PiQA: An Algebra for Querying Protein Data Sets

Sandeep Tata and Jignesh M. Patel
University of Michigan
{tatas, jignesh}@eecs.umich.edu

Abstract

Life science researchers frequently need to query large protein data sets in a variety of different ways. Protein data sets have a rich structure that includes its primary structure, which is described as a sequence of amino acids, and its secondary structure, which is described as a sequence of folding patterns of the protein. Both these structures are important as the amino acid sequence is often used to find homologous proteins, and the secondary structure can produce important hints about the functionality of proteins. While there are tools for querying each of these structures independently, there are no tools for declarative querying on both these structures. Even the tools that allow querying on either one of these structures are not based on any formal algebra, and as a result require complex rewriting of the tools programming logic when the "query evaluation plan" changes. This paper introduces PiQA, a Protein Query Algebra, which provides a rich set of algebraic operations on both the primary and secondary structure of proteins. Using PiQA one can pose several interesting complex queries involving both the primary and the secondary structure of proteins. In addition, simple existing tools that query only on the primary structure, such as BLAST, can also be expressed in this algebra. PiQA is an important first step in developing an algebra that can form the basis of a declarative querying language for querying protein data sets.

1. Introduction

Recent years have seen an enormous explosion in the sizes and uses of biological data. Several nucleotide and protein sequence data sets are growing at an exponential rate, doubling roughly every 16 months [1]. In addition, the nature of the searches against these databases is also changing, and scientists today would like to ask more complex queries against these data sets. Database management tools have an important role to play in querying such biological datasets [9, 15]. This paper focuses on one such aspect, namely the querying of protein data sets based on different structural attributes that describe each protein.

Proteins have the following four levels of structural organizations: primary, secondary, tertiary and quaternary structures. In this paper, we focus on querying the primary and secondary structures. The primary structure is simply a linear sequence of amino acids residues that forms the protein. The secondary structure describes how the linear sequence of amino acids residues orients itself, or folds, in three-dimensional space. There are three basic types of folds: alpha-helices, beta-pleated sheets, and turns or loops. Knowledge of a protein's secondary structure has been shown to provide important insights into its evolutionary relationships, and hence its function.

Typically, biologists are interested in finding similarities between a sequenced protein and others in the database so that they can understand the function of the sequenced protein. For instance, given a protein, they may want to determine if similar proteins exist in other species, and may also want to determine the function of the protein. Or they might be interested in knowing if there are other proteins that have a different primary structure, but have a similar secondary structure. The secondary structure of the protein is crucial to understanding the function that the protein performs [3, 16, 21], and hence it is important to be able to understand it in relation to the primary structure.

1.1. The Problem

Today when scientists investigate a protein, they usually search databases of known proteins based on the primary sequence. The search is typically carried out using tools such as BLAST [4, 5], Such search tools essentially finds homologous matches. These search tools return approximate answers, and often a scientist may have to post-process these results, or run the search iteratively (as in PSI-BLAST). In addition, the scientist may query multiple data sets producing a large number of approximate matches that may feed into the next stage of their analysis. With protein queries, in many cases the next step after matching on the primary sequences may be to examine the protein of interest with the secondary structures of other known proteins in the database. The matching on the secondary structure is important as the functionality of proteins is strongly influenced by its actual folding pattern, and even proteins that are not close homologs may exhibit similar behavior if their folding patterns are similar.

As an example, a biologist might have just sequenced the hemoglobin protein in monkeys and may be interested

in hemoglobin and other proteins in other species that are "similar" to this protein. Such comparisons are also useful in tracking evolutionary changes in the structure of the protein [23]. In certain other instances, when a biologist is trying to find a protein that matches a certain structural fingerprint – i.e. a certain spatial arrangement, they might have a secondary structure in mind and want to find proteins in the database that have a similar structure.

In many cases, these steps of querying on the primary and secondary structures may be repeated many times, and for many different databases. Often the iteration between these steps is driven by a manually coded program, which may need to be modified every time the underlying *query* changes. In addition, this entire process may needs to be carried out for each distinct experiment that is undertaken in a lab. A declarative query tool that permits querying on both the primary and secondary structures can not only reduce the time spent in posing such queries, but can also allow the biologist to pose more complex queries than are currently used today.

As an example, using currently existing tools one cannot express the following query in a straightforward way: *Match the given primary sequence of length 120, but ignore mismatches in the segment from positions 44 to 78 if it is on a loop in the secondary structure.*

In the next several sections we shall describe an algebra that can be used to query protein data sets based on both the primary and secondary structures. The algebra supports approximate matching, and also includes operators that allow extensions of two or more approximate matches to calculate a "longer" match. We believe that the algebra is expressive enough to express a large class of interesting queries on both the primary and secondary structures of proteins.

The motivations for developing such algebra are fairly obvious to a database audience. The algebra is a first step in providing a declarative query language-based interface to the user, rather than the more cumbersome procedural paradigm that is currently being used for queries across both primary and secondary structures. In addition, the algebra can also be exploited by a query optimizer to produce efficient query plans.

The key contribution of this paper is the presentation of PiQA – a Protein Query Algebra that enables us to express queries on both the primary and secondary structures of proteins. To the best of our knowledge PiQA is the first algebra that allows querying on both these structures. Using PiQA we also show how existing queries on only the primary structure can be expressed in this algebra. In addition, we also illustrate the use of the algebra in query optimization.

