Chapter 11
Declarative and Efficient Querying on Protein Secondary Structures

Jignesh M. Patel, Donald P. Huddler, and Laurie Hammel

Summary

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 datasets. This work highlights a specific problem involving searching large volumes of protein datasets based on their secondary structure. In this chapter 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 have implemented these algorithms in Periscope, which is a native database management system that we are building for declarative querying on biological datasets. Experiments based on our implementation 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 and can provide scientists with interactive secondary structure querying options even on large protein datasets.

11.1 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; analyzing large volumes of genetic and biological datasets involves being able to maintain and query large genetic and protein databases. If efficient methods are not available for retrieving these biological datasets, then unfortunately the progress of scientific analysis is encumbered by the limitations of the database system.

This chapter looks at a specific problem of this nature that involves methods for searching protein databases based on secondary structure properties. This work is a part of the Periscope project at the University of Michigan, in which we are investigating methods for declarative querying on biological datasets. In this chapter, we define a problem that the scientific community faces regarding searching on protein secondary structure, and we develop a query language and query-processing techniques to efficiently answer these queries. We have built a secondary structure querying component, called Periscope/PS^2, based on the work described in this chapter, and we also describe a few experimental and actual user experiences with this component of Periscope.

11.1.1 Biological Background

Proteins have four different levels of structural organization, primary, secondary, tertiary, and quaternary; the latter two are not considered in this chapter. 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 [321]. 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 [307]. Therefore, examining the types, lengths, and start positions of its secondary structure folds can aid scientists in determining a protein's function [10].

11.1.2 Scientific Motivation

The discovery of new proteins or new behaviors of existing proteins necessitates complex analysis in order to 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 [10]. These phases may be intertwined as the analysis of matches may provide interesting results that could be further explored using more refined searches.

This simplification of the searching process glosses over the actual definition of protein similarity. The reason 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 primary structure matching is beneficial, while another may be interested in finding secondary structure similarities in order to predict biomolecular interactions [196]. 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* regarding posing queries and *efficient* regarding 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 [12, 13, 218, 429, 454], to the best of our knowledge there are no tools available for querying on the secondary structure of proteins. This chapter 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.

11.1.3 Chapter Organization

In this chapter, we first 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 selectivities and segment selectivities, the choice of the algorithm can have a dramatic impact on the performance of the query.

Next 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 depends on the accuracy of its statistics, for this application we need to accurately estimate both the segment and the 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 on biological datasets, and in this chapter, we also present results from actual uses of this search technique.

11.2 Protein Format

The first task to perform is to establish the format for representing proteins in our system. This format largely depends 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 just a predicted measure. To obtain the secondary structure for a given protein, therefore, it is usually necessary to enter its primary structure into a prediction tool that will return the protein's predicted secondary structure.

The tool used to predict the secondary structure information for the proteins in our database is Predator [134]. Predator is a secondary structure prediction tool based on the recognition of potentially hydrogen-bonded residues in a single amino acid sequence. Even though, we use this particular tool, our query-processing techniques can 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 11.1 contains a portion of a sample protein in our database.

Name:	$t2\_1296$
Id:	1
Length:	554
Primary structure:	MSAQISDSIEEKRGFFT..
Secondary structure:	LLLLEELLLLLLLHHHH..
Probability:	99755577763445443..

Fig. 11.1. Sample protein.

11.3 Query Language and Sample Queries

Next we determine the types of queries that are useful to scientists in examining secondary structure properties and design a query language to express these queries. Based on interviews with scientists who regularly perform secondary structure protein analysis, we are able to formulate three initial classes of queries and a query language.

11.3.1 Query Language

As these queries are defined, an intuitive query language begins to emerge. Because only three types of secondary structure can occur in a protein sequence, helices (h), beta-sheets (e), and turns or loops (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 in order to satisfy the query. A quick note on terminology: throughout this chapter we refer to segment predicates either as query predicates or simply predicates. 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 11.2. We will now look at three important classes of queries that can be expressed using the language defined.

Query → Segments
Segments → Segment*
Segment → <type lb ub>
Type → e | h | l | ?
lb → any integer >= 0
ub → any integer >= 0 | ∞
Segment Constraint: lb <= ub

Fig. 11.2. Query language definition.

11.3.2 Exact Match Queries

In the simplest situations, scientists would like to find all proteins that contain an exact query sequence. An example of such a query is

find all proteins that contain 'hhheeeelll', or 3 h's followed by 4 e's followed by 3 l's

The user would express this query as a sequence of three predicates: $\{< h33 >< e44 >< l33 >\}$. Our algorithms take the exactness of these predicates

literally; they do not return matches that are a part of a larger sequence. For example, the sequence hhhheeeelll would not be returned as a match for the example query because it contains four h's at the beginning of the sequence, not three as specified in the first query predicate.

11.3.3 Range Queries

While exact matching is important, in some cases it may be sufficient to find matches of approximate length. For example, one might want to pose the following query:

> *find all proteins that contain a loop of length 4 to 8 followed by a helix of length 7 to 10*

Here the exact position where the amino acids stop looping and start to form a helix is not as important as the fact that there is a loop followed by a helix. This range query would be expressed as {<l 4 8><h 7 10>}. Note that this example provides the motivation for expressing both the upper and lower bounds of the segment length.

11.3.4 Gap Queries

Another feature scientists would like to be able to express in their queries is the existence of gaps between regions of importance. For example, one scientist may be interested in matching two portions of a query protein exactly but may not care if the connecting positions hold any similarity. One of the big drawbacks of current primary structure search tools is that there is no effective way to specify gaps in the query sequence. Using our query language, these gaps can easily be expressed. The query sequence {<h 4 6><? 0 ∞ ><l 5 5>} would solve the gap query:

> *find 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.

11.4 Query Evaluation Techniques

This section describes four methods for evaluating the types of queries just defined. The first approach uses a protein scan and the last three utilize a segmentation technique similar to that described in section 11.3 that represents proteins as sequences of segments. Section 11.5 will provide details about a statistics-based query optimizer that chooses between these methods.

11.4.1 Complex Scan of Protein Table (CSP)

The first approach performs a scan of the protein table itself. Before proceeding with the details of the scan, we will first describe the schema that is used to store proteins in the protein table. The schema for the protein table is

```
table   protTbl (
        name                char(30),
        id                  int primary key,
        length              int,
        primary_structure   char(length),
        secondary_structure char(length),
        probability         int(length));
```

The descriptions for these fields are the same as in Figure 11.1. The latter three fields are character or integer arrays, where *length* is the length of the protein in amino acids.

