Indexing for Large DNA Database sequences

Samer Mahmoud Wohoush
Palestine Polytechnic University
Halhul, Palestine
samer_wh@yahoo.com

Mahmoud Hasan Saheb
Palestine Poletechnic University
Hebron, PO.Box 198, Palestine
alsaheb@ppu.edu

Abstract—**Bioinformatics data consists of a huge amount of information due to the large number of sequences, the very high sequences lengths and the daily new additions. This data need to be efficiently accessed for many needs. What makes one DNA data item distinct from another is its DNA sequence. DNA sequence consists of a combination of four characters which are A, C, G, T and have different lengths. Use a suitable representation of DNA sequences, and a suitable index structure to hold this representation at main memory will lead to have efficient processing by accessing the DNA sequences through indexing, and will reduce number of disk I/O accesses. I/O operations needed at the end, to avoid false hits, we reduce the number of candidate DNA sequences that need to be checked by pruning, so no need to search the whole database. We need to have a suitable index for searching DNA sequences efficiently, with suitable index size and searching time. The suitable selection of relation fields, where index is build upon has a big effect on index size and search time. Our experiments use the n-gram wavelet transformation upon one field and multi-fields index structure under the relational DBMS environment. Results show the need to consider index size and search time while using indexing carefully. Increasing window size decreases the amount of I/O reference. The use of a single field and multiple fields indexing is highly affected by window size value. Increasing window size value lead to better searching time with special type index using single filed indexing. While the search time is almost good and the same with most index types when using multiple field indexing. Storage space needed for RDMS indexing types are almost the same or greater than the actual data.**

Keywords-component; Large database, DNA sequence, index structure, sequence transformation, wavelet transformation, RDMS indexing.

I. INTRODUCTION (HEADING 1)

Dealing with string of characters for large database is not easy in term of space and access time. Genome databases as NCBI have a huge size because of the daily addition of new data. Electronic books and biological data are good examples for large databases that include text and sequences. For genome database we can consider DNA sequence as a key value that distinguishes a sequence from another.

Most of the work on genome database tries to find small size, efficient digit value that can represents DNA sequence. The problem of 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 need to be accessed.

We will consider the DNA sequence as the key value for genome database. Transforming this key to a digit is required to increase efficiency. Wavelet transformation technique [1] for DNA sequences is a suitable choice for our needs to do transformation as it gives us two advantages.

Firstly, it saves sequence order while considering amount of overlapping carefully. Secondly, transforming characters to digits depends on frequencies of characters.

Little amount of storage is needed as after finding first level wavelet coefficients[1], the second level can be calculated depending on the first level instead of referring to the original sequence again. Our evaluation use substring searching for matching identical pattern by sliding window, this will reduce candidate list of sequences need to be checked, and at the last stage refer to database disk for validations after pruning, in other words, optimization for the number of I/O operations.

Relational database provide different types of indexing like BTree, RTree[2], and hashing. Using these types of indexing to store Wavelet transformation will be discussed later.

The rest of the paper is organized as follows: section 2 reviews of related works. Section 3 presents data samples and methodology. Our results will be presented in section 4. Section 5 discusses the results. Conclusions will be discussed in section 6.

II. RELATED WORK

Different methods had been used to transform and index huge database systems. Dynamic programming [3, 4] has time and space complexity of O(nm) for two strings S and Q of lengths n and m, for database comparisons it will needs matrix of size nxm. Hence for long sequence and large database this method will be not practical in term of both space and time. It finds the difference between S and Q using a heavy computation method; the edit distance.

The use of r Binary masks [3] of size n, M1, M2,…,Mr to move through S, of size m, by word size of w has complexity of O(nmr/w). For large value of m, this complexity will be very close to dynamic programming.

Dictionary based indexing [3] for a database of sequences Si (i:1,2,…,n), creates index structure of size n corresponding to database size, predefining query lower bound length (L) to be equal to log(n) assumed. Query with larger length will be partitioned into smaller parts. All substrings of length L mapped to integers using hashing function and for queries

larger than L split it 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. Dictionary based index size is larger than the database.