In Section 2, we describe related tools and techniques for querying protein datasets. Section 3 describes the types and operators in PiQA. Section 4 presents several sample queries to demonstrate the expressive power of the algebra. Section 5 uses an example to illustrate the use of the algebra for query optimization. Section 6 concludes the paper and points to directions for future work.

2. Related Work

Surprisingly, there is little previous work on developing an algebraic framework for querying biological data sets. Recently, Hammer and Schneider [13] proposed a long-term approach towards developing an algebra that abstracts several biological processes. Seshadri et al. [19, 20] describe techniques for querying sequence databases. However, these techniques primarily focus on aggregate-based analysis of sequences, and are not directly applicable for querying biological sequences, which often require pattern matching and approximate matching operators. There has been a lot of work in string matching, including proposals for a declarative language based on alignment calculus for strings [11]. However, these techniques can only be applied to primary sequence matching without approximations.

The algebraic constructs that we develop in this paper employ many of the constructs that have been developed for nested relational algebras [2, 14, 18, 22]. However, we have been able to express the queries that we target using only a limited form of nesting, namely PNF relations [18], with only one level of nesting. Consequently, the optimizations too are simpler than those developed for more general forms of nesting [8, 10, 17].

2.1. Primary Structure-based Searching

A number of tools have been developed for searching on nucleotide sequences and primary protein sequences. The most frequently used tool in this category is the BLAST [4, 5] family of search programs. BLAST works in three steps: in the first step it finds all K-mers (strings from the alphabet of length K) that score above a certain threshold with some part of the query string. In the next step, it searches the database to find hits. In the final step, BLAST extends the hits according to certain heuristics and returns a list of *high-scoring segment pairs.* This score is a measure of similarity.

2.2. Secondary Structure-based Searching

Searching based on the secondary structure of proteins has recently been examined by Hammel and Patel [12]. The authors define an intuitive query language that can be used to express queries on secondary structure and also

developed techniques for evaluating and optimizing these queries.

3. The PiQA Algebra

The algebra that we describe is a multi-sorted algebra. The operators can be composed to specify complex queries involving both the primary structure and the secondary structure. We have formulated the algebra as an *extension* to relational algebra so that we still have the advantage of modeling data as relations. More precisely, the relations in our model are in the Partitioned Normal Form (PNF) [18]. (PNF relations restrict the class of general nested relations to guarantee the desirable property that a nest operation is the inverse of an unnest operation.)

We shall first describe all the types in the algebra, and then describe each of the operators, the types of their operands, and the type of the result. In the interest of space, we do not describe the basic relational algebraic constructs [7], and extensions of these constructs to accommodate PNF relations [18].

3.1. Types

The basic types in the algebra are:

- Basic Scalar Types – Integers, Characters etc.
- Matches
- Sequences
- Tuples
- Relations (sets)

3.2 Description of the Types

3.2.1. Matches. A *Match* is used to correlate a query pattern to a sequence. It is a 2-tuple where the first component is an identifier. The identifier is a unique key that identifies a particular sequence. We could have used a Skolem function instead, but for simplicity we use an identifier. The second component is a sequence of 3-tuples. Each 3-tuple is a *match element*. Match elements are described by three attributes: *position* – which refers to the position of the match element in the string, *length* – which refers to the length of that match element, and a *score* – which refers to the score of that match element.

For instance, **(CG2B, ((22, 7, 6), (44, 12, 9)))** is a match which could have been the result of some operation, and it means that the sequence referred to by CG2B matched at position 22 and at position 44 with lengths of the matches being 7 and 12, and the scores being 6 and 9 respectively.

For ease of presentation, in some of the examples below, we represent the matches in an alternative form. In this alternative representation, the match is represented as a 4-tuple where the first component is an identifier. The remaining components of a match are sequences. The second component is a sequence of integers which refer to positions in a string, the third component is a sequence of integers which refer to the lengths of each of the matches whose positions are referred to by the previous sequence, and the fourth sequence in a match comprises the integers that represents the scores. In this alternative representation, the previous example would be expressed as: **(CG2B, (22, 44), (7, 12), (6, 9)).**

Several operators that we describe operate on sets of matches. We can view a set of matches as a nested relation with the first identifying component of the match serving as a key which functionally determines the other attributes in the relation. With this interpretation, we observe that these sets are in Partition Normal Form [18].

For ease of presentation, in some of the examples in this paper, we represent the third and fourth components of the match in the alternative representation as a function from natural numbers (sequence positions) to natural numbers (the match positions, lengths or the scores).

In addition in this paper, we do not consider null matches.

3.2.2. Regular Expressions and Matches. A *regular expression* is used to represent a match criterion. As in [12], a regular expression is expressed as a sequence of *segment predicates*, each of which must be matched to satisfy the entire expression. Each segment predicate is described by the type and the length of the segment. The type of the segment is drawn from the alphabet of the underlying sequence, and depends on the sequence being queried. For the protein secondary structures, the allowed segment types are h, e, and l, for the alpha-helices, beta-sheets, and loops, respectively. In addition we also add a fourth option, '?', which stands for a gap segment and allows scientists to represent regions of unimportance in a query. The length of the segment is specified using an upper bound and a lower bound, each of which could be 0. In addition the upper bound could be specified as ∞. Segment predicates over other structures are similar, except that the type used is set to the symbols in the underlying alphabet, with the addition of the '?' symbol.