The general plan for the scan is that 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 is performed using a nondeterministic finite state machine (FSM) technique similar to that used in regular expression matching [371]. 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.

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 positions, each of these places must be reported separately. To achieve this result, our algorithm checks for all possible occurrences in each protein by running the FSM matching test once for each position in the protein's secondary structure.

The FSM itself is constructed once for each query. In our algorithm the FSM consists of a lookup table with next-state assignments for the three possible types of inputs as well as information regarding the final, or matching, and exiting, or nonmatching, states. This complex scan of the protein table is able to solve any of the three types of queries described in section 11.3 containing any number of predicates.

11.4.2 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 [312] in which the interest is 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. The corresponding segment table schema, then, is

```
table   segTbl (
        segment_id          int primary key,
        id                  int,
        type                char,
        length              int,
        start_position      int,
        foreign key (id)    references protTbl (id));
```

A multiattribute 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. Tables 11.1 and 11.2 show an example of several small protein entries with their corresponding segment tuples.

Table 11.1. Sample protein table.

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

Table 11.2. Sample segment 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

The remaining three query evaluation techniques all incorporate some variation on the following plan description to produce proteins that satisfy a given query. In general, each nongap predicate of a query can be evaluated using either a scan of the segment table or a probe of the segment index. Once individual matching segments of the query have been retrieved, they can be merged based on their protein id; the start position information can

then be used to satisfy the ordering constraints between segments to produce the 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 through 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. We now present an example of each of the three types of queries in our query language along with their corresponding SQL translations. The exact match query {<e 8 8><h 6 6>} is expressed in SQL as

```
select * from protTbl p, segTbl s1, segTbl s2
where s1.type = 'e' and s1.length = 8
and s2.type = 'h' and s2.length = 6
and s1.id = s2.id and s1.id = p.id
and s2.start_pos - (s1.start_pos + s1.length) = 0
```

Note that the start position information is utilized to account for the predicate ordering. The range query {<e 8 10><l 6 7>} is essentially the same as the exact match query only with range bounds on the length constraints instead of single equality bounds, as shown here:

```
select * from protTbl p, segTbl s1, segTbl s2
where s1.type = 'e' and s1.length BETWEEN 8 and 10
and s2.type = 'l' and s2.length BETWEEN 6 and 7
and s1.id = s2.id and s1.id = p.id
and s2.start_pos - (s1.start_pos + s1.length) = 0
```

Queries involving gaps become a little more complicated because the start position information must also be utilized to account for the gap constraints as well as the predicate ordering. The query {<e 8 10><? 3 5><h 2 2>} is expressed as

```
select * from protTbl p, segTbl s1, segTbl s2
where s1.type = 'e' and s1.length BETWEEN 8 and 10
and s2.type = 'h' and s2.length = 2
and s1.id = s2.id and s1.id = p.id
and s2.start_pos - (s1.start_pos + s1.length) <= 5
and s2.start_pos - (s1.start_pos + s1.length) >= 3
```

We will now describe the remaining three query evaluation techniques, which all incorporate the methodologies of the foregoing plan with some minor variations.

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 the segments returned by the scan then participate in the aforementioned INLJ to retrieve their actual proteins. It is important to note that the retrieved proteins do not yet match the query, as there may be other query predicates that have not yet been tested. If there are additional predicates in the query, each protein is scanned using the FSM technique described in section 11.4.1 to produce the final matching result.

Index scan of segment index (ISS). The index scan query plan is essentially identical to the SSS method with one exception. While the SSS method uses a scan of the segment table to produce segments matching the most highly selective predicate, the ISS method instead utilizes the B+-tree index built on the type and length attributes of the segment table to retrieve segments that match the most highly selective predicate. Once these matching segments have been found, an INLJ with the protein table id index and a possible complex scan (for queries with more than one predicate) are performed in the same fashion as in the SSS plan.

Multiple index scans of segment index (MISS(n)). The final method described in this chapter, the multiple index scan technique, is a generalization of the ISS plan. The basic change is that instead of performing only 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 nongap predicates in the query. We will refer to this value of n as the MISS number. 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.

11.5 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 the 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 [21, 356]. We also have a slightly simpler method of query optimization that uses heuristic cutoffs to determine

the most efficient query plan. Both the cost functions and heuristic cutoffs 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 [194, 195, 198, 287, 356]. 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 to determine the selectivities of the query predicates and a more complex one to estimate the resulting protein selectivity. Section 11.5.1 introduces the basic histogram and section 11.5.2 describes the complex histogram. Sections 11.5.3 and 11.5.4 explain how the query optimizer uses these histograms in the heuristic cutoff and cost function methods, respectively.

11.5.1 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 $k \times 3$ 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.[1] 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 100, 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 100 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 on each new protein addition without significant time penalty. With the protein dataset 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

[1] Note that the numbering of the rows and columns in the matrix starts from 1 instead of 0. As there is no practical reason for being able to express segments of length 0, segments of length 1 are the smallest segments we consider. We want to keep the segment length identical to the associated histogram row number, so the row numbering starts with 1; the column numbering also starts with 1 (and ends at 3) for uniformity.

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 on average. In terms of space requirements, the histogram contains information about more than 99% of all segments and occupies only 1.2 KB of disk space.

11.5.2 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 [194, 195, 356] or multiple *unordered* substrings [198].

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 simple histogram. Due to the large number of proteins found in protein databases and their long sequence lengths, the first two dimensions are divided into equal-width buckets to reduce space requirements. For example, in our experimental dataset 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 fourth starting position bucket and whose protein id lies within the third protein id bucket.

Result cardinality estimation. We will initially present our cardinality estimation algorithms using an example. Consider the query {<P_1 ><P_2 > }, which has two predicates, P_1 and P_2. Table 11.3 shows all possible arrangements for the two predicates in a histogram with three buckets (SPB_1, SPB_2, and SPB_3) 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 according to the definitions 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 with this configuration. In the second configuration, corresponding to cases 4–6 in Table 11.3, both predicates match segments in the same bucket.

Table 11.3. Arrangement possibilities for two query predicates in three start position buckets.

