mSQL: SQL Extensions and Database Mechanisms for Managing Biosequences

Willard S. Willard Wenguo Liu Shulin Ni
Rui Mao Weijia Xu Daniel P. Miranker

Computer Sciences Department
University of Texas at Austin
{willard, liu, shulin, rmao, xwj, miranker}@cs.utexas.edu

Abstract

mSQL is an extended SQL query language targeting the expanding area of biological sequence databases and sequence analysis methods. The core aspects include first-class data types for biological sequences, operators based on an extended-relational algebra, an ability to define logical views of sequences as overlapping q-grams and the materialization of those views as metric-space indices.

We first describe the current trends in biological analysis that necessitate a more intuitive, flexible, and optimizable approach than current methodologies. We present our solution, mSQL, and describe its formal definition with respect to both physical and logical operators, detailing the cost model of each operator. We describe the necessity of indexing sequences offline to adequately manage this type of data given space and time concerns. We assess a number of metric-space indexing methods and conclude that MVP-trees can be expected to perform the best for sequence data. We ultimately implement two queries in mSQL to show that, not only can biologically valid analyses be expressed in concise mSQL queries, such queries can be optimized in the same ways as those relying on a standard relational algebra.

1. Introduction

As biological data proliferates, the creation of new computational biological protocols is also proliferating. The focus of attention within sequence analysis is moving beyond the identification of proteins and their homology. New foci include identification of regulatory elements such as transcription binding sites and RNA coding genes [2,22,23,26,32]. Some investigational techniques, such as phylogenetic footprinting, often require the comparison of the entire contents of two or more genomes. While its domain of applicability is often stretched, BLAST targets protein sequence homology and its execution comprises a linear scan of the database. In many cases BLAST is either not equipped to discern the features of interest, or it is computationally infeasible to use BLAST. While BLAST can be applied to comparing two genomes, the resulting algorithm is $O(n^2)$, where n is the length of the genomes.

In reviewing the biological literature we made two important observations. First, much of the computation in these new analysis methods could easily be captured as database queries, but conventional relational query engines are incapable of querying sequences in biologically meaningful ways [28]. For example, a recent *Science* paper started with the question, "What highly similar subsequences co-occur in the genomes of humans, mouse and rat?" [2]. Yet it is clear the researchers labored to develop the computation and the final scientific interpretation required integrating the results with annotation data stored in the same database where the sequences were kept.

Second, the storage and identification of biological sequences comprises long functional units (e.g. genes, proteins and chromosomes), but the analysis and retrieval of those sequences is almost always concerned with finding ordered sets of short matching subsequences. (The Smith-Waterman local-alignment algorithm is the notable

exception.) Indeed, *Search for short, nearly matching subsequences* is now a top-level option at NCBI's BLAST homepage [27].

We have addressed both of these issues in the development of mSQL, a database query language that is part of a larger experimental DBMS targeting biological data. Two logical operators, *createfragments()* and its complement, *merge()*, are used to help resolve the dichotomy between the storage and identification of sequences as long units and the analysis as the comparison of short subsequences. Biologists are often concerned with *local-alignments*, small areas between pairs of sequences that are evolutionarily close, as defined by a weighting model. Local-alignment algorithms often start by performing exact matches on small subsequences (hot-spots). For this reason, mSQL reflects an algorithmic structure where sequences are decomposed into overlapping substrings of length q (q-grams) [12]. *Createfragments()* decomposes a sequence into q-grams, and *merge()* rejoins them. The algebra is much simpler than many previous efforts on string query systems, and a strong foundation in extended-relational algebra simplifies implementation and optimization [11].

We also introduce the concept of a *sequenceview*, which allows the database system to create a persistent materialization of the q-gram representation. In the current implementation of mSQL, *sequenceviews* are materialized as secondary indexes based on metric-space indexing.

A metric-space is defined as a set of points and a corresponding metric distance function, or *metric*. A metric, by definition, obeys the triangle inequality. That property can be leveraged to form tree-based index structures without embedding the data in a Euclidean space [7,17]. Both biological and multimedia applications may benefit from this approach [9,10,16,24].

The taxonomy of metric-space indexing algorithms contains three primary classes: radius-based, generalized hyperplane, and vantage point. To our knowledge there are no refereed papers where the performance of the different classes are compared on a biological workload. We assess the performance of one algorithm from each of the three major classes. We conclude that for sequence data, MVP-trees perform the best. We subsequently consider detailed parameterization needed to optimize the index structure. We then present empirical results demonstrating the scalability and biological accuracy of the system.

Finally, we present two biologically valid, working mSQL queries, one of which is an mSQL version of the genome comparison analysis mentioned above. In previous work we have published 6 other mSQL queries that encode use cases similar to the query above, all drawn primarily from *Science* and *Nature* [26]. While we have previously described the syntax of the language, this is the first formal presentation of its semantics and algebra. We use these two examples to detail how a query optimizer could optimize the query using the rules of the algebra.

2. mSQL

We have coined the term *mSQL* to describe our keyword and operator additions to the SQL92 specification. We will use the explanation of a simple query to find homologous regions in two genomes as a running example to detail the merits of the system. The schema for a simple genome database is defined with the following mSQL declaration:

```
create table genomes(
Organism varchar,
Acc_num integer,
DNA_Sequence dna,
constraint PK_genomes primary key (Acc_num)).
```

In addition to the standard SQL data types, mSQL includes built-in data types for DNA and Protein sequences, as well as Spectra (to support mass spectroscopy analysis). In the example, we assume for simplicity that DNA_Sequence is a string, as its complete definition entails biological semantics beyond the scope of this paper (i.e., equivalence of reverse complements). We will be looking for homologous regions of length 3 between the rat and mouse genomes that differ by at most 1 nucleotide. For brevity, the rat 'genome' is illustrated as two chromosomes and is 11 nucleotides long. The mouse 'genome' is just 6 nucleotides long. The example database is populated as shown in Table 1.

Table 1. Example genomes database

Organism	Acc_num	DNA_Sequence
rat	1	AGAAC
rat	2	CCGAT
mouse	3	AACAAT

From the logical perspective, our key contributions are two new operators: *createfragments()* and its complement, *merge()*. *Createfragments()* decomposes sequences into overlapping substrings, also known as q-grams. *Merge()* maps sets of overlapping q-grams back into larger sequences. We include a *groupfragments()* operator, which allows the user to ensure that *merge()* is applied only to groups of q-grams with corresponding offset differences. Finally, we have extended the *distance()* operator of spatial databases to allow for a choice of metric.

Algebraically *createfragments()* and *merge()* are defined similarly to the unnest and nest operators of an extended-relational algebra [19]. If the substrings did not overlap, the behavior of *createfragments()* would be identical to unnest applied to a representation of sequences as a set of substrings. *Merge()* has similarly been refined from nest to include biological semantics constraining the nesting of overlapping q-grams.

mSQL also introduces a new type of view, *sequenceview*. *A sequenceview* is a physical database construct analogous to SQL's *view*. Explicit in a

sequenceview is the materialization of *createfragments()* as a secondary metric-space index.

