

Declarative Querying for Biological Sequence Databases

Sandeep Tata¹

Jignesh M. Patel^{1,*}

James S. Friedman²

Anand Swaroop^{2,3}

Departments of ¹Electrical Engineering and Computer Science,

²Ophthalmology and Visual Science, and ³Human Genetics

University of Michigan, Ann Arbor, MI 48109, USA

{tatas, jignesh, jfriedmn, swaroop}@umich.edu

Abstract

The ongoing revolution in life sciences research is producing vast amounts of genetic and proteomic sequence data. Scientists want to pose increasingly complex queries on this data, but current methods for querying biological sequences are primitive and largely procedural. This limits the ease with which complex queries can be posed, and often results in very inefficient query plans. There is a growing and urgent need for declarative and efficient methods for querying biological sequence data. In this paper, we introduce a system called Periscope/SQ which addresses this need. Queries in our system are based on a well-defined extension of relational algebra. We introduce new physical operators and support for novel indexes in the database. As part of the optimization framework, we describe a new technique for selectivity estimation of string pattern matching predicates that is more accurate than previous methods. We also describe a simple, yet highly effective algorithm to optimize sequence queries. Finally, using a real-world application in eye genetics, we show how Periscope/SQ can be used to achieve a speedup of two orders of magnitude over existing procedural methods!

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 datasets. Not surprisingly, there is a growing and urgent need for a system that can support complex declarative and efficient querying on biological datasets.

A large majority of biological data is sequence data pertaining to DNA and proteins. DNA datasets are sequences over the nucleotide alphabet of size 4: A, C, G, and T. Proteins can be represented as sequences over the amino acid alphabet, which is of size 20. Proteins also have a secondary structure which refers to the local geometric folding. This too is represented as a sequence over the secondary structure alphabet of size 3: alpha helix, beta sheet, and loops. These sequence datasets are often stored with additional information such as gene annotations in the case of DNA, and 3-D structure and known functions in the case of proteins.

Several existing databases store detailed protein and DNA sequence information. Some of these databases are large and are growing very fast. For instance, GenBank, a repository for genetic information has been doubling every 16 months [8] – a rate faster than Moore’s law! Protein databases, such as PDB [9] and PIR [18, 29] have also grown rapidly in the last few years. The growing sizes of the databases exacerbate the current deficiency in querying methods.

Biologists 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 similar to the protein being studied. Studying a similar protein 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 of the protein, or on the geometrical structure of the protein, or a

*To whom correspondence should be addressed. Tel +1 (734) 647-1806, Fax +1 (734) 763 8094.

combination of these. With genetic databases, scientists perform approximate similarity searches to identify regions of interest such as genes, regulatory markers, repeating units, etc. For any approximate matching query, the desired output is an ordered list of results.

We note that existing sequence search tools such as BLAST [2] only provide a limited search functionality. With BLAST one can only search for approximate *hits* to a single query sequence. One cannot look for more complex matches such as one query pattern separated from another query pattern by a certain distance, or a query sequence with some constraints on other non-sequence attributes. Consider the following query: “Find all genes in the human genome that are expressed in the liver and have a TTGGACAGGATCGA (allowing for 1 or 2 mismatches) followed by GC-CGCG within 40 symbols in a 4000 symbol stretch upstream of the gene”. This is an instance of a relatively straightforward, yet important query that can be quite cumbersome to express and evaluate with current methods. One could code a specific query plan for this query in a scripting language. For example, the query plan may first perform a BLAST [2] or Smith-Waterman [21] search to locate all instances of the two query patterns on the human genome. Then, the results of these matches can be combined to find all pairs that are within 40 symbols of each other. Next, a gene database can be consulted to check if this match is in the region upstream of any known gene. Finally, another database search would be required to check if the gene is expressed in the liver. Note that there are several other ways of evaluating this query, which may be more efficient. Moreover, current tools do not permit expressing such queries declaratively, and force the programmer to pick and encode a particular query plan. Researchers frequently ask such questions and current procedural methods are quite cumbersome to use and reuse.

In this paper, we describe a system called Periscope/SQ, which takes on the challenge of building a declarative and efficient query processing tool for biological sequences. The system makes it possible to declaratively pose queries such as the one described above. We also describe techniques to efficiently evaluate such queries, and using a real world example, demonstrate that Periscope/SQ is faster than current procedural techniques by over two orders of magnitude!

Periscope/SQ is part of a larger research project - called Periscope - which aims to build a declarative and efficient query processing engine for querying on *all* protein and genetic structures [17]. This includes the sequence, the secondary structure and the 3D structure. The SQ component stands for “Sequence Querying” and is the focus of this paper.

The main contributions of this paper are as follows:

- We present the design of the Periscope/SQ system that extends SQL to support complex sequence querying opera-

tions declaratively and efficiently.

- To optimize complex sequence queries, fast and accurate estimation methods are critical. We present a technique for estimating the selectivity of sequence pattern matching predicates based on a new structure called the Symmetric Markovian Summary. We show that this new summary structure is less expensive and more accurate than existing methods.
- We introduce novel query processing operators and also present an optimization framework that yields query plans that are faster than existing procedural querying methods by two orders of magnitude.
- We present a case study of an actual application in eye genetics, 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 discusses our extensions to SQL. Our query processing technique includes novel string/sequence predicate estimation methods, which are presented in Section 3, and query optimization and evaluation methods, which are presented in Section 4. Section 5 contains the results of our experimental evaluation, including an actual application in eye genetics. Section 6 describes related work, and Section 7 contains our conclusions and directions for future work.

2 Extending a Relational DBMS

Biologists often pose queries that involve complex sequence similarity conditions as well as regular relational operations (select, project, join, etc.). Consequently, rather than build a stand-alone tool only for complex querying on sequences, the best way to achieve this goal is to extend an existing object-relational DBMS [24] to include support for the complex sequence processing. For the Periscope project, we have chosen to extend the free open-source object-relational DBMS Postgres [28].

Our query language, which extends the SQL query language, is called PiQL (pronounced as “pickle”). PiQL incorporates the new data types and algebraic operations that are described in our query algebra PiQA [26]¹.

The purpose of this section is to describe *very briefly* the types and operators in PiQL. Readers who are interested in the algebraic properties of these extensions and details of the query language extensions may refer to [26, 27].