Formally, a regular expression is defined using the rules shown in Figure 1.

As an example, consider the following expression on a protein secondary structure: *<e 3 5><? 0 ∞><l 7 7>*. This regular expression matches all proteins that contain a beta-sheet of length 3 to 5 followed at some point by a loop of length 7.

RegExp → {Segments}
Segments → Segment*
Segment → <type lb ub>
type → e | h | l | ? *(for protein secondary structures)*
type → A | R | N | D | …| ? *(for protein primary structures)*
type → A | G | C | T | ? *(for nucleotide sequences)*
lb → any integer ≥ 0
ub → any integer ≥ 0 | ∞
Segment Constraint: lb ≤ ub

Figure 1: Regular Expression Syntax

3.2.3. Sets and Sequences. *Sets* and *Sequences* are well known types. Sequences have the standard position (or index) operator which allows access to an arbitrary element in the sequences. For example, the i^{th} position in a sequence S is simply accessed as S(i).

In this algebra, we only permit a sequence of the basic scalar types. That is, we may have a sequence of integers, characters, etc. But we do not have sequences on complex types such as relations. We do not define operations on sequences directly, but on the matches that have sequences as a part. And therefore, not having sequences of more complex types does not detract from the power of expressing queries that PiQA targets.

Since a string is merely a sequence of characters (over a relevant alphabet), we will use the terms string and character sequence interchangeably.

3.3. Operators

In this section, we describe the PiQA operators.

3.3.1. Match operator (*)

$*: Set<strings> \times (str \cup regexp) \rightarrow Set$

The match operator searches the set of strings (the left operand) to find substrings that approximately match the right operand, which could be a string or a regular expression specifying a set of strings. The result of this operation is a set of matches, each consisting of the identifier of the corresponding string, the match-positions and their lengths and scores. Symbolically, a match expression is of the form:

T * (str | regexp), where T is a set of strings.

A common use of the match operator is to search on the primary structures using a string *str*, or searching on the secondary structures using a regular expression *regexp*.

The match operator is defined under *some* matching criterion, for instance PAM-30, PAM-70 or a BLOSUM62 matrix can be used to determine a match score between two primary protein structures. One may also choose to use an exact matching criterion or some other measure of approximate matching – gapped, un-gapped, etc. for secondary structures (and even for primary structures). We will not deal in depth with specific matching criteria in this paper. Though certain kinds of optimization may be possible if we know the matching criterion and scoring function used, in the interest of generality, our formulation will not be tied to any choice of matching criteria, except when explicitly specified.

Example:

Consider a protein table, P, with the following attributes: *id* – a unique identifier, *p* – a string representing the protein primary structure, and *s* – a string representing the protein secondary structure. In this example, we shall use the short-hand **P**.p to denote the set of primary sequences, and **P**.s to denote the set of secondary sequences.

P =

id	p	s
1	"GQISDSIEEKRGFF"	"HLLLLLLLLLHEE"
2	"EEKKGFEEKRAVW"	"LLEEEEEHHHHHL"
3	"QDGGSEEKSTKEEK"	"HHHHLLLEEEELLL"

str = "EEK"
regexp = <l 3 5>

P.p * str = {(1, (8), (3), (3)), (2, (1, 7), (3, 3), (2, 1)), (3, (6, 12), (3, 3), (1, 0))}

P.s * regexp = {(1, (2, 3, 4, 5, 6, 7, 8), (5, 5, 5, 5, 5, 4, 3), (1, 1, 1, 1, 1, 0, 1)), (3, (5, 12), (3, 3), (2, 2))}

The scores assigned in the example above have been arbitrarily assigned. However, we can also choose to explicitly specify the scoring criteria. If we for instance wished to specify that the matches be scored using the BLOSUM62 matrix, we would say:

$$\mathbf{P}.p *_{BLOSUM62} str \quad (1)$$

This is a simple way of expressing the idea used in BLAST for similarity searching. The nature of the matching operation has an effect on the semantics of other operators that we will describe in subsequent sections.

3.3.2. Nest (υ) and Unnest (μ) Operators

$\upsilon: Set<matches> \rightarrow Set<matches>$
$\mu: Set<matches> \rightarrow Set<matches>$

Unnest (μ) is a simple operator that flattens out a relation holding matches.

For instance, "μ ({(1, (8), (3), (3)), (2, (1, 7), (3, 3), (2, 1)), (3, (6, 6), (3, 8), (1, 7))})", produces the relation:

id	position	length	score
1	8	3	3
2	1	3	2
2	7	3	1
3	6	3	1
3	6	8	7

The Nest operation (υ) is merely the reverse. It collapses the unnested set into a set of matches.

To demonstrate the use of the nest and unnest operators, consider the simplistic BLAST expression in Equation 1. We can restrict the matches to only contain match elements with a score greater than a certain threshold *T*, as:

$$\upsilon_{(pos,\ score,\ len)}\,(\sigma_{score>T}\,(\mu_{match\text{-}sequence}(\mathbf{P}.p *_{BLOSUM62}\ str)) \quad (2)$$

Since the set of matches can be viewed as a PNF relation, the nest and unnest operators are inverses of each other. Furthermore, this inverse relationship is preserved under each of the operations in the algebra. (In the interest of space, we omit the proof in this paper, which requires exhaustively examining the operators.)