2.1 Createfragments($\kappa_{sequence:length}$)

Createfragments() takes two arguments: a table name and sequence attribute (in dot notation) followed by the length of the substring. The second parameter serves to define the width of the sliding window on the sequences when creating fragments. Since in our example we will be looking for homologous regions of length three, we set this parameter to be three. The following query demonstrates the syntax of the *createfragments()* operation:

```
select * from createfragments(genomes.DNA_Sequence, 3)
where Organism = 'rat'.
```

The results of the *createfragments()* operation are shown in Table 2.

Table 2. Results of createfragments() operation

Createfragments(genomes.DNA_Sequence, 3)

Organism	Acc_num	DNA_Sequence
rat	1	{0, AGA}
rat	1	{1, GAA}
rat	1	{2, AAC}
rat	2	{0, CCG}
rat	2	{1, CGA}
rat	2	{2, GAT}

The *createfragments()* operation is formally defined as follows: Assume a tuple $X = \{x_1, x_2, \ldots x_i, \ldots x_n\}$, where x_i represents a sequence of length m. Given x_i and q, the length of the fragment to create, the createfragments operation will create a relation with m-q+1 tuples, $X_1, X_2, \ldots X_j, \ldots X_{m-q+1}$. Each entity x of each tuple X_j is identical except for x_i, which consists of an offset equal to j and a subsequence of the original sequence from j to j + m. This operation is repeated for each tuple in the relation.

2.2 Sequenceview

```
1.create sequenceview rat_sview as
2.   select * from createfragments(genomes.DNA_Sequence, 3)
3.   where Organism = 'rat'
4.using base_pair_mismatch;
```

Figure 1. Creation of a *sequenceview*

By representing sequences as tables of q-grams we can use join operators to compare the contents of the sequences to each other. This by itself is not new. Gravano et al. explicitly materialized q-grams in an auxiliary table [12] such as the one shown in Table 2. However, the lengths of the actual mouse and rat genomes are ~2.6×10^9 bp and ~2.75×10^9 bp respectively. With a q-gram size of 200, (the size used in our application example in section 7), a whole genome comparison would require generating ~5.35×10^9 rows of 200 characters each, or ~1.07×10^{12} extra characters. Furthermore, even if it is feasible to materialize and compare q-grams at query time under some computationally simple models of q-gram similarity, biological sequences are more often compared using expensive similarity models of evolution, i.e., weighted edit distance.

Moreover, the data in biological sequence databases is write-once and monotonically increasing in size. Entries are rarely updated, and if they are updated the culture demands that it be accomplished through versioning. Under this workload it is safe to exclude the cost of maintaining a view in the face of updates and deletes to the base relations [3]. Once a *sequenceview* is materialized as an index, it may be accessed multiple times—reducing the impact of the cost of building the index and providing a fast access path for range and nearest neighbor queries. Figure 1 illustrates the creation of a *sequenceview* for the rat genome in the example.

A *sequenceview* is comprised of two major elements:

1. Sequences that are to be included in a *sequenceview* are derived in the standard method involving a SQL query, including the explicit decomposition of the sequences into q-grams using *createfragments() (lines 2-3)*.
2. A metric index is specified as the access-path for the q-grams *(line 4)*.

Part of the generality of metric-space indexes is that the development of the index mechanism is independent of the metric distance function (metric). Thus, in addition to the usual arguments to create a secondary index, the creation of a metric-space index is parameterized by the choice of metric.

Once *createfragments()* splits the sequences into a set of fragments of a certain fragment length, a metric-space index tree can be built over these fragment, which can then be used to accelerate matching of fragments. The details of materializing *sequenceviews* are described in section 4.

2.3 Merge($\lambda_{predicate}$) and Groupfragments(γ)

Given a set of q-grams, it is ultimately necessary to assemble them back into longer sequences. The set of q-grams may have been the argument to relational operators, and not all of the q-grams of the original sequence may be present in the computed result. Informally, if two q-grams overlap or merely adjoin with respect to the original sequence, we wish to merge them into a longer sequence.

The merge operation takes as arguments the attributes of a tuple to be merged. In terms of a relational algebra, two tuples of a relation, $X = \{x_1, x_2, \ldots x_n\}$ and $Y = \{y_1, y_2, \ldots y_n\}$, can be merged into one tuple $Z = \{z_1, z_2, \ldots z_n\}$ iff, for each attribute pair x_i, y_i:

- $x_i = y_i$ (yielding $z_i = x_i = y_i$); or
- if x_i and y_i represent sequence fragments (with offsets x_{ik}, y_{ik}, and lengths x_{im}, y_{im}),
  - $x_{ik} \leq y_{ik} \leq x_{ik} + x_{im}$, and
  - x_i from y_{ik} to $(x_{ik} + x_{im}) = y_i$ from y_{ik} to $(x_{ik} + x_{im})$.

Depending upon the biological significance of the query, it may be desirable to only reassemble fragments that have a specific offset distance between two or more different sequences. A *groupfragments()* operator is included so that *merge()* can be applied to sets of q-grams that obey this property. A fragment group is formally defined as follows:

For two sets of fragments, S1 = $\{s_1, s_2, \ldots s_n\}$, T = $\{t_1, t_2, \ldots t_n\}$, where each fragment has a sequence offset denoted x_{ik}, s_i, t_i and s_j, t_j belong in the same fragment group iff

- $t_{ik} - s_{ik} = t_{jk} - s_{jk}$.

The GROUP BY operator is overloaded to represent the *groupfragments()* operator, although the *groupfragments()* operator is quite different from the traditional SQL GROUP BY operator. While the traditional GROUP BY operator is based on equality, i.e., only equal objects are put into the same group, the *groupfragments()* operator groups fragments based on the equality of their offset differences. Also, unlike the standard GROUP BY, *groupfragments()* only has meaning when applied to more than one attribute.

2.4 Distance(δ(metric, a, b))

Below we illustrate the final SQL program for identifying q-grams from our two genomes that differ by at most one nucleotide:

```
select R.Organism, R.Acc_num, M.Organism, M.Acc_num,
    merge(R.DNA_Sequence, M.DNA_Sequence)
from rat_sview as R, mouse_sview as M
where distance('base_pair_mismatch',
    R.DNA_Sequence, M.DNA_Sequence) <= 1.0
group by R.Organism, M.Organism, R.Acc_num,
    M.Acc_num, R.DNA_Sequence, M.DNA_Sequence;
```

A metric-space join is similar in concept to a spatial join. The goal is to determine pairs of objects that fulfill a certain distance predicate. Analogous to spatial extensions of SQL, mSQL allows distance predicates for selects and joins. The syntax has been expanded to allow for the specification of a metric. In addition to built-in metrics, users may extend mSQL with their own definitions of new metrics.

Table 3. Results of the metric join

Org.	Acc_ num	DNA_ Sequence	Org.	Acc_ num	DNA_ Sequence
rat	1	{0, AGA}	mouse	3	{1, ACA}
rat	1	{1, GAA}	mouse	3	{2, CAA}
rat	1	{2, AAC}	mouse	3	{3, AAT}
rat	1	{2, AAC}	mouse	3	{0, AAC}
rat	2	{2, GAT}	mouse	3	{3, AAT}

Table 3 illustrates the results of the metric join after *groupfragments()* has been performed but before the *merge()*. The first and second row of the join result have been placed into the same fragment group by the *groupfragments()* operator. Since they also meet all of the requirements of the merge operation, the first two fragments can be merged into one fragment for sequences 1 and 2, respectively, yielding the final results of the query as shown in Table 4.

