AQL: an Alignment Based Language for Querying String Databases

Gösta Grahne
Department of Computer Science, Concordia University
Montreal H3G 1M8, Canada
grahne@cs.concordia.ca

Raul Hakli* Matti Nykänen Esko Ukkonen
Department of Computer Science
P.O. Box 26, FIN–00014 University of Helsinki, Finland
{Raul.Hakli,Matti.Nykanen,Esko.Ukkonen}@cs.Helsinki.Fi

Abstract

AQL stands for Alignment Query Language, and it is an extension of SQL. AQL allows for declarative querying of databases containing *strings* of characters as entries. For instance, in applications such as genomic databases, the DNA-sequences are strings, and they need to be queried based on their structure. The LIKE-operation in SQL does not go very far. Existing systems for string databases have a very ad hoc nature, usually giving the user a set of predefined simple query forms. On the other hand, there has recently been considerable theoretical advancement in the field of string databases. Several languages have been proposed, and their expressive power is becoming illuminated. However, there has been no practical demonstration of the feasibility of these languages. In this paper we describe an implementation of *Alignment Calculus*, a powerful string database language based on a modal logic extension of relational calculus. The implementation yields AQL. In AQL the user can write declarative string predicates in the WHERE-clause of a query. The predicates can be of an arbitrary arity, expressing for instance a unary property of a string being a palindrome, or expressing a binary property of one string being the reversal on another, and so on. The string predicates are translated into a novel form of a high level finite state automaton, and they are evaluated using an inference engine for chasing these automata. The AQL system has been implemented as an extension of a relational database management system. Our experiments show that the inference engine has the efficiency of an equivalent query computed in compiled Prolog. This is efficiency achieved already without using optimisation, something which the string queries due to their declarative nature are amenable to.

Keywords: string databases, query languages, multi-tape automata, biological sequence data, declarative languages, modal logic

*Supported by the Academy of Finland, grant number 18814.

1 Introduction

Current database management systems, especially those based on the relational model, have little support for string querying and manipulation. This can be a problem in several string-oriented application areas such as molecular biology and text processing. A string is one of the basic data types of Codd's relational model [3], which means that the strings are treated as atomic entities; thus a string can be accessed only as a whole and not on the level of the individual characters occurring within it. Modern object-oriented systems are usually alike in this sense. Although they offer support for complex objects, strings are usually treated as atomic. In SQL, the only non-atomic operator is the LIKE-operator which can be used in simple pattern matching tasks such as finding a substring in a field; however, the expressive power of the operator is limited.

Several methods for string and sequence manipulation and querying have been proposed. Ginsburg and Wang [5, 27, 6] have suggested Sequence Logic, which is predicate calculus extended with sequence predicates. The predicates are based on pattern merging with regular expressions, and they can be used to express properties of sequences and to manipulate sequences. Richardson [20] has modified temporal logic for list querying. Richardson's logic can be used to express properties of lists, which may also be nested. The expressive power of temporal logic is known not to cover regular languages [30], but Richardson's extensions such as rigid variables allow some context-free properties to be expressed. Mecca and Bonner [2, 14] extend Datalog for string manipulation, in the spirit of Sequence Logic. Their language is highly expressive with respect to sequence functions. Grahne, Nykänen, and Ukkonen have presented Alignment Calculus, a powerful modal logic that can be used in string querying [8, 9, 17].

The above theoretical investigations came up with interesting models for expressing string properties but, thus far, few working implementations have been developed. This means that the practical applicability of these models is not known. The research presented in this paper studies the practical aspects of Alignment Calculus by restricting its expressivity, modifying its syntax, and attaching it to a database management system. The resulting language, originally presented in [10], is called Alignment Query Language (AQL).

Some other more practically oriented proposals have been appeared in application areas like molecular biology where long DNA-sequences are represented as character strings. These suggestions include Searls' String Variable Grammars [21], pattern matching systems like QUEST [1], ANREP [16], PAMALA [15], and PALM [11], and database query language extensions like Proximal [4]. Apart from Proximal, these implementations have not been studied in a database setting. Our implementation has some similarities to the Proximal system in the idea of extending SQL-queries with special predicates describing properties of strings. Proximal, however, offers only a limited set of preinstalled predicates, while our system gives the user the possibility to define predicates of her own using AQL.

Other related work has been carried out in the study of complex object databases and sequence databases. These suggestions include for example the extended NF^2 model [19], the SEQ-model [22], and the AQUA data model [25]. Implementations based on these suggestions have also been reported [23, 28]. The difference between these approaches compared to ours is that they are interested in developing a new data model supporting ordered data types like lists of records while our focus is in the string fields of the data model in question.

In the next section, we will give an overview of the AQL. After that we define its syntax and give examples of its use. Then we discuss our prototype implementation and some ongoing improvement projects. Finally, we state some conclusions about the subject.

2 Overview of the language

AQL is based on Alignment Calculus, a modal logic extension of relational calculus. Expressions written in AQL can be used to define *predicates* describing properties of individual strings or relations between several strings. Let Σ be a set of symbols, called the alphabet, and let s_1, s_2, and s_3 be strings in Σ^*. As an example of a property of an individual string, a one-place predicate $\mathit{contains\_tata}(s_1)$ is true if string s_1 contains a substring '`tata`'. As an example of a relation between three strings, a predicate $\mathit{shuffle}(s_1, s_2, s_3)$ is true if string s_1 is a shuffle of strings s_2 and s_3. The shuffle of strings u and v is the set of all strings of the form $u_1v_1u_2v_2 \ldots u_kv_k$ where $u = u_1u_2 \ldots u_k$, $v = v_1v_2 \ldots v_k$, and each u_i and v_i can be of arbitrary length, including zero. Predicates like these can be defined and used as selection conditions in database queries.

