Emulating BLAST using database technology

Hans Philippi

Department of Information and Computing Sciences, Utrecht University

Technical Report UU-CS-2007-030

www.cs.uu.nl

ISSN: 0924-3275

Emulating BLAST using database technology

Hans Philippi

2007

Abstract

Sequence alignment is very important for molecular biologists. Blast is a well-known family of heuristic approaches to solve this problem efficiently. Special purpose algorithms and datastructures, however, do not support the data abstraction offered by database technology. In this paper, we show that modern extensible database management systems, like the Monet system, allow performance comparable to the special purpose solution, while offering the required data abstraction.

1 Introduction

There is no doubt about the importance of sequence alignment for molecular biologists. Homology searching means matching a specific genome string, the query, to a large collection of already known strings, the database. The database can either contain nucleotide strings, based on the ACGT-alphabet or amino acid strings, based on a twenty letter alphabet. Due to evolutionary changes, strings that are related show to some extend differences. So essentially we are dealing with approximate string matching on very large string collections. We focus on local alignment, i.e. matching between parts of the query and database strings; note that this is the most common type of search in bioinformatics.

Approximate string matching is solved by dynamic programming techniques. In the case of protein databases, the specific biochemical domain knowledge is represented in the underlying model by refined valuations of matches and mismatches (e.g. BLOSUM matrices, affine gap costs). See [2] for a thorough treatment. The Smith-Waterman algorithm is a dynamic programming technique that solves the local alignment problem, but scales quadratically. Performance requirements have forced the development of heuristic approaches, of which Blast ([3], [1]) is the most well-known.

Although the term *database* is widely used among people dealing with large collections of data in any format, computer scientists tend to have a narrower view on this concept. Crucial to them is the presence of a database management system (DBMS), providing access to and management of data at a higher level of abstraction. Query languages, such as SQL, are logic based and data access is formulated in a very concise way. Building a sequence alignment tool on a DBMS allows you to deal with your data set in a versatile way.

Gravano e.a. have shown how to query a string database using SQL ([4]). The burden of finding a good access strategy to the data is left to the query optimizer, although Gravano c.s. 'assist' the query processor at the SQL level. Some preliminary experiments have shown us that when this approach is applied in the case of protein strings, the performance is very unsatisfying. On a Xeon 1.8 machine with 2GB of memory, Blast answers queries on a Swissprot size database within one or two seconds, where a Gravano-like approach on Postgres

needs minutes. In spite of the overwhelming amount of effort put into the problem of generic query optimization for the relational model (see [6] for an overview), a dedicated and tuned program will in general outperform a classic relational database. To state it in a concise way: a DBMS gives you the comfort and ease of abstraction at the cost of performance.

goals

Our main goal is: *how to be fast without losing abstraction*. We propose a compromise between speed and data abstraction. We simulate Blast on a DBMS and reach comparable performance. Note that we do not have the ambition to outperform Blast, because it would mean that generic optimization would do better than domain specific optimization, which is quite unlikely.

A secondary goal is to show that the relational approach gives us the possibility to deal with different alignment methods in an easy way. The alignment heuristics can typically be split into two phases. During the *filtering* phase, the database is searched for promising spots. In the *expansion* phase, these spots are further investigated. Filtering is a typical DBMS task, whereas expansion is done by selected algorithms. Managing our data with a DBMS gives us the possibility to vary with these two phases in a flexible way. With a minimal amount of effort, we can switch from the Blast two-hit diagonal filtering to the filtering methods of Cafe, as proposed in [7]. Also the recent proposals to apply non-consecutive q-grams (see e.g. [8]) can easily be adopted in our prototype.

2 Preliminaries

Definitions

- A *string* is a mapping from an integer interval $[k..n]$ to the set of characters. We have the notion of substring. An *aastring* is a string based on a limited character set: the amino acid symbols.

- A *q-gram* is an amino acid string of length q, a fixed number that is typically 3 for protein databases. The term *word* is a synonym for q-gram and more common in the Blast community.

- Basically, our *database* is a set of aastrings, which all have $k = 1$. We have a fixed *query*, which is also an aastring having $k = 1$. Every aastring in our database is related to one annotation string, that gives some comments on the origins of this aastring.

- Between two amino acid (sub)strings, we have a notion of *similarity*, which expresses in a quantitive sense how strong the strings are related. Our similarity calculations are based on the techniques of [2] using a BLOSUM62 matrix.

- Two q-grams are *neighbours* if their similarity is above some given threshold for the given similarity metric.

- A *hit* is a 4-tuple $< s, j, qs, i >$ where s is an aastring in the database, qs is a query, and $s[j, j + q - 1]$ and $qs[i, i + q - 1]$ are neighbours.

