Supporting High Performance Bioinformatics Flat-File Data Processing Using Indices

Xuan Zhang Gagan Agrawal
Department of Computer Science and Engineering
Ohio State University
Columbus, OH, 43220
{zhangx,agrawal}@cse.ohio-state.edu

Abstract

As an essential part of in vitro *analysis, biological database query has become more and more important in the research process. A few challenges that are specific to bioinformatics applications are data heterogeneity, large data volume and exponential data growth, constant appearance of new data types and data formats. We have developed an integration system that processes data in their flat file formats. Its advantages include the reduction of overhead and programming efforts. In the paper, we discuss the usage of indicing techniques on top of this flat file query system. Besides the advantage of processing flat files directly, the system also improves its performance and functionality by using indexes. Experiments based on real life queries are used to test the integration system.*

1 Introduction

Modern biological researches are often complicated processes and involve many steps. Among them, database query is essential for many applications. It can be used to, for example, identify research objects, collect relative information, and search existing literature.

The biological databases range from the classic molecular databases, such as GenBank, and literature databases, such as PubMed, to the latest array data repository, such as GEO, and network database, such as BIND. Because of the diversity of the data they represent and other considerations, these databases differ greatly in their choice of database management system (DBMS).

Most of the biological databases are open to public. They could be downloaded as flat files. One possible solution to overcome the DBMS heterogeneity problem is to use these flat files directly. As an alternative to loading data into a DBMS and utilizing its query facility, the data could be extracted and examined from its original files.

The biological queries are often simple selection queries on one or few attributes without many testing conditions. Such queries can be supported on flat-file datasets with simple implementations. However, with this approach, the size of the databases is a major factor that affects the system's response time. Biological data is growing at a phenomenal speed and good performance is desirable by the users.

978-1-4244-1694-3/08/$25.00 ©2008 IEEE

Indices have been widely used by database research community to improve query performance. A variety of indexing algorithms have been proposed. They allow random retrieval of data entries by using pre-processed summarization of the dataset. When the query selectivity is low, use of index could significantly reduce the I/O cost.

In recent years, there has been much work on developing indexing mechanisms suitable for biological data [10, 12, 18, 23, 27, 9]. Despite such developments, it has not been easy to incorporate indexing mechanisms into the biological data processing tasks. There are at least two reasons for this. First, as we mentioned above, many data processing tasks simply process flat-file data with utility functions. Second, biological data varies greatly, and as a result, a universal indexing scheme does not exist.

Thus, it is highly desirable to have a toolkit, which will allow indexing functions to be incorporated into biological data processing. This paper reports the design, implementation, and evaluation of such a toolkit. Previously, we had worked on systems and technologies [28, 30, 29] that provided database-like query interface on top of flat-file biological data. However, in these systems, all data was scanned and processed in a streaming fashion, making the query processing time linear in terms of the dataset size.

In our enhanced system, we rely on users to provide indexing functions specific to their applications. The role of our integration system is to serve as a general platform that evokes these functions when appropriate. The indices used by our system follow the general form, which is a pair of an index value and a pointer. Two types of indexing functions are needed. *Index generation functions* specify how indices can be built. *Index retrieving functions* implement how indices are used to answer a query. Together these two types of indexing functions provide an efficient way to access data entries even if they are embedded in flat files. The general interfaces of the indexing functions add great flexibility to the system's functionality. For example, in our experiments, when DNA sequences are indexed appropriately, sequence similarity search can be implemented as a selection query.

Overall, the data integration system is improved in both performance and functionality by using indices. It allows data embedded in the flat files to be queried in a database-like fashion. It aims to provide a balance between data accessibility and query performance.

The rest of the paper is organized as follows. We initially give an overview of the work on indexing biological data in Section 2. The overall system structure is illustrated in Section 3. The algorithm and implementation details are presented in Section 4. Experimental results are discussed in Section 5. We conclude in Section 6.

2 Indexing Biological Data

This section reviews the indexing algorithms for biological data.

Indexing is a technique that searches for data entries using reduced information of them. The basic index is a structured list of tuples. Each tuple contains one data attribute value and one pointer to the corresponding data entry. By comparing the attribute values and obtaining the pointers, desired entries can be retrieved in shorter period of time. Many advanced indexing algorithms have been developed.

Many biology databases [2, 25, 8] embrace the index technology to improve the query performance. Indices are widely used on the primary key attributes and other alphanumeric valued attributes. The increasing popularity of public life science literature databases [6], such as PubMed Central [1] and Medline [2], have attracted many efforts on literature index and retrieval. Most of the indices are word based and similar to traditional electronic dictionary [5, 17, 24]. Some approaches, such as [19], use probabilities and filters to refine the entry retrieval. Gene Reference Into Function (GeneRIF) [16] was developed and maintained by the National Library of Medicine (NLM) Gene Indexing initiative. It enriches the literature search by linking PubMed articles about the basic biology of a gene or protein within eight organisms to the LocusLink, a database of gene products. ISAID [3] was developed to help user create complex, precise, and accurate indexing for full-text documents semi-automatically. The performance test done by Boyce and Lockard [4] showed that the automatic indexing procedures that are based on the full text of medical articles are comparable to manual indexing.

