Towards Declarative Querying for Biological Sequences

Sandeep Tata[1] Jignesh M. Patel[1,*] James S. Friedman[2] Anand Swaroop[2,3]

Departments of [1]Electrical Engineering and Computer Science,
[2]Ophthalmology and Visual Science, and [3]Human Genetics
University of Michigan, Ann Arbor, MI 48109, USA
{tatas,jignesh,jfriedmn,swaroop}@umich.edu

Abstract

The current and ongoing revolution in life sciences research has lead to fantastic achievements such as the sequencing of entire genomes of various organisms. Sequences are ubiquitous in life sciences applications and the sizes of many sequence databases is growing rapidly. At the same time, the complexity of the queries that scientists want to pose on sequences is also increasing rapidly. Unfortunately current methods for querying biological sequences are primitive as they employ largely procedural querying methods, which limit the ease with which complex queries can be posed, and often result in very inefficient query evaluation plans. There is a growing and urgent need for declarative and efficient methods for querying biological sequence data sets. In this paper we introduce a system called Periscope/SQ which addresses this need. Queries in our system are based on an well-defined algebraic framework, which is an extension of the relational algebra. Our system is designed to allow leveraging the extensibility features of object-relational DBMSs, and in this paper we describe various design and implementation issues that arise when integrating our algebra. We currently have a simple prototype implementation of Periscope/SQ on top of Postgres, and in this paper we also demonstrate the applicability of our ideas by presenting a case study of an application in eye genetics that is currently using this prototype.

1 Introduction

The life sciences community today faces the same problem that the business world faced over 25 years ago. They are generating increasingly large volumes of data that they want to manage and query in sophisticated ways. However, existing querying techniques employ procedural methods, with life sciences laboratories around the world using custom Perl, Python, or JAVA programs for posing and evaluating complex queries. The perils of using a procedural querying paradigm are well known to a database audience, namely a) severely limiting the ability of the scientist to rapidly express complex queries, and b) often resulting in very inefficient query plans as sophisticated query optimization and evaluation methods are not employed. However, existing database products do not have adequate support for sophisticated querying on biological data sets. This is unfortunate as new discoveries in modern life sciences are strongly driven by analysis of biological data sets. Not surprisingly, there is a growing and urgent need for a system that can support complex declarative and efficient querying on biological data sets.

Before undertaking the task of designing a database system for biological data, it is essential to understand the nature of biological data and the types of questions that biologists want to ask on such data sets. A large volume of biological data deals with protein and DNA sequences. Proteins are molecules that perform vital roles in cell functions like metabolism, growth, repair, immune response etc. [3]. All proteins are essentially polymer chains of amino acid residues. Most proteins are made from about 20 basic amino acids, and a protein can be represented as a sequence of symbols over an alphabet of size 20. Protein databases store this sequence representation along with other information that may describe the geometrical folding properties of the protein, and information about the protein function (if known). DNA molecules on the other hand provide the genetic code which carries the blueprint for synthesizing these proteins. DNA molecules are chains of 4 different nucleotides. They can be represented as a sequence over the alphabet {A,T,G,C} - the four nucleotides.

There are several large databases worldwide that store protein and DNA sequence information. Some of these databases are growing very fast. For instance, GenBank, a repository for genetic information has been doubling every 16 months [10] (at a rate faster than Moore's law)! Protein databases, such as PDB [7] and PIR [6,29] have grown rapidly in the last few years. There is a growing need for efficiently running complex queries on these databases, and our work focuses on this issue.

*Author to whom correspondence should be addressed.

Proceedings of the 2005 CIDR Conference

Biologists try to analyze these databases in several complex ways. Similarity search is an important operation that is often used for both protein and genetic databases, although the way in which similarity search is used is different in each case. When querying protein databases, the goal is often to find proteins that are functionally similar to the protein being studied. Such functional similarity can yield important information about the role of the "query" protein in the cell. The computational criteria for specifying similarity is approximate, and includes similarity based on the amino acid sequence structure of the protein, or similarity based on the geometrical structure of the protein, and often a combination of these different structural representations. With genetic databases, scientists are interested in approximate similarity searches. Often with genetic databases the query is used to find patterns that may identify portions of the genome that contain information for synthesizing proteins. With both protein and genetic databases, since pattern matching is based on an approximate match model, the desired output is a ranked order list of results.

The above discussions highlights that biological sequences are often queried for sophisticated approximate matches to queries that are specified as a string pattern. A natural question to ask is whether regular expressions can be used to support such pattern matching operations. In fact, for lack of better alternatives, regular expressions were often used to search biological sequence data until a few years ago. However, regular expressions are inadequate for approximate querying. They don't capture complex substitution models, such as using an arbitrary substitution matrix for scoring matches [9], or even a simpler edit distance based similarity matching. In addition, with regular expressions there is no support for approximate querying, which requires returning a ranked order list of results. Finally, using regular expressions for complex queries can be very inefficient since indexing and optimization methods are typically not leveraged in this setting.

The advent of heuristic algorithms like BLAST [4,5] has made it possible to quickly search for approximate matches with a more complex similarity model, such as a substitution matrix. However, BLAST too has its limitations. One can only input a search string, and look for approximate *hits* to that string in the database. One cannot look for more complex patterns such as one query sequence separated from another query sequence by a certain distance, or a query sequence with some constraints on other (non-sequence) attributes. Researchers often ask such queries, and much of their time is spent in post-processing results from tools like BLAST to check for such constraints.

Clearly a declarative querying system that allows the scientist to pose complex sequence queries can make the task of querying biological sequence databases much easier. In addition, such a system can also make use of sophisticated query evaluation methods that could allow for very efficient evaluations of complex queries. In this paper, we describe the design of a system called Periscope/SQ, which takes on the challenge of building a declarative and efficient query processing tool for biological sequences. Periscope/SQ is part of a larger research project, called Periscope, which has the goal of designing, implementing, and evaluating a declarative and efficient querying engine for querying on *all* protein and genetic structures [19]. For proteins the structures include not only sequence structure but also various geometrical structures that describe the protein folding patterns and detailed layout of each atom in a protein molecule in 3D space. The SQ component stands for "Sequence Querying" and is the focus of this paper.

The main contributions of this paper are as follows:

- We examine the shortcomings of the current querying paradigm in biological databases, and identify the need for an efficient and declarative querying system.
- We propose an algebra that is a simplification of our previous work [27] for expressing complex approximate sequence matching queries. We go beyond our previous work and also present how this algebra is amenable for implementation in an object-relational DBMSs.
- We present extensions to SQL to accommodate the new operators in the algebra.
- We describe the design of the Periscope/SQ system and our plans for implementing our algebraic framework on top of Postgres [1, 25].
- We describe several avenues for optimization that can be exploited for evaluating queries expressed in our system.
- We presents a case study of an actual application in eye genetics that is currently using our system, and demonstrate through a simple performance study the advantages of the Periscope/SQ approach.

