Querying and Mining Strings Made Easy

Majed Sahli[1], Essam Mansour[2], and Panos Kalnis[3]

[1] Saudi Aramco, Dhahran, Saudi Arabia,
majed.sahli@aramco.com
[2] Qatar Computing Research Institute, HBKU, Qatar,
emansour@qf.org.qa
[3] KAUST, Thuwal, Saudi Arabia,
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. 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 [16]. 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). More strings are being produced due to technological advances [17], and stored due to low storage costs [5]. For example, the National Center for Biotechnology Information [4] (NCBI) reported that the size of the genomic sequences stored in the GenBank repository has doubled approximately every 18 months. Ambitious projects that require large string analysis include the Cancer Genome Atlas [5] and the Square Kilometre Array Telescope [6].

[4] ftp://ftp.ncbi.nih.gov/genbank/gbrel.txt
[5] http://cancergenome.nih.gov
[6] https://www.skatelescope.org

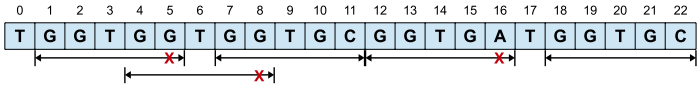


Fig. 1. Example string S over DNA alphabet $\Sigma = \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$. Matches for `GGTGC` are indicated, allowing one mismatch and overlapping matches.

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. Attempts to extend SQL with string operations do not provide native and generic string support because they are limited by their original data models [25, 13].

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[7], a k-mer counting tool used to analyze substring frequency spectra [15]. To analyze strings, users manually move data and run different applications or use pipeline systems to automate this process.

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, examples use DNA sequences for ease of exposition.

Example 1. *Suppose we have a DNA sequence S and we need to know if `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 1.*

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.

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

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

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

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.

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

The StarQL language is based on a simple and native string model. Considering strings as sequences of symbole and sets of related strings as collections allows the support of 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 suitable for parallel query processing and handling large strings.

In summary, our contributions are the following.

- We develop StarQL, a declarative query language for strings and provide a semantic-based optimization for StarQL queries.
- We propose StarIN, a scalable and efficient data structure for implementing StarQL that avoids the limitation of traditional string indexing techniques.
- We conduct comprehensive experiments on real datasets from Wikipedia and the Human genome DNA sequence and run on a supercomputer.

The rest of this paper is organized as follows. Section 2 summarizes the related work. 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 and conclude the paper in Section 6.

2 Related Work

Several string data models and languages are theoretically sound but impractical to implement [13]. Richardsons introduced one of the early declarative query language for strings [19]. 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 [26], both needed for string queries.

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 [18]) 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 [8]. 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 [9]. 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 [22]. This is beneficial if users need to compute moving averages on time series. However, SEQ model limits parsing tasks, such as matching.

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 [7]. Works of this type include extensions to the relational calculus [11, 14, 12, 4]. Periscope/SQ [25] extended PostgreSQL with matching operations over biological sequences and reported simple matching queries over sequences of 5,000 symbols only. It is challenging to express common string queries, such as motifs and k-mers, with only matching operations.

Most relational databases support a limited number of data types. To handle strings as first-class types, researchers moved to object-oriented databases [13, 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 [23].

3 A Declarative Strings Query Language

We consider strings as sequences of symbols from a certain alphabet, 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. A collection of strings has a certain alphabet. For example, the DNA alphabet consists of the four characters {`A, C, G, T`}. Evidently, string queries have one or more strings as input and may produce one or more strings as output.

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 optimizations based on the query semantics and operations.

StarQL adopts a declarative SQL-like syntax, which is easy to understand. Figure 2 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.

```
<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>
```

Fig. 2. Abstract BNF for StarQL.

3.1 Query Constructs

Administration The `IMPORT` and `DELETE` utilities provide simple ways for users to load, index, and purge string collections. Strings can be imported from the file system or from query results. The newly created collection is named and given a unique ID. The number of strings and the total length of all the strings are saved as collection properties. For example, a user imports a dataset of human DNA shotgun reads from disk by running the following query.

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

Matching The `EXACT` matching command finds exact matches of a pattern in a collection. The output 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. To find matches within a certain distance threshold, the `APPROXIMATE` matching command is used. The result is a collection of as many unique substrings that match with their respective counts in the original collection. The capabilities of a regular expression matching command are vital for several string applications. The `REGEX` matching command finds substrings that match a regular expression pattern. 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.