BLAST technique [5] used to find local similarity [6] and not global similarity. It is a string matching tool that has two phases: search all database sequences for a fixed substring length w (between 3 and 11) for exact matching (at i). And 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 exceed 't'. It stores pointer for location 'i'. So, space needed is more than the database size.

Suffix array [7] scans database strings using a window (window size w, overlapping amount Δ) and count repetition of all possible k-tuples. It stores result at vector of size σ^k (σ referred to alphabet chars A, C, G, T). Then it indexes those vectors at hierarchical binary tree and to compare new query with those vectors it uses Edit distance method. It runs 25 to 50 times faster than BLAST. Disadvantage of this method is the allowing of false drops and index size increase linearly with k value.

The Multi-Resolution index Structure (MRS) [5, 8] uses a sliding window of size w. MRS seeks the result set in different resolution levels. However, the authors only focus on the cost of MRS, and do not evaluate the filtrations efficiency of their proposed technique.

SST [9, 10] scans the database by window w and map results to vector of size 4^w. Then hierarchical clusters, non overlapping, built using k-means algorithm, as any new query need to be processed against the database, using cluster mean and neglect clusters that are far away from the new query. Disadvantages of SST are the complexity of calculations, and false clustering.

The use of blocked inverted index [11] consists of index file (distinct terms) and a set of inverted lists with large-scale full-text system. This method solves two problems: The high storage overhead and considering posting list structure with differentiation between short and long lists. Through this work [11], blocked inverted index where used with skipping approach and propose the random access blocked inverted index (RABI) which enhance space and storage efficiency. This approach divide index into blocks and do compression to different parts of the block using encoding method. For compression it uses Binary interpolative coding (BIC). Access is done at both levels block level and inner block level.

Build self-index [10] for data records using stuffing of delimiters, and give an upper bound, limited by a number of bits, of permanent space in worst case. Analysis's done for space and time efficiency. Storage experiments compare the effects of using stuffing and performance examines three process construct, recover, and retrieval. Results show the effectiveness of FM-index in space and performance. This paper shows the advantages of using FM-index with the addition of adding delimiters.

Handle structural mathematical text and mathematical operation [12] by index real-world scientific documents containing mathematical notation based on full text searching. Mathematical indexing address the following issues, extraction and storing of mathematical notation, and ranking function.

Full text search [13, 14] can be done by different ways. One way is by using N-gram which means we take N characters each time we do processing. N characters processed for searching, by start with 2-gram index then supplement with higher-gram index. Frequently used search terms selected for the incremental index for that this approach have two functions, search engine and index creation engine. For long sequence number of AND operation is large which cause low performance for search, incremental indexing should solve this problem by carefully selecting search terms using search intensive approach. Experimental results show the effectiveness of incremental index even for further stored terms. This method build the incremental index upon subset of terms not for all terms to save space and provide efficiency for most search terms, while in our case we need to have index structure to be ready before searching.

Building suffix array needs time O(n) and space O(n) for constant size text and Suffix tree needs O(nlogn) time and O(n) space. Suffix array and tree are suitable for pattern searching. Another method uses Compressed Suffix Array [15] and the output array of Burrows-Wheeler Transformation (BWT). In this approach a new algorithm has been developed using terminators at the end of each word.

The use of signature of files for documents retrieval for large database systems allows the use of parallel hardware architecture [16, 17] for full text searching. Controlling false drop and using suitable hashing function with buffer and good storage overhead. The parallel process can be applied for process a document and between documents too. It provide a way for don't care characters.

An ontology kit for full text searching [18] 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.

Most of mentioned works try to build lower bound (D) for the edit distance (ED). Edit Distance is time and space consuming (O(|n|*|n|) time & space) for the whole string. In general we can see that for three strings s1,s3 and s3, if D(s1,s2) > D(s1,s3), then ED(s1,s2) > ED(s1,s3)).

III. DATA AND METHODOLOGY

Data used in our experiments shown in table I, we have picked 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 I.

TABLE I. DATA SAMPLES USED BY OUR EXPERIMENTS

Species kingdom	File name	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

A. Solution approach

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' of sizes 8, 16, 32, 64, 128, and 256 chars Fig 1 shows the conversion steps.