Table 4. Results of the mSQL query

merge(R.DNA_Sequence, M.DNA_Sequence)

Org.	Acc_ num	DNA_ Sequence	Org.	Acc_ num	DNA_ Sequence
rat	1	{0, AGAAC}	mouse	3	{1, ACAAT}
rat	1	{2, AAC}	mouse	3	{0, AAC}
rat	2	{2, GAT}	mouse	3	{3, AAT}

3. Properties of the algebra

A well-defined algebra provides certain guarantees that that a query optimizer can depend upon when forming cost-effective query plans. In this section we list some of the properties of the algebra that can be used by the query optimizer, as well as the cost of each additional operator.

It is easy to see that *merge()* is the inverse of the *createfragments()* operation. Thus, $\lambda\kappa(R) = R$.

A metric join is defined as a theta-join with a distance function as one of the operators and a radius as the other, where both of the inputs to the distance function are attributes of relations. Thus a metric join obeys all of the properties of a theta-join. Also, since the distance function is defined to be commutative, where δ(metric, a, b) = δ(metric, b, a), (since metrics are symmetric), a metric join is commutative over the attributes of the distance function:

$R \,|\times|_{\delta(\text{metric, a, b}) \le r} S = R \,|\times|_{\delta(\text{metric, b, a}) \le r} S$.

Depending on the size of the relations *R* and *S* and the method used to perform the join, the cost associated with the join can be optimized. For instance, if the join is implemented as an indexed nested loop, the query optimizer can place the relation with fewer tuples on the outside of the loop. Metric join is also associative over the attributes of the distance function:

$(R \,|\times|_{\delta(\text{metric, a, b}) \le r1} S) \,|\times|_{\delta(\text{metric, b, c}) \le r2} T =$

$R \,|\times|_{\delta(\text{metric, a, b}) \le r1} (S \,|\times|_{\delta(\text{metric, b, c}) \le r2} T)$.

This rule can be used to select the order of the join operations over several distance functions. Furthermore, metric join is also distributive over the attributes of the distance function:

$(R \,|\times|_{\delta(\text{metric, a, b}) \le r1} S) \,|\times|\, (R \,|\times|_{\delta(\text{metric, a, c}) \le r1} T) =$

$(R \,|\times|_{\delta(\text{metric, a, b}) \le r1} S) \,|\times|_{\delta(\text{metric, a, c}) \le r1} T$.

A metric match is defined as a select operation with a distance predicate in which one of the attributes is a constant. Like select, it is commutative:

$\sigma_{\delta(\text{metric, V, a}) \le r1} (\sigma_{\delta(\text{metric, W, b}) \le r2} (S)) =$

$\sigma_{\delta(\text{metric, W, b}) \le r2} (\sigma_{\delta(\text{metric, V, a}) \le r1} (S))$.

The metric match operator also distributes over the metric join operator:

if attribute a occurs in *R* but not in *S*:

$\sigma_{\delta(\text{metric, V, a}) \le r1} (R \,|\times|_{\delta(\text{metric, b, c}) \le r2} S) =$

$(\sigma_{\delta(\text{metric, V, a}) \le r1} R) \,|\times|_{\delta(\text{metric, b, c}) \le r2} S$; or

if attribute a occurs in *S* but not in *R*:

$\sigma_{\delta(\text{metric, V, a}) \le r1}(R \,|\times|_{\delta(\text{metric, b, c}) \le r2} S) =$
$R \,|\times|_{\delta(\text{metric, b, c}) \le r2} (\sigma_{\delta(\text{metric, V, a}) \le r1} S)$; or

if attribute a occurs in both *R* and *S*:

$\sigma_{\delta(\text{metric, V, a}) \le r1}(R \,|\times|_{\delta(\text{metric, b, c}) \le r2} S) =$
$(\sigma_{\delta(\text{metric, V, a}) \le r1} R) \,|\times|_{\delta(\text{metric, b, c}) \le r2} (\sigma_{\delta(\text{metric, V, a}) \le r1} S)$.

By pushing down metric match operations, the query optimizer can reduce the size of temporary tables, thereby reducing the size of the input parameters to the expensive metric join operation.

The cost model for the new operators is listed in Table 5, where m and n represent the total number of input fragments for two different sequences and p represents the total length of an input sequence. As one can see, the general query plan will involve pushing the metric join as far as possible to the top, as it is the most expensive operation performed at query time.

Since the metric join is the most expensive operation, it is vital that mSQL uses an optimal index structure for biological information. In the next sections, we describe the materialization of *sequenceviews* as metric-space indices and compare the performance of various metric-space index trees.

Table 5. Cost model for algebraic operators

Operator	Cost	Description
Createfragments(κ)	O(p)	A linear scan
Merge(λ)	O(m)	A linear scan
Groupfragments(γ)	O(mlogm)	Sort followed by a linear scan
Metric match	O(logm)	Using a metric-space index tree
Metric join	O(mlogn)	Using nested indexed loops

4. Materializing sequenceviews

4.1 Managing biological sequences in a metric-space

The use of metric-space indexing distinguishes our approach from others where only exact or near exact matching fragments can be searched [21]. Given a query q-gram, our approach can identify a neighborhood of evolutionarily close q-grams through a range search. The *neighborhood* of a q-gram is determined by a metric distance function and a radius parameter of a given size.

The first step in developing a metric-space index for a given type of data is to define a metric distance function for that datatype. For nucleotides, we use weighted Hamming distance as the metric distance function. Comparing peptides is more difficult, because the protein space has a larger alphabet size and contains more complex relations than sequences of nucleotides. The primary challenge of indexing peptides in a metric space is to properly define the distance between amino acids. Sellers first proposed this problem in 1974 [29]. In our previous work we derived a biologically effective amino acid substitution matrix, mPAM, which satisfies the metric distance properties [39]. The metric distance function defined by mPAM enables effective indexing of protein sequence in a metric space [40].

4.2 General approaches of metric-space indexing

Having implemented a metric distance function for both nucleotides and peptides, the next step is to determine which type of metric-space index will provide the best performance for these data types. There are three categories of tree-based index algorithms for metric-space index trees: radius-based (RB) trees, generalized hyperplane (GH) trees, and vantage point (VP) trees. Our first concern is a basic assessment of the performance of each class on our workload. This assessment is important as analytic results have concluded that a linear search of high-dimensional data will outperform indexing methods [36]. Even though metric-space indexing enables the retrieval of data without any explicit interpretation of the data in Euclidean space, the data may intrinsically be of high dimension.

Excellent survey articles on metric-space indexing already exist [7, 17], so we present only the most basic description of the various methods' predicate structure. Radius-based trees were inspired by R-trees [15]. In a radius-based method, a hierarchy of bounding spheres replaces the rectangles used in R-trees. Bounding spheres are defined by a pair c, r, where c is the center of the sphere and the radius r determines the volume of the bounding sphere. Interior nodes of the tree contain a set of predicates p, with $d(c, p) < r$, such that all points p stored in the sub-tree are contained in the sphere. Ciaccia et al.'s M-trees are radius-based. Ciaccia et al.'s effort stands out as the first investigation of an external metric-space index structure with all of the page organization and dynamic properties expected of a database index [8]. The results we present are for the RB-tree, our improvement on the M-tree [24]. A challenge of M-trees, like R-trees, is that bounding predicates may overlap, reducing the opportunity to prune branches during search. RB-trees optimize the internal node structure and search mechanism of M-trees and contain an improved bulk-loading scheme.