	SPB_1 (0–49)	**SPB_2 (50–99)**	**SPB_3 (100–149)**
1	P_1	P_2	
2	P_1		P_2
3		P_1	P_2
4	P_1 and P_2		
5		P_1 and P_2	
6			P_1 and P_2

We now need formulas to estimate the number of matches in each of these cases. Once we have these formulas, the resulting cardinality will be the sum of the estimates from each of these cases. Table 11.4 contains a description of the variables that will be used in these formulas. Figure 11.3 gives the pseudocode for the general algorithm that is used to calculate the estimated result cardinality of a query. The result selectivity follows by dividing this cardinality by the total number of proteins in the database. We next present the estimations for cases in both these configurations. In the following discussion we will refer to these configurations as *distinct bucket* and *same bucket configurations*.

Table 11.4. Description of the cardinality estimation variables.

Variables	**Description**
$NumProtIdBuckets$	Number of protein id buckets in first dimension of histogram
$NumPosBuckets$	Number of start position buckets in second dimension of histogram
$NumProtPerBucket$	Number of proteins represented by a protein id bucket
$NumPosPerBucket$	Number of start positions represented by a start position bucket
PIB_i	ith protein id bucket
SPB_i	ith start position bucket
P_1	First predicate used in the estimation
P_2	Second predicate used in the estimation
Gap	Gap between P_1 and P_2
$X.start$	Lower bound on predicate X's range where X = P_1, P_2, or Gap
$X.end$	Upper bound on predicate X's range where X = P_1, P_2, or Gap

The calculations for both types of configurations are performed with the assumption that the segments are uniformly distributed throughout

```
This function gives the general framework for estimating the cardinality of a
query using two predicates, P1 and P2, and the Gap between them.
double Estimate_Cardinality(P1, P2, Gap)
{
double card = 0;

// do for each protein id bucket in the histogram
for (int i = 1; i <= NumProtIdBuckets; i++) {

// for each possible starting position bucket, do same bucket configuration
for (int j = 1; j <= NumPosBuckets; j++) {
card += Same_Bucket(PIBi, SPBj, P1, P2, Gap);
}

// for each possible starting position, do distinct bucket configuration
for (j = 1; j < NumPosBuckets; j++) {
card += Distinct_Bucket(PIBi, SPBj, P1, P2, Gap, j);
}
}
return card;
}
```

Fig. 11.3. Cardinality algorithm pseudocode.

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 protein 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. Figure 11.4 gives the pseudocode for the distinct bucket configuration algorithm.

The calculations for the same bucket configuration are more complex. When P_1 and P_2 are in the same start position 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 $n_{p1} = (1/50) \times$ (number of P_1 in SPB_1). Similarly, the number of proteins that match P_2 in positions 10 to 49 is $n_{p2} = (4/5) \times$ (number of P_2 in SPB_1). Now, assuming that there are 100 proteins in each protein id bucket, the estimated number of proteins that match the query in start position 9 for

```
This function determines the number of proteins that for a given protein id
bucket (PIB_k) and a given start position bucket (SPB_i) contain P_1 in SPB_i
and P_2 in any subsequent start position bucket (for the same protein id bucket
PIB_k).
Distinct_Bucket(PIB_k, SPB_i, P_1, P_2, Gap, i)
{
int P_1_count = number of P_1 in PIB_k and SPB_i (from histogram);
int P_2_count;

// need to be aware of the last possible start position
// that occurs in the given position bucket (ith bucket)
int SPB_i_end = (i × NumPosPerBucket);

// also need to be aware of the first possible start position that occurs in SPB_i
int SPB_i_start = ((i - 1) × NumPosPerBucket) + 1;

// also need to calculate the first and last possible start positions in each
// of the subsequent start position buckets, SPB_j
int SPB_j_start, SPB_j_end;
double sum = 0;

// calculate for each of the following position buckets
for (int j= i+1; j < NumPosBuckets + 1; j++) {
SPB_j_end = (j × NumPosPerBucket);
SPB_j_start = ((j - 1) × NumPosPerBucket) + 1;

// take gaps and predicate lengths into account
if (SPB_j_start <= SPB_i_end + P_1.end + Gap.end) {
if (SPB_j_end >= SPB_i_start + P_1.start + Gap.start) {
sum += P_2_count (number of P_2 in PIB_k and SPB_i from histogram);
}
else { j = NumPosBuckets + 1; }
}
}
double result = (P_1_count × sum)/NumProtPerBucket
return result;
}
```

Fig. 11.4. Distinct bucket configuration algorithm pseudocode.

the given protein id bucket is $(n_{p1} \times n_{p2})/100$. To get the total estimate for the start position bucket SPB_1, we integrate all the possible start positions. In our actual estimates we also factor the lengths of the predicates into the analysis. Figure 11.5 gives the pseudocode for the same bucket configuration algorithm.

Histogram analysis. Next we examine the accuracy of the complex histogram as well as its space and time efficiency. Figure 11.6 tests the accuracy of these complex histogram estimates by comparing the actual number of proteins that match a given query with the estimated number.

```
This function determines the number of proteins that for a given protein id
bucket (PIB_k) and a given start position bucket (SPB_i) contain both P_1 and
P_2 in the correct order with the specified gap between them.
                Same_Bucket(PIB_k, SPB_i, P_1, P_2, Gap)
                                {
 // if P_1 and the Gap are longer than the bucket, then no matches will occur
                if (P_1.start + Gap.start) >= NumPosPerBucket) { return 0; }

                P_1_count = number of P_1 in PIB_k and SPB_i (from histogram);
                P_2_count = number of P_2 in PIB_k and SPB_i (from histogram);

 // P_2 cannot start before the minimum range of P_1 and Gap have occurred
                int length = P_1.start + Gap.start;
                double sum = 0;

                for (int j = 0; j < NumPosPerBucket - length; j++) {
                sum += ((NumPosPerBucket - j - length)/NumPosPerBucket)
                                × (P_2_count/NumProtPerBucket)
                                }

                double result = (sum × P_1_count)/NumPosPerBucket;
                                return result;
                                }
```

Fig. 11.5. Same bucket configuration algorithm pseudocode.

The query tested is a three-predicate query in which the gap, or middle predicate, is varied to produce different query result selectivities. The results from the dataset 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 selectivity is required by the cost functions and heuristics.