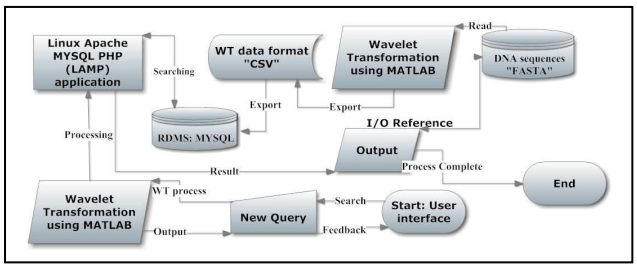

Figure 1. Schema chart shows transformation, building index structure, preprocessing new query, and comparison

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' 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' to be 2x. By this assumption after finding the first order wavelet transformation by scanning the database by window w1, we can find the wavelet transformation for window values wi for i>1 depending on previous window value (w1) and no need to scan the database again.

Part 1 algorithm:

Input:

Database of n sequences, n is a large value.

Each sequence will be donated by Si, i Є [1, .., n] with length Li.

Preprocessing:

We have different window sizes (Wx), x =1, …,5

Transform each Si into number format using Wavelet Transformation (WT), no need to

reference database again as WT for W2 can be calculated from W1 as: (A1,B1),(A2,B2) -> (A1+A2,A1-A2)

Initialize i=0

For each Wx value from (Wx min,…, Wx max){

Move Wx over Si to produce Si[i-Wx'] Calculate WT for each Si[i-Wx']

Wx' = Wx' +1 and i++

Output: set of subsequences (SSi,j), j Є[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, build data structure upon this index type.

Search by a query sequence:

Search for a new sequence NQ of length LNQ.

Convert NQ to WT to produce LNQ/Wx subsequences (NQi)

after moving Wx window over NQ.

Search the database for matching between NQi and SSi,j.

B. Wavelet coefficents

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 works [3]:

$$V(k,i) = (A(k,i), B(k,i)), \quad (1)$$

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

$$B_{k,i} = \begin{cases} 0 & k = 0 \\ A_{k-1,2i} - A_{k-1,2i+1} & 0 < k \le \log_2 n, \end{cases} \quad (3)$$

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

The sequences will be represented using six window sizes, wi for i= {1, 2, 3, 4, 5, 6}, Each window size 'wi' representation

will correspond to a final matrix for each sequence, this means we will have six matrices corresponds to wi value.

Second step: uploading 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 III 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 [19, 20] to store index in, webserver is Apache and language script used is PHP v5 for testing index reach, and Matlab version 7 used for wavelet transformation.

C. *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 indexing for XML [21].

"Normal" Indexes are the basic index type used by RDMS and require data field to be ordered, Normal Index have no restraints 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 (log n) 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 Logn / 2 K while a binary tree is Log 2 K, where Search k-key values are K1, K2, …, Kn –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 [22].

In Hash index, bucket reached by key using a hashing function. Records with different key values may map to same bucket; thus entire bucket has to be searched sequentially to locate record.

Bucket Overflows caused by insufficient buckets and distribution of records (Overflow chaining) Collision handling with O (1) complexity, for worst cases performance may deteriorates to O (n). An ideal hash function is uniform/Random and worst map to one bucket. Space complexity for formal Hash function is O (nlogn), where n: number of keys [22, 13]. 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.

IV. 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 II 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 III show the results of this approach.

TABLE II. EVALUATION OF SAMPLE DATA (UNDER SIX RESOLUTIONS W) USING DIFFERENT INDEX TYPES.

Index type	W1	W2	W3	W4	W5	W6
DEFAULT	0.002	0.029	0.233	0.677	0.961	1.153
Normal Index	0.010	0.167	1.380	4.05	5.891	6.350
PRIMARY	0.093	0.190	1.470	4.233	6.064	6.428
Fulltext	0.002 0.002 0.003	0.026 0.028 0.029	0.216 0.227 0.228	0.259 0.257 0.227	1.099 1.155 0.993	1.478 1.478 1.548
UNIQUE	0.009	0.172	1.406	4.077	5.931	6.357
Hash	0.010 0.010 0.011	0.164 0.167 0.170	1.363 1.376 1.378	4.059 4.13 4.252	5.920 6.010 6.117	6.282 6.363 6.386
BTree	0.010 0.010 0.010	0.166 0.166 0.166	1.366 1.379 2.383	4.092 4.054 4.249	5.914 6.043 6.088	6.266 6.336 6.410

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.