3.3.3. Union Operator ($\bigcup$)

$$\bigcup : Set\langle matches\rangle \times Set\langle matches\rangle \rightarrow Set\langle matches\rangle$$

The union operator generates a set that consists of all the matches in the two sets it operates on. If match elements with a common protein exist in the two sets, then they are combined in a single match in the result.

Symbolically, the operation is represented as:

$$\mathbf{T} = \mathbf{R} \bigcup \mathbf{S}$$

where **R** and **S** are two sets of matches, and **T** is their union.

Example:

R = {(1, (3, 6, 9), (3, 3, 3), (2, 2, 2)), (2, (1, 4), (4, 4), (3, 4))}

S = {(2, (5), (4), (3)), (5, (1, 8), (5, 5), (4, 5))}

T = **R** ⋃ **S** = {(1, (3, 6, 9), (3, 3, 3), (2, 2, 2)), (2, (1, 4, 5), (4, 4, 4), (3, 4, 3)), (5, (1, 8), (5, 5), (4, 5))}

3.3.4. Intersection Operator ($\bigcap$)

$$\bigcap : Set\langle matches\rangle \times Set\langle matches\rangle \rightarrow Set\langle matches\rangle$$

The intersection of two sets of matches consists only of matches with proteins common to both sets. Within each match only match elements common to both sets are included.

Symbolically, the operation is represented as:

$$\mathbf{T} = \mathbf{R} \bigcap \mathbf{S}$$

where **R** and **S** are two sets of matches, and **T** is their exact intersection.

Example:

R = {(1, (3, 6, 9), (3, 3, 3), (2, 2, 2)), (2, (1, 4), (4, 4), (3, 4)), (3, (7, 13, 22), (7, 7, 7), (5, 6, 6))}

S = {(2, (1, 5), (4, 6), (3, 5)), (3, (13), (7), (6)), (5, (1, 8), (6, 6), (5, 6))}

T = **R** ⋂ **S** = {(2, (1), (4), (3)), (3, (13), (7), (6))}

We observe that:

$$T = \upsilon_{(pos,\ score,\ len)}(\mu_{match\text{-}sequence}(R) \bigcap \mu_{match\text{-}sequence}\,(S))$$

3.3.5. Difference Operator (−)

$$-: Set\langle matches\rangle \times Set\langle matches\rangle \rightarrow Set\langle matches\rangle$$

The difference of two sets of matches consists of matches that are present in the first set and not in the second set.

Symbolically, the operation is represented as:

$$\mathbf{T} = \mathbf{R} - \mathbf{S}$$

where **R** and **S** are two sets of matches, and **T** is their difference.

Example:

R = {(1, (3, 6, 9), (3, 3, 3), (2, 2, 2)), (2, (1, 4), (4, 4), (3, 4)), (3, (7, 13, 22), (7, 7, 7), (5, 6, 6))}

S = {(2, (1, 5), (4, 6), (3, 5)), (3, (13), (7), (6)), (5, (1, 8), (6, 6), (5, 6))}

T = **R** − **S** = {(1, (3, 6, 9), (3, 3, 3), (2, 2, 2)), (2, (4), (4), (4)), (3, (7, 22), (7, 7), (5, 6))}

The keen reader may notice that the set union, intersection, and difference operators defined in this paper are similar to the extended set operators defined in [18].

3.3.6. Match Extension Operators (|, ||)

$$|: Match \times Match \rightarrow Match$$
$$||: Set\langle Matches\rangle \times Set\langle Matches\rangle \rightarrow Set\langle Matches\rangle$$

The match extension operator | operates on two matches and returns a match that is the list of all matches that can be formed by concatenating a match from Match-1 with a match from Match-2. That is, the result of the

operator is the list of matches in its left operand that could be extended in length using the right operand. The new scores in the extended matches can be specified using a general function that takes as input the two matches.

Symbolically, let

$m_1 = (pid_1, (a_1, a_2, a_3, \ldots a_p), L_1, S_1)$ be a match, and

$m_2 = (pid_2, (b_1, b_2, b_3, \ldots b_q), L_2, S_2)$ be another match.

$m_1 \mid m_2$ is defined only when $pid_1 = pid_2$,

and is equal to $(pid_1, (c_1, c_2, c_3, \ldots, c_r), L_3, S_3)$,

where $c_i = a_j$, and for some k,

$a_j + L_1(j) = b_k$, and $L_3(i) = L_1(j) + L_2(k)$

Clearly, the operator is not commutative.

If **R** and **S** are sets of matches, then the match extension is specified as follows:

$\mathbf{T} = \mathbf{R} \parallel S = \{M \mid M = m_1 \mid m_2, m_1 \in R, m_2 \in S\}$

Example:

R = {(1, (1, 8, 22), (4, 10, 6), (4, 8, 5)), (2, (3, 7), (3, 4), (3, 3))}

S = {(1, (5, 15, 28), (10, 2, 7), (9, 2, 6)), (5, (1), (7), (5))

T = **R** || **S** = {(1, (1, 22), (14, 13), (13, 11))}, assuming that the final score is the sum of the two scores.

The general form of the match extension operator concatenates two matches that are at most at a distance *d* and recomputes the score of the new match. The match extension operator described above is obtained by putting k=0 in the generalized form. One can mathematically describe the operator as follows:

$m_1 = (pid_1, (a_1, a_2, a_3, \ldots a_p), L_1, S_1)$ be a match, and

$m_2 = (pid_2, (b_1, b_2, b_3, \ldots b_q), L_2, S_2)$ be another match.