In both AQL and Alignment Calculus, the basic semantical construction is the *alignment*. Figure 1 presents five different alignments $A_1, \ldots, A_5$ of the three strings `a`, `g`, and `ag`. A designated *window column* runs through each alignment. The alignments are connected together with accessibility relations called *transposes*. Figure 1 shows as arrows six transposes between the alignments $A_1, \ldots, A_5$. Each arrow is labelled with $[\text{rows}]_{\texttt{direction}}$ to denote that the second alignment is obtained from the first by *sliding* the designated rows one character into the designated direction; for example, alignment A_3 is obtained from alignment A_2 by sliding rows 2 and 3 one character to the left. Intuitively, the string handling part of Alignment Calculus will consist of a language for specifying paths through such alignment-transpose structures, or equivalently a sequence of slides and window column tests. Note in particular that sliding is allowed to both directions.

In alignments, it is possible to express propositions such as "the characters on the first and third rows positioned at the window column are equal". These kinds of propositions are called *window formulae*. This proposition is true in alignment A_2 but false in alignment A_3 of Figure 1. This window formula can be written as $x_1 = x_3$, where x_i refers to the window column character of string i. In a similar fashion, the window formula $x_2 = \texttt{a} \vee x_2 = \texttt{g}$ can be interpreted to mean that the character positioned at the window column on the second string is either `a` or `g`. Furthermore, an expression like $x_2 = \perp$ means that no character of the second string is located at the window column. This means that the second string is located either one step to the left or to the right of the window column. In the *initial alignment* every string is located one step to the right of the window columns as in alignment A_1 of Figure 1.

An *atomic string formula* can be obtained by adding a transpose to an arbitrary string. For example, $[x_1]_l x_1 = x_3$ holds in an alignment if the first and third row characters on the window

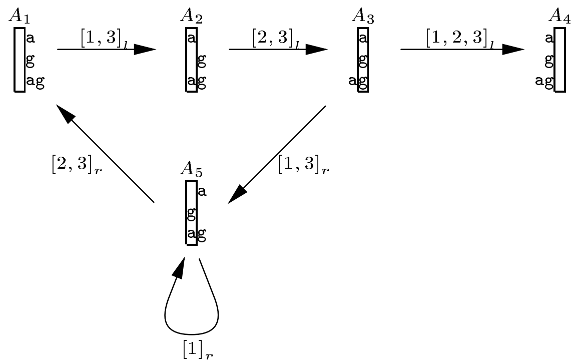


Figure 1: Alignments and their transposes

column are equal after sliding the first row one step to the left. The formula above holds in alignment A_5 but not in alignment A_2 of Figure 1.

Furthermore, *string formulae* are regular expressions composed of atomic string formulae using concatenation, Kleene closure, and union operators. For instance, $([x_1, x_2, x_3]_l x_1 = \perp \ \wedge \ x_2 = \perp).(([x_1, x_3]_r x_3 = \mathrm{a}) + ([x_0]_r \neg x_0 = \perp)^*)$ is a string formula. Alignment Calculus is then domain relational calculus equipped with string formulae [7].

Before going into the details of AQL, let us now give an example of how the string properties can be expressed. In Alignment Calculus, a query returning all strings in relation R containing a substring '`tata`' would be expressed as

$$\begin{array}{lll} x & | & R(x)\wedge \\ & & ([x]_l x \neq \perp)^*.([x]_l x = \mathtt{t}).([x]_l x = \mathtt{a}).([x]_l x = \mathtt{t}).([x]_l x = \mathtt{a}). \end{array}$$

The intuitive idea in this query is to extract all such strings x from the database relation R for which it is possible to proceed to some character position, where the four successive characters are '`t`', '`a`', '`t`', and '`a`', respectively.

By AQL we mean SQL extended with *Alignment Declarations*, which correspond to the string handling parts of the Alignment Calculus. Consider the string part of the previous query, that is, the part after the conjunction symbol '$\wedge$'. This part defines the *contains_tata*(x) predicate. Using Alignment Declarations it would be written as

```
scan* x on not x=']'
scan x on x='t'
scan x on x='a'
scan x on x='t'
scan x on x='a'
```

or more briefly as follows:

```
scan* x on not x=']'
read x on "tata"
```

The idea of AQL is to add these kinds of expressions into SQL-queries and use them as selection predicates. Suppose that we have a database relation **sequence_table** with an attribute called **sequence**. The relation consists of the DNA-sequences represented as strings as shown in Table 1.

id	...	**sequence**
D12466		`gatcaaataaagttataaaattgccttc`
H55186		`gtggagaagccgccactccgagaagggaa`
XYZ123		`gatcaatattgaaataaaaattgccttc`
XXYYZZ		`cattgattcaacatactttaaa`
X64000		`cttagttacaacatactttaaa`

Table 1: The contents of the **sequence_table**

Suppose then that we are interested in finding all the sequences satisfying the predicate *contains_tata*. The following AQL-query should then return all the sequences in the table **sequence_table** containing a substring '`tata`'.

```
SELECT id, sequence
FROM sequence_table S
WHERE x=S.sequence AND
      scan* x on not x=']'
      read x on "tata"
```

The results of this query are shown in Table 2.

id	sequence
D12466	gatcaaataaagttataaaattgccttc

Table 2: The query result

3 The syntax of AQL

The syntax of the Alignment Declarations of AQL is a redefinition of the syntax of a subset of Alignment Calculus [8, 17]. We now define the relation between expressions written in Alignment Calculus and expressions written in Alignment Declarations of AQL.

