, the main objectives was to test the proposed methods and asses the validation of our work.

The implementation of M-WTBT has been developed using Matlab and C++. The experiments has two steps; the Genome DNA sequence transformation to produce *Wsum* values using Matlab 7.7, and building the index structure using C/C++ for the transformed data of CSV format.

5.1 Material and methodology

For our experiment, we used the sample data shown in Table 5.1. The number of base pair and file size is listed at Table 5.1. The sample data consists of three chromosomes of the 23 pairs of chromosomes in humans. People normally have two copies of this chromosome.

Table 5.1: Sample data used for the experiments.

Chromosome	Number of bp	File size
18	85 Million	79,638,800 bytes
20	66 Million	64,286,038 bytes
22	49 Million	52,330,665 bytes

Figure 5.1 bellow overview the main steps we follow during our experiments. The input data in FASTA format is transformed to Modified WT using Matlab. The input sequence is extracted without the header.

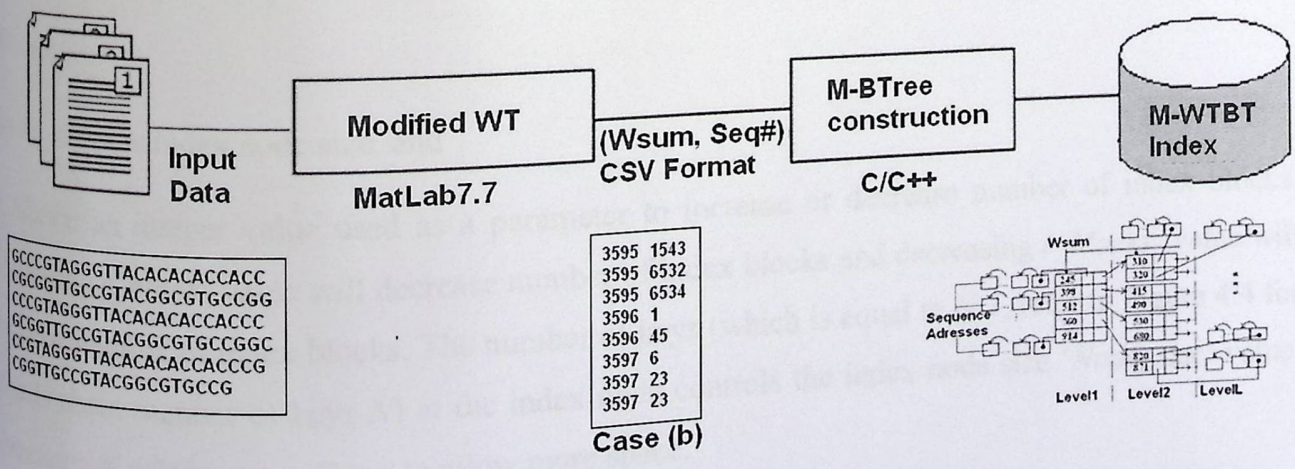


Figure 5.1: M-WTBT experiment steps.

The transformed data (*Wsum*) is a comma list value (CSV) file format that will be used in index construction program. M-BTree program is implemented using C++ functions and the output will be the final M-WTBT structure.

5.2 Managing M-WTBT Index node and block size

Figure 5.2 shows the memory and index block relation, where each index block is holding set of index keys.

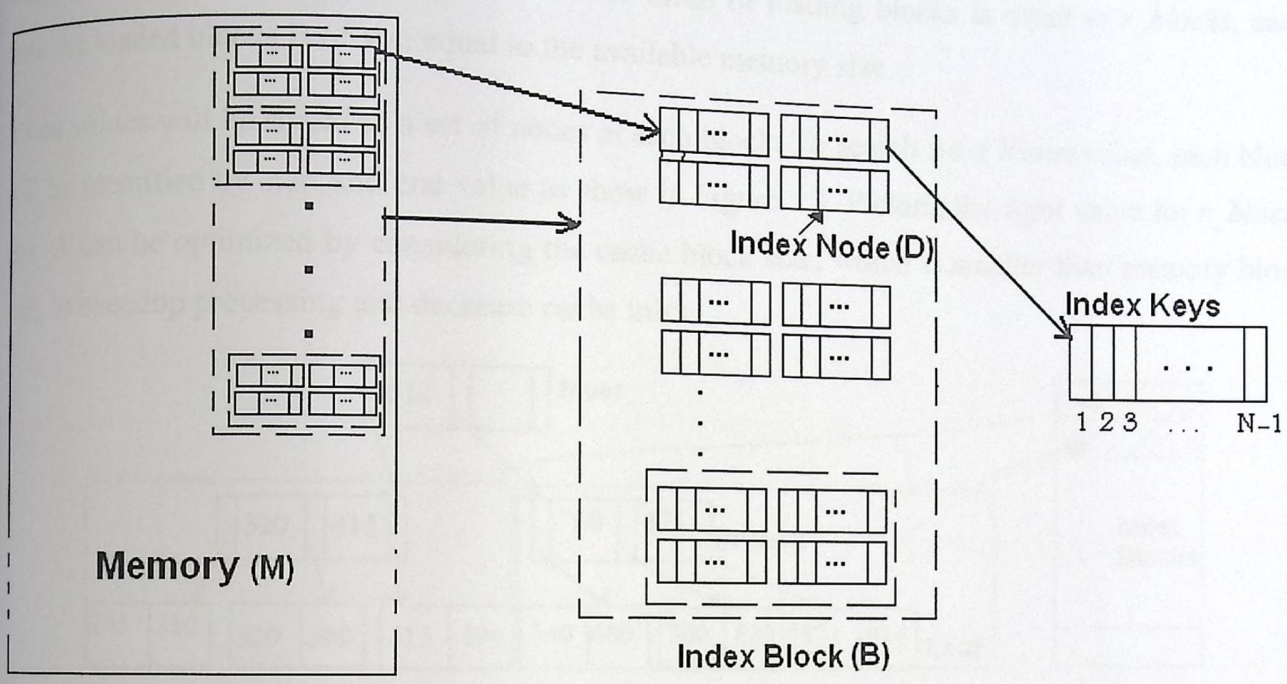


Figure 5.2: Memory, Index Block, and Index Node relation.

Index block size (B):

$$B = \frac{M}{r\_blocks \cdot Node\_size} \quad \text{.....(5.1)},$$

Where,

Node_size: the index node size, and

r_blocks: an integer value used as a parameter to increase or decrease number of index blocks. Increasing *r_blocks* value will decrease number of index blocks and decreasing *r_blocks* value will increase number of index blocks. The number of keys (which is equal to *N-1*, refer to Figure 4.4 for details about number of keys *N*) at the index node controls the index node size '*Node_size*' value, increasing *N* will increase *D* too to allow more space.

This is true if index can fit in memory, but if index size is larger than memory, partially loading of index can be used according to index block size as follow:

$$r\_blocks = \frac{M}{\text{Node\_size}} \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (5.2)$$

And

$$\text{Node\_size} = (N-1)\cdot s \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (5.3)$$

$$B = \text{Node\_size}\cdot r\_blocks = M \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (5.4)$$

Where s is the index key size. The number of times of loading blocks is equal to *r_blocks*, each time the loaded blocks is of size equal to the available memory size.

Wsum values will be sorted in a set of nodes at each block. To search for a *Wsum* value, each block will be identified by start and end value as show in Figure 5.3. Picking the right value for *r_blocks* and N can be optimized by considering the cache block size, which is smaller than memory block size, to speedup processing and decrease cache miss.

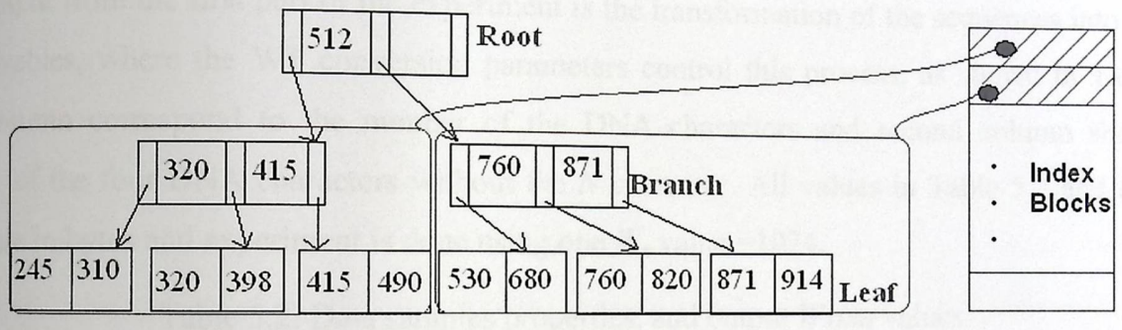


Figure 5.3: Index Block management.

