# Searching on the Secondary Structure of Protein Sequences

**Laurie Hammel** **Jignesh M. Patel**

Department of Electrical Engineering and Computer Science
University of Michigan,
1301 Beal Avenue, Ann Arbor, MI 48109, USA
{lhammel, jignesh}@eecs.umich.edu

## Abstract

In spite of the many decades of progress in database research, surprisingly scientists in the life sciences community still struggle with inefficient and awkward tools for querying biological data sets. This work highlights a specific problem involving searching large volumes of protein data sets based on their secondary structure. In this paper we define an intuitive query language that can be used to express queries on secondary structure and develop several algorithms for evaluating these queries. We implement these algorithms both in Periscope, a native system that we have built, and in a commercial ORDBMS. We show that the choice of algorithms can have a significant impact on query performance. As part of the Periscope implementation we have also developed a framework for optimizing these queries and for accurately estimating the costs of the various query evaluation plans. Our performance studies show that the proposed techniques are very efficient in the Periscope system and can provide scientists with interactive secondary structure querying options even on large protein data sets.

## 1. Introduction

The recent conclusion of the Human Genome Project has served to fuel an already explosive area of research in bioinformatics that is involved in deriving meaningful knowledge from proteins and DNA sequences. Even with the full human genome sequence now in hand, scientists still face the challenges of determining exact gene locations and functions, observing interactions between proteins in complex molecular machines, and learning the structure and function of proteins through protein conservation, just to name a few. The progress of this scientific research in the increasingly vital fields of functional genomics and proteomics is closely connected to the research in the database community in that analyzing large volumes of biological data sets involves being able to maintain and query large genetic and protein databases [19, 27]. If efficient methods are not available for retrieving these biological data sets, then unfortunately the progress of scientific analysis is encumbered by the limitations of the database system.

This work looks at a specific problem of this nature that involves methods for searching protein databases based on secondary structure properties. We will define a problem that the scientific community faces regarding current protein search techniques and provide a query language and a system to efficiently answer these queries.

### 1.1 Biological Background

Proteins have four levels of structural organization, primary, secondary, tertiary, and quaternary; the latter two are not considered in this work. The primary structure is the linear sequence of amino acids that makes up the protein; this is the structure most commonly associated with protein identification [24]. The secondary structure describes how the linear sequence of amino acids folds into a series of three-dimensional structures. There are three basic types of folds: alpha helices (h), beta sheets (e), and turns or loops (l). Because these three-dimensional structures determine a protein's function, knowledge of their patterns and alignments can provide important insights into evolutionary relationships that may not be recognizable through primary structure comparisons [22]. Therefore, examining the types, lengths, and start positions of its secondary structure folds can aid scientists in determining a protein's function [2].

### 1.2 Scientific Motivation

The discovery of new proteins or new behaviors of existing proteins necessitates complex analysis in order to

**Proceedings of the 28th VLDB Conference,**
**Hong Kong, China, 2002**

determine their function and classification. The main technique that scientists use in determining this information has two phases. The first phase involves searching known protein databases for proteins that "match" the unknown protein. The second phase involves analyzing the functions and classifications of the similar proteins in an attempt to infer commonalities with the new protein [2]. These phases may be intertwined as the analysis of matches may provide interesting results that could be further explored using more refined searches.

The above simplification of the search process glosses over the actual definition of protein similarity. The reason for this is that no real definition of protein similarity exists; each scientist has a different idea of similarity depending on the protein structure and search outcome goal. For example, one scientist may feel that matching on primary structure is beneficial, while another may be interested in finding secondary structure similarities in order to predict biomolecular interactions [16]. In addition to these complications, there is a plethora of differing opinions even within same-structure searches. One scientist may want results that exactly match a small, specific portion of the new protein, while another may feel that a more relaxed match over the entire sequence is more informative.

What is urgently needed is a set of tools that are both *flexible* with regards to posing queries and *efficient* with regards to evaluating queries on protein structures. Whereas there are a number of public domain tools, such as BLAST, for querying genetic data and the primary structure of proteins [1, 3, 4, 27, 28], to the best of our knowledge there are no tools available for querying on the secondary structure of proteins. This work addresses this void and focuses on developing a *declarative* and *efficient* search tool based on secondary structure that will enable scientists to encode their own definition of secondary structure similarity.

Another motivation for this work stems from the desire for more efficient search tools. For example, the BLAST [1, 3, 4, 28] queries submitted to their web site can take hours or days to return. This is a combination of two factors: high server loads caused by multiple concurrent users, and the efficiency of the query evaluation algorithm. The server load bottleneck can be alleviated by downloading the BLAST code and running it on a local server. This solution is inadequate, however, as most of the query evaluation algorithms employ sequential scans to answer a query [13]. This translates into long response times, which are unacceptable considering that often scientists want to pose queries interactively to systematically validate or invalidate a number of hypotheses. As the set of hypotheses is typically altered based on previous query answers, long query response times may dramatically slow down the rate of progress of their research. Therefore a major goal of our secondary structure search tool is to employ very efficient query evaluation techniques.

### 1.3 Contributions and Paper Organization

This paper makes the following contributions. We define a simple and intuitive query language for posing secondary structure queries based on segmentation. We identify various algorithms for efficiently evaluating these queries and show that depending on the query and segment selectivities, the choice of the algorithm can have a dramatic impact on the performance of the query.

We develop a query optimization framework to allow an optimizer to choose the optimal query plan based on the incoming query and data characteristics. As the accuracy of any query optimizer is dependent on the accuracy of its statistics, for this application we need to accurately estimate both the segment and overall result selectivities. We develop histograms for estimating these selectivities and demonstrate that these histograms are very accurate and take only a small amount of space to represent.

Finally, we implement our techniques in Periscope, a native DBMS that we have developed for querying biological data sets, as well as a commercial object-relational database management system (ORDBMS). Periscope allows us to test algorithms that we cannot test using the commercial system; and using actual data sets, we are able to experimentally demonstrate Periscope's effectiveness. We believe that Periscope is extremely efficient and will be a valuable addition to the arsenal of search tools that are needed for life sciences research.

The remainder of the paper is organized as follows: in Section 2 we describe the protein format that we use. Section 3 describes our query language. Our methods for evaluating queries are discussed in Section 4, while Section 5 describes the framework for a query optimizer and our estimation techniques. Section 6 contains experimental results, and Section 7 discusses future work and concludes the paper.

## 2. Protein Format

The first task to accomplish is to establish the protein format we will use in our system. This format is largely dependent on the prediction tool that is used to generate the secondary structure of proteins in our database. For the majority of known proteins, their secondary structure is a predicated measure; only a few hundred proteins actually have known secondary structures. In order to obtain the secondary structure for a given protein, therefore, it is necessary to enter its primary structure into a prediction tool that will return the protein's predicted secondary structure. Most available prediction tools are between 60% and 70% accurate.

