A Query Language for Large String Processing

Majed Sahli
Saudi Aramco
majed.sahli@aramco.com

Essam Mansour
Qatar Computing Research Institute, HBKU
emansour@qf.org.qa

Panos Kalnis
King Abdullah University of Science & Technology
panos.kalnis@kaust.edu.sa

ABSTRACT

With the advent of large string datasets in several scientific and business applications, there is a growing need to perform ad-hoc analysis on strings. Currently, strings are stored, managed, and queried using procedural codes. This limits users to certain operations supported by existing procedural applications and requires manual query planning with limited tuning opportunities. SQL can be used to analyze strings; but it provides limited support for native string operations because it is based on the relational model. It is unnatural to represent and access strings using relational algebra and tuple calculus.

This paper presents StarQL, a generic and declarative query language for strings. StarQL is based on a native string data model that allows StarQL to support a large variety of string operations and provide semantic-based query optimization. String analytic queries are too intricate to be solved on one machine. Therefore, we propose a scalable and efficient data structure that allows StarQL implementations to handle large sets of strings and utilize large computing infrastructures. Our evaluation shows that StarQL is able to express workloads of application-specific tools, such as BLAST and KAT in bioinformatics, and to mine Wikipedia text for interesting patterns using declarative queries. Furthermore, the StarQL query optimizer shows an order of magnitude reduction in query execution time.

1. INTRODUCTION

Strings are sequences of symbols. Textual content on the Internet and genomic sequences are examples of important strings [20]. Textual content holds information critical for corporations to understand consumer behaviour, banking firms to identify fraudulent activities, and governmental agencies to find criminal groups. Generally, string analysis involves a single long string (e.g., the human genome or the Wikipedia text) or large collections of short strings (e.g., DNA reads or words in a Wikipedia article).

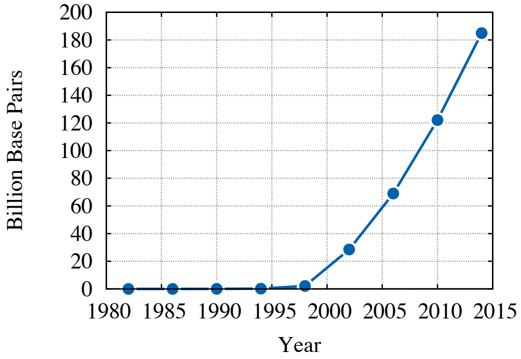


Figure 1: An example of the explosion of sequential data from bioinformatics as reported by NCBI[1].

More strings are being produced due to technological advances [22], stored due to low storage costs [6], and propagated due to collaboration and information sharing [9]. According to the National Center for Biotechnology Information [1] (NCBI), the size of the genomic sequences stored in the GenBank repository has doubled approximately every 18 months as shown in Figure 1. Ambitious projects that require large string analysis include the Cancer Genome Atlas [2] and the Square Kilometre Array Telescope [3].

In string analysis, multiple operations are executed to extract information. One of the most basic string operations is pattern matching. It is a core operation used in most string algorithms. However, even this core and basic operation can be simple, as in the case of exact matching; or more involved, as in approximate matching. Counting pattern matches leads to the problem of identifying frequent patterns, which in turn leads to the motif extraction problem. String operations have different semantics when dealing with a single string as opposed to multiple strings. For instance, matches within a single string provide insights different from those of a single match in several strings.

One could map a string to a relation and its symbols to attributes to analyze strings using SQL. However, strings are usually large and vary in size, and the order of their symbols matters. Alternatively, considering a whole string as a single attribute is not a feasible solution because string operations require primitives not served by SQL's `LIKE` operator, such as repeated patterns and common substrings.

[1] ftp://ftp.ncbi.nih.gov/genbank/gbrel.txt
[2] http://cancergenome.nih.gov
[3] https://www.skatelescope.org

Attempts to extend SQL with string operations do not provide native and generic string support because they are limited by their original data models [32, 15]. For example, PiQL [31] and Sequence Datalog [21], attempted to support string queries by extending the relational data model and Datalog, respectively. These languages are constrained by being application-specific and by providing very basic operations that make them impractical.

Hence, procedural codes are currently used to analyze string datasets. For example, BLAST [1] is used for matching, where it finds regions of local similarity between biological sequences. Another example is KAT [4], a k-mer counting tool used to analyze substring frequency spectra. To analyze strings, users manually move data and run different applications or use pipeline systems to automate this process [5].

This paper presents a declarative query language for strings, called StarQL. StarQL provides native support for string operations and generic primitives that cover users' needs in different applications. While StarQL is generic and works for any string and application, most examples use DNA sequences for ease of exposition.

Example 1. *Suppose we have a DNA sequence S and we need to know if the pattern* `GGTGC` *is frequent in S or not. Assume a pattern is frequent if it appears 5 times or more in the string and that matches need not be exact as shown in Figure 2.*

SQL can be used to count matches of a candidate pattern but matches are limited to the capabilities of the `LIKE` *operator. Using weighted scoring matrices in the case of DNA sequence similarity is not an option. Using procedural code, we can implement a string scanner and a distance function to find and count matches. However, hard-coded queries contradict with the essence of ad-hoc analysis.*

In StarQL, finding out the number of matches for `GGTGC` *in S is achieved by the following query.*

```
SELECT COUNT(MATCH(dna, "GGTGC", user_dist(2))); (1)
```

Finding out if a pattern is frequent or not is a simple task. A more involved and realistic task is to extract all frequent patterns in a string. Such patterns are referred to as *motifs* and they require counting the matches for a large number of candidate motifs. SQL cannot handle candidate motifs generation so users need procedural code that implements Apriori-based or pattern-growth algorithms to extract motifs. However, extracting motifs in StarQL is equivalent to the following simple and customizable query.

```
SELECT RMOTIFS(dna, freq=100, \
  minlen=3, maxlen=9, edit(2));          (2)
```

The StarQL language is based on our native string model, called StarDM, that supports a large variety of string operations and is extensible to support application-specific operators. To escape the procedural code trap, StarQL supports user-defined functions. For instance, matching is a universal string operation but different applications use different matching criteria. In Example 1, `user_dist` is a user-defined distance function used when matching DNA sequences given a weighted scoring matrix. Moreover, the native string model allows StarQL queries to be smartly rewritten based on their operation semantics to reduce execution time. The paper also proposes a scalable and efficient data structure that is suitable for parallel query processing and handling large sets of strings.

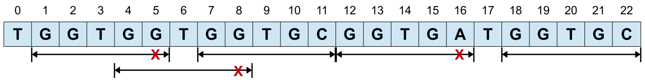


Figure 2: Example string S over DNA alphabet $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$. Matches for `GGTGC` are indicated, allowing one mismatch and overlapping matches.

In summary, our contributions are the following.

- We introduce StarDM, a native string model that deals with a string as a sequence of symbols, and the input/output of any string operation as a set of strings. Moreover, we classified string operations into categories that support a wide range of applications.
- We develop StarQL, a declarative query language for strings and provide a semantic-based optimization for StarQL queries.
- We prupose StarIN, a scalable and efficient data structure for realizing StarDM and implementing StarQL. StarIN avoids the limitation of traditional string indexing techniques by striking a balance between preprocessing time, index size, and parallel support.
- We conduct comprehensive experiments on real datasets, namely Wikipedia text and the Human genome DNA sequence. We show how StarQL expresses a BLAST workload and gets optimized to achieve significant reduction in execution time. StarIN allows our proof-of-concept system to efficiently scale to thousands of compute nodes on a supercomputer.

The rest of this paper is organized as follows. In the next section, we introduce the StarDM data model. In Section 3, we introduce the syntax and semantics of StarQL, our query language. Section 4 presents our StarIN data structure. We then evaluate our language in Section 5. Section 6 summarizes the related work before we conclude the paper in Section 7.