Match Type The match type is used to define attributes that store match information. A match is a set of triples (p, l, s) which refer to the position, length, and score for each part

¹PiQL stands for Protein Query Language – the full versions of both PiQA and PiQL can be used to query sequences and protein geometrical structures. Since DNA datasets do not typically include geometrical structures, querying on DNA only requires the subset of these methods that allows for querying on biological sequences.

Example PiQL Queries

1. CREATE TABLE prot-matches (pid INT, p STRING, match MATCH_TYPE)
2. SELECT * FROM MATCH(R,p,"EEK",EXACT,3)
3. SELECT AUGMENT(M1.match, M2.match, 0, 10) FROM MATCH(prot.s.p,"VLLSTTSA", MM(PAM30)) M1, MATCH(prot.s.p,"REVWAYLL", MM(PAM30)) M2
4. SELECT CONTAINS(AUGMENT(M1.match, M2.match, 0,10),M3.match) AS resmatch FROM MATCH(prot.s.p,"VLLSTTSA", MM(PAM60)) M1, MATCH(prot.s.p,"REVWAYLL", MM(PAM60)) M2, MATCH(prot.s.s,"LLLLL", EXACT) M3 WHERE score(resmatch) \geq 15

Figure 1. Example PiQL Statements

id	p	s
1	GQISDSIEEKRHH	HLLLLLLLLLHEE
2	EEKKGFEKRAVW	LLEEEHHHHHL
3	QDGGSEEKSTKEEK	HHHLLLEEEELL

Table 1. Relation R

of the match. Operations for match type are implemented as user defined functions on this new data type. For example, Query 1 in Figure 1 shows how to create attributes of this type.

Match Operator The Match operator finds approximate matches for a query string. It is implemented as a table function which takes as input a string, an attribute name, a match model (described later), and a cutoff score. The operator returns a relation with the match attribute. As an example of this operator, consider Query 2 shown in Figure 1 that finds all instances of the string "EEK" in attribute *p* of relation *R* (Table 1). The result of the PiQL query returns the relation *R* with an additional match column as shown in Table 2. The matching portions are shown in boldface in Table 1. These are referred to by position, length, and score in the match column of Table 2.

Since local similarity search is a crucial operation in querying biological sequences, one needs to pay close attention to the match-model. In practice, the commonly used match models include exact match model, *k*-mismatch model, and the substitution matrix based models. A *k*-mismatch model allows for at most *k* differences (mismatches) between the query and the match sequence. The general substitution matrix based models use a matrix that specifies the precise score to be awarded when one symbol in the query is matched with a different symbol in the database. For a more detailed discussion of various matching models, we refer the reader to [4].

While Periscope/SQ supports the three match models listed above, in this paper, we focus on the exact match

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. Match Results

model and the *k*-mismatch model, which are the most frequently used models for querying DNA and RNA sequences. The substitution matrix model is primarily used for protein sequences. An interested reader is referred to the extended version of this paper [27] for discussions on the current techniques used in Periscope/SQ for this model.

Match Augmentation Operator The augmentation operator computes the union of the matches of its operands if they are within a specified distance. This operator is implemented as a function that takes as arguments the two match fields, and the minimum and maximum distances. As an example, consider Query 3 in Figure 1, which will find all matches that are the form "VLLSTTSA" followed by "REVWAYLL" with a gap of 0-10 symbols between them. Each component is found using a match operator, and combined using the augmentation operator.

Other Operators Periscope supports many other operators like nest, unnest, contains, not-contains, etc. In the interest of space, we omit the details. See [26,27] for details.

Complex Query Example: As an example of a complex PiQL query consider the following query:

Find all proteins that match the string "VLLSTTSSA" followed by a match of 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.

The PiQL query for this example is shown as Query 4 in Figure 1. 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.

3 Selectivity Estimation

The introduction of sequence/string matching predicates poses an important problem while trying to optimize PiQL queries. Since an optimizer relies on fast and accurate selectivity estimation methods, poor estimation methods can lead to inefficient query plans (see Section 4). We address this issue by first presenting a new technique for exact match predicates that is more accurate than previous methods. Then, we describe extensions of this technique for the *k*-mismatch and

the substitution matrix models.

Our estimation method uses a novel structure called the Symmetric Markovian Summary (SMS). SMS produces more accurate estimates than the two currently known summary structures, namely: Markov tables [1], and pruned count suffix trees [12, 13]. A Markov table stores the frequencies of the most common q -grams. (A q -gram is simply a string of length q that occurs in the database.) Pruned suffix trees are derived from count suffix trees. A count suffix tree is a suffix tree [14] where each node contains a count of the number of leaves in the subtree rooted at that node. This is equal to the number of occurrences of the string corresponding to that node. To find the number of occurrences of the pattern “computer” using a count suffix tree, we simply traverse the edges of the tree until we locate the node that is at the end of a path labeled “computer”, and return the count value. The *pruned* count suffix tree uses a pruning rule to store only a small portion (say, the first few levels,) of the entire count suffix tree [12]. Observe that a pruned count suffix tree in effect stores the frequencies of the *most commonly occurring patterns* in the database.

Notice that in these previously proposed strategies, the summary structures are biased towards recording the patterns that occur frequently. The estimation algorithms then typically assume a default frequency for patterns that are not found in the summary. For instance, this could be the threshold frequency used in pruning a count suffix tree. If a query is composed mostly of frequently occurring patterns, then this bias towards recording the frequent patterns is not an issue. However, if the query tends to have a higher selectivity (i.e., matches very few tuples), such a summary can bias the estimation algorithm towards greatly overestimating the result size. As the experimental evaluation in Section 3.4 shows, these existing algorithms perform very poorly when it comes to negative queries (where 0 tuples are selected) and queries with high selectivity.

The key strength of SMS is that it captures *both* the frequent and rare patterns. Our estimation algorithm that uses SMS not only produces more accurate estimates for the highly selective predicates (the “weak spot” of previous methods), but also produces better estimates for predicates with lower selectivities. In the following section, we now describe our estimation algorithm, and the SMS structure.

3.1 Preliminaries

In a traditional database context, the selectivity of a string predicate is the number of rows in which the query string occurs. Alternately, we can define it as the number of occurrences of the query string in the database. Multiple occurrences in each row make these two metrics different. This alternate definition is more useful in bioinformatics where we are interested in finding all occurrences of a query string

even if they are in the same row (eg. same chromosome). This is the definition of selectivity we use in the rest of the paper. Our technique can also be adapted to return the number of rows, and thereby be used for text queries.