The tool used to predict the secondary structure information for the proteins in our database is Predator [10]. Predator is a secondary structure prediction tool based on recognition of potentially hydrogen-bonded residues in a single amino acid sequence; it is 65% accurate. We chose this particular tool because we were

```
name:  t2_1296
id:  1
length:  554
primary structure:  |GQISDSIEEKRGFFSTKR..
secondary structure:|HLLLLLLLLLLHHHEEEE..
probability:        |855577763445449476..
```

**Figure 1: Sample Protein**

able to download the code and run it locally on our own machine rather than submitting the database proteins one-by-one to their site. However, our techniques will work with other protein prediction tools as well.

Predator returns the protein name, its length in amino acids, its primary structure, and its predicated secondary structure along a number in the range 0-9 for each position. This number indicates the probability that the prediction is accurate for the given position. We add a unique id to each protein for internal purposes. Figure 1 contains a portion of a sample protein in our database.

## 3. Query Language and Sample Queries

Next we determine the types of queries that are useful to scientists in order to examine secondary structure properties and design a query language to express these queries. Based on interviews with scientists who perform secondary structure protein analysis on a regular basis, we are able to formulate three initial classes of queries. As these queries are defined, an intuitive query language begins to emerge. Due to the fact that only three types of secondary structure can occur in a protein sequence, 'h', 'e', and 'l', and as these types normally occur in groups as opposed to changing at each position, it is natural to characterize a portion of a secondary structure sequence by its type and length. For example, because the sequence 'hhhheeeelll' is more likely to occur than 'helhelehle,' it is intuitive to identify the first sequence as three different segments: 4 h's, 4 e's, and 3 l's.

The formal process for posing a query is to express the query as a sequence of *segment predicates*, each of which must be matched to satisfy the query. Each segment predicate in the query is described by the type and the length of the segment. It is often necessary to express both the upper and lower bounds on the length of the segment instead of the exact length. Finally, in addition to the three type possibilities, 'h', 'e', and 'l', we also use a fourth type option, '?', which stands for a gap segment and allows scientists to represent regions of unimportance in a query. The formal query language is defined in Figure 2. A quick note on terminology: throughout this paper we will refer to segment predicates as either query predicates or simply predicates.

We will now look at three important classes of queries that can be expressed using the language described above. In the simplest situations, scientists would like to find proteins that contain an exact query sequence, such as {<h 3 3><e 4 4>}. Our algorithms take the exactness of these

```
Query -> {Segments}
Segments -> Segment*
Segment -> <type lb ub>
type -> e | h | l | ?
lb -> any integer >= 0
ub -> any integer >= 0 | ∞
Segment Constraint:  lb <= ub
```

**Figure 2: Query Language Definition**

predicates literally in that matches that are part of a larger sequence are not returned. For example, the sequence 'hhhheeee' would not match the above query because it contains four 'h's, not the specified three. While exact matching is important, in some cases it may be sufficient to find matches of approximate length. This can be expressed using range queries such as {<h 3 5><l 2 8>}, which finds all proteins that contain a helix of length 3 to 5 followed by a loop of length 2 to 8. Another feature scientists would like to be able to express in their queries is the existence of gaps between regions of importance. A gap query can be expressed as {<h 4 6><? 0 ∞><l 5 5>}, which finds all proteins that contain a helix of length 4 to 6 followed at some point by a loop of length 5. These three classes of queries provide an initial functionality for our system to solve; we will look at more complex queries in our future work. A more formal definition of the three query classes may be found in the full-length version of this paper [12].

## 4. Query Evaluation Techniques

This section describes four methods for evaluating the types of queries defined above. The first approach uses a protein scan while the last three utilize a segmentation technique similar to that described in Section 3 that represents proteins as sequences of segments.

### 4.1 Complex Scan of Protein Table (CSP)

The first approach performs a scan of the protein table itself. One by one, each protein in the database is retrieved, its secondary structure is scanned, and its information is returned if the secondary structure matches the query sequence. The matching check is performed using a non-deterministic finite state machine (FSM) technique similar to that used in regular expression matching [26]. Each secondary structure is input to the FSM one character at a time until either the machine enters a final (matching) state or it is determined that the input sequence does not match the query sequence. The FSM itself is constructed once for each query.

As protein sequences can be long, sometimes consisting of thousands of amino acids, it is common for a query sequence to match more than once in a given protein. Scientists are interested in each match, not just each matching protein. In other words, if a sequence matches a given protein in two distinct places, each of these places must be reported separately. To achieve this result our algorithm checks for all possible occurrences in a protein by running the FSM matching test once for each position in the protein's secondary structure.

### 4.2 General Segmentation Technique

The last three approaches are based on a segmentation scheme that represents proteins as a sequence of segments. This segmentation technique is similar to the one described in [23] in which they are interested in retrieving sequences of integers. The idea is to break the secondary structure of a protein into segments of like types. These segments are stored in a separate segment table. Along with the type and length of each segment, the protein id of the segment's originating protein and the start position of the segment in that protein are also stored. A multi-attribute B+-tree index is built on the segment table's type and length attributes. A clustered B+-tree index is also built on the protein id of the protein table to facilitate protein retrieval. Table 1 and Table 2 show an example of several small protein entries with their corresponding segment tuples. The protein table fields are the same as described in Figure 1.

| name | id | len | primary | secondary | prob. |
|---|---|---|---|---|---|
| A | 1 | 5 | mtgpi | lleee | 99401 |
| B | 2 | 6 | liffki | hhheee | 983121 |

**Table 1: Sample Protein Table**

| seg id | id | type | length | start position |
|---|---|---|---|---|
| 1 | 1 | l | 2 | 1 |
| 2 | 1 | e | 3 | 3 |
| 3 | 2 | h | 3 | 1 |
| 4 | 2 | e | 3 | 4 |

**Table 2: Sample Segment Table**

The remaining three segmentation techniques all incorporate some variation on the following plan description to produce proteins that satisfy a given query. In general, each non-gap predicate of a query can be evaluated using either a scan of the segment table or an index probe. Once individual matching segments of the query have been retrieved, they can be merged based on protein id; the start position information can then be used to satisfy the ordering constraints between segments to produce final matching results.

In all three techniques, once the matching protein ids have been found, they must still be joined with the protein table in order to obtain the actual proteins. This is accomplished by an index-nested loops join (INLJ) of the protein ids with the B+-tree index built on the protein id attribute of the protein table. These protein ids (obtained from the segment predicate matches) are first sorted in order to improve the performance of the INLJ. This join provides quick retrieval of the actual proteins stored in the protein table, especially as the B+-tree index is clustered on the protein id attribute.