The following pseudo-code describes an algorithms and the methodology for building M-WTBT index structure:

Algorithm 5.1

The Modified WT algorithm steps:

```
Function MWT (sequences folder D, WT conversion parameters, memory size M, window size
Wx, conversion parameters(1)){
loop through all sequences ( fasta file format at D) %loop 1
      open fasta file for read
      read file header
      set block size for loading according to M (2)
      determine number of total block to fill
      for each loaded block 1:number of blocks% for 1
            read one block
            scan by window size Wx
                  calculate A, C, G, T counters
            calculate {A_cof,B_cof} WT according to Wx
            calculate Wsum according to Wx
            store Wsum
      end% end for1
      for all Wsum values %for 2
            remove duplicated Wsum values
            Store result list to csv file format
      end for %end for2
end loop %loop 1
End function
```

Algorithm 5.2

The Modified BTree algorithm steps:

```
Initial parameters:
Wsum size 'Wsum_size, LL pointer size
MAX,MIN number of keys at node N 100, 50
struct Wsum {keyvalue, sequence pointer SP}
struct item {Wsum, seq#, pointer NLP}
Processing:
open csv data file
loop until EOF %loop 1
      read one line contains two pair values(Wsum, Seq#)
      check the tree if Wsum already exists
            search all values from right to left
      if exists then %if condition1
            add SP to Wsum struct
      else   add new item to node
      end if %if condition1
end loop %loop 1
```

(1) Conversion parameters: referring to section 4.1.1

(2) Memory size determines number and size of index blocks.

5.3 Experiments

First output from the first part of the experiment is the transformation of the sequences into distinct *Wsum* values, where the WT conversion parameters control this process, as shown in Table 5.2. First column correspond to the number of the DNA characters and second column shows the number of the four DNA characters without the *N* character. All values in Table 5.2 and the next tables are in bytes and experiment is done using one W_x value =1024.

Table 5.2: Data samples properties, and output *Wsum* values.

Chromosome	Sequence	{A, C, G, T}	*Wsum*
18	78,000,000	74,640,000	645,768
20	63,000,000	59,505,520	514,744
22	51,000,000	34,649,979	299,777
Total	192,000,000	168,795,499	1,460,289

Results show that the input data properties highly affect the *Wsum* output. As if original data has high degree of repetition the number of distinct *Wsum* values will decrease.

According to our sample data, we have applied Equations 4.18, and 4.19 referring to Section 4.4.2, to calculated required storage space by M-WTBT index structure for two cases (*Wsum* and *Wsum* with position). Columns Equation18 and Equation19, at Table 5.3, show the output of the two equations. The parameters we used are: chromosome length L_i, BTree branches $(N)=100$, $W_x=1024$, and $n=L_i-W_x$.

Then we have built the M-WTBT index using the *Wsum* values. Last step was to apply the proposed enhancement by Section 4.7 by using the *Wsum* difference instead of the actual value. The last column of Table 5.3 is the experiment result using the *Wsum* values. Table 5.4 shows the percentage, corresponding to the actual chromosome size, of each of the four cases: with *Wsum*

value according to Equations 4.18, *Wsum* with position according to Equation 4.19, using the enhancement, and the experiments.

Table 5.3: Data samples using Equation 18, Equation 19, and Equation 20 and experiment.

Chromosome	Equation18	Equation 19	Enhancement	Experiment
18	3,148,187,085	1,195,394,754	57,037,494	118,828,097
20	2,490,353,771	952,983,069	41,858,158	80,496,453
22	1,423,041,864	554,880,526	21,628,165	52,751,644
Total	7,061,582,720	2,703,258,349	120,523,817	252,076,194

Results obtained by Equation 4.19 are better than that returned by Equation 4.18, but both are very large compared with the original data size.

Table 5.4: Comparing original data size with index.

Chromosome	Equation18	Equation19	Enhancement	Experiment
18	4036%	1533%	73%	152%
20	3953%	1513%	66%	127%
22	2790%	1088%	42%	103%

Decreasing the number of the *Wsum* values lead to decrease index size. Using our structure, the index size is close to the input data size. When applying the enhancement over *Wsum* values, the index size become smaller than the input data, as shown by Table 5.4. During analysis at Section 4.4.2, values at the same sequence have no repetition which causes the large difference between Equations 4.18, 4.19, and the experiment result.

So, lets consider the amount of repetition at the same sequence denoted by repetition factor (`rf`), and suppose that the total number of *Wsum* produced by Equation X is: T*Wsum* for the Equation 4.18, Equation 4.19, and Equation 4.20.

$$rf = \frac{TWsum}{\text{output value from Experiment}} \quad \text{.....(5.5)}$$

Where.

Total number of *Wsum* values = T*Wsum*/rf

5.4 Environment

The experiments were conducted using Intel Core 2 Duo DELL PowerEdge 2950 Inter(R) Xeon® CPU, E5440 @2.83 GHz, 7.99GB of main memory and 2 × 2048 KB L2 cache, running Linux 10.2 (kernel 2.6.18). The program was compiled by the gnu C++ compiler 4.1.2 using optimization level; refer to appendix C for more details.

5.5 Memory restrictions

During the implementation of index structure, long sequences like chromosome 20 requires large memory and long time to accomplish processing. For example a single human DNA genome, with 1,320,000bp long, will require 366,882,848 bytes (350 MB) of memory for holding {A, B} WT coefficients.

The memory used by software is limited by the physical memory installed, despite that the physical memory can be extended by virtual memory set by operating systems parameters. The memory allowed for C++ is set using the parameters at limits.h file and depends on platform (MS Windows, Linux, or MAC). For example, an 'int' array defined by [1000][1000] under Ms windows 32 bit operating system will needs: $1000^{x}1000^{x}32=32$MB memory space. This means that comparing large sequences using the native DNA characters directly is virtually impossible.

5.6 Comparing M-WTBT index size

For comparing our work with previous works, it was hard to use standard data samples that are commonly used as a standard data for Genome DNA indexing and searching. We found it hard to compare our work with one specific work only, because there are a structure difference always exists either in sample data, searching method, or index properties. We used another approach of comparison; we have picked more than one previous work [51, 10, 12, 55, 24] and compared the common parts.

1) Comparing our work with the work of [51]: they used window size of 500 with 50% overlap amount, the initial size of the vector file (without compression) is between 5 and 50 % of the original sequence file size. Our index structure after enhancement, without compression, and with larger window size produces index size around 40-70% of the input sequences size as shown in Table 5.4.

2) The work of [10] compared two chromosomes of size 40MB, and it required 400MB free memory. There work used a very large size of memory because no transformation is used or feature vector to extract main characteristics. And for that we have used transformation to minimize memory requirements as mentioned at Section 2.2 and 2.2.2.

3) Different works of suffix tree [12] shows the high storage space complexity needed, most efficient work claimed that Suffix tree require *12.5n* space and the enhanced Suffix tree require *8n* space. Where n is the input data size. Our index size is much smaller than 8n.

4) The work of [55] build a tree (called ACGT-Words Tree) using more than one Wx value as shown in Figure 5.4. Figure 5.4 compares ACGT-Words Tree with Suffix Tree.

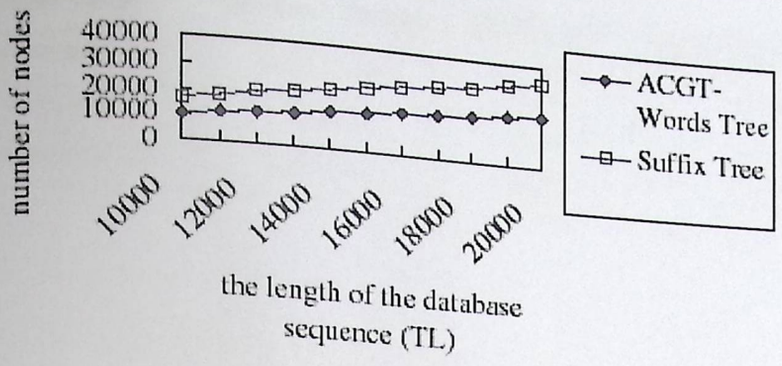


Figure 5.4: Comparing number of nodes for ACGT-Words Tree and Suffix Tree [55].