2. NATIVE STRING SUPPORT

In this section, we introduce our StarDM string model, in which a string is a sequence of symbols, and the input/output of any string operation is a set of strings. Using StarDM, we define operations that natively support string queries based on the set theory. The StarDM closure property along with relevant metadata allow for expressing and solving complex string queries. This native support enriches our query language with expressiveness and advanced query optimization based on the semantics of the operations, as we discuss in Section 3. Moreover, the simplicity of StarDM allows for efficient and scalable implementations, as we highlight in Section 4.

2.1 StarDM: A Native String Model

Evidently, string operations have one or more strings as input and produce one or more strings as output. Therefore,

[4] http://www.tgac.ac.uk/KAT/

[5] https://seqware.github.io/docs/6-pipeline/

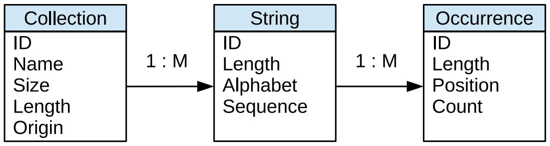


Figure 3: The metadata of StarDM collections.

a query language for strings will have to deal with collections of strings. Theoretically, a set is an ideal representation for a collection of strings because it allows (i) no strings (the empty set), (ii) one string (a singleton), or (iii) many different strings in a collection.

Our native string data model is called StarDM. In StarDM, we consider strings as sequences of symbols, grouped into collections. A string of zero symbols is an empty string, and a collection of zero non-empty strings is an empty collection. Depending on how string collections are generated, they could consist of several long strings or many short strings.

In StarDM, a collection of strings has a certain alphabet. Let Σ be an alphabet, a finite set of elements. The elements of Σ are called letters, characters, or symbols. A typical example of alphabets is the English alphabet of 26 letters. In biology, the DNA alphabet consists of the four characters $\{\texttt{A},\texttt{C},\texttt{G},\texttt{T}\}$. A string $\texttt{S}$ over Σ is a finite concatenation (ordered sequence) of symbols taken from Σ. The length, or size, of a string $\texttt{S} = a_0a_1 \ldots a_{n-1}$, denoted by $|\texttt{S}|$ is n, the number of its elements from the alphabet with repetitions. So, the length of $\texttt{S} = \texttt{ACAAT}$ is 5. Extending these properties to a collection of strings, size and length become distinct. The size of a collection is the number of strings in it, and its length is the summation of the lengths of the strings. Similarly, the alphabet of a collection is the set union of the alphabets of its strings.

The i-th element of a string $\texttt{S}$ of length n is denoted by $\texttt{S}[i]$ and i is its position in $\texttt{S}$, where $0 \leq i < n$. We will assume that $\texttt{S}[n]$ contains the special end character $\$ \notin \Sigma$, which cannot occur anywhere but in the end of a string. We denote by $\texttt{S}[i,j]$ a substring of $\texttt{S}$ defined as follows.

$$\texttt{S}[i,j] = \begin{cases} \texttt{S}[i]\texttt{S}[i+1]\ldots\texttt{S}[j] \text{ of length } j-i+1 & j > i \\ \texttt{S}[i] \text{ of length 1} & j = i \\ \$ \text{ of length zero} & j < i \end{cases}$$

Given a string $\texttt{S}$ and the three strings (possibly empty) $\texttt{p}$, $\texttt{s}$, and $\texttt{t}$ such that $\texttt{S} = \texttt{pst}$; $\texttt{p}$ is a *prefix* of $\texttt{S}$, $\texttt{s}$ is a *substring* of $\texttt{S}$, and $\texttt{t}$ is a *suffix* of $\texttt{S}$. We say that the string $\texttt{s}$ of length m occurs in $\texttt{S}$ at position i, or that the position i is a match for $\texttt{s}$ in $\texttt{S}$, if $\texttt{s} = \texttt{S}[i, i{+}m{-}1]$, or $\texttt{s}$ is similar to $\texttt{S}[i, i{+}m{-}1]$ for some similarity measure and threshold. Given a distance function, similarity between two strings increases as the distance between them decreases.

StarDM maintains metadata about a collection of strings, a string, and an occurrence of a specific string as shown in Figure 3. This metadata assists in answering statistical string queries, filter collections, ands optimize execution. In contrast, PiQL [31] maintains metadata about the application, for example, protein structure. StarDM maintains several properties as follows. The `ID` is a unique identifier of a collection, a string, or an occurrence. The `NAME` is a textual description of a collection. The `SIZE` is the number of strings in a collection. The `LENGTH` is the total length of all the strings in a collection or the length of a string. The `ALPHABET` of a collection is the set of symbols used in all its strings. For example, if a collection has two strings, $\{\texttt{aba, cba}\}$, then the alphabet of the collection is "abc". The `ORIGIN` of a collection is the path or query that resulted in the creation of the collection. The transformations of strings from one collection to another can be tracked using this attribute. It is worth noting that some metadata is stored (e.g., collection name) and some is computed (e.g., occurrence position).

Table 1: Supported set operations; where A, B, and C are collections of strings.

Operator	Procedural form	Algebraic
Intersection	C = intersection(A, B)	A ∩ B
Union	C = union(A, B)	A ∪ B
Difference	C = difference(A, B)	A \ B

Table 2: Supported matching operations; where A, B, and C are collections of strings; *t* is a binary for *all* or *any* and *d* is a distance metric threshold.

Operator	Procedural form	Algebraic
Exact	C = exact(A, B, t)	A E B
Approximate	C = approx(A, B, t, d)	A $A_{all\|any,d}$ B
Regex	C = regex(A, B, t)	A $R_{all\|any}$ B

2.2 String Operations

Our StarDM model supports and extends string algebra operations suggested by different works [23, 5]. The following operations are chosen to support most string queries required in different applications, such as matching and frequent patterns. They are not too specific, as genomic assembly, nor too generic, as simple concatenation. In StarDM, we classify string operations into six categories, namely set operations, matching, filtring, extraction, generation, and aggregation.

2.2.1 Set operations

The set operations on string collections are identical to their counterparts in the set theory (see Table 1). Given A and B two collections of strings, the intersection operation outputs the set of strings common to A and B. Similarly, the union operation results in the set of strings in A or B, whereas the difference operation filters B out of A. For example, if A = $\{s_1, s_2, s_3\}$ and B = $\{s_2, s_3, s_4\}$ then A ∩ B = $\{s_2, s_3\}$; A ∪ B = $\{s_1, s_2, s_3, s_4\}$; and A \ B = $\{s_1\}$.

2.2.2 Matching operations

Matching is the basic textual problem and one of the most used string operations. An exact matching operator may return a boolean value indicating the existence of a match or it may return the number of matches, if any. A more involved requirement is to return the positions of the matches in the searched string. Variations of this operator include approximate matching and evaluating regular expressions.

In Table 2, given A and B two collections of strings, the

Table 3: Supported filtering operations; where A, B, and C are string collections; and d is a distance metric threshold.

Operator	Procedural form	Algebraic
Prefix Filter	C = is-prefix(A, B, d)	A P_d B
Suffix Filter	C = is-suffix(A, B, d)	A S_d B
Substring Filter	C = contains(A, B, d)	A C_d B

exact match operation outputs the set of substrings from A that match a string in B. Similarly, the approximate match operation finds matches of B in A given a distance metric threshold. For most flexible matching, regular expressions are used. In this case, B is a set of regular expressions and the operation outputs the set of substrings from A that match expressions in B. The *all* and *any* variants of the operators dectate the semantics over the two collections. In particular, an *all*-match requires matching against every string in B, whereas an *any*-match requires at least one string in B that matches. Specifically, matching over two collections is logically an operation over the Cartesian product of these collections. For example, if A = {TEST, BEST, ESTATE} and B = {TT} then A E B = ∅; A $A_{ham(1)}$ B = {TE, ST, AT}; and A R {T*T} = {TEST, TAT}.