Most string datasets (English text or DNA or protein sequences) can be modeled quite accurately as a sequence emitted by a Markov source. That is, we assume that the source generates the text by emitting each symbol with a probability that depends on the previous symbols emitted. If this dependence is limited to k previous symbols, then we call this a Markovian source with memory k , or simply a k^{th} order Markov source. In [12], the authors show that for most real world data sets, this k is a fairly small number. We refer to this property as the “short-memory” property, to mean that most real world sequences do not have significant long range correlations.

3.2 The Estimation Algorithm

Now, suppose that we have a query $q = a_1a_2a_3\dots a_n$. The number of occurrences of the string q in the database is the probability of finding an occurrence of q times the size of the database. Equivalently, this is *(the probability that the Markov source emits q) \times (the size of the database)*. If $P(q)$ denotes the probability of the source emitting q , then:

$$\begin{aligned} P(q) &= P(a_1) \times P(a_2/a_1) \times P(a_3/a_1a_2) \times \\ &\quad \dots \times P(a_n/a_1\dots a_{n-1}) \\ &= P(a_1) \times \prod_{i=2}^n P(a_i/a_1\dots a_{i-1}) \end{aligned}$$

We can exploit the short-memory assumption and use the fact that $P(a/b_1\dots b_n)$ is the same as $P(a/b_{n-k+1}\dots b_n)$, where k is the memory of the Markovian source. The expression can now be rewritten as $P(q) = P(a_1) \times \prod_{i=2}^n P(a_i/a_{i-k}\dots a_{i-1})$. If we had a table where we could look up values for $P(a_i/a_{i-k}\dots a_{i-1})$, this probability can be computed easily. The Symmetric Markovian Summary (SMS) provides these values.

The crux of the estimation algorithm is in making the best use of these values, and using reasonable approximations when these values are not found in the summary.

Algorithm **StrEst**, as shown in Figure 2, computes the estimates using the equation described above. While retrieving a probability from the summary, it first looks for $P(a/Y)$. If this value is not found, it searches the summary for $P(a/Z)$, where $Y = bZ$ for some symbol b . It successively searches for shorter suffixes of Y , and if nothing else is found, it returns $P(a)$. This algorithm may make as many as $k|q|$ probes of the summary. The basic intuition behind this approach is that we *expect* $P(a/bZ)$ can be approximated by $P(a/Z)$.

Estimation Function StrEst(q , $summary$)

1. $p = 1.0$
2. For $i = 1$ to $|q|$
 3. $s = q_1 \dots q_{i-1}$
 4. If $Prob(q_i/s)$ is stored in the summary, $v = Prob(q_i/s)$
 5. Else, $v = Prob(q_i/s')$,
where s' is the longest suffix of s such that $Prob(q_i/s')$ is in the summary
 6. $p = p \times v$
7. End For
8. Return $p \times DBsize$

Figure 2. Estimation Function StrEst

Other Match Models For the k -mismatch model, we use a simple estimation technique. For small values of k , we list all possible strings that have at most k mismatches with the query string. We compute their selectivities using the exact match model, and add them up. For larger values of k , we use a different approach. We compute a *representative selectivity* s_r for the set of strings (W) that have at most k differences with the query string. The number of such strings is: $|W| = \sum_{i=1}^k C(L, i) \times (A - 1)^i$. L is the length of the string and A is the alphabet size. (For an i -mismatch string, you choose i symbols from the L and replace them with one of $A - 1$ symbols for a mismatch.) We then compute the selectivity as $s_r \times |W|$. An obvious choice for s_r is the exact match selectivity of the query string. A better choice is the average selectivity of the set of strings with l mismatches, where l is a small number like 1 or 2. Such an average will effectively sample a larger subset of W and produce a better estimate (as also supported by the experimental results presented in Section 3.4.4).

For predicates using the general substitution matrix model, a simple estimation method is to use a heuristic that computes the selectivity of an *equivalent* k -mismatch predicate by choosing an appropriate k . Another alternative is to examine the properties of the substitution matrix to expand the query string into a set of closely homologous strings and to use existing estimation methods for each string. A detailed exploration and evaluation of methods for this model is part of future work. See [27] for preliminary results.

3.3 The Symmetric Markovian Summary

The Symmetric Markovian Summary (SMS) is essentially a lookup table that stores various probabilities of the form $P(a/Y)$, where a is a symbol in A (the alphabet,) and Y is a string of length at most k . If we let D_k denote the set of all probabilities where Y is exactly of length k , then $|D_k| = |A|^{k+1}$. In the simplest case when $k = 0$, this reduces to storing the unconditional probability for each symbol in the alphabet. Ideally, one would like to have the summary $S = \cup_{i=0}^k D_k$ for some sufficiently large k .

Algorithm H1($String, k, B$)

- OCC = [], STR = [], PROB = [] $A = \text{Alphabet} \cup \{null\}$
1. Calculate the frequency of each q -gram s for q varying from 1 to k as OCC(s).
//Now calculate conditional probability
 2. For every a, Y such that $|Y| < k$
 3. $PROB(a/Y) = OCC(Ya)/OCC(Y)$
 4. End For
 5. Create Priority Queue PQ of Size B bytes
 6. Fix unconditional probabilities into PQ.
 7. For each entry in Prob
 8. $priority = |A|^{-|Y|+1} \times |Prob(a/Y) - Prob(a/X)|$
where X is the longest suffix of Y present in PQ.
 9. $PQ.insert(<a/Y, Prob(a/Y), priority>)$
 10. If Size of PQ exceeds B , drop lowest priority element and adjust the priorities of affected elements.
 11. End For
 12. PQ contains the Symmetric Markovian Summary

Figure 3. Algorithm H1 to construct SMS

The size of such a table grows exponentially with the value of k , making it impractical especially for large alphabets. Therefore, we need to choose a smaller subset of S such that these probabilities provide an accurate estimate. The basic idea behind SMS is to choose only the *most important* probabilities from S . A probability value is less important if we would incur only a small error if we didn't store it and approximated it with a different probability instead (when using algorithm StrEst).

We present two algorithms H1 and H2 that use different notions of the importance of a probability to construct an SMS. These two methods differ in the manner in which they compute the importance of an entry. Before describing these algorithms in detail, we first present the intuition behind defining a good notion of importance.