In a binary GH-tree, two data points are selected as *pivots*. The surface of the hyperplane is determined by all the points equidistant between the pivots. A data point is stored in the sub-tree with the closest pivot. More precisely, given pivots c_1 and c_2, a point p is stored in the sub-tree corresponding to c_1 if $d(c_1, p) < d(c_2, p)$. Otherwise it is stored in the sub-tree corresponding to c_2. Brin describes GNATs (geometric near-neighbor access trees), which generalize the binary structure of a GH-tree to consider many pivots and larger fanout of interior nodes [5].

Predicates in a multi-vantage point (MVP) tree consider multiple pivots as well as multiple partitioning

radii. In a single vantage point tree, a pivot c and a radius r are selected, just as in a radius-based method. However, in vantage point methods the radius forms a partitioning, not a bounding sphere. If we seek a binary partition, r is selected such that the number of points that fall outside the partition sphere is equal to the number of points inside the sphere. Therefore, r is actually the median of the distances for all data points to the pivot. Similarly, multiple medians can be selected so that the distance range from each vantage point may be broken into intervals. The Cartesian product of the intervals forms a set of data partitions. The resulting structure is similar to a kd-tree. The construction is applied recursively to form an index tree [4].

For GH-trees and MVP-trees, partitions form crisp boundaries. The results presented below show that the overlap in the bounding predicates of a radius-based method put it at a disadvantage. An advantage of the general hyperplane method over radius-based methods is that a GH-tree has no radius and its clusters do not overlap.

4.3 Optimal index structure for biological sequences

Each of the three metric space index structures carries with it advantages and disadvantages. To select the one best fitted for indexing biological data, we implemented all three algorithms and tested on a biological benchmark.

The benchmark suite is curated and furnished by NCBI. The dataset contains 6433 yeast protein sequences (about 2,892,155 residues). The query set contains 103 sequences whose true positive hits have been identified by human experts and whose curation is continually refined [28]. The benchmark suite was downloaded in August 2002 (ftp.ncbi.nlm.nih.gov/pub/impala/blastest).

Each index structure was tested by running range queries on the yeast protein dataset, which was indexed for global alignment of 5-grams using the mPAM matrix.

Our implementations of the GH-tree and the VP-tree are slightly different from the original structures. Originally, both GH-trees and VP-trees reside in main memory. In our implementation both algorithms are parameterized such that their index nodes are sized to 4 kilobyte disk pages. Thus, a measure of the number of I/O operations is representative of the number of nodes visited. In our implementation of the MVP-tree, the number of vantage points and the split number of each vantage point are maximized to make index nodes occupy a full disk page. We use the farthest-first-traversal (FFT) algorithm to select vantage points [18]. FFT is a k-center algorithm, guaranteed to generate a clustering in which the maximum cluster radius is within a factor of 2 of optimal. In our GH-tree implementation we also compute the radius of the bounding sphere containing all the points in the partition. The retrieval algorithm integrates additional pruning due to the bounding sphere predicate into the GH-tree search procedure.

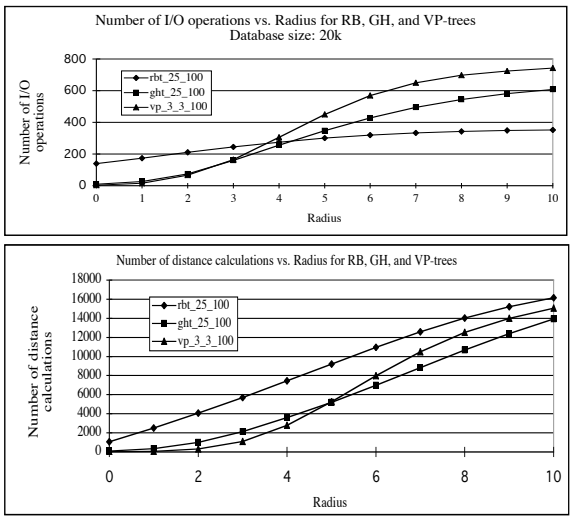

Figure 2. Comparison of metric-space index structures: RB-tree, GH-tree, and VP-tree

The experiment results are presented in Figure 2. Since in most applications of metric-space indexing the distance calculations are expensive, the goal of the index structure is to minimize both the amount of I/O and the number of distance calculations. Figure 2 shows the relationship between the range search radius and the number of distance calculations and the amount of I/O for each index tree. From this figure we can see that the RB-tree has the largest number of distance calculation and the MVP-tree has the largest number of I/O operations for large radii. However, the MVP-tree yields the best performance for small range query radii. In our applications, the most biologically effective results are obtained at small search radii. Thus we select the MVP-tree as the best index structure for protein sequences.

4.4 MVP-tree parameter selection

Having chosen an MVP-tree for our index structure, it is necessary to choose the proper parameters for that tree. There are three parameters associated with our MVP-tree implementation: the number of vantage points in each node, the number of children of each vantage point and the maximum number of data points in each node. There is always a trade-off between the number of leaves visited and the number of distance calculations needed. The more data points a node has, the less leaf nodes need to be visited. However, a tree with bigger nodes requires more distance calculations on each search. Figures 3 and 4 show the experimental results for various parameter combinations. Based on these results, we decided to use two vantage points per node, two children per vantage point and a maximum number of 100 data points per leaf node.

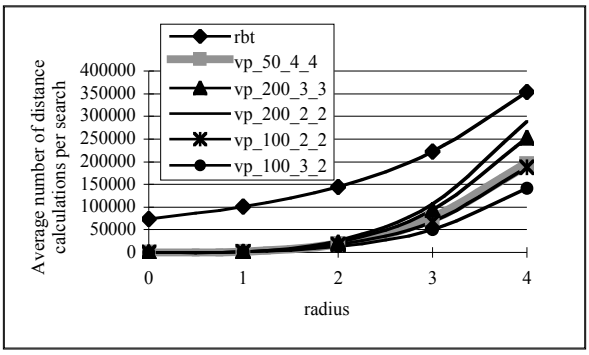

Figure 3. Average number of distance calculations per query for various tree structures.

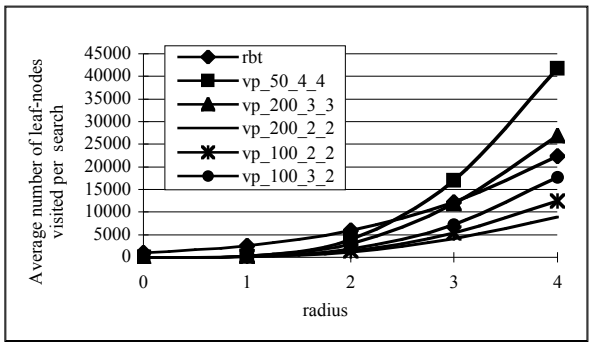

Figure 4. Average number of leaf nodes visited per query for various tree structures

4.5 Paged MVP-tree for materializing sequenceviews

We decided to use q-grams of length 5 for peptide indexing [40] based on our results. Since the alphabet size of a peptide sequence is 20, the complete space of q-grams will quickly be covered as the number of q-grams increases. Furthermore, in a real dataset, some q-grams have a significantly higher repetition rate than others because of the non-uniform distribution of amino acids.