Next we analyze the time required to compute these estimates. For a histogram to be practical, this estimation time must be small. We performed this cardinality estimation on several seven-predicate queries with varying segment selectivities. Each of the queries has four nongap predicates that are alternated with three gap predicates. For each query we ran the cardinality estimation using two, three, and four of the nongap predicates and recorded the estimation time for each. For the two-predicate trials the two most highly selective predicates are used in the estimation; the same rule is applied in the three-predicate cases where the three most selective predicates are utilized. All four of the nongap predicates are used for the four-predicate trials. The data set used for this test consists of 56,000 proteins and 1,100,000 segments; the protein and segment tables were 56 and 66 MB in size, respectively. Figure 11.7 contains the execution time results of these cardinality estimations.

The results in Figure 11.7 show that the estimation time is small when the two or three most highly selective predicates are used to calculate

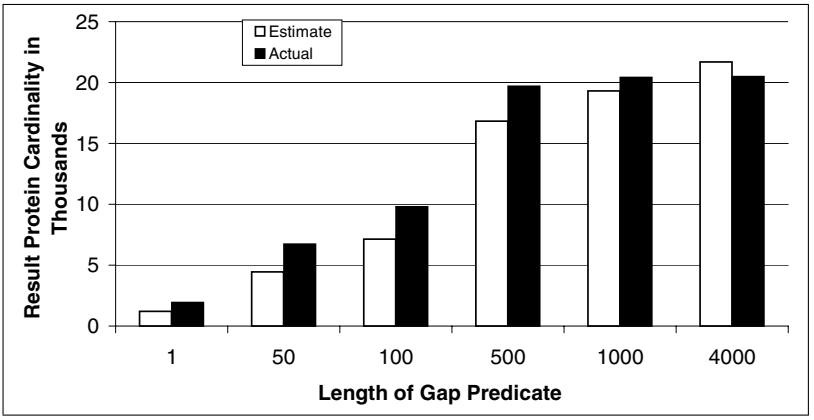

Fig. 11.6. Complex histogram accuracy, three-predicate query - {<l 15 15><? 0 X><h 24 24>}, varied gap predicate.

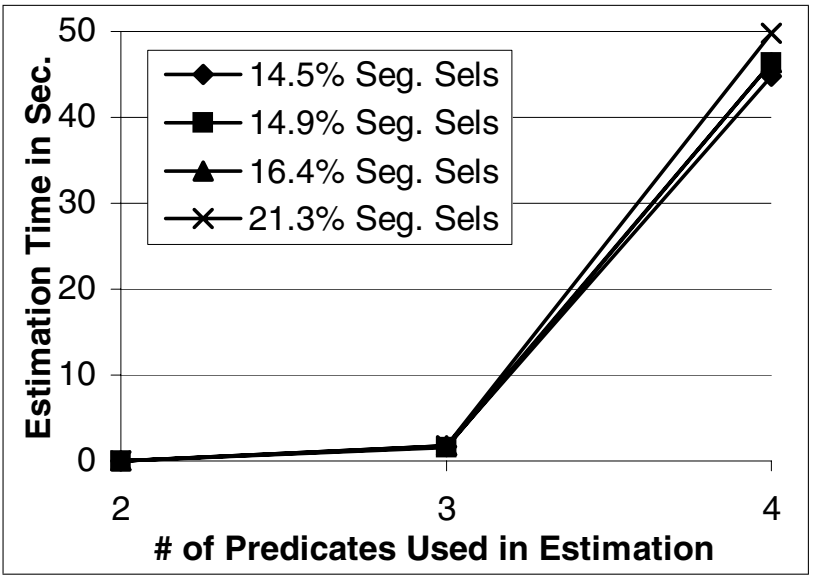

Fig. 11.7. Cardinality estimation time for seven-predicate queries with varied segment selectivities.

the estimations. With four-predicate estimations, however, the calculation time will probably surpass the query evaluation time, as validated in the experimental section because the number of calculations required is factorial in the number of query predicates and start position buckets. We also find in analyzing the accuracy of the estimations that adding the third and fourth

predicates to the estimation does not significantly improve the accuracy found by only using two query predicates.

Thus, based on this empirical evidence, in our implementation we look only at the two or three most highly selective predicates for estimation purposes. In choosing to use the two most highly selective predicates over two random query predicates, we ensure that the estimated query result space is reduced as much as possible. Calculating the individual query predicate selectivities to determine the two most highly selective predicates is very efficient when the simple histogram techniques described in Section 11.5.1 are used.

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

11.5.3 Heuristic Cutoff Method

Periscope's query optimizer has two different methods for determining the most efficient query plan, a heuristic cutoff method and a cost formula method. Both utilize the basic and complex histogram techniques described for determining segment and result selectivities. The heuristic cutoff method, which is simpler and less detailed, is the default technique in our current implementation, although the cost formula technique can be turned on with a simple flag option.

The heuristic cutoff process arose from our early experimentation with the four query evaluation methods, in which we found that each query plan performs differently for different queries depending on the selectivities of the query predicates and the cardinalities of the results. Further analysis of these performance trends allowed us to form some guidelines for choosing one query plan over the other based on the selectivity values. These guidelines then were translated into an algorithm that uses heuristic cutoff values to determine the most efficient plan. The cutoff values vary from system to system and should be established on each system through initial testing; they may also be tuned during further experimentation stages. Figure 11.8 contains the pseudocode for the heuristic cutoff algorithm.

The actual query optimization process using this method happens as follows. The basic histogram is used to determine the segment selectivities of all the nongap predicates in the query and the complex histogram is used to calculate the result protein selectivity. These results are input into the heuristic cutoff algorithm, which is then evaluated, and the optimal plan is returned. The system then uses this method to evaluate the query.