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

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

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

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. 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.

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

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 works similarly for

suffixes. The `SUBSTRING` filtering command finds the strings that share a certain substring; either exactly or approximately. The `LENGTH` filtering command finds strings of a certain length 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}`. The resulting filtered k-mers are {`CCC, CCA, CGT, GTA, CTT, TTA, CAT, ATA`}. This is accomplished using the following query.

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

3.2 User-Defined Functions

StarQL supports user-defined functions 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. A distance function accepts two strings as input and outputs a scalar value indicating the dissimilarity of these strings. Biologists can augment StarQL with weighted matrices to measure distances between DNA sequences.

3.3 String Query Optimizations

It is not always straightforward to optimize string queries because the order of executing string operations could change the final results. 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 multiple 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.

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

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 reduce 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

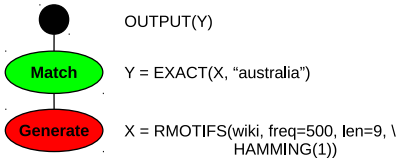
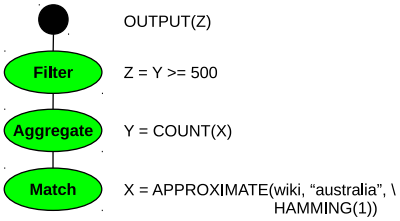
Algorithm 1 StarQL Query Optimizer Algorithm

```
1: procedure OPTIMIZE(Q)
2:     T ← tokenize(Q)
3:     T.initial = Q.first_token
4:     while T.initial NOT collection_id do
5:         T ← tokenize(T.initial)
6:     end while
7:     T.filters ← Q.last_token
8:     if T.filters in MATCH || EXTRACT then
9:         push_down(T.filters)
10:    end if
11:    S ← detect_semantics(T)
12:    P ← S.optimal_plan
13:    return P
14: end procedure
```

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. The optimizer takes advantage of such cases to find an equivalent set of operations with less cost given the data and the user parameters.

Table 1. An example for a schema-based optimization in StarQL.

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

PLAN A (naïve)	PLAN B (optimized)
	

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 1 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.

4 A Scalable Data Structure

This section introduces a scalable and efficient data structure that supports the efficient implementation and parallel execution of StarQL operations.

4.1 StarIN: A Scalable Index for Strings

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.

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 with initial count of 1. StarIN also 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.

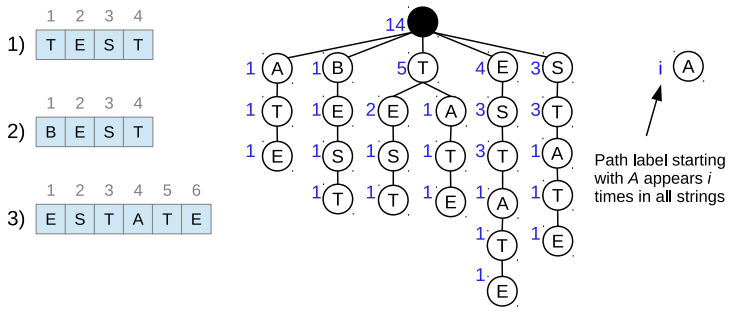


Fig. 3. Example proposed index as opposed to the generalized suffix tree.

Figure 3 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 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.

4.2 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.

(9)
```
SELECT RMOTIFS(wiki, FREQ=1000, MINLEN=4, MAXLEN=10, \
  EDIT(3)) AS wikipats;
```

We indexed the Wikipedia archive using StarIN. Our system executes first the repeated motifs operation in parallel. Since the motifs search space is a combinatorial tree, it is logically partitioned into many sub-trees. In analytics workload, the number of sub-trees affects the utilization of the computing resources [20]. On a supercomputer, the StarQL optimizer estimates the query workload using a sampling technique and determines that 2,048 cores can be fully utilized. 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 quickly executed on the resulting collection, `wikipats`.

5 Experimental Evaluation

This section presents different aspects of evaluating our StarQL language: the expressiveness power, the StarQL query optimizer, the scalability of StarIN, and the overall performance of using StarQL. We implemented StarQL in a strings database system using C/C++ and MPI based on StarQL and StarIN. The system was demonstrated using large datasets and different varieties of queries [21]. The implementation uses a master/worker architecture. As an MPI-based system, it can be used in workstations, clusters, or supercomputers. Our large-scale string database system is available for download [8].