For a sequence with length=20000 bp, using N=100, and consider the repetition percentage is 0.4, we will have the following:

The number of nodes of M-WTBT (20000/N) ^{x} 0.4 = 80 which is very small compared with ACGT-Words Tree. Note that the repetition of *Wsum* values between sequences will not add new nodes to structure, but using the work of Chang, each value is considered a new addition to node even if it is a repetition.

5) Lastly, the work presented at [24], they have used the human chromosomes 18, 20 and 22 for experiments. They used the head subsequence of size W_x =*100, 200, 500*, and 1000 from a chromosome and then deleted the subsequence from the chromosome. Table 5.5 shows results of the work of [24].

Table 5.5: Efficient similarity search based on indexing in large DNA databases output [24].

	Size of window	The number of subsequences
Chromosome 18	100	2,244,146
	200	2,243,946
	500	2,243,346
	1000	2,242,346
Chromosome 20	100	1,702,151
	200	1,701,752
	500	1,701,152
	1000	1,700,152
Chromosome 22	100	3,661,382
	200	3,661,182
	500	3,660,582
	1000	3,659,582

Where we compared our work using W_x =1000, for chromosomes 18, 20, and 22 we found the following:

Table 5.6 shows the percentage of number of subsequences produced our work according to number of subsequences produced by Jeong work.

Table 5.6: Comparing the number of subsequences of In-Seon Jeong and M-WTBT.

Input data	In-Seon Jeong	M-WTBT	M-WTBT/ Jeong
Chromosome 18	2,242,346	645768	28.7%
Chromosome 20	1,700,152	514,744	30.2%
Chromosome 22	3,659,582	299,777	8.2%

Chapter 6

Discussion and Conclusions

The main purpose of having index system is to facilitate reaching the needed record in effective access method by minimize time and effort and maximize searching speed. Indexing systems uses filtration to decrease the number of candidate record, so no need to scan the whole database records to reduce number of I/O operations.

This chapter is dedicated to summarize the results and ideas presented in the previous chapters, as well as the conclusions that can be derived from the gathered and collated data.

Experiments results show the advantage of multiple fields over single field indexing for search time, even if special index type, like full-text, is used. Choosing the type of the index must consider the nature of the input data. Special type indexing based on previous knowledge of the data properties will produces best storage space, search time, and enhance data transformation.

After analysis for Genome DNA sequence properties, BTree index is used to hold the transformed data (*Wsum*) instead of B+Tree, with modification, to reduce index size. Comparing M-WTBT with Tree based indexing, where no transformation is used (where W_x is variable and parameters *Li, n, X, N,* and *Wsum_size* are fixed), reveals the advantages of our index structure over Tree based indexing.

The building of index system has a set of factors affecting the structure, input data properties, scanning method, data transformation, and searching. Concerning the M-WTBT, increasing W_x value will increase index size and decreases I/O operations; however, its value must be selected according to query length.

Using more than one W_x value enhances index structure by allowing: parallel processing, similarity searching, multi-resolution intersection candidates, and handle queries with different lengths.

Exact matching is the final stage of searching for exact matching, the use of *Wsum* position enhance the efficiency while increasing index size. For approximate matching, our structure is easily adapted to methods used for approximate alignment.

Analysis shows that our index structure occupied less storage size compared with Tree based indexing while changing W_x value. Experiment results reveal the decreasing in storage size according to *Wsum* repetition at one sequence. We conclude that our index structure is more effective than index structure built without character transformation.

Most previous research assumes a static index scenario: index is created from a collection of input data once, and then the structure is never changed. However, on productive real-world systems this is often not admissible. Instead, available resources may be changed; data size may increases or decreases. Therefore, adaptation to changes is supported by our index structures.

Our index structure has a flexibility to fit to user needs according to the available resources. User has the following variables that can be tuned and changed:

- Window value W_x and the total number of resolution.
- The modified BTree branches N and number of levels L.
- The use of N-gram WT instead of 1-gram WT, where index size will reduce for higher N value.
- Available memory size to determine index block and node size.
- Allow using similarity search instead of exact searching by: increasing amount of overlapping and using K-NN for distance measurement, and setting WT conversion parameters values so as allowing a range values for *Wsum*.

The suggested enhancement to reduce index size by storing *Wsum* difference, instead of the actual *Wsum* value, where applicable reduces index size heavily as shown by Section 5.2. We suggest two ways of using the *Wsum* difference: either using *Wsum* value at root node only or using *Wsum* at root and 'even' level numbers as shown by Figure 4.11 and 4.12. Both methods will reduce index size while require a little calculations using the sum function.

At the end of this study, we conclude that any index structure, for a huge size data, will contains the main functions shown in Figure 6.1. This figure shows the general characteristic of indexing system.

For a huge size input data, mostly the data is less in size than index, so why not loading data instead of the index. Loading input data require scanning all data files (see Appendix E for more details about Genome sequence file format), which require storage and search time cost and it is a slow operation. Data item size is larger than index item size, for example consider the process of one FASTA file compared with processing BTree node. And index systems allow a set of processes to manage the data, which is hard to be done using the actual data in original format, like partially loading according to a specific criteria, parallel processing, removing duplicate easily, analysis,..etc.

Input data require transformation to more efficient domain, and carefully considering its properties. Transformation will produce new data format that can be tuned using a feedback. Feedback is the tuning process between indexing and transformation to control *Wsum* properties (like size, range of values, single/composite index field, and index type).

For query searching, index may be fully or partially loaded according to: index size, available resources, and query to search for. If index size is small enough to fit in memory, the whole index will be loaded. If index size is larger than the available resources, partially loading need to be managed. Our index structure allows partially loading of index according to *Wsum* values.

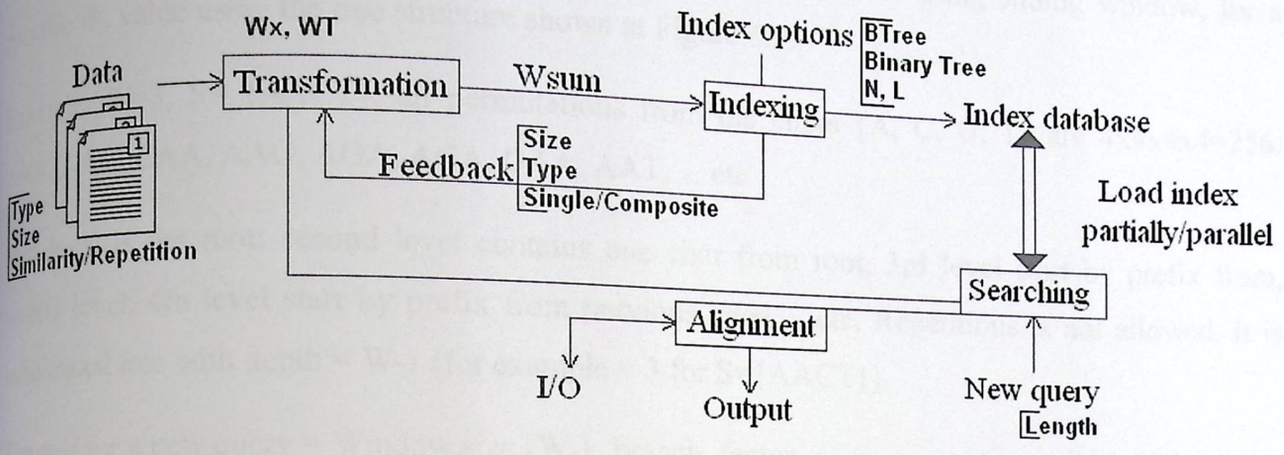


Figure 6.1: Overall indexing structure.

For new query searching, transformation is the first step to work on domain more suitable to index structured followed. Then index block/s, which containing the desired *Wsum* value/s, will be loaded to memory.

We can conclude as a result of this work, that indexing system enhances the overall application performance. A set of factors, related to input data and index construction, need to be considered to improve index structure to achieve better storage space and search time.

Suggestions for future work

This section introduces an additional work that could be done at a later date to build upon the current research. A number of open issues that arise during the development of our work that needs further studying and completion.

The first topic is to update the NLP to be implemented using displacement. Concerning the modified BTree, the next level pointer NLP can be changed to use displacement instead of the actual pointer for each link as the following figure shows:

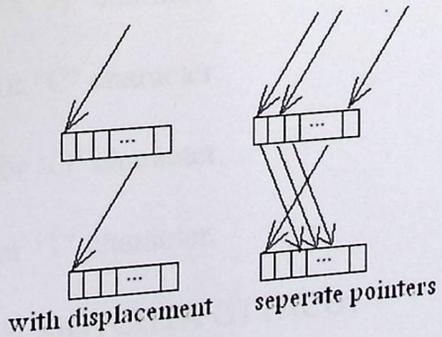


Figure 6.2: The use of displacement to reduce pointer size.