The remainder of this paper is structured as follows: Section 2 describes other related work. Our proposed algebraic framework is described in Section 3. Section 4 discusses language extensions to SQL, issues in optimization, and the overall design of our system. Section 5 is a case study of a life sciences application that is currently using our prototype implementation, and finally Section 6 presents our conclusions and directions for future work.

2 Related Work

A closely related previous effort is the work by Hammer and Schneider [14], which outlines an approach to expressing complex biological phenomenon through algebraic operations. Their approach aims to build a completely new algebra that is very powerful in expressing *all* biological operations such as transcription, translation, crossover, mutations, etc. However, our approach more carefully charts out the operations for querying sequences and aims at extending relational algebra so that we can take advantage of all the existing relational infrastructure.

In [13], the authors propose an alignment calculus on strings to query string databases. They also describe a system that was built based on this algebra [12]. The language

lets a user express very complex queries, by permitting complex string processing predicates to be written using alignment calculus declarations. However, the notion of an approximate match is hard to capture in this context. Also, to our knowledge, no performance evaluations have been carried out for this system.

Miranker et. al. suggest an approach for querying biological sequences in [18]. They borrow some constructs from our previous algebraic proposal, called PiQA [27], to describe complex queries, and largely focus on designing and exploiting efficient metric space indexing structures for querying sequences.

Previous work in querying sequences by Seshadri, Livny, and Ramakrishnan [22, 23], describe techniques for storing and declaratively querying sequences. However, this work is tailored towards handling time series style data where windowing, projecting, aggregating over subsequences are important. In our work, we are interested in operations on biological sequences which are quite different as it involves approximate pattern matching queries with complex match models.

There is a rich history of work on nested relational algebras [2,16,21,28], and our algebraic constructs are based on some of the concepts in these papers. In contrast to this previous work, our focus is on biological sequence querying which leads to a richer class of approximate pattern matching operators.

3 PiQA - The Protein Query Algebra

Expressing complex pattern queries and sophisticated predicates on biological sequences is awkward, and often impossible with current relational systems. The *like* predicate, which has traditionally been popular for string matching is inadequate for approximate matching queries. In addition, queries involving the like predicate are often slow. Our approach to this problem is to extend the relational algebra so that it can support complex and approximate pattern matching on biological sequences naturally. The extensions we propose are minimal - so that they can be incorporated in an existing object-relational system with a small amount of effort.

3.1 Types and Operators

Our algebraic extensions are based on the Protein Query Algebra (PiQA) [27]. PiQA permits queries on both the primary and secondary structures of proteins. The primary structure of a protein is simply the sequence of amino acid residues that constitutes the protein. Since there are 20 different amino acids, the primary structure can be viewed as a string with characters from an alphabet of size 20. The secondary structure uses a one-dimensional string structure to represents how this sequence of amino acid residues folds in space. There are three kinds of folding patterns: alpha helices, beta sheets, and turns or loops, and the secondary structure can be viewed as a string with characters from an alphabet of size 3. Knowledge of both the primary and secondary structure is important to understand the protein's function. Table 1 shows three proteins with their primary structure (p) and their secondary structure (s).

In our work presented in this paper, we focus on a subset of PiQA that largely deals with primary sequence querying. We go beyond our previous work and present a simpler form of PiQA and also present enhancements to make it amenable for integration with a relational DBMS.

3.1.1 Types

There are two new types in PiQA: the hit and the match. These types are described below:

Hit A hit is basically a triple *(p,l,s)*. When specified together with some sequence, the hit *(p,l,s)* means that there is a *hit* at position *p* of length *l* with a score of *s* on the given sequence. For instance, suppose that *A* = (2,3,3) is a hit on the sequence *SEQ* = "TGGTTTAGGAGGTA". This hit refers to the "GGT" substring, which could have matched some query for a score of 3. This hit can be shown in the original database sequence as "T**GGT**TTAGGAGGTA", with the hit portion highlighted in bold-face.

Match A match is simply a set of hits. For example, consider the sequence *SEQ* = "TGGTTTAGGAGGTA", and a query to find "GGT" followed by a "GGA" within 10 symbols. A match for this query using an exact matching paradigm is *X*= {(2,3,3), (8,3,2)}. This match describes two hits in the data sequences as shown in bold-face in "T**GGT**TTA**GGA**GGTA".

Several operations are defined on the Match type, as listed below:

Start(match) is the lowest *p* value of all the hits in the match.

End(match) is the highest *p+l* value of all the hits.

Length(match) is End(match)-Start(match).

Flatten(match, *f*) is the match {(Start(match), Length(match), *f*(match)}, where *f* is a score-combination function.

3.1.2 Operators

A formal definition of the operators and a detailed discussion can be found in [27]. Here we only provide a brief description.

Match ($*_{params}$) The match operator operates on a relation with a string attribute, a string, the attribute to operate on, and a match model. The resulting relation is the input relation augmented with a match attribute. The match attribute simply contains the set of all hits to the sequence in that row in *R* that matched the query string under a specified match model. We illustrate this with an example. Suppose that we have a relation *R(id,p,s)*: *id* is a unique identifier, and *p,s* are strings (representing the primary and the secondary

id	p	s
1	GQISDSIEEKRGHH	HLLLLLLLLLHEE
2	EEKKGFEEKRAVW	LLEEEEEHHHHHL
3	QDGGSEEKSTKEEK	HHHHLLLEEEELLL

Table 1: Relation R

id	p	s	match
1	GQI...	HLL...	{(8,3,3)}
2	EEK...	LLE...	{(1,3,3),(7,3,3)}
3	QDG...	HHH...	{(6,3,3),(12,3,3)}

Table 2: S = R$*_{EX}$"EEK"

structure). Let *str*="EEK" be the query string. *R* $*_{p,MM}$ *str* produces a relation *S(id,p,s,match)* . The subscript *p* specifies the string attribute in the relation to match on, and the subscript *MM* specifies the match model. Section 3.2 discusses match models in detail. Example instance of *R* is shown in Table 1, and the result of the match operation is shown in Table 2.

Instead of matching with just a string, one can also use a simple regular expression in the match operator. The details are in [27].

Nest(ν), Unnest(μ) These are the standard nest and unnest operators used in relational algebra with set-valued attributes. The semantics are identical to the description in [21, 27]. They flatten out the set-valued attribute (match in this case) to an atomic attribute (hit) by creating a row for each element in the set-valued attribute with all the atomic attributes that existed in the original table.

Match-Augmentation($||_{d^-,d^+}$) This operator operates on two relations (say *R1* and *R2* - both having a match attribute), and produces a new relation that contains all the non-match attributes, a new match attribute, and a key/id attribute. The match attribute is the union of the match of the left relation and a match on the right relation if the one from the right relation has the same (specified) id-field, and is between a distance of d^- and d^+ after the match of the relation on the left. When this distance range is omitted, it is assumed to be zero (for adjacent matches). If the match field in an operand contains several hits, then the operator computes *flatten(m)* and uses that value for computation. As is obvious, the augmentation operator needs to be given the list of attributes in the two table that need to be equal before it can compute a tuple in in the result relation. For ease of presentation, we omit this in future examples unless it is non-obvious.