This segmentation query plan can be conveyed in standard database terminology through SQL queries using the segment and protein tables. For example, in SQL the query {<e 3 9><? 3 5><h 2 2>} is expressed as:

"select * from proteinTbl p, segTbl $s_1$, segTbl $s_2$
where $s_1$.type = 'e' and $s_2$.type = 'h' and $s_1$.id = $s_2$.id
and $s_1$.id = p.id and $s_1$.length >= 3
and $s_1$.length <= 9 and $s_2$.length = 2
and $s_2$.start_pos – ($s_1$.start_pos + $s_1$.length) <= 5
and $s_2$.start_pos – ($s_1$.start_pos + $s_1$.length) >= 3;"

#### 4.2.1 Simple Scan of Segment Table (SSS)

In this technique the entire segment table is scanned for segments that match the most highly selective predicate of the query. All of the segments returned by the scan then participate in the aforementioned INLJ to retrieve their actual proteins. If there are additional predicates in the query, each retrieved protein is then scanned using the FSM technique described in Section 4.1 to determine the final matching verdict.

#### 4.2.2 Index Scan of Segment Index (ISS)

The index scan query plan is essentially identical to the SSS method with one exception. Instead of scanning the segment table, the ISS method probes the segment index with the most selective segment predicate.

#### 4.2.3 Multiple Index Scans of Segment Index (MISS(*n*))

The final method described in this paper, the multiple index scan technique, is a generalization of the ISS plan. The basic change is that instead of only performing one index probe, the B+-tree index is now probed *n* times with the *n* most highly selective query predicates, where *n* can range from two to the total number of predicates in the query. The segment results of each individual index probe are sorted, first by protein id and then by start position, and written to separate files.

The newly written files then participate in an *n*-way sort-merge join to find query segments with the same protein id. At this point the start position information is used to determine whether the segments occur in the correct order within the protein and if the proper gap constraints between them are met. If the segments match the query constraints, then the corresponding protein id is returned. As with the previous two plans, the protein id then participates in an INLJ with the protein id index followed by a possible complex scan to test for any remaining query predicates.

## 5. Query Optimizer and Estimation

When a query is posed to Periscope, the system must decide which of the four plans should be used to evaluate the given query. In this section we present the framework of a query optimizer that is used to make this decision. As in the classic System R paper, our query optimizer utilizes cost functions that model the CPU and I/O resources of each plan [5, 25]. These cost functions take as input the estimations of the selectivity of each of the query predicates and the selectivity of the result. Traditional database management systems utilize histograms to provide such estimations [14, 15, 17, 20, 25]. The unique, restricted nature of the segment query language and the composition of protein secondary structure allows the Periscope query optimizer to incorporate these standard techniques and expand the estimation capabilities of histograms beyond their typical capacity. We utilize two histograms in our current implementation: a basic one that determines the selectivities of the query predicates and a more complex one to estimate the result protein selectivity.

### 5.1 Basic Histogram

The basic histogram contains information about the number of segments in the segment table for a given type and length pair. As there are only three possible types, 'e', 'h', and 'l', and as the segments are usually relatively small in length, it is neither space nor time consuming to maintain exact counts for the majority of protein segments. The basic histogram is stored in the form of a *k x 3* matrix, where *k* is the number of length buckets in the histogram and the second dimension has one value for each of the three possible types, 'e', 'h', and 'l'. For example, position [7][2] holds the number of <h 7 7> segments. The last bucket is used to represent all segments with length greater than or equal to *k*. For range predicates, an estimate is computed by summing the counts in the appropriate range of buckets. This estimate is *exact* for all segment predicates that are less than *k* in length.

In our current implementation, the number of buckets is set to one hundred, since segments rarely have a length of longer than one hundred positions. This size is also small enough to ensure a compact storage representation for the histogram. Segments over a length of one hundred are considered to have a default low selectivity.

This histogram may be populated during or immediately following the loading of the segment table. Updates can be performed upon each new protein addition without significant time penalty. With the protein data set that we use for our experimentation, which contains 248,375 proteins and their associated 10,288,769 segments, this histogram requires only 13 seconds to build and is created immediately after the loading of the segment table. The time spent by the query optimizer in estimating query predicate selectivities using this histogram is minimal, less than a millisecond per predicate on average. In terms of space requirements, the histogram contains information about greater than 99% of all segments and occupies only 1.2 KB of disk space.

### 5.2 Complex Histogram

The second histogram, which has a more complex structure, is used to estimate the selectivity of the entire query result, not just of a given query predicate. This calculation procedure surpasses traditional histogram estimation techniques in that it finds the probability of *multiple* attributes occurring in a specific order in the same string, possibly separated by gap positions. This estimation technique is in contrast to traditional histograms that are used to estimate the occurrence of a single attribute [14, 15, 25] or multiple *unordered* substrings [17].

#### 5.2.1 Description

The complex histogram is stored as a four-dimensional matrix; the first dimension corresponds to the protein id attribute, the second dimension to the start position attribute, and the third and fourth dimensions represent the same length and type attributes as in the basic histogram. Due to the large number of proteins found in protein databases and their long sequence lengths, the first two dimensions are divided into equi-width buckets to reduce space requirements. For example, in our experimental data set with 248,375 proteins and 10,288,769 segments, we use one hundred buckets each for the first, second and third dimensions and three buckets for the fourth dimension (corresponding to the three types 'e', 'h', and 'l'). Position [3][4][7][2], for example, holds the number of <h 7 7> segments whose starting position is in the range of the 4th bucket and whose protein id lies within the 3th bucket.

#### 5.2.2 Result Cardinality Estimation

In the interest of space, we explain our cardinality estimation algorithm using an example; a more detailed explanation of the algorithm is provided in the full-length version of this paper [12]. Consider the query: {<$P_1$><$P_2$>}, which has two predicates $P_1$ and $P_2$. Table 3 shows all possible arrangements for the two predicates in a histogram with three buckets for the start position ranges 0-49, 50-99 and 100-149, respectively. For simplicity we assume here that these three start position buckets correspond to the same protein id bucket. Note that the type and length attributes of the buckets shown in the table are implicitly defined based on the definition of the predicates $P_1$ and $P_2$.

The arrangements of these two predicates fall into two configurations. In the first configuration, the predicates match segments in distinct start position buckets. For the two-predicate example, cases 1-3 show all possible arrangements in this configuration. In the second configuration, corresponding to cases 4-6 in Table 3, both predicates match segments in the same bucket.

We now need formulas to estimate the number of matches in each of these cases. Once we have these formulas, the result cardinality is the sum of the estimates from each of the cases. The result selectivity follows by dividing the cardinality by the total number of proteins in the database. We next present the estimations for cases in both these configurations. In the proceeding discussions we will refer to these configurations as *distinct bucket* and *same bucket* configurations.

| | $B_1$ (0-49) | $B_2$ (50-99) | $B_3$ (100-149) |
|---|---|---|---|
| 1 | $P_1$ | $P_2$ | |
| 2 | $P_1$ | | $P_2$ |
| 3 | | $P_1$ | $P_2$ |
| 4 | $P_1$ & $P_2$ | | |
| 5 | | $P_1$ & $P_2$ | |
| 6 | | | $P_1$ & $P_2$ |

**Table 3: Arrangement Possibilities for Two Query Predicates in Three Start Position Buckets**

The calculations for both types of configurations are performed with the assumption that the segments are uniformly distributed throughout the protein id and start position buckets. The *distinct bucket* configuration estimate is calculated by multiplying the number of matching first-predicate segments found in the first start position bucket by the number of second-predicate matches found in the second bucket divided by the number of proteins ids in each protein id bucket. The division operation is necessary because of the uniform distribution assumption. This formula can be generalized to estimate the number of results from *n* predicates in *n* distinct start position buckets and can also incorporate gap information to automatically disregard start position buckets that do not satisfy the gap requirements. For brevity, exact details of the algorithm are omitted here but are presented in [12].