2.2.3 Filtering operations

Filtering a collection of strings can be thought of as a form of set difference operation with a condition. In Table 3, given A and B two collections of strings, we filter the left operand according to conditions that involve the right operand. The prefix and suffix filter operations output the set of strings in A that have a prefix or a suffix from B within a distance. The substring filter is more generic, where A is filtered by matching strings from B regardless of position. Given two string collections, filters are applied on the Cartesian product of these collections. Moreover, collections can be filtered according to metadata properties, such as length, size, and alphabet. For example, if A = {TEST, BEST, ESTATE} and B = {TT} then A $P_{ham(1)}$ B = {TEST}; A $S_{ham(1)}$ B = {TEST, BEST, ESTATE}; and A $C_{ham(0)}$ B = ∅.

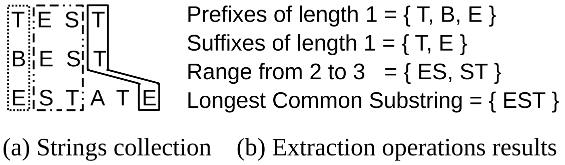
2.2.4 Extraction operations

Extraction operations allow for processing prefixes, suffixes, or ranges within a collection of strings as shown in Table 4. Given a collection of strings A, we allow extracting prefixes with a range of lengths, between p_1 and p_2. If a string is shorter than p_1, it is not considered a valid prefix and thus eliminated altogether. Suffixes are handled in the same manner without the need to know the length of strings beforehand. The range extraction operation takes absolute positions and extracts the substrings. If a string is shorter than p_2, no substring would be extracted. The longest common substring operation (LCS) outputs a collection of a single string, the longest common substring.

Figure 4 shows an example of extraction operations over a sample collection. If we were to extract prefixes (or suffixes) of length 5 from the example strings then there would be only one. Moreover, the example has 2 unique suffixes of length 1. The output collection maintains the unique strings with counts of appearances. This will be detailed when the data structures are discussed in Section 4.1. In this case, ”T” will have a count of 2 because there were 2 “T” suffixes. The range example has a similar case for “ES”.

Table 4: Supported extraction operations; where A and C are collections of strings and $p_1 \leq p_2$ are integer representing lengths for prefixes and suffixes or positions for ranges.

Operator	Procedural form	Algebraic
Prefixes	C = prefixes(A, p_1, p_2)	$\prod_{(START,[p_1,p_2])}$ A
Suffixes	C = suffixes(A, p_1, p_2)	$\prod_{(END-[p_1,p_2],END)}$ A
Range	C = range(A, p_1, p_2)	$\prod_{(p_1,p_2)}$ A
LCS	C = lcs(A)	$\bigwedge$ A


T E S T
B E S T
E S T A T E
Prefixes of length 1 = { T, B, E }
Suffixes of length 1 = { T, E }
Range from 2 to 3 = { ES, ST }
Longest Common Substring = { EST }

(a) Strings collection (b) Extraction operations results

Figure 4: Extraction operations example.

2.2.5 Generation operations

Several string operations generate new strings [19, 26, 27]. In Table 5, we defined operators for motifs and k-mers, both highly used when analyzing strings. The common motifs problem is to find frequent patterns that appear in a number of strings. Given A a collection of strings, the common motifs operation outputs the set of strings of length l_1 to l_2 that have matches in at least f strings from A. A similar but more challenging problem is the repeated motifs problem, where a pattern is frequent if it appears a certain number of times in a single string. In this case, the output is the union of repeated motifs in each string in A. The set of k-mers contains the substrings of length k. Our k-mer operation finds the set of strings of a certain length that appear exactly in at least one string from A.

2.2.6 Aggregation operations

Aggregate operations are deterministic and result in a single scalar value. It is important to know the size of a string collection and the length of strings in a collection. In collections that result from other operations, each string will also have a count of appearances in the collection. Table 6 shows the aggregate string operations. The length of a collection is the summation of the its member string lengths. The size is the number of these nonempty strings. Operations such as average, sum, maximum, and minimum can operate on outputs of aggregate operations.

2.3 Extensibility

The list of operations discussed in Section 2.2 is not exhaustive. Given our unified model for strings, any operator that takes one or more collections as input and outputs a collection is a valid operator. Our categories of string operations can be extended with new operations that have application-specific logic. For example, in bioinformatics, genome assembly is an operation that takes a collection of short reads and outputs a collection of a single long sequence. Assembly can be added to the existing string gen-

Table 5: Supported generation operations; where A and C are string collections, $l_1 \leq l_2$ are lengths, f is a frequency threshold, and d is a distance threshold.

Operator	Procedural form	Algebraic
Common Motifs	C = cmotifs(A, l_1, l_2, f, d)	$\bigotimes_{l_1,l_2,f,d}$ A
Repeated Motifs	C = rmotifs(A, l_1, l_2, f, d)	$\bigoplus_{l_1,l_2,f,d}$ A
K-mers	C = kmers(A, l_1)	$\bigodot_{l_1}$ A

Table 6: Supported aggregation operations; where A is a collection of strings, A.a is a specific string in A, and I is a numeric result.

Operator	Procedural form	Algebraic
Length	I = length(A)	$\coprod$ A
Size	I = size(A)	|A|
Count	I = count(A.a)	CA.a

eration operations. User-defined functions are incorporated through dynamic loading (e.g., by linking a shared library).

Another way to extend StarDM functionality is to define functions that are used within existing operators to embed application logic. For example, Table 2 shows that matching operations take as input a string distance function. This function takes as input two strings and returns a scalar value indicating the disimilarity between the input strings. Typical distance functions include the Hamming and Edit distance metrics. However, in bioinformatics, users may require the use of application-specific metrics, such as weighted matrices for DNA similarity scoring.

3. A DECLARATIVE QUERY LANGUAGE

This section introduces StarQL, a declarative query language for strings. StarQL is based on our StarDM model. In StarQL, queries are categorized according to their applicability and results to administrative and analytic queries. Administrative queries are used to manage the string database and its string collections. Analytic queries are used to extract information. StarQL provides novel query optimization techniques based on the semantics and categories of the query operations. StarQL adopts a declarative SQL-like syntax, which is easy to understand. Figure 5 shows an abstract BNF of some of the StarQL constructs. Complex string analysis can be performed by easily nesting different constructs to form queries. Next, we discuss StarQL constructs, grouped according to functionality, and give examples of their usage.

3.1 Query Constructs

3.1.1 Administration

The IMPORT utility provides a simple way for users to load and index strings. Strings can be imported from the file system or from query results. The input parameters of an import command are: the path of the strings or the query that generates them, and the name of the new collection that will be created. The original path is not required for further operations on the imported strings but a pointer to the original path is kept for reference. The newly created collection

```
<query>      := <query> | <import> ; | <select> ; |
                <delete> ; | <aggregate> ;
<identifier> := $PATH$ | $ID$
<delete>     := DELETE <identifier>
<dist>       := HAMMING($int$) | EDIT($int$) | USER($int$)
<length>     := MINLEN $int$ MAXLEN $int$ | LEN $int$
<motif-type> := RMOTIFS | CMOTIFS
<motif-ops>  := FREQ $int$ <length> <dist>
<slct-cls>   := <identifier> | <generator> | <extractor>
<motifs>     := <motif-type>(<slct-cls> <motif-ops>)
<kmers>      := KMERS(<slct-cls> <length>)
<generator>  := <motifs> | <kmers>
<type>       := ALL | ANY
<matches>    := EXACT(<slct-cls> <type> <slct-cls>) |
                EXACT(<slct-cls> "$PATTERN$") |
                REGEX(<slct-cls> <type> <slct-cls>) |
                REGEX(<slct-cls> "$PATTERN$") |
                APPROX(<slct-cls> <type> <slct-cls> <dist>) |
                APPROX(<slct-cls> "$PATTERN$" <dist>)
<range>      := RANGE(<slct-cls> FROM $pos1$ TO $pos2$)
<prefixes>   := PREFIXES(<slct-cls> <length>)
<suffixes>   := SUFFIXES(<slct-cls> <length>)
<extractor>  := <prefixes> | <suffixes> | <range> | <matches>
<prefix>     := PREFIX(<slct-cls> <slct-cls> <dist>) |
                PREFIX(<slct-cls> "$PATTERN$" <dist>) |
<suffix>     := SUFFIX(<slct-cls> <slct-cls> <dist>) |
                SUFFIX(<slct-cls> "$PATTERN$" <dist>) |
<substring>  := SUBSTR(<slct-cls> <slct-cls> <dist>) |
                SUBSTR(<slct-cls> "$PATTERN$" <dist>) |
<filter>     := <prefix> | <suffix> | <substring> |
                <matches> | <metadata>
<sort>       := ORDER BY <metadata>
<whr-cls>    := <whr-cls> | <filter> | LIMIT $int$ | <sort>
<select>     := SELECT <slct-cls> [AS <identifier>]
                [WHERE <whr-cls>]
<import>     := IMPORT <identifier> AS <identifier> |
                IMPORT <select> AS <identifier>
```