The operator tags the rows of both input tables with a unique attribute, and then computes the tuples for the result relation. A unique field for the result relation is constructed by concatenating the unique fields of the input relations. This works to ensure that two distinct matches to the same sequence do not get combined into one set. This would happen if two rows were generated with identical atomic attributes and different match attribute values. Since the algebra is in PNF [21] (a discussion of this follows), these rows would get combined if not for the unique identifier that is added.

Example: Consider the relations S and T shown in in Tables 2 and 3 respectively. The table V produced by a match-augmentation: *S* $||_{0,10}$ *T*, is shown in Table 4.

id	match
1	{(3,3,2)}
2	{(10,3,3)}

Table 3: Relation T

matchid	S.id	S.p	S.s	T.id	match
V1	2	EEK..	LLE..	2	{(1,3,3), (10,3,3)}
V2	2	EEK..	LLE..	2	{(7,3,3), (10,3,3)}

Table 4: V = $S||_{0,10}T$

Other Operators Contains(Φ), Not-Contains(Ψ), and the extended versions of Union($\cup^e$), Intersection ($\cap^e$) and Difference($-^e$) operators are defined in the same sense as in [27]. Union, Intersection, and Difference operators have an extended definition in the context of operating on relations in Partition Normal Form (PNF). This is explained in detail in [21]. The contains operator only returns those matches in the left relation that completely contain some match in the right relation. The not-contains operator returns the exact opposite.

To illustrate the use of the PiQA algebra consider the following two queries:

Query1: Find all proteins that match the string "VLLSTTSSA" followed by matching the string "REVWAYLL" such that a hit to the second pattern is within 10 symbols of a hit to the first pattern. The secondary structure of the fragment should contain a loop of length 5.

Π_{id}(((Prots.p $*_{MM}$ "VLLSTTSSA") $||_{0,10}$
(Prots.p $*_{MM}$ "REVWAYLL")) Φ
(Prots.s $*_{MM}$ "LLLLL"))

Query2: Find all positions that match the string "TGCAT", followed by a match to the string "TAAT" within 50-250 symbols of a hit to the first string, followed by a match to the string "CA" 50-250 symbols after a hit to the previous string.

$\Pi_{id,match}$((DB.seq $*_{MM}$ "TGCAT") $||_{50,250}$
(DB.seq $*_{MM}$ "TAAT") $||_{50,250}$
(DB.seq $*_{MM}$ "CA"))

We observe that the match-augmentation operator can be expressed as a join operation with a complex condition on the match attribute and the id-attribute followed by a projection. For instance, the example above that produced V as in Table 4, which was written as $S||_{0,10}T$ can

be rewritten in terms of the basic operators. The expression can be computed as: $S \rhd \lhd_{COND} T$ where COND = (S.id=T.id AND *Aug*(S.match,T.match,0,10)!= NULL). *Aug* is a function that computes the augmented match when it exists and is null otherwise. From the S.match and T.match attributes of the resulting joined relation, the Aug function can compute the column that the || operator computes. Similarly, the contains, and the not-contains operators can also be expressed as joins followed by a projection. Therefore, the only real additions to relational algebra are the hit and match types and the match operator. The increase in expressive power comes from them. The other operators provide a syntactically compact way of representing complex operations involving the new types. Nest and Unnest of course, can't be expressed in terms of anything else and are vital to the algebra.

It is useful to note that the new operators are defined in a way so that the result is always a relation in Partitioned Normal Form (PNF) [21]. This is ensured as the augmentation operator generates a unique key for each row of the input relations and uses them to generate a unique id for the output relation. The notion of using identifiers has been discussed in the object-relational community [11]. The fact that the algebra is in PNF makes nest and unnest inverses of each other. It is possible to work with relations outside PNF with a more careful use of nest and unnest operators (and not requiring that the augmentation operator generate a unique key). However, we chose to keep our algebra in PNF as optimization of nest and unnest is easier when the algebra is in PNF.

3.2 The Match Model

Since one of the crucial operations in querying biological sequences is local-similarity search, one needs to pay close attention to the match model. There are several match models that are commonly used. These include exact match model, k-mismatch model, and substitution matrix based models with gap penalty models. An exact match model simply requires that we find exact matches for the query with any substring in the database. A *k*-mismatch model allows for at most *k* mismatches between the query and any database substring. Insertions and deletions are not permitted with this model. Finally, the general substitution matrix based models use a substitution matrix that specifies the precise score for matching any two pairs of symbols/characters. Insertions and deletions are permitted, and often the gap penalty model may use an *affine* gap penalty model, which scores gaps by assigning a high penalty for opening a gap, but has a very low penalty for continuing a gap. A more detailed discussion of various matching models is beyond the scope of this manuscript, and we refer the interested reader to an excellent treatise on this subject [9].

At the level of the algebra, we can leave the actual match model unspecified. The expressive power of the algebra is not dependent on the complexity of the match model. Note that so long as we are able to consistently specify score combination functions for an operator using just the hits or matches, we can use any match model. We shall illustrate with an example. Suppose that the match model permits gapped matches. Further, assume that the gap penalty is linear, i.e., all gaps are penalized the same amount. Then, if we are interested in stringing together fragments of a match into a longer match, tolerating some gaps between them, a score combination function like f(m1,m2) = score(m1) + score(m2) -k$\times$(start(m2)-start(m1)-length(m1)) would make sense. We basically add the scores and award a penalty proportional to the gap between the two hits. However, if the match model used affine(non-linear) gap penalties, then a linear score combination function like this might not be consistent. However, in many cases it might be acceptable to use a simple score combination function even when a non-linear match model is used.

The substitution matrix based model with gap penalties is the most commonly used matching model in bioinformatics. Along with parameters for gap penalties, this class of similarity models can be used to do exact matching, *k* -mismatch style matching, and sophisticated similarity matching with matrices like the commonly used PAM and BLOSUM matrices [8, 15]. Using a matrix where substituting a symbol with a different symbol awards a large penalty will force exact matching. The subscript *EX* is used to denote an exact matching model. If we use unit positive score for substitution with the same symbol, and a unit negative score for all other symbols, this matrix would imitate the *k*-mismatch model. We could specify *k* by setting an appropriate score cutoff. We use the subscript *KM(k)* to denote the *k*-mismatch model. The general substitution matrix is represented with the subscript *MM(Mat)* where *Mat* is the substitution matrix.

Finally, we note that other match models can be accommodated in our framework by introducing a new parameter for the Match operator, and adding appropriate algorithm(s) for the match operator.

4 Integration with a Relational System

Biologists often pose queries that involve complex similarity conditions as well as regular relational operations. For instance, a similarity query could have a group-by clause on the attribute that describes the family to which the protein belongs to. Often, one needs to process the results in various ways - collecting statistics on the scores, joining these results with other tables to get additional information about the matching sequences, or using the result table as an input to a new query. Consequently, it is highly desirable that both regular relational querying and complex approximate sequence querying facilities be available within a single system and language. The best way to achieve this goal is to extend an existing object-relational DBMS [26] to include support for the complex and approximate pattern matching operations.

