Design and Implementation of a String Database Query Language

Gösta Grahne[a], Raul Hakli[b*], Matti Nykänen[b†], Hellis Tamm[b‡] and Esko Ukkonen[b§]

[a]Department of Computer Science, Concordia University, Sir George Williams Campus, 1455 De Maisonneuve Blvd. West, LB 903 11, Montreal, QC, Canada, H3G 1M8.

[b]Department of Computer Science, P.O. Box 26 (Teollisuuskatu 23), FIN-00014 University of Helsinki, Finland.

Current data management and information retrieval systems lack advanced string processing capabilities needed in string-oriented application areas like computational molecular biology. Several theoretical models for string processing have been proposed but they either have not been implemented in practice or the implementations are too restricted or platform-dependent to be generally useful. In this article, we introduce the language Alignment Declarations designed for string querying and restructuring. The language extends the capabilities of existing database query languages by allowing the user to define database predicates that express structural properties of strings (e.g. containment of certain patterns) or relations between several strings (e.g. similarity measures). These predicates can be created and executed within the same database session and also stored for later sessions. We also describe the design and implementation of a working system.

Keywords: String databases; Query languages

1. Introduction

One of the basic data types in data management systems is the *string*, a variable length sequence of characters in some alphabet, typically ASCII. Items of string data, like items of any other basic data types are *atomic*, meaning that they are treated as indivisible entities without internal structure, not as complex objects composed of some simpler constituents. This reflects one of the leading ideas in the relational data model [1, Chapter 3], namely to reveal whatever structure the data has by organizing it into a collection of relations and normalizing that collection in order to eliminate nonsimple domains whose elements are not atomic.

Strings, however, differ from the other basic types of data such as integers, reals, characters and Boolean values in one important aspect: a string is a composition. It is composed of some other, simpler entities, in fact of other basic data items, namely characters. So strings are not atomic in the same sense the other data items are.

*Partially supported by the Academy of Finland grant 18814.
†Partially supported by the Academy of Finland grant 42977.
‡Supported by the Academy of Finland grant 22586.
§Supported by the Academy of Finland grant 44449.

Nevertheless, treating strings as atoms is sufficient for most cases. Even though people's names, for instance, consist of several letters and are thus syntactically molecular, it rarely makes a semantical difference what particular letters constitute a given name. One does not ordinarily need to perform queries for names that contain the letter 'a' but not the letter 'o', for example. Usually it is enough to query strings using the identity relation for comparison and, occasionally, an ordering relation for performing alphabetical sorting. Practical implementations offer an additional capability of substring matching in the form of the LIKE-operator of SQL or something similar.

In some fields, however, the constitution of a string does make a difference. A linguist may be interested in certain syntactical structures within words and a molecular biologist may be looking for certain patterns of nucleotides that may imply the presence of a gene in a DNA molecule. In situations like these treating strings as atoms is insufficient (after all, DNA is not an atom but a molecule), rather it is necessary to be able to express relations that take the inner structures of the strings into account. For example, one might want to find all strings that contain a certain pattern defined as a regular expression, but there is no way to employ that regular expression as a selection condition in an SQL query. The LIKE-operator is restricted to finding exact substrings only, not patterns defined as regular expressions. Even extending SQL to allow that kind of pattern matching would not be sufficient for application areas like computational molecular biology. Biologically interesting sequence structures contain dependencies between different parts of a sequence. Expressing those kinds of dependencies requires a more expressive language than regular languages, in some cases it is even necessary to go beyond context-free languages [2,3].

Another problem is that there are also interesting structural relations between several strings, for instance similarity or distance measures [4, Chapter 3.6.1] or intermolecular connections [5] in bioinformatics. Currently, there is no way to perform any structure-based string comparison or to use an arbitrary string relation as a join condition. Even a simple task like substring matching is possible only by using the LIKE-operator to select strings that contain a certain constant substring, but it is not possible to make a more general query and ask for all pairs of strings such that the one is a substring of the other.

Sometimes it would also be useful to have the possibility to generate new strings by performing certain operations to some of the strings contained in the database. For example, a linguist might want to generate new words by concatenating certain prefixes with certain nouns. Or a biologist might want to take a DNA table and construct a corresponding RNA table by performing a DNA to RNA transcription operation for the DNA strings. In cases like these the queries should not only extract data from the database but also generate new string data.

So this is what we call the problem of strings in databases. Long strings are clearly complex objects but the database management systems treat them as atomic. Data-intensive string-oriented application areas require possibilities for complex pattern matching for selecting individual strings, possibilities for comparing different strings and possibilities for generating new strings by combining and modifying existing strings. All these tasks require relaxing the atomicity condition on strings and call for a capability to go inside strings and inspect their constitutive characters.

Even though numerous theoretical approaches have been proposed to solve the problem of strings in databases [6–13], few working implementations have been reported. One difficulty in this area is the trade-off between the increased complexity and the practical usefulness of the language. Even if a theoretical solution exists, it is labourious to come up with an easy-to-use high level language for formulating the queries and an efficient query evaluation engine for performing the computations. This has led some researchers to suggest very modest extensions of the relational language [11,13] while others have defined very powerful languages and then set restrictions for safety and other reasons [9]. In our work, the latter approach was chosen because we wanted to avoid committing ourselves to limited application areas or imposing other a priori restrictions. Instead we preferred an approach where the user would be free to express any computable query and we could then try to determine whether it is safe to actually evaluate that query.

Another problem is that the models and implementations easily become too tightly bound to specific database models or data management systems. So string handling extensions have been presented in context of relational systems [7,8,11,14–16], object-oriented models [17,18] and Datalog [9,10] but few of the ideas carry easily over from one context to another. Some of the approaches even bypass the data management issue altogether and concentrate only on the pattern matching and string manipulation task [19,20]. Consequently, none of the suggestions above have yet found their way into everyday practice of string data management. Data intensive application areas like molecular biology, for example, are still depending on restricted querying possibilities and the separation of data storages and data manipulation tools.

This article addresses the questions involved in designing a string-oriented database query language extension and implementing it in practice. It collects together and expands upon the results presented in our previous conference articles [21,18,22,23]. First we presented the query language AQL [21] which extended SQL with our Alignment Declaration Language designed for string manipulation and based on the modal Alignment Calculus [8]. We then reported an implementation of the language [18] for the O_2 object-oriented database system, and began planning a database-independent version of the system [22,23]. This article completes our undertaking to design and implement a string query extension that can be used to supplement existing database management systems. The complete specification of the Alignment Declaration language together with the design of its implementation are presented here for the first time.

We will start our discussion by considering the overall situation of strings and data management in Section 2. In Sections 3 and 4, we will present the full syntax and semantics of the Alignment Declaration Language, first informally (Section 3) and then formally (Section 4). We will then go over to the practical part and describe the design of our implementation in detail in Section 5. The paper is concluded in Section 6.

2. Outline of the Approach

Our primary aim is to develop a string handling extension which could be incorporated into various different database management environments. A natural starting point is to develop one for the relational data model because it captures several essential features of data management in general. Several extensions of the relational model have been proposed [7,8,11,14–16]. Others have adopted different starting points such as Datalog [9,10] or object-oriented data models [17], but their string handling features tend to mix with – or even depend on – the features of the underlying query languages, while we wish to keep these two levels separate. That is, our aim is to offer a string handling mechanism which could treat strings as indivisible entities on the query language level and as lists of characters via our extension.

As an introduction to our approach, consider the following simplified biological data management issue. The EMBL nucleotide sequence database [24] contains an entry according to which the sequence

$$\underbrace{\texttt{ACTTA}}_{\text{human,}}\underbrace{\texttt{GTAGT}}_{\text{target}}\underbrace{\texttt{TGGAAGGGCTAATTCACTCCCAAAGAAGACAAGAT}}_{\text{and viral part}}$$

is human DNA infected by human immunodeficiency virus (HIV). The target is a part within human DNA after which the viral part is inserted. This piece of biological information suggests three string database querying tasks:

String selection: Are there other similarly infected sequences in the data?

> In database terms, select those strings which contain the target and the viral parts as a contiguous substring.
>
> More generally, select those strings that match some user-defined pattern written as, say, a regular expression.

String comparison: Is this sequence an infected form of that sequence?

> In database terms, compare strings s_1 and s_2 to see if s_1 can be obtained by inserting the viral part after some occurrence of the target part within s_2. If so, then report that the corresponding database entries are indeed related in this manner.
>
> Again, this comparison criterion must be definable by the user, since it specifies what kind of relatedness he is currently seeking in the data.

String generation: What would the data look like if infected in this manner?

> That is, the strings s_2 are in the database, and the user seeks those strings s_1 – *not* in the database – for which the "infected-form-of" relation above holds.

We propose to perform all these three tasks with a single unified string handling mechanism, namely *user-defined string comparison criteria, together with the added capability of producing strings that meet these criteria*. String selection is clearly a special case of string comparison where just one string is being compared against the user-defined criterion. String generation can also be recast as string comparison, if the generated strings s_1 are viewed as coming from a hypothetical infinite database table of all possible strings, just like the existing strings s_2 come from an actual database table. In this way the current concept of relatedness needs to be expressed only once – as a string comparison criterion – even though it has multiple uses.

In computational molecular biology, relatedness of sequences is measured with *alignments* [25, Part III]. For instance, the meaning of our "infected-form-of" relation is that

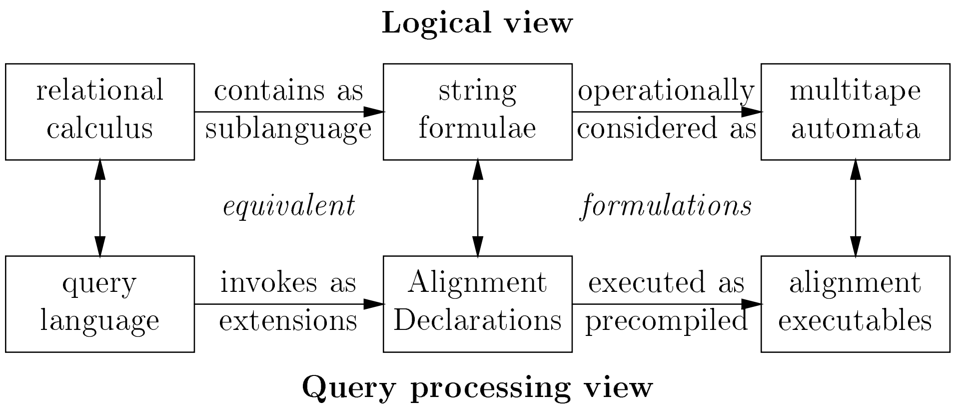

Figure 1. Different views to our approach [23, Figure 1].

the strings must align as follows:

$$\begin{aligned} s_1 &= \text{something }\texttt{GTAGTTGGAAGGGCTAATTCACTCCCAAAGAAGACAAGAT}\text{ other thing} \\ s_2 &= \underbrace{\text{something}}_{\text{exact match}}\ \texttt{GTAGT}\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ \underbrace{\text{other thing}}_{\text{exact match}} \end{aligned}$$

Here the period '`.`' denotes a gap, or omission of a character, in the alignment. Alignments like this can be seen as series of character-to-character matches and mismatches: in this case, first all characters must match pairwise exactly, then the target part must occur in both strings, then the viral part must match a gap, and finally the tails of the two strings must again match exactly. Accordingly, our language for specifying string comparison criteria will be a language for specifying such character comparison series.

Our overall approach is depicted in Figure 1: Viewed logically, we add into standard relational calculus a separate sublanguage called *String Formulae* [8, Section 2] for expressing user-defined predicates – that is, comparison criteria as rules for character comparison series – on strings. This sublanguage is based on (extended linear temporal [26]) logic, so that it fits its host language well. Some proposals [7,9,27] have developed sublanguages based on computing devices called transducers [28, Chapter 2.7], but in our opinion this introduces the concepts of *input and output* already in the string handling language, and therefore already in the supposedly declarative query design stage. Instead, we try to keep our language logical, and therefore wish to introduce these two computational concepts as late as possible, preferably only at the query evaluation stage. Only at that stage do we resort to computing devices, namely multi-tape automata, but even then we wish to consider them as acceptors rather than transducers [29]. This supports our aim that predicates should be expressible only once even though they can be used in a variety of ways.