To solve this problem, we bucket the fragments in index leaves. For each leaf index node, identical fragments are put into one bucket. Only one distance to each vantage point is stored for all of the fragments in each bucket. When a query is executed, if a bucket cannot be pruned, the query must only compute distance to one fragment in the bucket to determine whether or not all of the fragments in the bucket are valid results.

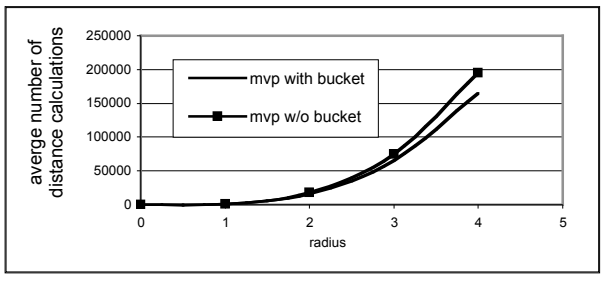

Figure 5. Average number of distance calculations using a bucketing structure with MVP-trees

Figure 5 shows that the bucketing structure can decrease the average number of distance calculations even for the benchmark dataset, which barely covers the space of q-grams with length 5. Such a data structure will be more effective and assure scalable search performance for a larger dataset.

5. Empirical results

In this section, we present our experimental results from an analysis conducted on a protein sequence domain with the benchmark described in section 4.3. The results show that the q-gram approach has scalable, sensitive search performance on biological sequences.

5.1 Quality of search result

The same benchmark described in section 4.3 was used to analyze the performance of PSI-Blast [30]. The quality of the search result is measured using receiver-operating characteristic (ROC) scores, a popular measure used in biology [13]. A similar method was used in [30] as well. For each query, the ROC_{50} value is computed by comparing the result list with the list of true positive hits. The ROC_{50} value has been computed as follows:

$$ROC_n = \frac{1}{nT} \sum_{i=1}^{n} t_i \qquad (1)$$

where t_i is the number of true positive hits ranked ahead of the ith false positive, and T is the total number of true positives.

Table 6. Comparison of average of ROC_{50} values for various searches

Search Method		Matrix	Average ROC_{50}
Sequential Search with Smith-Waterman local alignment algorithm		MPAM	0.48
		PAM250	0.59
		PAM70	0.5
Indexed Search	Radius 3	MPAM	0.45
	Radius 4		0.53
	AutoRadiusSearch		0.5
BLASTP		PAM250	0.53
		PAM70	0.42

In our indexed search, every query sequence is first broken into overlapping 5-grams. Each of these fragments is then queried with a certain radius against a pre-built database of the yeast protein sequence which has also been broken into overlapping 5-grams. The returned 5-grams are processed using a heuristic chaining algorithm to generate the homology search results. Table 6 compares the average ROC_{50} score for each query using different search radii as well as other search algorithms with the same benchmark. The top three rows are results computed using the Smith-Waterman local alignment

algorithm with different PAM matrices. The bottom two rows are the results of using Blastp program with PAM250 and PAM70 matrices. The middle three rows are indexed search results using our approach with different radii. Each query sequence normally requires many q-gram searches. In the normal range search based strategy, the same radius is used for all of the q-gram searches. For example, in radius 4 search, all the q-grams in the database that are within a distance of 4 to the query q-gram are returned. Although radius 4 search shows better accuracy than the radius 3 search (since more q-grams are being returned per query q-gram) the search process is also slower than the radius 3 search. To improve accuracy without losing too much performance, we developed another search strategy named AutoRadiusSearch. Am AutoRadiusSearch automatically adjusts the search radii based on the predicted number of matching q-grams for each q-gram in the query sequence.

The results show that, using metric space indexing, protein sequence homology search can yield accuracy comparable to Blastp on the same benchmark. Note that the accuracy of the results is actually affected by many factors, such as the value of the substitution matrices, the search strategy, the chaining strategy, etc. [14].

5.2 Scalability of the search

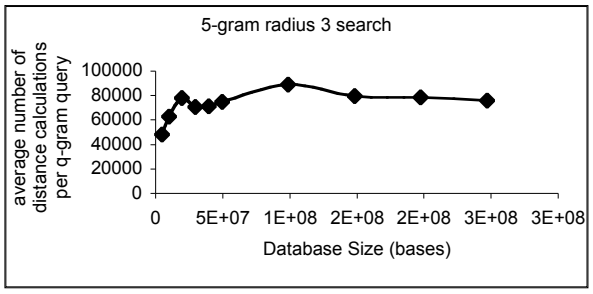

Figure 6a. Average number of distance calculations per q-gram search for different size datasets

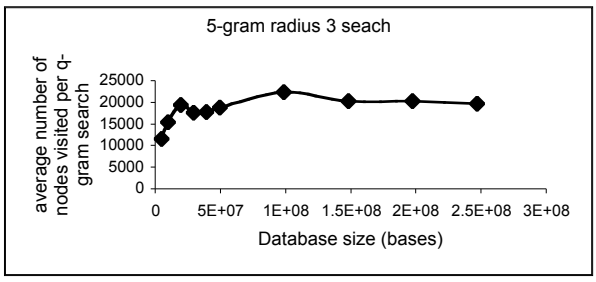

Figure 6b. Average number of leaf nodes visited per q-gram search for different size datasets

The entire benchmark database used for the accuracy test contains about 3 million amino acids. In order to get a more meaningful assessment of scalability, we acquired a larger data set that contains FASTA formatted amino acid translations extracted from GenBank/EMBL/DDBJ records that are annotated with one or more CDS features. The file was downloaded from Genbank in July 2003 (ftp://ftp.ncbi.nih.gov/genbank/genpept.fsa.z). A set of databases was built with different subsets of the data that were taken sequentially from the full dataset. The same set of queries from the yeast benchmark was used for all of the databases.

We used the same query set as used in the accuracy benchmark against various sizes of databases with the AutoRadiusSearch search strategy. The average number of distance calculations and the average number of leaf nodes visited are plotted in Figure 6. Both figures reveal scalability with the size of the database.

We have reason to believe that as the database grows the logical locality of the clusters starts to correspond better to the physical clustering on pages [24]. The effect is that entire contents of sub-trees could be found and returned in their entirety without further distance calculations, thus reducing the number of distance calculations. Similarly, entire sub-trees can be pruned, reducing the search cost.

5.3 The wall clock time measurement

mSQL is implemented in Java as an extension of the McKoi open-source relational database management system, which is also written in Java. Those who avoid Java for database management system research due to performance concerns are well justified. We have determined that 80% or more of our execution time is spent in object serialization and type casting routines in the disk I/O manager. These operations have no overhead in C++ due to the looser type system and more controllable memory allocation. At the outset of this work we were primarily concerned with issues revolving around validity of metric models of biological similarity and broad algorithmic assessments of relative behavior and scalability. Thus, we determined that we would be better served by the productivity offered by Java, even if this meant an additional expense later of porting the system to Postgres or a commercial RDBMS. We are certain that we made the correct decision.

Even so, some assessment of the absolute speed of this system is called for. While mSQL addresses far more general computation than is possible using BLAST, BLAST searches to determine protein sequence homology are the canonical comparison. For consistency within the database literature we used the same workload as was used for a similar measure in a recent VLDB paper by Meek et al [25]. We thank the authors for furnishing the queries.