The rest of this section describes the design of Periscope/SQ system. This system is still in its initial development stages, and in the following section we outline the challenges that lie ahead in building this system.

4.1 System Design

We are just starting to implement Periscope/SQ on top of Postgres [1]. Postgres is a good choice for several reasons. It is an open source project and there is an active community of developers working and supporting it. The relational query processing facilities in Postgres is stable and the query optimization with Postgres is more robust than other open sources database systems. More importantly, Postgres is an object-relational database system with an extensible type system. In addition, new index methods can also be added to the system with some effort.

To implement the algebraic extensions described in Section 3 we have to make a number of changes in Postgres. First, we need to extend the type system to define a match type. We also need to add new operators for the match type, which includes first, last, flat, etc., as user defined functions. The Postgres SQL parser must also be extended to support language extensions for new operators such as the match operator. Our plans for extending the SQL language is described in Section 4.2.

Efficient algorithms for implementing the new operators, such as the match operator, is an active area of research. The Smith-Waterman algorithm [24] is the gold-standard algorithm for implementing the match operator. However, this algorithm is very slow and impractical for large sequence data sets. Novel algorithms, such as the recently proposed OASIS algorithm [17], which speeds up the Smith-Waterman computation by an order of magnitude or more are attractive alternatives. However OASIS requires the use of suffix tree indices, which implies that we must implement this new indexing mechanism in Postgres. In addition, when evaluating the match operation with a simple match model, a *w-gram* index can be very practical. (Section 5 discusses the use of a w-gram approach in more detail.)

Finally, we will also need to modify the Postgres optimizer so it can take advantage of the optimization issues that are discussed in Section 4.3.

4.2 PiQL - The Query Language

In this section we propose a Protein Query Language, called PiQL (pronounced as "pickle"). PiQL is an extension of the SQL language and incorporates the new data types and algebraic operations that are described in Section 3. The PiQL extensions to SQL are listed below:

Match Type: A new type called the match type can be used to define attributes that store match information. (Operations for match type can be implemented as user defined functions on this new data type.) As an example of the use of this data type, consider the following table definition which stores matches to a query:

CREATE TABLE prot-matches (pid INT,
p STRING, match MATCH_TYPE)

Match Operator: This operator can be implemented as a table function which takes all the arguments as described in Section 3 and returns a relation with the match attribute. An example using this operator is:

*SELECT **
FROM MATCH(prots,"VLLSTTSA",Match_Model)

Nest and Unnest: These operations can be implemented as table functions that take the relation and the list of attributes to nest/unnest on as an argument. For example, an expression like *Unnest(prot-matches, match)* will unnest the *match* attribute in relation *prot-matches*. Similarly, an expression such as *Nest(prot-matches, pid)* will nest the relation *prot-matches* with the *pid* as the simple key attribute.

Match Augmentation Operator: This operator can be implemented as a function that takes as arguments the two match fields, and the minimum and maximum distances. For instance the following query will find all matches that are the form "VLLSTTSA" followed by "REVWAYLL" with a gap of 0-10 symbols between them.

*SELECT *, AUGMENT(M1.match, M2.match, 0, 10)*
FROM
MATCH(prots.p,"VLLSTTSA", MM(PAM30)) M1,
MATCH(prots.p,"REVWAYLL", MM(PAM30)) M2

Contains, Not-Contains, and other operators: These functions are supported with a syntax similar to the Match Augmentation operator. In the interest of space, we omit these details.

As an example of a query in PiQL, consider the following query from before: Find all proteins that match the string "VLLSTTSSA" followed by matching the string "REVWAYLL" such that a hit to the second pattern is within 10 symbols of a hit to the first pattern. The secondary structure of the fragment should contain a loop of length 5. Only report those matches that score over 15 points.

This query can be expressed in PiQL as:

*SELECT *, CONTAINS(AUGMENT(*
M1.match, M2.match, 0,10),M3.match) AS resmatch
FROM
MATCH(prots.p,"VLLSTTSA", MM(PAM60)) M1,
MATCH(prots.p,"REVWAYLL", MM(PAM60) M2,
MATCH(prots.s,"LLLLL", EXACT) M3
WHERE score(resmatch) $\geq$ *15*

The three MATCH clauses correspond to the match operators that would be needed to search for each of the specified patterns. The inner AUGMENT function in the SELECT clause finds the patterns that have "VLLSTTSA" followed by the "REVWAYLL". The CONTAINS call makes sure that only those matches that contain a loop of length 5 get selected. The CONTAIN statement throws out the remaining matches.

4.3 Optimization

The complex pattern queries that can be expressed in PiQA can often be evaluated using a number of alternative query plans. The costs of these plans is often significantly different, and picking the optimal plan is a challenging task. There are several opportunities for query optimization in our setting, which we discuss in this section.

4.3.1 Algorithms for Match

There are several choices for the algorithm that we can use to evaluate the match operator. The Scan algorithm is the simplest of them all - it scans the sequence data set from start to finish and compares each sequence with the query pattern for an exact match. With a complex match model, such as a *k*-mismatch model, a Finite State Automaton (FSA) is constructed for the query, and each sequence is run through this automaton. The OASIS [17] algorithm is a suffix tree based technique that can be used to quickly find matches to a query sequence for any given substitution matrix. The Smith-Waterman [24] algorithm is a dynamic programming based approach to finding local alignments and can be used even when there is no suffix tree index. The BLAST [4, 5] family of algorithms are a heuristic approaches to local-similarity searching that run fast and find *most* matches for a given query. Table 5 summarizes the different choices of algorithms for the match operator.

We also suggest another operator for matching when combined with the match augmentation operator. This match-and-augment operator can be used when you want to extend a set of matches with another set of (say exact) matches on the same dataset. For instance, consider the following expression:
AUGMENT(MATCH(A.seq,"ATTA",MM(BLOSUM62)), MATCH(A.seq."CA",EXACT), 0,50).

One way to compute this expression is by evaluating the first MATCH, then scanning 50 symbols to the right of each match that is found, and checking for the occurrences of "CA". We select and augment only those matches where we find the "CA". This is the match-and-augment operator. This operator can often be cheaper than performing two matches separately and combining the results with the augment operation. A rule that rewrote queries using this new operator can produce higher quality query plans in many cases. However, note that this rewrite can be used when the new string to be matched is using the exact or the *k*-mismatch model (which often happens when querying genetic sequences, and very rarely when querying protein sequences). The FSA-Scan algorithm cannot be used with the substitution matrix model.

Choosing the right algorithm can not only impact the performance greatly, but sometimes also the accuracy. If BLAST is used, then there is a possibility that some of the hits might be missed - so this algorithm should be used only in cases when is this acceptable. Smith-Waterman and OASIS on the other hand never miss matches and can always be used in all situations, though these algorithms can be more expensive to execute in some cases.