Viewed as a string database query processing tool, our approach amounts to first defining a string handling sublanguage called *Alignment Declarations* with a logical semantics, and then providing a means for *executing* these declarations from within expressions of the host query language.

Let us first see an example of string comparison within a database query.

Example 1 Assuming an SQL-like query language [1, Chapter 7.1], the query "fetch from table R all strings whose reversals are also in the same table" could be conceptually written as

```
SELECT y
FROM   R x,
       R y
WHERE  rev(x, y)
```

where 'rev' is a two-place predicate which is true exactly when strings x and y are reversals of each other. This predicate is in turn definable as an alignment declaration given in Example 3 below.

In Alignment Calculus [8, Section 2], our string-oriented variant of the relational calculus [1, Chapter 5.3], this query could be expressed as $y\colon R(y) \wedge \exists x.R(x) \wedge \mathrm{rev}_l(x, y)$, where '$\mathrm{rev}_l$' is the logical definition of the predicate written as a String Formula.

Example 1 suggests a variant that requests the reversals and not the original strings from the database. This query is an example of string generation, and it can be expressed as follows.

Example 2 The query "fetch the reversals of the strings stored in table R" could conceptually be written as

```
SELECT y
FROM   R x
WHERE  rev(x, y)
```

where 'rev' is as in Example 1. Now the generated variable y gets its values from the operation of string reversal instead of a database table.

The corresponding Alignment Calculus formula is $y\colon \exists x.R(x) \wedge \mathrm{rev}_l(x, y)$.

Example 2 assumes seamless integration of our Alignment Declarations into the query language: our declaration 'rev' is directly visible as part of the query, and communication between it and the overall query mechanism happens directly via the logical variables x and y of the query language. Unfortunately, current database systems are not in general easily extensible in this sense. In an extensible system it would, of course, be possible to program a subroutine that behaves like our 'rev' predicate, compile it and make it available on the query language level, but this is not the way we wish to proceed. Instead, we want to have a system where the user can introduce any kind of predicates written in our language into the current database management system without having to, say, halt the system in order to link in yet another precompiled library in a system-dependent way.

Since we want our extensions to support as many systems as possible, we will eventually settle on a design which assumes less support from the underlying system. Briefly, our supported query language must offer the ability to write and call *user-defined functions capable of returning relations* and not just scalar values [22]. (However, the latter requirement can be somewhat relaxed as shown in Section 5.3.) This property has indeed been advocated for future relational database management systems [30].

3. Alignment Declarations

In this section we introduce the language of Alignment Declarations. The language is based on Alignment Calculus [8] which is defined formally in Section 4. In the spirit of sequence alignment our language uses operators for comparing and moving strings allowing pairwise character matching between multiple strings. Related ideas of describing molecular biological alignments by means of formal languages have been proposed [31,32]. In effect, we provide a domain-specific embedded language for string processing whose primary motivation stems from computational molecular biology.

Basically, the Alignment Declaration language consists of *on-statements* which are character comparisons joined together with modalities for moving strings forward and backward, and structures for concatenating, repeating and nondeterministically choosing between alternative on-statements. An on-statement consists of the word `on` followed by a *condition part* and possibly preceded by a *scan part*. A condition part consists of Boolean combinations of character comparisons in the form `x='A'`, `x=y`, `x=[` and `x=]`. Respectively, these comparisons mean that the *current character* on a string denoted by variable `x` is '`A`', the same as the current character on a string denoted by `y`, the left end-marker or the right end-marker. Here the "current character" means the character that is under inspection on a particular string, initially the left end-marker denoting the beginning of the string, or actually the position just before the beginning of the string.

A scan part consists of either a `scan` or a `rightscan` followed by a list of string variables, meaning that in the condition part the focus is on the next (or previous) characters with respect to the current characters on those strings. So these can be thought of as modalities or operations moving a "pointer" back and forth on a string (similarly to the operators "next" and "previous" in some temporal logics [26]). Scanning beyond an end-marker of a string is allowed but ineffective; the pointer will not go further beyond either end. Intuitively, an `on`-statement holds if and only if the condition part evaluates to true when the variable names have been substituted with their current characters after adjusting the pointers on those strings that are mentioned in a possible scan part. For example, `scan x on x='A'` holds if and only if the next (with respect to the current) character on the string denoted by variable x is '`A`'.

These `on`-statements are expressions in the Alignment Declaration language and other expressions can be defined recursively. Assuming that Φ_1 and Φ_2 are expressions, (i) Φ_1 Φ_2 is an expression, (ii) `repeat * times` Φ_1 `end` is an expression and (iii) `choose` Φ_1 `|` Φ_2 `end` is an expression. There are additional expressions defined for convenience but the ones above are the basic structures that all other expressions can be reduced to. Expressions of the Alignment Declaration language are called *alignment declarations*.

The semantics can intuitively be understood in the following fashion: (i) Φ_1 Φ_2 holds if and only if Φ_1 holds and Φ_2 holds when evaluated starting from the same pointer positions at which the evaluation of Φ_1 ended, (ii) `repeat * times` Φ_1 `end` holds if and only if an n-time concatenation of Φ_1 holds for some value of $n \in \mathbb{N}$ and (iii) `choose` Φ_1 `|` Φ_2 `end` holds if and only if Φ_1 holds or Φ_2 holds.

The Alignment Declaration language can be used to define database predicates describing string properties as shown in the example below. In the following, we will denote string variables by typewriter letters (`x`, `y`, ...) and strings by italics ($x, y, \ldots$). Furthermore, if not stated otherwise we assume that variable `x` denotes string x, variable `y` denotes string y and so on.

Example 3 Consider the predicate $\mathrm{rev}(x, y)$ discussed in Example 1 and think about how one could define a binary relation saying that two strings are reversals of each other. One would probably want to say something like this: Two strings are reversals of each other if you can start from the beginning of one string and the end of the other and then go to the opposite ends checking at each step that the characters on both strings match. The following expression tries to capture this intuition:

```
rev(x,y)
   repeat * times
      scan x on
   end
   scan x on x=]
   repeat * times
      rightscan x on
      scan y on x=y
   end
   rightscan x on x=[
   scan y on y=]
```

Recall that the evaluation starts from the beginning of each string. For this reason, the first `repeat`-loop first proceeds to the end of string x regardless of the content of the string. The repeat-expression itself holds, no matter how many times the loop is repeated but the `scan`-statement following it will be true only if the loop has been repeated at least $|x|$ times. (Here and below $|x|$ denotes the length of string x.) Then in the next `repeat`-expression we go back on string x and forward on string y verifying at each step that their characters are equal. This loop can again be repeated zero or more times but in order to be able to continue after the loop, we must go all the way to the beginning of x. At this point we can say that y starts with the characters of x in reversed order but to be able to say that y really *is equal* to the reversal of x, we must still go to the last line and verify that we have reached the right end of y. So this alignment declaration, called $\mathrm{rev}(x, y)$, evaluates to true on strings x and y if and only if x and y are reversals of each other.

In addition to the basic structures given above, we have defined some extra reserved words for compressing the expressions, such as the instructions `scan*`, `rightscan*`, `reset`, `read` and `rightread`. The commands `scan*` and `rightscan*` can be used to nondeterministically read the string in either direction. For example, `repeat * times scan x on x='A' or x='B' end` can be written as `scan* x on x='A' or x='B'`. The `reset`-statement can be used to move back to the beginning position of a string. The instructions `read` and `rightread` can be used as abbreviations of multiple simple `scan` or `rightscan` statements. For example, `scan x on x='A' scan x on x='A' scan x on x='B'` can be written as `read x on 'AAB'`.

We will now give some more examples of properties of strings that can be expressed using the Alignment Declarations. Let s_1, s_2 and s_3 be strings denoted by string variables `x`, `y` and `z`, respectively.

Example 4 String s_1 is a prefix of string s_2:

```
prefix(x,y)
   scan* x,y on x=y and not x=]
   scan x on x=]
```

Intuitively, this predicate holds if it is possible to proceed on both strings s_1 and s_2 so that all the characters on s_1 and s_2 are equal until the end of string s_1.

Example 5 String s_1 is a substring of string s_2:

```
substring(x,y)
   scan* y on not y=]
   scan* x,y on x=y
   scan x on x=]
```

This predicate holds if it is possible to first proceed on string s_2 to such a position that it is then possible to proceed on both strings s_1 and s_2 so that all the characters on s_1 and s_2 are equal until the end of string s_1. The difference between this and the previous example is that string s_2 is first scanned to the beginning of the part where scanning of both s_1 and s_2 starts. Because of the declarative nature of the Alignment Declarations, it is not necessary to specify the exact point where string s_1 starts to move along with string s_2. It is enough to simply say that s_2 should be scanned first to a suitable position, whatever it might be, and then continue with both strings. It is the job of the string query evaluation engine to find the right position.

Example 6 String s_1 is a suffix of string s_2:

```
suffix(x,y)
   scan* x,y on
   scan x,y on x=] and y=]
   rightscan* x,y on x=y
   rightscan x on x=[
```

Here we proceed on strings s_1 and s_2 until they both have reached the right end-marker. Then we proceed symmetrically with respect to the prefix example above, this time moving in the opposite direction.

Example 7 String s_1 is a subsequence of string s_2:

```
subsequence(x,y)
   repeat * times
      choose
         scan x,y on x=y and not y=] |
         scan y on not y=]
      end
   end
   scan x on x=]
```

In this example, there is an iteration where each step consists of proceeding either on both s_1 and s_2 in case the characters to be read are equal, or only on s_2. This continues until the end of string s_1. This can be thought of as constantly proceeding on s_2 and picking up a character from s_1 only when it is equal to the character on s_2 in the current position. Proceeding this way, if all the characters of s_1 can be found in s_2, we say that string s_1 is a subsequence of string s_2, and the predicate holds.

Example 8 String s_1 is a shuffle of strings s_2 and s_3:

```
shuffle(x,y,z)
    repeat * times
        choose
            scan x,y on x=y and not x=] |
            scan x,z on x=z and not x=]
        end
    end
    scan x,y,z on x=] and y=] and z=]
```

This example is similar to the previous one, except that we can now proceed only when either the characters on s_1 and s_2 are equal or the characters on s_1 and s_3 are equal. Furthermore, all the strings should reach the end simultaneously.

Although the language does not include numeric values explicitly, it is possible to use extra variables for counting, like in the two following examples. In the first one, the numeric value is used to count the length of a certain (overlapping) part in two strings. The second one presents an example of defining new similarity measures that can be used to retrieve strings that are in some sense close to each other.

Example 9 The suffix of string s_1 equals the prefix of string s_2 in at least $|s_3|$ character positions:

```
overlap(x,y,z)
    scan* x on not x=]
    scan* x,y,z on x=y
    scan x,z on x=] and z=]
```

On each matching step on strings s_1 and s_2, string s_3 is also moved forward. If s_3 comes to an end, the overlapping part is long enough.

Example 10 The edit distance [25, Chapter 11.2] of strings s_1 and s_2 is less than or equal to $|s_3|$:

```
edit_distance(x,y,z)
    repeat * times
        choose
            scan x,y on x=y |
            scan x,y,z on not z=] |
            scan x,z on not z=] |
            scan y,z on not z=]
        end
    end
    scan x,y on x=] and y=]
```

In this example, we either proceed on strings s_1 and s_2 when their characters match or we choose one of the transformations (replacement, insertion or deletion) and proceed on string s_3 which functions as an error counter. In an evolutionary setting, the edit distance can be seen as a measure of the number of point mutations between biological sequences. Other distance or similarity measures used in molecular biology can also be defined.

By allowing two copies of the same string, it is possible to express properties beyond regular expressions, like in the two following examples of which the first one is context-free and the second one is context-sensitive but not context-free. Such combinatorial dependencies have arisen for instance in the theory of gene regulation [2] and in RNA secondary structure prediction [3].