  For a hit $< s, j, qs, i >$, we define the *diagonal* as $j - i$.

- A *candidate* is a pair of hits that refer to the same aastring s, the same query and have the same diagonal. They are nonoverlapping: $|j_1 - j_2| \geq q$ and within a given distance A: $|j_1 - j_2| \leq A$. The default value for A in BLASTP is 40.

- An *alignment* is a 7-tuple
  $< s[j - left, j + right], j, qs[i - left, i + right], i, left, right, score >$
  where s is an aastring in the database, qs is the query, and $s[j - left, j + right]$ and $qs[i - left, i + right]$ have a similarity $score$ according to our metric.

Example

```
qs = CWYWRWYY
 s = RRWYWAWYYRR
```

We have a hit $< s, 2, qs, 1 >$, because RWY is a neighbour of CWY: R and C are close according to BLOSUM62 and the last two characters are identical in both q-grams. We have a second hit $< s, 3, qs, 2 >$, that overlaps with the previous one. We have a another hit $< s, 7, qs, 6 >$ for WYY. The first and the third together are a candidate, because they have the same diagonal: $3 - 2 = 7 - 6$. Also the second and the third hit are a candidate.

Below you see a Blast-like representation of an alignment based on the last candidate: $< RWYWAWYY, 3, CWYWRWYY, 2, 1, 6, 50 >$. The score is again based on BLOSUM62.

```
Query:       1 CWYWRWYY
Sbjct:       2 RWYWAWYY
```

3 A relational view on Blast

We will describe how the two-hit diagonal filtering technique of BLASTP (the protein alignment version of Blast) can be simulated on a relational database. We choose to formulate the constituent queries in the relational algebra (RA). Almost every textbook on databases gives an introduction into the RA, which is generally based on set semantics.

The reason for this choice has to do with our DBMS, Monet. One of its front-end languages is Mil. Mil is conceptually based on the RA, but it offers more. One of the extensions is the multiplex operator, that allows you to evaluate any function with the appropriate signature over all rows in a table. To stay close to the expressiveness of Mil, we will use an extended version of the RA, which is based on ordered bag semantics and allows for sorting, grouping and aggregation. Translation to SQL is straightforward. See [5] and [6] for more details.

As already noted, the filtering phase will be based on the two-hit diagonal method. It is a matter of choice which expansion algorithm we will use. For our experiments, have chosen to adopt the simplest case, ungapped expansion, because we are primarily interested in the filtering performance.

Our knowledge of the Blast processing is based on [1]. We are aware of the fact that the actual implementation of BLASTP is more complicated than described here, but many of the details are (to our knowledge) undocumented in the scientific literature.

The database consists of the following tables, which can easily be generated for any protein database in the Fasta format, which is a character based format that is often used to distribute genome string collections. In Fasta data sets, the strings are listed interleaved with their

Table 1: Example database

Strings	
id	*string*
1	RRWYWAWYYRR
2	RRRWYWAWYWRR
3	RRWYWAAWYYRR

Annots	
id	*annot*
1	comments on string 1 ..
2	comments on string 2 ..
3	comments on string 3 ..

Qgrams		
id	*j*	*qg*
1	1	RRW
1	2	RWY
1	3	WYW
...	...	...
3	10	YRR

Qdecomp	
i	*qg*
1	CWY
2	WYW
3	YWR
...	...
7	YYC

Nbh	
qg	*qg2*
CWY	CWY
CWY	AWY
CWY	RWY
...	...
...	...

annotation. We have a simple tool to split this set into the `Strings` and `Annots` tables and to generate the q-grams.

```
Strings(id, string)
Annots(id, annot)
Q-grams(id,i,qg)
Qdecomp(j,qg)
Nbh(qg,qg2)
```

Strings are identified with a system generated `id`. It maintains the connection between the aastrings, the annotation strings and the q-grams. So for each aastring in `Strings`, we have a describing tuple in `Annots`.

The position of the q-grams in the strings is denoted by a j in the strings and an i in the query. The annotation table joins in (literally) at a very late stage. Table 1 shows our example database.

Our example query is CWYWRWYYC. Its q-gram decomposition can be found in the Qdecomp table.

BLASTP also finds approximate hits. We could calculate the q-gram neighbourhood, but in our case we prefer to join with a precalculated symmetric table of pairs of neighbour q-grams to obtain an extended version of Qdecomp. We have chosen for a minimal similarity of 11 (default) according to BLOSUM62.

$$Qdecompx := \pi_{j,qg \leftarrow qg2}(Qdecomp \bowtie Nbh) \tag{1}$$

The next step is finding the hits. This is also a straightforward natural join.

$$Hits := \pi_{id,i,j,diag \leftarrow (j-i)}(Qgrams \bowtie Qdecompx) \tag{2}$$

According to the two-hit diagonal filtering method, candidates are defined by two hits in the same string and on the same diagonal. This can be expressed by a self join on $Hits$.

$$Hits2 := \pi_{id2,i2,j2,diag2}(Hits) \tag{3}$$

Hits2 should be interpreted as an alias of (or view on) the $Hits$ table, not as a physical copy.