Match Model	Algorithms
Exact	Scan, OASIS
K-Mismatch	Scan(FSA), OASIS
Substitution Matrix	BLAST, OASIS, Smith-Waterman

Table 5: Algorithms for Match

4.3.2 Statistics and Result Size Estimation

As with any complex query processing system, the Periscope/SQ query optimizer is critically dependent on accurate methods for result size estimation. For instance, the number of hits a match operator produces is an important factor in determining the order in which different augment operations are evaluated. It is also useful in determining whether certain operator reordering or query rewrite rules lead to more efficient query plans. Similarly, being able to estimate the result size of a match augmentation operator is also important in deciding the order in which augmentation should be performed.

Next, we demonstrate the usefulness of accurate estimation methods using the following example:
(A.Seq $*_{MM(BLOSUM62)}$ *str1)* $||_{0,50}$ *(A.Seq* $*_{KM(2)}$ *str2)* $||_{0,50}$ *(A.Seq* $*_{EX}$ *str3)*

Suppose that the lengths of str1, str2, and str3 are 15, 7, and 2 respectively. This expression can be evaluated in several ways. We present three such plans in Figures 1 to 3. In Plan1 (see Figure 1), each of the match operators are computed separately, the results of the first and second are augmented. To this result, the results of the third match operation are augmented. Plan2, which is shown in Figure 2, involves augmenting the second and third sets of matches first, and then to the first set, augmenting this combined set of matches. The last plan, Plan3 shown in Figure 3, uses the match-and-augment operator for str2 and str3.

Using a simple model for the cost of each plan, we can show that the running times of these plans can vary significantly. Let $|D|$ denote the total number of symbols in the sequences of $A.Seq$. The cost of evaluating a match operator is $O(|D| \times q_{eq})$ when using a scan of the database using an FSA, where q_{eq} is the expected number of states of the FSA traversed before deciding if the substring is a hit or not. For the sake of this discussion, we assume that there are no indexes. We also assume that the cost for evaluating the first match operator - where the substitution matrix is used - is also the same. For the purpose of this example, we make a simplifying assumption that the database sequence is produced by a source outputting one of four symbols {A,T,G,C} distributed uniformly.

Assume that the number of matches that each of these match operators result in is $M_1, M_2,$ and M_3. Using a nested loops style algorithm, the simple match augmentation operator can be implemented in time proportional to $O(|R| \times |S|)$. The cost of Plan1 is : $c \times |D| \times (q_{1eq} + q_{2eq} + q_{3eq}) + M_1 \times M_2 + M_X \times M_3$, where M_X is the number of matches in the result of augmenting M_1 with M_2 and c is a constant of proportionality.

Each of the match operators contribute $O(|D| \times |q|)$,

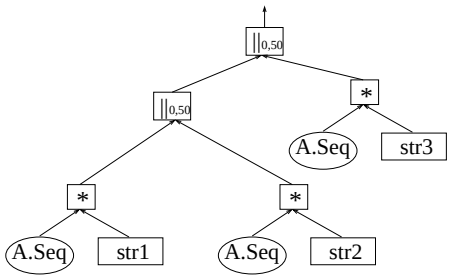

Figure 1: Plan 1

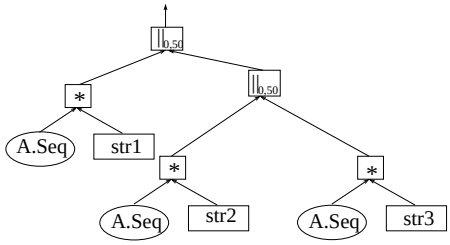

Figure 2: Plan 2

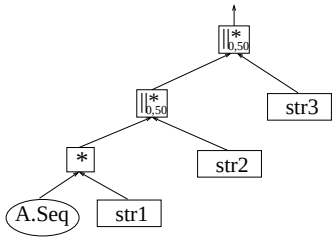

Figure 3: Plan 3

making up the $|D| \times (q_{1eq} + q_{2eq} + q_{3eq})$. The first augmentation costs $(M_1 \times M_2)$ and the second augmentation costs $(M_X \times M_3)$. Similarly, the cost of Plan2 is $c \times |D| \times +M_2 \times M_3 + M_Y \times M_3$, where M_Y is the size of the result of augmenting M_2 with M_3. Finally, Plan3 uses the match-and-augment operator, and its cost is $c \times |D| \times q_{1eq}) + M1 \times 50 \times 7 + M_X \times 50 \times 2$.

In order to compare the actual costs, we need to estimate the values of M_1, M_2, and M_3. Now, using simple probability, we can estimate that the probability for exact matches in each of these cases is : $\frac{|D|}{4^{15}}$, $\frac{|D|}{4^7}$, and $\frac{|D|}{4^2}$. Next, we need to adjust for the fact that the first match is an approximate match, and that the second match tolerates *2* mismatches. Estimating the number of approximate matches in a DNA sequence or a protein sequence given a query string is a non-trivial exercise. For simplicity, in this paper we use a uniform probability model, and employ a very simply estimation model. Assume that the estimate for M_1 is $\frac{|D|}{3^{15}}$ and for M_2 is $\frac{|D| \times 330}{4^7}$. Estimating M_X and M_Y too is a difficult exercise, and for the purpose of this example let us assume that M_X is 8 and that M_Y is 3 million. (A long string is likely to have far fewer matches than a short string). Now, if we assume that $|D|$ is 100M symbols, we can estimate different values as summarized in Table 6.

Next, we need to estimate q_{eq}. Using the uniform distribution assumption, and some probabilistic calculations, one can compute this approximately. In the case of the first match operation, we need to use a substitution matrix based algorithm. Assuming that such an algorithm would need to look at all the symbols of the string, we estimate q_{1eq} to be 15. For the second match operation, assuming that *k* is 2, we calculate q_{2eq} to be 3.33. Finally, q_{3eq} is 1.25. Assuming that *c* is 1/3 (since the cost of comparing two characters is lower than the cost of computing an augment function), we can compute the relative costs of each plan. These costs are summarized in Table 7. Note that the costs shown in this table are simply estimates of the computational cost of each plan, and does not include disk I/O costs - a more sophisticated cost model is needed for modeling I/O costs.

As we can see from Table 7, the cheapest plan is Plan3 which uses the match-and-augment operator twice. Plan2 is very expensive, primarily because of a poor choice for the order of the join (augment) operators. This example also illustrates that in some cases rewriting a series of

Expression	Estimate
$\|D\|$	100M
M_1	7
M_2	2M
M_3	6M
M_X	8
M_Y	3M

Table 6: Estimates of Intermediate Result Sizes

Plan	Expression	Cost
1	$c \times \|D\| \times 19.58 + M_1 \times M_2$ $+M_X \times M_3$	714×10^6
2	$c \times \|D\| \times 19.58 + M_2 \times M_3$ $+M_Y \times M_1$	12×10^{12}
3	$c \times \|D\| \times 15 + M_1 \times 50 \times 7$ $+M_X \times 50 \times 2$	500×10^6

Table 7: Query Plan Estimates

match and augment operations as a combined match-and-augment operation can be very beneficial.