That query set consists of 100 queries ranging in length from 6 to 56 amino acids with an average length of 16. We tested this dataset against the complete yeast protein database of roughly three million amino acids using a search strategy in which all of the nearest neighbors of the query fragment were returned. The average search time per query sequence is about 3

seconds. For comparison, the average search time per sequence using BLAST on the same dataset with an E value of 20,000, as suggested by Meek et al., is about 0.2 seconds. At this small database size our Java-based system is an order magnitude slower. Given the vastly superior programming model, (SQL vs. Perl scripts), the broader applicability of mSQL and the opportunity on more complex problems for a query optimizer to produce algorithmically superior plans, we see this level of performance as a successful accomplishment. Given our scaling results one can anticipate that a port of mSQL to Postgres will yield BLAST-like speed on larger databases.

6. Application examples

Two application examples (three-way genome compare and miRNA) are given in the following section to demonstrate the functionality of *sequenceview* and the optimizability of the operators defined above.

6.1 Three-way genome compare

A paper in the journal Science recently reported on "Ultraconserved Elements in the Human Genome" [2]. It compared the human, rat, and mouse genomes in search of absolutely conserved orthologous regions, i.e. regions that exhibit 100% identity with no insertions or deletions. Each region was also required to be 200 base pairs in length or longer. This specification can easily be written as an mSQL query, one that is in fact more general.

The query assumes the following SQL schema:

```
HumanGenome {SID int, chromosome varchar, dna_seq DNA};
RatGenome {SID int, chromosome varchar, dna_seq DNA};
MouseGenome {SID, chromosome, dna_seq DNA}
```

To perform a three-way genome compare a sequenceview must first be created for each genome:

```
create sequenceview HV as
select SID, dna_seq
from createfragments(HumanGenome.dna_seq, 200)
using hamming.
```

Sequenceviews for the mouse and rat genomes are created with similar arguments.

The runtime query is as follows:

```
select HV.SID, RV.SID, MV.SID,
        merge(HV.dna_seq, RV.dna_seq, MV.dna_seq)
from HV, RV, MV
where distance('hamming', HV.dna_seq, RV.dna_seq) = 0
and distance('hamming', HV.dna_seq, MV.dna_seq) = 0
group by HV.SID, RV.SID, MV.SID,
        HV.dna_seq, RV.dna_seq, MV.dna_seq.
```

This query selects all subsequences of at least length 200 from the three genomes that exhibit a one hundred percent identity match. Notice that, although we specified a radius of zero as per the specification in the above-mentioned paper, any size radius could be specified without a significant degradation in the performance of the query, since we have created an index with O(logm) search times for each genome.

By utilizing the properties of our algebra, the query optimizer can create an optimal plan for this query. With H' = $\kappa_{dna\_seq:200}(H)$, R' = $\kappa_{dna\_seq:200}(R)$, and M'= $\kappa_{dna\_seq:200}(M)$, the above query is equivalent to:

$\lambda\gamma_{H'.SID, R'.SID, M'.SID, H'.dna\_seq, R'.dna\_seq, M'.dna\_seq}$
$(H' |\times|_{\delta(hamming, H'.dna\_seq, R'.dna\_seq) = 0} R')$ $|\times|$
$(H' |\times|_{\delta(hamming, H'.dna\_seq, M'.dna\_seq) = 0} M')$.

Since the distance function distributes over a metric join, the query optimizer can perform two joins instead of three. The query can be optimized to the following:

$\lambda\gamma_{H'.SID, R'.SID, M'.SID, H'.dna\_seq, R'.dna\_seq, M'.dna\_seq}$
$(H' |\times|_{\delta(hamming, H'.dna\_seq, R'.dna\_seq) = 0} R')$
$|\times|_{\delta(hamming, H'.dna\_seq, R'.dna\_seq) = 0} M'$.

See figure 7.

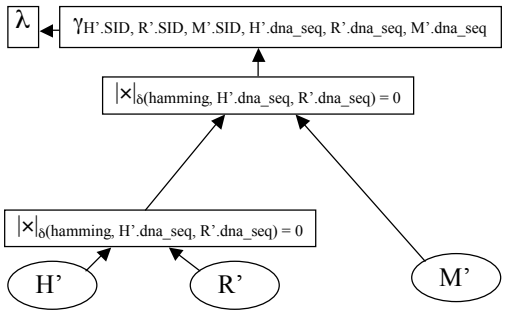

Figure 7. Optimized query plan for three-way genome compare

The optimizer can then use the associativity property of the metric join to further optimize the join such that the two smallest relations form the inputs to the topmost join.

These optimizations may appear rather mundane, but this is precisely the point. Since our algebra is based on a standard extended relational algebra, it is possible to reuse many of the optimizations already present in a standard query optimizer. This fact also suggests straightforward integration of this approach with existing systems.

6.2 miRNA query

MicroRNAs (miRNAs) are very small RNAs (about 22 nucleotides long) that may regulate gene expression in plants, animals and fungi. The mechanism of the regulation is interesting since an miRNA pairs with 3' UTR (untranslated region) of the mRNA of the regulated gene. TargetScan is a tool that predicts miRNA target sites conserved across multiple genomes. The first step of the algorithm is to search a set of orthologous 3' UTR sequences from one organism for perfect Watson-Crick complementary matches to bases 2-8 (from the 5' end) of the miRNA, and then extend matches allowing G-U pairs [23]. Such searching processes can be expressed in mSQL as shown below.

We assume the following schema:

```
miRNA {name varchar, seq RNA}
Genome {SID int, dna_seq DNA}
Features {SID int, name varchar, gene varchar, complement boolean, first int, last int}
```

The feature table contains ranges of mRNAs and CDSs (coding sequences).

First, sequenceviews are created for miRNA and genome sequences:

```
create sequencview M7 as
select * from createfragments(mirna.seq, 7) using hamming;
create sequenceview G7 AS
select * from createfragments(Genome.dna_seq, 7) using hamming.
```

A view is then constructed for 3' UTR sequence fragments using information from the feature table:

```
create view UTR as
select G7.* from G7, Features F1, Features F2
where F1.gene = F2.gene and F1.name = 'CDS'
and F2.name = 'mRNA' and F1.complement = true
and F1.SID = G7.SID
and (G7.dna_seq.offset between F2.first and F1.first).
```

For brevity, we only consider the coding sequences that are located on the complement strands. The queries for the other case are similar. The runtime query is as follows:

```
select merge(UTR.dna_seq) gseq, merge(M7.seq) mseq
from (select UTR.SID, UTR.dna_seq.offset goffset,
        M7.name, M7.seq.offset moffset
        from UTR, M7
        where M7.seq.offset = 1
        and distance('hamming',
        UTR.dna_seq, M7.seq) <= 0)
        as SEED, UTR, M7
where distance('hamming-rna', UTR.dna_seq, M7.seq) <= 0
and M7.name = SEED.name
and UTR.SID = SEED.SID
and UTR.dna_seq.offset – SEED.goffset =
        M7.seq.offset – SEED.moffset
group by UTR.SID, M7.name, UTR.dna_seq, M7.seq
having mseq.offset <= 1 and mseq.offset + mseq.length >= 7.
```

In this query, we first find 3' UTR fragments that are exact complementary matches of bases 2-8 of miRNAs using a metric join. Then we find matches (allowing G:U pairs) that are adjacent to the exact matches. The matched fragments are merged. The HAVING clause filters the result so that only merged miRNA fragments which are extensions of bases 2-8 are returned.