The atomic window formulae $x = y$ and $x = \mathtt{a}$ of Alignment Calculus correspond to expressions $E_{x=y} \equiv$ `x=y` and $E_{x=\mathtt{a}} \equiv$ `x='a'`, where `x` and `y` are tape variables, and `a` is an element in Σ. There are two different expressions in AQL that correspond to the atomic window formula $x = \perp$. They are `x='['`, which is true in the beginning of a sequence, and `x=']'`, which becomes true in the end of a sequence. Markers '`[`' and '`]`' are called the left and the right end markers, respectively.

In Alignment Calculus, the atomic window formulae can be used to construct the window formulae $\neg P$, $P \wedge Q$, and $P \vee Q$. In AQL, the corresponding formulae are $E_{\neg P} \equiv$ `not` E_P, $E_{P \wedge Q} \equiv E_P$ `and` E_Q, and $E_{P \vee Q} \equiv E_P$ `or` E_Q. The modal operators of the Alignment Calculus have been replaced by tape shifting instructions `scan` and `rightscan` so that the atomic string formulae $[x_1, \ldots, x_k]_l \phi$ and $[x_1, \ldots, x_k]_r \phi$ are written in the following fashion: $E_{[x_1,\ldots,x_k]_l \phi} \equiv$ `scan` $x_1, \ldots, x_k$ `on` E_ϕ and $E_{[x_1,\ldots,x_k]_r \phi} \equiv$ `rightscan` $x_1, \ldots, x_k$ `on` E_ϕ.

In Alignment Calculus, the atomic string formulae are used to construct the string formulae $\phi.\psi$, $\phi + \psi$, (ϕ) and ϕ^*. These correspond to the control structures of AQL, and they are written as $E_{\phi.\psi} \equiv E_\phi\ E_\psi$, $E_{\phi+\psi} \equiv$ `choose` E_ϕ `|` E_ψ `end`, $E_{(\phi)} \equiv$ `(`E_ϕ`)`, and $E_{\phi^*} \equiv$ `repeat` E_ϕ `*times`, respectively.

In addition to these, we have defined some extra reserved words that can be used to compress the expressions. These include instructions like `scan*`, `rightscan*`, `reset`, `read`, and `rightread`. The commands `scan*` and `rightscan*` can be used to nondeterministically read the string in either direction. The `reset`-statement can be used to rewind a tape in the beginning. The instructions `read` and `rightread` can be used as abbreviations of multiple `scan` or `rightscan` statements, as shown in the *contains_tata*-example above.

Table 3 shows the elements of the Alignment Declarations of AQL together with an explanation. Symbol ϕ denotes a window formula, and symbols Φ and Ψ represent sequences of multiple statements. To make it easier to understand the examples given, the explanations are given from a programmer's point of view giving them a procedural flavour; however, the underlying logical semantics is declarative. For instance, the explanation for a logical proposition of form `scan x on` ϕ can be given as "The proposition holds if ϕ holds in a configuration where string x is located one step to the left compared to the current configuration".

Statement	Explanation
`scan x on` ϕ	If ϕ is satisfied by moving one step forward on string x then make the move.
`rightscan x on` ϕ	If ϕ is satisfied by moving one step backward on string x then make the move.
`scan* x on` ϕ	Repeat `scan x on` ϕ for an arbitrary number of times.
`rightscan* x on` ϕ	Repeat `rightscan x on` ϕ for an arbitrary number of times.
`repeat` Φ `*times`	Repeat the statements Φ for an arbitrary number of times.
`choose` Φ `|` Ψ `end`	Nondeterministically choose either the statements Φ or Ψ.
`reset x`	Move to the beginning of string x.
`read x on "abc"`	Abbreviation of multiple `scan`-statements.
`rightread x on "abc"`	Abbreviation of multiple `rightscan`-statements.

Table 3: The elements of Alignment Declarations

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

Example 1 String s_1 is a prefix of string s_2:

```
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 2 String s_1 is a substring of string s_2:

```
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 the 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 the 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 inference engine to find the right position. ■

Example 3 String s_1 is a suffix of string s_2:

```
scan* x,y
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 reach the right end marker. Note that s_1 and s_2 do not have to be of equal length for both them to reach the end marker simultaneously because the transposes are defined so that sliding a string is always allowed but the string does not move further when the end marker has been reached. This can be seen in alignment A_4 of Figure 1: Sliding the first string to the right does not alter the alignment. After reaching the end markers, we proceed symmetrically with respect to the prefix example above, this time moving to the opposite direction. ■

Example 4 String s_1 is a subsequence of string s_2:

```
repeat
   choose
      scan x,y on x=y and not y=']' |
      scan y on not y=']'
   end
*times
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 after the slide 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 are to be found in s_2, we say that string s_1 is a subsequence of string s_2, and the predicate holds. ■

Example 5 The string s_1 is a shuffle of the strings s_2 and s_3:

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

This example is similar to the previous one, except for 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.

Example 6 The suffix of string s_1 equals the prefix of string s_2 in at least $|s_3|$ character positions ($|s_3|$ denotes the length of string s_3):

```
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 slided forward. If s_3 comes to an end, the overlapping part is long enough. ■

Example 7 The edit distance of strings s_1 and s_2 is less or equal than $|s_3|$:

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

In this example, whenever the characters on strings s_1 and s_2 do not match, we choose one of the transformations (replacement, insertion, or deletion) and proceed on string s_3. This time the counter strings s_3 must not reach the end too soon. ■

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.

Example 8 String s_1 is a palindrome:

```
scan* x on not x=']'
scan x on x=']'
repeat
   scan y
   rightscan x on x=y
*times
scan y on y=']'
rightscan x on x='['
```

Suppose now that string s_2 is a copy of string s_1. This way we can have a situation where we have two pointers to one string. In the implementation, this can be achieved by passing the same string twice as an argument to a predicate. First, string s_1 is slided all the way to the end while string s_2 stays still. Then we proceed forward on string s_2 and backward on string s_1 and verify that all the characters match. ■

Example 9 String s_1 is of the form ww for some string w.