Example 11 The task is to select from the database table `R` those strings s_1 that are palindromes. We reuse the reversal predicate `rev` from Example 3 to simply require that string s_1 is the reversal of itself:

```
SELECT x
FROM   R x
WHERE  rev(x,x)
```

This is how we can enforce at the query language level that we have two pointers to one string.

Example 12 String s_1 is of the form ww for some string w:

```
copy(x,y)
   repeat * times
      scan x,y on
      scan x on not x=]
   end
   scan x,y on x=]
   rightscan* x,y on x=y and not y=[
   rightscan y on y=[
```

Using the query language as in Example 11 above we can enforce s_1 and s_2 to be the same string; let n be its length. If n is odd then the `repeat`-loop fails. Otherwise, we have proceeded onto the last (n^{th}) character of s_1 and onto the $(n/2)^{\text{th}}$ character of s_2. Then it only remains to read s_1 and s_2 backwards to check that these $(n/2)$ characters are indeed equal.

Generally, to be able to use these kinds of predicates not only for extracting matching entries from databases but also for producing new strings one needs to introduce an *alphabet* specifying the set of characters that can occur within a string. In our system, we assume that the basic alphabet is the 8-bit ASCII character set. The user can define and name different subsets of the ASCII alphabet using an `alphabet`-expression and then restrict the allowed character values of different strings by introducing a `keep`-expression to surround the relevant parts of the declaration.

Consider adding these elements to the example above: an alphabet named `AB_alp` that consists of, say, two characters '`A`' and '`B`' could be defined as `alphabet AB_alp:'A','B'` which would be put at the beginning of the alignment declaration. After that one can wrap the whole declaration following the predicate name or parts of it inside a `keep`-expression: `keep x,y in AB_alp` ... `end`. This will then define a scope inside which the strings referred to by `x` and `y` consist of characters '`A`' and '`B`'. This is equivalent to adding to the condition part of each `on`-statement mentioning `x` an extra condition of the form `and (x='A' or x='B' or x=[ or x=])` and similarly for `y`.

Example 13 In this simple biologically motivated example, we use different alphabets for strings representing strands of DNA and RNA sequences. A DNA sequence consists of nucleotides of four different types: adenine, cytosine, guanine and thymine which are denoted by the letters ‘A’, ‘C’, ‘G’ and ‘T’ respectively. In RNA, thymine is replaced by uracil which is denoted by the letter ‘U’. Here string s_1 is a DNA string and s_2 is a transcribed RNA string meaning that it is equal to s_1 apart from the substitution of ‘U’s for ‘T’s:

```
alphabet DNA:'A','C','G','T'
alphabet RNA:'A','C','G','U'
transcription(x,y)
   keep x in DNA
      keep y in RNA
         scan* x,y on (x=y) or (x='T' and y='U')
         scan x,y on x=] and y=]
      end
   end
```

To allow the reuse of previously defined string predicates, a macro call facility has been added to the language. So a previously stored alignment declaration can be mentioned by name within the body of another declaration. A macro call can be thought of as replacing the actual call with the body of the called declaration.

Example 14 Suppose we have defined a predicate called substring(x, y) as in Example 5. Then we can use it to create another version of the suffix predicate of Example 6:

```
suffix2(x,y)
   call substring(x,y)
   scan y on y=]
```

Although our language is designed for defining truth-valued string predicates that can be used for querying, we have noticed it useful to have a means of adding extra annotations to the query result. If the strings are long, say with thousands of characters, it is not informative enough to find out that a certain feature or pattern can be found within a string. One would also like to know *where* it was found. For this purpose, we added the `mark`-command that writes either normal characters or an integer showing the current position of any string variable to an extra annotation string.

Example 15 Adding annotations to the alignment declaration of Example 5 yields the following:

```
substring_annotated(x,y)
   scan* y on not y=]
   scan x,y on x=y
   mark 'String 1 can be found between locations ',y
   scan* x,y on x=y
   scan x on x=]
   mark ' and ',y,' of string 2.'
```

As the query result we get for each pair of strings s_1 and s_2 an extra annotation string like "`String 1 can be found between locations 1578 and 1604 of string 2.`"

The string predicates defined in the examples above can be – directly or with small modifications – used in different application areas. A large number of the problems in computational molecular biology are variants of string matching (see e.g. [25]) so examples like 4, 5, 6 and 9 are often applicable in molecular biology. By using counter strings as in examples 9 and 10 to count the number of mismatches one can easily extend these predicates to approximate string matching as in the following example which combines elements of several of the previous examples.

Example 16 Here we consider pairs of DNA sequences where the reverse complement of s_1 approximately occurs in s_2. The maximum number of character mismatches allowed is given as the length of s_3. (We consider neither character insertions nor deletions in this example.)

```
alphabet DNA:'A','C','G','T'
reverse_complement_occurs_approximately(x,y,z)
   keep x,y in DNA
      scan* y on
      repeat * times
         scan x on
         choose
            rightscan y on not x=[ and (
               (x='A' and y='T') or (x='T' and y='A') or
               (x='G' and y='C') or (x='C' and y='G')) |
            rightscan y on
            scan z on not z=]
         end
      end
      scan x on x=]
   end
```

Also searching for certain patterns (or motifs), defined as regular expressions or Prosite patterns [25, Chapter 15.8] for example, is common in database searches in molecular biology and can be easily done in our system. As an example, consider the following simple pattern matching problem.

Example 17 Accept a protein sequence s_1 if it contains a Prosite pattern

`[AC]-x-V-x(4)-ED`

(s_1 contains a pattern which starts with an '`A`' (alanine) or a '`C`' (cysteine) and is followed by one arbitrary character (any amino acid), then a '`V`' (valine), four arbitrary characters and finally any other character except '`E`' (glutamic acid) or '`D`' (aspartic acid)).

```
alphabet protein:'A','R','N','D','C','Q','E','G','H','I','L',
        'K','M','F','P','S','T','W','Y','V'
pattern(x)
   keep x in protein
      scan* x on
      scan x on x='A' or x='C'
      scan x on
      scan x on x='V'
      repeat 4 times
         scan x on
      end
      scan x on not (x='E' or x='D' or x=])
   end
```

Palindromic and repeated structures (examples 11 and 12) are of great importance in molecular biology. For example, in the problem of determining the secondary structure of a transfer RNA molecule, one is interested in finding possible pairings of complementary nucleotides. These pairings can be nested, forming stems and loops in the molecule folding, or crossing, forming pseudoknots [3]. In the following example, we will look at the former situation where a single strand of RNA may fold up and base pair with itself.

Example 18 The tRNA sequence s_1 contains a part that could possibly form a stem-and-loop structure where the maximum length of the looping part (the unpaired bases at the turn of the strand) is given as the length of the third parameter and the minimum length of the stemming part (the part with complementary, matching nucleotides) is given as the length of the fourth parameter. As in Examples 11 and 12 above, we assume that s_2 is a copy of s_1.

```
alphabet RNA:'A','C','G','U'
stem_and_loop(x,y,loop,stem)
   keep x,y in RNA
      scan* x,y on not x=]
      scan* x,loop on not loop=]
      repeat * times
         scan x,stem on not stem=]
         rightscan y on (x='A' and y='U') or (x='U' and
            y='A') or (x='G' and y='C') or (x='C' and y='G')
      end
      scan stem on stem=]
   end
```

Let us finally present a database query that combines two Alignment Declarations.

Example 19 The query "fetch from the database table `R` those strings for which the table also contains an approximate infected form" can be written as follows:

```
SELECT y
```

```
FROM R y,
     R z
WHERE infected_form_of(x,y)
  AND edit_distance(x,z,'␣␣␣')
```

Here the allowed approximation is to be within edit distance three, and it uses the string predicate given in Example 10. The infection concept is the one explained in Section 2, and it can be expressed with the following string predicate:

```
infected_form_of(x,y)
   scan* x,y on x=y
   read x on 'GTAGTTGGAAGGGCTAATTCACTCCCAAAGAAGACAAGAT'
   read y on 'GTAGT'
   scan* x,y on x=y
   scan x,y on x=] and y=]
```

Example 19 shows how the support for string comparison and generation within the database query language enables the user to express complex interrelationships within his data. Or in database terms, now the user can express complex join conditions [1, Chapter 4.4] on the contents of the string columns of his database tables, perhaps involving generated strings as intermediaries. Were this support outside of the database management system, the user would have to first transfer the strings in his tables into this outside string processing tool, then perform the desired processing there, and finally translate the results back into the database. All these three tasks might be cumbersome for a given query. Moreover, Example 19 shows a query where the last of these tasks seems redundant to the user, since he is not asking for strings to be generated, but instead selects existing strings according to a criterion that involves intermediate generated strings which can be discarded after query evaluation.

4. Formal Definitions

This section gives the formal definitions of the Alignment Declaration language. A reader uninterested in technicalities can skip this section without loss of understanding of the general properties of the language. We will start with the formal syntax and semantics of the Alignment Calculus that our language builds on. We will then move on to the syntax and semantics of Alignment Declarations by first introducing the core part of the language and then extending it with the additional features yielding the full Alignment Declaration language.

4.1. Formal Syntax and Semantics of Alignment Calculus

We present here for completeness a brief formal description of Alignment Calculus, our logic for string relations. The description given here is a simplified but equivalent variant of our earlier exposition [8, Section 2].

Let $\mathcal{V} = \{x_i : i \in \mathbb{N}\}$ be the set of *variable symbols*; we denote below their occurrences in formulae with 'ν' (possibly subscripted). Let in addition $\mathcal{R} = \{R_i : i \in \mathbb{N}\}$ be the set of *relation symbols*; each relation symbol $R \in \mathcal{R}$ also has an associated *arity* $\mathrm{ar}(R) \in \mathbb{N}$. Then the context-free grammar [28, Chapter 4.2] is in Figure 2.

$$
\begin{array}{lllllllllllll}
S & \longrightarrow & A \wedge S & \mid & A \\
A & \longrightarrow & \neg A & \mid & \exists \nu A & \mid & R(\nu_1, \ldots, \nu_{\mathrm{ar}(R)}) \text{ for } R \in \mathcal{R} & \mid & (S) & \mid & B \\
B & \longrightarrow & C + B & \mid & C \\
C & \longrightarrow & D.C & \mid & D \\
D & \longrightarrow & D^* & \mid & (B) & \mid & E \\
E & \longrightarrow & [\nu]_l & \mid & [\nu]_r & \mid & \nu = \mathtt{[} & \mid & \nu = \mathtt{]} & \mid & \nu = c \text{ for } c \in \Sigma
\end{array}
$$

Figure 2. The grammar for Alignment Calculus

Alignment Calculus has two layers: the outer layer is syntactically derived from the start symbol S and is semantically the relational calculus [1, Chapter 5.3] of string relations, as will be explained in Section 4.1.1. That is, this layer treats strings as indivisible atomic values. Hence it is an abstraction of those query facilities that already exist in databases without any string-specific operations. (We omit operations on non-string types for brevity.)

On the other hand, the inner layer is derived from the nonterminal B and is our sub-language called the *string formulae* for defining string relations by character comparisons, as will be explained in Section 4.1.2. That is, this layer opens the atomic strings of the outer layer into sequences of characters that can be manipulated at will. Hence it is an abstraction of our proposed string handling extensions to existing query facilities.

4.1.1. Semantics of the Relational Layer

An *interpretation* $\mathfrak{M}$ maps every relation symbol $R \in \mathcal{R}$ into some subset of $(\Sigma^*)^{\mathrm{ar}(R)}$. An *assignment* $\theta\colon \mathcal{V} \mapsto \Sigma^*$ maps in turn every variable symbol into the corresponding string. Then the truth relation "the formula ϕ is true in the interpretation $\mathfrak{M}$ and the assignment θ" is customarily denoted as $\mathfrak{M} \models \phi\,\theta$ and defined by induction on the structure of ϕ:

1. $\mathfrak{M} \models \psi \wedge \chi\,\theta$ holds if and only if both $\mathfrak{M} \models \psi\,\theta$ and $\mathfrak{M} \models \chi\,\theta$ hold.