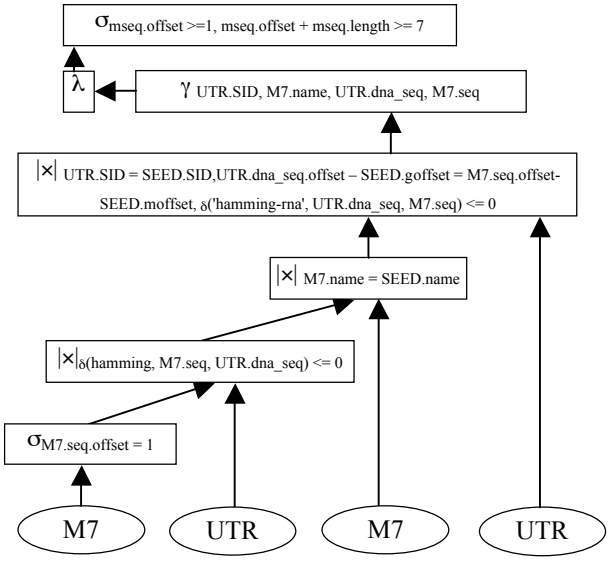

Figure 8. Optimized query plan for miRNA query

miRNAs are very short and there aren't many of them (It is estimated that there about 200-255 miRNA genes in the human genome [23]). Assuming there are fewer tuples in M7 than in UTR, the query optimizer can choose an indexed nested loop using the metric index of UTR for the metric join. After the SEED table is returned, there are two alternative paths to join it with UTR and M7. Since miRNAs are short, the optimal plan is to join the SEED table with M7 first. The result is joined with UTR by a theta join. The query optimizer can then choose the join order based on the selectivity of the attributes SID and name. The query plan is shown in Figure 8.

7. Related work

The growth of the amount of sequence data and the importance of manipulating it easily has produced dual trends. The growth has instigated algorithmic research into approaches where sequence data is preprocessed off-line and organized in data structures such that on-line queries can be executed quickly. The importance has instigated a number of SQL-level programming systems aiming at simplifying the coding of sequence analysis.

Algorithmically, interest in leveraging metric-space search methods to improve sequence analysis algorithms has been expanding. Modern efforts appear to have started with the SST system, which reports 1 and 2 order of magnitude speed improvements over BLAST [10]. SST comprises nearest-neighbor searches based on Hamming distance using a vector space mapping of q-grams and a TSQV tree. Results were reported only for sequence assembly. Chen and Aberer proposed a system composed of M-trees and a metric upper-bound on local alignment scores. No performance figures have been published. The most far-seeing work was a paper by Wang and Shasha in 1990, which was one of the earliest to propose algorithms to search metric spaces by precomputing and storing selected distances [35]. They further characterized the metric-join problem and presented results on finding similar proteins for a set of 151 proteins made up of less than 21 amino acids. Halperin et al. developed an embedding of the BLOSUM matrices in Hamming spaces and employed a probabilistic locality sensitive hashing algorithm to the protein sequence homology problem [16].

At the language level, Patel's work on PiQA, the Protein Query Algebra, is closest to mSQL [34]. In PiQA sequences are strings. To integrate biological semantics PiQA introduces two new types, *hit and match,* and a new operator, *match*. The *match* operator may be applied to a table containing strings and is parameterized by a matching model, which consists of either a regular expression or a sequence coupled with a similarity model. The output of the *match* operation augments the input table with a new set-valued attribute of type *match*. The individual values are hits, which are triples, containing the

position, length and score of the hit. Since match is set-valued, the PiQA algebra also requires that the query engine support the nest and unnest operators.

Since match can take a regular expression as an argument, PiQA does well with queries seeking to identify protein motifs. To date, the work does not speak to integrating methods beyond sequential scan for *matching* those expressions. Full genomic comparisons, (genome joins) are not yet well addressed in PiQA. On the other hand, mSQL, while addressing full genomic comparisons, does not speak to matching regular expressions. It seems likely that mSQL's fast access paths to q-grams could be used to accelerate regular expression matching. A fair characterization is that mSQL and PiQA have complementary strengths. In moving forward, an integration of the ideas is more likely than a competition.

A number of efforts have applied information retrieval algorithms to biological sequence retrieval. These commonly involve inverted indexes on q-grams [6,21,33,37]. These systems are proving to be very fast and useful when used as either a coarse filtering mechanism or when applied to genomic analysis problems on evolutionarily close sequences [5]. Another specialized index structure, the suffix-tree, has also been tested with respect to both accuracy and scalability on very large datasets [22].

8. Conclusion and future research

In his introduction and invited content to a recent special issue of the Data Engineering Bulletin on Genomic Databases, Patel makes the case that expanding the declarative programming abilities of SQL engines to include solving bioinformatics problems is a problem whose solution would provide new productivity to computational biology. The mSQL system is a step in this direction. An additional argument not developed in that special issue is: what is the penalty of not having such an ability? In addition to explosive growth in the amount of data in biology, biology is also facing explosive growth in the number of databases. We know that data federation is another large theme in biological databases. Our conjecture is that the lack of easy sequence query systems is one source of new databases. When biology problems like finding and identifying the function of conserved genomic regions in mammals is hard work, it makes sense that the results be entered in a new database and added to the corpus. If that same analysis takes an afternoon and can be recreated in a few hours on a workstation, then the value of those results may be insufficient to create a new database. All use case examples of mSQL queries come from biological studies that have archived their results in new databases.

Next steps for mSQL include more validation with more use cases. If the algorithms do not overcome the performance penalty of the Java implementation the system will be ported to Postgres or a commercial RDBMS.

Many open research problems remain. Our current implementation of a metric-space join is a straightforward indexed nested loop algorithm. When n is in the range of 10^7-10^{10}, even O(nlogn) algorithms become computationally challenging. Given a tree-structured access path one can anticipate merge-join like algorithms that would tend toward O(n) execution time, (assuming output size is small). Wang and Shasha detailed a simple recursive descent of the index for self-join that will work for most, if not all, tree-based methods of metric-space indexing [35]. Other than their contribution, the topic of metric-space joins is completely open. We anticipate that generalizing such an algorithm will necessarily make some assumptions about the underlying index and much work needs to be done.

10. References

[1] Altschul, S.F., Gish, W., Miller, W., Myers, E. W., and Lipman, D. J. Basic local alignment search tool. *J. Mol. Biol.* 215: 403-410, 1990.

[2] Bejerano, G, Pheasant, M, Makunin, I, Stephen, S, Kent, W.J., Mattick, J.S., and Haussler, D. Ultraconserved elements in the human genome. *Science*. 2004 May 28;304(5675):1321-5.

[3] Blakeley, J.A., Larson, P.A., and Tompa, F.W. Efficiently Updating Materialized Views. In *Proc. ACM SIGMOD*, 61-71, Washington D.C., June 1986.

[4] Bozkaya, T., and Ozsoyoglu, M. Indexing Large Metric Spaces for Similarity Search Queries. *ACM Transactions on Database Systems (TODS),* 24(3):361-404, 1999.