```
repeat
   scan x,y
   scan x on not x=']'
*times
scan x,y on x=']'
rightscan* x,y on x=y and not y='['
rightscan y on y='['
```

Assume as in the example above that s_1 and s_2 are the same string of length n. If n is odd then the `repeat`-loop fails. Otherwise we have proceeded onto the last (nth) character of s_1 and onto the $(n/2)$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. ∎

4 The prototype system

A prototype system using Alignment Query Language was implemented as an extension to the PostgreSQL database management system [31]. The project of extending a database system to allow the users to construct their own predicates poses certain requirements for the database management system. In our first prototype, we chose the PostgreSQL database management system because it can easily be extended with user defined functions written in the C programming language [31]. While this feature is common among object-oriented database management systems, it usually is not available in systems based on the relational model. Although we wanted to test our ideas using the relational model in the first prototype, the Alignment Declarations are not connected to any specific data model. In fact, our current projects include a similar extension to object-oriented database management system O_2 [13], and a World Wide Web -based system to query biological flat file databases [26].

The first prototype system can be used by defining predicates using the Alignment Declarations, and by using these predicates in queries. The communication between the DBMS and our inference engine is carried out by using two functions written in C. The first one creates a new predicate to be available in later queries, and the latter one evaluates the predicate using the other parameters as operands.

Figure 2 represents the overall architecture of our system. When the Alignment Declarations of AQL are introduced, a corresponding finite state automaton is constructed and saved to the disk for later use. When the predicate is used in a query, the automaton is loaded into memory and simulated using the data base strings as input. We are using multi-tape, two-way, nondeterministic finite state automata, where each string is coded into a separate tape, and for each k-ary predicate there is a corresponding k-tape automaton.

The model of the automaton differs from standard automata in the definition of the transitions between the states of the automaton. The transitions are defined using an *acceptance condition*, which is an atomic string formula of AQL. This means that instead of the usual transition labels defined by combinations of input symbols, the transitions are labelled with conditions which must be true for the transitions to be possible. The truth of the formulae depends on the symbols appearing on the tapes. Consider for instance an acceptance condition `scan 1,2 on 1=2 and not 1=']'`. This formula is true if and only if after moving both the first and the second tape one step forward, the character appearing on the first tape is the same as the character appearing on the second tape but this character must not be the right end marker.

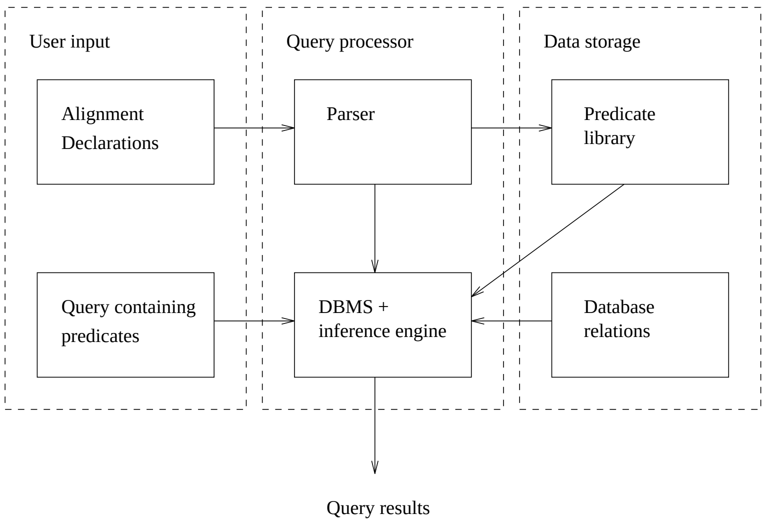


Figure 2: The AQL system

These acceptance conditions are evaluated during the simulation when a predicate is used in a query. The operands of the predicate, that is the input strings, are encoded into the tapes adding special left and right end markers, '`[`' and '`]`'. If the simulation reaches an accepting final state, the property defined by the predicate holds for the input strings. The input strings can be either strings contained in the database or constant parameters given in the query.

Formally, the k-tape automaton $\mathcal{A}$ is defined as a 5-tuple $(Q, \Sigma \cup \{\mathtt{[}, \mathtt{]}\}, s, F, T)$, where Q is a finite set of *states*, Σ is an input *alphabet*, $s \in Q$ is a *start* state, $F \subseteq Q$ is the set of *final* states, and T is a *transition relation*. The transition relation consists of *transitions* of the form $\langle\langle q, \Phi\rangle, q'\rangle$, where q and q' are states in Q, and Φ is the acceptance condition. The acceptance condition Φ is either ϵ or an atomic string formula, where the variables have been replaced by integers functioning as indices to the tapes.

Usually in the case of two-way automata, the direction of the movement is explicitly stated in the relation. In the definition above, however, this information is implicitly included in the acceptance condition, for it functionally depends on the string formula. For the tape heads mentioned in a `scan`-formula, the direction is left, while the tape heads unmentioned hold their position. Similarly, for the tape heads mentioned in a `rightscan`-formula, the direction is right, while the other tape heads remain still.

The *configurations* of $\mathcal{A}$ on *input* $W = \langle w_1, \ldots, w_k\rangle$, where for each $i = 1, \ldots, k$ w_i is a string in Σ^*, are tuples of the form $\langle q, n_1, \ldots, n_k\rangle$, where $q \in Q$, and $0 \leq n_i \leq |w_i| + 1$ for every tape $i = 1, \ldots, k$. This means that the current state of the automaton is q, and each tape head i is located in position n_i, that is, on the n_ith character of string w_i. In addition, $w_{i_0} =$'`]`' and $w_{i_{|w_i|+1}} =$'`]`' for each string w_i.

The evaluation of the atomic string formulae is based on the following definitions. $S \Vdash \phi$ means that formula ϕ is *applicable* in configuration $S = \langle q, n_1, \ldots, n_k\rangle$ on input $W = \langle w_1, \ldots, w_k\rangle$.

1. $S \Vdash (i = \mathtt{[})$, if and only if $w_i[n_i] = \mathtt{[}$.
2. $S \Vdash (i = \mathtt{]})$, if and only if $w_i[n_i] = \mathtt{]}$.

3. $S \Vdash (i = a)$, if and only if $w_i[n_i] = a$, where $a \in \Sigma$.

4. $S \Vdash (i = j)$, if and only if $w_i[n_i] = w_j[n_j]$.

5. $S \Vdash (\phi$ `and` $\psi)$, if and only if $S \Vdash \phi$ and $S \Vdash \psi$.

6. $S \Vdash$ (`not` ϕ), if and only if $S \nVdash \phi$.

7. $S \Vdash$ (`scan` $i_1, \ldots, i_p$ `on` ϕ), if and only if $[i_1, \ldots, i_p]_l S \Vdash \phi$, where $[i_1, \ldots, i_p]_l S$ is obtained from S by moving one step forward each head $i \in \{i_1, \ldots, i_p\}$ not already located on the right end marker.

8. $S \Vdash$ (`rightscan` $i_1, \ldots, i_p$ `on` ϕ), if and only if $[i_1, \ldots, i_p]_r S \Vdash \phi$, where $[i_1, \ldots, i_p]_r S$ is obtained from S by moving one step backward each head $i \in \{i_1, \ldots, i_p\}$ not already located on the left end marker.