Consider using one pointer and a displacement value for the same level, this method will require less storage space than if we used a separate pointer for each node as shown from Figure 6.2.

Pointer size = pointers size + displacement size, Pointer size is related to the value of L.

Other work is to consider building tree of permutations, instead of using sliding window, for a specific W_x value using the tree structure shown at Figure 6.3.

Example: W=3, S= {AACT}, all permutations from the chars {A, C, G, T} are 4x4x4x4=256: AAA, AAC, CAA, AAG, AGA, AGA, GAA, AAT,… etc

First level is the root; second level contains one char from root, 3rd level start by prefix from, second level, 4th level start by prefix from previous level …etc. Repetitions is not allowed, it is unbalanced tree with depth = W-1 (for example = 3 for S={AACT}).

The cost for a new query = Window size (W_x) · branch_factor ……..(6.1)

And the database index size = (4^{Wx}) + pointers field size……..(6.2)

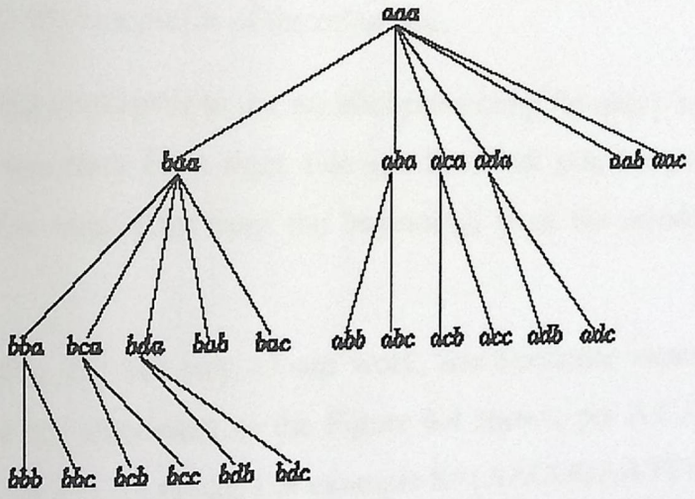


Figure 6.3: Radix Tree.

Also using WT with binary masks is another suggested work. While moving the window W_x over the sequences, uses binary masks for each chars:

"A" binary mask that put 1only for 'A' character.

"C" binary mask that put 1only for 'C' character.

"G" binary mask that put 1only for 'G' character.

"T" binary mask that put 1only for 'T' character.

Example: For W=16, for S = {AACGGTCGTA GTTACG}

Mask 1: "A" mask = "1100000001000100" =4+64+32768+65536

Mask 2: "C" mask = "0010001000000010" = 4+1024+16384

Mask 3: "G" mask = "0001100100100001"=

Mask 4: "T" mask = "0000010010011000"

Another interesting possibility is to use W_x values intersection. When searching database sequences using a query sequence, take different W_x values for the query.

Example:

S={GCCTGCAACGGTTACGATCTCC}, S length (n) =22

Pass over S by window W_x =4 from location '0' to location 'n/2', this will result by X candidates.

Pass over S by window W_x =8 for the chars from location n/2 to location n-1, this will result by Y candidates. Then consider take intersection between the candidates X and Y and do database reference for final check.

Note: Consider why using n/2 or another value and if one of the two the windows (i.e. the substring result from W_x =4 and W_x =8) is a prefix of the other one.

One particularly interesting concept is to use parallel processing for query searching when check for matching. Check query sequence from right side and from left side for prefix substring. Consider taking three substrings for matching, from the beginning, from the middle, and from the end of sequence.

For exact matching, during the last step of our work, use Sequence exact matching using curve. Consider using the curve for alignment as the Figure 6.4 shows, put A.C,G,T in 2-D axis by give weight to each char (A:1, B:2, C:-1,G:-2). For example S={AACGGAATTTTCGG}.

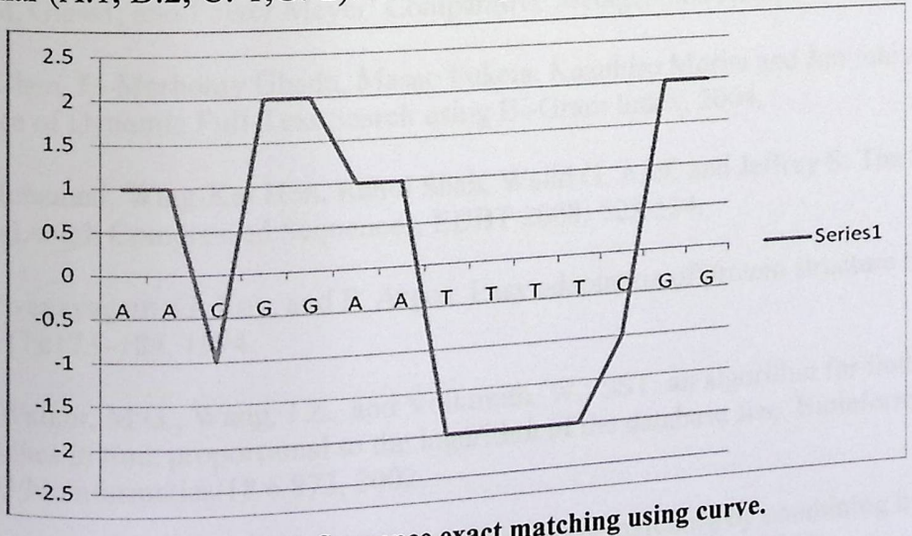


Figure 6.4: Sequence exact matching using curve.

Finally, the use of compression after WT transformation on *Wsum* values before filling the modified BTree index structure could lead further decrease of index size.

References

[1] Aleksandar Stojmirovic and Vladimir Pestov: Indexing Schemes for Similarity Search In Datasets of Short Protein Fragments, Journal Information Systems archive Volume 32 Issue 8, December, doi:10.1016/j.is.2007.03.001, 2007.

[2] Antognini Christian: Bloom Filters, 2008.

[3] Barroso, L. A.: Web search for a planet, The google cluster architecture. IEEE, (pages 1 and 28.), 2003.

[4] Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L Salzberg: Ultrafast and memory-efficient alignment of short reads to the human genome, doi:10.1186/gb-2009-10-3-r25, Genome Biology, 2009.

[5] Burton Howard Bloom, Bloom, "Space/time trade-offs in hash coding with allowable errors", Communications of the ACM 13 (7): 422–426, doi:10.1145/362686.362692, 1970.

[6] David W. Mount: Adapted from "Alignment of Pairs of Sequences," Chapter 3, in Bioinformatics: Sequence and Genome Analysis, 2nd edition, 2004.

[7] David A. White, R. Jain: Similarity indexing with the SS-tree, In Proceedings of the 12th ICDE Conference, pp. 516–523, 1996.

[8] DA Benson, I Karsch-Mizrachi, D J Lipman, J Ostell, and D L Wheeler: Genbank. Nucleic Acids Res, 34(Database issue):16-20, Jan 2006.

[9] Durian, B., J. Holub, H. Peltola and J. Tarhio: Improving practical exact string matching. Inf. Process. Lett, 110: 148-152, 2010, doi:10.1016/j.ipl.2009.11.010.

[10] Dominique Lavenier : Ordered Index Seed Algorithm for Intensive DNA Sequence Comparison, IEEE 2008.

[11] Donald E. Knuth: The Art of Computer Programming Volume 1: Fundamental Algorithms. Addison-Wesley Professional; 3rd edition (November 14, 1997).

[12] Di Wang, Guoren Wang, Qingquan Wu1, Baichen Chen: Finding LPRs in DNA Sequences Based on a New Index – SUA, 2005.

[13] Elizabeth M. Glass1, and Folker Meyer: Comparative Metagenome Analysis, 2010.

[14] El-Sayed Atlam, El-Marhomy Ghada, Masao Fuketa, Kazuhiro Morita and Jun-ichi Aoe: A Compact Memory Space of Dynamic Full-Text Search using Bi-Gram Index, 2004.

[15] Eltabakh Mohamed, Wing-Kai Hon, Rahul Shah, Walid G. Aref, and Jeffrey S: The SBCTree: An Index for RunLength Compressed Sequences, EDBT 2008: 523-534.

[16] Gareth Chelvanayagam, G. Roy, and P. Argos: Easy adaptation of protein structure to sequence Protein Eng., (7):173–184, 1994.