2. $\mathfrak{M} \models \neg\psi\,\theta$ holds if and only if $\mathfrak{M} \models \psi\,\theta$ does not hold.

3. $\mathfrak{M} \models \exists\nu\psi\,\theta$ holds if and only if $\mathfrak{M} \models \psi\,\theta[\nu \leftarrow w]$ holds for some $w \in \Sigma^*$, where the corresponding assignment function is defined as

$$
\theta[x_i \leftarrow w](x_j) = \begin{cases} w & \text{if } i = j \text{ and} \\ \theta(x_j) & \text{otherwise} \end{cases} \tag{1}
$$

   and it modifies θ to map x_i into w.

4. $\mathfrak{M} \models R(\nu_1, \ldots, \nu_{\mathrm{ar}(R)})\,\theta$ holds if and only if $\left\langle \theta(\nu_1), \ldots, \theta(\nu_{\mathrm{ar}(R)}) \right\rangle \in \mathfrak{M}(R)$ holds.

Once fully defined in Section 4.1.2, Alignment Calculus is considered a string database query language as follows. The string database schema $\mathcal{S}$ identifies string tables $T_1, \ldots, T_k$ and names them uniquely with relation symbols $R_1, \ldots, R_k \in \mathcal{R}$ so that the number of columns in table T_i equals $\mathrm{ar}(R_i)$ for each $1 \leq i \leq k$. An instance $\mathcal{I}$ of schema $\mathcal{S}$ leads to an interpretation $\mathfrak{M}_{\mathcal{I}}$ which maps the relation symbol R_i into the current contents of the table T_i for each $1 \leq i \leq k$. Then an Alignment Calculus formula ϕ can be considered a query to $\mathcal{I}$ provided that all its relation symbols are among the $R_1, \ldots, R_k$. If the free variables of query ϕ – defined as usual based on Case (3) of the truth definition above – are $x_1, \ldots, x_m$, then the answer is the relation $\{\langle \theta(x_1), \ldots, \theta(x_m) \rangle : \mathfrak{M}_{\mathcal{I}} \models \phi\,\theta\} \subseteq (\Sigma^*)^m$.

4.1.2. Semantics of the String Layer

String formulae can be considered to be regular expressions [28, Chap. 2.5] over the alphabet E of Fig. 2. The truth of a *formula word* $\Phi \in E^*$ generated by a string formula is defined with respect to an assignment θ and an *alignment* $\eta \colon \mathcal{V} \mapsto \mathbb{N}$; it is denoted $\theta \models \Phi\,\eta$ to emphasize that now θ is the immutable interpretation while η is modified. Intuitively variable $x_i \in \mathcal{V}$ is currently pointing to the $\eta(x_i)^{\text{th}}$ position within the string $\theta(x_i)$; or equivalently, that the strings $\theta(x_0), \theta(x_1), \theta(x_2), \ldots$ are now aligned to each other so that their positions $\eta(x_0), \eta(x_1), \eta(x_2), \ldots$ coincide. We continue our truth definition begun in Section 4.1.1 as follows:

5. $\mathfrak{M} \models \phi\,\theta$ holds for a string formula ϕ if and only if ϕ generates some formula word Φ which satisfies $\theta \models \Phi\,\eta_0$ for the *initial* alignment η_0; this is defined to yield $\eta_0(x_i) = 0$ for every $x_i \in \mathcal{V}$.

   Here we move from the outer layer which treats strings as indivisible entities into the inner layer which introduces access to individual characters via alignments.

6. $\theta \models \Lambda\,\eta$ holds for the empty formula word $\Lambda \in E^*$ for all assignments θ and alignments η.

   This is the terminal case of the inductive definition below.

7. $\theta \models x_i = [\,\Psi\,\eta$ holds if and only if $\eta(x_i) = 0$ holds and furthermore $\theta \models \Psi\,\eta$ holds inductively.

   In particular, $\theta \models x_i = [\,\eta_0$ holds for each $x_i \in \mathcal{V}$ and assignment θ as it means that x_i is now pointing to just before the left end of its string.

   Moreover, the inductive requirement $\theta \models \Psi\,\eta$ means that the further restrictions imposed by the formula word Ψ must also be met.

8. $\theta \models x_i =]\,\Psi\,\eta$ holds if and only if $\eta(x_i) = |\theta(x_i)| + 1$ holds and furthermore $\theta \models \Psi\,\eta$ holds inductively.

   Similarly to Case (7) above this means that x_i is now pointing to just after the right end of its string. These are the two boundary cases for the position within string $\theta(x_i)$.

9. $\theta \models x_i = c\,\Psi\,\eta$ for $c \in \Sigma$ holds if and only if $1 \leq \eta(x_i) \leq |\theta(x_i)|$ holds, the $\eta(x_i)$th character of $\theta(x_i)$ equals c, and furthermore $\theta \models \Psi\,\eta$ holds inductively.

10. $\theta \models [x_i]_l \ \Psi \, \eta$ holds if and only if $\theta \models \Psi \, \eta[x_i \leftarrow \min(\eta(x_i), |\theta(x_i)|) + 1]$ holds inductively; Eq. (1) carries directly over from assignments to alignments.

    Hence η is modified so that from now on x_i points to the next character of its string, if any. In view of Case (5) above this can be seen as reading the string in the forward direction.

11. $\theta \models [x_i]_r \ \Psi \, \eta$ holds if and only if $\theta \models \Psi \, \eta[x_i \leftarrow \max(\eta(x_i), 1) - 1]$ holds inductively.

    Hence $\eta(x_i)$ is modified into the opposite direction to the Case (10) above.

4.2. Formal Syntax of Alignment Declarations

Here we give the formal definition of the syntax of the Alignment Declarations. We will first define the core part of our language and show how the expressions in this core language are translated to expressions in Alignment Calculus. Then we proceed by defining the full Alignment Declaration language and show how the additional expressions are constructed from the expressions in the core language. Once all the expressions of the language are reduced to expressions of Alignment Calculus, the semantics of the language is obtained from the semantics of Alignment Calculus given above.

4.2.1. The Core Language

As before, let $\mathcal{V} = \{x_i : i \in \mathbb{N}_+\}$ be the set of variable symbols. The core language of the Alignment Declarations consists of the following components (square brackets '[' and ']' denote optional parts):

`on`-statements:	`on` $[\phi]$
`scan`-statements:	`scan` $x_1, \ldots, x_n$ `on` $[\phi]$
`rightscan`-statements:	`rightscan` $x_1, \ldots, x_n$ `on` $[\phi]$
successive statements:	$\Phi\Psi$
`repeat`-statements:	`repeat * times` Φ `end`
`choose`-statements:	`choose` Φ `|` Ψ `end`

where ϕ denotes a formula consisting of Boolean combinations (`and`, `or`, `not`) of character comparisons of the form x_i`=`x_j, x_i`='`v`'` (where $v \in \Sigma$), x_i`=]` and x_i`=[`, and Φ and Ψ denote arbitrary statements of the Alignment Declaration language.

For every expression in the core language a corresponding formula in Alignment Calculus can be obtained by applying the following transformation function *Tr* from the set of Alignment Declaration expressions to Alignment Calculus expressions:

1. `on` ϕ is transformed into a formula $Tr(\phi)$ using the following rules for ϕ. We let $x, y \in \mathcal{V}$ and assume that ϕ is in a form where the negations are pushed inside formulae using De Morgan's rules `not` $(\phi_1$ `and` $\phi_2) \equiv$ `not` ϕ_1 `or not` ϕ_2 and `not` $(\phi_1$ `or` $\phi_2) \equiv$ `not` ϕ_1 `and not` ϕ_2 and also the rule `not not` $\phi_1 \equiv \phi_1$. Symbol λ stands for the case where there is no condition ϕ in an `on`-statement.

    (a) $Tr(\lambda) \equiv x = x$.

    (b) $Tr(x$`=[`$) \equiv x = \texttt{[}$.

    (c) $Tr(x$`=]`$) \equiv x = \texttt{]}$.

(d) $Tr(x=\texttt{'}v\texttt{'}) \equiv x = v$, where $v \in \Sigma$.

(e) $Tr(x\texttt{=}y) \equiv x = y$.

(f) $Tr(\phi_1 \texttt{ and } \phi_2) \equiv Tr(\phi_1).Tr(\phi_2)$.

(g) $Tr(\phi_1 \texttt{ or } \phi_2) \equiv Tr(\phi_1) + Tr(\phi_2)$.

(h) $Tr(\texttt{not } x\texttt{=[}) \equiv x = v_1 + x = v_2 + \ldots + x = v_n$ for all $v_i \in \Sigma \cup \{\texttt{]}\}$.

(i) $Tr(\texttt{not } x\texttt{=]}) \equiv x = v_1 + x = v_2 + \ldots + x = v_n$ for all $v_i \in \Sigma \cup \{\texttt{[}\}$.

(j) $Tr(\texttt{not } x=\texttt{'}v\texttt{'}) \equiv x = v_1 + x = v_2 + \ldots + x = v_n$ for all $v_i \in \Sigma \setminus \{v\} \cup \{\texttt{[},\texttt{]}\}$.

(k) $Tr(\texttt{not } x\texttt{=}y) \equiv x = v_1.y = u_1 + \ldots + x = v_n.y = u_m$ for all pairs v_i, u_j where $v_i \neq u_j$ and $v_i, u_j \in \Sigma \cup \{\texttt{[},\texttt{]}\}$.

2. $Tr(\texttt{scan } x_1, x_2 \ldots, x_n \texttt{ on } \phi) \equiv [x_1]_l.[x_2]_l.\ldots[x_n]_l.Tr(\phi)$.

3. $Tr(\texttt{rightscan } x_1, x_2 \ldots, x_n \texttt{ on } \phi) \equiv [x_1]_r.[x_2]_r.\ldots[x_n]_r.Tr(\phi)$.

4. $Tr(\Phi\Psi) \equiv Tr(\Phi).Tr(\Psi)$.

5. $Tr(\texttt{repeat * times } \Phi \texttt{ end}) \equiv (Tr(\Phi))^*$.

6. $Tr(\texttt{choose } \Phi \texttt{ | } \Psi \texttt{ end}) \equiv (Tr(\Phi) + Tr(\Psi))$.

4.2.2. The Full Language

The full language of Alignment Declarations consists of the above core language and the additional expressions which can be defined in terms of the expressions of the core language. Let $x, x_1, x_2 \ldots$ be variable symbols. We use square brackets '[' and ']' to denote optional syntactical parts and usual brackets '(' and ')' together with the Kleene star '*' to denote syntactical parts which can be repeated zero or more times.

- `alphabet` Anm `:`C `(,`C`)`*, where C is either of the form `'`v_1`'` or `'`v_1`'-'`v_2`'`, defines an alphabet. Anm is a string denoting the name of the alphabet and v_i give the ASCII characters (perhaps as ranges) that the alphabet consists of. In the following, we use Alp to refer to either such a named alphabet or such a list of characters.

- x `in` Alp is a shorthand for a disjunction `(`x `='`v_1`' or` x `='`v_2`' or ... or` x `='`v_n`' or` x `=[ or` x `=])` for all characters $v_i \in Alp$.

- `keep` x `in` Alp Φ `end` is a shorthand for adding [`and`] x `in` Alp to those `on`-statements that mention variable x and belong to the expanded form of Φ which contains only core language and `mark`-expressions.

- `scan*` $x_1, \ldots, x_n$ `on` $[\phi]$ is a shorthand for `repeat * times scan` $x_1, \ldots, x_n$ `on` $[\phi]$ `end`.

- `rightscan*` $x_1, \ldots, x_n$ `on` $[\phi]$ is a shorthand for `repeat * times rightscan` $x_1, \ldots, x_n$ `on` $[\phi]$ `end`.

- `repeat + times` Φ `end` is a shorthand for Φ `repeat * times` Φ `end`.