The transitions of the automaton $\mathcal{A}$ can now be defined as follows: for each $q, q' \in Q, i \in \mathbb{N}, w_i \in \Sigma^*, k \in \mathbb{N}_+, n_i \in \mathbb{N}$, and $\phi \in \Phi$ if $\langle\langle q, \epsilon\rangle, q', \rangle \in T$ or $\langle\langle q, \phi\rangle, q'\rangle \in T$ and $\langle q, n_1, \ldots, n_k\rangle \Vdash \phi$, then configuration $\langle q', n'_1, \ldots, n'_k\rangle$ is a *next* configuration of $\langle q, n_1, \ldots, n_k\rangle$ (denoted $\langle q, n_1, \ldots, n_k\rangle \vdash_M \langle q', n'_1, \ldots, n'_k\rangle$), where

$$n'_i = \begin{cases} n_i & \text{if } w_i[n_i] = \texttt{[} \text{ and } d_i = -1 \text{ or } w_i[n_i] = \texttt{]} \text{ and } d_i = 1 \\ n_i + d_i & \text{otherwise.} \end{cases}$$

Here d_i is the direction of tape i, and it is determined as follows: For a transition of the form $\langle\langle q, \epsilon\rangle, q', \rangle$ the direction $d_i = 0$ for each tape i. For a transition of the form $\langle\langle q, \phi\rangle, q'\rangle$, the directions depend on the formula ϕ. If ϕ is of form `scan` $x_1, \ldots, x_j$ `on` ψ then $d_i = 1$ for each $x_i \in \{x_1, \ldots, x_j\}$, and $d_i = 0$, for each $x_i \notin \{x_1, \ldots, x_j\}$. If ϕ is of form `rightscan` $x_1, \ldots, x_j$ `on` ψ then $d_i = -1$ for each $x_i \in \{x_1, \ldots, x_j\}$, and $d_i = 0$ for each $x_i \notin \{x_1, \ldots, x_j\}$.

A *computation* of $\mathcal{A}$ on W is a sequence $C_1 C_2 C_3 \ldots$ of configurations on W, where C_{i+1} is a next configuration of C_i. Automaton $\mathcal{A}$ *accepts* W if and only if there is a finite computation $C_1 C_2 C_3 \ldots C_n$, where the first configuration C_1 is of form $\langle s, w_1, \ldots, w_k\rangle$ where s is the start state, the state of the last configuration C_n is in F, and C_n has no next configuration. The *language* $L(\mathcal{A})$ *accepted by* $\mathcal{A}$ consists of the tuples of the strings $W \in \Sigma^*$ for which an accepting computation exists.

Let us now give an example of an automaton. A 1-tape automaton

$$\mathcal{A} = (\{1, 2, 3, 4, 5, 6\}, \{\texttt{a}, \texttt{c}, \texttt{g}, \texttt{t}, \texttt{[}, \texttt{]}\}, 1, \{6\}, T)$$

corresponds to the *contains_tata* predicate introduced earlier, when $T = \{\langle\langle 1,$ `scan 1 on not 1=']'`$\rangle, 1\rangle, \langle\langle 1, \epsilon\rangle, 2\rangle, \langle\langle 2,$ `scan 1 on 1='t'`$\rangle, 3\rangle, \langle\langle 3,$ `scan 1 on 1='a'`$\rangle, 4\rangle, \langle\langle 4,$ `scan 1 on 1='t'`$\rangle, 5\rangle, \langle\langle 5,$ `scan 1 on 1='a'`$\rangle, 6\rangle\}$. The transition diagram of automaton $\mathcal{A}$ is illustrated in Figure 3. Automaton $\mathcal{A}$ accepts all sequences containing the string '`tata`'.

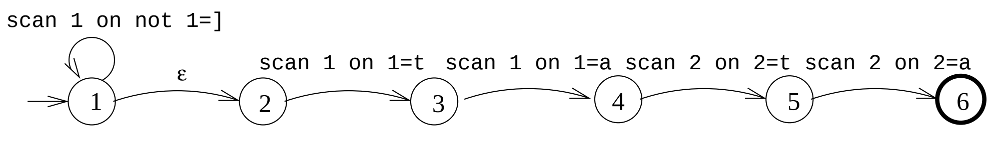


Figure 3: The contains-tata-automaton

In [10] it is proved that for each Alignment Declaration expression ϕ a corresponding automaton $\mathcal{A}_\phi$ can be constructed. Following [12, Theorem 2.3] the proof proceeds inductively on the structure of the expression ϕ by adding necessary states and transitions to the automaton. In the following, we give two examples of the phases of the construction; a summary of the corresponding structures of the automaton for every atomic sentence in the Alignment Declarations of AQL is given in Figure 4.

The construction of the automaton is carried out during the recursive, top-down parsing of the Alignment Declarations using two state variables: current state c and the next state to be added t. In the beginning, c has value 0 denoting the initial state, and t has value 1. For an expression of form `scan` $x_{i_1}, \ldots, x_{i_l}$ `on` ϕ, a new state t is added together with a transition $\langle\langle c, \texttt{scan } i_1, \ldots, i_l \texttt{ on } \phi'\rangle, t\rangle$, where ϕ' is obtained from ϕ by replacing each occurrence of x_{i_j} with i_j. The variables are then updated by assigning $c \leftarrow t$ and $t \leftarrow t + 1$ implying that successive statements generate successive states. A `rightscan`-expression is handled correspondingly.

A `repeat`-clause causes the current state c to be pushed into the stack while the expressions inside the loop are processed in an ordinary fashion. When the corresponding `*times` is encountered, a new variable p is introduced and given the value popped out of the stack. Then the empty transitions

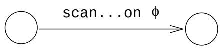
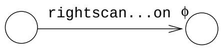
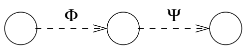
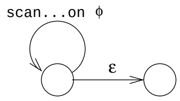
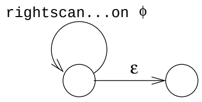
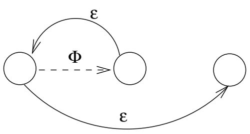
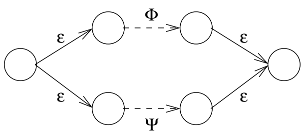
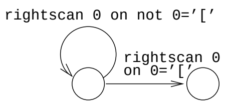
scan...on ϕ	
rightscan...on ϕ	
$\Phi\ \Psi$	
scan*...on ϕ	
rightscan*...on ϕ	
repeat Φ *times	
choose Φ | Ψ end	
reset x_0	

Figure 4: Corresponding structures

$\langle\langle s, \epsilon\rangle, p\rangle$ and $\langle\langle p, \epsilon\rangle, t\rangle$ are generated. Finally, the usual assignments $c \leftarrow t$ and $t \leftarrow t+1$ are carried out. The other expressions of the Alignment Declarations are processed as shown in Figure 4.

The last state added to the automaton is the accepting final state. Thus, the resulting multi-tape, two-way, nondeterministic finite state automaton has one start state, one final state, and no transitions out of the final state. Figure 5 shows the automaton constructed from the Alignment Declarations of the shuffle example (Example 5) presented earlier.

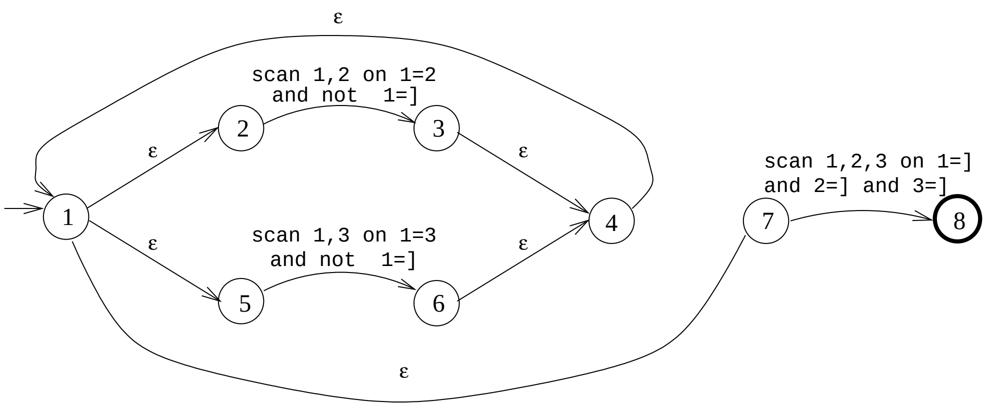


Figure 5: The shuffle-automaton

The compilation from Alignment Declarations to states and transitions takes place only once when the predicate is introduced. The states and transitions are saved to the disk using ordinary text file format where each transition is represented as a text row containing the following components: *[from_state] [to_state] [direction] [sliding_tapes] [acceptance_condition]*. Expressions *[from_state]* and *[to_state]* are integers referring to the states where the transition begins and ends, respectively. The *[direction]* is -1 for a slide to the right, 1 for a slide to the left, and 0 for no slide (empty transition). The expression *[sliding_tapes]* is a comma-separated list of tapes to be slided and the *[acceptance_condition]* is a prefix representation of the transition label, that is, either a `scan`- or a `rightscan`-statement. For example, a transition $\langle\langle 2,$ `scan 1,2 on 1=2 and not 1=']'`$\rangle, 3\rangle$ is represented as a following row:

```
2 3 1 1,2 A = 1 2 N = 1 ']
```

In the case of an ϵ-transition, the list of tapes and the acceptance condition are omitted. Note that the logical operators are abbreviated and the tape symbols are marked with an apostrophe. The logical expression is converted from infix to prefix form since it is represented as a tree in the actual data structures of the automaton.

The data structures are built when a query involving the predicate is invoked. The automaton is represented as an adjacency list organized into a table of states and lists of transition records. Each transition record consists of a state number representing the state the transition leads to, the direction of the slide (-1,0, or 1), a table of k bits, where bit i is 1, if tape i is to be slided, and 0 otherwise, a pointer to the root node of the tree representing the acceptance condition, and a pointer to the next transition record.

As shown in the algorithm below, the simulation of the automaton proceeds in a depth-first order using backtracking whenever necessary as is usual when simulating nondeterministic devices. If the simulation reaches an accepting final state, the corresponding predicate holds for the input strings.

current configuration $S = \langle q, n_1, \ldots, n_k \rangle \leftarrow \langle 0, 0, \ldots, 0 \rangle$
repeat
    for all transitions p of form $\langle\langle q, \phi \rangle, t\rangle$ **do begin**
        if $S \Vdash \phi$ **then begin**
            $push(S, p)$
            adjust tape heads $n_1, \ldots, n_k$ according to ϕ
            $q \leftarrow t$
        end if
    end for
    if $q \in F$ **or** stack empty **then** $end\_loop \leftarrow$ **true**
    else $(S, p) \leftarrow pop$
until *end_loop*
if $s \in F$ **then** accept input W
else reject

5 Results

We tested our system by building a small database consisting of biological nucleotide data from the EMBL Nucleotide Sequence Database [24] together with some artificial sequences constructed for testing purposes. The schema consisted of three relations: **sequence_table**, **organism_table**, and **comment_table**. Because of the limitations of the PostgreSQL we were able to use only rather short nucleotide sequences and perform limited queries, but the idea of letting the user construct her own predicates and using them as selection conditions seemed practical.

As an example of a tested query, consider the following session where we first introduce a 3-place predicate called $inversion(s_1, s_2, s_3)$, which becomes true if strings s_1 and s_2 are similar except for an inverted part of length $|s_3|$. The name of the database is bio1 which explains the `bio1=>` prompt.