$m_1 \mid_d m_2$ is defined only when $pid_1 = pid_2$,

and is equal to $(pid_1, (c_1, c_2, c_3, \ldots, c_r), L_3, S_3)$,

where $c_i = a_j$, and for some k,

$b_k - (a_j + L_1(j)) \le d$, and $L_3(i) = L_1(j) + L_2(k) + d$

If **R** and **S** are sets of matches, then the operation of match extension can be written as

$\mathbf{T} = \mathbf{R} \parallel_d S = \{M \mid M = m1 \mid_d m_2, m_1 \in R, m_2 \in S\}$

Example:

R = {(1, (1, 8, 22), (4, 10, 6), (4, 8, 5)), (2, (3, 7), (3, 4), (3, 3))}

S = {(1, (7, 15, 28), (10, 2, 7), (9, 2, 6)), (5, (1), (7), (5))}

T = **R** $\|_2$ **S** = {(1, (1, 22), (16, 13), (12, 11))}

3.3.7. Contains Operators (Φ, ϕ)

$\phi : Match \times Match \rightarrow Match$

$\Phi : Set<Matches> \times Set<Matches> \rightarrow Set<Matches>$

The contains operator ϕ is in a certain sense a generalization of the exact intersection operator, i.e., a weaker definition of intersection over a set of matches. It returns a match position in Match-1 (the first match input), if the corresponding string contains (is a superset of) the string that corresponds to a match position in Match-2.

Let

$m_1 = (pid_1, (a_1, a_2, a_3, \ldots a_p), L_1, S_1)$ be a match, and

$m_2 = (pid_2, (b_1, b_2, b_3, \ldots b_q), L_2, S_2)$ be another match.

$m_1 \phi m_2$ is defined only when $pid_1 = pid_2$

and is equal to $(pid_1, (c_1, c_2, c_3, \ldots, c_r), L_3, S_3)$,

where $c_i = a_j$,

and for some k, $a_j \le b_k$ and $a_j + L_1(j) \ge b_k + L_2(k)$

Symbolically, if **R** and **S** are sets of matches, then the contains operation on these sets can be expressed as:

$\mathbf{T} = \mathbf{R} \Phi \mathbf{S} = \{m \mid m_1 \in \mathbf{R}, m_2 \in \mathbf{S}, m = m_1 \phi m2\}$

Example:

R = {(1, (1, 8, 22), (4, 10, 6), (4, 8, 5)), (2, (3, 7), (3, 4), (3, 3))}

S = {(1, (5, 15, 28), (10, 2, 7), (9, 2, 6)), (5, (1), (7), (5))}

T = **R** Φ **S** = {(1, (8), (10), (8))}

3.3.8. Not-contains Operators (Ψ, ψ)

$\psi : Match \times Match \rightarrow Match$

$\Psi : Set<Matches> \times Set<Matches> \rightarrow Set<Matches>$

The not-contains operator ψ is in some sense a generalization of the difference operator. Over matches, the operation produces a match from its left operand if a match element from the right operand is not contained in the left operand.

Let

$m_1 = (pid_1, (a_1, a_2, a_3, \ldots a_p), L_1, S_1)$ be a match, and

$m_2 = (pid_2, (b_1, b_2, b_3, \ldots b_q), L_2, S_2)$ be another match.

$m_1 \psi m_2$ is defined only when $pid_1 = pid_2$

and is equal to $(pid_1, (c_1, c_2, c_3, \ldots, c_r), L_3, S_3)$,

where $c_i = a_j$,

and for no k, $a_j \le b_k$ and $a_j + L_1(j) \ge b_k + L_2(k)$

Symbolically, if **R** and **S** are sets of matches, then the result of the not-contains operation can be written as:

$\mathbf{T} = \mathbf{R} \Psi \mathbf{S} = \{m \mid m_1 \in \mathbf{R}, m_2 \in \mathbf{S}, m = m_1 \psi m2\}$

We note that $\mathbf{R} \Psi \mathbf{S} = \mathbf{R} - (\mathbf{R} \Phi \mathbf{S})$. A formal proof for this statement requires showing that $R \Phi S$ includes all the match elements in R that contain some S element, and the matches produced by this operation are precisely the ones that need to be excluded from R.

Example:

R = {(1, (1, 8, 22), (4, 10, 6), (4, 8, 5)), (2, (3, 7), (3, 4), (3, 3))}

S = {(1, (5, 15, 28), (10, 2, 7), (9, 2, 6)), (5, (1), (7), (5))}

$\mathbf{T} = \mathbf{R} \Psi \mathbf{S}$ = {(1, (1, 22), (4, 6), (4, 5)), (2, (3, 7), (3, 4), (3, 3))}

4. Examples of Queries in PiQA

In this section, we present examples of queries to demonstrate the expressive power of PiQA. These examples are expressed on a protein table **P**, which has the same structure as the example used in Section 3.3.1.

4.1. Sample Queries

1. Find all proteins that contain the primary structure sequence "QISDSIE" with the secondary structure of "DSI" being <H 3 3> or <L 3 3>.

(**P**.p * "QIS") || ((**P**.p * "DSI") Φ ((**P**.s * <H 3 3>) $\bigcup$ (**P**.s * <L 3 3>))) || (**P**.p * "E")

2. Find all proteins that contain the secondary structure <E 1 5><L 2 2><E 3 9> and a primary structure sequence "SSDGTQ" nowhere within it.