[17] Giladi, E., Walker, M.G., Wang, J.Z., and Volkmuth, W.: SST: an algorithm for finding near-exact sequence matches in time proportional to the logarithm of the database size. Bioinformatics. 18(6):873-7, doi:10.1093/bioinformatics/18.6.873, 2002.

[18] Gonzalo Navarro and M. Raffinot: Fast and flexible string matching by combining bit-parallelism and suffix automata, ACM Journal of Experimental Algorithmic (JEA) 5 (4), 2000.

[19] Hong Sun, Ozgur Ozturk, and Hakan Ferhatosmanoglu. Comri: A compressed multi-resolution index structure for sequence similarity queries. In Proceedings of the IEEE Computer Society Conference on Bioinformatics, page 553, 2003.

[20] Hon Wing-Kai, K. Sadakane and W.K. Sung: Breaking a Time-and-space barrier in constructing full-text indices. Proceeding of the 44th Annual Symposium on Foundations of Computer Science, Oct. 11-14, IEEE, pp: 251-260, doi: 10.1109/SFCS.2003.1238199, 2003.

[21] Hiroshi Yamamoto, Seishiro Ohmi, and Hiroshi Tsuji: Experimental Simulation on Incremental Three-gram Index for Two-gram Full-Text Search Systems, IEEE International Conference on Systems, Man and Cybernetics, vol. 5, pp. 4852-4857, Oct. 2003.

[22] Hugh E. Williams, Member, and Justin Zobel: Indexing and retrieval for genomic database, IEEE Computer Society, 2002.

[23] Inenaga, S. Hoshino, H. Shinohara, A. Takeda, M. S. Arikawa G. Mauri and G. Pavesi: On-line construction of symmetric compact directed acyclic word graphs. Discrete Appl. Math, 146: 156–179, 2001, doi:10.1016/j.dam.2004.04.012.

[24] In-Seon Jeong, Kyoung-Wook Park, Seung-Ho Kang, and Hyeong-Seok Lim: An efficient similarity search based on indexing in large DNA databases. Computational Biology and Chemistry 34(2): 131-136, 2010.

[25] Jiangtao Cui, Shuisheng Zhou, and Junding Sun: Efficient high-dimensional indexing by sorting principal component. Pattern Recognition Letters 28(16): 2412-2418, 2007.

[26] Jozef Misutka, and Leo Galambos: Mathematical Extension of Full Text Search Engine, Towards Digital Mathematics Library, 2008.

[27] Johanna Wenny Rahayu and David Taniar: Parallel Selection Query Processing Involving Index in Parallel Database Systems, IEEE, 2002.

[28] Jun Liang, Lin Xiao, and Di Zhang: An Efficient Approach for Building Compressed Full-text Index for structured Data, IEEE, 2009.

[29] Justin Zobe Rmit, Aistair Moffat, and Kotagiri Ramamohanarao: Inverted Files Versus Signature Files for Text Indexing, Journal ACM Transactions on Database Systems (TODS) TODS Homepage archive Volume 23 Issue 4, Dec. 1998.

[30] Kahveci, Ambuj K. Singh, Aliekber Gürel: An Efficient Index Structure for Shift and Scale Invariant Search of Multi-Attribute Time Sequences. ICDE 2002: 266.

[31] Kahveci Tamer: MAP: Searching Large Genome Databases, A. Singh Pacific Symposium on Biocomputing 8:303-314, 2003.

[32] Khan Mohammed Azharul Hasan: An Index Structure for Large Order Database Maintenance Using Variants of B Trees. Information Technology Journal, 7: 1061-1066, doi: 10.3923/itj.2008.1061.1066, 2008.

[33] Kunihiko Sadakane: Compressed suffix trees with full functionality, Theory Computer. System. 41 (4), 589–607, 2007.

[34] Kurt Stockinger, K. Wu, and A. Shoshani: Evaluation strategies for bitmap indices with binning, in: Proceedings of International Conference on Database and Expert Systems Applications (DEXA), vol. 3180, Springer, 2004.

[35] Lecroq Thierry, Fast exact string matching algorithms, Inf. Process. Lett. 102(6): 229-235, 2007.[31]

[36] Luis Gravano, Panagiotis G. Ipeirotis, H. V. Jagadish, Nick Koudas S. Muthukrishnan and Divesh Srivastava: Approximate String Joins in a Database (Almost) for Free, VLDB: 491-500, 2001.

[37] Luis Russo, G. Navarro and A. Oliveira: Fully-compressed suffix trees, in: Proc. of the Latin American Theoretical Informatics (LATIN), in: Lecture Notes in Computer Science, vol. 4957, Springer, 2008.

[38] Mahmoud Mhashi, Adnan Rawashdeh and Awni Hammouri, A Fast Approximate String Searching Algorithm, doi: 10.3844/jcssp.405.412, 2005.

[39] Mansour Nashat, Ramzi A. Haraty, Walid Daher, and Manal Houri: An auto-indexing method for Arabic text, Journal Information Processing and Management: an International Journal, Volume 44 Issue 4, July, 2008.

[40] Marina Barsky, Ulrike Stege, and Alex Thomo: Suffix trees for inputs larger than main memory. Inf. Syst. 36(3): 644-654, 2011.

[41] McMurry, J. Begley T.: The organic chemistry of biological pathways. Roberts & Company. ISBN 9780974707716, 2005.

[42] Michal Stabno and Robert Wrembel: RLH: Bitmap compression technique based on run-length and Huffman encoding. Inf. Syst. 34(4-5): 400-414, 2009.

[43] Mikaël Salson T. Lecroqa, M. Léonarda, and L. Mouchard: Dynamic extended suffix arrays, Journal of Discrete Algorithms Volume 8, Issue 2, Pages 241-257, June 2010.

[44] M. Taskin and Z. M. Ozsoyoglu: Improvements in distance-based indexing. SSDBM, pages 161–170, 2004.

[45] Murat Tasan, Z. Meral Ozsoyoglu: Improvements in Distance-Based Indexing, Proceeding SSDBM '04 Proceedings of the 16th International Conference on Scientific and Statistical Database Management IEEE, 2004.

[46] Nasser Yazdani and Hossein Mohammadi: DMP-tree: A dynamic M-way prefix tree data structure for strings matching. Computers & Electrical Engineering 36(5): 818-834, 2010.

[47] Norio Katayama and S. Satoh: The SR-tree: an index structure for high-dimensional nearest neighbor queries, in: SIGMOD, pp. 369–380, 1997.

[48] Nur'Aini Abdul Rashid', Rosni Abdullah, Abdullah Zawawi Haji Talib, and Zalila Ali: Fast Dynamic Programming Based Sequence Alignment Algorithm, IEEE, 2006.

[49] Oehmen, C.; Nieplocha, J. "ScalaBLAST: A scalable implementation of BLAST for high-performance data-intensive bioinformatics analysis". IEEE Transactions on Parallel & Distributed Systems, August 2006.

[50] Ozgur Ozturk, Hakan Ferhat, and Osman Oglu : Effective Indexing and Filtering for Similarity Search in Large Biosequence Databases, pp.359, Third IEEE Symposium on BioInformatics and BioEngineering (BIBE'03), 2003.

[51] Ozgur Ozturk and Hakan Ferhatosmanoglu: Vector Space Indexing for Biosequence Similarity Searches, IJAIT, International Journal on Artificial Intelligence Tools, 14-5, 2005.

[52] Patrick O'Neil and D. Quass: Improved query performance with variant indexes, in: Proceedings of ACM SIGMOD International Conference on Management of Data, 1997.

[53] Pokrzywa, R. and A. Polanski: BWtrs: A tool for searching for tandem repeats in DNA sequences based on the Burrows-Wheeler transform. Genomics, 96: 316-321, doi:10.1016/j.ygeno.2010.08.001, 2010.

[54] Punitha, and D.S. Gurub: Symbolic image indexing and retrieval by spatial similarity: An approach based on B-tree, Pattern Recognition 41(6): 2068-2085, 2007.

[55] Rob Williams: Computer Systems Architecture, Addison Wesley, Edition 2, 2006.

[56] Richard McCreadie, Richard and Macdonald, Craig and Ounis, Iadh: MapReduce indexing strategies: Studying scalability and efficiency, Information Processing and Management, doi:10.1016/j.ipm.2010.12.003, 2010.

[57] Sankoff D: Matching sequences under deletion/insertion constraints". Proceedings of the National Academy of Sciences of the USA, 1972.

[58] Sadeghipour Alireza Aghili, Ozgur D. Sahin, Divyakant Agrawal, Amr El Abbadi: Efficient Filtration of Sequence Similarity Search Through Singular Value Decomposition, IEEE, 2004.