In the presentation above, we have only presented a very simple model for estimating various selectivities. We have assumed that the database sequence is uniformly distributed over the genetic alphabet. This assumption does not hold in practice for real data sets. For instance, consider the strings "ATATATATATAT" and "GTGTCTATATAT". They are both of length 12. In chromosome 6 of the mouse genome, the first string occurs 8605 times, whereas the second string occurs just 42 times! An estimate that is off by an order of magnitude will clearly lead to poor plans. Clearly, we need more accurate techniques to estimate these selectivities. There has been a start in this direction by modeling DNA as a Markovian sequence and using methods as in [20]. We are exploring this and other approaches as part of our ongoing research.

5 Case Study

Our current implementation of Periscope/SQ involves a stand-alone module, called GeneLocator, which we describe in this section. GeneLocator implements the match, hit, and match-augmentation operators. The results produced by this module are loaded into Postgres for further querying using a traditional SQL engine. (In the future we

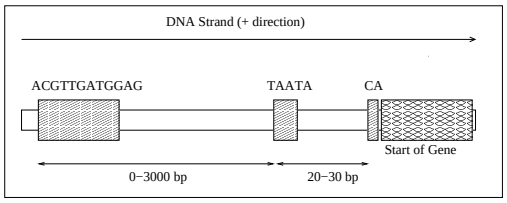

Figure 4: Promoter Binding Region

plan on extending Postgres directly as outlined in Section 4. The current prototype was built to quickly produce a working tool that could be used by our collaborators.)

GeneLocator is being developed in collaboration with researchers at the Kellogg Eye Institute at the University of Michigan to query genetic databases. In particular, GeneLocator is currently being used to find target promoter regions. In order to better understand the design and purpose of this application, a brief discussion of the relevant biological background is provided below.

5.1 Background

In a number of life sciences experiments, scientists working with genetic information are interested in finding portions of the genome that may code for certain genes. Our collaborators in eye genetics are studying degenerative eye diseases in older adults, and want to pose several sequence pattern matching queries to discover the genetic factors behind some of the age-related eye diseases. These researchers are interested in finding all the genes that are regulated by a particular transcription factor known to be present in the eye. A transcription factor is a protein that binds to the DNA at a particular site, causes transcription of a gene, and eventually, the synthesis of a protein. Transcription factors usually have a "signature" binding site: a short sequence of DNA about 10-15 bases long. This is seldom an exact signature - the factor will usually bind to sites that are slightly different from the signature (mismatches in a few positions).

A random string of 10-15 bases will have several tens of thousands of hits (allowing for 1-2 mismatches) on the genome of an organism like the mouse or a human. There are several other local factors that determine when a hit is interesting and a potential binding site. For instance, in the case of several transcription factors, there is a TATA-box (a pattern such as "TAATA") or a GC-box (a pattern like "GCGC") within a certain distance of the binding site. Finding such a pattern obviously increases the likelihood of a hit being a real binding site. Furthermore, transcription almost always begins at a "CA" site, which is a short distance following the TATA-box or the GC-box. Figure 4 pictorially represents the kind of pattern we are looking for.

In our initial searches, the match model that the scientists want to use is very simple. The signature for the first string (the binding site) is a *k*-mismatch model. The second string (like the "TAATA" or the "GCGC") and the third string "CA" use an exact search model. This query can be expressed as:

```
SELECT *, AUGMENT(AUGMENT(
  M1.match, M2.match, 0,2988),
  M3.match, 15,35) AS res
FROM
  MATCH(DB.dna,"ACGTTGATGGAG",KM(1)) M1,
  MATCH(DB.dna,"TAATA",EXACT) M2,
  MATCH(DB.dna,"CA",EXACT) M3,
  WHERE score(res) > 15
```

Typically the biologist will look at each of the results produced by the above query to check if the hit is "interesting", and if it worth designing an actual experiment to verify each result using in vivo methods. Since in vivo experiments are very expensive and time consuming, it is essential to employ computational methods to prune out hits that have a lower probability of producing interesting end results.

In practice we found that in many cases the above query produces a large number of matching results. To filter out result entries that may be less promising, we employed the following strategy: We took each hit result, and consulted an existing gene annotation database to check if the hit was within some distance upstream of any known gene. There are a number of gene annotation databases that contain information about portions of the genome that have been determined experimentally or predicted computationally as being coding regions. Such databases are available for several organisms from resources like NCBI.

We performed the additional "join" by simply loading the hit results and gene annotation datasets into Postgres and using a SQL query. The final result is equivalent to the following SQL query that uses the language extensions described in Section 4:

```
SELECT *, AUGMENT(AUGMENT(
  M1.match, M2.match, 0,2988),
  M3.match, 15,35) AS res, G.name
FROM
  MATCH(DB.dna,"ACGTTGATGGAG",KM(1)) M1,
  MATCH(DB.dna,"TAATA",EX) M2,
  MATCH(DB.dna,"CA",EX) M3,
  GeneAnnotations as G,
WHERE score(res) > 15 AND G.start > start(res) AND
  G.start - start(res) ≤ 5000 AND
  G.chromosome = DB.chromosome
```

In this query we assume that GeneAnnotations is a table with a schema like GeneAnnotations (id, chromosome, start, end, type, annotation), and the upstream distance criteria is 5,000 base-pairs.

5.2 Implementation Details

The GeneLocator module is currently implemented outside the Posgtres system, and supports a limited, but useful, set of pattern matching queries (these limited queries essentially cover the queries that the scientists in the eye genetics group currently want to pose). GeneLocator is accessed by

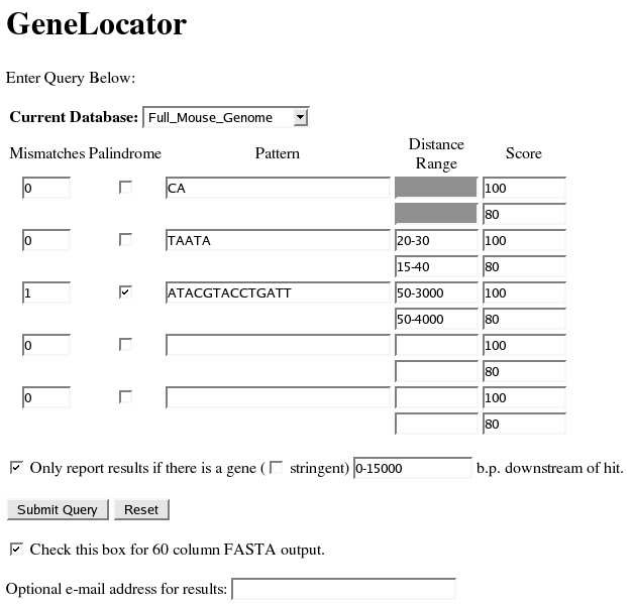

Figure 5: Screenshot of the GeneLocator Interface

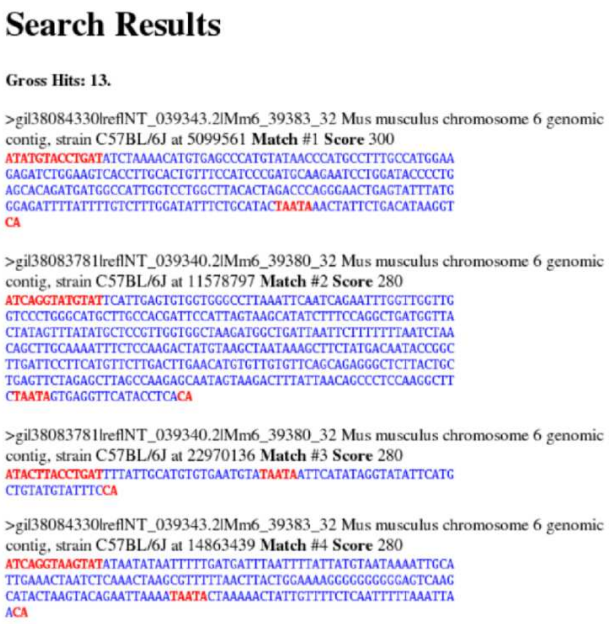

Figure 6: Screenshot of the Search Results

a web interface, which allows the end user to pose queries by filling out a simple form. A screenshot of this interface is shown in Figure 5.