- `repeat` k `times` Φ `end` where $k \in \mathbb{N}_+$ is a shorthand for successive statements consisting of exactly k repetitions of Φ.
- `choose` Φ_1 `|` Φ_2 `|` $\ldots\Phi_n$ `end` is a shorthand for the following kind of a construct:

```
choose Φ_1  |
      choose Φ_2  |
            ⋮
                  choose Φ_{n-1}  | Φ_n  end
            ⋮
      end
end.
```

- `reset` x is a shorthand for `rightscan*` x `on on` x`=[`.
- `read` x `on` `'`$v_1v_2\ldots v_n$`'` is a shorthand for
`scan` x `on` x`='`v_1`'` `scan` x `on` x`='`v_2`'`... `scan` x `on` x`='`v_n`'`.
- `rightread` x `on` `'`$v_1v_2\ldots v_n$`'` is a shorthand for
`rightscan` x `on` x`='`v_1`'` `rightscan` x `on` x`='`v_2`'`... `rightscan` x `on` x`='`v_n`'`.
- `call` *macro_name*$(x_1, x_2, \ldots, x_n)$ is a shorthand for repeating the body of an alignment declaration introduced and named earlier. The variables of the declaration named "*macro_name*" are replaced with the variables $x_1, x_2, \ldots, x_n$. The number of variables in the macro call must match the arity of the declaration called.

  A variable name can not occur more than once in one macro call: this restriction ensures that the aforementioned variable replacement remains unambiguous.
- `mark` *Marking*(`,`*Marking*)* where *Marking* is either a string constant (in single quotes) or a string variable will write to a separate marking tape (denoted by x_0) the string given by the user or, in the case of *Marking* being a string variable name, a number representing the current position on the corresponding string.

  In practice, this number is written in decimal notation by a special operation. In theory it could be written in unary notation within the core language itself as follows, where `x` is the string variable in question:

```
scan x_0 on x_0='A'
repeat * times
   scan x_0 on x_0='B' and not x=[
   rightscan x on
end
on x=[
scan x_0 on x_0='A'
```

```
repeat * times
   rightscan x₀ on x₀='B'
   scan x on
end
rightscan x₀ on x₀='A'
repeat * times
   scan x₀ on x₀='B'
end
scan x₀ on x₀='A'
```

The idea in this expression is that the current position of string x, say n, is written to the annotation tape x_0 in unary as $\texttt{AB}^n\texttt{A}$ where ‘A’ and ‘B’ are two distinct characters reserved for this purpose.

A `mark`-command with multiple parameters in the fashion of `mark` $Marking_1$, $Marking_2$, . . . , $Marking_n$ is a shorthand for repeating n successive `mark`-commands in the following fashion: `mark` $Marking_1$ `mark` $Marking_2$ . . . `mark` $Marking_n$.

The complete syntax for the Alignment Declaration language is given as Appendix A.

5. System Architecture

Recall that our task was to extend ordinary database query languages in a way that makes it possible for the user to formulate queries that are based on structural properties of strings. These queries include ones that select strings with certain properties, compare different strings and generate new strings by modifying the ones in the database. Our proposal was to achieve this functionality by allowing the user to define his own string comparison criteria by first expressing these criteria using our string handling sublanguage, Alignment Declaration language, saving this expression as a string predicate, and then employing this string predicate in queries. We will now take a look at how this can be achieved in practice.

Our design is based on two previously implemented prototypes [21,18] which extended the Postgres and O_2 database management systems. The system described here is implemented as an extension to the MySQL database management system. The main components of our system are shown in Figure 3 which gives an overview of our design as a Data Flow Diagram (DFD) in the Object Modeling Technique (OMT) software design formalism [33, Chapter 6].

In the diagram, the user has two distinct roles, **Declarer** and **Querent**, depending on the task at hand. As the Declarer, the user can define a new string predicate by first writing it in the Alignment Declaration language introduced in Section 3 and calling a user-defined function which will compile the expression into a flowchart-type form which can be later executed by another user-defined function. This compiled form is stored in the special data store named **Alignment executables** from where it can be retrieved by name during query processing. This storage of previously compiled executables is also used during the compilation of new ones to support the macro facility of the language.

An alignment declaration may refer to not only local but also global alphabet definitions. These global definitions are just strings of allowed characters, and the Declarer can store

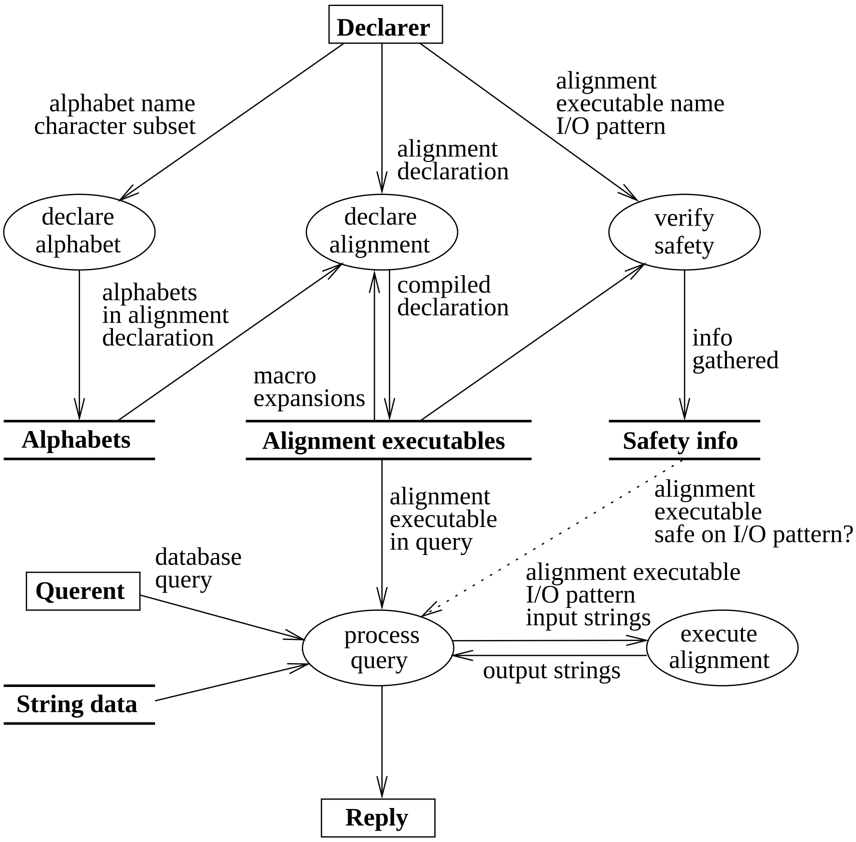

Figure 3. Design of the implementation [22, Figure 2].

them beforehand by name into a special data store named **Alphabets** to make them available to compilation, similarly to macros above.

As the Querent, the user can issue database queries which refer to existing string predicates. This is done by embedding into an SQL query a call for a user-defined function which is able to evaluate such predicates. Processing such a query involves retrieving the corresponding executables and submitting them to execution along with the input strings which they are called upon to process. This execution can also generate new output strings that are to become part of the query answer.

The generation of new strings raises a theoretical and practical problem of query safety: Is it safe to evaluate a query containing string processing operations, or is there a danger of looping endlessly, perhaps generating voluminous amounts of string output? Therefore one task for the Declarer is to determine these safety properties for those alignment declarations that he has written. The system assists in this by offering a verifier which is the third user-defined function in the system. This verifier tries to infer whether or not a given executable can be safely used in a given input-output situation that occurs in a query. These inference results can be stored in **Safety info** for later use. Such inference is costly and should therefore be done only once. In addition, such inference is necessarily incomplete, and therefore the user can augment **Safety info** with safety properties obtained by other means. Ideally, the results of safety analysis should be able to affect query processing, so that evaluating unsafe queries could be aborted. However, there is no implementation-independent way to achieve this; therefore we do not insist on it, and leave the corresponding arrow in Figure 3 dotted.

In the following, we will discuss the central technical issues involved in implementing this kind of a system. First we will focus on the compilation and evaluation of the string predicates in Section 5.1. Then we will concentrate on the problems involving safety in Section 5.2. In Section 5.3 we will finally look at the interplay between our extension and the database management system, in particular the MySQL system which we have most recently been working with.

5.1. Alignment Executables

After the user has defined a new string predicate using the Alignment Declaration language, the system compiles it into an executable form. This form is simply a numbered sequence of instructions, whose symbolic form and semantics are given in Table 1. This compilation is straightforward: The alignment declaration is first translated into a kind of nondeterministic automaton based on its similarity to regular expressions [28, Chapter 2.5]. Then this automaton is translated into an executable after a series of optimizing control and data flow analyses. Details are omitted for brevity. Note, however, that the declarative nature of the language imposes certain limits on the possible optimizations; for instance, not all search can be eliminated from the evaluation method, since the underlying automaton model is inherently nondeterministic by nature [34].

We must therefore fix some particular search or evaluation order for our compiled executables. We have chosen a *depth-first* order for efficiency, as in the logic programming language Prolog [35, §10]. Hence Table 1 refers to a stack-based execution model.

Example 20 Below is an alignment executable corresponding to the declaration in Example 3. Here we have restricted the alphabets of both strings x and y to consist of

Table 1
The alignment executable instruction set.

INSTRUCTION	ADVANCING SEMANTICS	RETREATING SEMANTICS
`GUESS` *addr*	Push *next* into the stack and set $next \leftarrow addr$.	Set $mode \leftarrow$ **true** and $next \leftarrow next + 1$.
`NEXT/PREV` x	Push *next* into the stack, and set $x \leftarrow x \pm 1$ and $next \leftarrow next + 1$.	Set $x \leftarrow x \mp 1$.
`IF` x`=[ THEN` *addr*	If x is now at the left end of its string, then set $next \leftarrow addr$; otherwise set $next \leftarrow next + 1$.	– (That is, this case cannot arise, since the advancing semantics did not push *next* into the stack.)
`IF` x`=c THEN` *addr*	If x is now at an unbound character, then bind it to `c`, push *next* into the stack and set $next \leftarrow addr$; otherwise if x is now at a character bound to `c`, then set $next \leftarrow addr$; otherwise set $next \leftarrow next + 1$.	Unbind again the character that x is now on, and set $mode \leftarrow$ **true** and $next \leftarrow next + 1$.
`IF` x`=] THEN` *addr*	As above for the terminating right end-marker `]`.	As above.
`ACCEPT`	– (Execution stops.)	–
`FAIL`	Set $mode \leftarrow$ **false**.	–
`GOTO` *addr*	Set $next \leftarrow addr$.	–

loop
  if *mode* **then**
    if *next* points to an `ACCEPT` instruction **then**
      exit "next solution found"
    else
      Execute from Table 1 the ADVANCING semantics of the instruction that *next* points to
    end if
  else if the *stack* is empty **then**
    exit "no more solutions to find"
  else
    Pop *next* from the *stack*;
    Execute from Table 1 the RETREATING semantics of the instruction that *next* points to
  end if
end loop

Figure 4. The alignment executable interpreter.

characters ‘A’ and ‘B’ only. When comparing this program with the alignment declaration in question, then lines 0–4 below correspond to the first `repeat`-loop and its following `scan`-statement of Example 3, and instructions on lines 5–17 will do the work of the rest of the declaration.

```
 0: NEXT x
 1: IF x=]   THEN 5
 2: IF x='A' THEN 0
 3: IF x='B' THEN 0
 4: FAIL
 5: PREV x
 6: NEXT y
 7: IF x='A' THEN 16
 8: IF x='B' THEN 14
 9: IF x=[   THEN 11
10: FAIL
11: IF y=]   THEN 13
12: FAIL
13: ACCEPT
14: IF y='B' THEN 5
15: FAIL
16: IF y='A' THEN 5
17: FAIL
```

The core of our interpreter is given in Figure 4. The Boolean variable *mode* indicates whether the search is currently advancing towards a possible solution or retreating after a failure. When *mode* is **true** the variable *next* points to the next instruction in the executable. The *stack* contains those past values of *next* at which the search faced a choice, so that when the search later retreats, the bad choice can be retracted and another tried instead. In fact, the interpreter can be seen as an implementation for an extremely simple variant of Prolog with only tail recursion and lists of ASCII characters.