[59] Smith, T.F. and M.S. Waterman: Identification of common molecular subsequences. Journal of Molecular Biology, 147(1):i195-i197, doi:10.1016/0022-2836(81)90087-5, 1981.

[60] Shimin Chen, Phillip B. Gibbonsy, Todd C. Mowry, and Gary Valentinx: Fractal Perfecting B+Trees: Optimizing Both Cache and Disk Performance, SIGMOD '02 Proceedings of the 2002 ACM SIGMOD international conference on Management of data, 2002.

[61] Stephen Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman: Basic local alignment search tool. Journal of Molecular Biology, 215(3):403–410, 1990.

[62] Sundaresan Muthukrishnan and S. C. Sahinalp: Approximate nearest neighbor and sequence comparison with block operations, STOC '00 Proceedings of the thirty-second annual ACM symposium on Theory of computing, 2000.

[63] Sun Wei and Da-xin Lui: A Hybird Method for Efficient Indexing of XML Documents, IEEE, 2005.

[64] Su Jian, Weng Wenyong, and Wang Zebing: An Ontology Enhanced Development Kit for Full Text Search, IEEE, 2009.

[65] Sun Wu and U. Manber: A fast algorithm for multi-pattern searching, Report TR-94-17, 1994.

[66] Tan Apaydin, G. Canahuate, H. Ferhatosmanoglu, and A.S. Tosun: Approximate encoding for direct access and query processing over compressed bitmaps, in: Proceedings of International Conference on Very Large Data Bases (VLDB), 2006.

[67] Tavakoli, N. and H. Modaress-Razavi: An architecture for parallel search of large, full-text databases. Proceedings of the International Conference on Databases, Parallel Architectures and Their Applications, PP: 342-349. doi: 10.1109/PARBSE.1990.77159, 1990.

[68] Thomas H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein: Introduction to Algorithms, 3nd ed., MIT Press, 2009.

[69] Ye-In Chang, Wei-Horng Yeh, Jiun-Rung Chen, and Jen-Wei Hu: An ACGT-Words Tree for Efficient Data Access in Genomic Databases. CIBCB 2007: 143-150.

[70] Yung-Hsing Peng, Chang-Biau Yang, Chiou-Ting Tseng, Chiou-Yi Hor: The indexing for one-dimensional proportionally-scaled strings. Inf. Process. Lett. 111(7): 318-322, 2011.

[71] Udi Manber, G. Myers, Suffix arrays: a new method for on-line string searches, in: Proc. of Symposium on Discrete Algorithms (SODA), 1990.

[72] Veli Makinen: Compact suffix array – a space efficient full-text index, Fundamental Informaticae 56 (1–2), 191–210 (Special Issue – Computing Patterns in Strings), 2003.

[73] Vladimir I. Levenshtein: Binary codes capable of correcting deletions, insertions and reversals. Soviet Physics-Doklady, 10:707–710, 1966.

[74] William A. Pearson, R. Doolittle Academic Press: Rapid and sensitive sequence comparison with fastap and fasta. In Methods in Enzymology ed., San Diego 183, pages 63– 98, 1990.

[75] Wong, S.S., W.K. Sung and L. Wong. CPS-tree: A compact partitioned suffix tree for disk-based indexing on large genome sequences. Preceding of the 23rd International Conference on Data Engineering, April 15-20, Istanbul, pp: 1350-1354. DOI: 10.1109/ICDE.2007.369009, 2007.

[76] Xiaozhu Liu: Efficient Maintenance Schema of Inverted Index for Large-scale Full-Text Retrieval, State Key Lab of Software Engineering. IEEE, 2010.

[77] Xiaohui Yu, Junfeng Dong: Indexing high-dimensional data for main-memory similarity search. Inf. Syst. 35(7): 825-843, 2010.

[78] Zhenjie Zhang, Marios Hadjieleftheriou, Beng Chin Ooi1, and Divesh Srivastava: Bed-Tree: An All-Purpose Index Structure for String Similarity Search Based on Edit Distance, 2010.

[79] Zhenqiang Tan, Xia Cao, Beng Chin Ooi, and Anthony K. H. Tung: The ed-tree: An Index for Large DNA Sequence Databases. SSDBM 2003: 151-160.

[80] Ziv Jacob and Abraham Lempel: Compression of Individual Sequences Via Variable-Rate Coding, IEEE Transactions on Information Theory, September 1978.

Appendices

Appendix A. Tree number of nodes [Section 4.4]

Bellow is the derivation for the number of nodes at a tree. For a tree of level (L) and the number of sub-nodes is N,

Llwith N sub nodes	Number of nodes
	1 : 1
1	4 : 1+N
2	13:1+N+N^2
3	40:1+N+N^2+N^3
4	…
…	…: 1+N+N^2+ … +N^(L-1)
L	

Total number of nodes = $\frac{[N^L - 1]}{N - 1}$, this can be proved by suppose this is valid and check it vice versa:

$$1+N^1+N^2+\ldots+N^{L-1} = \frac{[N^L - 1]}{N - 1} \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (1)$$

multiply both sides with $(N-1)$

$$(N-1)(1+N^1+N^2+\ldots+N^{L-1}) = [N^L - 1]$$

then,

$$N^1+N^2+\ldots+N^L - 1 - N^1 - N^2 - \ldots - N^{L-1} = N^L - 1 \ldots\ldots\ldots (2)$$

delete terms from N^1 *to* N^{L-1} *from left side ,what left is* :

$$-1+N^L = N^L - 1$$

This means that the suggestion is valid and total number of nodes = $\frac{[N^L - 1]}{N - 1}$

Appendix B. Equation 4.18, Equation 4.19, and Equation 4.20 results [Section 4.5.1]