The calculations for the *same bucket* configuration are more complex. When $P_1$ and $P_2$ are in the same bucket, $P_1$'s start position could be anywhere within the range of that bucket. We assume a uniform distribution of the start positions of the two predicates. For each possible first-predicate start position, we calculate the chances of the second predicate being in the proceeding start positions and in the same protein. For example, in case 4, the number of proteins that match $P_1$ at position 9 is $n_{p1} = (1/50) * (number\ of\ P_1\ in\ B_1)$. Similarly, the number of proteins that match $P_2$ in positions 10 to 49 is $n_{p2} = (4/5) * (number\ of\ P_2\ in\ B_1)$. Now, assuming that there are one hundred proteins in each protein id bucket, the estimated number of proteins that match the query in start position 9 for the given protein id bucket is: $(n_{p1} * n_{p2})/100$. To get the total estimate for the bucket $B_1$ we integrate over all the possible start positions. In our actual estimates we also factor the lengths of the predicates into the analysis. In the interest of space the exact details of this calculation are omitted here; see the full-length version [12].

### 5.2.3 Histogram Analysis

Next we examine the accuracy of the complex histogram as well as its space and time efficiency. Figure 3 tests the accuracy of these complex histogram estimates by comparing the actual number of proteins that match a given query with the estimated number. The query tested is a three-predicate query in which the gap, or middle predicate, is varied to produce different result selectivities. The results from the data set of 248,375 proteins show that the histogram estimates are accurate to within approximately 80% of the actual result size. This degree of accuracy is sufficient for the optimizer's needs, as only a general idea of the result selectivity is required by the cost functions.

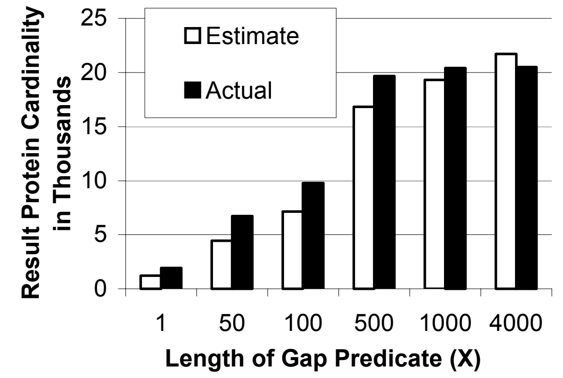

**Figure 3: Complex Histogram Accuracy, Three-Predicate Query: {<l 15 15><? 0 X><h 24 24>}, Varied Gap Predicate**

Another consideration to take into account is the time required to compute these estimates. The number of calculations performed is factorial in the number of predicates and start position buckets, and the estimation time should reflect that. We tested the estimation times of various queries with different numbers of predicates and discovered that indeed, the estimation time requirements dramatically increased with the number of query predicates. We also noticed that adding more predicates does not significantly improve the accuracy found by only using two of a query's predicates. Thus, based on this empirical evidence, in our implementation we only look at the two or three most highly selective query predicates for estimation purposes. We choose these predicates because they have the greatest impact on the reduction of the query result space. In the experiment shown in Figure 3, the estimation time is around 20 milliseconds.

In the current implementation we create the complex histogram immediately following the loading of the segment table. The complex histogram takes 22 seconds to build and requires 5.8 MB of disk space, which is only 1% of the size of the segment table.

## 5.3 Cost Formulas

We use cost formulas to model the I/O time and CPU resources needed for each evaluation method for a given query. The underlying functionalities of each of the methods are similar and use a number of "basic blocks" including index scans, table retrievals, and finite state machine matchings. We developed cost models, which are along the lines of the cost models in [25], for each of these basic blocks. These models are then incorporated into the individual cost models for the various algorithms. Histograms are used to estimate the query segment selectivities and the result protein selectivity. Standard statistics such as table cardinalities and tuple sizes are maintained and used in the cost model. In addition, a number of system-dependent "fixed" constants such as page size, maximum index fanout, and weighted I/O and CPU costs are used. A more thorough examination of the query plan cost functions may be found in the full-length version of this paper [12].

The actual query optimization process happens as follows. First, the simple histogram is used to determine the segment selectivities of all the non-gap predicates in the query and the complex histogram is used to calculate the result protein selectivity. These results are input into the different cost formulas along with the table and index information. Then, the optimizer evaluates these cost formulas for the CSP, SSS, and ISS plans, as well as for each of the MISS($n$) plans. Finally, the plan with the lowest cost formula is returned as the optimal plan and the system uses this method to evaluate the query.

## 6. Experimental Evaluation

In this section we evaluate the algorithms presented in Section 4 using both a commercial ORDBMS and our native system, Periscope. The experiments presented compare the performance of these algorithms and also show the different effects that the segment and result selectivities have on these algorithms. We also used many of these same experiments to tune the cost models in Periscope's optimizer. Consequently, for all the experiments presented in this section, the Periscope optimizer always picks the cheapest plan. A detailed validation of the optimizer cost models is beyond the scope of this paper and will be addressed in the future.

### 6.1 Experimental Setup

We implemented our query evaluation techniques in two different database systems, a commercial ORDBMS and Periscope, our own system built on top of the SHORE storage manager from the University of Wisconsin [6]. SHORE provides various storage manager facilities including file and index management, buffer pool management, concurrency control, and transaction management. The commercial system runs on Windows; Periscope can run on either Linux or Windows. To compare the performance of the ORDBMS with Periscope we used a machine with an 850 MHz Intel Pentium III processor running Microsoft Windows 2000 Professional and configured with 128 MB of memory and a 10 GB IBM DJSA-210 IDE disk. For all other tests we used a Linux 2.4.13 machine with 896 MB of memory, a 1.70 GHz Intel Xeon processor, and a Fujitsu MAN3367MP hard drive with a SCSI interface and a 40 GB capacity. In both configurations SHORE is compiled for a 16 KB page size, and the buffer pool size is set to 64 MB. The numbers presented in this study are cold numbers, i.e. the queries do not have any pages cached in the buffer pool from a previous run of the system. Each of the experimental queries is run five times and the average of the middle three execution times is presented in the following graphs.

#### 6.1.1 System Implementations

In this section we describe the specifics of both the Periscope and the ORDBMS implementations. Both systems contain the same tables, indices, and schema information: a protein table, a B+-tree index on the protein id attribute, a segment table, and a B+-tree index on the type and length segment table attributes.