Interpreting a given alignment executable starts at its initial instruction 0 in the advancing mode with an empty stack. Its variables contain their corresponding inputs between the left and right end-markers `[` and `]`, positioned to read the left end-marker `[`. If the algorithm in Figure 4 exits with a solution, then the given strings do indeed satisfy the predicate expressed by the alignment declaration that produced the executable. The reader can easily verify that for example strings $x =$ AB and $y =$ BA are accepted by the string reversal executable in Example 20, while for example strings $x =$ AB and $y =$ B are not.

If the value of some variable in the executable is initially unknown and should be constructed by the algorithm in Figure 4, then only the left end-marker `[` is initially present in the string for that variable, followed by as yet unbound character positions. These are to be bound incrementally from the left during the search without leaving unbound gaps. For instance the first loop in Example 3 disregards the characters in string x. On the other hand, the corresponding instructions 2 and 3 in Example 20 do specify explicit characters for all those disregarded positions. The alignment declaration compiler inserts such explicit instructions to prevent unbound gaps from ever appearing in the generated output; otherwise those gaps would have to be filled after the search succeeds, and this would imply separate algorithms for simulation with and without output generation. As it stands, however, both tasks can be carried out with the same algorithm in Figure 4. The reader can again easily verify that for example given string $x =$ AB this algorithm does find a solution with string $y =$ BA.

The same search problem may have several solutions, especially when strings are constructed. Each subsequent solution can be obtained by invoking the algorithm in Figure 4 with the same stack with which the previous successful search halted but in the retreating mode; this amounts to treating the successful final instruction `ACCEPT` as if it had been `FAIL` instead. This is similar to obtaining multiple answers one by one from a Prolog implementation.

5.2. Finiteness Analysis

Database theory contains the notion of a *domain independent query* [1, Definition 5.3.7] with the following intuition: In theory, the logically correct meaning of a database query ϕ is defined with respect to the infinite domain $\mathcal{U}$ of all possible data items. In practice, however, ϕ is evaluated against the current database which is naturally finite. Thus the answer of this evaluation can be trusted only if ϕ is independent of the size of the evaluation domain in the following sense: Given the current contents of the database, there always exists some finite $\mathcal{U}' \subset \mathcal{U}$ such that evaluating ϕ on domain $\mathcal{U}'$ yields the same result as evaluating it on domain $\mathcal{U}$. Hence $\mathcal{U}'$ is a sufficient finite approximation of $\mathcal{U}$.

In our string databases the proper infinite domain $\mathcal{U}$ would be Σ^*, while the candidates for the finite domains $\mathcal{U}'$ are $\Sigma^{\leq \ell} = \{w \in \Sigma^* \colon\ |w| \leq \ell\}$, the sets of strings with length at most $\ell \in \mathbb{N}$. Hence we call the string database query ϕ domain-independent if and only if there exists a function $\mathcal{W}_\phi \colon \mathbb{N} \mapsto \mathbb{N}$ with the following property: Evaluating ϕ on the truncated domain $\Sigma^{\leq \ell}$ yields the same result as evaluating it on the proper domain Σ^* if

$$\ell \geq \mathcal{W}_\phi(\max\{|v| : v \in \Sigma^* \text{ is in the current database}\})$$

[8, Definition 3.2]. In other words, if and only if for every possible database there exists an upper *limit* for the length of the strings that must be considered during evalution to obtain the logically correct result without any such limits.

Consider for instance the query in Example 2. It is domain independent in the above sense, because it is not necessary to consider longer strings than those currently stored in the database table R. In other words, $\mathcal{W}(n) = n$ is a limit function for this query. This can be inferred informally as follows. First, no string x needs to be longer than any in table R, or $\max\{|v| : v \text{ is in } R\}$. This limit accounts for the part "$R\ x$". And second, reversing such a string x does not increase the length of the resulting string y. This limit accounts for the part "$\mathrm{rev}(x, y)$". Combining these two limits leads to the overall limit.

Although determining domain independence is undecidable for the relational calculus [1, Corollary 6.3.2(b)], inferring *finiteness dependencies* [1, page 306] suffices in many cases. In Example 2, part "$R\ x$" satisfies the unconditional finiteness dependency "x has a finite domain". Similarly, part "$\mathrm{rev}(x, y)$" satisfies the conditional dependency "if x has a finite domain, then so does y". From these follows the desired dependency "y has a finite domain" via standard inference methods for functional dependencies.

Finiteness dependency inference therefore offers a method for determining the domain independence of the whole query if there is a method for determining if a given Alignment Declaration mentioned in the query (such as "rev" here) satisfies a given finiteness dependency (such as "if x has a finite domain, then so does y"). This latter problem can be rephrased as follows: if we consider some variables $\mathcal{I}$ (here just x) as input and the others $\mathcal{O}$ (here just y) as output, verify that there is no input for which the corresponding alignment executable $\mathcal{P}$ (here to "rev") can generate infinitely many different outputs.

Unfortunately this problem becomes undecidable as soon as more than one variable is moved in both directions [8, Theorem 5.1]. Furthermore, although the problem remains decidable with just one such variable [8, Theorem 5.2], the decision method is impractical. Hence we have developed the partial solution given in Figure 5. If this algorithm returns **true**, then the property is guaranteed to hold. On the other hand, **false** means merely

boolean function *limited*($\mathcal{P}$: the alignment executable to examine,
$\mathcal{I}$: the variables of $\mathcal{P}$ to consider as input)
return *halting*$(\mathcal{P}, \mathcal{O}) \wedge$ *looping*$(\mathcal{P}, \mathcal{I})$ where $\mathcal{O}$ consists of those variables of $\mathcal{P}$ that are not in $\mathcal{I}$, *halting* is given in Figure 6 and *looping* in Figure 7.
end *limited*

Figure 5. The finiteness checking algorithm.

boolean function *halting*($\mathcal{P}$: the alignment executable to examine,
$\mathcal{O}$: the variables of $\mathcal{P}$ to consider as output)

1: Let G be the *flow graph* of $\mathcal{P}$: that is, a node n_a for each address a and an edge $n_a \to n_b$ if the advancing semantics of the instruction at address a can store address b into the program counter *next* by Table 1.
2: **for all** $x_o \in \mathcal{O}$ **do**
3: Let G' be a copy of G, but without edges $n_a \to n_b$ where the instruction at address a is `IF` x_o`=] THEN` b (that is, without those edges that could add the right end-marker to the string being generated as x_o).
4: **if** G' has a path from node n_0 into some node n_a where the instruction at address a is `ACCEPT` (that is, a path from the start node into a solution node) **then**
5: **return false**
6: **end if**
7: **end for**
8: **return true**
end *halting*

Figure 6. The terminator checking algorithm.

that this particular algorithm could not guarantee it.

Consider for instance the call *limited*(the executable in Example 20, $\{x\}$) to the algorithm in Figure 5. That is, the algorithm is asked to verify that for no string x can "rev" accept more than a finite number of strings y.

The call to the subroutine *halting* in Figure 6 reduces to checking the flow graph G constructed on line 1 to see if node n_{13} can be reached from node n_0 without using the edge $n_{11} \to n_{13}$.

It cannot, so the subroutine call returns **true**. Thus it is guaranteed that the executable cannot `ACCEPT` without visiting the terminating right end-marker `]` of the output string y first.

In the call to the subroutine *looping* in Figure 7, an edge $n_a \to n_b$ is safe from deletion on line 10 if and only if it is a part of some (not necessarily simple) loop with equally many `PREV` x_i and `NEXT` x_i instructions; this follows from the properties of the closure

boolean function *looping*($\mathcal{P}$: the alignment executable to examine,
$\mathcal{I}$: the variables of $\mathcal{P}$ to consider as input)

Let G be the flow graph of $\mathcal{P}$ as in Figure 6.
if $\mathcal{I} = \emptyset$ **then**
  Delete from G each edge from one maximal strongly connected component into another.
else
  Let the working set W of variables be initially $\mathcal{I}$.
  while $W \neq \emptyset$ **do**
    Choose any arbitrary input variable $x_i \in W$;
    Let G_i be a copy of G where each unweighted edge $n_a \to n_b$ of G becomes a weighted edge $n_a \xrightarrow{s} n_b$ in G_i depending on the instruction at address a:

$$s = \begin{cases} +1 & \text{if it is } \texttt{NEXT } x_i \\ -1 & \text{if it is } \texttt{PREV } x_i \\ \pm 0 & \text{otherwise.} \end{cases}$$

    (That is, s is the effect of the instruction on x_i.)
    Let G_i^* be the closure of G_i with respect to the following edge addition rule: if both $n_a \xrightarrow{u} n_b$ and $n_b \xrightarrow{v} n_c$ are in G_i^* where $u \cdot v \neq +1$, then $n_a \xrightarrow{u+v} n_c$ must also be in G_i^*.
    Delete from G each edge $n_a \to n_b$ whose weighted counterpart $n_a \xrightarrow{s} n_b$ in G_i has no inverse $n_b \xrightarrow{-s} n_a$ in G_i^*.
    Let W be $\mathcal{I} \setminus \{x_i\}$ if some edge was deleted in the previous step, and $W \setminus \{x_i\}$ otherwise.
  end while
end if
return whether G lost all its edges or not.
end *looping*

Figure 7. The edge removing algorithm.

computed on line 9 [36, Theorem 5][37, Theorem 4]. No edge is safe in this example, so this subroutine call returns **true** as well. Thus it is guaranteed that the executable cannot shuffle its input x back and forth indefinitely.

Combining the results from the subroutines yields **true**: It is guaranteed that the interpreter in Figure 4 must eventually visit the end of output string y, and that this moment cannot be postponed indefinitely. This intuition can be formalized into a correctness proof for the algorithm in Figure 5 [29, Section 2][23, Section 4.2]; details are omitted for brevity.

On the other hand, calling *limited*(the executable in Example 20, $\{y\}$) returns **false**, even though the corresponding property does hold, because the edges corresponding to the weighted loop $n_0 \overset{0}{\to} n_1 \overset{0}{\to} n_2 \overset{0}{\to} n_0$ are safe from deletion.

It turns out that the algorithm in Figure 7 guarantees not only that the outputs generated by the interpreter in Figure 4 remain finite but also that the whole execution remains finite as well [23, Section 4.2]. That is, the algorithm in Figure 7 identifies a class of alignment executables for which the straightforward but generally incomplete depth-first search strategy in Figure 4 is in fact complete, in contrast to for example Prolog [35, §10]. (Indeed, similar reasoning and methods to ours have been employed in termination detection of logic programs [38].) As a result, the length limit functions $\mathcal{W}_\phi$ introduced at the beginning of this section are not required at query evaluation time; the interpreter is guaranteed to halt on its own without any external information on the maximum lengths of possible output strings. However, these length limit functions can still be useful in for example assessing the space requirements of query evaluation, and therefore we have developed a method for extracting them [29, Section 2.2].

The algorithm in Figure 5 provides a method for inferring finiteness dependencies satisfied by Alignment Declarations. This is similar to *treating operators as relations* in databases [30, Chapter 4]. Briefly, this suggestion means that a predefined operator like the numerical addition '+' should be treated as an infinite three-column table for the relation '$a + b = c$', as in logic. This table can be "queried" – that is, the corresponding computation carried out – only when the values of at least two of the three columns a, b and c are known, since only then will the answer be finite. That is, this table satisfies the finiteness dependencies "any two finite domains ensure the missing third". In addition, when proving a query ϕ domain independent, choosing a dependency affects choosing an evaluation strategy for ϕ: If for example column a is to be computed from columns b and c then the former must be subtracted from the latter.

Our aim is to provide similar uniformity for string handling in databases, but extend it to user-defined string operators. This is why we have chosen to study determining finiteness of string queries, rather than restricting beforehand the language actually available to the user into a part that is known in advance to ensure finiteness [7,9,10].