Figure 5: Abstract BNF for StarQL.

is named and given a unique ID. The number of strings and the total length of all the strings are saved as collection properties. The IMPORT construct also allows for duplicating an existing collection. The DELETE utility removes a previously existing collection from the database system. Any indexes and metadata can be purged but the origin of a deleted collection (path or another collection) is not affected. For example, a user imports a dataset of human DNA shotgun reads from disk by running the following query.

```
IMPORT "/datasets/shotgun/human" AS hdna;          (3)
```

3.1.2 Matching

The EXACT matching command finds exact matches of a query pattern in a collection. The output of an exact match operation is either a collection of one string, the matched pattern along with the number of exact matches, or an empty collection if no matches are found.

The APPROXIMATE matching command finds matches within a certain distance threshold from the query pattern. The distance function can be hamming distance or edit distance, for example. The search process is similar to that of exact matching except that we allow differences between matches and patterns within a user-defined threshold. The result is a collection of as many unique substrings that match with their respective counts in the original collection.

The REGEX matching command finds substrings that match a regular expression pattern. The capabilities of a regular expression matching command are vital for several string applications. Regular expressions can be evaluated using deterministic finite automaton (DFA) over the strings. The result is a collection of as many unique substrings that match with their respective counts in the original collection. For example, assume a user needs to find matches of the regex expression "AC..CA" in the previously imported collection hdna. The query to find and save the results is written as follows. Notice that a collection of one element can be supplied inline for convenience.

```
IMPORT (SELECT REGEX(hdna, "AC..CA")) AS re;     (4)
```

3.1.3 Extraction

The PREFIXES extraction command extracts all the unique prefixes in a collection of strings. The input is a collection along with the desired prefix length range. The output is a collection of all unique prefixes within required length range. The SUFFIXES extraction command is similar to PREFIXES but for suffixes. The RANGE extraction command extracts all the unique substrings that exist at a specific position in a collection of strings. The input is a collection along with the desired start and end positions. The output is a collection of all unique substrings that exist at the specified positions in all the strings.

In our running example, a user may be interested in the different substrings of length 2 that exist between a pair of "AC". In this case, a range extraction query could be used as follows. If re consisted of {ACCCAC, ACGTAC, ACTTAC, ACATAC}, then the output would be {CC, GT, TT, AT}.

```
SELECT RANGE(re, FROM 2 TO 3);              (5)
```

3.1.4 Generation

The following commands generate new strings from existing collections. The RMOTIFS command finds all the repeated motifs supported by at least one string in the collection operated on. The input is a collection along with the desired motif properties; namely, minimum length, maximum length, and frequency threshold. The CMOTIFS command finds all the common motifs supported by a user specified number of strings in a certain collection. The input is a collection along with the desired motif properties. In addition, motifs can be exact or approximate.

The K-MERS generation command finds all the unique k-mers in a collection of strings. The input is a collection along with the desired substring length k. The output is a collection of the unique k-mers from all the strings in the collection. For example, the k-mers of length 3 from the previously saved collection re are { AAC, CCC, CCA, CAC, ACG, CGT, GTA, TAC,ACT, CTT, TTA, ACA, CAT, ATA }. To generate these k-mers, the following query is used.

```
SELECT KMERS(er, LEN=3);                    (6)
```

3.1.5 Filtering

Existing collections can be filtered according to matches or metadata properties. The PREFIX filtering command finds the strings that share a certain prefix; either exactly or approximately. The input is a collection of strings, a prefix pattern, and a distance function and threshold. The output is a collection of strings with matching prefixes. The SUFFIX filtering command finds the strings that share a certain suffix; either exactly or approximately. The input is a collection of strings, a suffix pattern, and a distance function and threshold. The output is a collection of strings with matching suffixes. The SUBSTRING filtering command finds the strings that share a certain substring; either exactly or approximately. The input is a collection of strings, a substring pattern, and a distance function and threshold. The output is a collection of strings with matching substrings. The LENGTH filtering command finds strings of a certain length range. The input is a collection of strings and a length range. The output is a collection of strings of the specified range.

Continuing our running example, assume the user is interested in the k-mers of length 3 that include the 2 characters between the pair of "AC" in the regular expression matches, r = {CC, GT, TT, AT}. This is accomplished using the following query. The resulting filtered k-mers are {CCC, CCA, CGT, GTA, CTT, TTA, CAT, ATA}.

```
SELECT KMERS(er, LEN=3) AS k \
  WHERE SUBSTRING(k, r);                    (7)
```

3.2 User-Defined Functions

StarQL supports user-defined functions, which can be used to add new operations or introduce application-specific logic using routines executed by other functions. For example, one of the main routines used in string operations is the distance function. However, it is difficult to include every possible distance function. Therefore, user-defined functions are used to augment StarQL distance functions and support new metrics.

In bioinformatics, weighted matrices are used to measure the similarity of DNA sequences. The weights are based on a particular theory of evolution, where certain symbols are more likely to mutate into other symbols. Conventionally, rows and columns in a weighted matrix are associated with alphabet symbols. For instance, using the example matrix below, the distance between AACT and GAAT is $2+0+1+0 = 3$, whereas using Hamming distance it is 2.

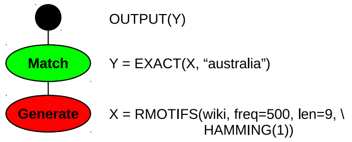
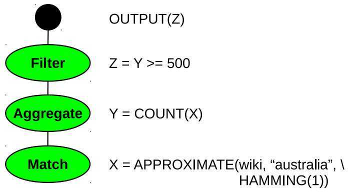
$$W = \begin{array}{c|cccc} & \mathtt{A} & \mathtt{C} & \mathtt{G} & \mathtt{T} \\ \hline \mathtt{A} & 0 & 1 & 2 & 1 \\ \mathtt{C} & 1 & 0 & 3 & 2 \\ \mathtt{G} & 2 & 3 & 0 & 3 \\ \mathtt{T} & 1 & 2 & 3 & 0 \end{array}$$

3.3 String Query Optimizations

It is not always straightforward to optimize string queries. This is because the order of executing string operations could change the final results. Because StarQL is based on StarDM, we can optimize string queries not only based on the cost of each string operation but also based on the semantics of these operations. To start, we find an execution order that preserves the query logic while generating less intermediate data. For example, if a query involves multi-

Table 7: An example for a schema-based optimization in StarQL. This query checks if the string "australia" is a repeated pattern in the Wikipedia collection.

```
SELECT RMOTIFS(wiki,freq=500,len=9,HAMMING(1)) as m WHERE EXACT(m,"australia");
```

PLAN A (naïve)	PLAN B (optimized)
 OUTPUT(Y) Match Y = EXACT(X, "australia") Generate X = RMOTIFS(wiki, freq=500, len=9, \ HAMMING(1))	 OUTPUT(Z) Filter Z = Y >= 500 Aggregate Y = COUNT(X) Match X = APPROXIMATE(wiki, "australia", \ HAMMING(1))

ple matching operations, one of which is against the longest common substring, then finding the longest common substring first reduces intermediate results and the search space for subsequent matching operations.