Indexing on other types of biological data have been studied, too. Lowe et al [15] extend Pindex, a system associates word phrases with Medical Subject Heading (Mesh) terms, to index medical images based on their free-text description. Tagare et al discussed similarity search of medical images and application of numerical concepts to image indexing in [26]. Suffix arrays were used to identify the conserved RNA secondary structure motif without sequence alignment [1]. Sequence similarity search is central to biological research and indexing approaches have also been explored. Shibuya and Rigoutsos [20] search for gene candidates through the use of the Bio-dictionary, which is based on indexing the redundant and unique patterns derived from an existing sequence database. Singh et al have developed a set of index structures for sequence alignment [10, 12, 14, 13] and substring search [11]. Their approaches are based on wavelet transformation and grid index structures. Frequency bases efforts can also be found in [18, 23]. Several advanced tree structures [27, 9] have been proposed for sequence homology searches, too.

[1] Please see http://www.pubmedcentral.nih.gov/

[2] Please see http://medline.cos.com/

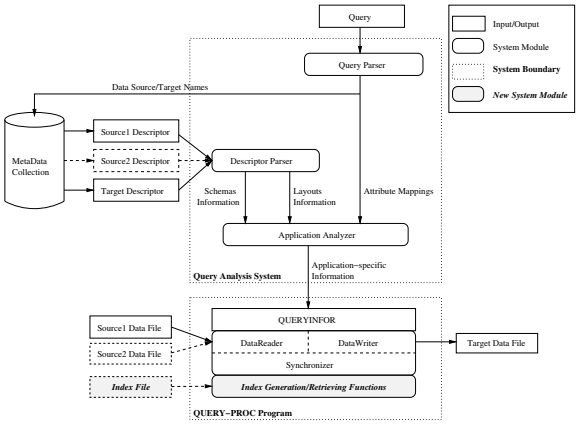

Figure 1. Overview of the Query System Using Indices

3 Challenges and System Overview

This section provides an overview of our system. We will discuss the major challenges involved in incorporating indexing to the query system and our approach.

Our query analysis system are summarized in Figure 1. Compared with our previous work, the main enhancement is on the QUERY-PROC program while a new module of indexing functions were added and it interacts with the Synchronizer module and accesses the index files. Each query submitted to the system is analyzed by the *Query Parser*. Datasets' and attributes' names are retrieved. Logical schema and physical layout information about the datasets are parsed by the *Descriptor Parser*. The *application analyzer* performs all the necessary analysis tasks and translates each query into a set of forthright directions in forms of tables and parameters. Collectively, we call them *QUERYINFOR*. The details of the query analysis system will be explained in Section 4.2. With a QUERYINFOR record plugged in, the general modules and the indexing function are complied into an executable program, QUERY-PROC. The QUERY-PROC interacts with the datasets and indices directly to answer the query. The algorithms of its modules are discussed in Section 4.3. Example indexing functions are also given in the same section.

3.1 Major Goals

Most biologists don't have sophisticated training on computer programming. So the interface simplicity remains one of our main design goals. An SQL like declarative language is used to specify requests. We continued to use a metadata description language to describe the flat file datasets. With these two languages, users need not to provide directions on how to extract values and how to process them to answer their queries. Both will be deduced and executed by the system. The metadata descriptors are independent of queries. Thus, they could be reused by various queries and users. This further simplify the usage.

In order to generate and use indices, the values of the indexed field have to be known. When data is embedded within a flat file, it is not trivial to get these values. Since these values are also needed to answer queries, we dele-

gated this task to a module named QUERY-PROC, which is responsible for answering queries. This simplifies the implementation of the indexing functions. Similarly, the pointer values are also provided by the QUERY-PROC program. The choice of pointer values is not trivial, either. Traditionally in DBMS, key attribute values are used as index pointers. This is also the case with many down-loadable index files, such as those provided by SWISSPROT. However, when data is in its original flat files and not well organized as in DBMSs, the key attribute look-up could be expensive. To save query execution time, we picked lower-level values, the byte offsets of the data entries in the file, as their pointers. With these pointers, the QUERY-PROC program can directly position the curser to the beginning of the target data entry and read the desired entry.

The only input that involves user's programming efforts are the indexing functions. This is unavoidable because different data fields contain different types of data and may request different indexing algorithms. Many biological data indexing algorithms have been developed and they are available either through web service or down-loadable source code. Besides the indexing function names, other information, such as the name of the field to be indexed and the location of the index file, is also needed. They are all properties of the dataset and included as part of the metadata.

Special care has also been given to data re-usability. Metadata descriptors, again, can be shared by all queries that access the same dataset. Indices are reused, too, in the forms of flat files. The QUERY-PROC program evokes the index generation function only when no current index file can be found. Otherwise, indices are loaded by the calling the index retrieving function.