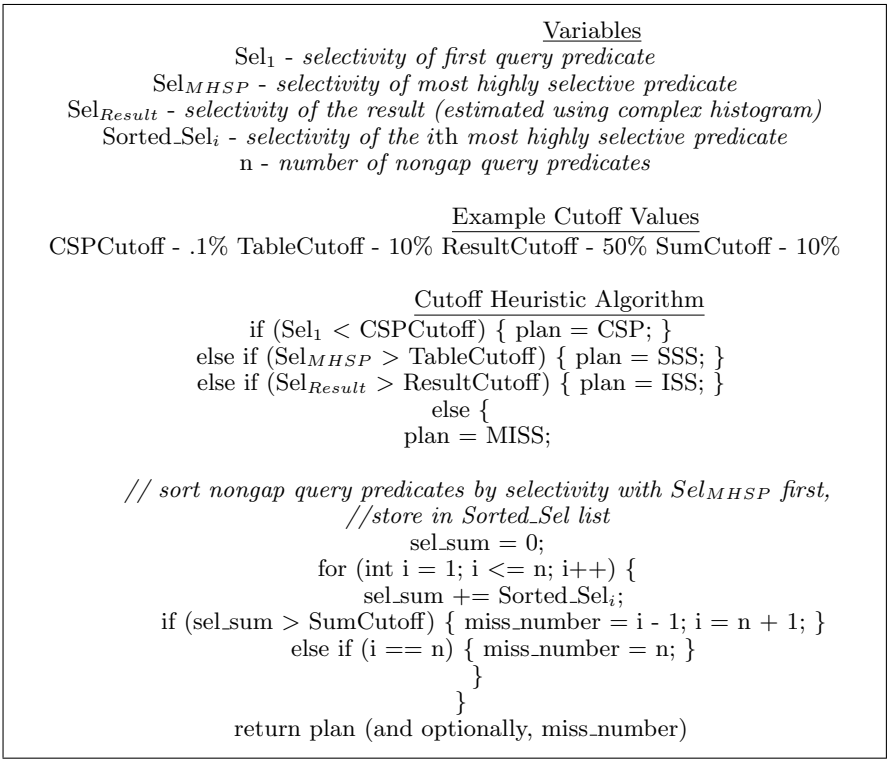

Fig. 11.8. Heuristic cutoff algorithm.

11.5.4 Cost Formula Method

The second method Periscope's query optimizer has for determining the most efficient query plan is the cost formula method. In this method, cost formulas are used 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 operations including index scans, table retrievals, and finite state machine matchings. We developed cost models, which are along the lines of the cost models in [356], for each of these basic operations. These models are then incorporated into the individual cost models for the various algorithms. The basic and complex histograms described in sections 11.5.1 and 11.5.2 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 sizes, maximum index fanout, and weighted I/O and CPU costs, are used. The following section describes the cost formulas for the basic operations that are

used by the query evaluation algorithms, and section 11.5.4 details how these cost formulas are incorporated to estimate the overall cost of each algorithm.

Cost functions for basic operations. A description of the variables and constants that are used in the cost functions is given in Table 11.5.

Table 11.5. Description of the cost function variables and constants.

Variables	Description
$\|\|S\|\|$	Total number of segments
$\|\|P\|\|$	Total number of proteins
$\|S\|$	Number of segment table pages
$\|P\|$	Number of protein table pages
$\|IS\|$	Number of segment index pages
$\|IP\|$	Number of protein index pages
$\|Q\ Pages\|$	Total number of pages in all the temporary files for a given query
$Sel(P_i)$	Selectivity of the ith query predicate
$Sel(Q)$	Result selectivity of the query
N	Total number of temporary files for a given query (MISS number)
$WCPU$	Weighted time for a CPU operation
WIO	Weighted time for an I/O operation
$MaxProtFanout$	Maximum fanout of the protein id index
$MaxSegFanout$	Maximum fanout of the segment <type, len> index
$AvgLen$	Average length of a protein in amino acids
$SegSize$	Size of a segment tuple in the DBMS
$PageSize$	Size of a page in the DBMS
$MaxNumberStates$	Number of possible states the FSM can be in at one time

One of the basic operations of the query evaluation methods is the complex scan of a protein. The time required for this operation depends on the average length of the proteins, the maximum number of states the FSM can be in at a given time, and the result selectivity of the query. This cost, $Cost_{CS}$, is estimated as

$$Cost_{CS} = WCPU \times AvgLen \times MaxNumberStates \times Sel(Q)$$

A basic operation used by the SSS plan is the scan of the segment table to find matching segments. This includes both the scanning time as well as the time to compare the segments to the given query predicate. The function $Cost_{SegScan}$ calculates the time required for this basic operation. It assumes that each comparison can be done in one CPU cycle. The cost of this operation is

$$Cost_{SegScan} = WIO \times |S| + WCPU \times ||S||$$

Another basic operation is the probing of the segment index to find matching segments; this operation also includes the retrieval of these segments. $Cost_{SegProbe}$ calculates the time required to probe the segment index and $Cost_{SegRet}$ calculates the time required to retrieve the matching segments. These functions require as input the selectivity of the given query predicate, $Sel(P_i)$. For $Cost_{SegProbe}$, the I/Os required include retrieving a fraction of the segment index pages as determined by the selectivity of the given query predicate. The CPU time required is the time to traverse to the leaf pages of the segment index to find the matching segments. Our calculation assumes that the height of the index is two. For $Cost_{SegRet}$, the I/O's required are to retrieve the data pages containing the matching segments. In the worst case, this is the maximum number of segment data pages; otherwise, it is determined by the number of matching segments, denoted by $Sel(P_i) \times ||S||$. The costs for the segment probing and retrieving operation are

$$\begin{aligned} Cost_{SegProbe} =& (WIO \times |IS| \times Sel(P_i)) + (WCPU \times 2 \times Sel(P_i) \times \\ & ||S|| \times log(MaxSegFanout)) \\ Cost_{SegRet} =& WIO \times min(|S|, Sel(P_i) \times ||S||) \end{aligned}$$

A related basic operation is the probing of the protein id index to find the matching protein ids; this operation also includes the retrieval of these proteins. The functions $Cost_{ProtProbe}$ and $Cost_{ProtRet}$ perform almost the same calculations as their corresponding segment functions described in the preceding paragraph. One difference occurs in the I/O portion of the $Cost_{ProtProbe}$ function, which calculates the I/Os required to retrieve the appropriate protein index pages. In the example segment case, the matching segments are found in one portion of the index, not randomly scattered throughout. The protein ids to be retrieved using the protein id index, however, although they are sorted, will not occur in one specific portion of the index but will instead be scattered throughout. Therefore, if the number of protein ids to be retrieved is greater than the number of protein id index pages, it is possible that all the index pages may need to be retrieved. This is accounted for in the $Cost_{ProtProbe}$ equation. The costs for the protein id probing and retrieving operations are

$$\begin{aligned} Cost_{ProtProbe} =& (WIO \times min(|IP|, Sel(P_i) \times ||S||)) + \\ & (WCPU \times 2 \times Sel(P_i) \times ||S|| \times log(MaxProtFanout)) \\ CostProtRet =& WIO \times min(|P|, Sel(P_i) \times ||S||) \end{aligned}$$