To ensure correctness, we use the syntax of StarQL to determine the semantics. In particular, the final output is always a subset of the operation after a `SELECT` keyword. Conditions after a `WHERE` keyword are interpreted from left to right, but not necessarily executed in this order. Using this convention, how StarQL interprets queries is clear to users. Only valid plans are compared internally to optimize efficiency. For instance, the following nested query extracts suffixes of length 3 from the repeated motifs found in Wikipedia.

```
SELECT SUFFIXES(RMOTIFS(wiki, \
  MAXLEN=20, FREQ=1000), LEN=3);          (8)
```

StarQL's query plans are based on the categories of StarQL operations, where execution plans start with operations that generate or extract strings, then apply operations that filter, limit or sort these strings. Furthermore, StarQL enables semantic-based optimizations, where query operations can be rewritten using other operations. While maintaining query logic, semantic-based optimizations aim at reducing computational complexity, intermediate results, and execution time.

Algorithm 1 describes the StarQL query optimizer. First, a query is tokenized and tokens are assigned to categories. Then, operations that can be reordered to minimize intermediate results without affecting semantics are shuffled. For instance, we do not push filter operations into generate operations to keep semantics intact. Finally, StarQL re-writes query operations based on their semantics. This is possible because some StarQL operations can be expressed in terms of other operations. For instance, approximate matches are used to find motifs. The optimizer takes advantage of such cases to find an equivalent set of operations with less cost given the data and the user parameters.

Consider for example, a user query that checks if the string "australia" is a repeated pattern in a Wikipedia collection. A naïve plan starts by generating the repeated motifs. Then the intermediate results will be filtered by the string "australia". The StarQL optimizer will rewrite this query in terms of counting approximate matches for the pattern we filter at the end. Table 7 shows the two plans. Plan A uses more resources and generates excessive intermediate results whereas Plan B eliminates the expensive repeated motifs operator and replaces it with a count of approximate matches.

Algorithm 1 StarQL Query Optimizer Algorithm

```
procedure OPTIMIZE($Q$)
    $T \leftarrow$ tokenize($Q$)
    $T.initial = Q.first\_token$
    while $T.initial$ $NOT$ $collection\_id$ do
        $T \leftarrow$ tokenize($T.initial$)
    end while
    $T.filters \leftarrow Q.last\_token$
    if $T.filters$ $in$ $MATCH$ $||$ $EXTRACT$ then
        $push\_down(T.filters)$
    end if
    $S \leftarrow detect\_semantics(T)$
    $P \leftarrow S.optimal\_plan$
    return $P$
end procedure
```

3.4 Scenarios and Workloads

Consider scenarios of users performing analytical tasks on strings; such as finding frequent patterns, matching, counting, and generating k-mers. Queries can be executed in different orders, nested, saved, and further processed. We next describe two different scenarios and discuss their expected workloads.

3.4.1 Literary scenario

Given the English text of Wikipedia, a librarian is curious about how writers start articles. She wants to explore the letters that frequently appear consecutively in the prefaces of Wikipedia pages. In StarQL, the k-mers operation finds symbols that appear consecutively and reports their frequencies. To find k-mers from the beginnings of articles, the k-mers are generated from the prefixes of the texts or are checked to exist in a specific range. To explore, the librarian may need to increase k-mers lengths as long as frequency is high. Given StarQL indexing, these queries are of low workloads. The following two StarQL queries find the top 20 k-mers of length 4, 8, or 16, that appear in the first 100 characters more than 200 times across the dataset.

```
IMPORT SELECT PREFIXES(wiki, LEN=100) \
  AS pref;
SELECT KMERS(pref, LEN=(4, 8, 16)) \        (9)
  AS k WHERE COUNT(k) > 200 ORDER BY \
  COUNT(k) DESC LIMIT 20;
```

The first query translates to creating a new collection in the database from the results of the prefix extractor. The second query is run by executing the k-mers operation 3 times, one time for each length, accumulating results after filtering on the count while keeping the top 20 results only.

3.4.2 *Bioinformatics scenario*

Given DNA and protein datasets, a biologist needs to find the patterns that are frequent within every genomic sequence and at the same time common between different sequences. Such patterns have potential functional importance and can be used to draw conclusions across species or to find mutations within a species. The task translates to generating repeated motifs, generating common motifs, then finding the motifs that are both repeated and common. The workload of such task is high due to the combinatorial search space over the string's alphabet. The following StarQL query is an example for this scenario.

```
SELECT EXACT(CMOTIFS(dna, LEN=10, \
  HAMMING(2), FREQ=10), RMOTIFS(dna, \   (10)
  LEN=10, EDIT(2), FREQ=10000));
```

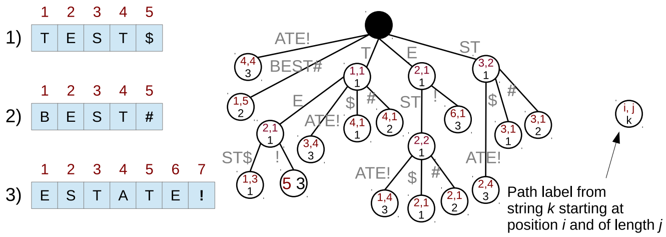
Without semantic-based optimizations, the query plan proceeds as follows. First, the inner sub-queries are evaluated and their results are saved as temporary collections. Then, the exact matching operator is evaluated to find the intersection between the repeated and common motifs. However, the StarQL optimizer uses the semantics of the operations to reduce the complexity of the query. In this case, we first generate repeated (or common) motifs, then count the matches to check if they are also common (or repeated).

4. A SCALABLE DATA STRUCTURE

This section introduces a scalable and efficient data structure for implementing the StarDM model and StarQL operations. Our data structure provides native string indexes inspired by the generalized suffix trees (GST) [17]. GST is the *de facto* index for a collection of strings and a powerful full-text index that is essential for string operations against a set of strings [17]. However, the GST index has several scalability limitations in terms of number of strings and parallel processing. This section highlights these limitations and introduces our data structure, which finds a balance between preprocessing time, index size, and parallel support. We aim at enabling parallel support for StarQL implementations to utilize large infrastructures and support large sets of strings. Moreover, the section discusses the implementation of StarQL operations based on our novel structures.

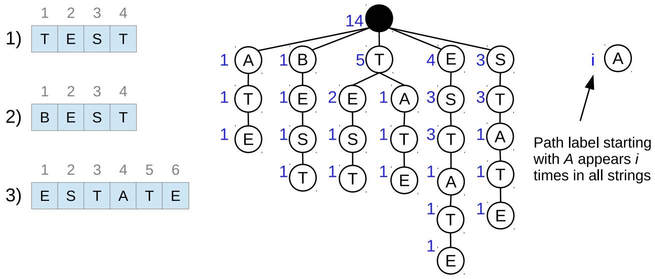
4.1 Limitations of Generalized Suffix Trees

A Generalized Suffix Tree (GST) is a suffix tree index of all suffixes of a given set of strings [17]. Consider the set of strings S = { TEST, BEST, ESTATE }. Figure 6 shows a GST of S. In average, the number of nodes in a suffix tree is $2n$, where n is the cumulative length of the indexed strings. However, each node stores at least 3 integers to identify its path label (the starting position, the label length, and a string identifier). For example, the first child of the root in Figure 6 indicates the starting position 4, the length 4, and the string 3 in order to recover the path label "ATE!". A hefty constant is hidden when describing the space requirement of a GST as linear. The size of a GST is orders of magnitudes more than the total size of the indexed strings. For large string processing, it is hard to deal with a huge and indecomposable structure.