3.2 Integrating Indexing Functions

In our implementation, the indexing functions are treated as plug-in modules of the query execution program. There is no constrain on the internal implementation of the algorithms. However, the system requires these indexing modules to have a unified interface. This requirement is necessary since Synchronizer, the caller of the indexing functions, is implemented as a general module and it needs the minimum information about how to invoke these functions. The interface includes the following components.

- Names of two functions. The index generation function implements the creation of indices from the original data file. The index retrieving function implements loading of the indices and the usage of indices.
- The input to the index generation function is individual data attribute value. The name of the target index file is also passed to this function as parameters.
- The index retrieving function is only aware of the index file and always assume such a file exists. It takes the attribute value to be searched as input and returns a list of pointers.

The interface poses some constrains on the implementation of the indexing algorithms. Our system stores information about single data entry and is memoryless. Only indices bases on individual entries are allowed. The current implementation also forbids generating indices from combined multiple attributes. However, these constrains don't affect most of the bioinformatics applications. Most of the existing codes can be wrapped to meet the requirements, which is illustrated by our experiments.

```
public bool index_loaded ← FALSE;
bool index_sorted ← FALSE;
idx_list id_index ← ∅;

//————————— index generation function —————————
//attribute value, index pointer and index file are passed as arguments
void genomeIndex(string value, int pointer, FILE* index_file) {
   id_index.push_in(idx(value, pointer));
   //each index is a pair of a (processed) value and a pointer
   write value, value.length() and pointer to index_file;
   if (NOT index_loaded)
      index_loaded ← TRUE;
}

//————————— index retrieving function —————————
//query value and index file are passed as arguments, pointer list are returned
int_list getGeneRandom(string query_value, FILE* index_file) {
   int_list candidate_list ← ∅;
   if (NOT index_loaded) {
      read indices from index_file into id_index;
      index_loaded ← TRUE;
   }
   if (NOT index_sorted) {
      sort id_index by ID values;
      index_sorted ← TRUE;
   }
   if (index_loaded AND index_sorted) {
      binary search id_index for query_value;
      if found
         add pointer to candidate_list;
   }
   return candidate_list;
}
```

Figure 2. Algorithm of Example Indexing Functions for Yeast Genome IDs

4 Algorithms and System Implementation

In this section, we explain our system and algorithms in details. We will explain the data description and query language briefly in Section 4.1. The implementation of the query analysis module and query processing program will be illustrated in details, because they relate directly to the usage of indexing. Other system components, although upgraded accordingly, are omitted in our discussion. Interested parties can refer to our earlier publications [28, 29] for more details.

4.1 Languages

This section describes the query and metadata description languages used in our system. With these languages, high-level description of queries and datasets can be easily written.

4.1.1 Query Language

The query language is similar to SQL and has been used in our previous systems. We will use a simple ID look-up query, Figure 4, as an example. The use of keywords, highlighted in the figure, is similar to SQL standard. In this query, the yeast genome database is searched by gene IDs. The query IDs are from a microarray chip dataset, CHIP-DATA. Once a matched gene is found, besides its gene ID, its full description is retrieved.

```
bool match_found ← TRUE;
int_list offset_list;
if(in compare_field_indexing, even valid entries are less than odd valid entries )
   { exchange Source1, Source2; }
while (not reach the end of Source1) { //scan Source1
   call DataReader on Source1 until a full entry extracted;
   clean offset_list;
   for (every tuple t in compare_field) {
      if (compare_field_indexing[2*t+1] is valid) { //Source2 index available
         call index_ret_fun[compare_field_indexing[2*t+1]] with buffer value for
                attr_indexed[2*t+1];
         store returned list as a candidate list;
      }
   }
   offset_list ← intersection of all candidate lists;
   sort offset_list;
   for (every offset i in the offset_list) {
      move cursor of Source 2 to position i;
      call DataReader on Source2 until a full entry extracted;
      match_found ← TRUE;
      for (every tuple t in compare_field) {
         if (compare_field_indexing[2*t+1] is invalid) { //Source2 index not available
            compare buffer value for compare_field[2*t] with buffer value for
                   compare_field[2*t+1];
            if (not match)
               { match_found ← FALSE; }
         }
      }
      if (match_found)
         { call DataWriter until a full entry written to Target; }
      clear Source2 values from value buffer;
   }
   clear Source1 values from value buffer;
}
```

Figure 3. The Algorithm for the Synchronizer Using Indices

```
AUTOWRAP GNAMES //target dataset
FROM CHIPDATA, YEASTGENOME //source dataset(s)
BY CHIPDATA.GENE = YEASTGENOME.ID //testing condition(s)
WHERE //target attributes and their value sources
        GNAMES.GENE = CHIPDATA.GENE
        GNAMES.DE = YEASTGENOME.DESCRIPTION
```

Figure 4. Query Example

4.1.2 Metadata Description Language

We also reuse and extend the *metadata description language* used in our previous tools. Each data resource is represented with one descriptor with two components, the *Dataset Schema Description* and *Dataset Layout Description*. The descriptors are stored on disk as flat files and can be re-used by different queries. Data mining techniques are available and they can help users to write the descriptors semi-automatically [21, 22, 31].