Having found the hits, we are able to calculate the candidates:

$$Candidates := \pi_{id,i,j}(Hits \bowtie_{\theta} Hits2) \tag{4}$$

where θ denotes the join condition:

$$\theta : (id = id2, diag = diag2, (i2 - i) \geq q, |i - i2| \leq A)$$

Note that the $(i2 - i) \geq q$ requirement also prevents double occurences of hit pairs.

Based on our knowledge of the PBLAST filtering methods, we come to the following conclusion:

Theorem 1 *The RA-expressions (1) to (4) cover the candidates that are found by the PBLAST filtering approach.*

The expanding phase requires the execution of an expansion algoritm on all candidates. Having a function `expand` with the appropriate signature

```
expand: <id,i,j> -> <id,i,j,sfrag,qsfrag,left,right,score>
```

we can evaluate this function for every row in the Candidates table. We have chosen for simple, ungapped alignment. In the experiments, the Blast parameters were set accordingly. `Expand` calculates an alignment and its score according to out similarity metric. `j-left` and `j+right` mark the borders of the matching interval in the database string. Using i in stead of j, we get the borders of the matching query substring.

The resulting set of tuples is filtered on the score value. For representation purposes, it is sorted in descending order on score.

$$Result := \downarrow_{score} (\sigma_{score>=threshold}([\ expand\ ]\ Candidates) \bowtie Annots) \tag{5}$$

So we see that the BLASTP filtering and expansion phases can be expressed in five (or essentially four) relational algebra expressions.

4 The Monet approach

Our DBMS of choice is Monet, developed at the CWI in Amsterdam ([5]). There are a few reasons for this choice.

- *main-memory approach*
  Monet is a main-memory DBMS (MMDBMS). The data and indexes are supposed to be resident in main-memory. Where the classical DBMS focuses on minimizing IO, Monet gains performance by optimizing for main-memory access and by applying cache-conscious techniques. A MMDBMS is not very suitable in a typical transaction processing environment, with high update rates. Its power will show up in cases where updates are rare and usage is read-intensive. Sequence alignment clearly fits this usage profile.

  With an amount of 2GB of memory, we can deal with medium-size protein databases. Experiments have been done with subsets as large as one-third to one-half of Swissprot, without memory related problems like swapping.

- *extensibility*
  Monet is extensible. You can add procedures and functions (in C/C++). You can execute these methods interleaved with the data manipulation primitives. Typically the expansion phase of Blast requires this feature.

- *layered levels of access*
  Monet provides a SQL interface, as well as an XQuery interface. Queries in these languages are translated into a lower level algebraic language: Mil. However, Mil is also available for end-users, although it is not primarily intended for this purpose. Finally, you even have the possibility to write specific algebraic operators in C and add them to the Mil collection. This makes sense in the case of very performance critical operations. Of course, a point of consideration here is the trade off between abstraction in data access and performance.

The RA-expressions of the previous section can be translated into Mil, Monets query algebra. This translation is complicated by the fact that Monet has adopted a binary data model. We will not go into details with respect to the decomposition problems (see [5]). Instead, we will keep our discussion at the level of the standard RA.

However, it should be noted that Monet is very eager on memory economy. The database is represented by columns that have absolute minimal memory requirements. The connecting identifiers are in most cases virtual and require no space. Still, look-ups based on the identifiers are extremely fast.

5 Measurements and discussion

We have experimented with two data sets. The first data set consists of the first 60.000 lines of a recent Swissprot database (which is about 1/3 of the total set). The second set takes the first 100.000 lines. Both selections allow the whole database to be in main memory. We took samples of different places in the string collection to serve as queries. Furthermore, we ran queries with lengths ranging from 20 to 100. We counted milliseconds.

The tests were done on a dual processor Xenon 1.8 GHz machine with 2GB of main memory and running under Linux. Besides the total running time of our Blast prototype, we list the performance on steps (2) and (5), because they are the most time consuming. The running time of step (1) can be ignored. Step (3) is only virtual.

Our first observation is that with respect to the running times, we are in the same order of magnitude of Blast for short queries. In that sense, our goal to *approach* the Blast performance has been reached.

All of the mentioned operators were written in Mil, except step (4). For performance reasons, we decided to write a Mil-extension in C.