(**P**.s * <E 1 5> <L 2 2> <E 3 9>) Ψ (**P**.p * "SSDGTQ")

3. Find all proteins that contain the primary structure sequence "SPPNKD" with the condition that the secondary structure for "PP" is not <E 2 2>.

(**P**.p * "S") || ((**P**.p * "PP") – (**P**.s * <E 2 2>)) || (**P**.p * "NKD")

4. Find all proteins that have the secondary structure <H 3 6> or <E 4 5> and the primary structure "NKN" contained in it.

((**P**.s * <H 3 6>) $\bigcup$ (**P**.s * <E 4 5>)) Φ (**P**.p * "NKN")

5. Match "AAANBPPPPF" with the database, but ignore mismatch in the segment NBPPP if it is on a loop.

(**P**.p * "AAA") || ((**P**.p * "NBPPP") $\bigcup$ (**P**.s * <L 5 5>)) || (**P**.p * "PF")

6. Match a protein with secondary structure <L 20 40><E 10 30> that has the fragment "AAPQS" in the loop segment.

((**P**.s * <L 20 40>) Φ (**P**.p * "AAPQS")) || (**P**.s * <E 10 30>)

4.2. Expressing BLAST

As mentioned earlier in the paper, BLAST is a family of similarity searching tools. One of the tools called *blastp* is used for searching protein datasets. The blastp operation was simplistically expressed in Equation 2 in Section 3.3.2. However, we can express this operation at a finer detail as described below.

Consider a blastp query for matching the sequence "QAANVP". To express this query in PiQA, we introduce the following notation: Let Δ_k denote the set of all possible protein strings of length k. If we are considering proteins from only the basic 20 amino acids, then the size of Δ_k would be 20^k.

The first step of BLAST, in which we prune out all the *k*-mers below a certain threshold *T*, can be expressed as:

$\mathbf{A} = \upsilon_{(pos,\ score,\ len)} (\sigma_{score>T} (\mu_{match\text{-}sequence} (\mathbf{B} *_{BLOSUM62} \Delta_3)))$, where **B** = {"QAA", "AAN", "ANV", "NVP"}

The second step of BLAST, in which the database is searched to find hits that match with any of the *k*-mers from the previous step, can be expressed as:

$\mathbf{R} = \mathbf{A} *_{Exact} \mathbf{P}.p$

The third step of BLAST involves extending the hits to form high-scoring segment pairs (HSPs). The first version of BLAST extends the hits residue by residue on both sides. In the second version, a two-hit method is used which extends hits only when two of the hits are within a certain distance. There are several variations on the heuristic for this step. We express this step as:

$\mathbf{C} = \mathbf{R} \|_{Maxdist} \mathbf{R} \|_{Maxdist} \mathbf{R} \|_{Maxdist} \mathbf{R} \ldots$

We could incorporate a transitive closure for the match extension operator in the algebra so that all possible ways of extending the hits are captured in the algebra. BLAST stops extending the hits when the score of the extended hit drops below a certain value of the maximum that it had reached since the start of the third step. A simple selection can be used to only output matches with a minimum score from C. When writing this programmatically, a while loop structure can be used to stop the hit-extension process precisely when it is desired. This is much like the use of while in SQL even though it is not part of the relational algebra.

Operation	Approximate Cost	Notes
Match (*),	O(|A|*avg-size of a tuple*query length)	Assuming that a full scan of the input table A is required, and using an FSA-style pattern matching as in [12].
nest (υ), unnest(μ)	O(|A|*avg-size of a tuple)	Assuming a full scan of the input table A.
Set Operations ($\bigcup, \bigcap, -$)	O($\|\nu(A)\| + \|\nu(B)\|$)	The two inputs are sets A and B; assuming an in-memory hash-based evaluation.
Contains and not-contains (Φ, Ψ), and match extension (||)	O(|A| + | B| + $\frac{\|\nu(A)\| * \|\nu(B)\|}{\|A\|}$)	The two inputs are sets A and B; cost assumes an in-memory hash-based implementation for pairing matches, and a nested-loops like algorithm for joining match elements.

Table 1: Costs Formulae

Operator	Commutative	Associative	Distributes Over
$\bigcup$	Yes	Yes	$\bigcap$
$\bigcap$	Yes	Yes	$\bigcup, -, \Phi$
$-$	No	No	
Φ	No	No	$\bigcup, \bigcap, -$
Ψ	No	No	
||	No	Yes	$\bigcup, \bigcap, -$
υ	n/a	n/a	$\bigcup, \bigcap, -, \|\|, \Phi, \Psi$
μ	n/a	n/a	$\bigcup, \bigcap, -, \|\|, \Phi, \Psi$

Table 2: Properties of Operators

5. Query Evaluation

To generate an optimal query plan, a query optimizer needs to estimate the cost of the various operations, and also needs to exploit the properties of the operators to generate all possible equivalent query plans. We explore these issues in turn in this section.

5.1. Cost Model

Simple cost formulae for the various operations are shown in Table 1. In the formulae, we omit the cost of producing the output, which in a practical implementation should be estimated and added to each formula. We also assume that the match sequences are not empty.