The yeast genome data conforms with the FASTA format and Figure 5 shows its descriptor. As an extension for explaining indexing information, the key word "INDEX" is used. Each index is a tuple of the following format

Attribute_name : Index_data_file_location : Index_Generation_Function : Index_Retrieving_Function : Functions_Location.

Multiple attributes may be indexed and comma signs (",") are used to separate them.

4.2 Query Analysis

Once a query is submitted to the system, a serial of analysis modules are evoked. The goal is to generate a QUERY-INFOR data structure which captures all the instructions needed for answering the query. This section explains the details, with emphasis on the indexing functionality.

```
Component I. Dataset Schema Description in DTD
  <?xml version='1.0' encoding='UTF-8'?>
  <!ELEMENT YEASTGENOME (ID, DESCRIPTION, EC*, SEQ)>
  <!ELEMENT ID (#PCDATA)>
  <!ELEMENT DESCRIPTION (#PCDATA)>
  <!ELEMENT EC (#PCDATA)>
  <!ELEMENT SEQ (#PCDATA)>

Component II. Dataset Layout Description
  DATASET "YeastData" {
    DATATYPE {YEASTGENOME} // Corresponding schema name
    DATASPACE LINESIZE = 100 { // File layout
      < // token <>: repetition, one or more times
        ">" ID
        " " DESCRIPTION
        [ "EC:" EC ] // token []: repetition, zero or more times
        < "\n" SEQ >
      >
    }
    DATA {data/Scerevisiae_prot_20062005} // File location
    INDEX {ID:data/protID.idx:genomeIndex:getGeneRandom:genomeIndex.h}
      // index information
  }
```

Figure 5. The Metadata Descriptor for Yeast Genome

4.2.1 Query and Descriptor Parser

The *Query Parser* is used in this system to parse the query. The testing conditions and attributes pairs in the "WHERE" clause are extracted. The names of the source and target datasets are also extracted. They are used to retrieve the right metadata descriptors. These descriptors are parsed by the *Descriptor Parser*. It has two components, *Schema Parser* and *Layout Parser*, each corresponding to one component of the descriptor. Accordingly, two parse trees, a schema tree and a layout tree, are generated. To this point, all information about the datasets and user query is converted to internal structures and resides in memory.

4.2.2 Application Analyzer

The *Application Analyzer* is the core module of the code generation system. It analyzes each query and captures all necessary directions for the general modules of the query answering program. These directions are organized into *QUERYINFOR*. Since all the metadata analysis is done by the Application Analyzer, the computation load is reduced for the QUERY-PROC and the overall performance is improved.

Specifically, the Application Analyzer performs following tasks.

- Rename leaves in the parse trees by integers. The integers can not only be used as reference to the entities, but also as pointers to look-up tables. Specifically,
  - Assign all the DLM-VAR nodes in the source and target layout trees with *node numbers*.
  - Categorize all the leaves in source schema trees into useful or useless attributes. The attributes not called by the query is useless and are labeled with

Layout Number	...	3	4	5	6
Schema Label	...	R3	R2	0	0

(a) Label Look-up Table

Layout Number	...	3	4	5	6
Delimiter	...	">"	" "	"EC:"	"\n"

(b) Delimiter Look-up Table

Layout Number	Reachables List
...	...
3	4
4	5, 6
5	5, 6
6	3

(c) Reachable Look-up Table

Index Number	0
attr_indexed	R3
index_gen_fun	genomeIndex
index_ret_fun	getGeneRandom
index_file	data/id.idx

(d) Index Look-up Table

Parameter	Value
compare_field	(R1, R3)
compare_field_indexing	(-,0)
semi_size	0, 0
regl_size	1, 2
complete_in	2, 6
complete_out	2

(e) Parameters

Figure 6. QUERYINFOR for Example Yeast Genome Query

0. The useful attributes are further categorized by their cardinalities into semi-structured or regular attributes. They are labeled separately with *node labels*.

  - Using mapping information provided by the Query Parser, label leaves in the target schema tree with the same labels of the corresponding nodes in the source schema trees.