There are two components to the importance of a probability. A straightforward indicator of importance is the error that might be incurred if the value were *not* in the summary. We call this the δ -value of the probability entry. Suppose that we exclude $P(a/Y)$ from the SMS, and use some $P(a/X)$ (where X is the maximal suffix of Y ,) from the summary to approximate it. We compute $\delta = |P(a/Y) - P(a/X)|$. Note that $P(a/Y)$ being more likely than $P(a/X)$ is just as important as it being less likely. It is this symmetric property that leads to a better summary.

An orthogonal but important factor that determines the importance of a probability entry is the likelihood that it will actually be used in some queries. This is basically a workload dependent factor. For instance, even if the probability $P(A/CACAC)$ has a higher δ value than $P(A/AC)$, it might still make better sense to choose $P(A/AC)$ to retain in the summary, simply because it is likely to be used more often than the former. For the workload as a whole,

the average error incurred from approximating $P(A/AC)$ will add up to more than the error from approximating $P(A/CACAC)$ since $P(A/AC)$ is likely to be used more often. The likelihood that a given probability entry will be used for a given workload is the γ -value of the entry. In the absence of any characterization of the queries, one can assume a uniform query distribution and assign a higher γ to shorter strings. We combine these two components to define importance as the product of δ and γ .

Formally speaking, for a given k , and a fixed summary size (B entries), we want to store a subset of values from each of D_0, D_1, \dots, D_k such that the values we prune away can be approximated well. Mathematically, we want to choose $T \subset \bigcup_{i=0}^k D_i$ such that $imp = \sum_{p \in T} (\gamma \times |p - Approx_T(p)|)$ is maximized. Here $Approx_T(p)$ is the value that will be used to approximate p in T , if p is excluded from T . We want a subset such that the total importance of each of the elements is the maximum over any subset of this size. In other words, we pick the subset that has the most important B elements. This is clearly a hard optimization problem. Constructing an optimal summary with a naive approach will take an unacceptably long time. We therefore present two heuristic approaches H1 and H2 that perform very well for a wide range of datasets.

Algorithm H1 This algorithm first computes D_0, D_1, \dots, D_k using a q-gram frequency table. Note that values from D_0 are the unconditional probabilities of occurrence of each of the symbols. We'll always need these for the first symbol of the query string. The algorithm first selects D_0 into the summary structure (maintained as a priority queue). For each of the entries in $D_i (i > 0)$, the algorithm computes $\delta = |P(a/Y) - P(a/X)|$. To find X , the maximal suffix of Y , it scans the priority queue. It then computes $\gamma = |A|^{-|Y|+1}$, and importance = $\delta \times \gamma$ and inserts the entry into the priority queue. If the queue size exceeds the maximum size of the summary, we remove the element with the lowest importance. We then scan the queue and adjust the δ value for those elements that were directly dependent on the entry we just deleted. This heuristic runs in time $O(nB \log(B))$, where B is the summary size, and n is the total number of probability entries being considered.

Algorithm H2 Though H1 is a good heuristic, an important drawback is that it is computationally expensive. H2 uses a simpler algorithm that runs faster than H1, but may yield a slightly less accurate summary. Instead of scanning the priority queue to find the X that is the maximal suffix, H2 simply uses the unconditional probability instead of the actual $Approx_T(p)$ entry. Everything else remains the same. Note that now, we do not have to adjust any values when we delete an element from the priority queue. The main advantage of this algorithm is that it is very simple, and fast. The running time for H2 is $O(n \log(B))$. Experimental evaluations show

that H2 is not much worse than H1, but is significantly faster to compute.

Both H1 and H2 store the summary as a list of pairs (" a/Y ", $P(a/Y)$) sorted on the first part. A lookup can be performed in $O(\log(B))$ time using binary search.

3.4 Evaluation of the Accuracy of SMS

In this section, we first compare the SMS-based algorithms H1 and H2. We also compare the SMS method with the method of [12], which is currently considered to be the best method for estimating the selectivities of exact match predicates. (Note that the recent work by Surajit *et al.* [3] uses an estimation method that is built upon existing summary structures such as the pruned suffix tree. Their technique uses a learning model to exploit the properties of English text, and is not applicable to biological data. We note that our contribution is orthogonal to [3] as their system can be built on top of SMS.)

3.4.1 Experimental Setup

Data sets: We tested our estimation methods on number of different biological datasets: a nucleotide (DNA) dataset [6] (Chromosome 1 of Mouse, 200 million symbols) and a protein dataset (the SwissProt [18] collection, 53 million symbols). We refer to these datasets as MGEN and SPROT respectively. We also tried a number of English text sources to verify the general applicability of our method. The results were similar to the biological datasets, and are presented in [27].

Query Sets: For MGEN, we generated 150 random strings ranging from lengths 6 to 12 so it would span all the selectivities. Similarly, for SPROT, we generated a set of 150 random strings of lengths ranging from 3 to 7.

Result Organization: For each algorithm, we classify the queries based on their actual selectivities. Queries that have less than 1% selectivity are classified as high selectivity queries. The ones between 1%-10% were classified medium selectivity, and those that had more than 10% selectivity were classified as low selectivity queries. The metric of accuracy we use is the average absolute relative error calculated as a percentage: $e = 100 \times \frac{|prediction - actual|}{actual}$. We refer to it simply as the *average error*.

Note that since highly selective queries produce only a few results, the error in estimating this class can potentially present a skewed picture. For instance, if the actual number of occurrences was just 1, and we predicted 2, that's a 100% error! A well established convention to not bias the result presentation for such cases, is to use a correction [3, 12]. While calculating the error, if the *actual* selectivity is less than $100/|R|$, we divide the absolute error in selectivity by

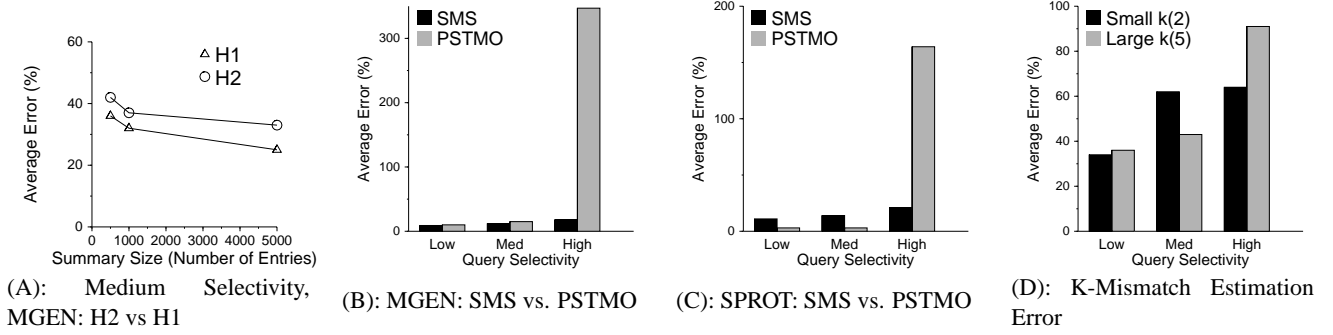


Figure 4. Evaluation of the Accuracy of SMS

$100/|R|$ instead of the actual value. $|R|$ is the number of tuples in the relation.