5.2. Generation of Query Plans

In generating query plans, the optimizer considers various reordering of the operators. For the set of operators that we have proposed, Table 2 shows the properties that can be exploited in query optimization.

5.3. Example of Query Optimization

Next, we illustrate the generation of query plans using the following query: *Match the primary structure sequence AAANBPPPPF with the database, but ignore a mismatch in the segment NBPPP if it is on a loop*.

In our algebra, this query is expressed as:

(**P**.p * "AAA") || ((**P**.p * "NBPPP") $\bigcup$ (**P**.s * <L 5 5>)) || (**P**.p * "PF")

Since the match extension operator (||) is associative and distributive over the union operator ($\bigcup$), we can generate a number of equivalent algebraic expressions, as listed below:

i. (**P**.p * "AAA") ||
((**P**.p * "NBPPP") $\bigcup$ (**P**.s * <L 5 5>)) ||
(**P**.p * "PF')

ii. ((**P**.p * "AAA") || (**P**.p * "NBPPP") || (**P**.p * "PF")) $\bigcup$
((**P**.p * "AAA") || (**P**.s * <L 5 5>) || (**P**.p * "PF"))

iii. ((**P**.p * "AAA" || **P**.p * "NBPPP") $\bigcup$
(**P**.p * "AAA" || **P**.s * <L 5 5>)) ||
P.p * "PF"

iv. (**P**.p * "AAA") || ((**P**.p * "NBPPP" || **P**.p * "PF") $\bigcup$
(**P**.s * <L 5 5> || **P**.p * "PF"))

Observe that in executing each of the plans listed above, we would need to perform three match operations. Also notice that all the strings that are in the result of the expression are likely to be in the result of *each* of the match operations. Instead of evaluating each match by reading the entire protein table P, we may be able to optimize this query by picking one of the three match operations and using the set of proteins that it matches to constitute the set over which the other matches are evaluated. This set is likely to be smaller than the full dataset P, which may lead to a more optimal query plan.

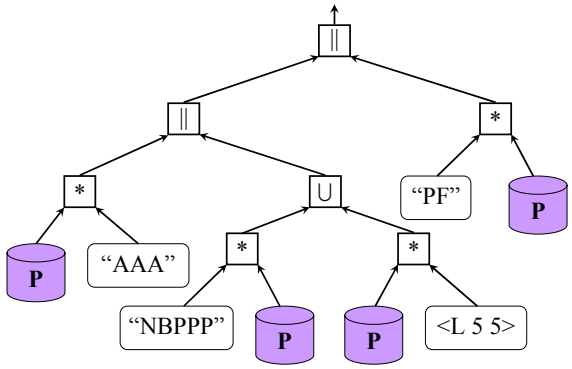

Figure 2: Query Plan for Sample Query

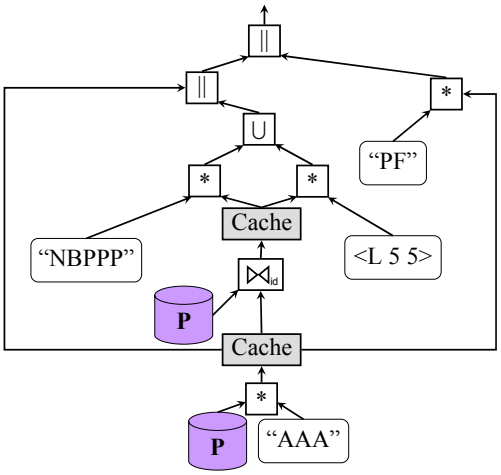

Figure 3: Alternative Query Plan

To pick an optimal query plan, the optimizer needs to accurately estimate the *selectivities* of the various predicates. In this example, we define selectivity as the fraction of the number of matching proteins in the data set that are selected by the predicate.

Consider the two predicates using "NBPPP" and "AAA" in the sample query. Let p_1 and p_2 denote these two predicates respectively. For the purpose of this example, consider the following protein table:

id	p	s
1	"AAATTRTAAUNBPPPPSTTT"	…
2	"PSSSQRRTTSTRRASAUWVV"	…
3	"UIPPSTTTGGGNBPPPTTTARAR"	…
4	"QQQPPLSSTTRRWRNNNBBB"	…
5	"AQATTTNBPPPVVVWUIPLAAR"	…

In this example, the selectivity of the predicates p_1 is 0.6 while that of p_2 is 0.2. Consequently, it is beneficial to evaluate the p_2 before evaluating p_1.

Two alternative query plans for evaluating the sample query are presented in Figures 2 and 3. In Figure 3, we use an operator *Cache* to store an intermediate result (either in memory or on disk) which can be consumed by multiple operators. If the selectivity of the predicate using "AAA" is significantly lower than the other predicates, the plan shown in Figure 3 is likely to be more efficient than the plan shown in Figure 2.

Finally, we note that if the query is required to produce all matches above a certain score, then the query optimizer is limited in the number of alternative query plans that it can consider.

6. Conclusions and Future Work

In this paper we have presented PiQA, an algebra for expressing queries on both the primary and secondary structures of proteins. The algebra provides a rich set of operators that permits approximate matching, combination of two or more matches, and various set operations on the matches. The algebra provides a unified approach to querying on both primary and secondary structures of proteins, and can also be used to optimize complex queries.

The next step in our research is to design and implement algorithms for the PiQA operators. We plan on building these algorithms in a system called Periscope that we are building to manage and query biological data sets. In addition, we also plan on designing and implementing a query language, called PiQL, based on the proposed algebra.