- Draw correspondence between schema tree leaves and layout tree leaves. The result is a *label look-up table*. The i-th element in the table is the label of the schema leaf associated with the DLM-VAR leaf with number i.
- Organize delimiters in a *delimiter look-up table* by the DLM-VAR leaf numbers.
- Calculate the lists of reachable nodes for the DLM-VAR leafs and record the lists in a *reachable look-up table*. Intuitively, they contain all possible DLM-VAR pairs that could be read or written next.
- Organize index information into *index look-up table*. It contains labels of the indexed attributes, index functions names and index file locations. Specifically,
  - Record labels of the attributed indexed in *attr_indexed[]*.
  - Record the names of the index generation and retrieving functions in *index_gen_fun[]* and *index_ret_fun[]*.
  - Record locations of the indices in *index_file[]*.
- For the test conditions, test their index availability in the index look up table. Record it as *compare_field_indexing*.
- Record query test conditions as pairs of labels in *compare_field*.
- Record the total numbers of useful source semi-structured and regular attributes in *semi_size[]* and *regl_size[]*.
- Examine whether index is available for each query field by checking compare_field against index look-up table. Record the results as pairs of pointers to the index look-up table in *compare_field_indexing[]*.
- Record the node numbers of last DLM-VAR nodes in *complete_in[]* and *complete_out*. They indicates the completion of reading and writing of one entry.

The QUERYINFOR for the example query on yeast genome is illustrated in Figure 6. The layout number 3 through 6 correspond to the attributes from the yeast genome file. Take its ID attribute (number 3) as an example. It was labeled as the third regular attribute, R3. The ID values are followed by the delimiter of the next attribute. It has only one reachable node which is node 4. The genome ID values are compared with gene names from the other dataset (R1) as indicated by the parameter *compare_field*. The 0 value in the parameter *compare_field_indexing* means that the genome IDs are indexed and the indexing information

can be found from the first entry of the index look-up table. The directions for the QUERY-PROC program can be solely inferred from the QUERYINFOR alone, as will be discussed in the next section.

4.3 Query Execution: The QUERY-PROC Program

The QUERY-PROC program accesses the datasets to answer the query. When index functions are available for the testing fields, it will create the indices if index files are not available, load the indices and use them to search for the data entries. Besides a query-specific QUERYINFOR record, a complete QUERY-PROC contains three general modules, DataReader , DataWriter and Synchronizer. The indexing functions are provided by the user.

4.3.1 Value Buffer, DataReader and DataWriter

The implementation of value buffer, dataReader and dataWriter is similar to the previous system. For the sake of completeness, we will review them briefly here. The details can be found in [28, 29]. Because of the big sizes of biological datasets, we limit the amount of data stored in memory so that a scalable performance could be achieved. At any time of the query answering process, only useful data values of one entry from each source dataset are kept in memory. The exact configuration of the value buffer is query-specific and determined by the parameter semi_size and regl_size from the QUERYINFOR. Within the buffer, two types of buffer units, string and string list, are used, each corresponds to one type of the attributes.

The value buffer is accessible to all QUERY-PROC general modules. DataReader and DataWriter, are responsible for transferring data between the datasets and the value buffer. The datasets are read/written by them sequentially. The reachable lists in QUERYINFOR are used by DataReader to search for the boundaries of the data values. The reachable delimiter that appears first in the source stream indicates the end of the attribute value. Partial values are merged by DataReader before written to the value buffer. On the other hand, when a value's length exceed the line limit, DataWriter splits it and inserts appropriate delimiters in between.

4.3.2 Indexing Functions

Designing a system that is easy to use is one of our design goals. The simplicity does not only refer to the high level description of the query and datasets, but also relate to the easiness of implementation or re-implementation of the indexing functions.

As discussed earlier, QUERY-PROC program is responsible for parsing of the datasets. Users only need to provide functions that write, load and search index values, assuming the values of the data attributes and pointers are correctly passed as arguments. The lower level complexity of the datasets are transparent to these functions.

A simple index generation and retrieving function are illustrated in Figure 2 to show the interface. This example was used to index the ID field of the yeast genome. Other algorithms, as reviewed in Section 2, can also be plugged into our system as long as the same interface can be provided.

4.3.3 Synchronizer

The Synchronizer serves as the central control module of the QUERY-PROC program. It determines when to call DataReader, DataWriter module and indexing functions. Figure 3 shows its pseudo-code.

In order to make the QUERY-PROC capable of handling as many queries as possible, we used the nested loops with indexing technique for join queries. It makes little assumption about the availability of the index and property of dataset. When no testing attribute is indexed, the Synchronizer compares every possible pair of data entries. The data entries are scanned sequentially, which agrees with our design philosophy. Whenever possible, indices are utilized to retrieve data entry randomly for a better performance. The execution of other types of queries, such as selection, can be easily incorporated into this structure.

The first sub-task of the Synchronizer is to check the availability of index data files. If an index file cannot be found but its indexing function is present, the source dataset will be scanned once using DataReader and the missing indices will be created by calling the right generation function.

Once all the necessary indices are ready, the Synchronizer starts to answer the query. In the outer loop, it calls the DataReader to scan the first dataset, *Source1*, until a full entry has be extracted. Use the values stored in the value buffer, the Synchronizer calls available index retrieving functions to search the other dataset, *Source2*, for candidate entries. When more than one index is used, the returned lists of entries are intersected. The Synchronizer then retrieves the entries in the candidate list one by one. The value buffer units allocated for the Source2 are filled accordingly. Whenever a full candidate entry is read, the test conditions with no index are checked. Only the entries that pass all the tests are the true answers. The DataWriter will then be called and the answer will be deposited to the target dataset. As the last step, the Synchronizer cleans the value buffer for the new circle.