Platform: All experiments in this paper were carried out on an 2.8 GHz Intel Pentium 4 machine with 2GB of main memory, and running Linux, kernel version 2.4.20.

3.4.2 Comparison of H1 and H2

In our first study, we examine the effect of using an SMS of type H1 versus one of type H2.

We ran the query sets using H1 and H2 on each of the datasets for varying summary sizes. We present the results for medium selectivity queries with MGEN in Figure 4(A). The results for low and high selectivity queries and other datasets are similar (see [27]) and are omitted here. From this figure, we see that as the summary size increases, both H1 and H2 have increased accuracy. However, H1 has a consistent advantage over H2. At larger summary sizes the error from H2 is within 10% of H1. Note that the cost of using H1 is significantly higher than the cost of H2. For instance, with the MGEN dataset and an SMS with 1000 entries, the time taken to construct H1 is 219 seconds, while H2 takes only 93 seconds. However, H2 incurs only a small loss in accuracy. Therefore, we conclude that *except* for cases where very high accuracy is needed, or if the summary size is very small, we use H1 to construct the summary. In all other cases, we use H2 as it is cheaper to construct, and nearly as accurate as H1.

3.4.3 Comparison with Existing Methods

In this section, we compare our SMS based algorithm with the algorithm proposed in [12]. For this experiment, we used algorithm H2 to construct the summaries. The algorithm in [12] uses a maximum overlap parsing along with a Markovian model for the text. The summary structure they use is a pruned count suffix tree. For ease in presentation, we refer to the method in [12] as the PSTMO algorithm.

For this experiment, we fixed the summary size to be 5% of the database size (results with 1% and 10% summary

sizes are similar - see [27]). We present the average absolute relative error for each class of query for each dataset in Figures 4(B) and 4(C).

For the MGEN dataset, SMS has a slight advantage over PSTMO for low and medium selectivity queries. However, for high selectivity queries, PSTMO has a very large error - over 340%, compared to only 18% with SMS! In the case of SPROT, we see that PSTMO has a slight advantage for low and medium selectivity queries. This is mostly due to the fact that the query set has many short strings. PSTMO stores the exact counts of these short strings and therefore ends up being very accurate for these queries. However, for longer strings (high selectivity), the error for PSTMO rises sharply to 164%. In contrast, SMS has a low error of 21%. As discussed in Section 3.3, SMS is more accurate because it is a symmetric digest of the information in the text.

An important class of queries that were not explored in the above study is the class of negative queries. While searching text databases, users commonly make spelling or typographical errors which result in the string predicate selecting zero records. Algorithms like PSTMO tend to provide very poor estimates for these queries. However, our SMS based algorithm works very well for these queries too. We have also experimented with negative queries, and the results are similar to the highly selective queries [27].

Execution times: We compared the execution time of the SMS based estimation algorithm with that of PSTMO. For DNA queries, the average time taken for our method is 3 microseconds while PSTMO takes 66 microseconds. For protein queries, we take 7 microseconds while PSTMO takes 18 microseconds. This advantage is due to the fact that PSTMO needs to repeatedly traverse a suffix tree. Traversing suffix tree nodes is expensive because it involves chasing a number of pointers. While the estimation time is an insignificant part of the query evaluation time, these numbers show that even though the SMS based algorithm is more accurate than PSTMO, it does not take longer to execute!

3.4.4 K-Mismatch Estimation

We examined the efficacy of our approach for estimating predicates using the k -mismatch model for different values of k . We present the results of the study for the case of a small k (2) and a large k (5) in Figure 4(D). Observe that the error in estimation in this case is generally higher than the exact model. This is because we use the estimates from the exact model to compute these estimates, and the cumulative error tends to be significantly larger. In spite of the relatively larger error, the estimates are reasonably accurate for queries of all selectivities.

In summary, we have presented an algorithm for estimating the selectivity of string/sequence predicates using a novel structure called the Symmetric Markovian Summary (SMS). Our estimation method using SMS is more accurate than existing algorithms, and also takes less time for computing the estimate. Existing methods are particularly poor in estimating the selectivity of highly selective predicates, which is gracefully handled by our approach. As our empirical evaluation shows, in some cases our approach is up to 5 times more accurate than the previous best algorithm.

4 Query Evaluation

The introduction of new operators in PiQL presents two significant challenges. First, we need efficient algorithms to execute new operators like match, augment, contains, etc. Second, we need to extend the optimizer to be able to optimize over the new operators. We first discuss algorithms for the crucial match operator. We then briefly describe algorithms for other operators and present a new physical operator called the Match-and-Augment. Finally, we present an optimization algorithm that is highly effective at finding good plans for a subset of queries.

4.1 Algorithms for Match

The algorithms for evaluating the match operator varies depending on the match model. In the simplest case - the exact match - a linear scan of the database can be used. The *Scan* algorithm scans the sequence from start to finish and compares each sequence with the query pattern for an exact match. With a 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 cost of this algorithm is $O(n \times q_{eq})$ where n is the length of the database, and q_{eq} is the expected number of states of the automaton that are traversed before deciding on a hit or a miss. For the more complex model using a substitution matrix, the linear scan or the FSA scan algorithm cannot be used directly. For this complex match model, we can use the Smith-Waterman [21] (SW) algorithm, which is a dynamic

programming local-alignment algorithm. Its time complexity is $O(m \times n)$ where m is the size of the query and n is the size of the database. The BLAST [2] family of algorithms is a heuristic approach to local-similarity searching that runs faster than SW, and finds *most* matches for a given query.