```
bio1=> SELECT NEWPREDICATE('inversion','
bio1'>    scan* x,y on x=y and not x=]
bio1'>    scan* y,z on not z=]
bio1'>    scan y,z on z=]
bio1'>    repeat
bio1'>       scan x on not x=]
bio1'>       rightscan y,z on x=y and not z=[
bio1'>    *times
bio1'>    rightscan y,z on z=[
bio1'>    scan* y,z on not z=]
bio1'>    scan z on z=]
bio1'>    scan* x,y on x=y and not x=]
bio1'>    scan x,y on x=] and y=]');

NEWPREDICATE
------------
           0
(1 row)
```

The resulting zero here indicates the number of syntactic errors in the expression. Thus, the predicate was defined correctly. In the following query, this predicate is then used to select those pairs of DNA-sequences s_1 and s_2, for which $inversion(s_1, s_2, 10)$ holds. The numerical value 10 is represented by a string parameter consisting of ten arbitrary characters. Table 1 presented earlier contains the DNA-sequences of the relation **sequence_table**.

```
bio1=> SELECT S1.id, S1.sequence, S2.id, S2.sequence
bio1-> FROM sequence_table S1, sequence_table S2
bio1-> WHERE STRINGPREDICATE3('inversion',
             S1.sequence,S2.sequence,'aaaaaaaaaa');