A correct evaluation strategy for a domain independent Alignment Calculus query could in fact be constructed from successful finiteness dependency inference [39]. Unfortunately the query evaluators in current database management systems are usually not open in this fashion, and certainly not in any implementation-independent manner. Note, however, that such openness has been advocated already for performance reasons [40]. The algorithm in Figure 5, our safety analyzer, can be easily incorporated in such open systems when they become available, leading to fully automatic safety analysis. In the meantime we provide our safety checker as a separate tool. In addition, such open systems could incorporate the aforementioned length limit functions as further information for selecting efficient query evaluation plans.

5.3. Query Processing

Earlier in this paper we have discussed how we can define new string predicates and store their compiled forms – alignment executables – in the database like any other data. However, data constituting an executable has a unique purpose of implementing its corresponding string operation in the database and thus has a special property of being interpretable as a program with string arguments producing a string relation as a result. We have discussed interpretation of these programs, apart from database setting, in Section 5.1. Let us now finally present a few ideas about the general way in which these newly defined string operations could be incorporated in database queries. We also share briefly our particular experiences in using the MySQL database platform for implementing our system.

Since one of our main goals is to make the string handling extension as universal and database independent as possible, we cannot assume much support from the database systems we are about to extend, either. Basically, as we have stated earlier, we do not expect a system to be able to integrate new string operations directly into the query language, although it would be ideal. Rather, we settle for a more realistic design which involves having a mediator between those two layers – a function which is able to simulate new string operations and which can be called from within database queries. So what we need is that the underlying database system is extensible with user-defined functions: With this support we can import our alignment executable interpreter in Figure 4 as a new user-defined function into the database system and invoke it from database queries, similarly to the built-in functions of the system. The alignment executable instructions in Table 1 were specifically designed to make their execution algorithm in Figure 4 straightforward and efficiently implementable.

Example 21 Assume for specificity that the underlying database supports returning relations as values of user-defined functions. Then the query in Example 2 could be expressed as follows:

```
SELECT y
FROM   R x,
       Alignment_executables z
WHERE  z.name='rev'
  AND  y IN interpret(z.code,'io',x.string)
```

That is, the alignment declaration in Example 3 that corresponds to the reversal predicate has been compiled beforehand into an alignment executable named `'rev'` and stored in the table **Alignment_executables** shown in Figure 3. It is first retrieved by its name as the tuple variable z goes through the table of executables, and then its code part is given to our simulation function `interpret` along with input-output pattern `'io'` and the string part of each value of the tuple variable x that ranges over the table R, meaning that the interpreter is run on the two-variable executable z.`code` with the first variable as input (i) $x.\mathsf{string}$ and the second as output (o). It outputs for each value of $x.\mathsf{string}$ the (singleton) set of all those strings y such that $x.\mathsf{string}$ and y satisfy the predicate. The set of all such y is finally the overall result of the entire query.

When comparing Example 21 to the original Example 2 a couple of drawbacks in our solution can be pointed out. First, as said above, new string operations are not directly accessible to the query but only through the interpretation function. Second, the interpreter in our extension cannot output the results directly to the logical variables of the query language, but the user must instead indicate by hand (via the choice of an appropriate input-output pattern) which argument positions of the predicate should be output via a temporary relation. However, the system offers automatic inference of permissible patterns as described in Section 5.2 and bookkeeping of the inference results in the table **Safety info** of Figure 3. We think there are two reasonable ways of dealing with the safety issue during the simulation: Either we leave the responsibility of choosing safe patterns to the user, or we have the interpreter examine the safety table to make sure the given input-output pattern is allowable before starting the simulation.

In the first case, the user may want to modify his query by adding extra conditions into the where-clause which involve checking the safety table to approve the pattern in question. The idea is to prevent starting the interpretation function in input-output situations which might not be safe. For this strategy to work as intended, the query evaluation order of the system must ensure that the conditions involving safety checking are tested before the interpreter function is started, otherwise we may still end up with an unsafe computation.

In the case where the interpretation function itself must decide whether to approve the given input-output pattern or not, the interpreter must also be able to abort the overall query evaluation when it finds out that the pattern might lead to infinite and thus unsafe computation or is simply incompatible with the number of variables of the executable. Dealing with situations like these would require similar intimate access to the query evaluation order planner of the underlying database management system as the operators-as-tables proposal discussed in Section 5.2. Some modern databases may provide their users with some control over query evaluation already. In the MySQL system, for example, query evaluation can be stopped if an error is detected within a user-defined function. In this way, it is possible for the interpreter to check the safety of the string operation on the given pattern first when beginning the simulation and not to start the actual interpretation at all if it is considered unsafe.

What if the database can return only scalar values from user-defined functions as, for example, is the case for the MySQL database system? Then, assuming that blocks of uninterpreted raw data, such as BLOBs or Binary Large Object Blocks, count as such scalars, the interpreter can return the entire state of the search – that is, the executable, the input strings and the stack – as one BLOB from which other user-defined functions can then extract the outcome of the search and the generated strings, and finally the interpreter can be restarted with the same BLOB to get the next solution. (Or in functional programming terms, each BLOB encodes the continuation [41, Chapter 3.3] of the remaining search.) Despite the possible tediousness of the whole process the result relation can still be obtained, tuple by tuple. However, in our MySQL-based extension we have taken a different, supposedly more efficient approach where the result relation is stored into an auxiliary table by the interpretation function. After the query involving the interpreter function call has terminated, the user can get information about the contents of the result relation by querying the newly created result table.

The next two examples show how the query of Example 21 could be issued in the MySQL database framework. These examples involve two different versions of the interpretation function – named 'simulate' and 'safe_simulate' – which are designed to comply with the two ways of dealing with the safety issue mentioned above. First we present an example which makes use of the unsafe version of the interpretation function.

Example 22 In MySQL the query of Example 21 could be put by means of the following consecutive queries:

```
SELECT 1
FROM   R x,
       Alignment_executables z
WHERE  z.name='rev'
  AND  simulate(z.code,'io',x.string,'result_table')

SELECT var_2
FROM   result_table

DROP TABLE result_table
```

Here, we use the unsafe version of the interpretation function named 'simulate' which does not check if the input-output pattern given in its argument list is safe for the current executable (although it does check the length and contents of the pattern) but leaves this task to the user. In this example, the pattern `'io'` is safe for the `'rev'` executable so the first query above is indeed safe. Compared with its counterpart in Example 21, its interface has an extra parameter to indicate a name for a new database table – `'result_table'` in this case – which will be created at the beginning of the simulation and where the result relation will be stored in the course of the interpreter run. So, instead of getting the simulation result relation as the return value from the interpretation function (because MySQL does not support this), the result relation is written into a database table as a side effect of the simulation. The first of the queries above runs the interpreter on the executable named `'rev'` with the first variable as input and the second as output using input strings from table R, with the interpreter storing the result relation into the table `'result_table'`. The second query outputs the column named var_2 of the table `'result_table'` where the generated values of the second variable of the string-reversing executable have been stored. The last query deletes the table `'result_table'` from the database.

If one wants to play it safe and not to worry whether the given input-output pattern is safe or not, he may use the other version of the interpretation function. This is illustrated by the example below.

Example 23 If one wants to use the interpretation function 'safe_simulate' instead of 'simulate', it is easily done with a few changes in the first query of Example 22. The interface of 'safe_simulate' is similar to the one of 'simulate' except that the first argument has to be a string constant referring to the name of the executable and the second argument – input-output pattern – has to be a constant as well (in 'simulate' it does not have to be a constant although it was in Example 22). The simulation query which involves an automatic safety check could then be written as the following:

```
SELECT 1
FROM   R x
WHERE safe_simulate('rev','io',x.string,'result_table')
```

Then the **Safety info** table of Figure 3 is consulted first to see if the executable 'rev' has been determined safe for the pattern 'io'. If not, the evaluation of the query stops with an error.

As exemplified above, there may be alternative approaches which can be taken when one considers how to deal with the safety issue or how to return query results to the user, depending on the possibilities which the underlying database system offers.

6. Conclusions

We have given an overview of a design that incorporates string processing capabilities into existing data management systems. The design is not restricted to any particular database management system or data model but it does require some extensibility features from the underlying platform. The string processing method is based on the language of Alignment Declarations [8,21] and the design evolves from our previous system-specific implementations [21,18]. In the future, we will concentrate on finishing the system-independent version of the implementation and making it generally available.

REFERENCES

1. S. Abiteboul, R. Hull, V. Vianu, Foundations of Databases, Addison-Wesley, 1995.
2. J. Collado-Vides, The search for a grammatical theory of gene regulation is formally justified by showing the inadequacy of context-free grammars, Computer Applications in the Biosciences 7 (3) (1991) 321–326.
3. D. Searls, Linguistic approaches to biological sequences, Computer Applications in the Biosciences 13 (4) (1997) 333–344.
4. J. Setubal, J. Meidanis, Introduction to Computational Molecular Biology, PWS Publishing Company, 1997.
5. D. Searls, Formal grammars for intermolecular structure, in: Intelligence in Neural and Biological Systems, 1995, pp. 30–37.
6. J. Richardson, Supporting lists in a data model (a timely approach), in: Very Large Data Bases Conference, 1992, pp. 127–138.
7. S. Ginsburg, X. Wang, Regular sequence operations and their use in database queries, Journal of Computer and System Sciences 56 (1) (1998) 1–26.

8. G. Grahne, M. Nykänen, E. Ukkonen, Reasoning about strings in databases, Journal of Computer and System Sciences 59 (1999) 116–162.
9. A. Bonner, G. Mecca, Sequences, Datalog and transducers, Journal of Computer and System Sciences 57 (3) (1998) 234–259.
10. A. Rajasekar, String-oriented databases, in: String Processing and Information Retrieval Symposium (SPIRE'99) & International Workshop on Groupware (CRIWG '99), IEEE (The Institute of Electrical and Electronics Engineers, Inc.), 1999, pp. 158–167.
11. G. Grahne, E. Waller, How to make SQL stand for String Query Language, in: R. Connor, A. Mendelzon (Eds.), International Workshop on Database Programming Languages, Vol. 1949 of Lecture Notes in Computer Science, 1999, pp. 61–79.
12. K. Lemström, L. Hella, Approximate pattern matching is expressible in transitive closure logic, in: Logic in Computer Science, 2000, pp. 157–167.
13. M. Benedikt, L. Libkin, T. Schwentick, L. Segoufin, String operations in query languages, in: ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems, 2001, pp. 183–194.
14. S. Ganguly, M. Noordewier, PROXIMAL: a database system for the efficient retrieval of genetic information, Computers in Biology and Medicine 26 (3) (1996) 199–207.
15. P. Pistor, R. Traunmüller, A database language for sets, lists and tables, Information Systems 11 (4) (1986) 323–336.
16. P. Seshadri, M. Livny, R. Ramakrishnan, SEQ: A model for sequence databases, in: IEEE International Conference on Data Engineering, 1995, pp. 232–239.
17. B. Subramanian, T. Leung, S. Vandenberg, S. Zdonik, The AQUA approach to querying lists and trees in object-oriented databases, in: IEEE International Conference on Data Engineering, 1995, pp. 80–89.
18. R. Hakli, M. Nykänen, H. Tamm, E. Ukkonen, Implementing a declarative string query language with string restructuring, in: G. Gupta (Ed.), First International Workshop on Practical Aspects of Declarative Languages (PADL'99), Vol. 1551 of Lecture Notes in Computer Science, Springer-Verlag, 1999, pp. 179–195.
19. C. Helgesen, P. Sibbald, PALM - a pattern language for molecular biology, in: International Conference on Intelligent Systems in Molecular Biology, 1993, pp. 172–180.
20. D. Searls, String Variable Grammar: A logic grammar formalism for the biological language of DNA, Journal of Logic Programming (1995) 73–102.
21. G. Grahne, R. Hakli, M. Nykänen, E. Ukkonen, AQL: An alignment based query language for querying string databases, in: C. Prabhu (Ed.), Databases for the Millennium 2000: Proceedings of the Ninth International Conference on Management of Data (COMAD'98), Tata McGraw-Hill Publishing Company Limited, 1998, pp. 235–251.
22. R. Hakli, M. Nykänen, H. Tamm, A declarative programming system for manipulating strings, in: Sixth Fenno-Ugric Symposium on Software Technology, 1999, pp. 29–40.
23. R. Hakli, M. Nykänen, H. Tamm, Adding string processing capabilities to data management systems, in: Proceedings of the Seventh International Symposium on String Processing and Information Retrieval (SPIRE 2000), IEEE Computer Society Press, 2000, pp. 122–131.
24. G. Stoesser, W. Baker, A. van den Broek, E. Camon, M. Garcia-Pastor, C. Kanz,

T. Kulikova, V. Lombard, R. Lopez, H. Parkinson, N. Redaschi, P. Sterk, P. Stoehr, M. Tuli, The EMBL nucleotide sequence database, Nucleic Acids Research 29 (1) (2001) 17–21.