[5] Brin, S. Near neighbor search in large metric spaces. In *Proc. of the 21st Conference on Very Large Database (VLDB),* pages 574-584, 1995.

[6] Califano, A., and Rigoutsos. I. FLASH: A fast look-up algorithm for string homology. In *International Conference on Intelligent Systems for Molecular Biology*, pages 56-64, 1993.

[7] Chavez, E., Navarro, G., Baeza-Yates, R., and Marroquin, J.L. Searching in metric spaces. *ACM Computing Surveys*. 33(3): 273-321, 2001.

[8] Ciaccia, P., Patella, M., and Zezula, P. M-Tree: An Efficient Access Method for Similarity Search in Metric Spaces. In *Proc. of the 23rd Conference on Very Large Database* (*VLDB)*, Athens, Greece, August 25-29, 1997.

[9] Ghias, A., Logan, J., Chamberlin, D., and Smith, B.C. Query by humming - musical information retrieval in an audio database. In *ACM Multimedia*, 1995.

[10] Giladi, E., Walker, G.M., Wang, J.Z., and Volkmuth, W. SST: an algorithm for finding near-exact sequence matches in time proportional to the logarithm of the database size. *Bioinformatics*. 18(6): 873-879, 2002.

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

[12] Gravano, L., Ipeirotis, P.G., Jagadish, H.V., Koudas, H., Muthukrishnan, S., and Srivastava, D. Approximate String Joins in a Database (Almost) for Free. In *Proc. of the 27th Conference on Very Large Databases (VLDB),* pages 491-500, 2001.

[13] Gribskov, M., and Robinson, N. L. Use of receiver operating characteristic (ROC) analysis to evaluate sequence matching. *Computers and Chemistry*. 20(1): 25-33, 1996.

[14] Gusfield, D. Algorithms on Strings, Trees and Sequences Computer Science and Computational Biology. *Press Syndicate of the University of Cambridge, USA,* pages 449-454, 1997.

[15] Guttman. A. *R-trees:* A Dynamic Index Structure for Spatial Searching. In *Proc. of SIGMOD,* 1984.

[16] Halperin, E., Buhler, J., Karp, R., Krauthgamer, R., and Westover, B. Detecting Protein Sequences Via Metric Embeddings. In *Proc. of the Eleventh International Conference on Intelligent Systems for Molecular Biology (ISMB 2003),* pages 122-199, Brisbane, Australia, 2003.

[17] Hjaltason, G.R. and Samet, H. *Index-driven similarity search in metric spaces. ACM Transactions on Database Systems (TODS),* 28(4), 2003.

[18] Hochbaum, D. S., and Shmoys, D. B. A best possible heuristic for the k-center problem. *Mathematics of Operational Research,* 10(2):180-184, 1985.

[19] Jaeschke, G., and Schek, H.J. Remarks on the algebra of non first normal form relations. In *Proc. of the 1st ACM SIGACT-SIGMOD symposium on Principles of Database Systems*, Los Angeles, California, March 29-31, 1982.

[20] Karakoc, E., Ozsoyoglu, Z.M., Sahinalp, S.C., Tasan, M., and Zhang, X. Novel Approaches to Biomolecular Sequence Indexing. In the *Bulletin of the IEEE Technical Committee on Data Engineering.* 27(3):40-47, 2004.

[21] Kent, W. J. BLAT-The BLAST like alignment tool. *Genome Research*, 12(4):656-664, 2002.

[22] Lenhard, B., Sandelin, A., Mendoza1, L., Engstrom, P., Jareborg, N., and Wasserman, W.W. Identification of conserved regulatory elements by comparative genome analysis. *Journal of Biology,* 2(13), 2003.

[23] Lewis, B.P., Shih, I.H., Jones-Rhoades, M.W., Bartel, D.P., and Burge, C.B. Prediction of Mammalian MicroRNA Targets. *Cell*, 115 (7), 2003.

[24] Mao, R., Xu, W., Singh, N., and Miranker, D. P. An Assessment of a Metric Space Database Index to Support Sequence Homology. In *Proc. of the 3rd IEEE Symposium on Bioinformatics and Bioengineering*, Washington, D.C., March 10-12, 2003.

[25] Meek, C., Patel, J.M., and Shruti Kasetty, S. OASIS: An Online and Accurate Technique for Local-alignment Searches on Biological Sequences. In *Proceedings of the 29th Conference on Very Large Databases (VLDB),* pages 910-921, 2003.

[26] Miranker, D.P., Briggs, W.J., Mao, R., Ni, S., and Xu, W. Biosequence Use Cases in MoBIoS SQL. In the *Bulletin of the IEEE Technical Committee on Data Engineering.* 27(3): 3-11, September 2004.

[27] NCBI BLAST. http://www.ncbi.nih.gov/BLAST. Accessed February 28, 2005.

[28] Patel, J.M. Letter from the Special Issue Editor. In the *Bulletin of the IEEE Technical Committee on Data Engineering.* 27(3):2-3, September 2004.

[29] Sahinalp, C., Macker, S.J., Tasan, M., and Ozsoyoglu, M. Distance Based Indexing for String Proximity Search. In *Proc. of the 19th International Conference on Data Engineering (ICDE)*, pages 125-138, Bangalore, India, March 5-8, 2003.

[30] Schaffer, A.A., Aravin, L., Madden, T.L., Shavirin, S., Spouge, J. L., Wolf, Y.I., Koonin, E.V., and Altschul, S.F. Improving the accuracy of PSI-BLAST protein database searches with composition-based statistics and other refinements. *Nucleic Acids Res*., 29(14): 2994-3005, 2001.

[31] Sellers, P.H. On the theory and computation of evolutionary distances. *J. Appl. Math. (SIAM),* 26: 787-793, 1974.

[32] Stuart, J. M., Segal, E., Koller, D., Kim, S.K. A Gene Coexpression Network for Global Discovery of Conserved Genetic Modules. *Science* 302: 249-55, 2003.

[33] Tan, Z., Cao, X., Ooi, B.C., Tung, A.K.H. The ed-tree: an index for large DNA sequence databases. In *Proc. of the 15th International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 151-160, 2003.

[34] Tata, S., and Patel, J. PiQA: An Algebra for Querying Protein Data Sets. In *Proc. of the 15th International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 141-151, 2003.

[35] Wang, T.L., and Shasha, D. Query processing for distance metrics. In *Proc. of the 16th International Conference on Very Large Databases (VLDB)*, pages 602-613, August 1990.

[36] Weber, R., Schek, H.-J. , and Blott, S. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of the Int. Conf. on Very Large Data Bases (VLDB),* pages 194-205, New York City, New York, August 1998.

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

[38] Xu, W., Briggs, W. J, Padolina, J., Liu, W., Linder, C. R., and Miranker, D. P. Using MoBIoS' Scalable Genome Joins to Find Conserved Primer Pair Candidates Between Two Genomes. *ISMB Bioinformatics*, 20(Supp1): i355-363, 2004.

[39] Xu, W., and Miranker, D.P. A metric model for amino acid substitution. *Bioinformatics*, 20(8): 1214-21, 2004.

[40] Xu, W., Miranker, D.P., Mao, R., and Wang, S. Indexing Protein Sequences in Metric Space. TR-04-06, October 2003. (http://www.cs.utexas.edu/users/mobios).