Another basic operation that is used by the MISS plans is the writing of temporary files. For each of the query predicates used in the MISS methods, the resulting matching segments must be sorted and written to temporary files. The function $Cost_{Write}$ calculates the time required to write one of these temporary files. (The time required for the sorting will be described later.) $Cost_{W}rite$ is based on the number of segments to be written, which also

depends on the selectivity of the given query predicate, $Sel(P_i)$. The number of segments to be written is multiplied by the storage size of a segment in the database storage manager and is divided by the size of a database page to determine the number of pages necessary for the temporary file. The cost for writing a temporary file is

$$Cost_{Write} = WIO \times Sel(P_i) \times ||S|| \times SegSize/PageSize$$

The merge of these temporary files is another basic operation that is used only by the MISS plans. The function $Cost_{Merge}$ calculates the I/O and CPU time necessary to merge the temporary files by protein id and start position. It requires as input the number of temporary files (or the MISS number), the total number of pages in these temporary files, and the selectivities of each of the corresponding query predicates. The only I/O cost incurred is the time to read each page in each temporary file, denoted as $|Q\ Pages|$. The CPU time is measured here for the worst case. In the average case each of the segments in each temporary file will have to be compared to each of the other segments in the other temporary files. Therefore, the CPU time can be found by computing the product of the number of segments in each temporary file. The number of segments in the ith temporary file is $Sel(P_i) \times ||S||$. The overall cost of the file merge, then, is

$$Cost_{Merge} = WIO \times |QPages| + WCPU \times \prod_{i=1}^{n} (Sel(P_i) \times ||S||)$$

The final basic operation involves sorting, either on the protein ids or on the segment type and lengths. Both sorts are implemented as quick sorts, which in general have an execution time of *O(m log m)*. In our cost formula m represents the number of segments being sorted, which depends on the selectivity of the query predicate used. The number of segments to be sorted is denoted by $Sel(P_i) \times ||S||$. The function $Cost_{Sort}$ calculates the time required to sort a set of segments; its equation is

$$Cost_{Sort} = WCPU \times Sel(P_i) \times ||S|| \times log(Sel(P_i) \times ||S||)$$

Cost functions for query plans. Now that we have examined the costs for each of the basic functionalities of the four query evaluation methods, we will look at how these basic operations are put together to formulate the overall query plan cost functions. The following cost formulas rely on the knowledge of the segment and result selectivities that are estimated using the two histograms described in sections 11.5.1 and 11.5.2. While they are not shown as inputs to the cost formulas that follow, it is assumed that they are available for use in the cost estimation. The cost for the CSP plan, $Cost_{CSP}$, is the easiest to calculate as it involves simply scanning the protein table and performing a complex scan for each protein:

$$Cost_{CSP} = WIO \times |P| \times ||P|| \times Cost_{CS}$$

The SSS method involves scanning the segment table to retrieve segments that match the most highly selective query predicate. These segments are then sorted by protein id and participate in an index probe of the protein id index to retrieve the actual proteins. A complex scan of the proteins may then be performed based on the number of predicates in the query. The cost formula for the SSS method, $Cost_{SSS}$, follows and assumes that the value for $Sel(P_i)$ that is used in the various basic operation formulas is the selectivity of the most highly selective predicate:

$$Cost_{SSS} = \begin{cases} Cost_{SegScan} + Cost_{Sort} + \\ Cost_{ProtProbe} + Cost_{ProtRet} + \\ (Sel(P_i) \times ||S|| \times Cost_{CS}) & \text{if number of query predicates} > 1 \\ \\ Cost_{SegScan} + Cost_{Sort} + \\ Cost_{ProtProbe} + Cost_{ProtRet} & \text{if number of query predicates} == 1 \end{cases}$$

The ISS method involves probing the segment index to retrieve segments that match the most highly selective query predicate. These segments are also sorted by protein id and participate in an index probe of the protein id index to retrieve the actual proteins. A complex scan of the proteins may then be performed based on the number of predicates in the query. The cost formula for the ISS method, $Cost_{ISS}$, follows and assumes that the value for $Sel(P_i)$ that is used in the various basic operation formulas is the selectivity of the most highly selective predicate:

$$Cost_{ISS} = \begin{cases} Cost_{SegProbe} + Cost_{SegRet} + \\ Cost_{Sort} + Cost_{ProtProbe} + \\ Cost_{ProtRet} + (Sel(P_i) \times \\ ||S|| \times Cost_{CS}) & \text{if number of query predicates} > 1 \\ \\ Cost_{SegProbe} + Cost_{SegRet} + \\ Cost_{Sort} + Cost_{ProtProbe} + \\ Cost_{ProtRet} & \text{if number of query predicates} == 1 \end{cases}$$

The MISS method is more complicated because multiple probes of the segment index are performed; their results are written to temporary files and then merged before participating in the protein index probe. A complex scan of the proteins may then be performed if the MISS number, n, is less than the total number of predicates in the query. When calculating the cost of the MISS plan, $Cost_{MISS}$, care must be taken in specifying the selectivities that are used by the various basic operations. Both the formulas $Cost_{ProtProbe}$ and $Cost_{ProtRet}$ described in section 11.5.4 use a selectivity value, $Sel(P_i)$,

to determine how many protein ids will need to be probed for and retrieved. In the SSS and ISS cost formulas, this selectivity value is simply the selectivity of the most highly selective predicate. In the MISS cost formula, however, this selectivity value represents how many protein ids are estimated to be returned from the merge of the n query predicates. $Cost_{MISS}$ assumes the worst when calculating this selectivity value by using the sum of the n segment selectivities. This is essentially saying that the number of protein ids that will be returned from the merge is the same as the total number of segments returned from the n segment index probes. Although this is an upper bound on the actual number of protein ids, it is still accurate enough to give a reasonable estimation. Therefore, it is assumed that the $Cost_{ProtProbe}$ and $Cost_{ProtRet}$ formulas will use the sum of the n query predicate selectivities as the required selectivity value. This same value also is used to determine the number of complex scans that may have to be performed at the conclusion of the MISS plan; this is shown in the $Cost_{MISS}$ formula that follows. The $Cost_{MISS}$ formula also assumes that the n predicates used in the n segment index probes, writes, and sorts are the n-most highly selective query predicates. The cost to evaluate a query using the MISS plan with a MISS number of n is