For the commercial ORDBMS we utilized its type-extensibility mechanism to create an array-like user defined type to support the primary structure, secondary structure, and probability protein table fields. In addition, we created a user-defined function labelled as *Comm-CSP* to implement the protein table scan technique.

The segmentation approach is implemented in the ORDBMS using the composite B+-tree index on the type and length attributes of the segment table. Translation from our queries to SQL is accomplished by using a number of selection predicates in the SQL query to ensure that the ordering constraints are satisfied and that the resulting segments are from the same protein (see Section 4.2 for an example). After loading the tables we update all the catalog statistics so that the ORDBMS's query optimizer has the most up-to-date statistical information. We let the built-in query optimizer pick the best plan and in the following graph label this approach as *Comm-Seg*.

As the Periscope system is a native system, it gives us the flexibility of writing our own operators. We implemented the four query evaluation techniques that are described in Section 4. In the following experimental sections, the abbreviations CSP, SSS, ISS, and MISS are all implicitly understood to be implementations of these algorithms in the Periscope system. When appropriate, the MISS plan will be shown for all possible numbers of query predicates, from two to the total number of predicates in the query. This will be denoted by MISS($n$), and the number of predicates used in the individual MISS plans will be referred to as the MISS number.

#### 6.1.2 Data Set

To produce a data set for our experiments, we first downloaded the entire PIR-International Protein Sequence Database. This database is a comprehensive, non-redundant protein database in the public domain and is extensively cross-referenced [11]. Since the PIR data set only contains primary protein structures, we then used the Predator tool [10] to obtain predicted secondary structures. The final data set consists of 248,375 proteins.

Each protein has approximately 41 segments, which results in 10,288,769 segments. The Periscope protein and segment tables are 259 and 355 MB in size, respectively, while in the commercial system the protein table is 390 MB and the segment table is 425 MB.

### 6.1.3 Queries

At this time we would like to discuss the intuition behind the queries that we use in our experimental evaluation. We found it surprisingly hard to find actual queries, as queries from past studies are usually not well documented and queries in current experiments are considered to be confidential because they could reveal a great deal about the actual experiment.

Looking back, we do not consider the lack of scientific queries a drawback. Our goal is to build a system that is efficient for any type of query, and the ability to design our own queries allows us systematically explore the entire search space. Hence we pick queries based on the actual data set, i.e. we do not try random queries that may have zero matches. This coarsely models reality as scientific query proteins are generally similar to actual proteins and will usually find at least a few matches.

In our exploration of the search space we will look at queries with both single and multiple predicates, with varying query segment selectivities, and with varying result protein selectivities. The complexity of our queries makes it difficult to arbitrarily change the result selectivity; we accomplish this in our system by widening or lessening the gap predicates between actual query predicates to return greater or fewer results.

## 6.2 Comparison with Commercial ORDBMS

The first experiment tests the simplest type of query, a single-predicate exact match query[1]. In this test the segment selectivity of the single predicate varies from 0.03% to 7%. Results are shown in Figure 4 for the Periscope CSP, SSS, and ISS methods as well as for the commercial system's Comm-CSP and Comm-Seg methods. Note that the MISS method is absent in this test simply because it reduces to the ISS plan for single-predicate queries.

This test shows that the Periscope methods outperform both the commercial methods. The execution time for the CSP consistently requires one-third of the time of the Comm-CSP, while the Comm-Seg method performs increasingly worse as the segment selectivity increases. With the large execution time scale it is hard to distinguish between the Periscope methods; the following experiments will provide more conclusive results.

[1] In the Comm-CSP method for this commercial ORDBMS, only one match is returned per protein; therefore, for this experiment only we modified the CSP method to also return only one match per protein.

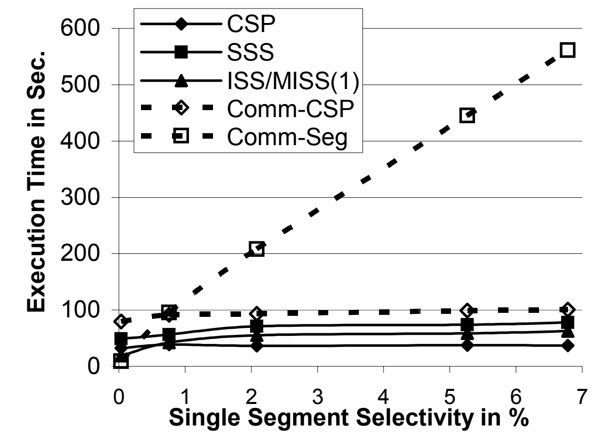

**Figure 4: Single-Predicate Exact Match Queries, Commercial ORDBMS & Periscope Results**

Additional experiments with the Comm-Seg and Comm-CSP methods involving more complex multiple-predicate range and gap queries exhibit the same result trends witnessed in Figure 4 and are more expensive relative to Periscope query execution times. In addition, in the commercial ORDBMS the choice between the Seg and the CSP plans has to be made explicitly by the user, using different SQL queries. Furthermore, to investigate the performance differences across the different query evaluation algorithms and to use the query optimizer, we need control over the choice of the query plan that is used by the database system. This control is easily available to us in Periscope, and for the rest of this study we only concentrate on the Periscope methods. Note, however, that our results are applicable to commercial ORDBMSs with appropriate modifications to the query optimizer and operator algorithms, which could be implemented by ORDBMS vendors.

## 6.3 Single-Predicate Queries

The next experiment tests the performance of single-predicate queries involving both exact match and range predicates. In this test the segment selectivity of the single predicate varies from 0.04% to 17%. Results for this experiment are shown in Figure 5 for the SSS, ISS, and CSP methods (MISS reduces to ISS as above).

This test shows that both the SSS and ISS methods outperform the CSP method regardless of the segment selectivity. This is because the CSP has to retrieve and perform a complex scan on each protein to find matches, whereas the other two methods only have to scan the segment table or probe the segment index to retrieve matching segments. The final protein id INLJ that is necessary in the SSS and ISS methods does not contribute significantly to the overall execution time because the number of proteins to retrieve has been drastically reduced. The ISS plan outperforms the SSS plan for predicates with selectivity less than 10%. For less selective predicates (those with higher predicate selectivity values), however, the SSS plan becomes more

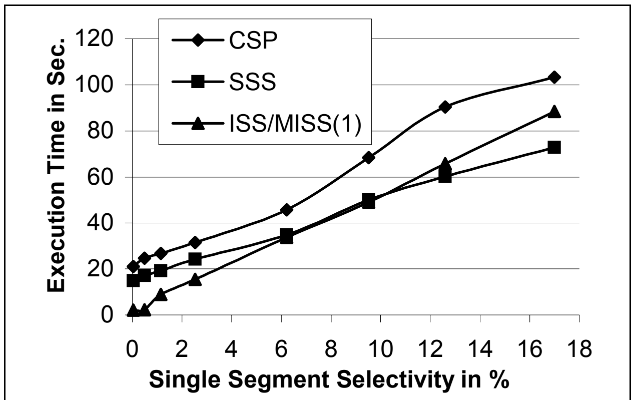

**Figure 5: Single-Predicate Queries, Varied Segment Selectivity**

efficient than the ISS method. This goes along with the rule of thumb commonly used in standard DBMSs that predicates with selectivity greater than 10% should no longer utilize B+-tree indices, but instead should be evaluated with simple table scans [7].