The query is composed as follows: the user enters a pattern in the first text box. The second pattern is entered in the next box, and the distance *upstream* from the first pattern is specified. Therefore, in order to search for a pattern similar to the one in Figure 4, we start with "CA" in the first text box. The second pattern "TAATA" is entered in the second text box. Then we specify a distance range (20-30 in this case) from the last character of this pattern to the first character of first string. The third pattern is similarly specified in the third text-box. This convention is used because a "CA" often marks the transcription start site (the part where the actual gene starts) and the distances to all patterns are usually specified relative to it. An exact match model can be specified by entering a zero in the column for number of mismatches. The *k*-mismatch model can be specified by entering *k* in the box. Checking the palindrome check-box searches for the given pattern *and* its reverse pattern. A simple scoring system is used: two distance ranges are given for each pattern (except the first). Depending on the distance within which the hit is found, the corresponding score is added for that hit. In the case of the query entered in Figure 5, if the TATA-box is found within 20-30 bp, it contributes 100 points. Otherwise, if it is between 15-40 bp away, then it contributes 80 points. If it is outside of this range, it is not joined in the final result. The information from the distance ranges is used to construct the score combination function for the "Flat" operator on a match. A check-box on the interface could be used to restrict the matches to only those that had known genes within a user-specified distance downstream.

Since we had prior information about the kinds of queries that were going to be posed, we hand-crafted a query execution plan using the relative selectivities of the query strings. In this pore-optimized plan, we first evaluate the match operator on Pattern3 (the longest - therefore the most selective) string. The remaining two strings are matched using the match-and-augment operator. The match algorithm used for this is the FSA-Scan. As a first step, the algorithm is implemented in PERL, which has excellent support for regular expression based matching. Also, for simplicity our current implementation does not use any indices. The results of a query are displayed as shown in Figure 6.

An example query that was posed to GeneLocator had pattern-1 set to "CA", pattern-2 set to "TAATA", and pattern-3 set to "GATTACAGCTCGATC". The first two patterns had exact match models. The third one used a *k*-mismatch model with *k* set to 1. We used a machine with a 2.8 GHz Pentium 4 processor with 2 GB of main memory. On the entire mouse genome, which is about 2.5GB, the program took about 45 minutes to find all the matches. When the additional condition of reporting only those matches that had a gene within 5000 bp downstream of this hit was applied, the total time for the query was 56 minutes. (The actual strings in the query have been changed to protect the privacy of the research. However, they are representative of the of an actual query.)

Once the results of a query are obtained, the biologists use the results to tune the match model and make small changes to the patterns and ask another query. The style of querying is exploratory since much of biological querying is inexact, and looking at the results of one query may produce insights or additional questions, that require posing additional queries. We observed that in some cases the string for pattern-3 was shortened and *k* was set to 2.

In another cases, the pattern for the TATA-box was made longer, and one mismatch was tolerated. More sophisticated queries are being planned using ideas from comparative genomics where the users look for patterns across several genomes, and matches that appear across more genomes are ranked as higher-scoring matches. As the complexity of the queries increases, the simple FSA scan algorithm and our simple pre-optimized plan do not meet the required performance. In order to efficiently support exploratory querying, we examined a simple pruning strategy using indexing.

5.3 Speeding up Execution

It turns out that there are several opportunities for speeding up even these restricted class of queries using different indexing techniques. The data for the mouse genome is stored in chunks of sequences called *contigs*. Each chromosome is made of a few tens of contigs. Now, given a pattern if we could prune out several contigs that don't match the query, then we can speed up the execution considerably by searching only those contigs that remain. This is the basic idea that we use in the technique described in the remainder of this section.

A *w*-gram index has all the subwords of length *w* in the genome, and the corresponding contigs in which they occur. Such an index can be used to select the contigs that have potential matches for the entire query. Suppose that we have a string of length l that we were trying to match under a *1*-mismatch model. Assuming that l is even, every match to the string must have an exact match to either the first half or the second half of the sting. We use the index to look up all the contigs that contain either of these two *half-words*, and search only these contigs for the original pattern. We can save a significant amount of searching depending on how selective the exact match queries are. Note that by searching a few extra symbols at the ends of the contigs, we don't miss any matches that might be split across two contigs in the same chromosome.

This idea can be generalized to any *k* for the *k*-mismatch model. The approximate match to the string of length l will necessarily contain an exact match of length at least $\lfloor(\frac{l}{k+1})\rfloor$. (The proof for this is omitted in the interest of space). Instead of using an FSA scan with the approximate string for the entire database, we only select those contigs that have exact matches to to one of the substrings formed by equally dividing the original string into *k*+1 parts. For instance, consider the string "AGCAGAACCCAGTGTGTATGAACATCTCTT". When using a 2-mismatch model, it will be split into three equal contiguous parts: "AGCAGAACCC", "AGTGTGTATG", and "AACATCTCTT". We look up the appropriate *w*-gram index for the list of contigs that exactly match these words. We then scan only those contigs using the full FSA for the original word with *k* mismatches. (We are of course assuming that a *w*-gram index of the right length already exists).

We executed the above query for the 1-mismatch and the 2-mismatch models against the mouse genome. The mouse genome is roughly 2.5 billion symbols long. The execution times are summarized in Table 8. In the 1-mismatch case, the index based approach executes in 14 minutes, while the naive execution algorithm executes in 117 minutes: a speedup of over 8X. In the 2-mismatch case, the index based approach provided a speedup of nearly 12X!

Algorithm	***k***	**Time**
Full Scan	1	117 min
Index Based	1	14 min
Full Scan	2	3031 min
Index Based	2	255 min

Table 8: Execution Times

As we start using longer queries with larger *k*, the naive algorithm is prohibitively expensive. As can be observed from Table 8, the naive algorithm takes over two days to execute the sample query. Index based approaches are essential for executing such queries efficiently.

Even though with the sample query the *w*-gram approach provides significant improvements, it is likely to be very slow for queries which employ a k-mismatch model with a larger k value (the half-words generalization method degrades in relative performance as the value of k increases). To support interactive querying for more complex queries, we will need to employ even faster query evaluation methods. More complex match model, such as using an arbitrary substitution matrix, makes the problem even harder. We plan on investigating these research issues as part of our future work.