The OASIS [15] algorithm is a suffix tree based technique for sequence similarity that can be used with any match model (including the substitution-based matrix model with affine gap penalties). In the case of the exact match, one can simply traverse down the suffix tree along the query string and collect all the leaf nodes under that node (this is essentially a simple suffix tree query). The cost of this algorithm is $O(q + r)$ where q is the length of the query and r is the number of matches. The cost of a k -mismatch search with a suffix tree is typically similar to an OASIS search.

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

Algorithms for other operators like augment, contains, not-contains are similar to a traditional join. Instead of a simple equality, the join condition tends to be a complex predicate involving match types [27].

4.2 A New Combined Operator

We have designed a new physical operator that combines matching with the match augmentation operator. We call this the Match-and-Augment (MA) operator. It can be used to extend a set of matches with another set of 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).
```

A simple way to compute this expression is to evaluate each match independently, and then use a join to compute the augment. Alternately, we can evaluate the first MATCH, then scan 50 symbols to the right of each match that is found, and check for the occurrences of "CA". In this process, we select and augment only those matches where we find the "CA". This is essentially the approach used in the MA operator. The MA approach can often be cheaper than performing two matches separately and combining the results with the augment operation.

4.3 Optimization

Our current optimization strategy uses a two stage optimization method. In the first step, we optimize the portion of the query that refers to the complex sequence predicates,

Algorithm Optimize

1. Compute selectivity $s(i)$ of each predicate
2. Compute cost $c(i)$ of evaluating each predicate
3. Let f be the most selective predicate
4. Let g be an adjacent predicate
5. t = cost of evaluating g , then combining it with f .
6. u = cost of using a match-and-augment operator
7. If $t > u$, then rewrite the plan as match-and-augment
8. If there is another adjacent predicate that has not been considered, pick it to be g . Go to step 5.
9. End

Figure 5. The Optimization Algorithm

and in the second stage we call the Postgres optimizer to optimize the traditional relational components of the query. We acknowledge that this two step process may miss opportunities for optimization across the two components. Our eventual aim is to integrate these two steps, but we start with this two step optimization as it is more amenable for rapid prototyping. In this section, we describe our methods for optimizing the complex sequence predicates.

The basic idea behind the optimization algorithm is as follows: Suppose that the query contains n match predicates connected together by operators like augments. We compute the selectivity of each match predicate, and pick the most selective predicate to start with. We examine the predicate *adjacent* to this and compute the cost of evaluating that match and combining it with the current predicate. Now, we compare this with the cost of using a match and augment operator. If it is cheaper, then we rewrite the plan to use a match and augment operation and examine another adjacent predicate in the same way. The algorithm terminates when an adjacent predicate cannot be combined using a match and augment or when all the predicates have been combined. The algorithm is outlined in Figure 5.

It is clear that the algorithm runs in time proportional to the number of match predicates. Although it explores a very small portion of the plan space, it is highly effective at finding good plans. We demonstrate this in Section 5 using extensive experimental evaluation. The optimizer uses SMS for predicate selectivity estimation. The cost model is fairly straightforward (CPU cost + I/O cost), and follows the complexity of the algorithms with empirically determined constants plugged in (see [27] for details).

5 Experimental Validation

In this section, we present the results of various experimental studies that we conducted to examine the performance of our system. Using several synthetically generated query loads, we explore a wide range of query situations. In addition, we also present results that are based on a real-life workload that was captured while a scientist was perform-

ing explorative querying using our tools. We used the full mouse genome [6] (2.6 billion symbols) as the dataset for the experiments in this section. We also performed experiments on several other genetic datasets and protein datasets, which show similar trends [27].

5.1 Impact of SMS-based Estimation

In order to understand the benefits of increased accuracy from the new SMS based estimation algorithm, we performed the following experiment. We randomly generated a hundred queries having three match predicates each. One of the predicates used a k -mismatch model, while the others used an exact match. The query load was executed for $k = 0, 1$, and 2 . (We use these relatively small values since k is usually a small number in practice. Our methods also work for larger values of k [27].)

The lengths of each of these predicates was randomly chosen to be between 6 and 14. Neither the suffix tree index, nor the match and augment operator is used in evaluating these queries. Each query was optimized by *exhaustively* searching over the plan space. (Note that in this experiment we are *not* using the linear optimization algorithm of Section 4.3, but rather, a simple exhaustive enumeration of *all* the query plans. This exhaustive optimization is guaranteed to pick the plan with the best estimated cost, thereby isolating any effects related to the optimization algorithm.)

We optimized the queries in two ways: In one case we used PSTMO [12] to estimate the selectivities while optimizing the query, and in another case, we used the SMS based estimation algorithm. We used a one percent summary in both cases. We found that the average running time of the query plan (which does not include the optimization time) was higher by about 43% when using PSTMO. Of the 100 queries, 90 queries were optimized identically by both algorithms, and 10 queries were optimized differently. These 10 query plans took roughly 4.6 times as long to execute when optimized using PSTMO as opposed to using SMS. The reason for this behavior is because PSTMO had overestimated the selectivity of some of the predicates by a margin large enough that it led to a different execution plan in each of these ten queries. This clearly drives home the advantage of using SMS based estimation over existing methods.

5.2 Impact of Using Match and Augment

In this experiment, we explore the effectiveness of using the new match and augment operator (MA), which was described in Section 4.2. For this experiment, we ran the set of 100 queries generated as above in two different ways. One plan was optimized with the match and augment operator and the other plan without it. For this experiment also, we used an exhaustive search optimization algorithm. The

k	Without MA	With MA
	Average (Std-Dev)	Average (Std-Dev)
0	3.04 (11.5)	0.19 (0.08)
1	46.71 (142.08)	0.55 (0.65)
2	226.76 (808.5)	13.55 (41.46)

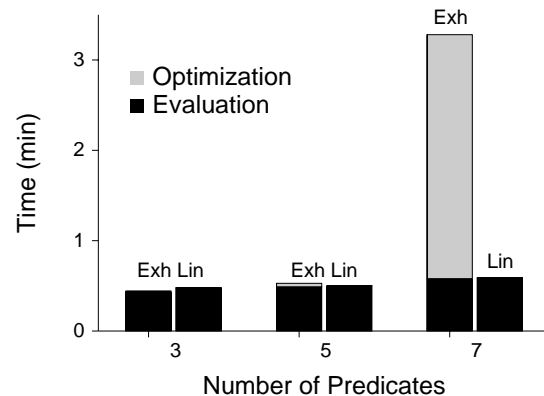
Table 3. Query Plan Evaluation Times (mins)

query plan evaluation times are summarized in Table 3 for each value of k . As is evident, the use of the new operator can lead to significant savings. The plan that used the match-and-augment operator executed 10 to 80 times faster on average!

In Table 3, we also provide the standard deviation of the times for the 100 queries. To get a better understanding of how often and how much the match and augment operator helps, we split the queries into three sets: the first set, where the new operator provides at most a 2X speedup (small advantage), the second bin where the speedup was greater than 2 but less than 10 (significant advantage), and the third bin where the speedup exceeded a factor of 10 (large advantage). We observed that for $k = 0$, in 65% of the queries were in the first category, around 20% in the second, and 15% in the third. Similarly for the case where $k = 1$, the split-up was 35%, 30%, and 35% respectively. Finally for $k = 2$, the query set split was 30%, 20%, 50% into the three categories. It is clear from the evidence that the new operator can be very useful in a significant number of queries.

5.3 Optimizer Evaluation

In this experiment, we compare two optimization algorithms. The first one is a conventional algorithm that *exhaustively* searches the plan space. The second algorithm is the linear time optimization algorithm described in Figure 5. For this experiment, a suffix tree index is available on the data, which increases the number of algorithms that the optimizer can choose from. We generated three sets of hundred queries each with 3, 5, and 7 predicates. One of the predicates in each query was randomly selected to use a k -mismatch model with k randomly chosen as one of 0, 1, 2. The average query optimization time and the evaluation time in each case is shown in Figure 6. The plan obtained using the linear time optimization algorithm always runs within 6% of the optimal plan's running time. For the exhaustive query optimization method, the time take to optimize the query is low for a small number of predicates (3 or 5), but is unacceptably large when more predicates (7 and above) are used. Performing an exhaustive search to find the optimal plan is a better option only in the case of 3 predicates. Overall, this experiment shows that the linear query optimization method is quite robust. The exhaustive optimization method can produce slightly better plans, but should only be used when the query has a small number of predicates.

**Figure 6. Optimization and Evaluation Times**

5.4 GeneLocator: An Application

The current implementation of Periscope/SQ has been used in an web-based application called GeneLocator that we have built in collaboration with researchers at the Kellogg Eye Institute at the University of Michigan. GeneLocator is a tool for finding target *promoter regions*. In order to understand certain genetic factors associated with eye diseases, scientists are trying to identify all genes that are regulated by a particular transcription factor (a regulatory protein, also called a promoter). Such proteins typically bind to a “signature” binding site: a short sequence of DNA about 10-15 bases long. The pattern usually allows for a few mismatches. The presence of a TATA-box (a pattern like “TAATA”) or a GC-box (a pattern like “GCGC”) within a certain distance downstream of the match to the signature often indicates that it is a potential binding site. Also, transcription almost always begins at a “CA” site, which is a short distance following the TATA-box or the GC-box. Figure 7 pictorially represents the kind of pattern the biologists are looking for. In PiQL, this query can be expressed as:

```
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 the above query, GeneAnnotations is a table with the following schema: *GeneAnnotations* (*id*, *chromosome*, *start*, *end*, *type*, *annotation*), and is loaded with the gene annotation data from NCBI [6]. The extra conditions in the WHERE clause in the above query filter out the matches to report only those that are a short distance upstream of a known gene.

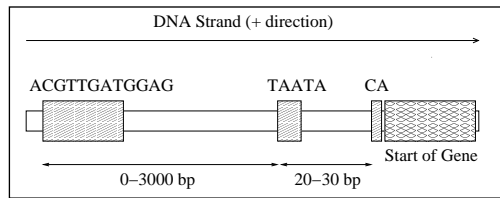


Figure 7. Promoter Binding Region

GeneLocator is accessed by a web interface, which allows the end user to pose queries by filling out a simple form. Our collaborators are working with the mouse genome, and use this tool for posing interactive queries. With their permission we logged the queries that they issued. Most of their queries had three match predicates. The inter-predicate distance and the number of mismatches allowed in the match model varied across the queries. One or two of the predicates often used an exact match model. The others used a k -mismatch model. The actual queries are not presented in order to protect the privacy of the research. For this application we built a suffix tree on the mouse genome using our suffix tree construction method [25].

5.5 Performance of GeneLocator

We compared the execution times of the set of queries logged using three different query plans. The first query plan does not use any indexes, and uses no optimization - a naive left to right evaluation of the augments is used to compute the result. The second plan uses a suffix tree and an exhaustive search to choose the cheapest plan. It does not use the match-and-augment operator. The third plan is optimized using the linear optimization method and includes the match-and-augment operator. The dataset used was entire the mouse genome (2.6 billion symbols). The execution times are as are shown in Table 4.

The first observation we can make from Table 4 is that using the suffix tree can dramatically improve the query execution time. This does not come as a surprise, since suffix tree index based algorithms are usually very efficient. Second, we observe that the plan with the match and augment operator executes faster than the version without it by nearly an order of magnitude. The current procedural methods that are used in life sciences research labs tend to resemble the first plan (no indexes, no optimization, simple operators) and therefore take an extremely long time to run. *The contribution of Periscope is not only that it provides a declarative and easy way to pose complex queries, but also that it executes them upto 450 times faster than existing procedural approaches!*

Results Using GeneLocator, the eye genetics researchers were able to identify several potential targets for the transcription factor of interest, which are now being verified using wet-lab experiments. These targets were computa-

Algorithm	Time (min)
Unoptimized Plan (No Index)	473.05
Optimized, No MA (With Index)	9.76
Optimized, With MA	1.02

Table 4. Execution Times

tionally identified using our system after just a few days of explorative querying. This process could easily have taken several weeks or months to accomplish using conventional methods. Encouraged by these results, we are now planning more ambitious projects in comparative genomics.

6 Related Work

Miranker *et al.* [16] use constructs similar to PiQA [26] to describe complex sequence queries. They largely focus on devising and exploiting metric space indexing structures for faster sequence queries. Our work, on the other hand, focuses on providing a declarative and efficient system in the context of many existing similarity measures.

Hammer and Schneider [11] outline an approach to express complex biological phenomenon (like transcription, translation, crossovers, and mutations) through algebraic operations. To our knowledge, no systems have been built using this algebra. In [7, 10], the authors propose an alignment calculus on strings and describe a system to query string databases. The language lets a user express very complex queries, by permitting 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.