$$Cost_{MISS} = \begin{cases} \sum_{i=1}^{n}(Cost_{SegProbe} + Cost_{SegRet} + \\ Cost_{Sort} + Cost_{Write}) + \\ Cost_{Merge} + Cost_{ProtProbe} + \\ Cost_{ProtRet} + (Cost_{CS} \times \\ \sum_{i=1}^{n}(Sel(P_i) \times ||S||)) & \text{if number of query predicates} > \text{N} \\ \\ Cost_{SegProbe} + Cost_{SegRet} + \\ Cost_{Sort} + Cost_{Write}) + \\ Cost_{Merge} + Cost_{ProtProbe} + \\ Cost_{ProtRet} & \text{if number of query predicates} == \text{N} \end{cases}$$

The actual query optimization process for the cost formula method happens as follows. First the simple histogram is used to determine the segment selectivities of all the nongap 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 MISS(n) plan. Finally, the plan with the lowest cost formula is returned as the optimal plan and the system uses this method to evaluate the query.

11.6 Experimental Evaluation and Application of Periscope/PS^2

In this section we first present an experimental performance evaluation of the tool Periscope/PS^2, which implements the query language and query-processing techniques described in the previous sections. Then we describe an actual case study demonstrating the use of this tool in practice.

11.6.1 Experimental Evaluation

In this section, we compare the algorithms presented in section 11.4. The goal of this section is to present a few key results for comparing the performance of the individual algorithms; more extensive performance comparison results can be found in [164]. For all the experiments presented in this section, the Periscope heuristic optimizer picks the cheapest plan.

Setup. We implemented our query evaluation techniques in Periscope, which is built on top of the SHORE storage manager from the University of Wisconsin [65]. 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. For these 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 an 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 graphs.

In the following experimental sections, the abbreviations CSP, SSS, ISS, and MISS are all implicitly understood to be implementations of the four algorithms presented in section 11.4. When appropriate, the MISS plan will be shown for all possible numbers of query predicates from two to the total number of nongap predicates in the query, denoted by MISS(n), and the number of predicates used in the individual MISS plans will be referred to as the MISS number.

Dataset. To produce a dataset for our experiments, we first downloaded the entire PIR-International Protein Sequence Database. This database is a comprehensive, nonredundant protein database in the public domain and is extensively cross-referenced [434]. Since the PIR dataset contains only primary protein structures, we then used the Predator tool [134] to obtain predicted secondary structures. The final dataset 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.

Performance comparison. In this experiment we use a complex 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 nongap 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 nongap predicates. Figure 11.9 shows the results of this experiment in which the position of the large query predicate varies from last in the query to first.

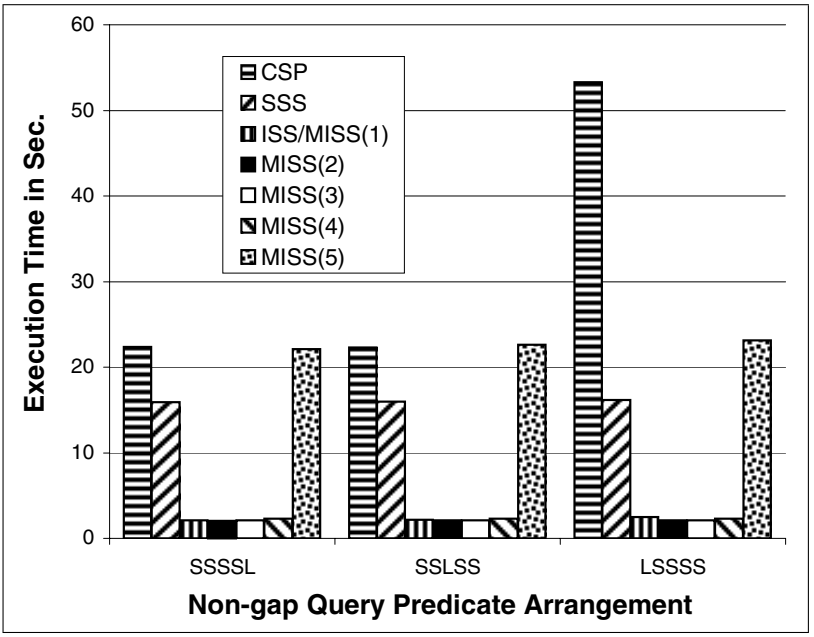

Fig. 11.9. Nine-predicate queries, S seg. sel. = ∼.03%, L seg. sel. = 7%, fixed result sel. < .1%.

The results show that the CSP method is the only method whose execution time varies widely depending on the position of the large predicate, which indicates 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 check more states. 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 query 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 are all very selective. In MISS(5), however, the index is also scanned for the large (less selective) predicate. This adds considerable length to the execution time (recall that the MISS algorithm picks predicates based on their selectivities, not their physical order in the query).

The most efficient MISS number, in general, depends 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 an FSM scan of each protein is also 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 time to probe the id index for the eliminated proteins, the time to retrieve them, and the time to 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 11.9 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 to query characteristics can significantly improve query response times. In addition, this experiment demonstrates that it is possible to evaluate these complex queries even on the large PIR dataset in few seconds, which allows the use of this tool in an interactive querying mode.

11.6.2 Example of Application of Periscope/PS^2

In this section, we outline an actual application of the Periscope/PS^2 tool. For this study, the PIR dataset described in section 11.6.1 was used.