5.4 Results

By posing the relevant queries using GeneLocator, the eye genetics researchers were able to identify several genes that are potential targets for a particular transcription factor that is of interest to this group. This was achieved after using GeneLocator for only a few days, and involved firing several queries using the declarative querying interface that is describe in Section 5.2.

The next phase of the research project involves biologically verifying these results. This includes performing wet-lab experiments to check if the transcription factor actually binds to the site in the cell, and conducting experiments to study how it regulates the gene. These experiments can easily take months to execute, and often the scientist carefully need to design the parameters for these experiments (such as choosing a cell line for the validation, the actual protocol for the experiment, etc.). Experiments to verify the computational results produced by GeneLocator are currently in this design phase.

6 Conclusions and Future Work

In this paper, we have presented Periscope/SQ - a system which employs a database approach for declarative querying on biological sequence data sets. We have described an algebraic framework that is used by our system to allow complex and approximate pattern matching queries. We

have also described various issues related to implementing these extensions in an object-relational system. This paper also describes extensions to SQL to accommodate operators in our algebraic framework, and also describes challenges in optimizing these queries. Finally, we have also described an actual application that is using a simple prototype implementation of our proposed system.

As part of our future plans, we plan on investigating query evaluation algorithms and query optimization methods for supporting the entire PiQA algebra.

Acknowledgments

This research was supported by the National Science Foundation under grant IIS-0093059, and by a research gift donation from Microsoft. Additional support for this research was provided by National Institutes of Health grant EY11115, the Elmer and Sylvia Sramek Charitable Foundation, and the Research to Prevent Blindness Foundation.

References

[1] The PostgreSQL Database System. *http://www.postgresql.org*.

[2] S. Abiteboul and N. Bidoit. Non First Normal Form Relations: An Algebra Allowing Data Restructuring. *Journal of Computer and System Sciences*, 33(3):361–393, December 1986.

[3] B. Alberts. *Molecular Biology of the Cell*. Garland Science, 4th edition, 2002.

[4] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic Local Alignment Search Tool. *Journal of Molecular Biology*, 215:403–410, 1990.

[5] S. Altschul, T. Madden, A. Schffer, J. Z. andZ. Zhang, W. Miller, and D. Lipman. Gapped BLAST and PSI-BLAST: A New Generation of Protein Database Search Programs. *Nucleic Acids Research*, 25:3389–3402, 1997.

[6] R. Apweiler, A. Bairoch, C. H. Wu, W. C. Barker, B. Boeckmann, S. Ferro, E. Gasteiger, H. Huang, R. Lopez, M. Magrane, M. J. Martin, D. A. Natale, C. O'Donovan, N. Redaschi, and L. S. Yeh. UniProt: the Universal Protein Knowledgebase. *Nucleic Acids Research*, 32:D115–D119, 2004.

[7] H. Berman, T. Battistuz, T. Bhat, W. Bluhm, P. Bourne, K. Burkhardt, Z. Feng, G. Gilliland, L. Iype, S. Jain, P. Fagan, J. Marvin, D. Padilla, V. Ravichandran, B. Schneider, N. Thanki, H. Weissig, J. Westbrook, and C. Zardecki. The Protein Data Bank. *Acta Crystallographica*, D58:899–907, 2002.

[8] M. O. Dayhoff, R. M. Schwartz, and B. Orcutt. A Model for Evolutionary Changes in Proteins. *Atlas of Protein Sequence and Structure*, 5:345–352, 1978.

[9] R. Durbin, S. R. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis : Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, 1st edition, 1999.

[10] Growth of GenBank, National Center for Biotechnology Information (NCBI), www.ncbi.nlm.nih.gov/Genbank/genbankstats.html, 2004.

[11] M. Gogolla. A declarative query approach to object identification. In *Proceedings of the 14th International Conference on Object-Oriented and Entity Relationship Modelling*, 1995.

[12] G. Grahne, R. Hakli, M. Nykanen, H. Tamm, and E. Ukkonen. Design and Implementation of a String Database Query Language. *Information Systems*, 28(4):311–337, 2003.

[13] R. Hakli, M. Nykanen, H. Tamm, and E. Ukkonen. Implementing a Declarative String Query Language with String Restructuring. In *PADL*, pages 179–195, 1999.

[14] J. Hammer and M. Schneider. Genomics Algebra: A New, Integrating Data Model, Language, and Tool for Processing and Querying Genomic Information. In *CIDR*, 2003.

[15] S. Henikoff and J. Henikoff. Amino Acid Substitution Matrices from Protein Blocks. *Proceedings of the National Academy of Sciences*, 89(22):10915–9, November 1992.

[16] G. Jaeschke and H. Schek. Remarks on the Algebra of Non First Normal Form Relations. In *Symposium on Principles of Database Systems*, pages 124–138, 1982.

[17] C. Meek, J. M. Patel, and S. Kasetty. OASIS: An Online and Accurate Technique for Local-alignment Searches on Biological Sequences. In *VLDB*, pages 910–921, 2003.

[18] D. P. Miranker, W. Xu, and R. Mao. MoBIoS: A Metric-Space DBMS to Support Biological Discovery. In *SSDBM*, pages 241–244, 2003.

[19] J. M. Patel. The Role of Declarative Querying in Bioinformatics. *OMICS: A Journal of Integrative Biology*, 7(1):89–92, 2003.

[20] M. Regnier and W. Szpankowski. On Pattern Frequency Occurences in a Markovian Sequence. *Algorithmica*, 22(4):631–649, December 1998.

[21] M. A. Roth, H. F. Korth, and A. Silberschatz. Extended Algebra and Calculus for Nested Relational Databases. *ACM Transactions on Database Systems*, 13(4):389–417, 1988.

[22] P. Seshadri, M. Livny, and R. Ramakrishnan. Sequence Query Processing. In *SIGMOD Conference*, pages 430–441, 1994.

[23] P. Seshadri, M. Livny, and R. Ramakrishnan. The design and implementation of a sequence database system. In *VLDB*, pages 99–110, 1996.

[24] T. F. Smith and M. S. Waterman. Identification of Common Molecular Subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[25] M. Stonebraker and G. Kemnitz. The POSTGRES Next Generation Database Management System. *Communications of the ACM*, 34(10):78–92, October 1991.

[26] M. Stonebraker, D. Moore, and P. Brown. *Object Relational DBMS: Tracking the Next Great Wave*. Morgan Kauffman, 2nd edition, 1999.

[27] S. Tata and J. Patel. PiQA: An Algebra for Querying Protein Data Sets. In *SSDBM*, pages 141–150, 2003.

[28] S. J. Thomas and P. C. Fischer. Nested Relational Structures. *Advances in Computing Research*, 3:269–307, 1986.

[29] C. H. Wu and D. W. Nebert. Update on Human Genome Completion and Annotations: Protein Information Resource. *Human Genomics, 95760-21:35, 2004.*, 1(3):1–5, February 2004.