As a consequence of the nested loops structure, the order of the target entries are the same as they are in the Source1 dataset. The algorithm will fail if reorganization of the data is required. However, we found that most bioinformatics applications do not need such reorganization.

5 Experimental Results

We tested our system with four applications. They are all based on real biological problems. According to the indexing functions used, they can be categorized into two groups. Classic indices are direct mapping of attribute values. These indices can be used to answer queries with value comparison. We refer to these queries as general database searches and three experiments were conducted and presented in Section 5.1. Indices can also be built on values calculated from the original attribute values. These indices are more complicated and can be used to answer more involved queries. Sequence similarity search is a good example from biology domain. By converting the nucleic acid sequences using wavelet, similar sequences can be extracted using these indices. We will present two sequence indexing algorithms in Section 5.2.

5.1 General Database Search with Index

We first tested our system using classic indices. A simple index is a list of tuples in the form (value, offset). The list is sorted by the indexed attribute value. Binary search method can used to search the index. The implementation of such indexing functions has been discussed in Section 4.3.2. Three experiments were conducted. In all of them, the performance of the QUERY-PROC program with and without indexing is compared.

5.1.1 Microarray Genes Information Look-Up

High throughput experiments, such as microarray, allow biologists to monitor multiple objects at the same time. We used our system to search the Comprehensive Yeast Genome Database (CYGD) with names of all 120 genes from a yeast kinase protein array [32]. The metadata analysis step took less than 0.01 second. To create simple indices on the gene names of CYGD, 0.72 second was used. With the indices, it took the QUERY-PROC program 20.89 seconds to answer the query. Without indexing, the brute-force method with two nested file scanning loops took 81.59 seconds to finish the same task. In this experiment, the use of a single index resulted in 73.5% reduction on running time. When the indices are reused, more performance gain could be achieved.

5.1.2 BLAST Output Enhancement

The outputs of BLAST programs only provide limited information about the returned sequences. We designed a query to add user desired information to the BLAST output. The SWISSPROT database was first searched using NCBI BLAST service with random protein sequences. Using the BLAST output as one source dataset, the generated QUERY-PROC program searches the SWISSPROT and adds full protein names and full protein sequences to the returned sequences. SWISSPROT flat file, which is of size 714.09 MB, was downloaded from one of the ExPASy servers [3].

Figure 7 summarizes the performance. Again, the metadata analysis time was less than 0.01 second and was omitted from the figure. The size of query is in term of the number of similar sequences returned by BLAST. As it increases, the average time spend on each query sequence decreased slightly. This is because the overhead of loading index only needs to be paid once. It also explains the improvement of the running time reduction on larger queries.

5.1.3 Link OMIM to SWISSPROT

Online Mendelian Inheritance in Man (OMIM) [7] is a catalog of human genes and genetic disorders. Its flat file is 63.04 MB. It contains 12,158 entries and 5,994 of them are referred by SWISSPROT. However, the reverse links are not available. Therefore it is difficult to retrieve related amino acid sequence of the gene responsible for a disease. We use our system to add such a link to OMIM. The original OMIM format was maintained except that one data field was added to each entry and it contained the ID of the referring SWISSPROT record.

[3]Please see http://us.expasy.org/sprot/download.html

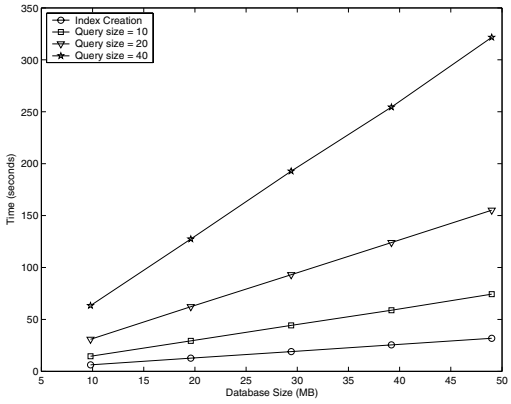

Figure 8. Performance of CYGD Similarity Search Using Singh's Algorithm

It took the system 0.01 second to analyze the query. A total of 794.62 seconds were used by the generated QUERY-PROC program to answer the query on bare datasets, in which 188.31 seconds were used to index the SWISSPROT ID field, 606.31 seconds were used to answer the query with the indices. Compared with the results from our previous system, the performance was 1321 times faster with indices.