kx	eq20	eq18	eq20-eq18	Log(Eq 20)	Log(Eq18)	Log(20-18)	Log(Eq19)	LOG(difference 18-19)
4	129591	2.14541E+11	-2.1454E+11	5.112574841	11.3315094	#NUM!	11.07394917	10.9821697
8	7.97195E+11	2.02494E+11	5.94701E+11	11.90156453	11.30641162	11.7742988	11.02897301	10.98043475
12	4.98246E+18	1.98478E+11	4.98246E+18	18.69744397	11.2977127	18.69744395	11.01332888	10.97937297
16	3.11404E+25	1.9647E+11	3.11404E+25	25.49332399	11.29329701	25.49332399	11.00542975	10.97869284
20	1.94627E+32	1.95266E+11	1.94627E+32	32.28920401	11.29062589	32.28920401	11.00068218	10.97821917
24	1.21642E+39	1.94463E+11	1.21642E+39	39.08508402	11.28883597	39.08508402	10.99752086	10.97786878
28	7.60263E+45	1.93889E+11	7.60263E+45	45.88096404	11.28755292	45.88096404	10.99526811	10.97759801
32	4.75165E+52	1.93459E+11	4.75165E+52	52.67684406	11.28658814	52.67684406	10.99358339	10.9773818
36	2.96978E+59	1.93124E+11	2.96978E+59	59.47272408	11.28583627	59.47272408	10.99227706	10.97720472
40	1.85611E+66	1.92856E+11	1.85611E+66	66.26860409	11.28523384	66.26860409	10.99123524	10.97705673
44	1.16007E+73	1.92637E+11	1.16007E+73	73.06448411	11.28474031	73.06448411	10.99038546	10.97693098
48	7.25044E+79	1.92455E+11	7.25044E+79	79.86036413	11.28432862	79.86036413	10.98967943	10.97682267
52	4.53152E+86	1.923E+11	4.53152E+86	86.65624415	11.28397995	86.65624415	10.98908375	10.97672828
56	2.8322E+93	1.92168E+11	2.8322E+93	93.45212416	11.28368087	93.45212416	10.98857461	10.97664522
60	1.7701E+100	1.92053E+11	1.7701E+100	100.2480042	11.2834215	100.2480042	10.98813455	10.9765715
64	1.1063E+107	1.91953E+11	1.1063E+107	107.0438842	11.28319443	107.0438842	10.98775051	10.97650557
68	6.9146E+113	1.91864E+11	6.9146E+113	113.8397642	11.28299397	113.8397642	10.9874125	10.97644622
72	4.3216E+120	1.91785E+11	4.3216E+120	120.6356442	11.28281571	120.6356442	10.98711278	10.97639249
76	2.701E+127	1.91715E+11	2.701E+127	127.4315242	11.28265615	127.4315242	10.98684524	10.97634358
80	1.6881E+134	1.91652E+11	1.6881E+134	134.2274043	11.28251249	134.2274043	10.98660499	10.97629886
84	1.0551E+141	1.91594E+11	1.0551E+141	141.0232843	11.28238248	141.0232843	10.9863881	10.97625779
88	6.5942E+147	1.91542E+11	6.5942E+147	147.8191643	11.28226425	147.8191643	10.98619135	10.97621992
92	4.1214E+154	1.91494E+11	4.1214E+154	154.6150443	11.28215627	154.6150443	10.98601207	10.97618489
96	2.5759E+161	1.91451E+11	2.5759E+161	161.4109243	11.28205727	161.4109243	10.98584806	10.97615238
100	1.6099E+168	1.91411E+11	1.6099E+168	168.2068044	11.28196617	168.2068044	10.98569745	10.97612211
104	1.0062E+175	1.91374E+11	1.0062E+175	175.0026844	11.28188206	175.0026844	10.98555869	10.97609386
108	6.2888E+181	1.91339E+11	6.2888E+181	181.7985644	11.28180416	181.7985644	10.98543045	10.97606742
112	3.9305E+188	1.91307E+11	3.9305E+188	188.5944444	11.28173182	188.5944444	10.98531157	10.97604262
116	2.4565E+195	1.91278E+11	2.4565E+195	195.3903244	11.28166445	195.3903244	10.98520108	10.97601931
120	1.5353E+202	1.9125E+11	1.5353E+202	202.1862044	11.28160157	202.1862044	10.98509812	10.97599734
124	9.5959E+208	1.91224E+11	9.5959E+208	208.9820845	11.28154273	208.9820845	10.98500197	10.97597662
128	5.9974E+215	1.912E+11	5.9974E+215	215.7779645	11.28148757	215.7779645	10.98491196	10.97595702
132	3.7484E+222	1.91177E+11	3.7484E+222	222.5738445	11.28143574	222.5738445	10.98482755	10.97593846
136	2.3427E+229	1.91156E+11	2.3427E+229	229.3697245	11.28138695	229.3697245	10.98474821	10.97592085
140	1.4642E+236	1.91135E+11	1.4642E+236	236.1656045	11.28134095	236.1656045	10.98467352	10.97590412
144	9.1513E+242	1.91116E+11	9.1513E+242	242.9614845	11.2812975	242.9614845	10.98460308	10.97588821
148	5.7196E+249	1.91098E+11	5.7196E+249	249.7573646	11.28125639	249.7573646	10.98453654	10.97587305
152	3.5747E+256	1.91081E+11	3.5747E+256	256.5532446	11.28121745	256.5532446	10.98447359	10.97585859
156	2.2342E+263	1.91065E+11	2.2342E+263	263.3491246	11.28118049	263.3491246	10.98441394	10.97584478
160	1.3964E+270	1.91049E+11	1.3964E+270	270.1450046	11.28114538	270.1450046	10.98435736	10.97583158
164	8.7274E+276	1.91035E+11	8.7274E+276	276.9408846	11.28111199	276.9408846	10.9843036	10.97581894
168	5.4546E+283	1.91021E+11	5.4546E+283	283.7367646	11.28108018	283.7367646	10.98425246	10.97580684
172	3.4091E+290	1.91007E+11	3.4091E+290	290.5326447	11.28104984	290.5326447	10.98420377	10.97579523
176	2.1307E+297	1.90995E+11	2.1307E+297	297.3285247	11.28102089	297.3285247	10.98415734	10.97578408
180	1.3317E+304	1.90982E+11	1.3317E+304	304.1244047	11.28099322	304.1244047	10.98411303	10.97577337

Appendix C. RDMS index types for SQL Server 2008, MYSQL 5.0, and Oracle 10g [Section 2.1.6.5]

Bellow tables list index types for the following RDMS: MSSQL server 2005, MYSQL, and Oracle.

1. SQL Server 2008*

Index type	Description
Clustered	A clustered index sorts and stores the data rows of the table or view in order based on the clustered index key. The clustered index is implemented as a B-tree index structure that supports fast retrieval of the rows, based on their clustered index key values.
Nonclustered	A nonclustered index can be defined on a table or view with a clustered index or on a heap. Each index row in the nonclustered index contains the nonclustered key value and a row locator. This locator points to the data row in the clustered index or heap having the key value. The rows in the index are stored in the order of the index key values, but the data rows are not guaranteed to be in any particular order unless a clustered index is created on the table.
Unique	A unique index ensures that the index key contains no duplicate values and therefore every row in the table or view is in some way unique. Both clustered and nonclustered indexes can be unique.
Index with included columns	A nonclustered index that is extended to include nonkey columns in addition to the key columns.
Full-text	A special type of token-based functional index that is built and maintained by the Microsoft Full-Text Engine for SQL Server. It provides efficient support for sophisticated word searches in character string data.
Spatial	A spatial index provides the ability to perform certain operations more efficiently on spatial objects (spatial data) in a column of the geometry data type. The spatial index reduces the number of objects on which relatively costly spatial operations need to be applied.
Filtered	An optimized nonclustered index, especially suited to cover queries that select from a well-defined subset of data. It uses a filter predicate to index a portion of rows in the table. A well-designed filtered index can improve query performance, reduce index maintenance costs, and reduce index storage costs compared with full-table indexes.
XML	A shredded, and persisted, representation of the XML binary large objects (BLOBs) in the xml data type column.

2. MYSQL5.0†

Index type	Description
Primary Key Indexes	is an index implies that all the columns are UNIQUE, all columns in the key must be defined as NOT NULL, a table can only have one PRIMARY KEY
Unique Key Indexes	All values at the field are unique, duplicated value is not allowed.

*Microsoft MSDN, Types of indexes, http://msdn.microsoft.com/en-us/library/ms175049.aspx, 2011.

†Mr. Javed Bhatti, How many types of indexes or keys are in MySQL?, http://www.mysqlfaqs.net/mysql-faqs/Indexes/How-many-types-of-indexes-or-keys-are-in-MySQL, 2011.

Normal Indexes also known as "Non-Unique Indexes",	Normal non-unique index. Non-distinct values for the index are allowed, so the index may contain rows with identical values in all columns of the index. These indexes don't enforce any structure on your data so they are used only for speeding up queries
Full-Text Indexes	It is a spetial type of index used with MyISAM tables, and can be created only for CHAR, VARCHAR, or TEXT field types., used mainly for A boolean search, natural language search, and query expansion search

3. Oracle, oracle 10G Release 2*

Index type	Description
Normal indexes	By default, Oracle Database creates B-tree indexes.
Bitmap indexes	which store rowids associated with a key value as a bitmap
Partitioned indexes	Which consist of partitions containing an entry for each value that appears in the indexed column(s) of the table
Function-based indexes	Which are based on expressions. They enable you to construct queries that evaluate the value returned by an expression, which in turn may include built-in or user-defined functions.
Domain indexes	Which are instances of an application-specific index of type index type"

* Managing Indexes, http://download.oracle.com/docs/cd/B19306_01/server.102/b14196/schema003.htm.

Appendix D. Optimization in GCC [Section 5.4,]*

In this article, we explore the optimization levels provided by the GCC compiler toolchain, including the specific optimizations provided in each. We also identify optimizations that require explicit specifications, including some with architecture dependencies. This discussion focuses on the 3.2.2 version of gcc (released February 2003) but it also applies to the current release, 3.3.2.

Levels of Optimization:

Let's first look at how GCC categorizes optimizations and how a developer can control which are used and, sometimes more important, which are not. A large variety of optimizations are provided by GCC. Most are categorized into one of three levels, but some are provided at multiple levels. Some optimizations reduce the size of the resulting machine code, while others try to create code that is faster, potentially increasing its size. For completeness, the default optimization level is zero, which provides no optimization at all. This can be explicitly specified with option -O or -O0.

Level 1 (-O1)

The purpose of the first level of optimization is to produce an optimized image in a short amount of time. These optimizations typically don't require significant amounts of compile time to complete. Level 1 also has two sometimes conflicting goals. These goals are to reduce the size of the compiled code while increasing its performance. The set of optimizations provided in -O1 support these goals, in most cases. The first level of optimization is enabled as:

```
gcc -O1 -o test test.c
```

Any optimization can be enabled outside of any level simply by specifying its name with the -f prefix, as:

```
gcc -fdefer-pop -o test test.c
```

We also could enable level 1 optimization and then disable any particular optimization using the -fno- prefix, like this:

```
gcc -O1 -fno-defer-pop -o test test.c
```

This command would enable the first level of optimization and then specifically disable the defer-pop optimization.