PiQA is the first step in developing a more comprehensive declarative querying interface on *all* protein structures. In the future, we plan on extending the algebra to allow querying on the tertiary and quaternary structures of proteins. Currently, the tertiary and quaternary structures are only known for a small subset of proteins [6], but this is likely to change in the future, making a fully integrated querying approach very useful.

In addition, in the future we also plan on investigating more complex probabilistic operations on sequences, and data types such as trees (e.g. phylogenetic trees) and graphs (e.g. protein interaction maps) that are often produced using sequence analysis. A key challenge is to keep the algebra manageable (i.e. implementable), while extending its functionality.

Finally, we note that although this paper concentrates on the protein structures, the algebra can also be used for querying nucleotide sequence data sets, which are similar in nature (from the querying perspective) to the protein primary sequences.

7. Acknowledgements

This work is supported in part by an IBM Faculty Award. We would like to thank the anonymous reviewers of SSDBM, H. V. Jagadish, Magesh Jayapandian, and Sidharth Sivasailam for their valuable comments and suggestions on earlier drafts of this paper.

8. References

[1] "Growth of GenBank," in http://www.ncbi.nlm.nih.gov/Genbank/genbankstats.html: NCBI (National Center for Biotechnology Information), Feb 2002.

[2] S. Abiteboul and N. Bidoit, "Non 1st Normal-Form Relations - an Algebra Allowing Data Restructuring," Journal of Computer and System Sciences, 33 (3), pp. 361-393, 1986.

[3] B. Alberts, Molecular Biology of the Cell, 4th ed. New York: Garland Science, 2002.

[4] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic Local Alignment Search Tool," Journal of Molecular Biology, 215 (3), pp. 403-10, 1990.

[5] S. F. Altschul, et al., "Gapped BLAST and PSI-BLAST: A New Generation of Protein Database Search Programs," Nucleic Acids Research, 25 (17), pp. 3389-402, 1997.

[6] H. M. Berman, et al., "The Protein Data Bank," Acta Crystallographica, D58, pp. 899-907, 2002.

[7] E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," Communications of the ACM, 13 (6), pp. 377-387, 1970.

[8] L. S. Colby, "A Recursive Algebra and Query Optimization for Nested Relations," presented at SIGMOD'89, Portland, Oregon, USA, 1989, pp. 273-283.

[9] S. B. Davidson, "Tale of Two Cultures: Are There Database Research Issues in Bioinformatics?," presented at SSDBM'02, Edinburgh, Scotland, UK, 2002, pp. 3.

[10] L. Fegaras and D. Maier, "Optimizing Object Queries Using an Effective Calculus," ACM Transactions on Database Systems, 25 (4), pp. 457-516, 2000.

[11] R. Hakli, M. Nykänen, H. Tamm, and E. Ukkonen, "Implementing a Declarative String Query Language with String Restructuring," presented at PADL'99, San Antonio, Texas, USA, 1999.

[12] L. Hammel and J. M. Patel, "Searching on the Secondary Structure of Protein Sequences," presented at VLDB'02, Hong Kong, China, 2002, pp. 634-645.

[13] J. Hammer and M. Schneider, "Genomics Algebra: A New, Integrating Data Model, Language, and Tool for Processing and Querying Genomic Information," presented at CIDR'02, Asilomar, California, USA, 2002, pp. 176-187.

[14] G. Jaeschke and H.-J. Schek, "Remarks on the Algebra of Non First Normal Form Relations," presented at PODS'82, Los Angeles, California, USA, 1982, pp. 124-138.

[15] F. Moussouni, et al., "Database Challenges for Genome Information in the Post Sequencing Phase," presented at DEXA'99, Florence, Italy, 1999, pp. 540-549.

[16] C. A. Orengo, A. E. Todd, and J. M. Thornton, "From Protein Structure to Function," Currrent Opinions in Structural Biology, 9 (3), pp. 374-82, 1999.

[17] Z. M. Özsoyoglu and J. Wang, "A Keying Method for a Nested Relational Database Management System," presented at ICDE'92, Tempe, Arizona, USA, 1992, pp. 438-446.

[18] M. A. Roth, H. F. Korth, and A. Silberschatz, "Extended Algebra and Calculus for Nested Relational Databases," ACM Transactions on Database Systems, 13 (4), pp. 389-417, 1988.

[19] P. Seshadri, M. Livny, and R. Ramakrishnan, "Sequence Query Processing," presented at SIGMOD'94, Minneapolis, Minnesota, USA, 1994, pp. 430-441.

[20] P. Seshadri, M. Livny, and R. Ramakrishnan, "SEQ: A Model for Sequence Databases," presented at ICDE'95, Taipei, Taiwan, 1995, pp. 232-239.

[21] S. A. Teichmann, A. G. Murzin, and C. Chothia, "Determination of Protein Function, Evolution and Interactions by Structural Genomics," Currrent Opinions in Structural Biology, 11 (3), pp. 354-63, 2001.

[22] S. J. Thomas and P. C. Fischer, "Nested Relational Structures," in Advances in Computing Research, vol. 3, 1986, pp. 269-307.

[23] J. L. Thorne, N. Goldman, and D. T. Jones, "Combining Protein Evolution and Secondary Structure," Molecular Biology and Evolution, 13 (5), pp. 666-73, 1996.