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 2 shows this query in PiQL and StarQL. While StarQL is more readable, PiQL [24] 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 our implementation. The BLAST workload was generated using the human and mouse immunoglobulin variable region

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

Table 2. 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");`

dataset from NCBI[9]. 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. However, its heuristics imply that the same BLAST command could yield different results between different runs.

We imported the dataset in our implementation and issued the following StarQL query. Since we implemented 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.

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

Furthermore, we compare our implementation capabilities against the following state-of-the-art procedural repeated motif extractors: PSMILE [6], FLAME [10], and VARUN [2]. Although the procedural codes are specialized, our implementation 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 our implementation finishes serially in 1 hour and using 12 cores in 7 minutes. our implementation is able to handle 3 order of magnitude larger strings and scaled efficiently on a supercomputer whereas the only parallel motif extractor [6] reported scaling to 4 cores.

5.2 The StarQL Query Optimizer

In this experiment, we show the benefit of our 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 4 shows the size of intermediate results and execution times for the query plans discussed in Table 1 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.

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

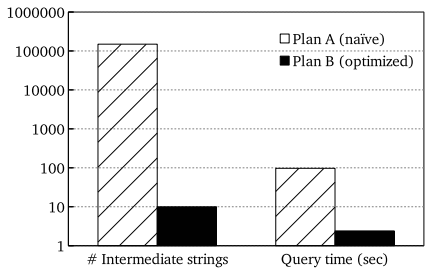


Fig. 4. Semantic-based optimizations of StarQL queries dramatically decreases intermediate results and reduce serial execution time by replacing operations while maintaining semantics. Plan A and Plan B are shown in Table 1.

5.3 Scalability and Parallel Support

Table 3. StarQL implementation 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

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 our implementation in less than a minute when scaling out to 256 cores. Table 3 shows our implementation 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, our implementation automatically tunes the execution parameters (i.e., problem decomposition and number of cores to use)

Table 4. 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

to achieve the near optimal resource utilization. Because we can generate many small tasks, we utilize our automatic tuning framework [hidden reference] 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 4 shows the gain in time and utilization by automatically tuning the execution of the same query on the supercomputer.

6 Conclusion

We 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. This paper proposed StarQL, a declarative query language for strings; and StarIN, a scalable and efficient data structure. We demonstrated the expressiveness of StarQL and the scalability of StarIN by utilizing a supercomputer to process string queries on large real datasets.

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 *Proc. of ICDE*, pages 524–533, 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. 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.
6. 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.

7. C. Date. *An Introduction to Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 8 edition, 2003.
8. T. Etzold and P. Argos. SRS — an indexing and retrieval tool for flat file data libraries. *Computer applications in the biosciences*, 9(1):49–57, 1993.
9. T. Etzold and P. Argos. Transforming a set of biological flat file libraries to a fast access network. *Computer applications in the biosciences*, 9(1):59–64, 1993.
10. A. Floratou, S. Tata, and J. M. Patel. Efficient and Accurate Discovery of Patterns in Sequence Data Sets. *TKDE*, 23(8), 2011.
11. S. Ginsburg and X. Wang. Pattern matching by rs-operations: Towards a unified approach to querying sequenced data. In *Proc. of PODS*, pages 293–300, 1992.
12. 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.
13. 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.
14. G. Grahne, M. Nykänen, and E. Ukkonen. Reasoning about strings in databases. In *Proc. of the Thirteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '94, pages 303–312, 1994.
15. D. Mapleson, G. Garcia Accinelli, G. Kettleborough, J. Wright, and B. J. Clavijo. Kat: a k-mer analysis toolkit to quality control ngs datasets and genome assemblies. *Bioinformatics*, 33(4):574, 2017.
16. 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.
17. 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.
18. B. D. O'Connor, B. Merriman, and S. F. Nelson. Seqware query engine: storing and searching sequence data in the cloud. *BMC Bioinformatics*, 11(12):S2, 2010.
19. 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.
20. 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.
21. M. Sahli, E. Mansour, and P. Kalnis. Stardb: a large-scale dbms for strings. In *Proc. of VLDB*, volume 8, pages 1844–1847, 2015.
22. 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.
23. 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.
24. 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.
25. S. Tata, W. Lang, and J. M. Patel. Periscope/SQ: Interactive exploration of biological sequence databases. In *Proc. of VLDB*, pages 1406–1409, 2007.
26. 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.