5.2 Similarity Search on Sequence Databases

Sequence similarity searches are important to biological research and many algorithms, such as BLAST, have been developed. Feature based indexing, in which sequences are first transformed into multi-dimensional feature space, is an important approach to search sequence databases. Among many algorithms developed by various research groups, we implemented two of them. The algorithm proposed by Singh et al [10] indexes the wavelet coefficients of sub-sequences using Minimum Bounding Rectangles. Ferhatosmanoglu et al [18] use R-trees and scalar quantization based structures to index transformed sequence data. In our experiments, we used these two index approaches to search real biological sequence databases. They are the Comprehensive Yeast Genome Database (CYGD) [4] from Munish Information Center for Protein Sequences (MIPS) and GenBank [5] mammalian sequence entries part 1 (gbmam1.seq). Both databases were downloaded in flat files. The query sequences were random DNA sequences. For every query sequence, twenty most similar sequences were extracted from the database. The experimental results are summarized in Figure 8 and 9.

In both experiments, we replicated the database to test the scalability of the system. The results show that the index creation and query answering time of the generated QUERY-PROC programs scaled linearly with respect to the size of the database. As the number of query sequences increases, the query answering time increases linearly, too.

On the other hand, the time used by the system to analyze the metadata and generate the QUERYINFOR structure was constant. Less than 0.01 second were used in all experiments

[4]Please see http://mips.gsf.de/genre/proj/yeast/

[5]Please see ftp://ftp.ncbi.nih.gov/genbank/

Query Size		3	5	12
Time (seconds)	Index Generation	186.16		
	Query Answer with Index	1.96	2.73	6.95
	Query Answer without Index	191.88	413.86	1097.86
Time Reduction	With Index Generation	2.0%	54.4%	82.4%
	Without Index Generation	99.0%	99.3%	99.4%

Figure 7. Performance of BLAST-ENHANCE Query

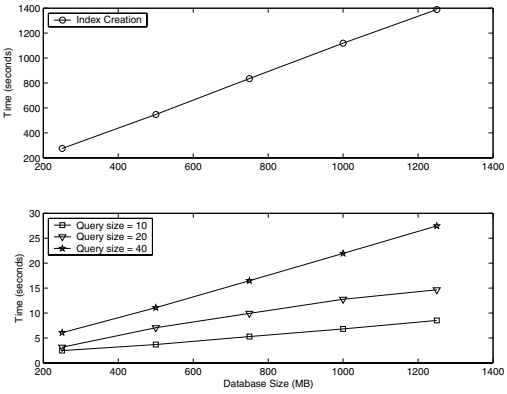

Figure 9. Performance of GENBANK Similarity Search Using Ferhatosmanoglu's Algorithm

and are negligible compared to the query answering times.

6 Conclusion

We enhanced our previous query processing system to answer queries with indices when they are available. The new system supports indexing functions that are provided by the users and allows various indexing mechanism to be applied. These functions can be reused by multiple datasets on different attributes. We believe that the system's interface is easy enough to use because declarative languages are used. Our approach is flat file based and requires no database support. Experiments showed that the use of indices on flat files can improve the system's performance and functionality.

References

[1] Mohammad Anwar, Truong Nguyen, and Marcel Turcotte. Identification of consensus rna secondary structures using suffix arrays. *BMC Bioinformatics*, page 244, 2006.

[2] Axel Bernal, Uy Ear, and Nikos Kyrpides. Genomes online database (gold): a monitor of genome projects worldwide. *Nucleic Acids Res*, January 2001.

[3] Daniel C. Berrios, Russell J. Cucina, and Lawrence M. Fagan. Methods for semi-automated indexing for high precision information retrieval. *J Am Med Inform Assoc*, pages 637–652, Nov–Dec 2002.

[4] B Boyce and M Lockard. Automatic and manual indexing performance in a small file of medical literature. *Bull Med Libr Assoc*, 63:378–385, October 1975.

[5] Margaret H. Coletti and Howard L. Bleich. Medical subject headings used to search the biomedical literature. *J Am Med Inform Assoc*, pages 317–323, Jul–Aug 2001.

[6] K L Curtis, A C Weller, and J M Hurd. Information-seeking behavior of health sciences faculty: the impact of new information technologies. *Bull Med Libr Assoc*, pages 402–410, October 1997.

[7] Center for Medical Genetics at Johns Hopkins University and National Center for Biotechnology Information at National Library of Medicine. Online mendelian inheritance in man, omim (tm). 1997.

[8] Eva Huala, Allan W. Dickerman, Margarita Garcia-Hernandez, Danforth Weems, Leonore Reiser, Frank LaFond, David Hanley, Donald Kiphart, Mingzhe Zhuang, Wen Huang, Lukas A. Mueller, Debika Bhattacharyya, Devaki Bhaya, Bruno W. Sobral, William Beavis, David W. Meinke, Christopher D. Town, Chris Somerville, and Seung Yon Rhee. The arabidopsis information resource (tair): a comprehensive database and web-based information retrieval, analysis, and visualization system for a model plant. *Nucleic Acids Res*, pages 102–105, January 2001.

[9] Ela Hunt, Malcolm P. Atkinson, and Robert W. Irving. Database indexing for large dna and protein sequence collections. *The VLDB Journal ł The International Journal on Very Large Data Bases*, pages 256–271, November 2002.