Background. The common bioinformatic tasks for life scientists are to (1) infer functional similarity, (2) infer structural similarity, (3) identify domains

or compact structural regions, and (4) estimate evolutionary relationships between the target under investigation and the database of characterized proteins [336]. A variety of computational tools are used for these activities; however, the majority relate to finding other database entries that share the primary sequence (i.e., the sequence of amino acids that make up the protein) similarity and assembling a multiple sequence alignment. From the alignment, complex judgments are made about the degree of "relatedness" and the functional implications of the array of potentially similar proteins [33]. If the target protein is not found to be similar in sequence to any other proteins with an experimentally determined atomic structure, the entire collection of aligned protein sequences are processed by secondary structure prediction algorithms and any common patterns noted. In some cases, the predicted secondary structure may lead to adjustments in the primary sequence alignment. This combined analysis often provides the basis for experimental decisions, for example, which example of an apparently conserved protein to clone and express, or to identify conserved regions that may bear directly on function and as such develop a "hit list" of potential point mutants. As currently practiced by working life scientists, bioinformatics analysis of proteins is not algorithmic but a complex heuristic process informed not only by the results of various database searches and computational analyses, but significantly by the individual scientist's experience and expert knowledge of the biology related to the query protein. That is, the analysis actually generates hypotheses: x is related to y, a and b share the same modular arrangement of domains, m and n are both DNA binding proteins. These hypotheses nucleate experimental work on the biological system in question.

In this context, Periscope has a variety of clear applications for life sciences researchers. A Periscope/PS^2 search string encodes local protein topology; as a consequence, the user is actually performing a direct arbitrary topological search against a database of protein topologies. To our knowledge, this search domain is unique to Periscope. Secondary structural pattern searches are of particular use for investigators studying protein structure and function. A distantly related protein or remote homolog (i.e., same superfamily but a different family; see [340] for definitions) will have low sequence similarity and perhaps escape the statistical significance threshold of the commonly used BLAST search heuristic [13]. Combining a Periscope/PS^2 search with a sequence-based search will augment the ability to select potential true homologs and perhaps analogs from unrelated proteins.

Actual example demonstrating the use of Periscope/PS^2. A standard BLASTP search performed on an experimentally uncharacterized protein, "conserved hypothetical protein from Streptococcus pyogenes" (GI 19745566), returns 14 hits. Buried in the search results in the low-confidence area, commonly referred to as the twilight zone [337], is the C3 exoenzyme of *C. botulinum* (E = 0.025), a bacterial exotoxin of considerable interest. Is our query protein truly related to the C3 exoenzymes, or is it a member of

the larger superfamily of ADP-ribosylating enzymes? The sequence BLASTP query results are suggestive. The secondary structure of GI 19745566 was predicted using SAM-T99sec [213]. Based on the secondary structure prediction, several Periscope search strings were developed and used to query the combined nonredundant database described in section 11.6.1.

Table 11.6 shows the final query and a summary of each matching result. Among the Periscope results is the *C. botulinum* C3 exoenzyme (PDB code 1GZE). The uncharacterized query protein's predicted secondary structure closely matches the experimentally determined secondary structure of the C3 exoenzyme. Furthermore both proteins are of similar overall length, and the matching secondary structure spans 56% of the experimentally determined structure. Based on the combined results of low but detectable sequence similarity and common local topologies, we can hypothesize that the uncharacterized *S. pyogenes* open reading frame is a member of the ADP-ribosylating toxin family.

In summary, Periscope/PS^2 allows direct arbitrary topological searches through a simple declarative query language. This tool can (1) enhance the detection of remote protein homology, (2) provide topological information for the classification of a protein as a remote homolog vs. analog of the query, and (3) provide a unique tool for exploring loop insertions and deletions in known protein families.

11.7 Conclusions and Future Work

Knowing 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 chapter addresses the problem of efficient and declarative querying of the secondary structure of protein datasets.

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 among the various algorithms, we have also developed novel histogram techniques to determine segment and result selectivities. We have implemented and evaluated the proposed techniques in a native DBMS we have developed called Periscope. As the experimental results show, the system we have developed can query large protein databases efficiently, allowing scientists to interactively pose queries even on large datasets.

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 so that the metric used for ranking the answers can be easily customized by the user, as the model for ranking proteins is usually not fixed but instead varies among scientists and may also change frequently during the course of an experiment. The ranking metric may take into account

Table 11.6. Results for the query <h 4 6> <? 6 12> <h 6 10> <? 17 30> <h 10 15> <l 4 20> <h 7 20> <l 8 20> <h 10 20> <l 3 6> <e 4 8>.

Accession number	Description	Match
AD1845	5-methyltetrahydrofolate-homocysteine S-methyltransferase [imported] - *Nostoc* sp. (strain PCC 7120)	352 to 464
E86671	Lysine-tRNA ligase (EC 6.1.1.6) [imported] - *Lactococcus lactis subsp. lactis* (strain IL1403)	390 to 488
A38912	NAD-asparagine ADP-ribosyltransferase (EC 2.4.2.-) C3 precursor - *Clostridium botulinum* phage (strain CST)	53 to 171
E70838	Hypothetical protein Rv0200 - *Mycobacterium tuberculosis* (strain H37RV)	117 to 224
D90551	Lipoprotein [imported] - *Mycoplasma pulmonis* (strain UAB CTIP)	123 to 240
C89045	Protein B0238.6 [imported] - *Caenorhabditis elegans*	32 to 154
C82251	GGDEF family protein VC1029 [imported] - *Vibrio cholerae* (strain N16961 serogroup O1)	107 to 228
D69105	Coenzyme PQQ synthesis protein - *Methanobacterium thermoautotrophicum* (strain delta H)	139 to 252
T41643	Probable involvement in cytoskeletal organization - fission yeast (*Schizosaccharomyces pombe*)	1253 to 1361
1GZE	Chain A, structure of the *Clostridium botulinum* C3 exoenzyme (L177c mutant)	13 to 132
747707	Exoenzyme C3 [*Clostridium botulinum* D phage]	20 to 138
P15879	ARC3_CBDP mono-ADP-ribosyltransferase C3 precursor (exoenzyme C3)	53 to 171

additional information that is present in the protein, such as the positional probability of 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 [117, 118, 295].

Search engines for querying biological datasets often employ a query-by-example interface. In BLAST, one of the most popular search 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 the sequence. 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 chapter focuses on 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 among 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 datasets in a variety of different ways. For example, a scientist may first query on the primary structure of a protein and then on the secondary structure or vice versa. Ideally a *combination* of both primary sequence and secondary structure searches will lead to more accurate protein function discovery [307]. This chapter addresses only the issue of efficient query-processing techniques for secondary structure. Hence the tool we have built would be an addition to the suite of biological querying tools that exist today. Developing techniques for integrated and declarative querying on all protein structures is an interesting database problem, and it is part of the long-term goal of the Periscope project.

Acknowledgments

This research has been supported in part by donations from IBM and Microsoft.