Path label from string *k* starting at position *i* and of length *j*

Figure 6: Generalized suffix tree example. The labels are shown for simplicity but not stored in GST implementations.



Path label starting with *A* appears *i* times in all strings

Figure 7: Example proposed index as opposed to Figure 6.

A GST is not scalable because it requires as many distinct terminating symbols as the number of indexed strings. This is infeasible considering the number of strings in a collection can be in the order of millions. Therefore, the alphabet size will be huge. Alternatively, GST leaf nodes will have to store a list of strings that share their path labels; adding to the space requirement. Moreover, parallel search in a GST requires extensive communication or excessive replication in the case of distributed processing. The generalized suffix trees were originally invented to save time and provide information on a set of small strings with minimal computation. However, at the targeted scale, random storage access to retrieve path labels is an unacceptable I/O burden and node size explodes in order to store string identifiers.

4.2 StarIN: A Scalable Index for Strings

We argue that parallel computation should be used with more basic data structures to support scalable and efficient string operations. StarIN is a novel suffix trie index that indexes all suffixes of all strings and retains the frequency of every path label. Because we are targeting large collections of strings; each node stores a single character, avoiding the need to reference strings to retrieve path labels. The path label frequency is used to answer and optimize many string operations without the need to access strings. StarIN is constructed in linear time by traversing the trie from the root using the suffixes. When a suffix exists, node counts are incremented. Otherwise, new nodes are created for the newly added suffix and the counts are initialized to 1.

Although the space requirements of suffix trees and tries are asymptotically the same, it is accepted that a suffix tree is space-efficient because it is a compressed trie. Nevertheless, long common labels are less expected in large collections of strings as the probability of having different combinations from a fixed alphabet increases with string length and collection size. Consequently, the construction complexity added for compacting path labels is unjustifiable given the expected space saving. In cases where most suffix tree labels are single characters, a trie is superior in both space requirement and access time. StarIN eliminates the need for different terminating symbols or maintaining string identifiers and compacting path labels. Some information that would have been readily available in a GST requires extra computation in StarIN. However, such information is efficiently generated when needed in a distributed fashion. Figure 7 shows an example StarIN index.

When exact positions or counts within each string are required, we utilize well-known string algorithms to efficiently extract this information in parallel. StarIN balances between preprocessing time, index size, and execution efficiency. For instance, Boyer-Moore search algorithm is run in parallel to find original strings that satisfy a certain filter query after pruning the search space using the suffix trie and an External R-Way merge sort algorithm is used to eliminate duplicate results after a pattern extraction query is executed.

We assume that collections imported by users consist of long strings (e.g., genomic sequences) while collections that result from user queries consist of relatively short strings (e.g., matches and k-mers). This assumption allows us to be space-efficient in the case of long strings, where strings are stored and indexed separately using suffix trees [18]. We still avoid a GST even for long strings because a huge single tree is challenging to manage and traverse in parallel. Operations over long strings avoid the search algorithms since their performance degrades as strings get longer. At the same time, multiple suffix trees fit the distributed nature of large-scale string analytics, where work on a collection could be partitioned among a number of workers. Because suffix trees are well studied; we next focus on our computation model.

Let us compare the use of the GST in Figure 6 and StarIN in Figure 7. Finding the number of times ”EST” appears in S requires traversing the GST to the node whose path label is ”EST” then counting the leaf nodes below it, i.e., exhaustively traverse the sub-tree to find three leaf nodes indicating the three appearances. During traversal, path labels are recovered by accessing the original strings. In our StarIN index, we traverse the suffix trie following nodes ”E”, ”S”, then ”T” to find that ”EST” appears three times. The total count of appearances is available by construction in our index.

Given the collection S from the previous example, let T be another collection of one string, T = {WEST}. To execute `APPROX(S, T, EDIT(1))`, the suffix trie of S is traversed to match ”WEST” allowing one edit. The output is the set of matches {BEST, TEST, EST}. Now assume T = {TA}, to filter strings in S with a suffix ”TA” allowing one mismatch, we execute `SUFFIX(S, B, HAMMING(1))`. The suffix trie for S is traversed to level 2 keeping the path labels that are within hamming distance 1 from ”TA”. To ensure a path label is a suffix, it must be a leaf or have its count is more than the sum counts of its children. In this case, the output is {ESTATE}.

4.3 Parallel Support for StarQL Operations

StarIN supports StarQL primitives, which include complex operations that require parallelization in order to finish in reasonable time. Tuning problem decomposition depends on the number of available workers and the load of the query. Given our StarIN data structure, a collection is decomposed into sub-collections and assigned to workers. The StarIN footprint of each sub-collection fits in memory. Complex query operators are solved in parallel by utilizing the underlying infrastructure and the decomposed data structure. Next we show by example how we utilize parallel computation to extract information that is not stored in our StarIN data structure.

Assume a user was interested in finding the positions where ”EST” appears in S. Using StarIN, we would know from traversing the trie that ”EST” appears three times. To find the exact positions, a parallel search is executed where the strings in S are distributed among workers to search for the three occurrences. This search is feasible because it is a bounded exact search and the cost is distributed between workers.

To extract the longest common substring in S, we run `LCS(S)` which first extracts the longest substrings that appear at least $|S| = 3$ times then verify that they exist in every string at least once. In the first step, the candidate solutions are ordered according to their length, {EST, ST, E, T}. In order to stop short, if possible, verification starts from the longest candidate. An exact parallel search is used to verify that ”EST” appears in every string, which is the case in this example and the result is {EST}. Finally, to generate 3-mers of S, `KMERS(S, LEN=3)`, the suffix trie branches of length three are simply spelled out {ATE, BES, TES, TAT, EST, STA}.

Example 2. *Consider a user needs to find text that appears frequently in Wikipedia. The user has to work around spelling mistake and simple differences such as noun plurals and verb tenses. First, the Wikipedia archive is imported into the string database using the* `IMPORT` *StarQL construct. The database indexes the dataset using StarIN and may partition or replicate indexes depending on size and available resources. To find all frequent patterns, a query to generate motifs is used. Motifs are patterns that appear frequently but not necessarily exactly. StarQL supports different distance functions for approximate matching. Running on 480 cores, the motifs search space (a combinatorial tree over the English alphabet) is partitioned to thousands of tasks. The workload is balanced by dynamically assigning tasks and the results are gathered and returned to the user.*

The user may decide to filter out motifs of length 4 or less as they correspond to common short words, such as articles and prepositions. The user allows an edit distance of 2 characters so words like ”fishes” and ”fishy” count as occurrences for the motif ”fish”. The length and approximate matching parameters are readily available in StarQL. The user in our example may form and submit the following StarQL query.

```
SELECT RMOTIFS(wiki, FREQ=1000, \
  MINLEN=4, MAXLEN=10, EDIT(3)) \          (11)
  AS wikipats;
```

The Wikipedia archive is represented logically by one collection but indexed using StarIN. The database first executes the repeated motifs operation in parallel. Since the motifs search space is a combinatorial tree, it is logically partitioned into many sub-trees. On a supercomputer, the StarQL optimizer finds that 2,048 cores can be fully uti-

Table 8: Expressing the same simple query using PiQL and StarQL syntax.

Language	Query
PiQL	`SELECT * FROM MATCH(R,p,"EEK",EXACT,3)`
StarQL	`SELECT EXACT(R, "EEK");`

lized given the query workload. The original archive is not accessed because StarIN is annotated with counts. The resulting repeated motifs are also in the form of a suffix trie. Therefore, further operations to extract the common suffixes, for example, are easily executed on `wikipats`.

5. EXPERIMENTAL EVALUATION

This section presents different aspects of evaluating our StarQL language: the expressiveness power, the StarQL query optimizer, the StarIN scalability, and the overall performance of using StarQL. We implemented StarDB [28] using C/C++ and MPI based on StarQL and StarIN. StarDB uses a master/worker architecture. As an MPI-based system, StarDB can be used in a workstation, cluster and supercomputer. StarDB is a large-scale string database that is available for download [6].