## 6.4 Multiple-Predicate Range and Gap Queries

This set of experiments test more complex queries involving multiple predicates with ranges and gaps. We tested the algorithms with a variety of complex queries, however in the interest of space we only present a few representative experimental results here.

### 6.4.1 A Complex Query

In this experiment we use a query with nine predicates, in which both the result protein selectivity and the various segment selectivities stay constant. The variable in this experiment is the ordering of the nine query predicates. There are five non-gap predicates, four of which have a segment selectivity of less than .03% (S) and one of which has a segment selectivity of 7% (L). The result protein selectivity is fixed at less than .1% by varying the four gap predicates, which are inserted between every two non-gap predicates. Figure 6 shows the results of this experiment in which the position of the larger query predicate varies from last in the query to first.

The results show that the CSP method is the only method that varies widely depending on the position of the large predicate. This implies that the execution time of the CSP method is very sensitive to the selectivity of the first predicate. Due to the nature of the FSM matching algorithm, queries in which the first predicate matches a large number of segments (like the L predicate) require the FSM to do more work. Because the leading predicate matches often, the number of times that the FSM tries to match the subsequent predicates increases, which in turn leads to longer CSP execution times.

This test also highlights the importance of the MISS number on the performance of the MISS method. For MISS(2-4) the index is scanned for various subsets of the four most highly selective predicates, which in this test

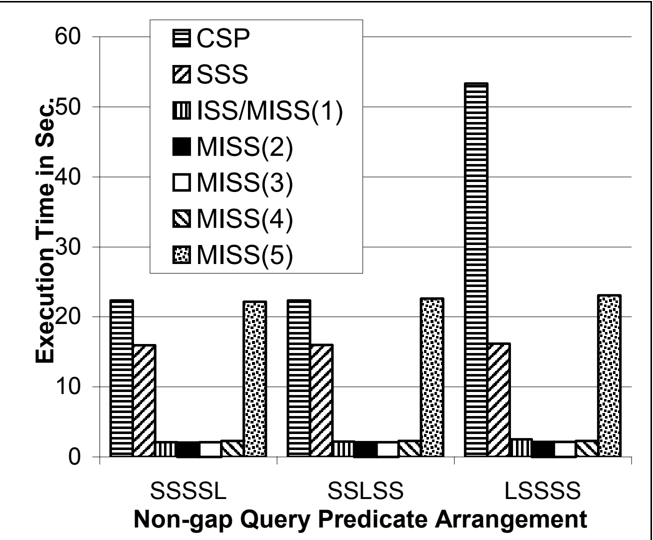

**Figure 6: Nine-Predicate Queries, S Seg. Sel. = ~.03%, L Seg. Sel. = 7%, Fixed Result Sel. < .1%**

are all very selective. In MISS(5), however, the index is also scanned for the larger (less selective) predicate. This adds considerable length to the execution time (recall that the MISS algorithm picks predicates based on their selectivities and not their physical order in the query).

The MISS number, in general, is dependent on the segment selectivities and the final protein selectivity. The MISS plan performs a number of index probes, which reduces the number of proteins to be retrieved and scanned. There is a balance between the costs incurred from performing these probes and the costs saved by the reduced number of proteins that must be retrieved. This balance is also influenced by the result protein selectivity in that the time required to perform a FSM scan of each protein is affected by the result selectivity (we explore this effect in the next set of experiments). The cost of adding another query predicate to the MISS(*k*) plan is the sum of the time to scan the segment index for the $k+1^{th}$ predicate, the time to sort the results by protein id and start position, and the time to add these results to the segment merge join. Evaluating the $k+1^{th}$ predicate, however, will further cut down on the number of protein ids that emerge from the merge join, which in turn reduces the number of protein tuples that have to be retrieved. The reduction factor is roughly inversely proportional to the selectivity value of the added predicate. The time saved is the sum of the times to probe the id index for the eliminated proteins, retrieve them, and perform their complex scans. When this time saved is greater than the time incurred by adding the $k+1^{th}$ predicate, the MISS number should increase to $k+1$; otherwise it is more efficient to remain at *k*.

Another important point to notice in Figure 6 is that in many cases the optimal MISS method is **an order of magnitude faster** than the CSP method! This experiment demonstrates that having flexible query plans that adapt

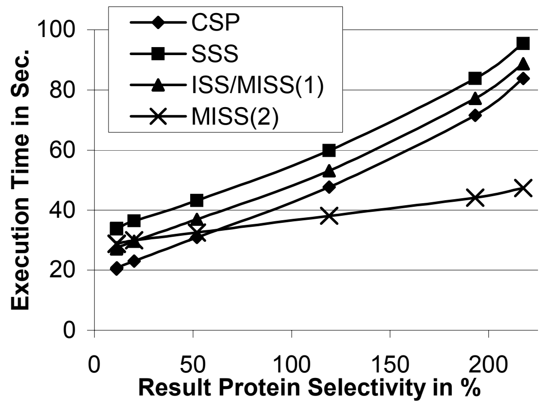

**Figure 7: Three-Predicate Query, $P_1$ Sel. = 3.73%, $P_2$ Sel. = 3.33%, Varied Result Selectivity**

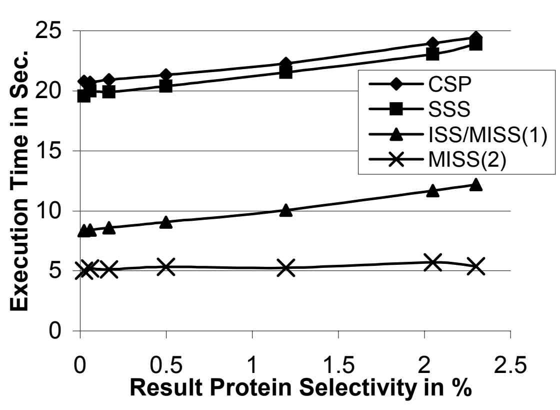

**Figure 8: Three-Predicate Query, $P_1$ Sel. = .384%, $P_2$ Sel. = .385%, Varied Result Selectivity**

to query characteristics can significantly improve query response times.

### 6.4.2 Effects of Segment and Protein Selectivities

In this experiment we use four three-predicate queries to demonstrate the effects of segment and protein result selectivities on the performance of the four algorithms. The same results hold for queries with greater than three predicates; examples of more experiments of this nature can be found in the full-length version of this paper [12].