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 IV shows percentage number of returned references to the whole database size while changing window size.

Ratio = (sequences retrieved) / (total size of dataset)

TABLE III. APPLYING INDEXES FOR EIGHT COLUMNS SEARCH FIELDS.

Index type	W1	W2	W3	W4	W5	W6
DEFAULT ON pk	0.018	0.010	0.067	0.209	0.281	0.279
Normal Index	0.001	0.001	0.001	0.001	0.001	0.001
Hash	0.001	0.001	0.001	0.001	0.001	0.001
BTree	0.001	0.001	0.001	0.001	0.001	0.001

V. DISCUSSION

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.

TABLE IV. 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

On the other hand when we used multiple fields' index, we have got much better results as shown in table III. Hash, BTree, and normal index on the eight fields give better results when compared with a single field index by table III.

Multiple field index cause overhead for calculation and index address updates in case that amount of updates is high and overhead for write operations and disk referring. But compared amount of overhead with multiple indexes (merge index) case, this overhead is less. For DNA database, update operations is much less than insert operations and can be neglected. To reduce size of WTR, singular value decomposition (SVD) [23] as a preprocessing step before building index structure for the genome database can be used.

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

VI. INDEX SPACE COMPLEXITY

Table V shows the space complexity of using one column index with the following index types: full-text, primary index, 8 column index of types unique, primary, and normal index.

TABLE V. SPACE COMPLEXITY TABLE (B: BYTES, KB: KBYTES), RANGE VALUES STANDS FOR SPACE USED FOR DATA AND FOR INDEX.

W value	UNIQUE	Primary	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

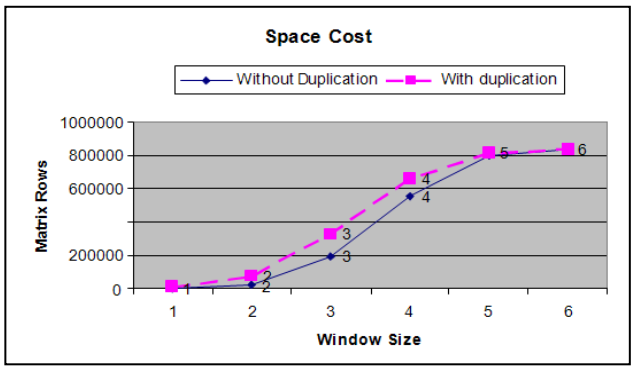

Figure 2. Space cost

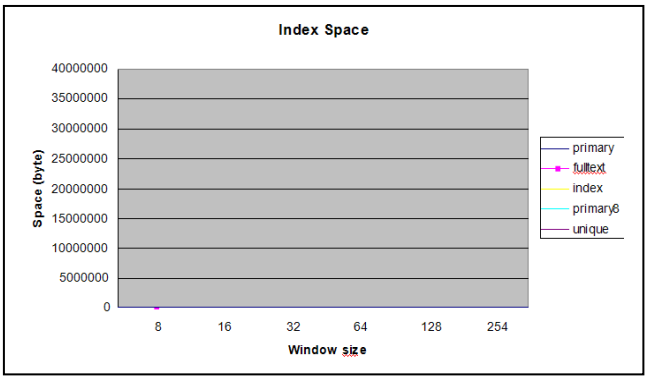

Figure 3. Space cost for all one field (primary, Full-text) and 8-column (index, primary, unique).

Tables VI and table VII show that space complexity variation while changing window size for different index types, it needs to be considered carefully. We can see that index size is larger than data size for all types, as seen from table VII where index size is low compared with data size.

TABLE VI. SPACE COMPLEXITY TABLE (B: BYTES, KB: KBYTES), RANGE VALUES STANDS FOR SPACE USED FOR DATA AND FOR INDEX.