5.1 StarQL Expressiveness

StarQL expresses queries in a natural and readable way. Consider the simple query of finding exact matches of `EEK` in a collection of protein sequences `R`. Table 8 shows this query in PiQL and StarQL. While StarQL is more readable, PiQL [31] is also limited to matching biological sequences. For instance, PiQL cannot express a simple query over a text archive like the following StarQL query, which returns the unique words of lengths 5 to 7 from Wikipedia prefixes.

```
SELECT PREFIXES(Wiki, minlen=5, maxlen=7);
```

BLAST is the widely used bioinformatics tool. We can express a BLAST script in StarQL to efficiently execute in StarDB. The BLAST workload was generated using the human and mouse immunoglobulin variable region dataset from NCBI[7]. This dataset is composed of 141,465 DNA sequences of lengths that range between 97 and 3,177,340. We invoked BLAST version 2.0 with the default parameters for the tool *blastn* and the query string `ACCGTTCAGTT`. To our surprise, BLAST returns one match that represents a sufficiently high-scoring ungapped alignment. In fact, BLAST heuristics imply that the same BLAST command can yield slightly different results between different runs.

We imported the dataset in StarDB and issued the following StarQL query. Since StarDB implements exact algorithms, our StarQL query finds all results. Firstly, we find two exact matches of the query string. Moreover, we find 35 approximately matching substrings that appear in the dataset 451 times. From here, using StarQL we can further process these results to analyze the dataset by running other operators without the need to move data between systems and without running other procedural tools.

```
SELECT APPROX(igSeqNt,"ACCGTTCAGTT",USER(1));
```

Furthermore, we compare StarDB capabilities against state-of-the-art procedural repeated motif extractors; namely PSMILE [7], FLAME [12], and VARUN [2]. Although the procedural codes are specialized, StarDB generates the same output up to 3 orders of magnitude faster. For example, for a certain exact-length motif query, FLAME runs for 4 hours while StarDB finishes serially in 1 hour and using 12 cores in 7 minutes. StarDB is able to handle 3 order of magnitude larger strings and scaled efficiently on a supercomputer whereas the only parallel motif extractor [7] reported scaling to 4 cores.

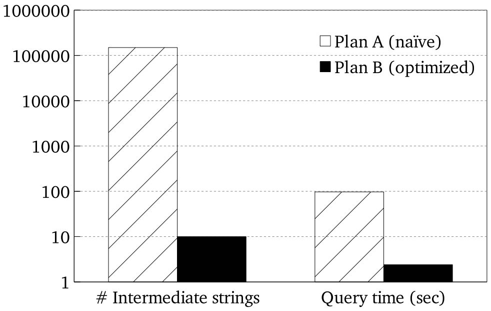


Figure 8: Semantic-based optimizations of StarQL queries dramatically decreases intermediate results and reduce serial execution time by replacing operations while maintaining semantics. Plan A and Plan B are shown in Table 7.

5.2 The StarQL Query Optimizer

In this experiment, we show the benefit of StarQL's semantic-based query optimization. In StarQL, string operators could result in large string collections so query plans with small intermediate results and less complex operations are favoured. Figure 8 shows the size of intermediate results and execution times for the query plans discussed in Table 7 of Section 3.3. The gain in memory footprint and execution time from semantic-based optimization is significant. Note that the optimized query executes more operations but (i) they are lightweight as the aggregate and filter operations answers are readily available in the data structure of StarQL model, and (ii) they result in less data access and intermediate results. Fewer intermediate results consumes less memory and requires less instructions to build and use in further steps.

5.3 Scalability and Parallel Support

For time-consuming string queries, scaling out to finish in reasonable time is essential for online analysis of strings. The parallelization of string operations supported by StarQL's data model and proposed data structures effectively achieves this goal. For a StarQL query that involves generating all motifs, the efficient representation of StarIN reduced the serial execution time from 4 hours to less than an hour and a half on the same hardware. This query is executed by StarDB in less than a minute when scaling out to 256 cores. Table 9 shows StarDB using a supercomputer to execute a more complex query in seconds instead of hours.

Due to the flexibility of StarIN, we are able to find the best problem decomposition and determine the degree of parallelism to highly utilize resources with minimal overhead. Therefore, StarDB automatically tunes the execu-

[6] http://cloud.kaust.edu.sa/Pages/stardb.aspx

[7] ftp://ftp.ncbi.nih.gov/blast/db/FASTA/igSeqNt.gz

Table 9: StarDB's scalability on a Blue Gene/P supercomputer. Query load is increased by increasing the allowed hamming diastance to 4. The serial execution time of the query is 5.2 hours. Speedup efficiency is the ratio of speedup to number of cores with an optimal value of 1.

```
SELECT RMOTIFS(dna, freq=10000, len=12,
                 hamming(4));
```

Cores	Time (sec)	Speedup Efficiency
512	38	0.97
1024	19	0.97
2048	10	0.97
4096	5	0.92
8192	3	0.76

Table 10: StarDB's automatic tuning enhances execution using the same number of cores by determining the best problem decomposition. Moreover, the utilization of resources is enhanced dramatically as indicated by measured speedup efficiency (SE).

```
SELECT RMOTIFS(dna, freq=10000, minlen=12,
                 hamming(3));
```

Cores	w/o Auto Tuning		with Auto Tuning	
	Execution time	SE	Execution time	SE
1	1.6 days	1.00	1.6 days	1.00
16	2.7 hours	1.00	2.7 hours	1.00
1,024	2.5 minutes	0.96	2.4 minutes	0.99
2,048	1.5 minutes	0.79	1.2 minutes	0.98
4,096	53 seconds	0.67	39 seconds	0.91

tion parameters (e.i., problem decomposition and number of cores to use) to achieve the near optimal resource utilization. Because we can generate many small tasks, we utilize our automatic tuning framework [25] to find the best decomposition (i.e., maximum number of tasks with minimal parallel overhead) and estimate the serial and parallel runtimes to predict utilization. Table 10 shows the gain in time and utilization by automatically tuning the execution of the same query on the supercomputer.

6. RELATED WORK

Several string data models and languages are theoritically sound but impractical to implement [15]. Richardsons introduced one of the early declarative query language for strings [24]. In his model, a string starts with a symbol and the every symbol is considered the next instance of the symbol to its left. However, it is known that temporal logic modalities have limited support for recursion or iteration [33], both needed for string queries.

There is no standard mechanism for managing and analyzing strings at the scale needed nowadays. Algorithms that handle long strings or large collections of strings are implemented within disjoint applications. For example, BLAST [1] is used for matching; it finds regions of local similarity between biological sequences. Another example is KAT [8], a k-mer counting tool used to analyze substring frequency spectra. Currently, string analytics require running multiple standalone applications. Users need to move data between applications that use different formats and have different requirements in order to draw conclusions. This gave rise to string analysis pipeline systems (e.g., SeqWare Pipeline [9]) for users to define the steps and order of execution.

Attempts for native string support in databases exist, but most cases take an application-specific approach. SRS is an information indexing and retrieval system [10]. It targets flat files and makes use of the internal structure of their formats. SRS only allows users to draw links between different files using atomic non-sequence fields [11]. For example, SRS users can only query the description fields of FASTA-formatted and EMBL-formatted nucleotide and peptide sequences. SEQ is a string database system based on distinct domains for string elements and their underlying order type [29]. This is beneficial if users need to compute moving averages on time series. However, SEQ model is limits parsing tasks, such as matching.

Traditional databases do not provide adequate support for strings and their operations. Present-day data management methods were designed and implemented to deal with challenges different than those of large-scale string analytics. To handle strings in database systems differently, several attempts were proposed in the last two decades. They were either mere extensions or novel approaches.