For longer queries, the difference is increasing. Note that Blast is more selective in step (4), because it has an additional local maximality requirement for Candidates. We could add this feature to our version of (4), but we decided that it would lead us to far away from the relational approach.

A final remark concerns the dual processor architecture of our machine. We noticed that Monet makes, to some extend, use of its possibilities, although it is not clear which operations benefit from it. One should estimate the running time advantage to be a factor somewhere between 1 and 2. This aspect is not reflected by the figures. On the other hand, one could

Table 2: Test results

Testset 60k				
length	Blast	Monet-total	step (2)	step (5)
20	460-470	630-670	530-580	60
30	510-540	1050-1070	730-740	240
50	470-590	2080-2360	1380-1450	480-670
100	680-700	4670-5183	2629-3000	1590-1700

Testset 100k				
length	Blast	Monet-total	step (2)	step (5)
20	760-760	1540-2000	1400-1810	90-120
30	790-850	2780-2930	2140-2470	340-520
50	940-940	5580-5680	4020-4200	1090-1280
100	1070-1190	12450-12780	8320-8450	2710-2910

argue that the fact that your DBMS supports parallellism is for free and inherent to our approach. It does not affect our conclusion that we approach the Blast performance.

6 Let's do CAFE too

In [7], Williams and Zobel propose an alternative filtering method. In stead of the two-hit diagonal filtering, they experiment with a few other q-gram based characteristics: *framecount*, *coverage* and *length*.

Using the example below, we will explain these concepts briefly. For simplicity, we identify equality and neighbourhood of q-grams. Note that the one step shifted representation of `qs` in the figure below corresponds with a diagonal value 1.

For this diagonal, we have 7 exact letter matches (= coverage). The number of matching 3-grams is 3 (= framecount). The total length of the substring of s from the leftmost to the rightmost lettermatch is 8 (=length).

```
 s = RRWYWAWYYRR
       ::: ::::
qs =  CWYWRWYYRW
```

Filtering is based on a function

```
f: <s,qs,diag,length,coverage> -> <score>
```

We can determine these parameters for each string/diagonal combination using the grouping operator $\Gamma_{grp,function}$ of the relational algebra.

$$Framecounts := \Gamma_{id,diag,count}(Hits) \tag{6}$$

$$Lengths := \Gamma_{id,diag,max(j)-min(j)+1}(Hits) \tag{7}$$

The coverage calculation cannot be expressed directly in the RA, but it is easily expressed in Mil.

7 Conclusions and further research

Our main goal was: *how to be fast without losing abstraction* and we note that it contains two claims.

The first is *being fast*. With running times that range from 1.5 to 10 times the running times of Blast, we conclude that we have approximated its performance.

The second is *without losing data abstraction*. The filtering and expansion is fully realized with the possibilities offered by Monet. In one case, we decided to implement an operator at the Mil/C extension level.

Our secondary goal was to illustrate the *advantages of data abstraction*. Although we have no experiments done yet to compare Blast with Cafe, out platform provides a way to realize such experiments quickly. So our prototype can also be regarded as a suitable platform for performance comparison of alignment methods.

With respect to future research, we have the ambition to attack the multiple alignment problem with Monet. Note that our approach enables us to reuse the existing subqueries, but with full flexibility for adaption and tuning.

8 Acknowledgements

The author thanks Martin Kersten at the CWI, Amsterdam, for his hospitality during the development of the prototype. Special thanks go to Stefan Manegold, for advice and support.

References

[1] Korf, I., Yandell, M., Bedell, J.: *Blast.* O'Reilly, 2003

[2] Durbin, R., Eddy, S.R., Krogh, A., Mitchison, G.: *Biological Sequence Analysis.* Cambridge University Press, 1998

[3] Altschul, S.F., Gish, W., Miller, W., Myers, E.W., Lipman, D.J.: *Basic local alignment search tool.* Journal of Molecular Biology 215 (1990) 403–410

[4] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava: *Approximate string joins in a database (almost) for free.* In Proc. of the Int. Conf. on Very Large Databases (VLDB), pages 491–500, Roma, Italy, Sept. 2001. Morgan Kaufmann Publishers Inc.

[5] Boncz, P.A., Kersten, M.L.: *MIL Primitives for Querying a Fragmented World.* The VLDB Journal, Vol. 8 (1999) 101-119

[6] Garcia Molina, H., Ullman, J.D., Widom, J.D.: *Database System Implementation.* Prentice-Hall, 2000

[7] Hugh E. Williams, Justin Zobel: *Indexing and Retrieval for Genomic Databases.* IEEE Transactions on Knowledge and Data Engineering, Vol. 14 (2002), 63-78

[8] Ma, Tromp, Li: *Patternhunter: faster and more sensitive homology search.* Bioinformatics Vol.18 No. 3 (2002)