The three-predicate queries we investigate here have fixed segment selectivities and varying result selectivities. The result selectivity is varied by modifying the middle, or gap, predicate. We increase the range of this gap predicate until the upper bound reaches ∞. In the first test the two non-gap query predicates have relatively high segment selectivities; in the second they have low segment selectivity values. The third tests a predicate with a high selectivity followed by a predicate with a low selectivity; the fourth reverses the order of these two predicates. The results of this experiment are shown in Figures 7, 8, 9, and 10, respectively.

Figure 7 contains the results of the query with two non-selective predicates separated by a gap predicate. Note that in this figure the result protein selectivity increases beyond 100% as some proteins match in multiple positions. It is interesting to note that the CSP plan outperforms the other methods when the result protein selectivity is less than 50% even though the selectivity of the first predicate is relatively high. This is because neither of the query predicate selectivity values are low enough to justify doing a table or index scan to reduce the number of proteins that have to be examined; it is faster to simply perform the complex scan of the entire protein table. When the result selectivity increases beyond 50%, however, the situation changes. The cost of performing a complex scan on a protein rises due to the increased number of matches, which causes the FSM to perform more comparisons. Consequently, it is more time-effective to probe the segment index for both the query predicates and merge the results. Only a subset of the proteins then needs to be retrieved and none need to be scanned; therefore, MISS(2) becomes the most efficient method. The SSS and ISS methods still require complex scans of the resulting proteins, and the subset of proteins retrieved is not sufficiently reduced due to the high selectivity of the most highly selective predicate. These factors, along with the time required to perform the segment table scan or index probe, account for the poor performance of the SSS and ISS methods.

The results of this experiment performed with two highly selective predicates are shown in Figure 8. Even though the result protein selectivity is small for these query predicates, the MISS(2) method still outperforms the other three. The two index probes are fast and do not return many results; consequently the merging phase and protein retrieval are very efficient. The ISS method is the next best query plan for this query. The single index probe is performed quickly with only a few results. The main time factor in this method is that the resulting proteins must be scanned with the FSM. The SSS plan shows the same trends as the ISS method; the difference in execution time is based solely on the time required for the segment table scan versus the segment index probe.

Figure 9 gives the results of a query with a non-selective predicate followed by a more selective predicate (lower selectivity value). The most noticeable result is that of the CSP method, whose performance degrades rapidly due to the high selectivity value of the first predicate. On the other end of the spectrum, the ISS method is initially the most efficient plan for this query. This is because the most highly selective predicate has a small selectivity value. Therefore the index probe takes a short amount of time and drastically reduces the number of proteins that must be scanned. The SSS method again exhibits the same characteristics as the ISS method, with the only difference being the longer time needed to scan the segment table than to probe the segment index. The MISS(2) plan remains consistent throughout and as the result protein selectivity increases, replaces ISS as the

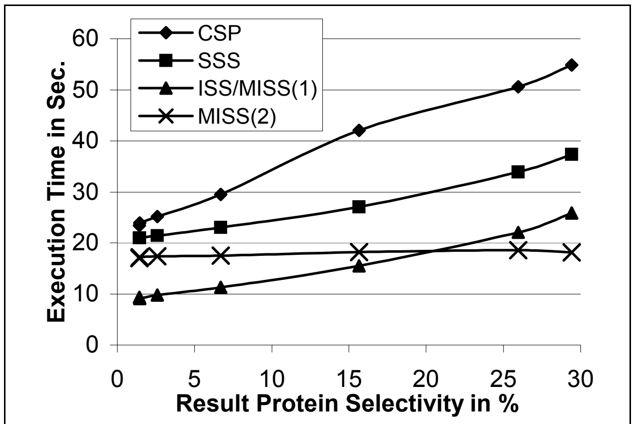

**Figure 9: Three-Predicate Query, $P_1$ Sel. = 4.25%, $P_2$ Sel. = .385%, Varied Result Selectivity**

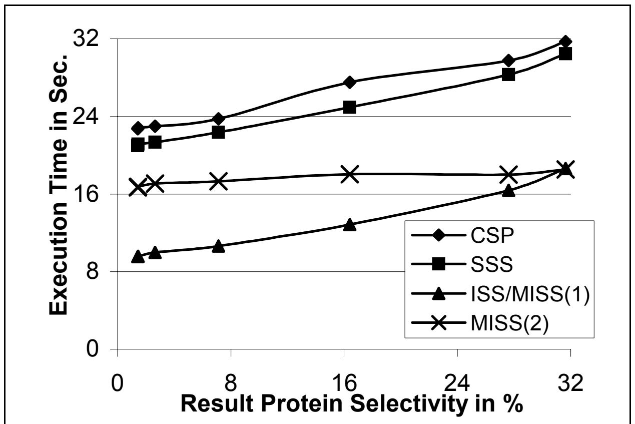

**Figure 10: Three-Predicate Query, $P_1$ Sel. = .385%, $P_2$ Sel. = 4.25%, Varied Result Selectivity**

most efficient method. Again, this is due to the fact that as the protein selectivity increases, the cost of scanning a protein also increases. Because the MISS(2) plan does not have to perform complex scans, it becomes more efficient. Initially, however, MISS(2) performs worse than ISS because it performs index scans not only for the highly selective third predicate but also for the less selective first predicate, which takes longer than its potential savings.

The final experiment reverses the order of the query predicates so that the most highly selective predicate occurs first, followed by the less selective predicate; results are found in Figure 10. The different query evaluation methods perform the same as in the previous experiment with respect to each other with ISS being the clear initial winner. The difference in these results is the scale of the execution time. The CSP method performs much faster than in the previous experiment due to the fact that the first predicate's selectivity is small; this also reduces the time for SSS and ISS as their performance is partially dependent on the time to scan a protein. The MISS(2) plan's execution time remains the same because it performs the same index probes in both tests. As the protein selectivity increases past the data points shown in Figure 10, it appears that the MISS(2) method will again outperform the ISS method by the same argument as in the previous experiment.

*As a final note we observe that our Periscope implementation is extremely efficient and returns results in a few tens of seconds for the 600 MB data set that we have used! This fast query response time is very desirable especially when scientists want to analyze data by posing a number of successive queries and refining these queries as they learn from the results of the previous query.*

## 7. Conclusions and Future Work

The secondary structure of proteins plays an important role in determining their function. Consequently, tools for querying the secondary structure of proteins are invaluable in the study of proteomics. This paper addresses the problem of efficient and declarative querying of the secondary structure of protein data sets.

Our contributions include defining an expressive and intuitive query language for secondary structure querying and identifying various algorithms for query evaluation. To help a query optimizer pick amongst the various algorithms, we have also developed novel histogram techniques to determine segment and result selectivites.

We have implemented and evaluated the proposed techniques in a native DBMS we have developed called Periscope. We have compared the performance of Periscope with a commercial ORDBMS and have shown that for the class of queries that we are considering, Periscope provides an extremely efficient implementation. As the experimental results show, the system that we have developed can query large protein databases efficiently, allowing scientists to interactively pose queries even on large data sets.