25. D. Gusfield, Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology, Cambridge University Press, 1997.
26. E. Emerson, Temporal and modal logic, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, Vol. B: Formal Models and Semantics, Elsevier, 1990, Ch. 16, pp. 994–1072.
27. A. Bonner, G. Mecca, Querying sequence databases with transducers, Acta Informatica 36 (2000) 511–544.
28. J. Hopcroft, J. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, 1979.
29. M. Nykänen, Using acceptors as transducers, Theoretical Computer Science 267 (1/2) (2001) 83–104.
30. C. Date, H. Darwen, Foundation for Object/Relational Databases – The Third Manifesto, Addison-Wesley, 1998.
31. D. Searls, K. Murphy, Automata-theoretic models of mutation and alignment, in: International Conference on Intelligent Systems in Molecular Biology, 1995, pp. 341–349.
32. F. Lefebvre, A grammar-based unification of several alignment and folding algorithms, in: International Conference on Intelligent Systems in Molecular Biology, 1996, pp. 143–155.
33. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, Object-Oriented Modeling and Design, Prentice-Hall, 1991.
34. W. Brauer, K.-J. Lange, Non-deterministic two-tape automata are more powerful than deterministic ones, in: Symposium on Theoretical Aspects of Computer Science, 1985, pp. 71–79.
35. J. Lloyd, Foundations of Logic Programming, 2nd Edition, Springer-Verlag, 1987.
36. M. Nykänen, E. Ukkonen, Finding paths with the right cost, in: C. Meinel, S. Tison (Eds.), Symposium on Theoretical Aspects of Computer Science, Vol. 1563 of Lecture Notes in Computer Science, 1999, pp. 345–355.
37. M. Nykänen, E. Ukkonen, The exact path length problem, Journal of Algorithms 42 (1) (2002) 41–53.
38. K. Sohn, A. van Gelder, Termination detection in logic programs using argument sizes (extended abstract), in: ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems, 1991, pp. 216–226.
39. G. Grahne, M. Nykänen, Safety, translation and evaluation of Alignment Calculus, in: Advances in Databases and Information Systems, electronic Workshops in Computing, British Computer Society, 1997, pp. 295–304, also available via `http://www.ewic.org.uk/ewic/workshop/view.cfm/ADBIS-97`.
40. P. Seshadri, M. Livny, R. Ramakrishnan, The case for enhanced abstract data types, in: Very Large Data Bases Conference, 1997, pp. 66–75.
41. R. Dybvig, The Scheme Programming Language: ANSI Scheme, 2nd Edition, Prentice-Hall, 1996.

A. Syntax for the Alignment Declaration language

The complete syntax for the Alignment Declaration language is given in the form of a context-free grammar [28, Chap. 4.2] below

$$
\begin{array}{rcl}
\mathit{Declaration} & \longrightarrow & \mathit{Header}\ \mathit{Body} \\
\mathit{Header} & \longrightarrow & \mathit{IntrAlph}\ \mathit{IntrPred} \quad | \quad \mathit{IntrPred} \\
\mathit{IntrAlph} & \longrightarrow & \mathit{IntrAlph}\ \mathit{AlphaDecl} \quad | \quad \mathit{AlphaDecl} \\
\mathit{AlphaDecl} & \longrightarrow & \texttt{alphabet}\ \mathit{AlphName}\ \texttt{:}\ \mathit{CharList} \\
\mathit{AlphName} & \longrightarrow & \mathit{Var} \quad | \quad \texttt{ASCII} \\
\mathit{Var} & \longrightarrow & \mathit{Letter} \quad | \quad \mathit{Var}\ \mathit{Letter} \quad | \quad \mathit{Var}\ \mathit{Digit} \\
\mathit{Letter} & \longrightarrow & \texttt{A} \mid \texttt{B} \mid \ldots \mid \texttt{Z} \mid \texttt{a} \mid \texttt{b} \mid \ldots \mid \texttt{z} \\
\mathit{Digit} & \longrightarrow & \texttt{0} \mid \texttt{1} \mid \ldots \mid \texttt{9} \\
\mathit{CharList} & \longrightarrow & \mathit{AlphChar} \quad | \quad \mathit{AlphChar}, \mathit{CharList} \\
\mathit{AlphChar} & \longrightarrow & \mathit{Char} \quad | \quad \mathit{Char} - \mathit{Char} \\
\mathit{Char} & \longrightarrow & \texttt{'}\mathit{C}\texttt{'} \\
\mathit{C} & \longrightarrow & \sqcup \mid \texttt{!} \mid \texttt{"} \mid \texttt{\#} \mid \ldots \mid \texttt{\&} \mid \texttt{(} \mid \texttt{)} \mid \ldots \mid \texttt{[} \mid \texttt{]} \mid \texttt{\^{}} \mid \ldots \mid \\
 & & \texttt{\~{}} \mid \texttt{\"{}} \mid \texttt{\textbackslash 0\textbackslash} \mid \texttt{\textbackslash 1\textbackslash} \mid \ldots \mid \texttt{\textbackslash 255\textbackslash} \mid \texttt{\textbackslash\textbackslash}
\end{array}
$$

(This means all printable characters in the 7-bit ASCII alphabet except ‘’’ and ‘\’ plus ‘\\’ denoting ‘\’ and decimal form ASCII-codes inserted between two backslashes.)

$$
\begin{array}{rcl}
\mathit{IntrPred} & \longrightarrow & \mathit{PredName}\texttt{(}\mathit{Param}\texttt{)} \\
\mathit{PredName} & \longrightarrow & \mathit{Var} \\
\mathit{Param} & \longrightarrow & \mathit{Param}\texttt{,}\ \mathit{StrVar} \quad | \quad \mathit{StrVar} \\
\mathit{StrVar} & \longrightarrow & \mathit{Var} \\
\mathit{Body} & \longrightarrow & \mathit{Slist} \\
\mathit{Slist} & \longrightarrow & \mathit{Stat}\ \mathit{Slist} \quad | \quad \mathit{Stat} \\
\mathit{Stat} & \longrightarrow & \mathit{SimpStat} \quad | \quad \mathit{CompStat} \\
\mathit{SimpStat} & \longrightarrow & \mathit{OnStat} \quad | \quad \mathit{ResetStat} \quad | \quad \mathit{ReadStat} \quad | \\
 & & \mathit{RightreadStat} \quad | \quad \mathit{CallStat} \quad | \quad \mathit{MarkStat} \\
\mathit{OnStat} & \longrightarrow & \mathit{OnPart} \quad | \quad \mathit{ScanPr}\ \mathit{OnPart} \quad | \\
 & & \mathit{RightscanPr}\ \mathit{OnPart} \quad | \quad \mathit{ScanstarPr}\ \mathit{OnPart} \quad | \\
 & & \mathit{RightscanstarPr}\ \mathit{OnPart} \\
\mathit{OnPart} & \longrightarrow & \texttt{on} \quad | \quad \texttt{on}\ \mathit{Expr} \\
\mathit{ScanPr} & \longrightarrow & \texttt{scan}\ \mathit{Param} \\
\mathit{ScanstarPr} & \longrightarrow & \texttt{scan*}\ \mathit{Param} \\
\mathit{RightscanPr} & \longrightarrow & \texttt{rightscan}\ \mathit{Param} \\
\mathit{RightscanstarPr} & \longrightarrow & \texttt{rightscan*}\ \mathit{Param} \\
\mathit{Expr} & \longrightarrow & \mathit{AExpr}\ \texttt{or}\ \mathit{Expr} \quad | \quad \mathit{AExpr}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{AExpr} & \longrightarrow & \mathit{BExpr}\ \texttt{and}\ \mathit{AExpr} \ \mid\ \mathit{BExpr} \\
\mathit{BExpr} & \longrightarrow & \texttt{not}\ \mathit{BExpr} \ \mid\ \mathit{CExpr} \\
\mathit{CExpr} & \longrightarrow & \texttt{(}\mathit{Expr}\texttt{)} \ \mid\ \mathit{DExpr} \\
\mathit{DExpr} & \longrightarrow & \mathit{StrVar}\texttt{=}\mathit{StrVar} \ \mid\ \mathit{StrVar}\texttt{=}\mathit{Char} \ \mid \\
 & & \mathit{StrVar}\texttt{=}\mathit{Endmark} \ \mid\ \mathit{InStat} \\
\mathit{Endmark} & \longrightarrow & \texttt{[} \ \mid\ \texttt{]} \ \mid\ \texttt{before} \ \mid\ \texttt{after} \\
\mathit{InStat} & \longrightarrow & \mathit{StrVar}\ \texttt{in}\ \mathit{InRest} \\
\mathit{InRest} & \longrightarrow & \mathit{AlphName} \ \mid\ \mathit{CharList} \\
\mathit{ResetStat} & \longrightarrow & \texttt{reset}\ \mathit{StrVar} \\
\mathit{ReadStat} & \longrightarrow & \texttt{read}\ \mathit{StrVar}\ \texttt{on}\ \mathit{String} \\
\mathit{String} & \longrightarrow & \texttt{'}\mathit{S}\texttt{'} \\
\mathit{S} & \longrightarrow & \mathit{C} \ \mid\ \mathit{SC} \\
\mathit{RightreadStat} & \longrightarrow & \texttt{rightread}\ \mathit{StrVar}\ \texttt{on}\ \mathit{String} \\
\mathit{CallStat} & \longrightarrow & \texttt{call}\ \mathit{MacroName}\texttt{(}\mathit{Param}\texttt{)} \\
\mathit{MacroName} & \longrightarrow & \mathit{Var} \\
\mathit{MarkStat} & \longrightarrow & \texttt{mark}\ \mathit{Marking} \\
\mathit{Marking} & \longrightarrow & \mathit{Marking}\texttt{,}\mathit{Marking} \ \mid\ \mathit{StrVar} \ \mid\ \mathit{String} \\
\mathit{CompStat} & \longrightarrow & \mathit{RepeatStat} \ \mid\ \mathit{ChooseStat} \ \mid\ \mathit{KeepStat} \\
\mathit{RepeatStat} & \longrightarrow & \mathit{RepeatPr}\ \mathit{Slist}\ \mathit{EndPr} \\
\mathit{RepeatPr} & \longrightarrow & \texttt{repeat}\ \mathit{Count}\ \texttt{times} \\
\mathit{Count} & \longrightarrow & \texttt{*} \ \mid\ \texttt{+} \ \mid\ \mathit{Int} \\
\mathit{Int} & \longrightarrow & \mathit{Digit} \ \mid\ \mathit{Int}\ \mathit{Digit} \\
\mathit{EndPr} & \longrightarrow & \texttt{end} \\
\mathit{ChooseStat} & \longrightarrow & \texttt{choose}\ \mathit{Slist}\ \texttt{|}\ \mathit{Slist}\ \mathit{ChooseRest} \\
\mathit{ChooseRest} & \longrightarrow & \texttt{|}\ \mathit{Slist}\ \mathit{ChooseRest} \ \mid\ \mathit{EndPr} \\
\mathit{KeepStat} & \longrightarrow & \texttt{keep}\ \mathit{InStat}\ \mathit{Slist}\ \mathit{EndPr}
\end{array}
$$