id    |sequence                    |id    |sequence
------+----------------------------+------+----------------------
D12466|gatcaaataaagttataaaattgccttc|XYZ123|gatcaatattgaaataaaaattgccttc
XYZ123|gatcaatattgaaataaaaattgccttc|D12466|gatcaaataaagttataaaattgccttc
(2 rows)
```

As can be seen from this example, our system can be useful for detecting interesting relations between sequences both in molecular biology and other application areas. In addition, we think that the idea of user defined predicates adds to the expressivity and usefulness of ordinary data base systems.

We have made some tests for the performance of our system by constructing a stand-alone version of the parts of the program simulating the automata and measuring the execution time using different predicates and input strings. It turned out that the automaton based simulation of Alignment Declarations was slightly faster than evaluating the predicates in compiled SWI-Prolog [29], encoding strings as Prolog lists. In general, the efficiency of the string-predicates in AQL correspond to brute-force pattern matching. The main benefits of AQL lie in its simplicity compared to embedded programming languages, and its expressive power well beyond the LIKE operator of SQL and regular languages. Due to its declarative nature, queries written in AQL are amenable to optimisation. Indeed, the optimisation of queries on strings is in our view a very fruitful area of research.

We are currently working on a new prototype which extends the O_2 database management system and is also capable for generating new strings not contained in the database [13]. Another project is a WWW query system for a biological database [26] where an ordinary query form is extended with a possibility to use Alignment Declarations as an additional search condition. These new implementations are also free from the limitations of the PostgreSQL system so the length of the strings is not limited and the reliability of queries has improved.

6 Conclusions

Alignment Query Language is a language for expressing properties of strings which can be used for string querying in database applications. A prototype system was built as an extension to the PostgreSQL database management system. In the system, the expressions of the language are translated into a novel type of high level finite automata, and the evaluation is carried out by simulating these automata. The current implementation still lacks the possibility of *string restructuring*, or the generation of new strings not already in the database. Then for instance in Example 1 only string s_2 is required to be in the database, while its prefixes s_1 are generated during query evaluation. The theory on how this aspect of Alignment Calculus could be implemented has been developed in [7, 17, 18]. Our ongoing implementation projects include a new prototype that also allows for the aforementioned string restructuring, and a World Wide Web -based query system for a molecular biology database [13, 26].

References

[1] Abarbanel, R.M., Wieneke, P.R., Mansfield, D.A., et al. Rapid searches for complex patterns in biological molecules. *Nucleic Acids Research* 12, 1 (1984), 263–280.

[2] Bonner, A.J., and Mecca, G. Sequences, Datalog and Transducers. To appear in the *Journal of Computer and System Sciences (JCSS)* special issue on the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems.

[3] Codd, E.F. A relational model for large shared data banks. *Communications of the ACM* 13, 6 (1970), 377–387.

[4] Ganguly, S., and Noordewier, M. Proximal: a database system for the efficient retrieval of genetic information. *Computers in Biology and Medicine* 26, 3 (1996), 199–207.

[5] Ginsburg, S., and Wang, X. Pattern matching by rs-operations: towards a unified approach to querying sequenced data (extended abstract). In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (1992), pp. 293–300.

[6] Ginsburg, S., and Wang, X.S., Regular sequence operations and their use in database queries. *Journal of Computer and System Sciences* 56, (1998), 1–26.

[7] Grahne, G., Nykänen, M. Safety, translation and evaluation of Alignment Calculus. In *Proceedings of the First East-European Symposium on Advances in Databases and Information Systems (ADBIS'97)* (1997), pp. 295–304.

[8] Grahne, G., Nykänen, M., and Ukkonen, E. Reasoning about strings in databases. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (1994), pp. 303–312.

[9] Grahne, G., Nykänen, M., and Ukkonen, E. Reasoning about strings in databases. To appear in the *Journal of Computer and System Sciences (JCSS).*

[10] Hakli, R. *Extending a Database System with a String Language* (in Finnish). MSc thesis, Department of Computer Science, University of Helsinki, Finland, 1997.

[11] Helgesen, C., and Sibbald, P.R. PALM – a pattern language for molecular biology. In *Proceedings of the First International Conference on Intelligent Systems in Molecular Biology* (1993), pp. 172–180.

[12] Hopcroft, J.E., and Ullman, J.D. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley 1979.

[13] Kekkonen, M., Koskelainen, J., Niemi, J., et al. Design document for the BiO_2 system (in Finnish). Tech. rep., Department of Computer Science, University of Helsinki, Finland, 1997.

[14] Mecca, G., and Bonner, A.J. Sequences, Datalog and transducers. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (1995), pp. 23–35.

[15] Mehldau, G. *A Pattern Matching System for Biosequences*, Tech. rep. TR 91-21, Department of Computer Science, The University of Arizona, Tucson, Arizona, 1991.

[16] Mehldau, G., and Myers, G. A system for pattern matching applications on biosequences. *Computer Applications in the Biosciences* 9, 3 (1993), 299–314.

[17] Nykänen, M. *Querying String Databases with Modal Logic*, PhD thesis, Department of Computer Science, University of Helsinki, Helsinki, 1997.

[18] Nykänen, M. Using acceptors as transducers. To appear in *Third International Workshop on Implementing Automata (WIA'98).*

[19] Pistor, P., and Traunmueller, R. A database language for sets, lists and tables. *Information Systems* 11, 4 (1986), 323–336.

[20] Richardson, J. Supporting lists in a data model (a timely approach). In *Proceedings of the 18th International Conference on Very Large Data Bases* (1992), pp. 127–138.

[21] Searls, D.B. String variable grammar: a logic grammar formalism for the biological language of DNA. *The Journal of Logic Programming* 24, 1 & 2 (1995), 73–102.

[22] Seshadri, P., Livny, M., and Ramakrishnan, R. SEQ: A model for sequence databases. In *IEEE International Conference on Data Engineering* (1995), pp. 232–239.

[23] Seshadri, P., Livny, M., and Ramakrishnan, R. The design and implementation of a sequence database system In *Proceedings of the Twenty-second International Conference on Very Large Data Bases* (1996), pp. 99–110.

[24] Stoesser, G., Sterk, P., Tuli, M.A., et al. The EMBL Nucleotide Sequence Database. *Nucleic Acids Research* 25, 1 (1997), 7–13.

[25] Subramanian, B., Leung, T., Vandenberg, S., and Zdonik, S. The AQUA approach to querying lists and trees in object-oriented databases. In *IEEE International Conference on Data Engineering* (1995), pp. 80–89.

[26] Tuomimäki, L. *Querying Heterogeneous Sequence Databases in Computer Networks* (in Finnish). MSc thesis, Department of Computer Science, University of Helsinki, Finland, 1998.

[27] Wang, X. *Pattern Matching by Rs-Operations: Towards a Unified Approach to Querying Sequenced Data.* PhD thesis, University of Southern California, 1992.

[28] Wegner, L.M. Let the fingers do the walking: object manipulation in an NF^2 database editor. In *New Results and New Trends in Computer Science*, H.Maurer, Ed, Lecture Notes in Computer Science 555, Springer-Verlag, 1991, pp. 337–358.

[29] Wielemaker, J. *SWI-Prolog Reference Manual.* University of Amsterdam, The Netherlands, http://www.swi.psy.uva.nl/usr/jan/SWI-Prolog/Manual/Title.html, 1997.

[30] Wolper, P. Temporal logic can be more expressive. *Information and Control 56* (1983), 72–99.

[31] Yu, A., and Chen, J. *The POSTGRES95 User Manual.* University of California at Berkeley, California, http://www.eol.ists.ca/%7Edunlop/postgres95-manual/, 1995.