There are a number of directions for our future work, including developing algorithms to produce results in some ranked order. We would like to design a framework such that the ranking metric can be easily customized by the user, as the model for ranking proteins is usually not fixed but instead varies across scientists and may also change frequently during the course of an experiment. The ranking metric may take into account additional information that is present in the protein, such as the positional probability in the secondary structure, which is currently one of the fields produced as output by protein structure predication tools. Techniques that have been developed for ranking results in other contexts may be applicable here [8, 9, 18, 21].

Search engines for querying biological data sets often employ a query-by-example interface. In BLAST, one of the most popular tools for searching genes and the primary structure of proteins, the system is presented with a query sequence and the search engine finds the best matches to this sequence [1, 3, 4]. The input sequence is converted into a set of segments, and segment-matching techniques are employed to evaluate the query. While our work presented in this paper focuses on such segment-matching techniques for querying on the secondary structure of proteins, we would also like to explore the use of a query-by-example interface for our current system. Query-by-example interfaces require additional input that allows the user to influence the mapping of the query into segments to be matched. This additional input can be fairly complex; as an example the user may be allowed to specify a scoring matrix to assign weights to different portions of the input query. The "right" interface for specifying this mapping model can vary between users, and designing an interface that is both intuitive and easily-specified is a challenge that we hope to undertake as part of our future work.

Experiments in the life sciences often involve querying a number of biological data sets in a variety of different ways. Ideally, a *combination* of both primary sequence and secondary structure searches will lead to more accurate protein function discovery [22]. This paper only addresses the issue of efficient query processing techniques for secondary structure. Hence the tool that we have built would be an addition to the suite of biological querying tools that exist today. Managing data that is related to the entire experiment, including queries using a number of different tools on a number of different data sets, is in itself an interesting database problem and is part of the long-term goal of the Periscope project.

## Acknowledgements

This research has been supported by a fellowship from the University of Michigan and by a gift donation from Eli Lilly. We would like to thank Jack Dixon, Don Huddler, and Jeanne Stuckey at the Univ. of Michigan's Dept. of Biological Chemistry, and Ernst Dow of Eli Lilly, for their helpful discussions about the data analysis needs of the scientific community. Much of our work presented in this paper stems from our discussions with these groups.

## References

[1] www.ncbi.nlm.nih.gov/BLAST/.

[2] B. Alberts, D. Bray, J. Lewis, M. Raff, K. Roberts, and J. D. Watson. *Molecular Biology of the Cell*, 3rd ed. Garland Publishing, Inc., 1994.

[3] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic Local Alignment Search Tool. *J. Molecular Biology*, 215: 403-410, 1990.

[4] S. F. Altschul, T. L. Madden, A. A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped BLAST and PSI-BLAST: A New Generation of Protein Database Search Programs. *Nucleic Acids Research*, 25(17), 1997.

[5] M. M. Astrahan, et al. System R: A Relational Approach to Database Management. In *ACM Transactions on Database Systems*, 1(3): 97-137, 1976.

[6] M. J. Carey, et. al. Shoring up persistent applications. In *SIGMOD*, 383-394, 1994.

[7] D. DeWitt, The Wisconsin Benchmark: Past, Present and Future. *The Benchmark Handbook for Database and Transaction Processing Systems*, J. Gray, ed., Morgan Kaufman Pub., 1991.

[8] R. Fagin. Combining Fuzzy Information from Multiple Systems. In *PODS*, 216-226, 1996.

[9] R. Fagin, A. Lotem, and M. Naor. Optimal Aggregation Algorithms for Middleware. In *PODS*, 102-113, 2001.

[10] D. Frishman, P. Argos. Incorporation of Non-local Interactions in Protein Secondary Structure Prediction from the Amino Acid Sequence. In *Protein Engineering.*, 9(2): 133-142, 1996.

[11] D. George, W. Barker, H. Mewes, F. Pfeiffer, and A. Tsugita. The PIR-International Protein Sequence Database. *Nucleic Acids Research*, 24: 17-20, 1996.

[12] L. Hammel, J. M. Patel. Searching on the Secondary Structure of Protein Sequences. Technical Report, University of Michigan, 2002.

[13] E. Hunt, M. P. Atkinson, and R. W. Irving. A Database Index to Large Biological Sequences. In *VLDB*, 2001.

[14] Y. E. Ioannidis. Universality of Serial Histograms. In *VLDB*, 256-267, 1993.

[15] Y. E. Ioannidis, V. Poosala. Balancing Histogram Optimality and Practicality for Query Result Size Estimation. In *SIGMOD*, 233-244, 1995.

[16] R. M. Jackson, R. B. Russell. The Serine Protease Inhibitor Canonical Loop Conformation: Examples Found in Extracellular Hydrolases, Toxins, Cytokines, and Viral Proteins. *J. Molecular Biology*, 296(2): 325-334, 2000.

[17] H. V. Jagadish, O. Kapitskaia, R. T. Ng, and D. Srivastava. Multi-dimensional Substring Selectivity Estimation. In *VLDB*, 1999.

[18] T. Kahveci, A. K. Singh. An Efficient Index Structure for String Databases. In *VLDB*, 2001.

[19] F. Moussouni, N. W. Paton, A. Hayes, S. Oliver, C. A. Goble, and A. Brass. Database Challenges for Genome Information in the Post Sequencing Phase. In *DEXA*, 1999.

[20] M. Muralikrishna, D. J. DeWitt. Equi-Depth Histograms For Estimating Selectivity Factors For Multi-Dimensional Queries. In *SIGMOD*, 28-36, 1988.

[21] A. Natsev, Y. Chang, J. R. Smith, C. Li, and J. S. Vitter. Supporting Incremental Join Queries on Ranked Inputs. In *VLDB*, 2001.

[22] C. A. Orengo, A. E. Todd, and J. M. Thornton. From Protein Structure To Function. In *Current Opinion in Structural Biology*, 9: 374-382, 1999.

[23] S. Park, D. Lee, and W. W. Chu. Fast Retrieval of Similar Subsequences in Long Sequence Databases. In *KDEX*, 1999.

[24] W. K. Purves, G. H. Orians, and H. C. Heller. *Life, the Science of Biology*, 4th ed. Sinauer Associates, Inc., 1995.

[25] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access Path Selection in a Relational Database Management System. In *SIGMOD*, 23-34, 1979.

[26] M. Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.

[27] H. E. Williams, J. Zobel. Indexing and Retrieval for Genomic Databases. In *IEEE Transactions on Knowledge and Data Engineering*, 14(1): 63-78, 2002.

[28] Z. Zhang, A. A. Schaffer, W. Miller, T. L. Madden, D. J. Lipman, E. V. Koonin, and S. F. Altschul. Protein Sequence Similarity Searches Using Patterns As Seeds. *Nucleic Acids Research*, 26(17): 3986-3990, 1998.