Previous work on querying traditional sequences [19, 20] is largely tailored towards handling time series data and cannot be directly used for querying biological sequences.

Recognizing the need for supporting sequence query matching in a relational framework, commercial DBMS vendors have recently started supporting BLAST calls from SQL statements [5, 22, 23]. However, these methods only provided limited sequence searching capabilities, allowing only simple pattern search (for example match-augmentation is not supported), and can only work with the BLAST match model.

Krishnan, Vitter, and Iyer presented one of the earliest approaches for estimating the selectivity of exact wildcard string predicates in [13]. The more recent work by Jagadish *et al.* [12] improves on [13] by using a short-memory Markovian assumption instead of an independence assumption. These methods employ pruned suffix trees as the summary of the text in the database. Suffix trees are versatile data structures, however, they have the drawback of being biased towards storing more frequent patterns. The SMS based approach we propose does not have this bias and is

more accurate than existing techniques.

Chaudhuri, Ganti, and Gravano [3] recently proposed a technique which takes advantage of the frequency distribution properties of the English text to increase the accuracy of estimation techniques. The method is based on the fact that English text often has a short identifying substring. This method has not been shown to be applicable to other datasets such as DNA and protein sequences. The estimation methods that we propose in this paper can easily fit into the overall framework of [3] for use in text databases.

7 Conclusions and Future Work

In this paper, we have presented Periscope/SQ - a system that permits declarative and efficient querying on biological sequences. The declarative querying interface provided by this system extends traditional SQL to provide integrated querying of relational and biological sequence data. In producing this system, we have made significant contributions in various aspects of query processing for biological sequences. Periscope/SQ employs a novel method for estimating the selectivity of sequence predicates that is more accurate than previous methods. In addition, the system employs novel physical operators and query optimization strategies for processing complex biological sequence queries. Periscope/SQ is being used in an actual eye genetics application to evaluate queries in less than one hundredth the time it used to take before, demonstrating the huge impact that this approach can have for scientists querying biological sequences.

As part of future work, we are investigating methods for extending the declarative query processing framework to cover other biological data types, including protein structures and biological networks.

References

- [1] A. Aboulmaga, A. R. Alameldeen, and J. F. Naughton. Estimating the Selectivity of XML Path Expressions for Internet Scale Applications. In *VLDB*, pages 591–600, 2001.
- [2] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic Local Alignment Search Tool. *Journal of Molecular Biology*, 215:403–410, 1990.
- [3] S. Chaudhuri, V. Ganti, and L. Gravano. Selectivity Estimation for String Predicates: Overcoming the Underestimation Problem. In *ICDE*, pages 227–238, 2004.
- [4] R. Durbin, S. R. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge Univ. Press, 1st edition, 1999.
- [5] B. A. Eckman and A. Kaufmann. Querying BLAST within a Data Federation. *IEEE Data Engineering Bulletin*, 27(3):12–19, 2004.
- [6] GenBank, NCBI, 2004. www.ncbi.nlm.nih.gov/GenBank.
- [7] 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.
- [8] Growth of GenBank, National Center for Biotechnology Information (NCBI). www.ncbi.nlm.nih.gov/Genbank/genbankstats.html, 2004.
- [9] H. M. Berman et al. The Protein Data Bank. *Acta Crystallographica*, D58:899–907, 2002.
- [10] R. Hakli, M. Nykanen, H. Tamm, and E. Ukkonen. Implementing a Declarative String Query Language with String Restructuring. In *PADL*, pages 179–195, 1999.
- [11] J. Hammer and M. Schneider. Genomics Algebra: A New, Integrating Data Model, Language, and Tool for Processing and Querying Genomic Information. In *CIDR*, 2003.
- [12] H. V. Jagadish, O. Kapitskaia, R. Ng, and D. Srivastava. One-dimensional and Multi-dimensional Substring Selectivity Estimation. *The VLDB Journal*, 9(3):214–230, 2000.
- [13] P. Krishnan, J. S. Vitter, and B. Iyer. Estimating Alphanumeric Selectivity in the Presence of Wildcards. In *SIGMOD*, pages 282–293, 1996.
- [14] E. M. McCreight. A Space-economical Suffix Tree Construction Algorithm. *Journal of the ACM*, 23(2):262–272, 1976.
- [15] 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.
- [16] D. P. Miranker, W. Xu, and R. Mao. MoBioS: A Metric-Space DBMS to Support Biological Discovery. In *SSDBM*, pages 241–244, 2003.
- [17] J. M. Patel. The Role of Declarative Querying in Bioinformatics. *OMICS: A Journal of Integrative Biology*, 7(1):89–92, 2003.
- [18] R. Apweiler et al. UniProt: the Universal Protein Knowledgebase. *Nucleic Acids Research*, 32(D):115–119, 2004.
- [19] P. Seshadri, M. Livny, and R. Ramakrishnan. Sequence Query Processing. In *SIGMOD*, pages 430–441, 1994.
- [20] P. Seshadri, M. Livny, and R. Ramakrishnan. The Design and Implementation of a Sequence Database System. In *VLDB*, pages 99–110, 1996.
- [21] T. F. Smith and M. S. Waterman. Identification of Common Molecular Subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.
- [22] S. M. Stephens, J. Y. Chen, M. G. Davidson, S. Thomas, and B. M. Trute. Oracle Database 10g: a platform for BLAST search and Regular Expression pattern matching in life sciences. *Nucleic Acids Research*, 33:D675–D679, 2005.
- [23] S. M. Stephens, J. Y. Chen, and S. Thomas. ODM BLAST: Sequence Homology Search in the RDBMS. *IEEE Data Engineering Bulletin*, 27(3):20–23, 2004.
- [24] M. Stonebraker, D. Moore, and P. Brown. *Object Relational DBMS: Tracking the Next Great Wave*. Morgan Kaufman, 2nd edition, 1999.
- [25] S. Tata, R. A. Hankins, and J. M. Patel. Practical Suffix Tree Construction. In *VLDB*, pages 36–47, 2004.
- [26] S. Tata and J. M. Patel. PiQA: An Algebra for Querying Protein Data Sets. In *SSDBM*, pages 141–150, 2003.
- [27] S. Tata and J. M. Patel. Towards Declarative Querying of Biological Sequences. Technical Report CSE-TR-508-05, University of Michigan, 2005.
- [28] The PostgreSQL Database System. www.postgresql.org.
- [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, 2004.