[10] Tamer Kahveci and Ambuj K. Singh. Efficient index structures for string databases. In *The VLDB Journal*, pages 351–360, 2001.

[11] Tamer Kahveci and Ambuj K. Singh. Accelerating substring searching: breaking the i/o barrier. UCSB Technical Report, 2002.

[12] Tamer Kahveci and Ambuj K. Singh. Fast alignment of large genome databases: a demonstration. In *19th International Conference on Data Engineering, 2003. Proceedings*, pages 768–770, March 2003.

[13] Tamer Kahveci and Ambuj K. Singh. Optimizing similarity search for arbitrary length time series queries. *IEEE Transactions on Knowledge and Data Engineering*, pages 418–433, 2004.

[14] Tamer Kahveci and Ambuj K. Singh. Map: Searching large genome databases. pages 303–314, 3004.

[15] Henry J. Lowe, Bruce G. Buchanan, Gregory F. Cooper, and John K. Vries. Building a medical multimedia database system to integrate clinical information: an application of high-performance computing and communications technology. *Bull Med Libr Assoc*, pages 57–64, January 1995.

[16] Joyce A. Mitchell, Alan R. Aronson, James G. Mork, Lillian C. Folk, Susanne M. Humphrey, and Janice M. Ward. Gene indexing: Characterization and analysis of nlms generis. pages 460–464, November 2003.

[17] Linda S Murphy, Sibylle Reinsch, Wadie I Najm, Vivian M Dickerson, Michael A Seffinger, Alan Adams, and Shiraz I Mishra. Searching biomedical databases on complementary medicine: the use of controlled vocabulary among authors, indexers and investigators. *BMC Complement Altern Med*, page 3, 2003.

[18] Ozgur Ozturk and Hakan Ferhatosmanoglu. Effective indexing and filtering for similarity search in large biosequence databases. In *BIBE*, pages 359–366, 2003.

[19] Daniel L. Rubin, Caroline F. Thorn, Teri E. Klein, and Russ B. Altman. A statistical approach to scanning the biomedical literature for pharmacogenetics knowledge. *J Am Med Inform Assoc*, pages 121–129, Mar–Apr 2005.

[20] Tetsuo Shibuya and Isidore Rigoutsos. Dictionary-driven prokaryotic gene finding. *Nucleic Acids Res.*, pages 2710–2725, June 2002.

[21] Kaushik Sinha, Xuan Zhang, Ruoming Jin, and Gagan Agrawal. Learning Layouts of Biological Datasets Semi-Automatically. In *Proceedings of Data Integration in Life Sciences (DILS)*, July 2005.

[22] Kaushik Sinha, Xuan Zhang, Ruoming Jin, and Gagan Agrawal. Using Data Mining Techniques to Learn Layouts of Flat-File Biological Datasets . In *Proceedings of IEEE Symposium on Bioinformatics and Bioengineering (BIBE)*, October 2005.

[23] 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.

[24] Brian P Suomela1 and Miguel A Andrade. Ranking the whole medline database according to a large training set using text indexing. *BMC Bioinformatics*, page 75, 2005.

[25] Swiss-prot protein knowledgebase release 41.21 statistics. http://us.expasy.org/sprot/relnotes/relstat.html, 2003.

[26] Hemant D. Tagare, C. Carl Jaffe, and James Duncan. Medical image databases: A content-based retrieval approach. *J Am Med Inform Assoc*, pages 184–198, May–Jun 1997.

[27] Zhenqiang Tan, Xia Cao, Beng Chin Ooi, and A.K.H Tung. The ed-tree: an index for large dna sequence databases. In *Conference on Scientific and Statistical Database Management*, pages 151–160, July 2003.

[28] Xuan Zhang and Gagan Agrawal. Enabling information integration and workflows in a grid environment with automatic wrapper generation. In *Grid 2005, held in conjunction with SC 2005*. ACM Press, November 2005.

[29] Xuan Zhang and Gagan Agrawal. A Tool for Supporting Integration Across Multiple Flat-File Datasets. In *Proceedings of the Conference on Bioinformatics and Bioengineering (BIBE)*. IEEE Computer Society, October 2006.

[30] Xuan Zhang, Ruoming Jin, and Gagan Agrawal. Assigning Schema Labels Using Ontology and Heuristics. In *Proceedings of the Conference on Bioinformatics and Bioengineering (BIBE)*. IEEE Computer Society, October 2006.

[31] Xuan Zhang, Ruoming Jin, and Gagan Agrawal. Assigning schema labels using ontology and hueristics. In *BIBE*, pages 269–280, 2006.

[32] H Zhu, J Klemic, S Chang, P Bertone, A Casamayor, K Klemic, D Smith, M Gerstein, M Reed, and M Snyder. Analysis of yeast protein kinases using protein chips. *Nature Genetics*, 26:283–289.