Level 2 (-O2)

The second level of optimization performs all other supported optimizations within the given architecture that do not involve a space-speed trade-off, a balance between the two objectives. For example, loop unrolling and function inlining, which have the effect of increasing code size while also potentially making the code faster, are not performed. The second level is enabled as:

```
gcc -O2 -o test test.c
```

---

* M. Tim Jones, Optimization in GCC, http://www.linuxjournal.com/article/7269, Jan 26, 2005.

Table 1 shows the level -O2 optimizations. The level -O2 optimizations include all of the -O1 optimizations, plus a large number of others.

Level 2.5 (-Os)

The special optimization level (-Os or size) enables all -O2 optimizations that do not increase code size; it puts the emphasis on size over speed. This includes all second-level optimizations, except for the alignment optimizations. The alignment optimizations skip space to align functions, loops, jumps and labels to an address that is a multiple of a power of two, in an architecture-dependent manner. Skipping to these boundaries can increase performance as well as the size of the resulting code and data spaces; therefore, these particular optimizations are disabled. The size optimization level is enabled as:

```
gcc -Os -o test test.c
```

In gcc 3.2.2, reorder-blocks is enabled at -Os, but in gcc 3.3.2 reorder-blocks is disabled.

Level 3 (-O3)

The third and highest level enables even more optimizations (Table 1) by putting emphasis on speed over size. This includes optimizations enabled at -O2 and rename-register. The optimization inline-functions also is enabled here, which can increase performance but also can drastically increase the size of the object, depending upon the functions that are inlined. The third level is enabled as:

```
gcc -O3 -o test test.c
```

Although -O3 can produce fast code, the increase in the size of the image can have adverse effects on its speed. For example, if the size of the image exceeds the size of the available instruction cache, severe performance penalties can be observed. Therefore, it may be better simply to compile at -O2 to increase the chances that the image fits in the instruction cache.

Appendix E. Genome DNA Sequence formats [Chapter 6]

Format	Gap	Multi	Description
gcg gcg8	Yes	No	GCG 9.x and 10.x format with the format and sequence type identified on the first line of the file. GCG 8.x format where anything up to the first line containing ".." is considered as heading, and the remainder is sequence data.
embl em	Yes	No	EMBL entry format, including all the fields in the latest release format. The Staden package and others use EMBL or similar formats for sequence data.
swiss sw swissprot	Yes	No	SWISSPROT entry format, including all the fields in the latest release format.
nbrf pir	Yes	No	NBRF (PIR) format, as used in the PIR database sequence files. This format was used for some years as an interchange format with the reference data followed by the sequence data. This unofficial PIR format is what EMBOSS supports. If there is enough interest, we can also use NBRF database format with separate files for sequence (the main EMBOSS input/output) and for features. Documentation of this format is hard to find, but we do have a copy from PIR. The sequence files include the ID and description but no citation or feature information.
pdb	No	No	PDB protein databank format ATOM lines
pdbseq	No	No	PDB protein databank format SEQRES lines
pdbnuc	No	No	PDB protein databank format nucleotide ATOM lines
pdbnucseq	No	No	PDB protein databank format nucleotide SEQRES lines
fasta ncbi	Yes	No	FASTA format with optional accession number and database name in NCBI style included as part of the sequence identifier. eg >database | accession | id description or >name description or >name accession description
gifasta	Yes	No	FASTA format including NCBI-style GIs (alias)
pearson	Yes	No	FASTA format with no further processing of the "ID" eg: >name description Used where fasta or ncbi format interprets the ID in an unwanted way, this format skips the further ID parsing stage of reading these files.
fastq	No	No	FASTQ short read format ignoring quality scores
fastq-sanger	No	No	FASTQ short read format with phred quality
fastq-illumina	No	No	FASTQ Illumina 1.3 short read format

fastq-solexa	No	No	FASTQ Solexa/Illumina 1.0 short read format
genbank gb ddbj	Yes	No	GENBANK entry format, including the feature table..
refseqp	Yes	No	Refseq protein entry format
genpept	Yes	No	Refseq protein entry format (alias)
codata	Yes	No	Codata entry format
strider	Yes	No	DNA strider output format
clustal aln	Yes	No	ClustalW ALN (multiple alignment) format.
phylip	Yes	Yes	Phylip interleaved and non-interleaved formats
phylipnon	Yes	Yes	Phylip non-interleaved format
	Yes	No	ACEDB sequence format
dbid	Yes	No	FASTA format variant with Database name first, then ID name then an optional accession number eg: >database name description or >database name accession description
msf	Yes	No	Wisconsin Package GCG MSF (mutiple sequence file) file format
hennig86	Yes	No	Hennig86 output format
jackknifer	Yes	No	Jackknifer interleaved and non-interleaved formats
nexus paup	Yes	No	Nexus/paup interleaved format
treecon	Yes	No	Treecon output format
mega	Yes	No	Mega interleaved and non-interleaved formats
igstrict	Yes	No	Intelligenetics sequence format strict parser
ig	Yes	No	Intelligenetics sequence format
staden	Yes	No	This format is actually obsolete, the latest version of the Staden package does not support it anymore (see "experiment" format for the new Staden package format). Staden format was a just the sequence in simple text with, optionally, comments at any position in the sequence. When EMBOSS reads in "staden" format, it recognizes a comment at the top of the sequence as the sequence dientifier and removes any comments inside the sequence. Some alternative nucleotide ambiguity codes are used and should be converted.
text plain	Yes	No	Plain text. This is the format with no format. The whole of the file is read in as a sequence. No attempt is made to parse the file contents in any way. Anything is acceptable in this format. This means that any character will be included in the sequence, even digits and punctuation. Use this format only when you are sure that the input sequence file is correct

			and contains only what you want to be considered as your 'sequence'.
gff2	Yes	No	GFF feature file with sequence in the header Normally used as a pure feature format, but can hold the sequence as part of the structured header.
gff3 gff	Yes	No	GFF3 feature file with sequence
stockholm pfam	Yes	No	Stockholm (pfam) format
selex	Yes	No	SELEX format is used by Sean Eddy's HMMER package. It can store RNA secondary structure as part of the sequence annotation.
fitch	Yes	No	Fitch program format
mase	Yes	No	Mase program format
raw	No	No	Like text/plain format except that it removes any whitespace or digits, accepts only alphabetic characters and rejects anything else. This means that it is safer to use this format than plain format. If you have digits and spaces or TAB characters, these are removed and ignored. If you have other non-alphabetic characters (for example, punctuation characters), then the sequence will be rejected as erroneous. Gap characters, '-', and translated STOP codon characters '*' are legal.
experiment	Yes	No	The Staden package stores single sequencing experiment reads in a format derived from EMBL. All EMBL tags are allowed, plus many extras. Unusually, the extra tags are allowed to continue beyond the '//' line which only marks the end of the sequence. The "EX" experiment line is used to create a sequence description. Accuracy values are stored, or at least the largest value for each sequence position. To date no EMBOSS program is using these values.
abi	Yes	No	ABI trace file format. This is the format of file produced by ABI sequencing machines. It contains the 'trace data' i.e. the probabilities of the 4 bases along the sequencing run, together with the sequence, as deduced from that data. The sequence information is what is normally read in and used by EMBOSS programs, although the trace data is available and may be utilised by some specialised EMBOSS programs. The code for this is heavily based on David Mathog's fortran library with a description of ABI trace file format (abi.txt): ftp://saf.bio.caltech.edu/pub/software/molbio/abitools.zip

Appendix F. T1 and T2 analysis [Section 4.3]

R1	R2	X	r	SUM	T1	T2	(T2-T1)/T2
10000	12000	16	32	12000	544000	576000	0.055555556
11000	13000	16	32	13000	592000	624000	0.051282051
12000	14000	16	32	14000	640000	672000	0.047619048
13000	15000	16	32	15000	688000	720000	0.044444444
14000	16000	16	32	16000	736000	768000	0.041666667
15000	17000	16	32	17000	784000	816000	0.039215686
16000	18000	16	32	18000	832000	864000	0.037037037
17000	19000	16	32	19000	880000	912000	0.035087719
18000	20000	16	32	20000	928000	960000	0.033333333
19000	21000	16	32	21000	976000	1008000	0.031746032
20000	22000	16	32	22000	1024000	1056000	0.03030303
21000	23000	16	32	23000	1072000	1104000	0.028985507
22000	24000	16	32	24000	1120000	1152000	0.027777778
120000000	12000000000	16	32	12000000000	3.8592E+11	5.76E+11	0.33
120000000	12000000000	16	16	12000000000	1.9392E+11	3.84E+11	0.495

(1) Note: appendices D, E, and F are taken from references without any modifications