Relational databases can be used to store strings in tables using columns of text or binary large object (blob) data types. However, relational databases deal with strings as atomic entities and queries over their internal structure are limited to the `LIKE` construct in the de facto query language, SQL. Simple extensions build on the rich and well-established data management literature and systems by introducing strings as relational domains [8]. Works of this type include extensions to the relational calculus [13, 16, 14, 4]. Periscope/SQ [32] extended PostgreSQL with matching operations over biological sequences. It is challenging to express common string queries, such as motifs and k-mers, with only matching operations. Moreover, complex queries require the efficient utilization of large infrastructures to finish in reasonable times. Therefore, Periscope/SQ reported simple matching queries over sequences of 5,000 symbols.

Most relational databases support a limited number of data types. To handle strings as first-class types, researchers moved to object-oriented databases [15, 3]. Nevertheless, support of string operations in object-oriented databases does not provide an ultimate solution as meta data overhead grows. Generally, extentions of existing databases are limited by their original data models and are undesired as they require the modification of mature systems [30].

7. CONCLUSION

This paper proposed StarQL, a declarative query language for strings. StarQL is designed to support string processing that targets large datasets and large infrastructures (i.e., clusters and supercomputers). The paper presented a native string data model that allows StarQL to provides high degree of expressiveness within different appli-

[8] http://www.tgac.ac.uk/KAT/

[9] https://seqware.github.io/docs/6-pipeline/

cation domains. For instance, StarQL managed easily to express workloads written in BLAST. Moreover, our proof-of-concept system implementing the StarQL language and its data model shows orders of magnitudes better performance comparing to state-of-the-art procedural systems. This is because StarQL is equipped with semantic-based optimization. We already need to deal with large strings that may not fit on a single machine. Similarly, some string queries are computationally demanding and require parallel execution to finish in reasonable times. Therefore, we proposed StarIN, a scalable and efficient data structure that strikes a balance between index size, preprocessing time and machine scalability.

8. REFERENCES

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

[2] A. Apostolico, M. Comin, and L. Parida. VARUN: discovering extensible motifs under saturation constraints. *IEEE/ACM Transactions on Computational Biology Bioinformatics*, 7(4):752–26, 2010.

[3] N. Balkir, E. Sukan, G. Ozsoyoglu, and G. Ozsoyoglu. Visual: a graphical icon-based query language. In *Data Engineering, 1996. Proceedings of the Twelfth International Conference on*, pages 524–533, Feb 1996.

[4] M. Benedikt, L. Libkin, T. Schwentick, and L. Segoufin. String operations in query languages. In *Proc. of PODS*, pages 183–194, 2001.

[5] M. Benedikt, L. Libkin, T. Schwentick, and L. Segoufin. Definable relations and first-order query languages over strings. *J. ACM*, 50(5):694–751, Sept. 2003.

[6] J. A. Blake, C. J. Bult, et al. Beyond the data deluge: data integration and bio-ontologies. *Journal of biomedical informatics*, 39(3):314–320, 2006.

[7] A. M. Carvalho, A. L. Oliveira, A. T. Freitas, and M.-F. Sagot. A parallel algorithm for the extraction of structured motifs. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, pages 147–153, 2004.

[8] C. Date. *An Introduction to Database Systems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 8 edition, 2003.

[9] K. Dube, E. Mansour, and B. Wu. Supporting collaboration and information sharing in computer-based clinical guideline management. In *18th IEEE Symposium on Computer-Based Medical Systems (CBMS 2005), 23-24 June 2005, Dublin, Ireland*, pages 232–237, 2005.

[10] T. Etzold and P. Argos. SRS — an indexing and retrieval tool for flat file data libraries. *Computer applications in the biosciences : CABIOS*, 9(1):49–57, 1993.

[11] T. Etzold and P. Argos. Transforming a set of biological flat file librariesto a fast access network. *Computer applications in the biosciences : CABIOS*, 9(1):59–64, 1993.

[12] A. Floratou, S. Tata, and J. M. Patel. Efficient and Accurate Discovery of Patterns in Sequence Data Sets. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 23(8):1154–1168, Aug. 2011.

[13] S. Ginsburg and X. Wang. Pattern matching by rs-operations: Towards a unified approach to querying sequenced data. In *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '92, pages 293–300, New York, NY, USA, 1992. ACM.

[14] S. Ginsburg and X. S. Wang. Regular sequence operations and their use in database queries. *J. Comput. Syst. Sci.*, 56(1):1–26, Feb. 1998.

[15] G. Grahne, R. Hakli, M. Nykänen, H. Tamm, and E. Ukkonen. Design and implementation of a string database query language. *Information Systems*, 28(4):311–337, 2003.

[16] G. Grahne, M. Nykänen, and E. Ukkonen. Reasoning about strings in databases. In *Proceedings of the Thirteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '94, pages 303–312, New York, NY, USA, 1994. ACM.

[17] D. Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology.* Cambridge University Press, 1997.

[18] E. Mansour, A. Allam, S. Skiadopoulos, and P. Kalnis. Era: efficient serial and parallel suffix tree construction for very long strings. *Proceedings of the VLDB Endowment*, 5(1):49–60, Sept. 2011.

[19] E. Mansour, A. El-Roby, P. Kalnis, A. Ahmadia, and A. Aboulnaga. RACE: A scalable and elastic parallel system for discovering repeats in very long sequences. *PVLDB*, 6(10):865–876, 2013.

[20] A. Mathur, A. Sihag, E. Bagaria, S. Rajawat, et al. A new perspective to data processing: Big data. In *Processdings of INDIACom*, pages 110–114, 2014.

[21] G. Mecca and A. J. Bonner. Query languages for sequence databases: Termination and complexity. *IEEE Trans. on Knowl. and Data Eng.*, 13(3):519–525, May 2001.

[22] T. P. Niedringhaus, D. Milanova, M. B. Kerby, M. P. Snyder, and A. E. Barron. Landscape of next-generation sequencing technologies. *Analytical chemistry*, 83(12):4327–4341, 2011.

[23] A. Rajasekar. String-oriented databases. In *Proceedings of the String Processing and Information Retrieval Symposium & International Workshop on Groupware*, SPIRE '99, 1999.

[24] J. Richardson. Supporting lists in a data model (a timely approach). In *Proceedings of the 18th International Conference on Very Large Data Bases*, VLDB '92, pages 127–138, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[25] M. Sahli, E. Mansour, T. Alturkestani, and P. Kalnis. Automatic tuning of bag-of-tasks application. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, April 2015.

[26] M. Sahli, E. Mansour, and P. Kalnis. Parallel motif extraction from very long sequences. In *Proceedings of the ACM International Conference on Information and Knowledge Management (CIKM)*, 2013.

[27] M. Sahli, E. Mansour, and P. Kalnis. Acme: A scalable parallel system for extracting frequent patterns from a very long sequence. *The VLDB Journal*, 23(6):871–893, Dec. 2014.

[28] M. Sahli, E. Mansour, and P. Kalnis. Stardb: a large-scale dbms for strings. In *Proc. of VLDB*, volume 8, pages 1844–1847, 2015.

[29] P. Seshadri, M. Livny, and R. Ramakrishnan. The design and implementation of a sequence database system. In *Proceedings of the 22th International Conference on Very Large Data Bases*, VLDB '96, pages 99–110, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

[30] M. Stonebraker and U. Cetintemel. "one size fits all": An idea whose time has come and gone. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE '05, pages 2–11, Washington, DC, USA, 2005. IEEE Computer Society.

[31] S. Tata, J. Friedman, and A. Swaroop. Declarative querying for biological sequences. In *Data Engineering, 2006. ICDE '06. Proceedings of the 22nd International Conference on*, pages 87–87, April 2006.

[32] S. Tata, W. Lang, and J. M. Patel. Periscope/SQ: Interactive exploration of biological sequence databases. In *Proc. of VLDB*, pages 1406–1409, 2007.

[33] P. Wolper. Temporal logic can be more expressive. In *Foundations of Computer Science, 1981. SFCS '81. 22nd Annual Symposium on*, pages 340–348, Oct 1981.