W value	UNIQUE	Primary	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 VII. SPACE COMPLEXITY TABLE (B: BYTES, KB: KBYTES), RANGE VALUES STANDS FOR SPACE USED FOR DATA AND FOR INDEX.

W value	UNIQUE	Primary	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

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

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

When comparing time with size of one field index, we found that best time performance achieved by DEFAULT ON pk and 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. Lowest index size is primary and Full-text as shown by Fig. 3.

Our study shows that using multi-fields index improve performance over all types of indexing in spite of the type of index we used. 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.

CONCLUSION

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.

Space and time complexity shows 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 VI and VII. 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.

REFERENCES

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

[2] An efficient similarity search based on indexing in large DNA databases, In-Seon Jeong, Kyoung-Wook Park, Seung-Ho Kang, Hyeong-Seok Lim, 2010.

[3] An Efficient Index Structure for String Databases. Tamer Kahveci Ambuj K. Singh Department of Computer Science, University of California Santa Barbara, CA 93106 {amer,ambuj}cs.ucsb.edu, 2001.

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

[5] MAP: Searching Large Genome Databases. T. Kahveci, A. Singh Pacific Symposium on Biocomputing 8:303-314(2003).

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

[7] S. Muthukrishnan and S. C. Sahinalp. Approximate nearest neighbor and sequence comparison with block operations, 2000.

[8] CoMRI: A Compressed Multi-Resolution Index Structure for Sequence Similarity Queries. Hong Sun1, Ozgur Ozturk1, Hakan Ferhatosmano glu, IEEE, 2003.

[9] E. Giladi et al., SST: An Algorithm for Finding Near-Exact Sequence Matches in Time Proportional to the Logarithm of the Database Size. Bioinformatics 18, 873–877, 2002.

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

[11] Efficient Maintenance Schema of Inverted Index for Large-scale Full-Text Retrieval, Xiaozhu Liu, State Key Lab of Software Engineering Wuhan University Wuhan 430072, China , School of Automation Wuhan University of Technology IEEE, 2010.

[12] Mathematical Extension of Full Text Search Engine, Jozef Misutka, Leo Galambos, Department of Software Engineering, Charles University in Prague, Ke Karlovu 3, 121 16 Prague, Czech Republic, 2008.

[13] Experimental Simulation on Incremental Three-gram Index for Two-gram Full-Text Search Systems, Hiroshi Yamamoto Seishiro Ohmi Hiroshi Tsuji IEEE, 2003.

[14] A Compact Memory Space of Dynamic Full-Text Search using Bi-Gram Index, El-Sayed Atlam, El-Marhomy Ghada, Masao Fuketa, Kazuhiro Morita and Jun-ichi Aoe, Department of Information Science and Intelligent Systems, University of Tokushima Tokushima,770-8506, Japan 2004.

[15] Breaking a Time-and-Space Barrier in Constructing Full-Text Indices, Wing-Kai Hon, Kunihiko Sadakane_ Wing-Kin Sung IEEE, 2003.

[16] Parallel Selection Query Processing Involving Index in Parallel Database Systems. J. Wenny Rahayu David Taniar, IEEE, 2002.

[17] An Architecture for Parallel Search of Large, Full-text Databases, Nassrin Tavakoli and Hassan Modaress-Razavi, Department of Computer Science, The University of North Carolina at Charlotte, Charlotte, NC 28223 IEEE, 1990.

[18] An Ontology Enhanced Development Kit for Full Text Search, Su Jian, Weng Wenyong, Wang Zebing, Lab of Digital City & Electronic Service, Zhejiang University City College, Hangzhou 310015, China IEEE, 2009.

[19] Alexander Rubin, Senior Consultant, MySQL AB, Full Text Search in MySQL 5.1 New Features and HowTo, http://www.mysqlfulltextsearch.com/full_text.pdf, 2006.

[20] Moshe Shadmon, The ScaleDB Storage Engine, http://www.scaledb.com/pdfs/ScaleDB_MySQL_Preso2009.ppt, 2009.

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

[22] The SBCTree: An Index for RunLength Compressed Sequences, Mohamed Y. Eltabakh ,Wing-Kai Hon, Rahul Shah, Walid G. Aref, Jeffrey S. Vitter Purdue University, 2008, 2008.

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